UNIVERSITY OF PATRAS
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF ELECTRONICS AND COMPUTERS
APPLIED ELECTRONICS LABORATORY

AI ALGORITHMS' HARDWARE ACCELERATION STUDY FOR WIND TURBINE SURFACE DAMAGE DETECTION ON ARCHITECTURES BASED ON XILINX'S DPU PROCESSING MODULE

DIPLOMA THESIS

GEORGIOS RAFAIL GIOSMAS

SUPERVISOR: MICHAEL BIRBAS

PATRAS – FEBRUARY, 2025

University of Patras, Department of Electrical and Computer Engineering.
Georgios Rafail Giosmas

CERTIFICATION

It is certified that the Diploma Thesis titled

AI ALGORITHMS' HARDWARE ACCELERATION STUDY FOR WIND TURBINE SURFACE DAMAGE DETECTION ON ARCHITECTURES BASED ON XILINX'S DPU PROCESSING MODULE

of the Department of Electrical and Computer Engineering student

GEORGIOS RAFAIL GIOSMAS

Registration Number: 1072796

was presented publicly at the Department of Electrical and Computer Engineering at

27/02/2025

and was examined by the following examining committee:

Michael Birbas, Associate Professor, ECE(supervisor)

George Theodoridis, Associate Professor, ECE(committee member)

Vassilis Paliouras, Professor, ECE(committee member)

The Supervisor	The Director of the Division
Michael Birbas Associate Professor	George Theodoridis Associate Professor

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΗΛΕΚΤΡΟΝΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΜΕΛΕΤΗ ΕΠΙΤΑΧΥΝΣΗΣ ΑΛΓΟΡΙΘΜΩΝ ΤΕΧΝΗΤΗΣ ΝΟΗΜΟΣΥΝΗΣ ΣΕ ΥΛΙΚΟ ΓΙΑ ΑΝΙΧΝΕΥΣΗ ΕΛΑΤΤΩΜΑΤΩΝ ΣΤΙΣ ΕΠΙΦΑΝΕΙΕΣ ΑΝΕΜΟΓΕΝΝΗΤΡΙΩΝ ΠΑΝΩ ΣΕ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ ΒΑΣΙΣΜΕΝΕΣ ΣΤΗΝ ΕΙΔΙΚΗ ΜΟΝΑΔΑ ΕΠΕΞΕΡΓΑΣΙΑΣ DPU ΤΗΣ XILINX

Δ Ι Π Λ Ω Μ Α Τ Ι Κ Η Ε Ρ Γ Α Σ Ι Α

ΓΕΩΡΓΙΟΣ ΡΑΦΑΗΛ ΓΙΟΣΜΑΣ

ΕΠΙΒΛΕΠΩΝ: ΜΙΧΑΗΛ ΜΠΙΡΜΠΑΣ
ΠΑΤΡΑ - ΦΕΒΡΟΥΑΡΙΟΣ, 2025

Πανεπιστήμιο Πατρών, Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών.
Γεώργιος Ραφαήλ Γιοσμάς

ΠΙΣΤΟΠΟΙΗΣΗ

Πιστοποιείται ότι η Διπλωματική Εργασία με τίτλο

ΜΕΛΕΤΗ ΕΠΙΤΑΧΥΝΣΗΣ ΑΛΓΟΡΙΘΜΩΝ ΤΕΧΝΗΤΗΣ ΝΟΗΜΟΣΥΝΗΣ ΣΕ ΥΛΙΚΟ ΓΙΑ ΑΝΙΧΝΕΥΣΗ ΕΛΑΤΤΩΜΑΤΩΝ ΣΤΙΣ ΕΠΙΦΑΝΕΙΕΣ ΑΝΕΜΟΓΕΝΝΗΤΡΙΩΝ ΠΑΝΩ ΣΕ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ ΒΑΣΙΣΜΕΝΕΣ ΣΤΗΝ ΕΙΔΙΚΗ ΜΟΝΑΔΑ ΕΠΕΞΕΡΓΑΣΙΑΣ DPU ΤΗΣ XILINX

του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών

ΓΕΩΡΓΙΟΣ ΡΑΦΑΗΛ ΓΙΟΣΜΑΣ ΤΟΥ **ΙΩΑΝΝΗ**
Αριθμός Μητρώου: 1072796

Παρουσιάστηκε δημόσια στο Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών στις

27/02/2025
και εξετάστηκε από την ακόλουθη εξεταστική επιτροπή:

Μιχαήλ Μπίρμπας, Αναπληρωτής Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών(επιβλέπων)

Γεώργιος Θεοδωρίδης, Αναπληρωτής Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών (μέλος επιτροπής)

Βασίλειος Παλιουράς, Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών (μέλος επιτροπής)

Ο Επιβλέπων	Ο Διευθυντής του Τομέα
Μιχαήλ Μπίρμπας Αναπληρωτής Καθηγητής	Γεώργιος Θεοδωρίδης Αναπληρωτής Καθηγητής

PREFACE

I would like to express my sincere gratitude to the PhD candidate Eleutherios Mylonas for his guidance and assistance in this Diploma Thesis throughout the year. I would also like to thank my Professor and Supervisor Michael Birbas for assigning to me such an interesting and state-of-the-art topic. My appreciation also goes to my family for their support and understanding during my entire undergraduate studies.

ABSTRACT

AI ALGORITHMS' HARDWARE ACCELERATION STUDY FOR WIND TURBINE SURFACE DAMAGE DETECTION ON ARCHITECTURES BASED ON XILINX'S DPU PROCESSING MODULE

STUDENT NAME, SURNAME: **SUPERVISOR NAME, SURNAME:**

GEORGIOS RAFAIL GIOSMAS **MICHAEL BIRBAS**

In this diploma thesis we examine a Computer Vision problem. We try to implement an algorithm for wind turbine surface damage detection and then accelerate it on an FPGA. Wind turbine surface damage detection is considered a high risk and high cost procedure as it is time consuming and exposes humans into difficult situations. Solving the problem with Computer Vision techniques, like Convolutional Neural Networks, has been proven an efficient approach reducing time and achieving high accuracy. Based on the dataset from [2] and the research from [1] we train YOLOv5 nano, a CNN architecture widely known for its inference speed, and acquire an algorithm able to detect defects on wind turbines. Then, we benchmark the algorithm on different CPUs, GPUs and TPUs to analyze its accuracy and get a standard for its inference speed. Subsequently, we utilize a famous AI development environment, Vitis AI, to process the trained model and make it suitable for inference on a FPGA. The FPGA for our purpose is a Xilinx ZCU104, from the Zynq® UltraScale+™ MPSoC family. First the model is quantized with Vitis AI Quantizer and then is compiled with Vitis AI Compiler ending up in a format that can run on a DPU(DPUCZDX8G, B4096), a programmable engine specialized in running inference of CNNs, on FPGAs. The results indicate our algorithm achieves a decent

acceleration as it is 4 times faster in inference, in comparison with the CPU. Also, configuring the FPGA with 2 DPUs speeds up the total execution time for the procedure even more. Lastly, we evaluate the performance of our algorithm on a Raspberry Pi 3. The scripts used in this diploma thesis can be found on github: https://github.com/GeorgiosGiosmas/Diploma_Thesis_2025.

ΕΚΤΕΤΑΜΕΝΗ ΕΛΛΗΝΙΚΗ ΠΕΡΙΛΗΨΗ

ΜΕΛΕΤΗ ΕΠΙΤΑΧΥΝΣΗΣ ΑΛΓΟΡΙΘΜΩΝ ΤΕΧΝΗΤΗΣ ΝΟΗΜΟΣΥΝΗΣ ΣΕ ΥΛΙΚΟ ΓΙΑ ΑΝΙΧΝΕΥΣΗ ΕΛΑΤΤΩΜΑΤΩΝ ΣΤΙΣ ΕΠΙΦΑΝΕΙΕΣ ΑΝΕΜΟΓΕΝΝΗΤΡΙΩΝ ΠΑΝΩ ΣΕ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ ΒΑΣΙΣΜΕΝΕΣ ΣΤΗΝ ΕΙΔΙΚΗ ΜΟΝΑΔΑ ΕΠΕΞΕΡΓΑΣΙΑΣ DPU ΤΗΣ XILINX

ΟΝΟΜΑΤΕΠΩΝΥΜΟ ΦΟΙΤΗΤΗ: **ΟΝΟΜΑΤΕΠΩΝΥΜΟ ΕΠΙΒΛΕΠΟΝΤΟΣ:**

ΓΕΩΡΓΙΟΣ ΡΑΦΑΗΛ ΓΙΟΣΜΑΣ **ΜΙΧΑΗΛ ΜΠΙΡΜΠΑΣ**

Στην παρούσα διπλωματική εργασία εξετάζουμε ένα πρόβλημα Όρασης Υπολογιστή(Computer Vision). Προσπαθούμε να υλοποιήσουμε έναν αλγόριθμο που θα έχει την δυνατότητα να υπολογίζει ελαττώματα στις επιφάνειες ανεμογεννητριών και έπειτα τον επιταχύνουμε πάνω σε ένα FPGA αξιοποιώντας ένα Deep Learning Processing Unit(DPU), μια προγραμματιζόμενη μηχανή της Xilinx.

Ο εντοπισμός ελαττωμάτων στις επιφάνειες ανεμογεννητριών θεωρείται μια υψηλού ρίσκου και υψηλού κόστος διαδικασία, καθώς απαιτεί και καταναλώνει αρκετό χρόνο και εκθέτει ανθρώπους σε επικίνδυνες καταστάσεις. Η επίλυση τέτοιου είδους προβλημάτων με τεχνικές Όρασης Υπολογιστή, όπως Συνελικτικά Νευρωνικά Δίκτυα(CNNs), έχει αποδειχθεί αρκετά αποτελεσματική αφού μειώνει κατά πολύ τον χρόνο της διαδικασίας και προσφέρει αρκετά υψηλή ακρίβεια, η οποία προσεγγίζει κατά πολύ τις ικανότητες ενός εμπειρογνώμονα.

Έχοντας ως βάση την παραπλήσια έρευνα από το [1] εκπαιδεύουμε το YOLOv5 nano, ένα Συνελικτικό Νευρωνικό Δίκτυο ευρέως γνωστό για το υψηλό Inference Speed του, και αποκτούμε έναν αλγόριθμο ικανό να εντοπίζει ελαττώματα στις επιφάνειες ανεμογεννητριών διακρίνοντάς τα σε δύο κλάσεις, την Ζημιά και την Βρωμιά. Ο αλγόριθμος προκύπτει χρησιμοποιώντας μια λίγο διαφορετική εκδοχή του Σετ Δεδομένων Εικόνων από το [2], καθώς μετά από ικανό αριθμό πειραματισμών,

με το Σετ Δεδομένων, κρίθηκε ότι η συγκεκριμένη λύση είναι η καλύτερη δυνατή. Η απόδοση του αλγορίθμου στο κομμάτι της ακρίβειας(accuracy) συνοψίζεται στον παρακάτω πίνακα όπου υπάρχουν αποτελέσματα για την κάθε κλάση ξεχωριστά αλλά και για τις δύο μαζί.

Class	Images	Instances	P	R	mAP50	mAP50-95
all	899	2785	0.834	0.791	**0.83**	**0.563**
dirt	899	173	0.916	0.884	0.922	0.738
damage	899	2612	0.752	0.697	0.739	0.388

Έπειτα, αξιολογούμε την απόδοση του αλγορίθμου σε διάφορα CPUs, TPUs και GPUs για να να θέσουμε ένα σημείο αναφοράς ως προς την ταχύτητά του. Η απόδοση μετριέται από τους χρόνους Pre-process, Inference και NMS που επιτυγχάνει ο αλγόριθμος, ανά εικόνα, σε κάθε διαφορετική συσκευή υλικού(hardware). Τα αποτελέσματα για batch-size = 1 φαίνονται στον παρακάτω πίνακα:

Hardware	Pre-Process(ms)	Inference(ms)	NMS(ms)
TPU v2-8 Google Colab	22.1	**609.9**	12.2
CPU of PC(AMD Ryzen 5 3600X)	0.4	**33**	0.6
Tesla T4 GPU Google Colab (15102MiB)	0.2	**6.8**	1.9
NVIDIA L4 GPU Gogle Colab (22700MiB)	0.2	**6.5**	1.6

Στην συνέχεια, αξιοποιούμε ένα γνωστό περιβάλλον ανάπτυξης AI εφαρμογών, το Vitis AI, για να επεξεργαστούμε το εκπαιδευμένο μοντέλο και να το μετατρέψουμε σε μια μορφή κατάλληλη να τρέξει στο FPGA. Το FPGA που χρησιμοποιούμε είναι ένα ZCU104 της Xilinx, από την οικογένεια Zynq® UltraScale+™ MPSoC. Εφόσον το εκπαιδευμένο μοντέλο είναι σε pytorch μορφή,

πρώτα το κβαντίζουμε με τον κβαντιστή του Vitis AI, vai_q_pytorch, και μετά το μεταγλωττίζουμε με τον μεταγλωττιστή του Vitis AI, καταλήγοντας σε μια μορφή η οποία μπορεί να τρέξει σε ένα DPU(DPUCZDX8G, B4096), μια προγραμματιζόμενη μονάδα επεξεργασίας που ειδικεύεται στο Inference Συνελικτικών Νευρωνικών Δικτύων. Η επιλογή της αρχιτεκτονιής B4096, για το DPUCZDX8G, έναντι των υπολοίπων έχει να κάνει με την επιδίωξη της μέγιστης απόδοσης στο DPU, μιας και η συγκεκριμένη αρχιτεκτονική έχει την δυνατότητα να εκτελεί τις περισσότερες πράξεις ανά κύκλο ρολογιού συγκριτικά με τις υπόλοιπες μιας και απαιτεί τους περισσότερους πόρους από το FPGA. Η απόδοση του αλγορίθμου στο FPGA φαίνεται στους παρακάτω πίνακες:

	Class	Images	Instances	P	R	mAP50	mAP50-95
DPU	all	899	2785	**0.756**	**0.696**	**0.751**	**0.47**
	dirt	899	173	0.801	0.763	0.814	0.589
	damage	899	2612	0.711	0.63	0.688	0.35

	Images per Device	Pre-Process(ms)	Inference(ms)	NMS(ms)
1 DPU	899	20.77	**7.94**	3.5

	Images per Device	Pre-Process(ms)	Inference(ms)	NMS(ms)
2 DPUs	450	21.49	**8.2**	4.41
	449	21.58	**10.57**	5

Τα αποτελέσματα υποδεικνύουν ότι ο αλγόριθμός μας λαμβάνει μια ικανοποιητική επιτάγχυνση αφού επιτυγχάνει Inference Time 4 φορές πιο γρήγορο από αυτό του CPU, παρουσιάζοντας όμως και μια μικρή πτώση στην ακρίβεια. Επίσης, όταν το FPGA διαμορφώνεται με 2 DPUs, ο συνολικός χρόνος εκτέλεσης ολόκληρης της διαδικασίας είναι ακόμα μικρότερος. Για την ακρίβεια, στην περίπτωση των δύο DPUs ο συνολικός χρόνος εκτέλεσης του κώδικα μειώνεται από

τα 91 δευτερόλεπτα στα 65 δευτερόλεπτα. Από την άλλη όμως, στην περίπτωση των δύο DPUs το ένα DPU φαίνεται να παρουσιάζει μια μικρή καθυστέρηση συγκριτικά με το άλλο, εξαιτίας καθυστερήσεων που προκύπτουν από τον Scheduler στην προσπάθειά του να οργανώσει το Inference μεταξύ των δύο DPUs.

Τέλος, μελετούμε την απόδοση του αλγορίθμου μας σε ένα Raspberry Pi 3. Στο Raspberry Pi 3 μελετούμε την απόδοση του αλγορίθμου σε pytorch μορφή αλλά και σε μορφή onnx. Το αλγόριθμο τον λαμβάνουμε σε onnx μορφή αξιοποιώντας ένα από τα scripts που διαθέτει το YOLOv5 repository. Ο λόγος που χρησιμοποιούμε και την συγκεκριμένη μορφή είναι διότι έχει αποδειχθεί ότι δίνει καλύτερες επιδόσεις για Συνελικτικά Νευρωνικά Δίκτυα όταν τρέχουν σε CPUs. Η απόδοση στο Raspberry Pi 3 φαίνεται στον παρακάτω πίνακα:

	Pre-Process(ms)	**Inference(ms)**	**NMS(ms)**
Raspberry pytorch format	6.8	**887.5**	4.8
Raspberry onnx format	9.74	**633.6**	7.46

Τα αποτελέσματα στο Raspberry Pi 3 υποδεικνύουν μεγάλη αύξηση στον Inference χρόνο και καλύτερη απόδοση του αλγορίθμου όταν είναι σε onnx μορφή, όπως και αναμενόταν. Επίσης, ο Pre-process χρόνος είναι μικρότερος συγκριτικά με τον αντίστοιχο στο FPGA λόγω της ταχύτερης μνήμης του Raspberry Pi 3.

Οι κώδικες που χρησιμοποιήθηκαν στην συγκεκριμένη διπλωματική εργασία υπάρχουν στο github: https://github.com/GeorgiosGiosmas/Diploma_Thesis_2025.

Table of Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
CNN	Convolutional Neural Network
YOLO	You Only Look Once
ML	Machine Learning
SGD	Stochastic Gradient Descend
RNN	Recurrent Neural Networks
2D	Two Dimensional
3D	Three Dimensional
YOLOv5 n, s, m, l, x	YOLO version 5 nano, small, medium, large, extra large
DPM	Deformable Part Model
R-CNN	Recurrent Convolutional Neural Network
IoU	Intersection over Union
ReLU	Rectified Linear Unit
mAP	Mean Average Precision
AP	Average Precision
TP	True Positives
FP	False Positives
FN	False Negatives
TN	True Negatives
CSP	Cross Stage Partial
CBS	Convolution, Batch normalization, SiLU
SPPF	Spatial Pyramid Pooling Fast
CSP-PAN	CSP Path Aggregation Network
HSV	Hue Saturation Value
SiLU	Sigmoid-WeightedLinear Unit

COCO	Common Objects in context
ResNet	Residual Neural Network
RPN	Region Proposal Network
CPU	Central Processing Unit
GPU	Graphics Processing Unit
FPGA	Field Programmable Gate Arrays
TPU	Tensor Processing Unit
NMS	Non Maximum Suppression
ONNX	Open Neural Network Exchange
OpenVINO	Open Visual Inference and Neural Network Optimization
CLB	Configurable Logic Block
DPU	Deep Learning Processing Unit
DSP	Digital Signal Processor
LUT	Look up Table
INT8	8-bit integer
FLOAT32	32-bit floating point number
vai_q_tensorflow, pytorch, onnx	Vitis AI Quantizer Tensorflow, Pytorch, ONNX
PTQ	Post Training Quantization
QAT	Quantization Aware Training
XIR	Xilinx Intermidiate Representation
VART	Vitis AI Runtime
API	Application Programming Interface
IoT	Internet of Things

1. Introduction

1.1 AI Today

In 1997, IBM's Deep Blue did something that no machine had done before. It became the first computer system to defeat a reigning world chess champion in a six-game match under standard tournament controls. Big Blue's victory against Garry Kasparov, a human considered among the best in chess world, set the path for a future in which supercomputers and AI could simulate human thinking.

Deep Blue has had an impact on computing in many industries. It gave engineers insights into ways to solve complex problems requiring human capabilities, by designing application specific computers. Customer service, Automated Stock recognition, Speech Recognition, Chats bots, Computer Vision and Robotic Process Automation are just a handful of all these complex problems nowadays computers and AI are required to solve.

Computer vision is a Machine Learning technology, a Deep Learning technology to be more accurate, which enables computers to derive meaningful information from digital images and videos and then take the appropriate action. Like other types of AI, computer vision seeks to perform and automate tasks that replicate human capabilities. In any case, computer vision seeks to replicate both the way humans see (Object Detection), and the way humans make sense of what they see (Classification).

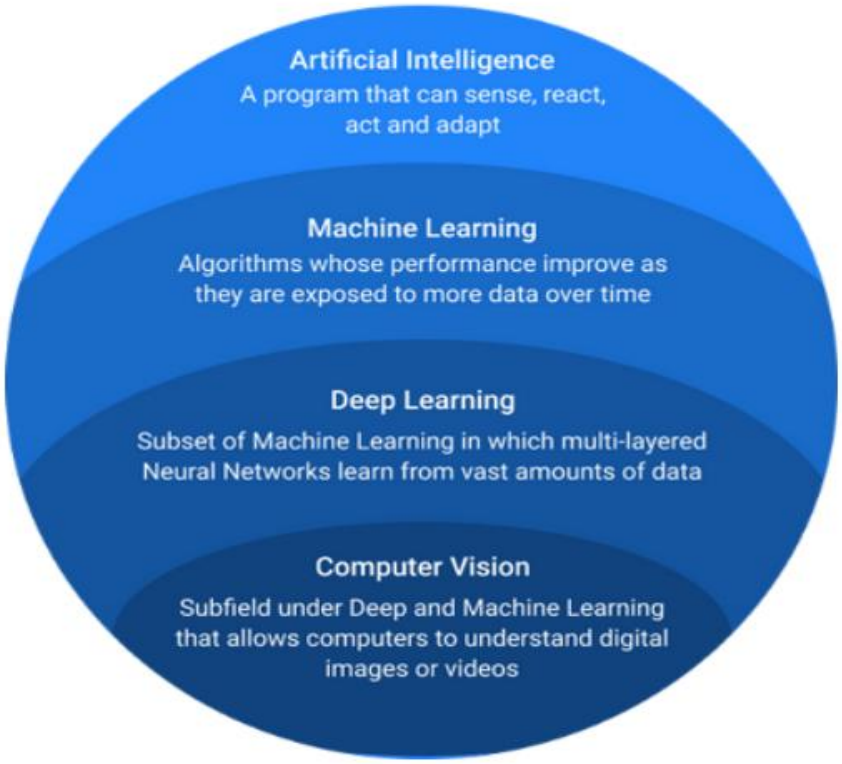

Figure 1: AI fields.

In this diploma thesis, we examine a Computer Vision problem, specifically a problem that has to do with detection of objects within an image. The algorithm we are going to implement is an object detector able to locate defects on surfaces of Wind Turbines, utilizing a famous CNN architecture, YOLO. Our main purpose is to train a newer version of YOLO, version 5, on a wind turbine dataset and study YOLO's performance on different hardware architectures, especially on a FPGA. On the FPGA we will utilize a DPU, a programmable engine, to run and evaluate the inference of our algorithm.

1.2 Machine Learning

Machine learning is a sub-field of Artificial Intelligence. It presupposes the existence of adaptive mechanisms that allow computers to learn in various ways like through acquired experience, by example or based on analogies and similarities. Learning capabilities can improve the performance of an intelligent system over time. Machine learning mechanisms form the basis for the development of these adaptive systems.

The learning subsystem of a Machine Learning algorithm (model) can be divided into three main parts:

- **A Decision Process**: The algorithm is fed with some input data, which can be labeled or unlabeled, and produces an estimate about a pattern in the data, after having processed them. The estimate is usually a prediction or a classification.
- **An Error Function**: An error function evaluates the prediction of the model. If there are existing labels for the given data, an error function can make a comparison between the predictions and the labels, to assess the accuracy of the model.
- **A Model Optimization Process**: This is a procedure where the model's weights are adjusted properly, so the predictions can estimate the labels more accurately. The ML algorithm will evaluate iteratively the model and optimize the weights until a certain threshold of accuracy has been reached.

The ML models fall into three primary categories, based on the data they process and the way these data are being processed:

Supervised Machine Learning Models are trained with labeled data sets. These data sets are made from humans and allow the models to improve their accuracy over time. For example, our algorithm would be trained with pictures of wind turbines consisting of defects labeled by humans, and the machine would learn ways to identify defects on wind turbines on its own.

Unsupervised Machine Learning Models are fed unlabeled data and try to find patterns in these data. Unsupervised machine learning can find patterns or trends that people aren't explicitly looking for. For example, an unsupervised machine learning program could look through online sales data and identify different types of clients making purchases.

Reinforcement Learning Machine Learning Models are trained by being told which are the right choices to be made. If the model makes these right choices is rewarded, otherwise not, providing a very capable learning system. Reinforcement learning can train models to play games or train autonomous vehicles to drive.

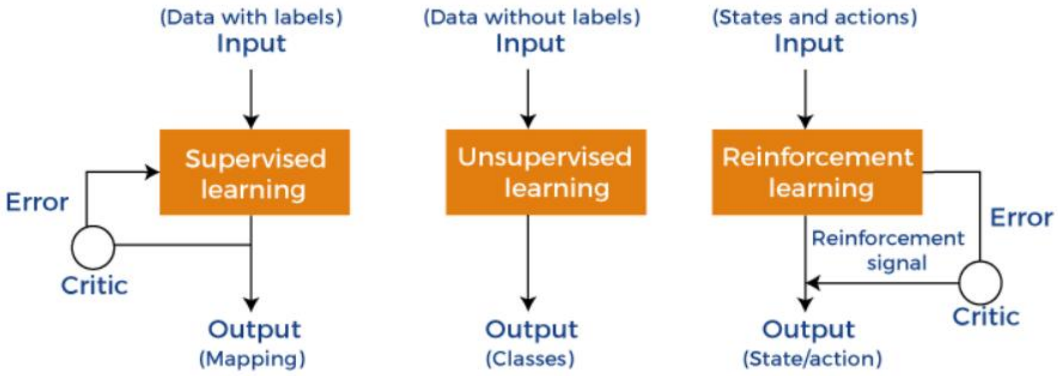

Figure 2: Categories of Machine Learning models based on how they get trained.

Deep Learning is a sector of complex Machine Learning methods. These complex models can gather knowledge from experience and process data without human intervention. They rely on layers of artificial Neural Networks, to train, from programmed instances of features or distinctions. These multilevel layers allow models to detect and train from their own mistakes, providing a much better accuracy for their specified task, in comparison with a simple Machine Learning Model.

1.3 Neural Networks

A Neural Network is a Machine Learning process which makes decisions using interconnected nodes, structured in a form that closely represents the way neurons in the human brain connect and communicate with each other. Neural networks make decisions using an adaptive system whereby they solve problems by learning from their mistakes, thus largely replicating the way a human makes decisions, tries to solve difficult problems, and reaches conclusions.

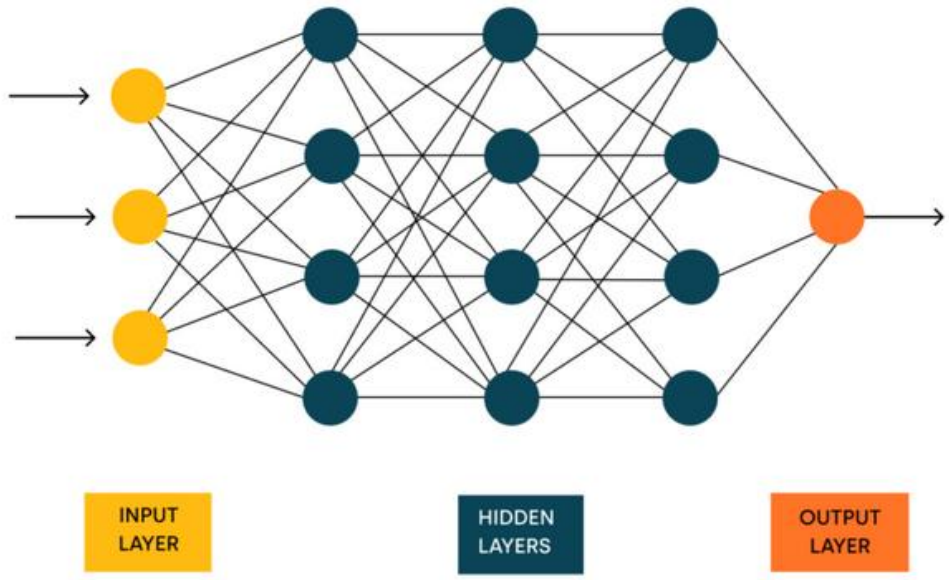

Figure 3: A simple Neural Network.

A basic Neural Network consists of nodes connected to each other divided into three different layers:

The Input Layer ⇒ It is the layer that processes the input data as it enters the Neural Network. It analyzes and categorizes the data before forwarding it to the next layer of the Network.

The Hidden Layer ⇒ It is the layer into which the input data enters after being processed by the Input Layer. In this layer, the basic calculations that will determine the final decision of the Neural Network are usually done. It is by far the largest layer of the three overall, as it is not limited in the number of Neurons (Nodes) it has and can consist of more than one layer.

The Output Layer ⇒ It is the decision layer. It can consist of one or more Neurons, depending on the problem the Neural Network is trying to solve.

A node in any of the above layers has connections to other nodes, which are characterized by some values called Weights, a threshold value (bias) and an Activation Function. The data is passed through the nodes in the form of an independent linear regression model. Specifically, the output of each node is obtained as figure 4 shows:

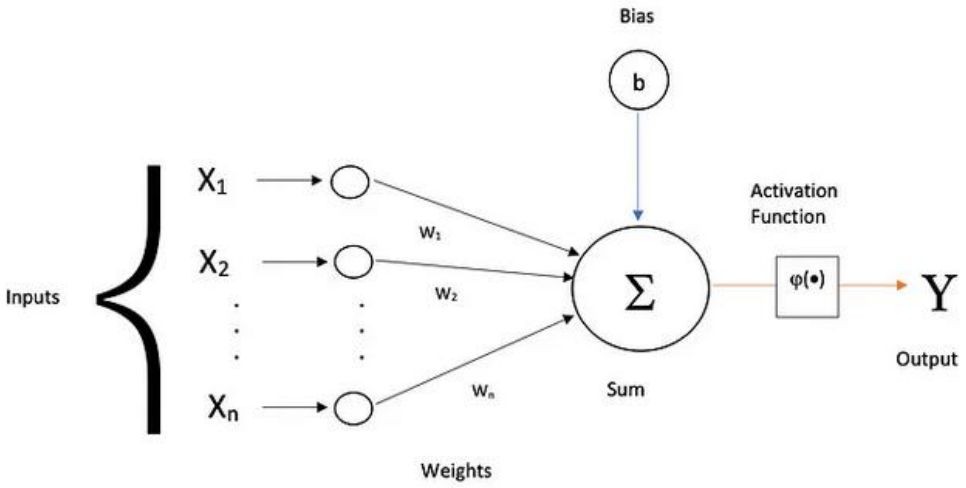

Figure 4: Computation of the output of a single Node.

The values of the outputs of previous nodes connected to this node are multiplied by their respective weight and then all of them are summed together with the node's threshold. This result is input to the activation function and the output of this function corresponds to the final output of the node.

Before training the Neural Network, the initial values of the Weights and Thresholds for each node are usually random. During the training of the Neural

Network these values are changed and adjusted according to a function, the Cost/Error Function, which essentially evaluates the correctness of the results at the output of the network. The cost function during training is sought to be minimized through the use of an optimization method (such as SGD). This method aims to find the total or local minimum of the Cost Function after a sufficient number of iterations, by updating the parameters of the Neural Network after each iteration. At the end of the required iterations, the parameters of the Neural Network (Weights and Thresholds) usually acquire optimal values enabling the Neural Network to solve the given Algorithm and achieve high accuracy.

An **activation** function is a mathematical function applied to the output of a neuron. The purpose of an activation function is to introduce non-linearity into the model, allowing the network to learn and represent complex patterns in the data. Without non-linearity, a neural network would essentially behave like a linear regression model, regardless of the number of layers it has. Below we provide the most common activation functions used by Neural Networks.

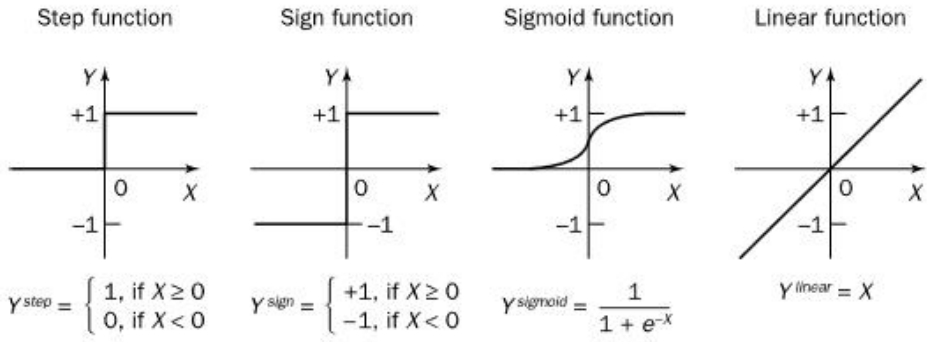

Figure 5: Activation functions of a Neuron.

As mentioned above, a **Loss function is** a function used to measure the performance of the model by calculating the deviation of its predictions from the correct, “ground truth” predictions. Depending on the problem required to be solved, classification or regression, below we provide the most commonly used Loss Functions:

i. Mean Square Loss, used in regression:

$$MSE = \frac{1}{n}\sum_{i=1}^{n} (\text{Yi} - \bar{Y}i)^2$$

ii. Mean Absolute Error, used in regression:

$$MAE = \frac{\sum_{i=1}^{n} |yi - xi|}{n}$$

iii. Binary Cross-Entropy Loss, used in classification:

$$L = -\frac{1}{m}\sum_{i=1}^{m} yi \cdot \boldsymbol{log}\,(\bar{yi})$$

iv. Hinge Loss, used in classification:

$$L = (0{,}1 - y \cdot f(x))$$

Neural Networks can be categorized according to their architecture. Neural Networks can be Fully Connected Neural Networks, Convolutional Neural Networks(CNNs), Recurrent Neural Networks(RNNs), or combinations of the previous architectures.

Fully Connected Neural Networks ⇒ Fully Connected Neural Networks have the basic Neural Network structure as described previously. They consist of the Input Layer, the Hidden Layers and the Output Layer. All other types of Neural Networks usually have a Fully Connected network inside them at some point.

Convolutional Neural Networks ⇒ Convolutional Neural Networks specialise in image analysis and processing applications. These networks use mathematical principles such as convolution and matrix multiplication to identify patterns and extract features from an image. We will analyze the basic structure of a CNN in the next chapter thoroughly.

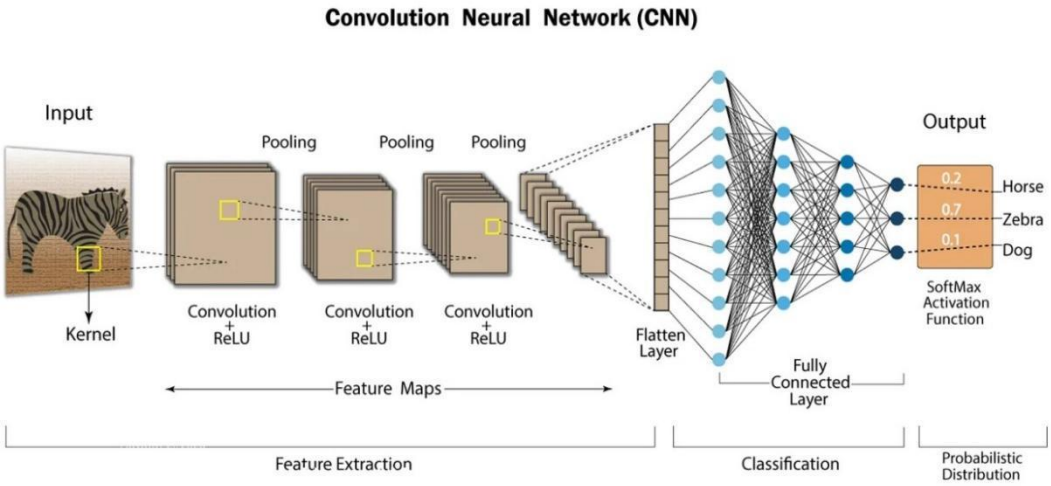

Figure 6: A convolutional Neural Network.

Recurrent Neural Networks ⇒ Recurrent Neural Networks are deep neural networks with feedback loops. In these networks, data is input sequentially and exhibits a time dependence due to these loops. Each value produced in a Recurrent Neural Network, may be dependable to a previously computed value. These networks specialize in applications involving the prediction of future outcomes for example in the stock market.

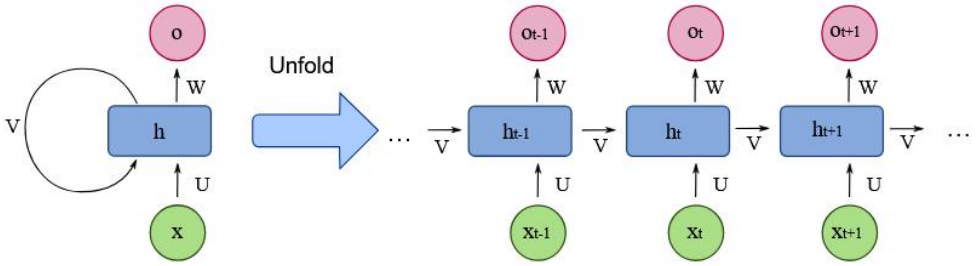

Figure 7: A Recurrent Neural Network.

1.4 Convolutional Neural Networks

As mentioned in 1.3, Convolutional Neural Networks are a special type of Deep Learning algorithm, specialized in processing images. These networks are known by their complexity over other neural network architectures because they consist of different processing stages. These stages include extraction of features from a given image, downsampling of a processed image and prediction or classification for a given image.

In a CNN, at the input processing stage, images are fed in a multidimensional array form. This image form consists of dimensions, color channels(usually Red, Green, Blue) also referred to as depth, and number of batches. Batches are one of the strongest features CNNs provide and offer the capability of processing more than one image in a single forward pass of the network, by stacking them together. With batches the training time of the CNN is reduced considerably, but problems of overfitting appear, when the batch size is too high, leading to an accuracy drop.

$$INPUT = (batch\_size, image\_depth, image\_height, image\_width)$$

After the input stage, the images enter the main network which includes convolutional and pooling layers. The purpose of convolutional layers is to extract features from the images by applying a number of specific filters. These filters are small matrices(usually 2x2 or 3x3), with values configured depending on the desired feature they are meant to extract. The procedure of 2D filtering is described from the figure below:

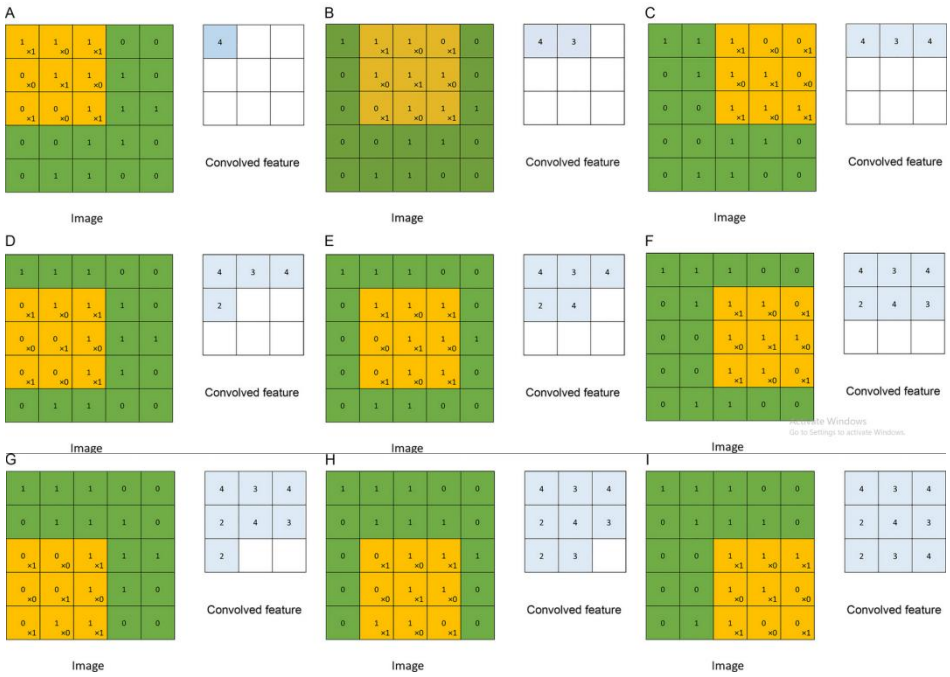

Figure 8: A 2D convolution example.

The filter is applied by scanning the pixels of an image from top left to the bottom right. With a step called stride, sub-matrices of pixels from the original image are created. These sub-matrices have dimensions same as the filter's dimensions and their total number represents the total number of pixels of the output. For example for stride=1 the next sub-matrix arises from moving the current sub-matrix 1 pixel to the right or 1 pixel to the bottom. If stride was equal to 2 we would move 2 pixels to the right or 2 the bottom, and the same goes for all the other possible values. The pixels of the output are produced from the dot product of each sub-matrix with the applied filter. Therefore, the result is a matrix with dimensions certainly smaller than the input image.

In complex Neural Network architectures, each Convolutional Stage consists of more than one filter, each of them with its own special values. In comparison with 2D convolution described above the dimensions of filters are 3D matrices due to the dependability to the colour channels. In any case, the output pixels are often also forwarded to an Activation function in order to introduce Non-Linearity to the Model.

Following a convolution layer the network may have a Pooling layer. The main purpose of a pooling layer is to reduce the size of an image, by maintaining one single value, the maximum or the average, from a group of pixels. This transformation prevents overfitting and maintains only the dominant features from each image. The

concept of stride is also used inside these layers but slight differently. In pooling, strides refers to the dimension of the rectangular formed matrix from which we will maintain only one value. Below, in figure 9 the process of pooling is demonstrated:

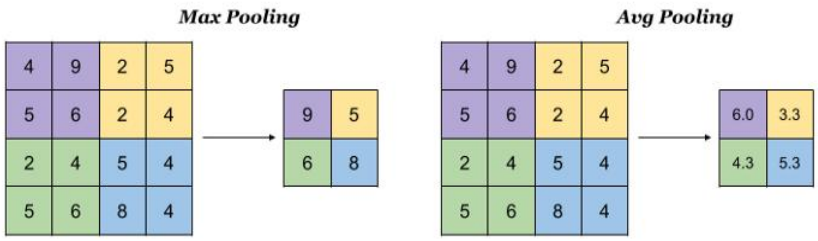

Figure 9: A pooling example.

As we can see from the figure the pixels of the input image are scanned again from the top-left to the bottom-right. Sub-matrices with dimensions (stride, stride) are formed, with the exception that in comparison with convolutional layers, each pixel can belong to only one sub-matrix. Afterwards, the output pixels are computed from maintaining the maximum or the average value from each sub-matrix.

With completion of processing in the main network, the input image has dramatically reduced its size and now consists only of important features. Before entering the decision network the image should be converted from a matrix to a vector. This is a quite simple process, called flattening, and can be achieved by placing the rows one after another creating a 1xN vector, or the columns one after another creating a Nx1 vector. In most cases, the latter approach is used.

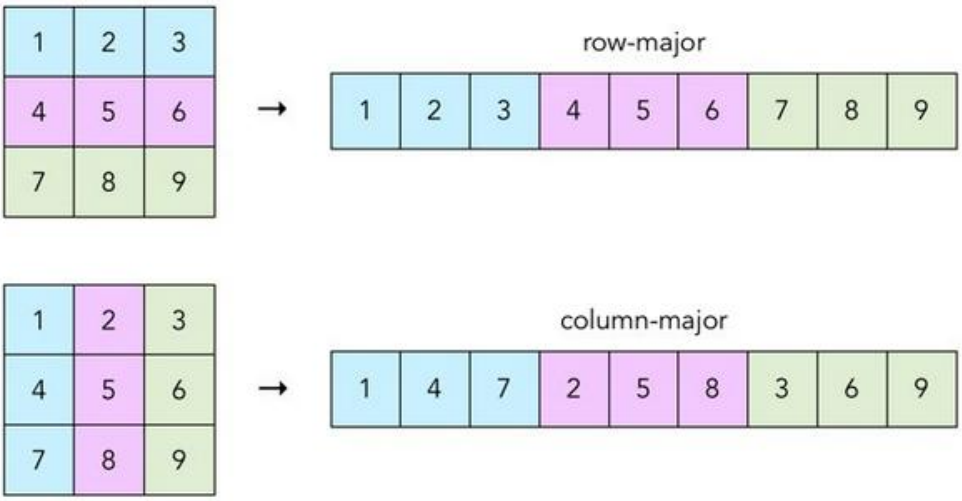

Figure 10: Conversion of matrix to vector.

Finally, the CNN includes a layer meant for making a final decision. This layer is usually a fully connected layer which receives as input the vector produced before. After processing the vector, this dense layer produces as output a probability for each output neuron which indicates the confidence of the class, where the image is predicted to belong to.

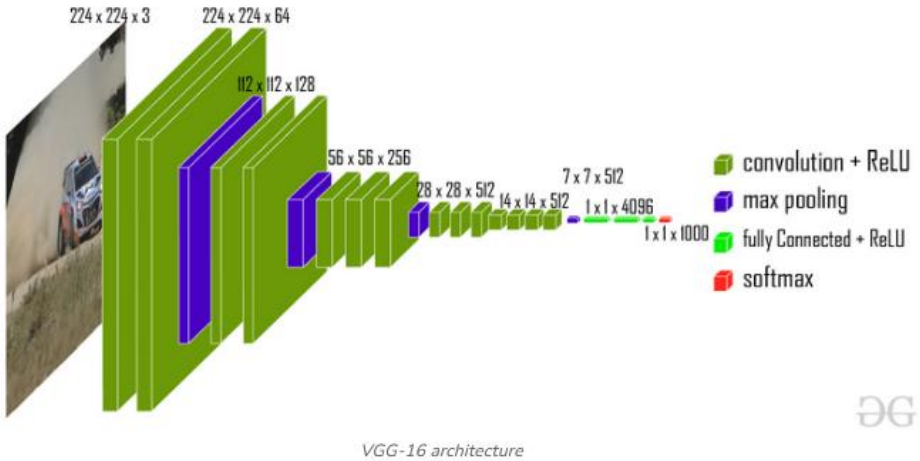

Figure 11: Famous CNN architecture VGG-16, consisting of all the computational steps described in this chapter.

2. YOLOv5 Description

2.1 YOLO Introduction

The algorithm being developed in this thesis is based on YOLO. YOLO is a CNN for object detection and known for achieving state-of-the-art performance along with other famous object detectors. The reason is that YOLO can be considered as a single regression problem so it can be optimized end-to-end. YOLO can be fed with images and predict bounding boxes and probabilities for these bounding boxes in a single evaluation. Other object detection methods, like DPM and R-CNN, use Sliding Windows and Region Proposals methods and then run classifiers, in order to locate objects.

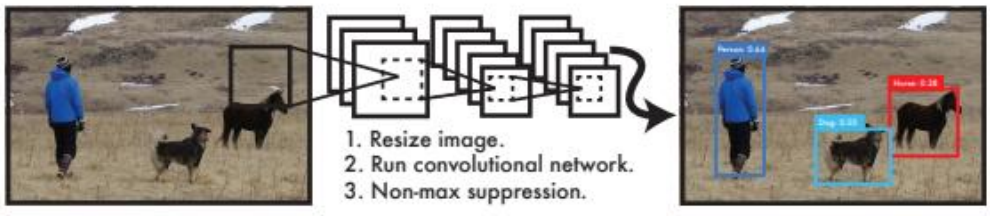

Figure 12: Evaluation procedure in YOLO.

In comparison with these object detection methods, YOLO maintains a mAP two times higher and is extremely fast achieving a processing speed of 45 frames per second. A faster version, suitable for real time detection, is able to process 150 frames per second. When it comes to background errors, YOLO makes less than half mistakes due to its ability to process the entire image simultaneously during training. Furthermore, YOLO learns general representation patterns of objects after training on natural images and performs better when tested on art work. On the other hand, YOLO lacks in accuracy. Despite being able to locate objects on images it makes plenty localization errors when predicting the bounding boxes.

2.2 YOLO Detection Method

As pointed out in 2.1, YOLO can be considered as a single regression problem as it makes predictions in a single evaluation. So, all the object detection components other object detectors use in YOLO are unified in a single neural network.

This processing system locates objects by using grid cells. The input image is fed to the system and is divided into an $S \times S$ grid. If the center of an object is inside a grid cell, then this cell is responsible for locating that object. Each grid cell predicts a fixed amount of bounding boxes and confidence scores for every bounding box. The bounding box predictions consist of 5 numbers:

$$[\, x, y, width, height, confidence \,]$$

The (x, y) coordinate pair represents the center of the bounding box while width and height define its size. These four predictions are relevant with the location of the grid and the size of the image. Confidence is a probability derived from the equation $confidence = Pr(Object) \cdot IOU_{pred}^{truth}$, where the first operand is the probability of an object to be inside the bounding box and the second operand is the intersection over union, or the common area, of the ground truth box with the predicted one. Confidence is of great importance as it defines the accuracy of the model. If the model is confident enough that an object exists inside the predicted bounding box, confidence should be high and close to 1, otherwise confidence should be low and close to 0.

Additionally, each grid cell predicts C conditional class probabilities $Pr(Class_i \mid Object)$, where C is the number of classes. The values for the C extra probabilities are computed as a result of the multiplication between these class

probabilities and confidence. These probabilities are relevant with the grid cell and not with the number of bounding boxes inside the cell.

$$Pr(Class_i \mid Object) \cdot Pr(Object) \cdot IOU_{pred}^{truth} = Pr(Class_i) \cdot IOU_{pred}^{truth}$$

Finally, YOLO designs a probability map containing the highest probabilities of each grid cell to specify the location of objects inside the image and give a better perspective of how it sees the entire image.

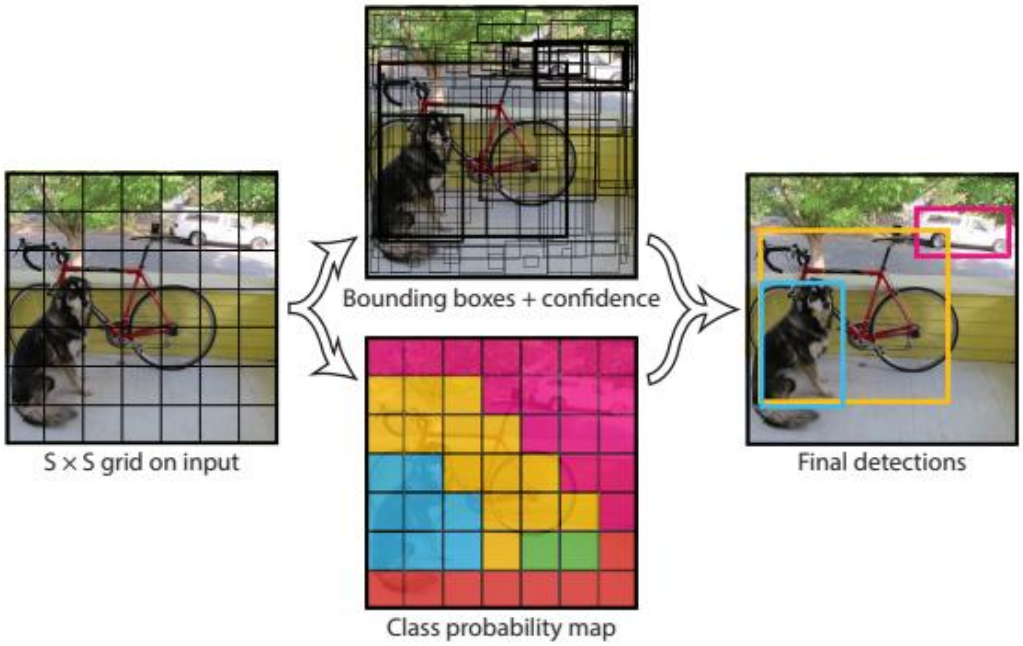

Figure 13: Detection system of YOLO.

To summarize, YOLO divides the input image into an $S \times S$ grid. For each grid cell the system predicts B bounding boxes, consisting of coordinates and confidence scores, and C class probabilities. In total, for each grid cell we have $5 \cdot B + C$ predictions. As a result, the output vector of the system is a vector of size:

$$[S \times S \times (5 \cdot B + C)]$$

2.3 YOLO Architecture

YOLO belongs to the category of Convolutional Neural Networks. Its convolution layers are responsible for extracting features from the input images and its fully connected layers predict probabilities and bounding boxes. There are 24 convolutional layers and 2 fully connected layers, in total. The faster version of YOLO mentioned before has 9 convolutional layers instead and fewer filters in those layers, but keeps all the other parameters the same.

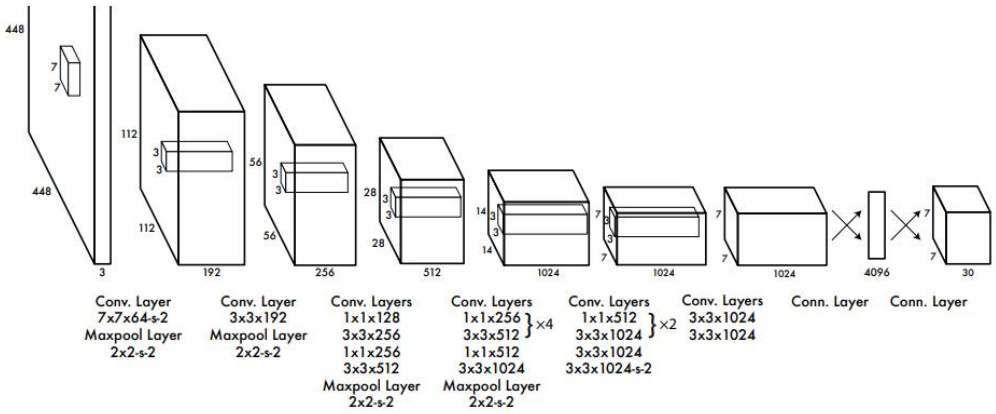

Figure 14: Architecture of YOLO.

Some of the key features of this architecture, presented in figure 14, are:

- This model is evaluated on the PASCAL VOC detection dataset [14].
- The input layer resizes the image to $448 \times 448 \times 3$ resolution.
- The model consists of alternating 1×1 convolutional layers, which reduce the feature space from preceding layers, followed by 3×3 convolutional

layers, meant for extracting the main features. There are also 2×2 Maxpooling layers in specific parts of the network.

- Some of the convolutional layers are pretrained with existing datasets. To be more accurate the 20 first convolutional layers are pretrained on ImageNet dataset [15](with 224×224 input image) and the other 4 convolutional layers, along with the fully connected layers have random initialized weights.

- The predictions of the network are normalized between 0 - 1. This stands for both coordinates and probabilities.

- Non-maximum suppression is used to the output prediction tensor to reduce the amount of overlapping predictions. This is achieved by keeping the best prediction of all.

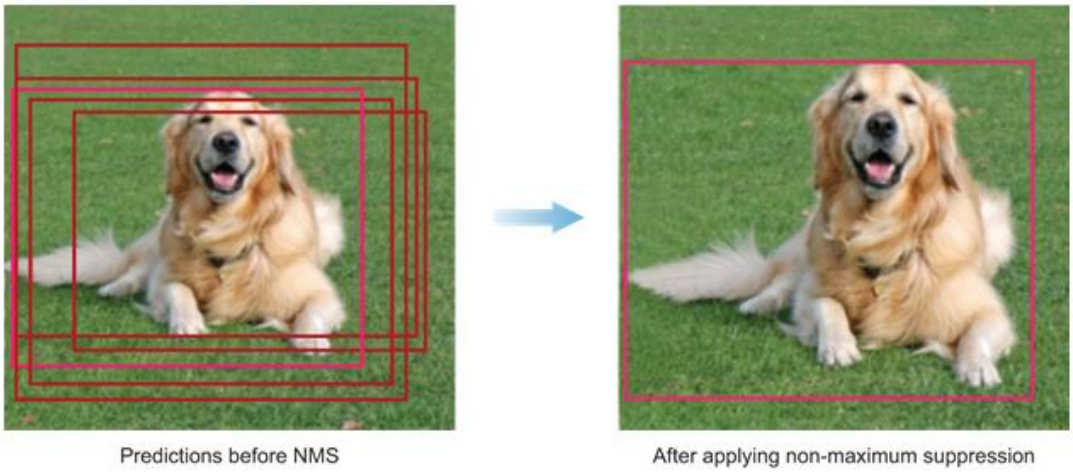

Figure 15: Example of Non-Maximum Suppression.

- The final layer uses the linear activation function while every other layer uses leaky rectified linear activation(Leaky ReLU) described from the equation below:

$$\varphi(x) = \begin{cases} x & if\ x > 0, \\ 0.1x & otherwise \end{cases}$$

- For error function the sum-squared error is used as it is easy to optimize.

Unfortunately, this loss function is not able to maximize mAP. The reason is that it weights localization and classification errors equally, which is not ideal as grid cells that do not contain any objects overpower the gradient with their confidence scores being towards zero, making the model unstable. This issue is resolved by adding two parameters $\lambda_{coord} = 5$ and $\lambda_{noobj} = 0.5$ to increase the influence of the localization term and decrease the influence of the classification term.

Moreover, this loss function weights errors in large and small boxes equally. This means that small deviations in small boxes matter way more than those in larger boxes. This is partially fixed by using the square root of width and height in the loss function, instead of their value as it is.

During training, despite predicting multiple bounding boxes per grid cell, YOLO needs only one predictor to be responsible for the object in the cell. So the model keeps the predictor with the highest IoU with the ground truth bounding box. Essentially, every predictor is optimized in specific classes, aspect ratios and sizes of objects.

The final loss function with the above improvements can be seen in figure 16:

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

Figure 16: Loss function of YOLO.

In figure 16, 1_i^{obj} denotes if the object appears in cell i and 1_{ij}^{obj} denotes that the jth predictor in cell i is responsible for the prediction.

2.4 mAP Metric

2.4.1 Introduction

YOLO uses mAP as a metric for its performance. This metric is the most famous among all Object detectors as it provides a mechanism to evaluate if the predicted classes are the actual classes and how much the predicted bounding boxes are close enough to the ground truth boxes. Mean Average Precision is computed as the mean average of the AP of all classes. The equation can be seen below:

$$mAP = \frac{1}{C} \cdot \sum_{i=1}^{C} AP_i$$

In the above equation, i denotes the class and C the number of classes. So, basically in order to compute mAP, YOLO needs to compute first the AP for each class. AP can be computed using some other metrics, the Confusion Matrix, IoU, Precision and Recall.

2.4.2 Intersection over Union(IoU)

Intersection over Union is a metric responsible for calculating how close is a predicted bounding box with the corresponding ground truth one. IoU measures the intersection between the predicted box and the ground truth box and divides it with their union. If the boxes overlap a lot then IoU is close to 1, otherwise IoU is close to 0. So, IoU is a number in the range 0 - 1. Figure 17 demonstrates how IoU is computed between bounding boxes:

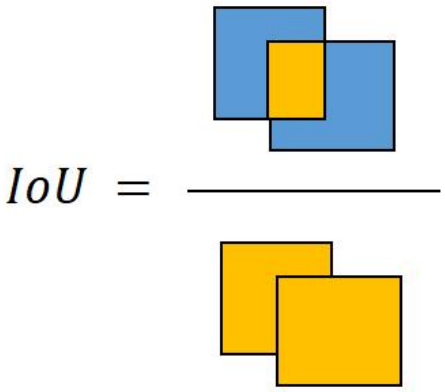

Figure 17: IoU computation.

2.4.3 Confusion Matrix

The Confusion Matrix is a matrix designed for each class that includes four specific attributes, True Positives(TP), False Positives(FP), False Negatives(FN) and True Negatives(TP). For the computation of these attributes it is first necessary, a mechanism to be defined responsible for when a prediction is considered to be correct. A prediction is considered to be correct when the label of the predicted bounding box is the same with the label of the ground truth bounding box and the IoU between the bounding boxes is greater than a specific threshold in the range 0 - 1. Afterwards, the four attributes are computed as follows:

- **True Positives**: The number of times where the model made a prediction and it was correct.

- **False Positives**: The number of times where the model made a prediction but it was wrong.

- **False Negatives**: The number of times where the model did not make a prediction but it was wrong.

- **True Negatives**: The number of times where the model did not make a prediction and it was correct.

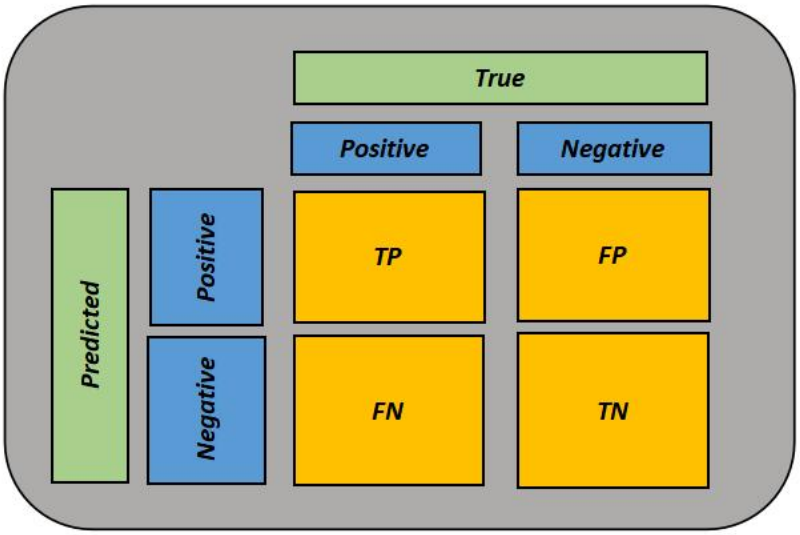

Figure 18: Example of the Confusion Matrix.

2.4.4 Precision and Recall

After finishing computations regarding TP, FP, FN, TN YOLO utilizes the results to compute the next important metrics, Precision and Recall.

- **Precision**: This metric tells us out of total predictions for a class, how precise was the model. For example, if a class is 'human', precision tells us how many detected humans were actual humans out of all 'human' predictions. The equation for precision can be seen below:

$$Precision = \frac{Correct\ Predictions}{Total\ Predictions} = \frac{TP}{TP + FP}$$

- **Recall**: This metric tells us the ability of the model at recalling classes from images. For example, out of the total humans inside images how many were the model able to detect. The equation for recall can be seen below:

$$Recall = \frac{Correct\ Predictions}{Total\ GroundTruth} = \frac{TP}{TP\ +\ FN}$$

2.4.5 Precision and Recall Curve

The last step before computing AP for every class is to draw the Precision-Recall curve. This is possible by computing the values of precision and recall for different threshold values for the IoU. As already mentioned, the threshold is a number in the range of 0 - 1, so starting from 0 we can change its value with a fixed step all the way up to 1. In figure 19, we can see an example of Precision-Recall curves for 3 different classes:

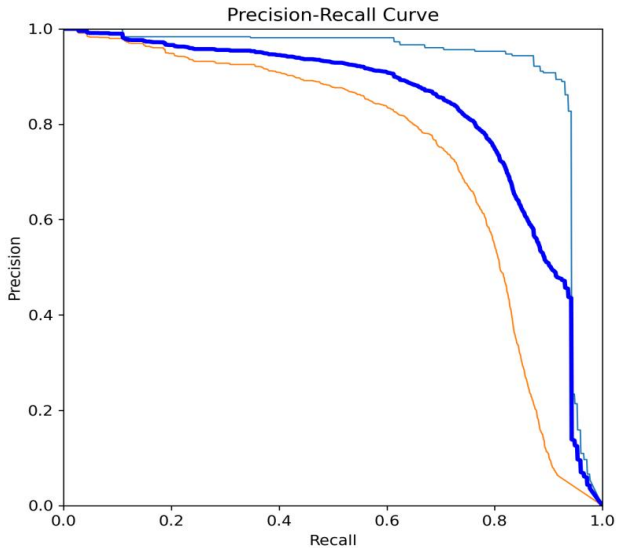

Figure 19: Precision - Recall curves for 3 different classes.

By looking at figure 19 we can realize that when the threshold is low, recall is high and precision is low, because the number of predictions increases reducing the probability of missing a ground truth table. On the other hand, when the threshold is high, precision is high and recall is low, because the model is more confident in what it predicts. Lastly, by computing the integral of the Precision-Recall curve we can finally derive the AP for a single class. The integral can be seen below:

$$Average\ Precision(AP) = \int_{r=0}^{1} p(r) \cdot dr$$

In case we desire to compute mAP for a specific IoU threshold, e.g IoU=0.5, we follow the same procedure but now we filter in only predictions with IoU > 0.5, instead of using all the thresholds in the range 0 - 1.

Although YOLO is extremely fast and scores a high mAP, it still has limitations regarding its performance. The system has spatial constraints issues due to its fixed amount of predicted bounding boxes per grid cell. Also, despite being capable of quickly identifying objects, the model struggles when it comes to locating small objects. So, the model appears sensitive to different aspect ratios and configurations of objects. Finally, YOLO has as its main source of error incorrect localizations. As we discussed above, the error function treats errors in small and large boxes the same way leading to setting up an upper limit for the improvement of detection.

2.5 YOLOv5 Neural Network Architecture

Following the original YOLO algorithm plenty of new versions have been developed aiming at fixing the issues of their predecessor and optimizing its performance. In this thesis, for our algorithm, we will utilize YOLOv5, a powerful object detection algorithm developed by Ultralytics. This is the 5th version of the original YOLO algorithm and has many improvements regarding its performance.

The architecture of YOLOv5 algorithm can be broken down into three main parts, the Backbone, the Neck and the Head:

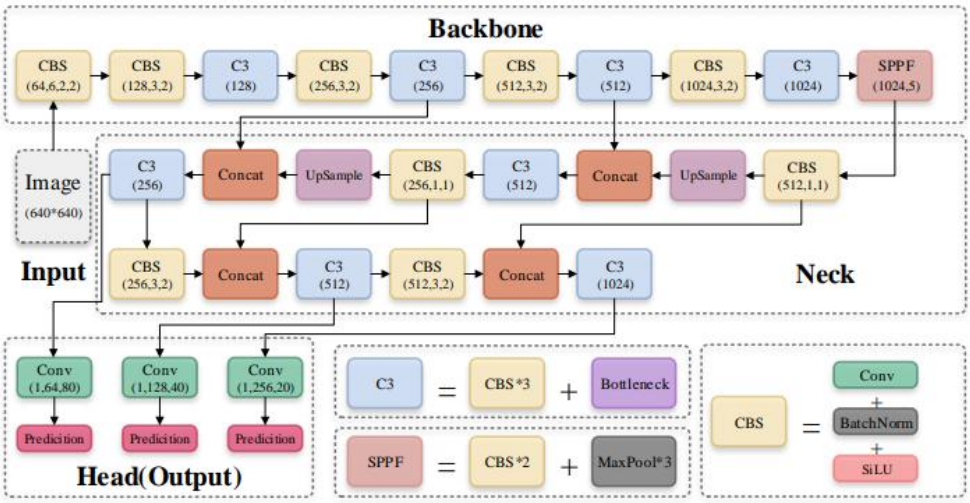

Figure 20: Architecture of YOLOv5.

- Backbone: The Backbone is the structure where the image first enters after being resized into $640 \times 640 \times 3$ resolution. Let as add that images can be loaded in batches. This architecture is designed using the CSP-Darknet53 structure and is responsible for feature extraction. It mainly consists of C3 and CBS components which are responsible for downsizing the image and obtaining feature maps of different sizes, 3 to be more accurate. The structure of C3 and CBS can be seen in figure 20. A more detailed structure of C3 and Bottleneck can be seen in figures 21 and 22 respectively:

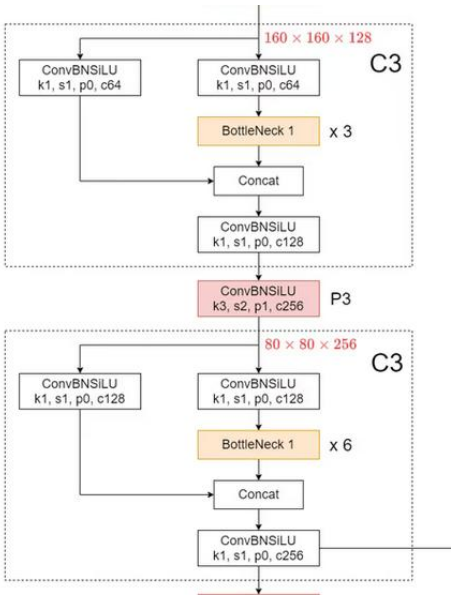

Figure 21: Architectural variations of C3.

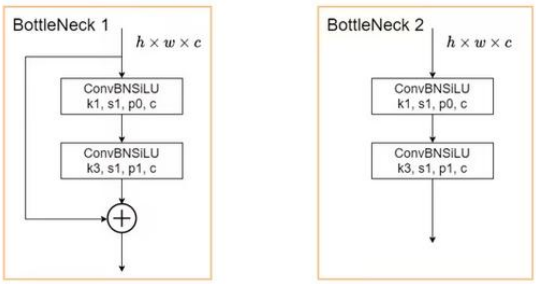

Figure 22: Architectural variations of Bottleneck.

- Neck: This part is responsible for connecting the Backbone with the Detection Head. In this part, SPPF and New CSP-PAN structures are utilized offering an extensive feature fusion before entering the Head. The Neck accomplishes the generation of 3 new feature maps P3(size 80×80), P4(size 40×40) and P5(size 20×20) after processing the 3 feature maps coming from the Backbone. These new feature maps are capable of locating objects in different scales small(by P3), medium(by P4) and large(by P5) resolving the inability of the original YOLO architecture to identify small objects. We need to clarify that the terms feature maps and grid cells are not identical. In YOLOv5, the size of a feature map implies a grid of the same size. In other terms, a feature map is the information extracted from the input image, whereas a grid cell is the spatial position of that information. The architecture of SPPF can be seen in figure 23 and the CSP-PAN in figure 24:

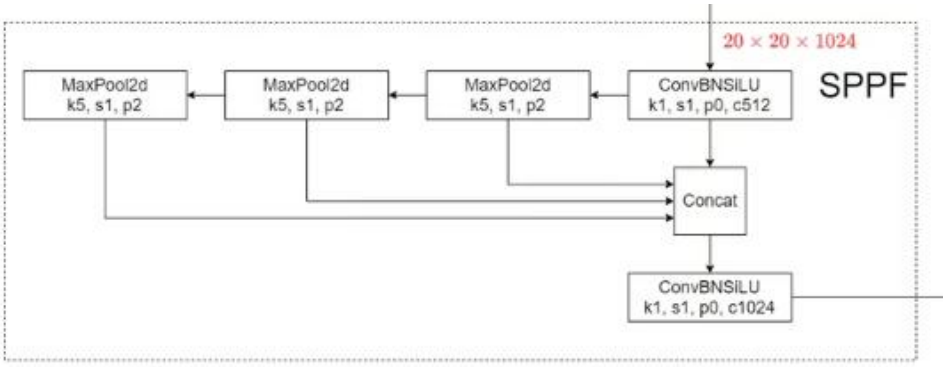

Figure 23: The SPPF structure.

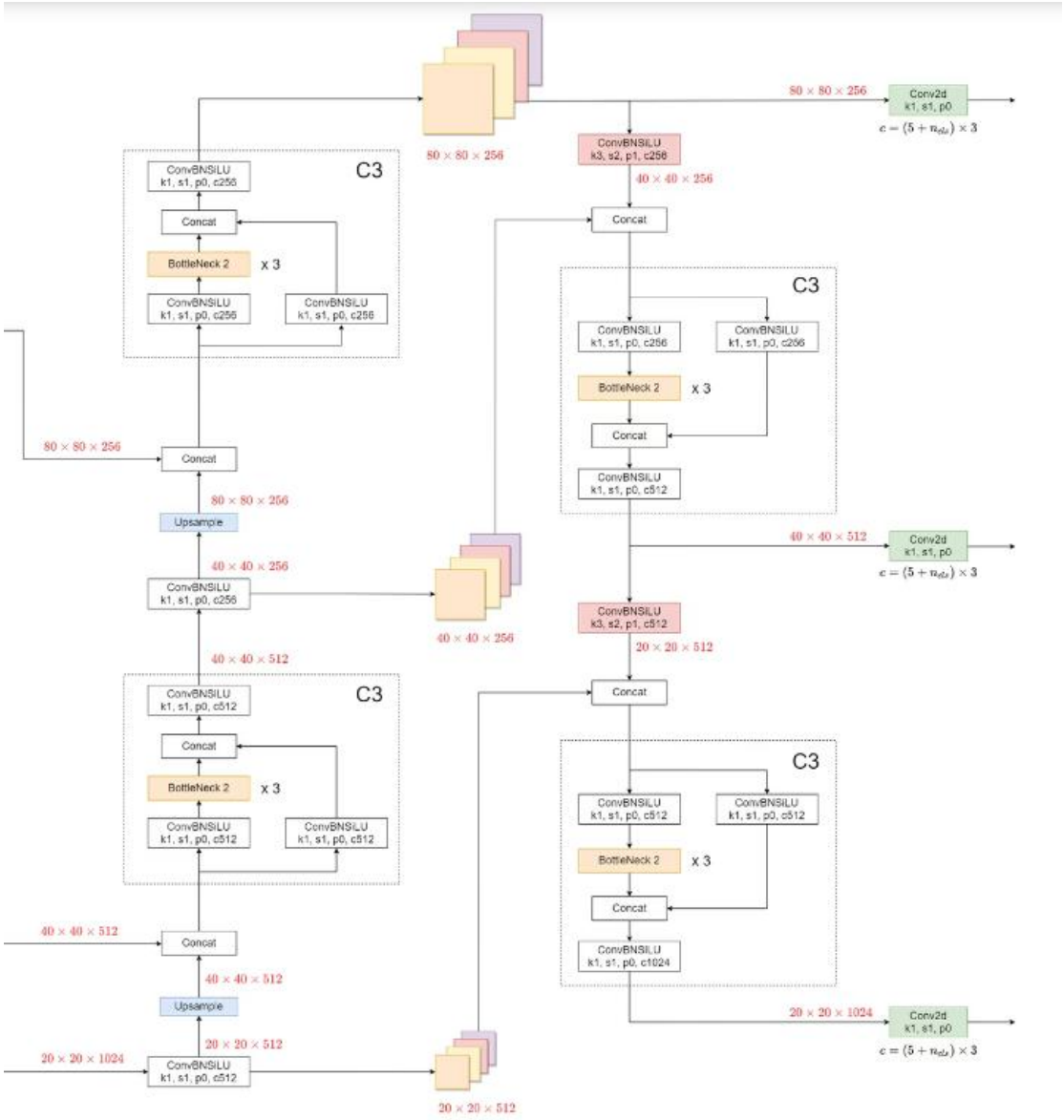

Figure 24: The CSP-PAN structure.

- Head: This is the final part of the architecture. The head makes predictions for every pixel in the feature maps having as reference point 3 anchors boxes and not the grid cell like the original YOLO architecture did. The anchors boxes are predefined(9 in total, 3 for every feature map) and help predictions to align more closely to the aspect ratios and scales of objects. So, basically the Head consists of 3 Detection Heads(detection layers), one for every feature map. The total number of predictions for each Detection Head is equal to $3 \cdot (5 + number\_of\_classes)$, where 3 denotes the number of anchors per detection layer, and 5 denotes the $[x, y]$ coordinates, width, height and object score. Like the original YOLO model we still get class probabilities for every class, but for every predicted bounding box instead of each grid cell like we did in YOLO. The output vectors of Detection Heads can be seen below:

$$[batch\_size, 3 \cdot (5 + number\_of\_classes), 80, 80]$$

$$[batch\_size, 3 \cdot (5 + number\_of\_classes), 40, 40]$$

$$[batch\_size, 3 \cdot (5 + number\_of\_classes), 20, 20]$$

For the computation of the final output, the output vectors of the detection heads undergo a permutation and are combined into one vector together.

$$[batch\_size, 25200, 7]$$

(where 25200 comes from 80x80x3 + 40x40x3 + 20x20x3)

2.6 YOLOv5 Other Improvements

The improved YOLOv5 architecture does not stand alone but is accompanied with other features leading to the achievement of a state-of-the-art performance. One of these key components is the built- in Data Augmentation system, which helps in generalization and reduction of overfitting. The Data Augmentation system allows augmentation techniques to be applied to the image, or the batch of images, in every forward pass during training. With this, in every epoch, the model processes different variations of the input data leading to enhancement of accuracy.

Some of the key Data Augmentations methods this build-in system uses are:

- Mosaic Augmentation: This is an image processing technique that combines 4 images into 1.

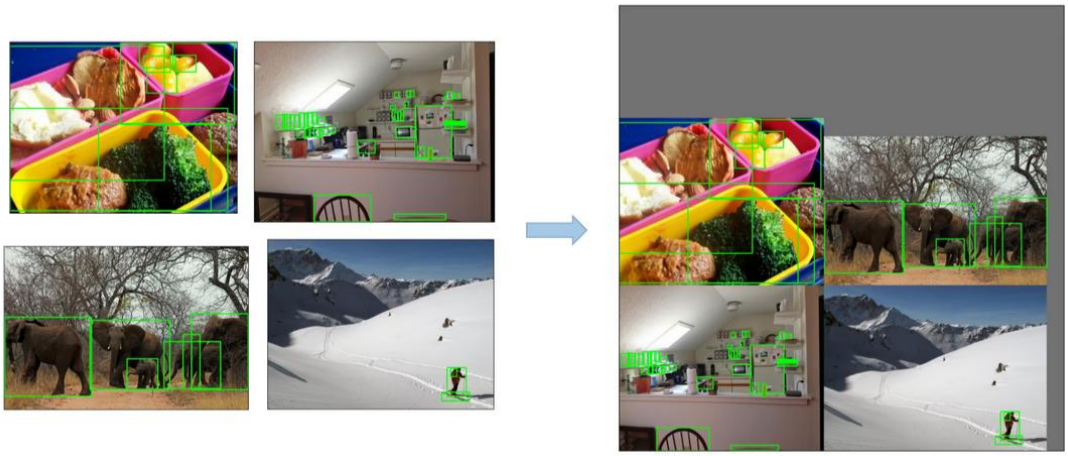

Figure 25: Demonstrates the Mosaic Augmentation Technique.

- Copy-Paste Augmentation: This method copies random patches from an image and pastes them into another image.

Figure 26: Demonstrates the Copy-Paste Augmentation Technique.

- MixUp Augmentation: This method creates composite images by taking a linear combination of two images and their associated labels.

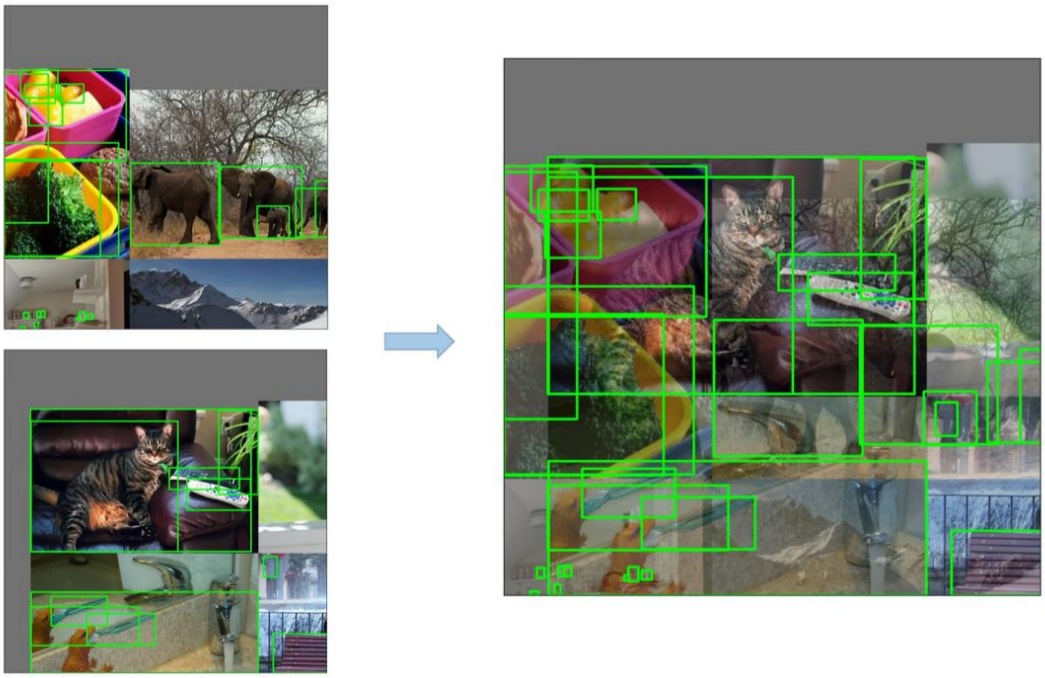

Figure 27: Demonstrates the MixUp Augmentation Technique.

- HSV Augmentation: This method performs random changes to the Hue, Saturation and Value of the images.

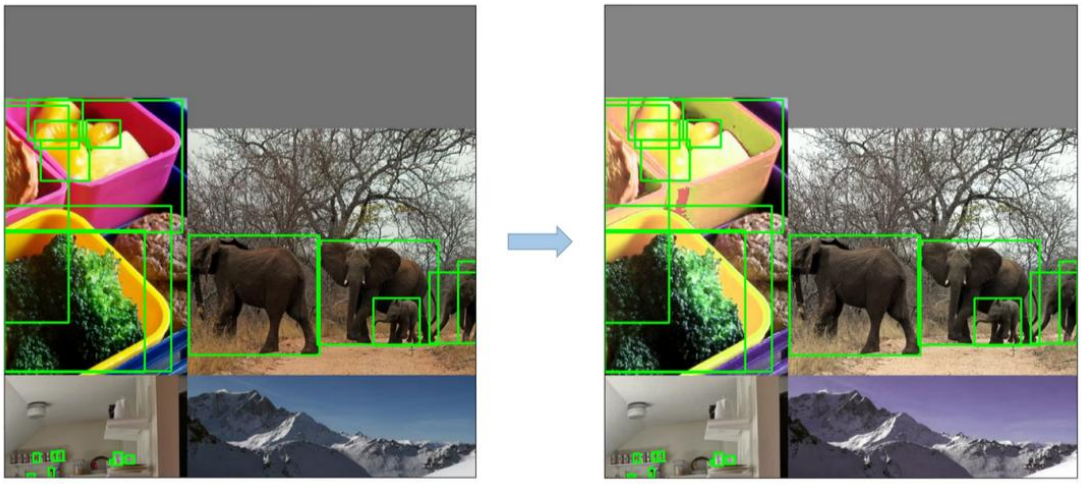

Figure 28: Demonstrates the HSV Augmentation Technique.

The loss function of the improved YOLOv5 architecture is a combination of 3 components, Classes loss, Objectness loss and Location loss. The Classes loss is a Binary Cross-Entropy loss responsible for measuring the error for class predictions. The Objectness loss, is a Binary Cross-Entropy loss responsible for determining whether the object is present in a grid cell or not. Lastly, the Location loss is an IoU loss which measures the error regarding the existence of an object within the grid cell. The equation of the complete Loss function can be seen below:

$$Loss = \lambda_1 L_{cls} + \lambda_2 L_{obj} + \lambda_3 L_{loc}$$

Additionally, the Objectness loss is computed as the sum of the objectness losses of the 3 Detection Heads. But each of these losses weights differently in the overall Loss, to ensure that predictions at different scales contribute appropriately to the total loss. The formula of the entire Objectness loss can be seen below:

$$L_{obj} = 4.0 \cdot L_{obj}^{small} + 1.0 \cdot L_{obj}^{medium} + 0.4 \cdot L_{obj}^{large}$$

As an activation function YOLOv5 replaces the Leaky rectified linear activation function, with SiLU. Figure 29 demonstrates the differences between them.

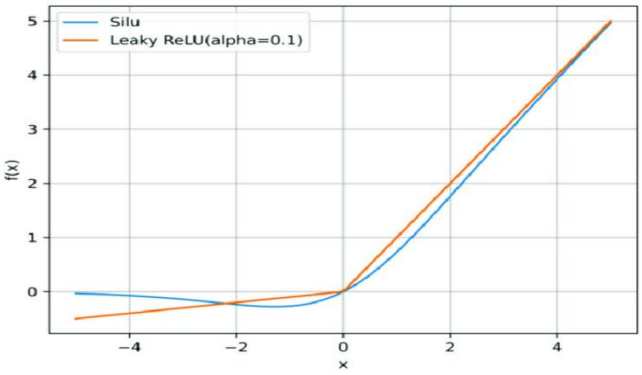

Figure 29: SiLU and LeakyReLU activation functions.

In contrast with the original architecture, YOLO provides multiple variants of the architecture(YOLOv5n, YOLOv5s, YOLOv5m, YOLOv5l, YOLOv5x) offering a trade-off between inference speed and computational power. These variants mainly differ from their entire number of parameters, so the engineer who wants to use one of them can choose based on his available resources. YOLOv5n(nano) is the variant we will take advantage of to develop our algorithm as it is the smallest and the quickest one among the rest. Figure 30 presents some metrics for all these YOLOv5 variants trained on COCO dataset [20].

Model Name	Params (Million)	Accuracy (mAP 0.5)	CPU Time (ms)	GPU Time (ms)
YOLOv5n	1.9	45.7	45	6.3
YOLOv5s	7.2	56.8	98	6.4
YOLOv5m	21.2	64.1	224	8.2
YOLOv5l	46.5	67.3	430	10.1
YOLOv5x	86.7	68.9	766	12.1

Figure 30: YOLOv5 variants and their performance metrics.

3. Wind Turbine Surface Damage Detection

3.1 Introduction

This thesis aims to develop the defect detection algorithm based on the work from the scientific paper “Drone Footage Wind Turbine Damage Detection”[1]. Thus, this section, is deemed necessary to first analyze the work from [1] before beginning our own implementation.

Offshore wind turbine surface inspection is considered an expensive and high-risk operation as it requires a huge amount of time for maintenance and human exposure to dangerous situations. An approach using Computer Vision techniques such as Object Detection with CNNs is proved more efficient, as it speeds up the procedure, erases dangers and offers a precision close up to a human with expertise.

Figure 31: Wind turbine.

In [2] a new publicly available wind turbine dataset is presented, created from [21], among with a comparison between ResNet-101 Faster R-CNN and YOLOv5 for wind turbine surface damage detection. Results show that YOLOv5 outperforms ResNet-101 Faster R-CNN in predicting bounding boxes. On the other hand, ResNet-101 Faster R-CNN estimates an entire instance of damage as a single prediction and achieves greater precision in designing the bounding boxes.

3.2 Method

To create the dataset the high resolution unlabeled images(5280x2970) from [21] have been split into smaller images(586x371) over 13000 [2] in total. This resolution is chosen to fit the input of both ResNet-101 Faster R-CNN and YOLOv5. To categorize defects two classes have been considered, dirt and damage. Instances of dirt represent a dark shading seen on the surface of the wind turbine blades, whereas instances of damage represent markings on the tower, the nacelle and the blade, dark burn marks on the blades and blistering and cracking of the coating on the tower and nacelle.

The labeling has been done by hand utilizing the free online tool makesense.ai[22]. This tool offers the opportunity to export the created labels directly to the format of the proposed CNNs and many other formats. The labeling process resulted to 9351 instances, with 8770 instances of damage and 581 instances of dirt. The labeled images were a little less than 3000. The unbalance between the number of instances between classes occurs from the fact the dirt tends to cover large continuous areas, whereas damage leaves more discrete markings where each of them needs to be represented with a separate bounding box.

Figure 32: Figure 32: Instances of Damage.

Figure 33: Instances of Dirt.

For the benchmarking of the proposed dataset four CNNs architectures have been chosen:

- ResNet-101 Faster-RCNN
- YOLOv5 Small(S)
- YOLOv5 Medium(M)
- YOLOv5 Large(L)

ResNet-101 Faster-RCNN has been initialized with weights pretrained on the ImageNet dataset [15]. The YOLO architectures have been initialized with weights pretrained on the COCO dataset [20].

3.3 Experimental Results

To balance the issue of the unequal distribution of classes inside the dataset, data augmentation techniques have been used. These techniques include HSV augmentation, scaling, translation, flipping and the famous mosaic technique. The mosaic technique combines four images into a 2x2 mosaic. The HSV augmentation applies a random gain to the HSV values. The values for the hue, the saturation and the gain are 0.015, 0.7, 0.4 respectively. As for the other techniques there is a 50% chance for horizontal flipping, a random scaling between 0.5x-1.5x of the original image and a random translation in the x and y of 10% of the total image resolution.

Before training, the dataset has being split 70%-30% into training set and validation set. The YOLOv5 models are configured with a batch-size of 16 and are trained for 1000 epochs. The ResNet-101 Faster-RCNN is configured with a batch-size of 64 and was trained for 100000 epochs.

The Optimization Method was chosen to be Stochastic Gradient Descend(SGD) with the following hyperparameters:

	Learing Rate	**Weight Decay**	**Momentum**
YOLOv5(S, M, L)	0.01	0.0005	0.937
Faster R-CNN	0.0025	0.0001	0.9

Table 1: Hyperparameters for training.

The ResNet-101 Faster-RCNN uses the weighted sum of four losses. The Binary Cross Entropy Loss for the RPN, the L1 Localization loss for the RPN, the Softmax Cross Entropy Classification Loss for the final output, and the L1 Localization Loss for the final output. The YOLOv5 architectures use the combination of three losses as they were described in 2.6. The Means Square Error of the predicted bounding boxes, the Binary Cross Entropy of the predicted object confidences and the Binary Cross Entropy of the predicted classes.

All four architectures use mAP as a metric. To benchmark their performance mAP-50 (threshold>=0.5) and mAP50-95(threshhold 0.5-0.95 at 0.5 intervals) are computed.

Figures 34, 35, 36 show mAP50, mAP50-95 and loss respectively for the validation set during the entire training.

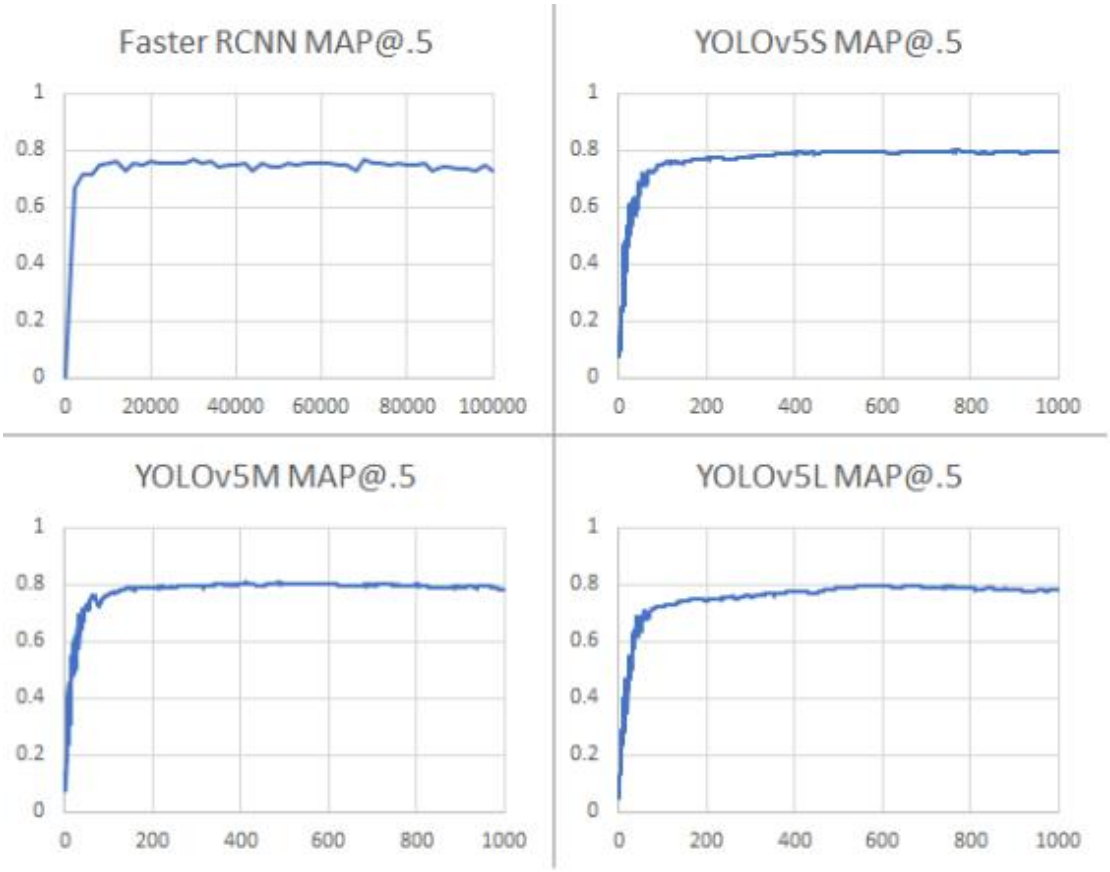

Figure 34: mAP values at 0.5 IoU.

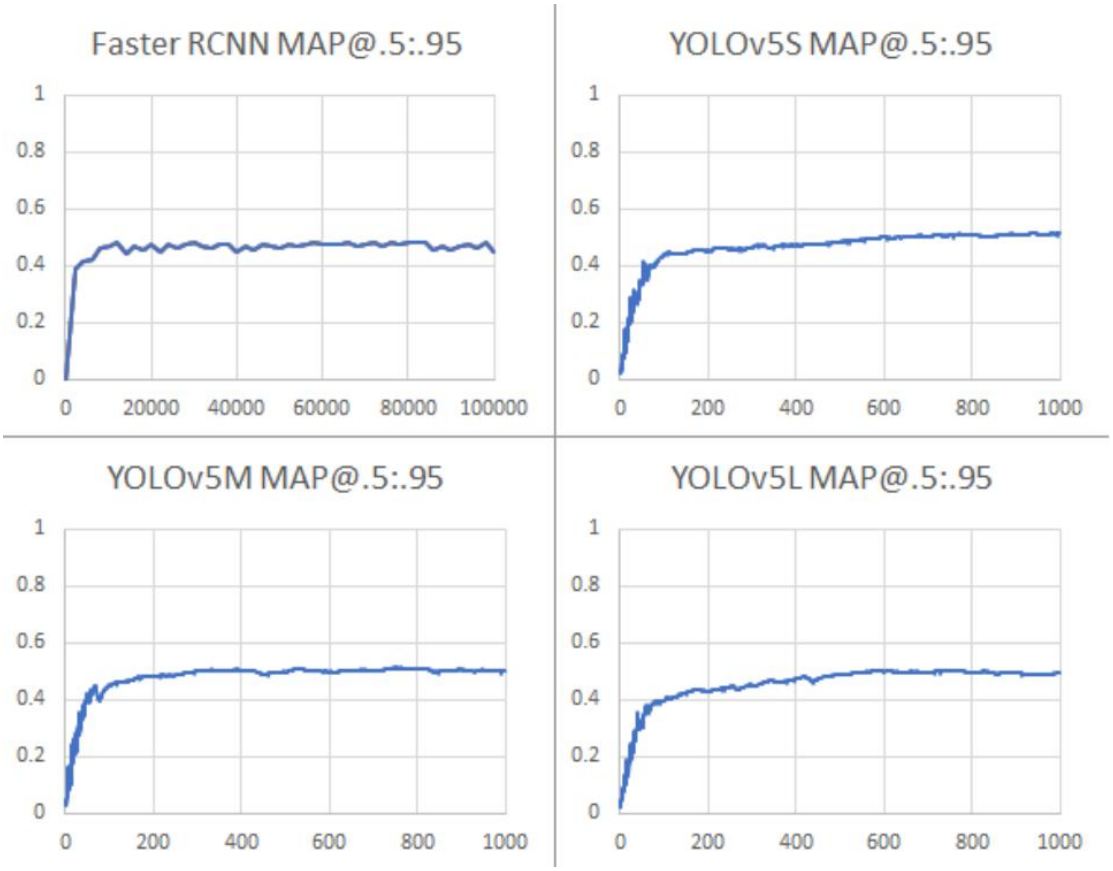

Figure 35: mAP values at 0.5 through 0.95 IoU.

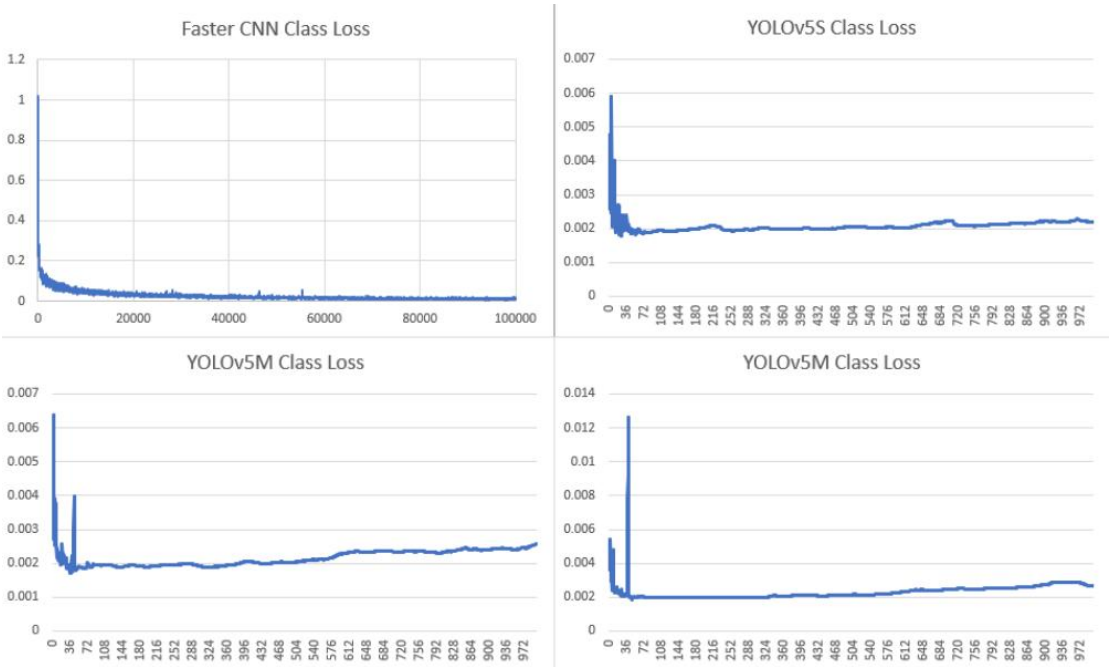

Figure 36: Class Loss Values

The table below shows the values of Precision, Recall, mAP50 and mAP50-95 for the validation set at the final epoch.

	Precision	Recall	mAP@0.5	mAP@.5:.95
Faster R-CNN	0.76422	0.73123	0.7539	0.4743
YOLOv5s	**0.82386**	0.79045	**0.7937**	**0.5121**
YOLOv5m	0.80792	0.79152	0.7837	0.4989
YOLOv5l	0.78386	**0.82118**	0.7836	0.4931

Table 2: Metrics for Faster R-CNN and YOLO architectures.

The results shown that YOLOv5 Small(S) outperformed all the other architectures. However, the loss functions' graphs indicate the YOLOv5 architectures suffer from overfitting.

The final step for the proposed research, after benchmarking, was to examine the applicability of the trained models on videos from Youtube with drone footage, to assess their capabilities in locating defects from a live video and their inference rate. YOLOv5 architectures outperformed ResNet-101 Faster-RCNN in inference speed. That was expected due to YOLOv5's architecture, designed as a single regression model as analyzed in 2.1. The performance of the four architectures on the YouTube videos is depicted on the table below:

	YOLOv5s	YOLOv5m	YOLOv5l	Faster R-CNN
Speed	**17.6** FPS	**12.2** FPS	**8.5** FPS	**1.7** FPS

Table 3: Performance of YOLO architectures on Youtube Videos.

The only issue with YOLOv5 architectures' performance was lack in accuracy. ResNet-101 Faster-RCNN estimated an entire instance of damage as a single prediction quite precisely, whereas YOLOv5 identified only portions of damage and in some cases missed a few of them.

(a) (b)

Figure 37: Surface damage detection on moving windturbines from images captured by drone: (a) YOLOv5S (b) ResNet-101 Faster-RCNN.

3.4 Conclusion

To summarize, the scientific paper [1] presented a new dataset for wind turbine damage detection, trained four different CNN architectures on this dataset and benchmarked the performance of the trained models. The next step was to examine how these models would work when analyzing a live video from a moving drone. The results proved that this approach offered satisfactory accurary and that could gradually replace the traditional costly and time consuming method with dangerous human exposure.

By utilizing their research, in the following chapters, we will try to train with this dataset a faster version of YOLOv5, YOLOv5 nano(N), using the same parameters. Later on, we will examine the inference speed of the trained model first on some CPUs/GPUs, to set a standard for its speed, and then we will study the acceleration of the model on a FPGA. Lastly, we will test the model's speed when running on a Raspberry Pi.

4. YOLOv5 Nano Training Procedure

4.1 Download/Set up of yolov5 repository

This section focuses on describing the procedure of training YOLOv5 nano(N) on the proposed dataset from [2].

First and foremost, it is necessary to clone the yolov5 repository from the github of ultralytics and install the requirements.txt file in a Unix based environment that includes Python. Yolov5 repository contains all the necessary scripts for training the model on a custom dataset, for evaluating the trained model and for exporting the trained model in the desired format.

```
git clone https://github.com/ultralytics/yolov5  # clone
cd yolov5
pip install -r requirements.txt  # install
```

Figure 38: Set up of yolov5 repository.

4.2 Dataset Arrangement

After setting up the yolov5 repository it is necessary to arrange our custom dataset in the appropriate form. Since our data are already labeled in txt format and available in Kaggle, we just download them in our local folder. Figure 39 demonstrates the format labels are saved inside a txt file:

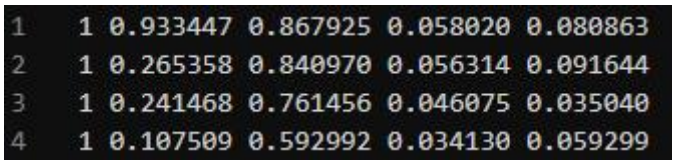

Figure 39: Labels in txt format. Every row indicates coordinates for a ground truth box, 4 in total for this figure. The first column indicates the class, the second and third columns indicate the center of the ground truth box and the fourth and fifth columns indicate the width and height of the ground truth box respectively. All the elements are normalized in the range 0 - 1.

Furthermore, we have to create two different folders, one for the images and one for the labels. Inside each of these two folders, we create two more subfolders, train and val, for the training set and the validation set. Using the script **train_val_split.py** we split the data from the local folder into training set and validation set and rearrange them inside the new created folders accordingly.

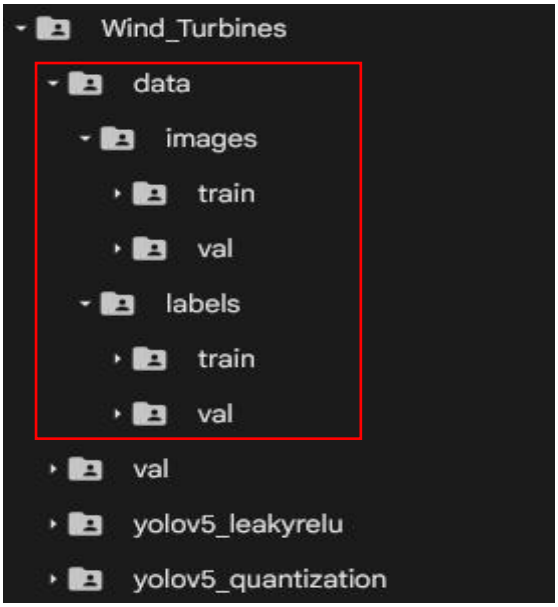

Figure 40: Arrangement of the dataset inside folders.

Then, we create one yaml file. The yaml file offers flexibility in accessing the training and validation sets. It contains the paths to the created folders and enumerates the classes for the custom dataset. Figure 41 demonstrates the yaml file used for our training procedure:

```yaml
path: /content/gdrive/My Drive/YOLOv5_nano_original_dataset_small/data
train: images/train
val: images/val

# Classes
names:
  0 : dirt
  1 : damage
```

Figure 41: Yaml file for our training.

4.3 Training tips by Ultralytics

Before starting the training procedure it worths mentioning some tips for best training results provided by Ultralytics, in order to assess the completeness of the proposed dataset and estimate its performance:

- Number of **images per class** ≥ **1500**.
- Number of **instances**(labeled objects) **per class** ≥ **10000**.
- **Image variety**. It is recommended images from different times of day, different seasons, different weather, different lighting, different angles, different sources to be used for real-world use cases.
- **Labels consistency**. All labels from all classes must be labeled. Partial labeling would not work.
- **Labels accuracy**. Labels must closely enclose each object.
- **Backround Images**. Backround images are images with no labels. They are added to the dataset to reduce false positives(FP). It is recommended 0 - 10% of backround images in the dataset.

Our dataset in general terms meets the requirements of the training tips. The only issues are the small amount of images in the dirt class, which will be compensated by the build-in Data Augmentation system and the unbalance between the labeled and backround images. The last is more concerning as the number of Backround images is approximately **78%** of the total dataset so it might determine the performance of the model.

4.4 Training of the Model

For training the model we will utilize Google Colab, a hosted jupyter notebook service which provides access to computing resources like GPUs and TPUs. This environment specializes in machine learning applications. We train the model using the following command:

```
python train.py --data mydata.yaml --weights yolov5n.pt --imgsz 640 --batch 16 --epochs 1000 --cache
```

Figure 42: Command for training the model.

The script for training the model is in **yolov5_leakyrelu/train.py**. The inputs for the arguments can be explained as follows:

- **data**: Takes the value mydata.yaml which is the file explained above.
- **weights**: Takes the value of yolov5n.pt which are the pretrained weights for yolov5n(nano) on COCO dataset.
- **imgsz**: Takes the value of 640 which is the size that input images should be resized to. This is the default input size for yolov5 models.
- **batch**: Takes the value 32 and represents the batch size.
- **epochs**: Takes the value 1000 and is the number of epochs for training.
- **cache**: Is a value that indicates that data should be cached into ram instead of the disc.

All the other hyperparameters regarding the optimization technique maintain their default values as are the same with the values used in [1]. The result of the training procedure provides a model in pytorch format.

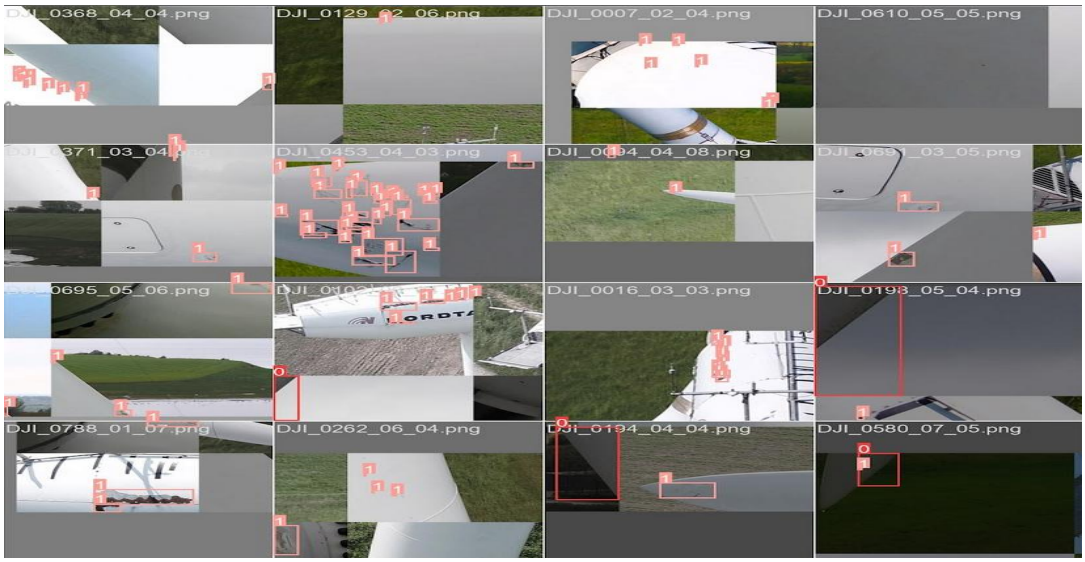

Figure 43: Batch of images during training.

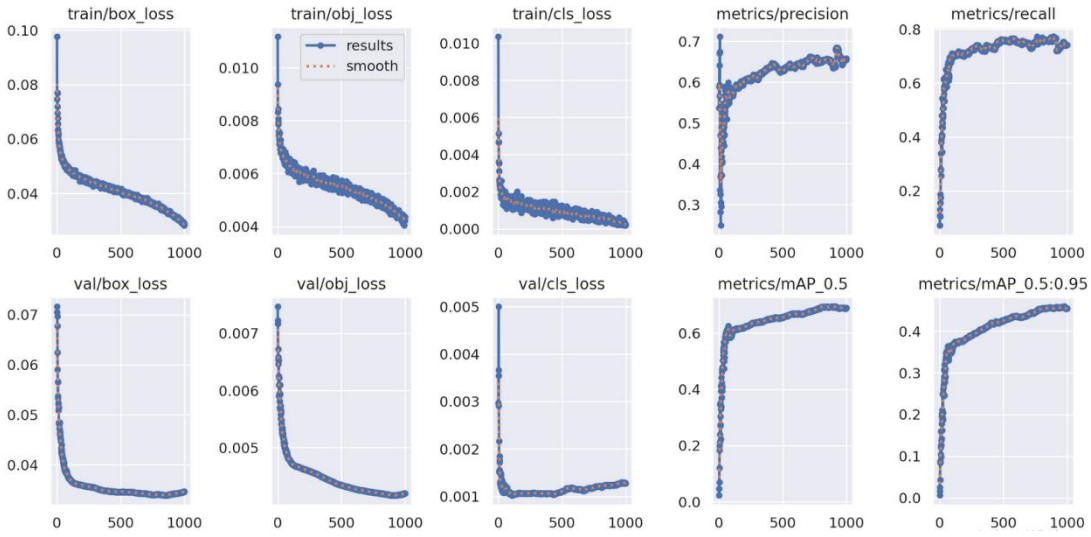

Figure 44: Training results after 1000 epochs(original dataset).

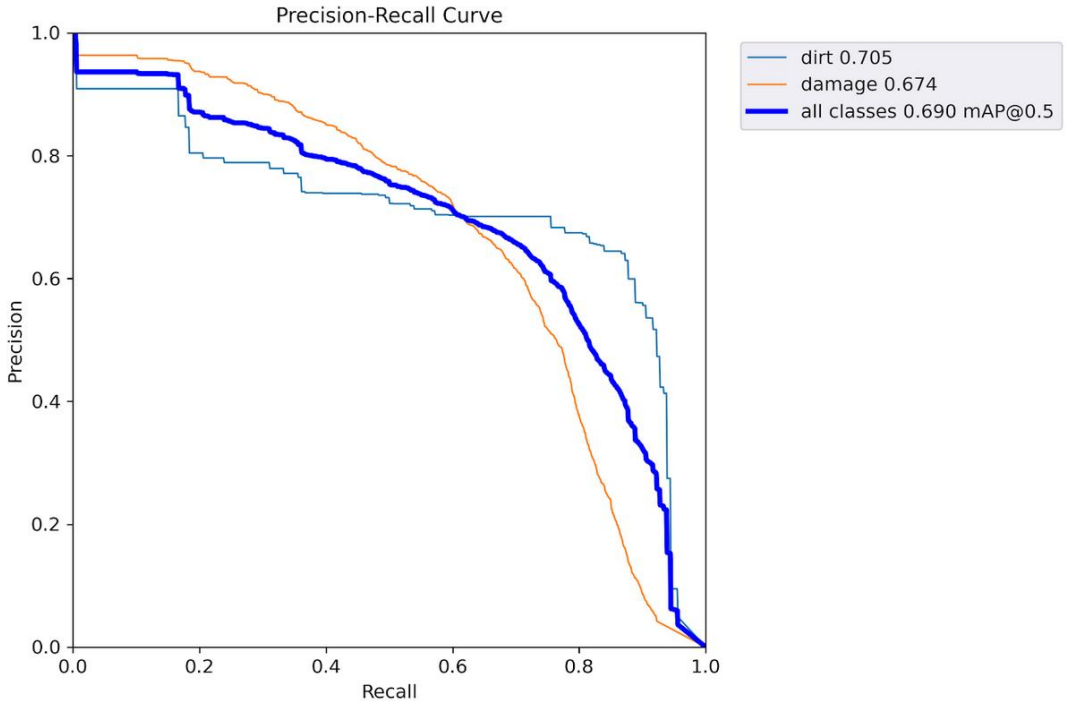

Figure 45: Precision-Recall curve.

```
     Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
   999/999      1.53G    0.02885  0.004357  0.0002209          0        640: 100%|██████████| 590/590 [00:40<00:00, 14.61it/s]
                Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 127/127 [00:11<00:00, 10.95it/s]
                  all       4042       2774      0.655      0.742      0.689      0.455

1000 epochs completed in 14.560 hours.
Optimizer stripped from runs/train/exp5/weights/last.pt, 3.8MB
Optimizer stripped from runs/train/exp5/weights/best.pt, 3.8MB

Validating runs/train/exp5/weights/best.pt...
Fusing layers...
Model summary: 157 layers, 1761871 parameters, 0 gradients, 4.1 GFLOPs
                Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 127/127 [00:12<00:00, 10.02it/s]
                  all       4042       2774      0.646      0.751       0.69      0.459
                 dirt       4042        180       0.66      0.817      0.705      0.554
               damage       4042       2594      0.632      0.685      0.674      0.364
Results saved to runs/train/exp5
```

Figure 46: Performance of first model on first validation dataset.

From results it is clear that we are experiencing a significant degradation in accuracy. The yolov5 models Small, Medium and Large used in [1] achieved a mAP around 0.79, but in our case we got 0.69. This could come from the architecture of yolov5 nano as it is much smaller than the other 3, so it is possible to suspect that could lack in accuracy, and from the unbalance between labeled and backround images as we mentioned above.

Based on the previous results we will perform another training of the model but this time we will use as a dataset only the labeled images, around 3000 in total.

The purpose of doing this is to examine how the model will perform without backround images. We should mention though that since we will use a quarter of the original dataset it could be possible to experience a greater degradation in accuracy.

To retrain our model we first need to separate the labeled images from the original dataset, rearrange the images in training set and validation set, and then change the base path inside our yaml file. We accomplish this by using the script **data_with_labels.py**. The model can be retrained using the same command from figure 42 but now with the updated yaml file.

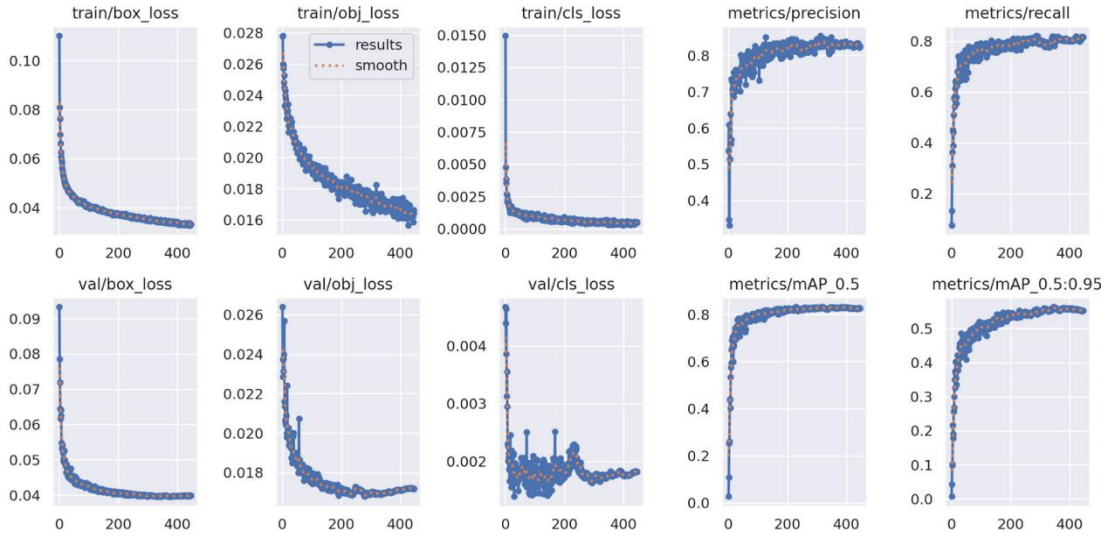

Figure 47: Training results after 1000 epochs(second dataset).

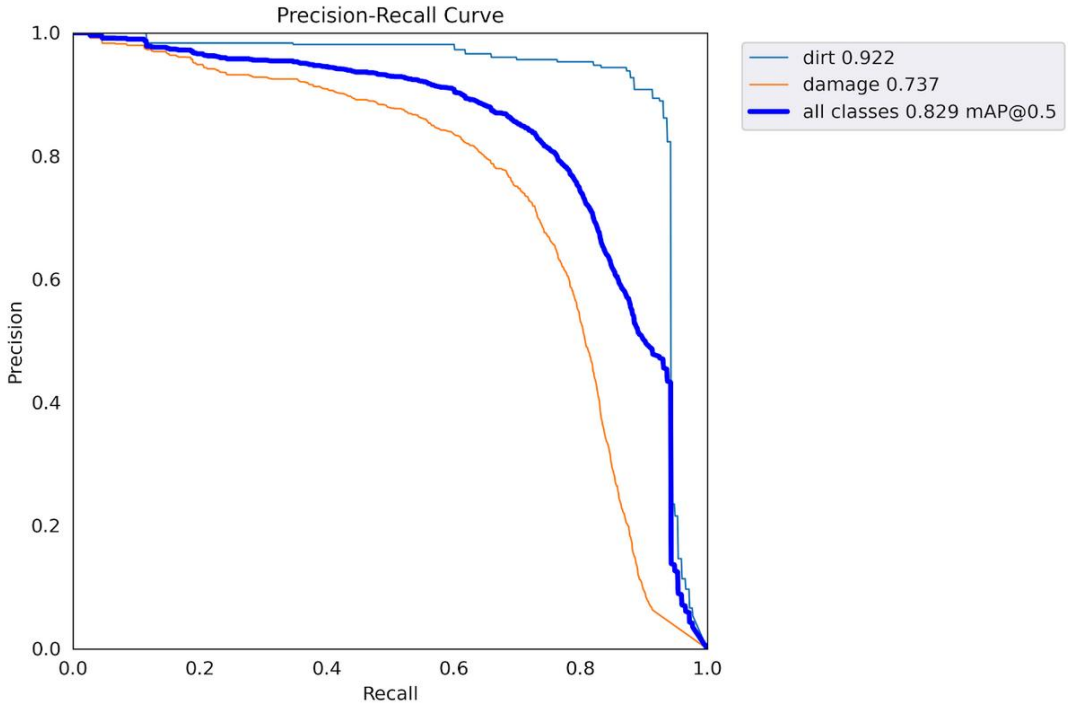

Figure 48: Precision-Recall curve for second model.

```
Model summary: 157 layers, 1761871 parameters, 0 gradients, 4.1 GFLOPs
val: Scanning /home/georgegio/Vitis-AI-3.0/yolov5/data/data/labels/val... 899 images, 0 backgrounds, 0 corrupt: 100%| | 899/899 [00:10<00:00, 89.06it/s]
val: New cache created: /home/georgegio/Vitis-AI-3.0/yolov5/data/data/labels/val.cache
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%| | 57/57 [00:38<00:00,  1.46it/s]
                   all        899       2785      0.834      0.791       0.83      0.563
                  dirt        899        173      0.916      0.884      0.922      0.738
                damage        899       2612      0.752      0.697      0.739      0.388
Speed: 1.7ms pre-process, 38.2ms inference, 0.7ms NMS per image at shape (16, 3, 640, 640)
Results saved to runs/val/exp17
```

Figure 49: Performance of second model on second validation dataset.

From results we can see that the model achieved a mAP of 0.83. This value is higher than the one the previous model reached and closer to the accuracy of the other yolov5 architectures from [1]. It seems that the problem was the unbalance between labeled and backround images but we should not forget that this dataset is much smaller than the original one. To ensure a fair comparison between the two models we will evaluate the second model on the validation dataset of the first, which includes around 4000 images, to assess its capabilities and on unlabeled data.

```
val: Scanning /content/gdrive/My Drive/Wind_Turbines/data/labels/val.cache... 899 images, 3143 backgrounds, 0 corrupt: 100% 4042/4042 [00:00<?, ?it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 253/253 [06:43<00:00,  1.60s/it]
                   all       4042       2774      0.698      0.814      0.714      0.504
                  dirt       4042        180      0.662      0.906      0.676      0.558
                damage       4042       2594      0.734      0.722      0.753      0.449
Speed: 0.1ms pre-process, 3.5ms inference, 1.3ms NMS per image at shape (16, 3, 640, 640)
Results saved to runs/val/exp22
```

Figure 50: Performance of second model on first validation dataset.

We also evaluate how the second model performs on the entire original dataset including all labeled and backround images.

```
val: New cache created: /content/gdrive/My Drive/Wind_Turbines/data/labels/train.cache
                 Class     Images  Instances          P          R      mAP50   mAP50-95:   0% 0/842 [00:00<?, ?it/s]WARNING ⚠ NMS time limit 1.300s exceeded
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 842/842 [03:03<00:00,  4.59it/s]
                   all      13470       9349      0.668      0.828      0.704      0.498
                  dirt      13470        580      0.601      0.905      0.648      0.545
                damage      13470       8769      0.736      0.751      0.761       0.45
Speed: 0.2ms pre-process, 2.9ms inference, 2.0ms NMS per image at shape (16, 3, 640, 640)
Results saved to runs/val/exp21
```

Figure 51: Performance of second model on entire original dataset.

The second model achieved greater mAP than the first model on the same dataset proving that the number of labeled images matters way more than the total number of images during training. It also proves that the smallest yolov5 architecture can reach and maintain the performance of the other bigger architectures up yo a certain point.

It worths exploring how yolov5 nano model can perform on other datasets derived from the original one. These datasets can include greater number of labeled images by applying data augmentation techniques on the labeled data from the original dataset and balanced amounts of backround images chosen from the original ones. Additionally, it is interesting an entirely new dataset to be created by acquiring new images based on the training tips of the ultralytics.

Nevertheless, the main purpose of this thesis is the acceleration of this algorithm on hardware and not the creation of the optimal dataset for achieving the best possible mAP with yolov5 architecures for wind turbine surface damage detection. So, we will not analyze these dataset ideas any further. From now on, as a float model we will utilize the second trained model, occuring from the 3000 images dataset, as it performed better and favors us in trasporting lesser data between the hardware devices on next sections, reducing transportation times.

4.5 Inference on different CPUs/GPUs

After succesfully having selected our desired trained model we will first start assessing its performance across different CPUs/GPUs. Before starting our assessment we provide a summary of evaluation results in table 4 for our trained model on the validation dataset of 899 images. These would be the reference results during inference on different CPUs/GPUs, inference on FPGA and inference on Raspberry Pi. These are the results of the pytorch model(yolov5n_cd_pt.pt). Its evaluation is possible with the following command:

```
python3 val.py --weights models/best.pt --batch-size 16 --data data/data.yaml --imgsz 640
```

Figure 52: Evaluation Command for batch size equal to 16.

Class	Images	Instances	P	R	mAP50	mAP50-95
all	899	2785	0.834	0.791	**0.83**	**0.563**
dirt	899	173	0.916	0.884	0.922	0.738
damage	899	2612	0.752	0.697	0.739	0.388

Table 4: Reference results for the second YOLOv5n model trained on the small dataset.

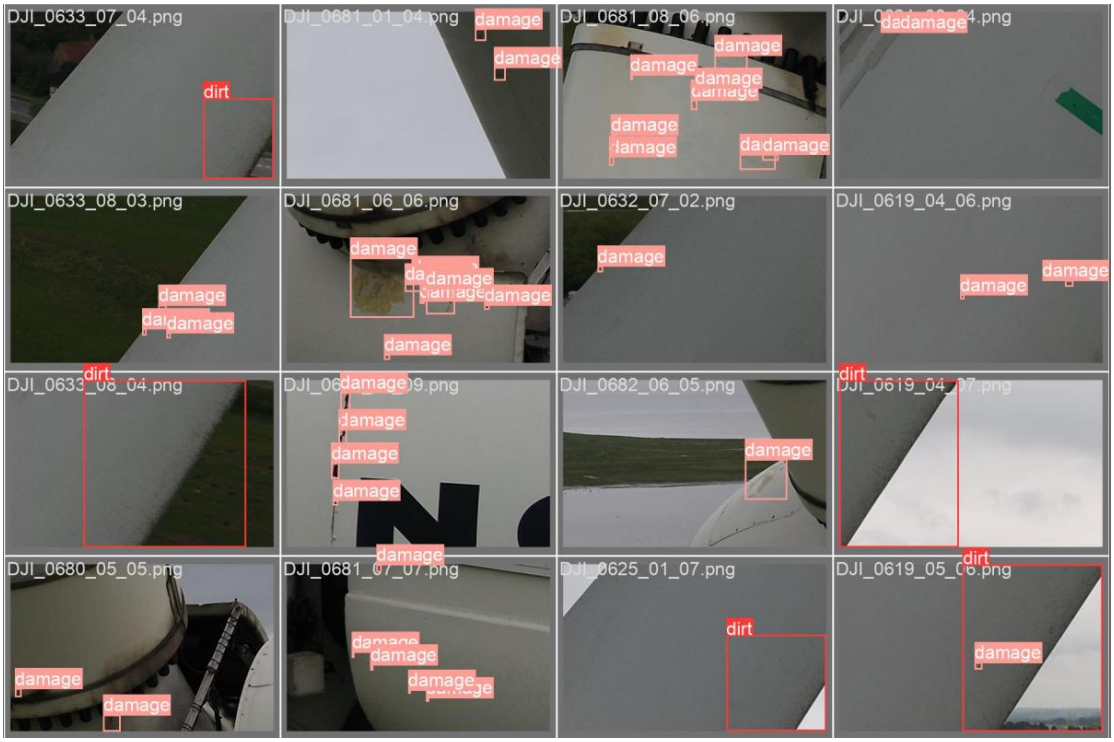

Figure 53: Labels for batch of images from validation dataset.

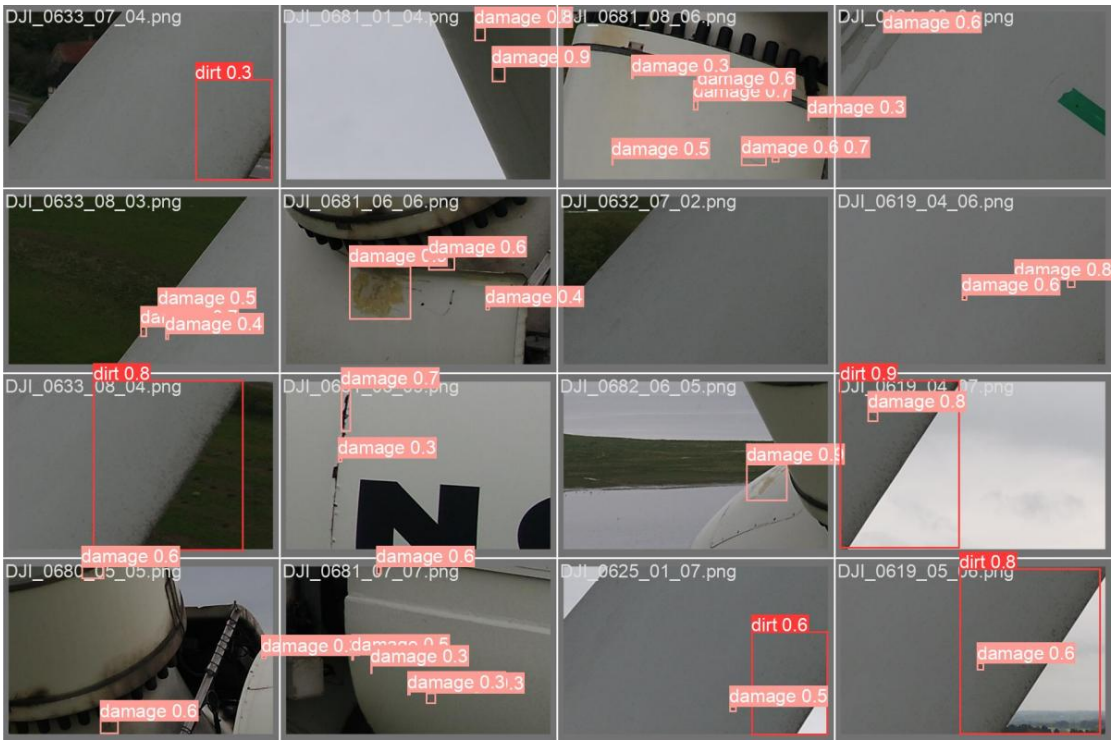

Figure 54: Predictions for batch of images from validation dataset.

For the inference of our pytorch model we will utilize our local CPU and some TPUs/GPUs from Google Colab. Table 5 demonstrates inference results for the pytorch model with batch size equal to 16. Pre-process, is the time spent for preparing the input batch in the input form that the model desires. This procedure includes transforming pixels from integer format to floating point format and normalization of their values as the model works with processing normalized pixels. Inference, is the time spent from the model to compute its output from the moment it gets its input. Finally, NMS is the time spent for the NMS post processing procedure. This procedure involves filtering predictions according to specific thresholds set by the user. So, basically we are removing predictions with low confidence scores and predictions that overlap from the $[batch\_size, 25200, 7]$ vector. All of the above times are computed as averages from the times of all images.

Hardware	Pre-Process(ms)	Inference(ms)	NMS(ms)
TPU v2-8 Google Colab	3.6	**210.4**	9.4
CPU of PC(AMD Ryzen 5 3600X)	1.5	**44.8**	0.8
Tesla T4 GPU Google Colab (15102MiB)	0.2	**4.1**	3.2
NVIDIA L4 GPU Google Colab (22700MiB)	0.1	**0.8**	2.7

Table 5: Inference results for pytorch model with batch size equal to 16.

The FPGA we will use in the next section can process only an image at a time so we rerun inference for the pytorch model but for batch size equal to 1 this time, to ensure a fair comparison.

Hardware	Pre-Process(ms)	Inference(ms)	NMS(ms)
TPU v2-8 Google Colab	22.1	**609.9**	12.2
CPU of PC(AMD Ryzen 5 3600X)	0.4	**33**	0.6
Tesla T4 GPU Google Colab (15102MiB)	0.2	**6.8**	1.9
NVIDIA L4 GPU Gogle Colab (22700MiB)	0.2	**6.5**	1.6

Table 6: Inference results for pytorch model with batch size equal to 1.

The results show that the most expensive GPUs outperform any CPUs/TPUs. These GPUs are 5 times faster than the CPU of our local machine. This is expected as those hardware devices are specifically designed for neural network architectures and for working with big batches of images. What's more concerning is the performance of the TPU. The TPU is a device designed to process matrices so it should at least have a performance close to the CPU if not better. The most logical explanation is that we are are experiencing transportation speed issues regarding the communication between the server, where the TPU is located, and our workbench in Google Colab.

Another way to improve inference performance on a CPU is by changing the format of our trained model. Ultralytics provide a table with different available formats for inference of yolov5.

Format	`export.py --include`	Model
PyTorch	-	`yolov5s.pt`
TorchScript	`torchscript`	`yolov5s.torchscript`
ONNX	`onnx`	`yolov5s.onnx`
OpenVINO	`openvino`	`yolov5s_openvino_model/`
TensorRT	`engine`	`yolov5s.engine`
CoreML	`coreml`	`yolov5s.mlmodel`
TensorFlow SavedModel	`saved_model`	`yolov5s_saved_model/`
TensorFlow GraphDef	`pb`	`yolov5s.pb`
TensorFlow Lite	`tflite`	`yolov5s.tflite`
TensorFlow Edge TPU	`edgetpu`	`yolov5s_edgetpu.tflite`
TensorFlow.js	`tfjs`	`yolov5s_web_model/`
PaddlePaddle	`paddle`	`yolov5s_paddle_model/`

Figure 55: Available formats for yolov5 inference.

The benchmarks for these formats on a Colab Pro CPU are demonstrated in figure 56.

```
benchmarks: weights=/content/yolov5/yolov5s.pt, imgsz=640, batch_size=1,
Checking setup...
YOLOv5 🚀 v6.1-135-g7926afc torch 1.10.0+cu111 CPU
Setup complete ✅ (8 CPUs, 51.0 GB RAM, 41.5/166.8 GB disk)

Benchmarks complete (241.20s)
                   Format  mAP@0.5:0.95  Inference time (ms)
0                 PyTorch        0.4623               127.61
1             TorchScript        0.4623               131.23
2                    ONNX        0.4623                69.34
3                OpenVINO        0.4623                66.52
4                TensorRT           NaN                  NaN
5                  CoreML           NaN                  NaN
6   TensorFlow SavedModel        0.4623               123.79
7     TensorFlow GraphDef        0.4623               121.57
8         TensorFlow Lite        0.4623               316.61
9     TensorFlow Edge TPU           NaN                  NaN
10          TensorFlow.js           NaN                  NaN
```

Figure 56: Benchmarks for inference of different yolov5 formats.

The above results imply that if we transform our final trained model from its original pytorch format to an ONNX or a OpenVINO format we can almost cut inference time in half. Yolov5 repository disposes an exportation script for the purpose of acquiring all of the above formats. We export the pytorch model to ONNX and OpenVINO formats using the following commands:

```
!python export.py --weights runs/train/exp/weights/yolov5n_cd_pt.pt --include onnx --batch-size 1
```

Figure 57: Exportation of yolov5n_cd_pt.pt to onnx format.

```
!python export.py --weights runs/train/exp/weights/yolov5n_cd_pt.pt --include openvino --batch-size 1
```

Figure 58: Exportation of yolov5n_cd_pt.pt to openvino format.

The exportation code is in **yolov5_leakyrelu/export.py**. The inference results for the new formats of our trained model can be seen in table 7:

Class	Images	Instances	P	R	mAP50	mAP50-95
all	899	2785	0.766	0.75	0.785	0.493
dirt	899	173	0.797	0.798	0.831	0.601
damage	899	2612	0.735	0.703	0.738	0.384

Table 7: Inference results for onnx and openvino models with batch size equal to 1.

The exportation to different formats seems to affect accuracy as it decreases it from mAP 0.83 to 0.785 for both formats. The reason might be in how the neural network is saved into each desired format each time. Since the original format of yolov5 is the pytorch one it makes sense for that format to achieve the best possible accuracy as it is the original representation of the neural network's architecture. Because the decrease in accuracy is consider within limits we will accept the above results.

Hardware	Pre-Process(ms)	Inference(ms)	NMS(ms)
TPU v2-8 Google Collab	77.8	**416.9**	49
CPU of PC(AMD Ryzen 5 3600X)	1	**23.2**	1.3

Table 8: Inference results for onnx model with batch size equal to 1.

Hardware	Pre-Process(ms)	Inference(ms)	NMS(ms)
CPU of PC(AMD Ryzen 5 3600X)	0.5	**21.9**	0.9

Table 9: Inference results for openvino model with batch size equal to 1.

Inference times show a dropage from 33ms to 23.2ms and 21.9ms respectively, but we receive a slight increase in Pre-process and NMS. Cumulatively, in CPU, for each image we spend for processing 8.5ms less in onnx format and 10.7ms less in openvino format. That is a very desirable result as these formats seem suitable for speeding up yolov5 nano's inference in Raspberry Pi. We examine this thoroughly in a future section.

5. FPGA Implementation and Acceleration

5.1 Setting up the Host Environment

This is the main section of this diploma thesis. Here, a procedure will be described regarding how our trained model can be run on an FPGA. FPGAs are hardware devices designed to perform different kind of computations by utilizing configurable logic blocks(CLBs). That helps users to reconfigure the hardware to meet their specific requirements. Therefore, FPGAs are considered suitable for hardware acceleration by implementing complex parts of algorithms in desired formats and neural networks by speeding up matrix multiplications. Xilinx's FPGA devices consist of programmable engines, Deep Learning Processing Units(DPUs), capable of running convolutional neural network inference faster. There are specialized instruction sets for each of these DPUs, these micro-coded processors, that enable efficient processing for different convolutional neural network architectures achieving their acceleration on FPGAs.

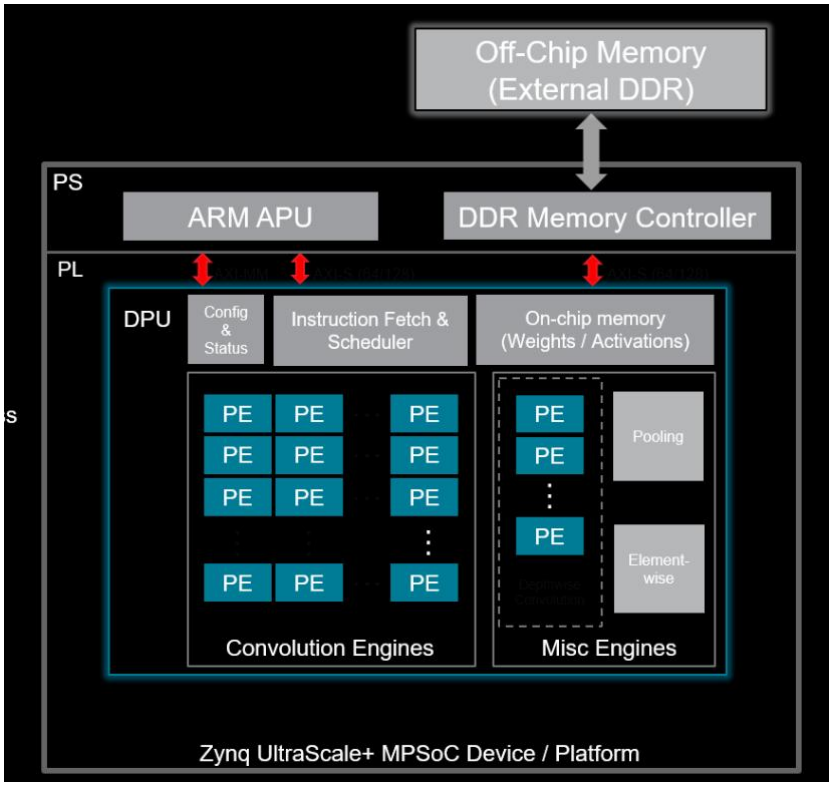

Figure 59: DPU architecture.

Figure 59 analyzes the basic structure of the DPU for Zynq UltraScale+ MPSoC devices. The main processing unit is the PE, programmable engine, which contains general elements of AMD's programmable logic fabric, such as DSP, BlockRAM, UltraRAM, LUTs and Flip Flops. This processing unit also disposes a Scheduler for scheduling inference across the DPU or multiple DPUs and an Instruction Fetcher for the compiled instructions of the input model.

However, before running inference on a DPU our trained model should be processed accordingly. This procedure includes utilization of specific tools that will transform the model from the original pytorch format(.pt) into a format(.xmodel) able to be run on the DPU. We will accomplish this by using docker, a platform suitable for running containerized applications and microservices [25] and Vitis AI, an artificial inteligence development environment [26].

To begin with, we need to set up our host environment and enable these tools. First, we download docker in a Linux environment like Virtual Box. In a terminal:

sudo apt-get update

sudo apt install docker.io

sudo systemctl enable docker

Now, docker runs in our environment and is ready to enable containerized applications. The next step is download the Vitis AI repository from github. Vitis AI is an AI development environment specialized in accelerating AI inference on Xilinx hardware platforms. It consists of libraries, models, tools, IP cores and example designs offering to users without FPGA knowledge the ability to develop and run deep learning inference applications.

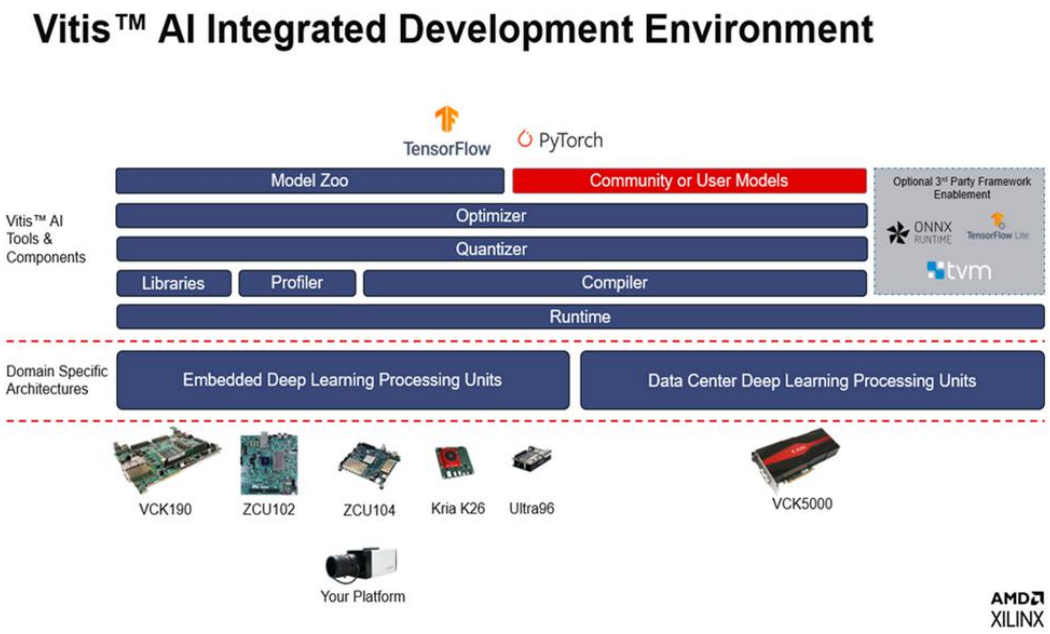

Figure 60: Vitis AI Integrated Development Environment version 3.0.

We clone Vitis AI repository to obtain all the necessary components and then we pull the appropriate Vitis AI Docker Image:

```
[Host]$ git clone https://github.com/Xilinx/Vitis-AI
[Host]$ cd Vitis-AI
```

Figure 61: Command for cloning Vitis AI.

```
docker pull xilinx/vitis-ai-pytorch-cpu:ubuntu2004-3.0.0.106
```

Figure 62: Command for pulling the appropriate image of Vitis AI.

The only thing that is left is to run the **docker_run.sh** script from Vitis AI repository and finally get the Vitis AI Docker Container up and running:

```
sudo ./docker_run.sh xilinx/vitis-ai-pytorch-cpu:ubuntu2004-3.0.0.106
```

Figure 63: Command for running the image of Vitis AI.

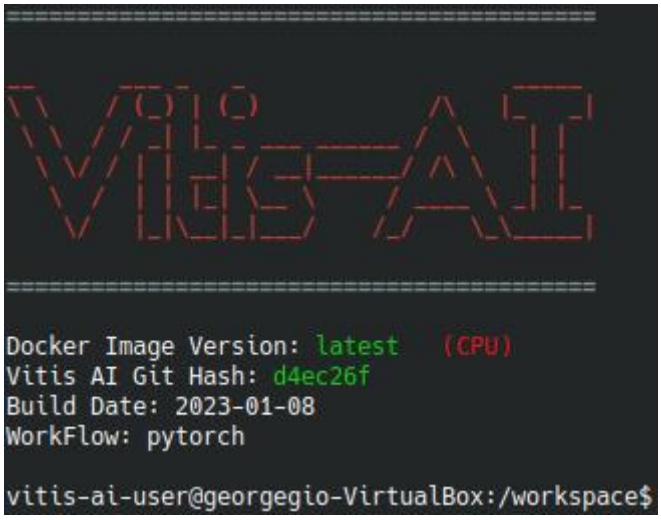

Figure 64: Vitis AI Docker Container version ubuntu2004-3.0.0.106.

To finish setting up our host we enable **vitis-ai-pytorch** anaconda environment provided by Vitis AI development kit, to ensure safety and availability of packages for our pytorch workflow, and **download** the **yolov5** repository inside the Vitis AI repository.

```
conda activate vitis-ai-pytorch
```

Figure 65: Activation of vitis-ai-pytorch anaconda environment.

Figure 66 demonstrates the workflow we are going to use in order to bring the pytorch model into the final xmodel format:

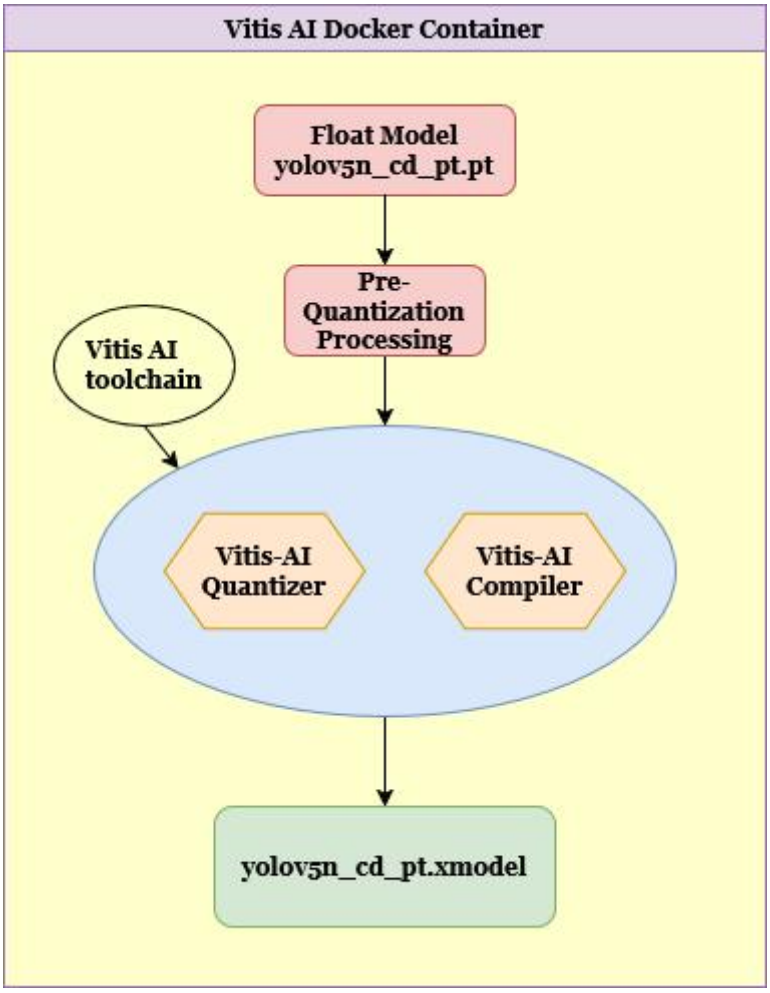

Figure 66: Workflow for processing the float model.

5.2 Pre-Quantization Processing

Before using the Vitis AI development tools we have to preprocess our model accordingly. Preprocessing includes making changes in the architecture of our model to guarantee compatibility with Vitis AI tools, especially Vitis AI Quantizer.

Inside yolov5 repository we need to make the following changes:

- **Replace** the **SiLU** activation function with **LeakyReLU**(using 26/256 as a negative slope). Vitis AI does not support SiLU activation function and out of all supported activations LeakyReLU with this negative slope is considered the best solution. This replacement should be done in **models/common.py**(line 66 and 147) and **experimental.py**(line 55).

```
self.act = nn.LeakyReLU(negative_slope=0.1, inplace=True) # nn.SiLU()
```

Figure 67: Replacement of SiLU with LeakyReLU.

After that it is **important to redo the training from the beginning** to acquire a new pytorch(.pt) model, otherwise we will face compatibility issues when using Vitis AI tools occuring from this architectural change. The new trained model, **yolov5n_wt.pt**, will be the model used from now on and referred as **float** model. Let us note that the results presented in section 4 are computed with the LeakyReLU activation function due to this change.

- Inside **models/yolo.py** there is **class Detect(nn.Module)** which implements the **Head** of yolov5n. This class includes **permute** and **view** methods that are not supported in Vitis AI, so is deemed necessary to modify it. We will remove some parts from the forward method inside the class and re-implement them as separate functions in the post-processing stage.

```python
def forward(self, x):
    """Processes input through YOLOv5 layers, altering shape for detection: `x(bs, 3, ny, nx, 85)`."""
    z = []  # inference output
    for i in range(self.nl):
        x[i] = self.m[i](x[i])  # conv
        bs, _, ny, nx = x[i].shape  # x(bs,255,20,20) to x(bs,3,20,20,85)
        x[i] = x[i].view(bs, self.na, self.no, ny, nx).permute(0, 1, 3, 4, 2).contiguous()

        if not self.training:  # inference
            if self.dynamic or self.grid[i].shape[2:4] != x[i].shape[2:4]:
                self.grid[i], self.anchor_grid[i] = self._make_grid(nx, ny, i)

            if isinstance(self, Segment):  # (boxes + masks)
                xy, wh, conf, mask = x[i].split((2, 2, self.nc + 1, self.no - self.nc - 5), 4)
                xy = (xy.sigmoid() * 2 + self.grid[i]) * self.stride[i]  # xy
                wh = (wh.sigmoid() * 2) ** 2 * self.anchor_grid[i]  # wh
                y = torch.cat((xy, wh, conf.sigmoid(), mask), 4)
            else:  # Detect (boxes only)
                xy, wh, conf = x[i].sigmoid().split((2, 2, self.nc + 1), 4)
                xy = (xy * 2 + self.grid[i]) * self.stride[i]  # xy
                wh = (wh * 2) ** 2 * self.anchor_grid[i]  # wh
                y = torch.cat((xy, wh, conf), 4)
            z.append(y.view(bs, self.na * nx * ny, self.no))

    return x if self.training else (torch.cat(z, 1),) if self.export else (torch.cat(z, 1), x)
```

Figure 68: Forward method in Detect(nn.Module) class before modifications.

```python
# New forward method for quantization.
def forward(self, x):
    z = []  # inference output
    for i in range(self.nl):
        x[i] = self.m[i](x[i])  # conv
    return x
```

Figure 69: Forward method in Detect(nn.Module) class after modifications.

Figures 68 and 69 demonstrate the modifications inside the forward method. It is clear that our output has changed from $[batch\_size, 25200, 7]$ to a 3 element list. This list contains as elements the output vectors of the detection heads analyzed in section 2:

$$
\begin{bmatrix}
[batch\_size, 3 \cdot (5 + number\_of\_classes), 80, 80], \\
[batch\_size, 3 \cdot (5 + number\_of\_classes), 40, 40], \\
[batch\_size, 3 \cdot (5 + number\_of\_classes), 20, 20]
\end{bmatrix}
$$

5.3 Quantization Process

Inference, is a process with high demands in memory bandwidth as it aims for low-latency and high-throughput when it comes to Edge Applications. Quantization and Pruning are techniques designed to address these issues and achieve high performance with energy efficiency at cost of a low decrease in accuracy. Pruning reduces the number of required operations by removing weights or nodes inside the network. On the other hand, Quantization transforms weights and activations from floating point format to a fixed-point integer format.

Vitis AI environment, disposes both a Quantizer and a Pruner. For inference on a FPGA only quantization is necessary so the pruning step will be omitted. Vitis AI Quantizer converts the 32-bit-floating-point weights and activations to 8-bit integers(INT8) format. This reduces computing complexity, makes the model faster and maintains a prediction accuracy close enough to the float's model.

Figure 70 shows the workflow of Vitis AI Quantizer:

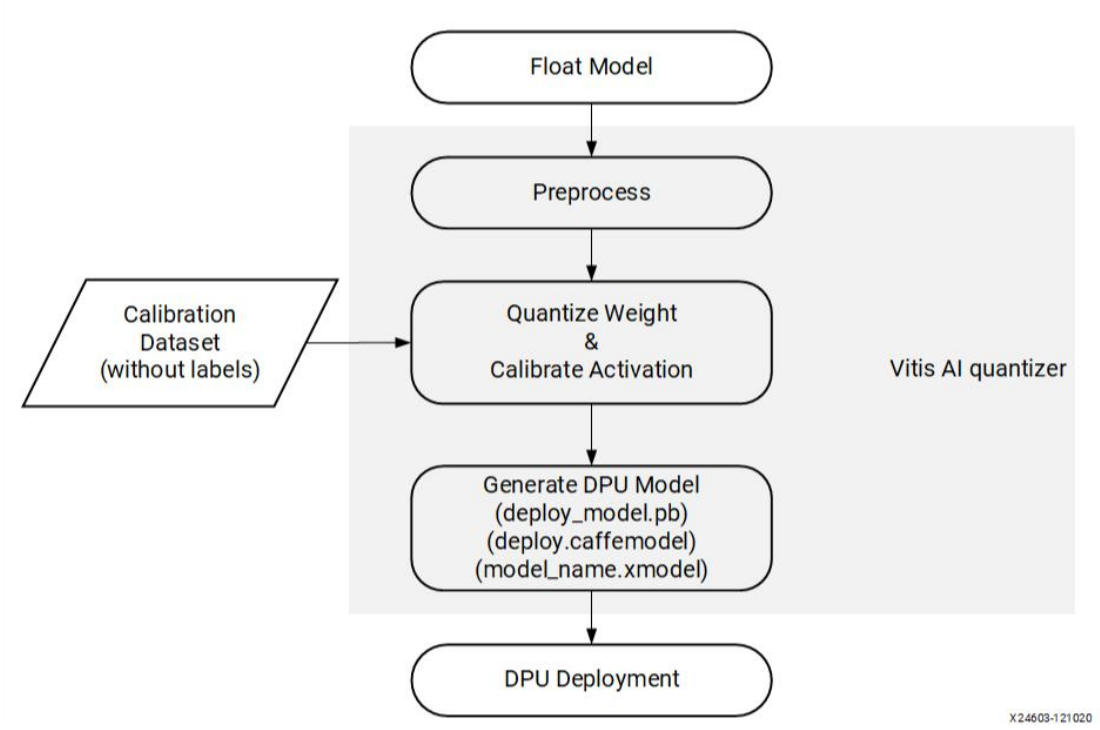

Figure 70: Workflow of Vitis AI Quantizer.

The Vitis AI Quantizer takes the float model as input and first performs pre-processing. Pre-processing includes batchnorms folding and removal of nodes not required for inference. Afterwards, quantization to the given bit width is performed for weights, biases and activations. Vitis AI environment contains four versions of quantizers vai_q_tensorflow, vai_q_tensorflow2, vai_q_pytorch and vai_q_onnx each of them designed to process the corresponding types of models. For our model we will use the vai_q_pytorch quantizer since the trained model is in pytorch format.

There is an optional tool called Inspector which can be used before quantization. The Inspector performs a quick scan on the model and then provides information regarding which of its operators can be quantized along with guidance messages for the operators that can not.

To start using the Inspector and all the others tools of the Quantizer after setting up our host environment, as described in 5.1, we inspect our model by running the following command in the terminal:

```
python yolov5n_quant.py --quant_mode float --inspect --target DPUCZDX8G_ISA1_B4096 --model_dir models
```

Figure 71: Inspection Command.

```
[VAIQ_NOTE]: All the operators are assigned to the DPU(see more details in 'quantize_result/inspect_DPUCZDX8G_ISA1_B4096.txt')
[VAIQ_NOTE]: =>Finish inspecting.
```

Figure 72: Inspection Result.

Inspection indicates that every operator inside the model can be assigned to the DPU which means the float model is in appropriate form for quantization.

To quantize our model Vitis AI offers two quantization techniques:

- Post Training Quantization(PTQ): PTQ is a quantization technique applied to the model after training. It reduces its size and affects its accuracy. For PTQ a portion of the original dataset is needed, about 100 - 1000 data, to be forwarded through the model and determine the activation values in a lower precision.

- Quantization Aware Training(QAT): QAT is a quantization method only used when PTQ leads to a huge decrease in accuracy. QAT involves training/fine-tuning a model with quantized parameters from the beginning. For this procedure the entire original dataset is needed.

Out of these two quantization techniques we will proceed with PTQ. If the accuracy of the quantized model arises lesser than a specific threshold then we might reconsider using QAT.

To start using the Quantizer we choose 150 images from the training dataset, for calibration, and then we run the following command in the terminal:

```
python yolov5n_quant.py --quant_mode calib --model_dir models --data_dir data/data_fast_finetune/images --target DPUCZDX8G_ISA1_B4096
```

Figure 73: Command for quantization calibration.

This command produces two files, a DetectMultiBackend_int.py file which is the quantized vai_q_pytorch model format and a quant_info.json which includes quantization steps of tensors.

To generate the quantized xmodel file meant for the compiler we run the following command:

```
python yolov5n_quant.py --quant_mode test --model_dir models --data_dir data/data_fast_finetune/images --target DPUCZDX8G_ISA1_B4096 --batch_size 1 --deploy
```

Figure 74: Command for exporting the quantized model.

This command results in producing a DetectMultiBackend_int.xmodel file which will be fed into the Compiler and two additional files, a DetectMultiBackend_int.pt file and a DetectMultiBackend_int.onnx file. The last files can be used to evaluate the accuracy of the quantized model. By using DetectMultiBackend_int.pt and the evaluation script **model_inference.py** we run the following command to evaluate the quantized model's accuracy:

```
python model_inference.py --model_dir quantize_result/DetectMultiBackend_int.pt --val_data_dir data/data/images/val --batch_size 1
```

Figure 75: Command for evaluating the accuracy of the quantized model.

```
                 Class     Images  Instances          P          R      mAP50   mAP50-95:
                   all        899       2785      0.752      0.743      0.777      0.494
                  dirt        899        173      0.798      0.775      0.814      0.602
                damage        899       2612      0.706      0.712       0.74      0.386
Speed: 1.3ms pre-process, 178.8ms inference, 0.5ms NMS per image at shape (1, 3, 640, 640)
-------- End of YOLOv5 nano on Wind Turbines data set, evaluation test --------
```

Figure 76: Results from evaluation of the quantized model.

The accuracy of the quantized model achieved a mAP of 0.777 which is a little lower than the float's model accuracy(0.83) but within limits so, we will accept this value and proceed in compiling the model.

The script **yolov5n_quant.py** which is run for the inspection, the calibration and the exportation commands is located in **yolov5_quantization** directory in **github**.

5.4 Compilation Process

Subsequently, having sucessfully quantized our float model we are ready to feed it to Vitis AI Compiler and map it into a DPU specific instruction set. Vitis AI Compiler is used to construct an internal computation graph which consists of independent control and data flow representations. This graph form is called Xilinx Intermediate Representation(XIR).

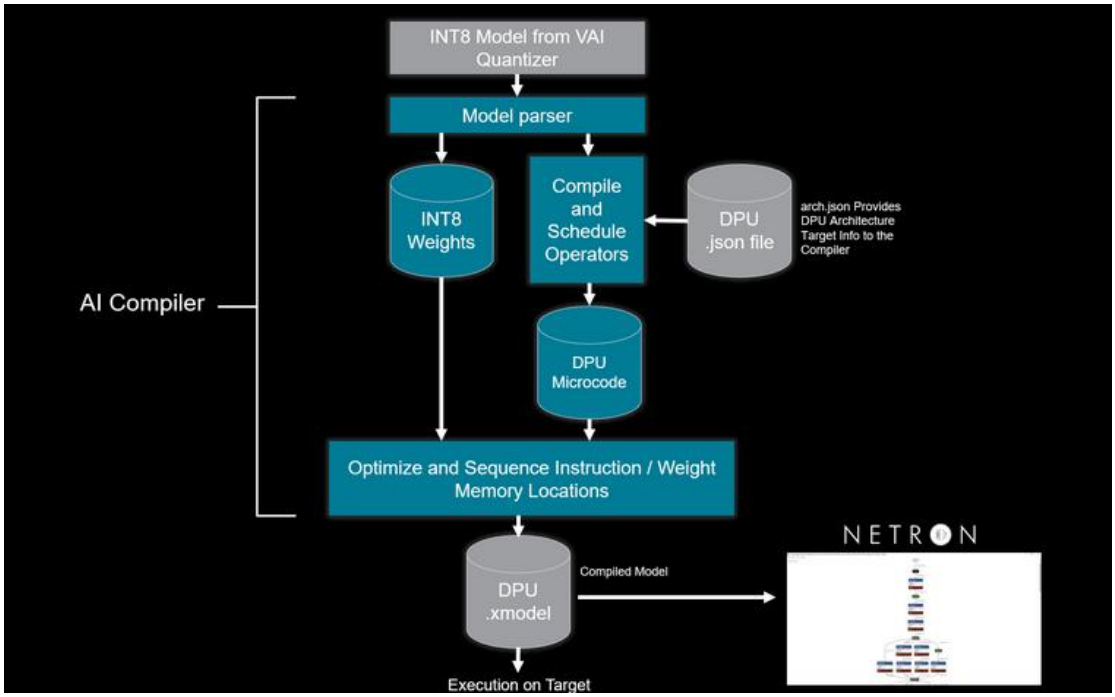

Figure 77: Vitis AI Compiler workflow.

Following the creation of the graph, the compiler performs optimizations to the graph and splits it, if necessary, into smaller subgraphs based on whether the operators of each subgraph can be executed on the DPU. Then, additional architecture-aware optimizations are applied to each subgraph. For this step an arch.json file should be also fed as input. This file contains information about the DPU architecture and the instruction set that the model should be compiled to. Table 10 shows DPUs for different hardware platforms for Xilinx's devices.

DPU Name	Hardware Platform
DPUCZDX8G	Zynq® UltraScale+™ MPSoC
DPUCVDX8H	Versal ACAP VCK5000 evaluation kit
DPUCVDX8G	Versal® ACAP VCK190 evaluation board, Versal AI Core Series
DPUCV2DX8G	Versal® ACAP VEK280 evaluation board, Versal AI Core Series

Table 10: DPUs on different hardware platforms.

The inclusion of the correct arch.json matters a lot as runtime errors will occur if the model is not compiled for the correct DPU architecture. For our thesis, we will utilize a ZCU104 FPGA board from Zynq® UltraScale+™ MPSoC family [29]. As shown in table 10 this board disposes a DPUCZDX8G DPU. The DPUCZDX8G module can be configured in different architectures, in terms of size and resource allocation. These architectures are B512, B800, B1024, B1152, B1600, B2304, B3136 and B4096. The nomenclature indicates the number of computations executed per DPU clock cycle. Figure 78 demonstrates how DPU names are formed for each Hardware platform:

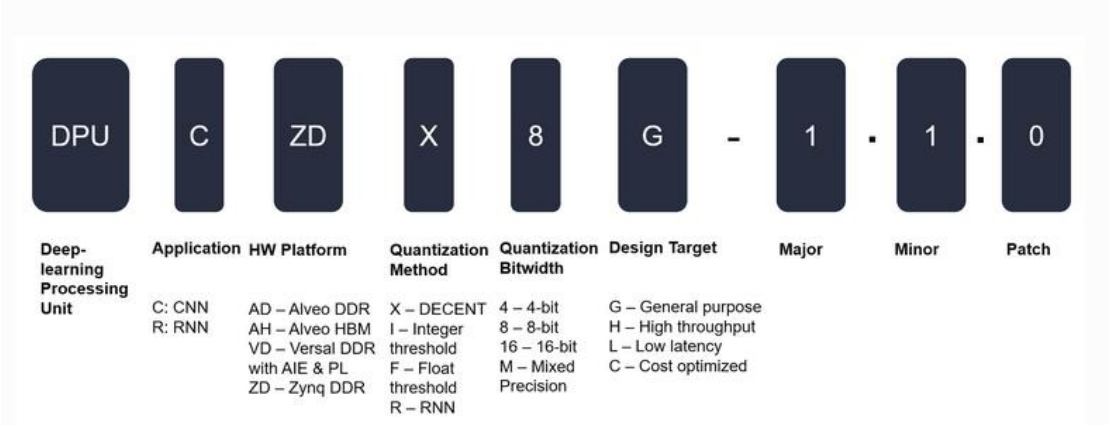

Figure 78: DPU name formation.

So, to compile our model for our ZCU104 board we should feed as input the arch.json file shown in figure 79. We choose the B4096 architecture as it offers peak performance so it is expected to be the fastest.

Figure 79: arch.json for ZCU104 board.

The compilation process finishes with the vitis ai compiler generating an instruction stream for each DPU subgraph and attaching it to each subgraph. Finally, the optimized graph with all the necessary information and instructions is serialized into a compiled xmodel file. Figure 80 shows the command used for model's compilation:

```
vai_c_xir -x quantize_result/DetectMultiBackend_int.xmodel -a quantize_result/arch.json -o yolov5n_cd_pt -n yolov5n_cd_pt
```

Figure 80: Compilation command.

```
**************************************************
* VITIS_AI Compilation - Xilinx Inc.
**************************************************
[UNILOG][INFO] Compile mode: dpu
[UNILOG][INFO] Debug mode: null
[UNILOG][INFO] Target architecture: DPUCZDX8G_ISA1_B4096
[UNILOG][INFO] Graph name: DetectMultiBackend, with op num: 448
[UNILOG][INFO] Begin to compile...
[UNILOG][INFO] Total device subgraph number 5, DPU subgraph number 1
[UNILOG][INFO] Compile done.
[UNILOG][INFO] The meta json is saved to "/workspace/yolov5/yolov5n_cd_pt/meta.json"
[UNILOG][INFO] The compiled xmodel is saved to "/workspace/yolov5/yolov5n_cd_pt/yolov5n_cd_pt.xmodel"
[UNILOG][INFO] The compiled xmodel's md5sum is d1a8d38851df2034a2fcd37b9b0e44d7, and has been saved to "/workspace/yolov5/yolov5n_cd_pt/md5sum.txt"
```

Figure 81: Compilation results.

We can create a png or svg file containing the entire graph with the following command:

```
xir png yolov5n_cd_pt/yolov5n_cd_pt.xmodel graph.png
```

Figure 82: Saving graph in a png image.

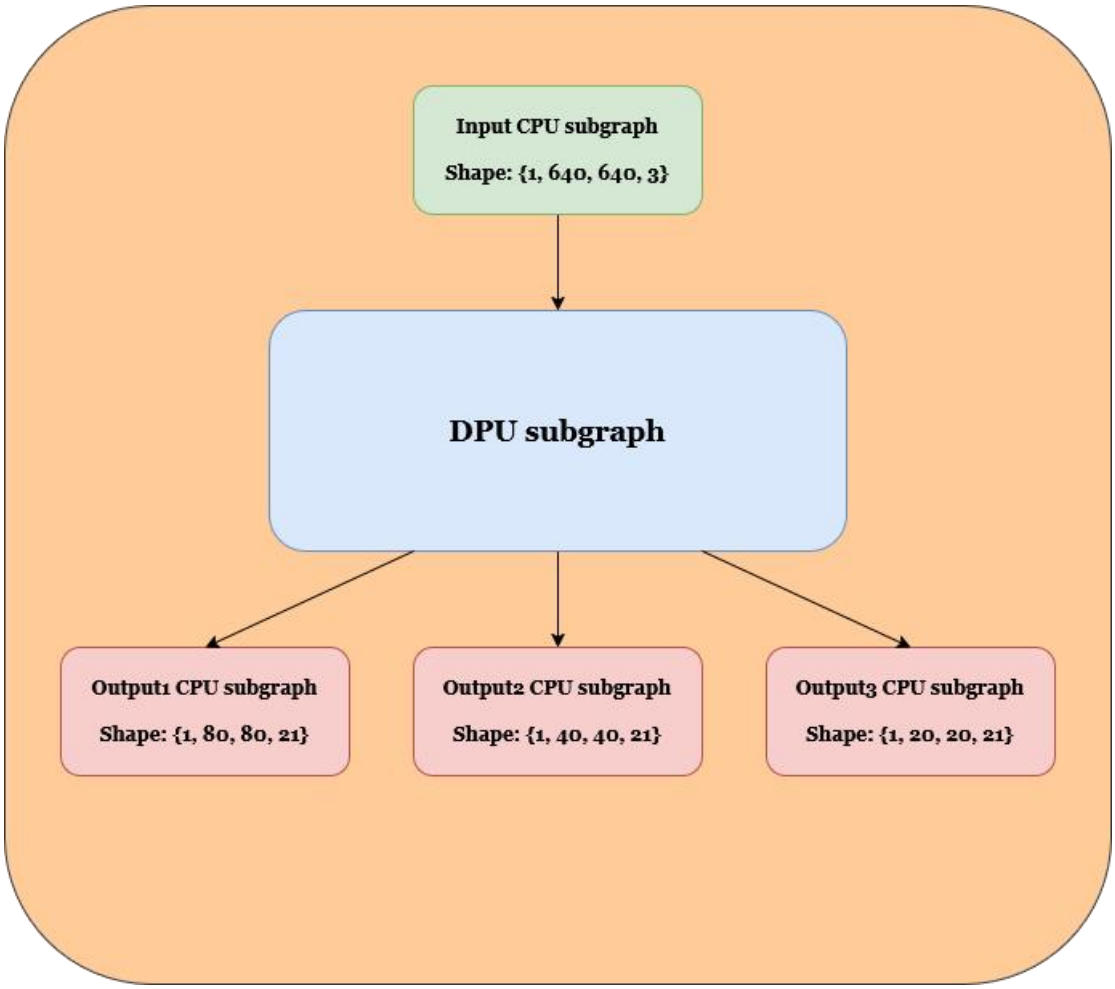

Figure 83: Summary of graph from XIR.

From the graph we can see that the xmodel includes 4 CPU subgraphs which will be run on the embedded ARM processor of the ZCU104 FPGA and one big DPU subgraph that will be run on the DPU. From the 4 CPU vectors one corresponds to the input vector and the other three correspond to the output vectors of the three detection heads.

Lastly, the original size of the float model was **3.7 MB**. The size of the compiled model is **2.5 MB**. The original model experienced a **32.4%** reduction in size.

5.5 Execution on DPU

To run the compiled model on the FPGA we need to utilize VART from Vitis AI. VART is a set of API functions that support the integration of the DPU into software applications. VART will help us process the compiled model, deploy it on the DPU and then run it. Our board is already set with the appropriate SD card image, which includes VART and the DPU architecture we discussed above.

VART APIs also gives us the opportunity to perform some necessary actions on software level before running the compiled model:

- Deserialization of the compiled xmodel and extraction of all availlable subgraphs, both for CPU and for DPU.

- Creation of runner instances. Each DPU subgraph will be assigned to a runner instance which is the entity that can run on the DPU. All the other CPU subgraphs will run on the embedded ARM processor of the FPGA.

- Initialization of input/output buffers of the DPU.

- Pre-processing of input data to the appropriate form. This includes normalization and transformation of pixels to INT8 type, as this is the data form the DPU accepts.

- Execution of runner instances on DPU.

- Post processing of results.

To ensure safety when using the FPGA we install miniconda and create a custom environment. Inside the custom conda environment we download Python and a modified version of requirements.txt file. We run inference on the DPU with **yolov5_quantization/dpu_configuration.py** using the following command:

```
python3 dpu_configuration.py --model_dir yolov5n_cd_pt/yolov5n_cd_pt.xmodel --val_data_dir data/data/images/val --threads 1
```

Figure 84: Command for execution on DPU.

	Class	Images	Instances	P	R	mAP50	mAP50-95
DPU	all	899	2785	**0.756**	**0.696**	**0.751**	**0.47**
	dirt	899	173	0.801	0.763	0.814	0.589
	damage	899	2612	0.711	0.63	0.688	0.35

Table 11: Accuracy on the DPU.

	Images per Device	Pre-Process(ms)	Inference(ms)	NMS(ms)
1 DPU	899	20.77	**7.94**	3.5

Table 12: Performance on the DPU.

The DPU achieves a mAP of 0.751 which is slightly lower than the mAP the quantized model achieved. Again, the reason for that must be in the difference between the pytorch format and the xmodel format. In any case, the difference in accuracy is considered acceptable due to the difference in precision of INT8 and FLOAT32 types.

The inference speed was faster than the speed achieved by the pytorch model on the CPU, about 4 times faster, and 3 times faster from the inference speed achieved by the onnx and openvino formats. The entire execution of the python script took 91 seconds. Overall, we got a decent acceleration.

The image loaded on the ZCU104 hardware device has the ability to be configured to run two DPUCZDX8G(B4096) DPUs simultaneously. Which means we can split the validation dataset in half and assign each part to a different DPU. This enables parallelism in processing and can offer a dramatic decrease in total execution time. We can run inference on two DPUs using the same script by only changing the

--thread parameter from 1 to 2:

```
python3 dpu_configuration.py --model_dir yolov5n_cd_pt/yolov5n_cd_pt.xmodel --val_data_dir data/data/images/val --threads 2
```

Figure 85: Command for execution on two DPUs.

	Images per Device	Pre-Process(ms)	Inference(ms)	NMS(ms)
2 DPUs	450	21.49	**8.2**	4.41
	449	21.58	**10.57**	5

Table 13: Performance on 2 DPU modules.

When we configure the FPGA with 2 DPUs it seems that one of them experiences delays in pre-process, inference and NMS times. The reason for that arises from the fact that the Scheduler of the processing unit has to schedule inference across two DPUs compared than before. Scheduling inference on more than 1 DPUs involves transporting the data from RAM to the on-chip memory of each DPU, where the DPU's input buffers are located, and vice versa alternately. Furthermore, total execution time was reduced from 91 seconds to 65 seconds, proving that parallel procedure of the dataset speeds up the entire procedure even more.

```
Compute Units
  PL Compute Units
    Index   Name                                Base_Address    Usage   Status
    0       DPUCZDX8G:DPUCZDX8G_2               0x80001000      1347    (IDLE)
    1       DPUCZDX8G:DPUCZDX8G_1               0x80000000      12889   (IDLE)
```

Figure 86: Allocation of images on the two DPU units.

6. Inference on Raspberry Pi

In this section of our diploma thesis we will study the performance of our algorithm when running on a Raspberry Pi 3[31]. Raspberry Pi is a small portable computer mainly used for IoT applications.

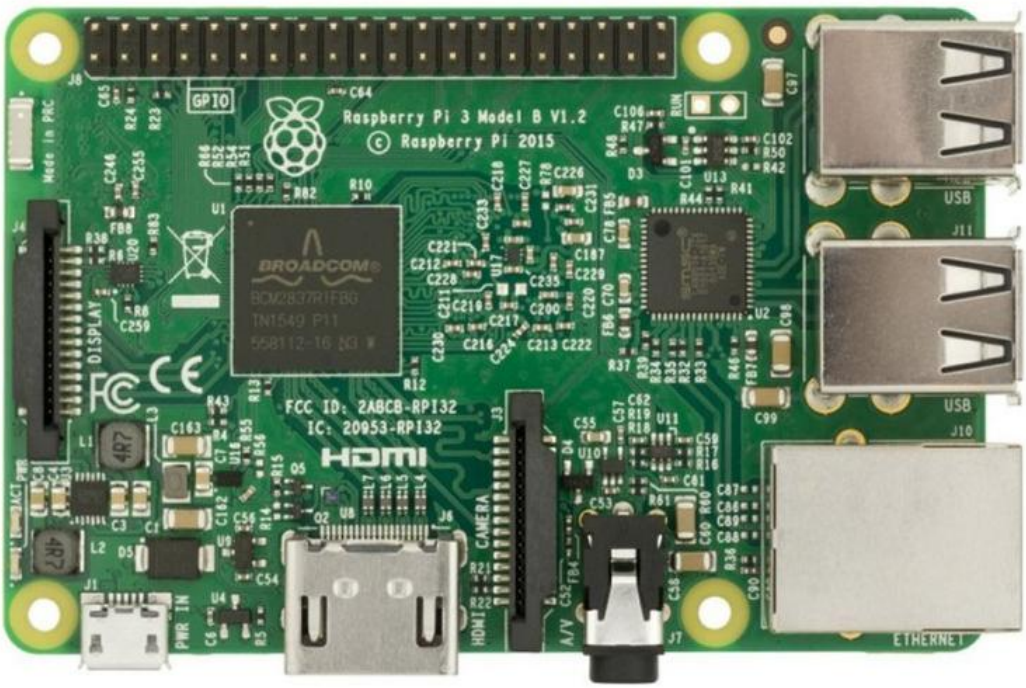

Figure 88: Raspberry Pi 3.

We connect to the Raspberry Pi with **ssh** and then move in it the original YOLOv5 API, with the LeakyReLU modification, along with the validation dataset. We also set a miniconda environment where we download python and the requirements.txt file. We run the validation dataset using the same command from figure 52. The results for the pytorch and the onnx formats can be seen in the table below:

	Pre-Process(ms)	Inference(ms)	NMS(ms)
Raspberry pytorch format	6.8	**887.5**	4.8
Raspberry onnx format	9.74	**633.6**	7.46

Table 14: Inference on Raspberry Pi for batch-size 1.

This Raspberry Pi disposes a quad-core 64-bit Broadcom BCM2837 ARM Cortex-A53 SoC processor running at 1.2 GHz, so it is not expected to reach the performance of the FPGA. A huge increase in Inference time is observed proving that the device struggles in handling a complex CNN architecture like YOLOv5. However, what comes out the most is the Pre-process time. In comparison with the FPGA one, the pre-processing stage is 3 times faster. Since, both the ZCU104 and Raspberry have similar processors, the reason for that increase must be relevant with RAM's speed. The Raspberry has a faster RAM and also the FPGA spends time in assigning the data from RAM to the on-chip memory of the DPU. Nevertheless, the FPGA outperforms the Raspberry Pi by far, as it is expected.

7. Experimental Results

In this section, we gather all the results from the previous chapters to make a final assessment regarding the performance of our developed algorithm. Table 15 provides us the outcomes from all inferences in every format.

	mAP50	mAP50-95	Inference(ms)	Average Time for Images(seconds)
CPU pytorch format	**0.83**	**0.563**	33	30.566
CPU onnx and openvino formats	0.785	0.493	23.2	22.924
Raspberry pytorch format	**0.83**	**0.563**	887.5	808.3
Raspberry onnx and openvino formats	0.785	0.493	633.6	585.1
1 DPU	0.751	0.47	7.94	28.96
2 DPU	0.751	0.47	10.57	16.71
TESLA T4	**0.83**	**0.563**	6.8	8.00
NVIDIA L4	**0.83**	**0.563**	**6.5**	**7.46**

Table 15: Summarization of results.

The Average Time for Images is computed by multiplying the number of images from the validation dataset(899) with the sum of pre-process, inference and NMS times, for each hardware device. With this, we get an estimate of how much time is needed for all the images to be processed by our trained model.

The results prove, that the original format of the model, the pytorch one, is achieving the best possible accuracy. This outcome probably comes from the fact the entire YOLOv5 API, and as such the YOLOv5 architecture, is written in pytorch. The best Inference and Average Times are achieved by the NVIDIA L4 which is the most specialized device in matrix multiplications. The DPU achieves an inference time 4 and 3 times faster than the CPU's pytorch and onnx formats respectively, proving that we fulfilled a decent acceleration. When the FPGA is configured with two DPUs, Average Time almost becomes half proving that the FPGA can offer parallel processing of images in the validation dataset. We also confirmed this by measuring the total execution times(91 and 65 seconds) for the script in 5.5. Additionally, when using 2 DPUs we receive a slight increase in pre-process, inference and NMS times per image, for one of the two DPUs, due to delays arising from the Scheduler of the FPGA. Finally, inference on a Raspberry Pi 3 gives a poor inference time per image in contrast with the other hardware devices, but lower pre-process time than the FPGA one.

8. Conclusion - Future Work

Our work for this thesis led to the implementation of wind turbine surface damage detection algorithm utilizing a variation of YOLOv5 architecture, YOLOv5 nano. This algorithm reached an accuracy close enough to the capabilities of an expert in the wind turbines field and proved that Convolutional Neural Networks can closely resemble the way humans detect objects in more specific applications.

The trained model was able to be run on a FPGA, a DPU to be more accurate, after being processed with Vitis AI and achieved a decent acceleration when run on 1 or 2 DPUs.

The next step is, trying to make modifications to the DPUCZDX8G IP Block architecture of ZCU104. If we re-configure the DPU module with another architecture(B512, B800, B1024, B1152, B1600, B2304, B3136) it would be possible to load more than 2 DPUs on our FPGA(3 or 4) and increase the number of images being processed per second. Unfortunately, these architectures are considered slower than the B4096 so it is expected to receive an increase in Inference Time. It is also possible if we choose to use more than 2 DPUs, to get a greater increase in Pre-process, Inference and NMS times per image as the Scheduler will have a much more complex work in assigning inference among the DPUs. Nevertheless, it is never bad to explore and the other options.

Additionally, it worths exploring the option of Accelerating the entire application. Xilinx offers other software tools specialized not only in accelerating the CNN but the entire application. This is method that could result in a huge decrease in pre-process and NMS times.

Ultimately, in 4.4 we mentioned a couple of methods for increasing the accuracy of the trained model by applying some changes to the dataset used for training. These methods included modifications on the original dataset like Data Augmentation techniques, to increase the number of labeled data, and reduction of the number of Backround Images. Also, the creation of a new dataset is recommended, from the beginning, with more images with defects.

BIBLIOGRAPHY

[1] Ashley Foster, Oscar Best, Mario Gianni, Asiya Khan, Keri Collins, Sanjay Sharma, Drone Footage Wind Turbine Surface Damage Detection, School of Engineering, Computing and Mathematics(SeCAM), University of Plymouth, UK, URL: https://ieeexplore.ieee.org/document/9816220

[2] Ashley Foster and Oscar Best and Mario Gianni and Asiya Khan and Keri Collins and Sanjay Sharma, "YOLO Annotated Wind Turbine Surface Damage," yolo-annotated-wind-turbines-586x371, 2021, URL: https://www.kaggle.com/datasets/ajifoster3/

[3] IBM, Deep Blue, URL: https://www.ibm.com/history/deep-blue

[4] IBM, Machine Learning, URL: https://www.ibm.com/topics/machine-learning

[5] Microsoft, Azure, "What is Computer Vision?", URL: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-computer-vision#object-classification

[6] MIT Sloan, Machine Learning Explained, URL: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained

[7] Viso AI, Deep Learning, URL: https://viso.ai/deep-learning/what-is-deep-learning/

[8] MICHAEL NEGNEVITSKY, Artificial Intelligence, A Guide to Intelligent Systems, (Second Edition), Chapter 6 Artificial Neural Networks, ADDISON WESLEY, URL: http://www.academia.dk/BiologiskAntropologi/Epidemiologi/DataMining/Artificial_Intelligence-A_Guide_to_Intelligent_Systems.pdf

[9] Amazon, “What is a Neural Network?”, URL: https://aws.amazon.com/what-is/neural-network/

[10] IBM, “What is a Neural Network?”, URL: https://www.ibm.com/think/topics/neural-networks

[11] IBM, “What is Loss function?”, URL: https://www.ibm.com/think/topics/loss-function

[12] Geeks for Geeks, Introduction to Convolution Neural Networks, URL: https://www.geeksforgeeks.org/introduction-convolution-neural-network/

[13] Joseph Redmon , Santosh Divvala, Ross Girshick, Ali Farhadi, You Only Look Once: Unified, Real-Time Object Detection, University of Washington, Allen Institute for AI, Facebook AI Research, URL: https://ieeexplore.ieee.org/document/7780460

[14] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. International Journal of Computer Vision, 111(1):98–136, Jan. 2015, URL: https://link.springer.com/article/10.1007/s11263-014-0733-5

[15] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV), 2015, URL: https://arxiv.org/abs/1409.0575 .

[16] Medium, Aqeel Anwar, What is Average Precision in Object Detection & Localization Algorithms and how to calculate it?, URL: https://towardsdatascience.com/what-is-average-precision-in-object-detection-localization-algorithms-and-how-to-calculate-it-3f330efe697b

[17] Kili Technology, Mean Average Precision(mAP): A Complete Guide, URL: https://kili-technology.com/data-labeling/machine-learning/mean-average-precision-map-a-complete-guide

[18] Haiying Liu, Fengqian Sun, Jason Gu and Lixia Deng, SF-YOLOv5: A Lightweight Small Object Detection Algorithm Based on Improved Feature Fusion Mode, Sensors 2022, 22, 5817, URL: https://www.mdpi.com/1424-8220/22/15/5817

[19] Ultralytics, A Comprehensive Guide to Ultralytics YOLOv5, URL: https://docs.ultralytics.com/yolov5/

[20] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollar, “Microsoft coco: ´ Common objects in context,” 2015, URL: https://arxiv.org/abs/1405.0312

[21] A. Shihavuddin and X. Chen, “Dtu - drone inspection images of wind turbine,” 2018, URL: https://data.mendeley.com/datasets/hd96prn3nc/2

[22] P. Skalski, “Make Sense”, 2019, URL: https://github.com/SkalskiP/make-sense

[23] Google, Colab, URL: https://colab.google/

[24] Arm, “What is a FPGA?”, URL: https://www.arm.com/glossary/fpga

[25] Docker Docs, Docker Desktop, URL: https://docs.docker.com/desktop/

[26] AMD, Vitis AI, Documentation 3.0, URL: https://xilinx.github.io/Vitis-AI/3.0/html/index.html#

[27] AMD, Technical Information Portal, Vitis AI User Guide (UG1414) 3.0, URL: https://docs.amd.com/r/3.0-English/ug1414-vitis-ai/Vitis-AI-Overview

[28] Hackster IO, YOLOv5 Quantization & Compilation with Vitis AI 3.0 for Kria, URL: https://www.hackster.io/LogicTronix/yolov5-quantization-compilation-with-vitis-ai-3-0-for-kria-7b005d#toc-quantizing-yolov5-pytorch-with-vitis-ai-3-0-5

[29] AMD, Xilinx, Zynq Ultrascale+ MPSoC ZCU104 Evaluation Kit, URL: https://www.xilinx.com/products/boards-and-kits/zcu104.html

[30] Xilinx, Vitis AI Tutorials [Github Repository], URL: https://github.com/Xilinx/Vitis-AI-Tutorials

[31] Raspberry Pi, URL: https://www.raspberrypi.com/