UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Allan Henrique Kamimura

Study and Implementation of an Embedded Image Processing System in Autonomous Inspection Robots

São Carlos

2024

Allan Henrique Kamimura

Study and Implementation of an Embedded Image Processing System in Autonomous Inspection Robots

Monografia apresentada ao Curso de Engenharia Aeronáutica, da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro Aeronáutico.

Advisor: Prof. Dr. Glauco Augusto de Paula Caurin

São Carlos

2024

Catalog card prepared by Patron Service at "Prof. Dr. Sergio Rodrigues Fontes" Library at EESC/USP

```
        Kamimura, Allan Henrique
K15s       Study and implementation of an embedded image
        processing system in autonomous inspection robots / Allan
        Henrique Kamimura ; Supervisor Glauco Augusto de Paula
        Caurin. -- São Carlos, 2024.

            Monograph Bachelor Final Thesis (Undergraduate in
        Aeronautics Engineering) -- São Carlos School of
        Engineering, at University of São Paulo, 2024.


           1. Computer vision. 2. Embedded systems.
        3. Object detection. 4. Depth estimation. 5. Real time
        processing. 6. Edge computing. 7. Robotics automation.
        8. Autonomous navigation. I. Title.
```

Elena Luzia Palloni Gonçalves – CRB 8/4464

FOLHA DE APROVAÇÃO
Approval sheet

Candidato / *Student*: Allan Henrique Kamimura	
Título do TCC / *Title* **:** Study and Implementation of an Embedded Image Processing System in Autonomous Inspection Robots	
Data de defesa / *Date*: 19/12/2024	

Comissão Julgadora / *Examining committee*	**Resultado / *result***
Professor Titular Glauco Augusto de Paula Caurin Instituição / Affiliation: EESC - SAA	Aprovado
Henrique Borges Garcia Instituição / Affiliation: TORADEX	Aprovado
Tiago Fontes de Oliva Costa Instituição / Affiliation: EMBRAER	Aprovado

Presidente da Banca / Chair of the Examining Committee:

[illegible]

Professor Titular Glauco Augusto de Paula Caurin
(assinatura / *signature*)

To my mother
whose faith never faltered,
even when mine did ....

ACKNOWLEDGEMENTS

Thanks to my advisor, Professor Glauco Augusto de Paula Caurin, for providing me with a direction during my internship and later on my graduation thesis. Your insights and encouragement have been invaluable in shaping my academic journey. Thank you for inspiring me to write this work.

I also extend my gratitude to Toradex, the company where I had the privilege of working as an intern, for providing the hardware and resources. Your support during my internship made it possible to explore and implement the ideas presented in this thesis and I am sincerely thankful for the opportunity to work with such a cutting-edge company.

Lastly, I want to thank Mr. Laurence Moroney, The "AI Guy" who inspired me to take the first steps into deep learning and TensorFlow. Thanks for making AI accessible for everyone, with easy to digest MOOC lessons and interesting code examples.

“What we observe is not nature in itself but nature exposed to our method of questioning.”
Werner Heisenberg

“The higher we soar, the smaller we appear to those who cannot fly.”
Friedrich Nietzsche

“Toda a sabedoria humana não vale um par de botas curtas.”
Machado de Assis

ABSTRACT

Kamimura, A. H. **Study and Implementation of an Embedded Image Processing System in Autonomous Inspection Robots**. 2024. 54 p. Monograph (Conclusion Course Paper) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2024.

This study presents an end-to-end analysis of computer vision models applied to embedded scenarios that can be used for further research. Includes a brief background on commonly used models and comparison metrics. One of the main challenges in deploying such models is that in most cases they are not intended for embedded devices, as they are optimized for accuracy with no concern for runtime speed, memory or model complexity. To address this issue, the trade-off of using strategies such as model quantization and prediction delegates is explored in detail. As a result, we obtain a hybrid object detection and monocular depth estimation solution using a mobilenet-v2 backbone with an inference time of 0.019 s per frame, capable of real-time processing, running on the verdin i.MX8M Plus NPU.
Code is available at: https://github.com/AllanKamimura/ComputerVisionPipelines

Keywords: Computer Vision, Embedded Systems, Object Detection, Depth Estimation, Real-Time Processing, Edge Computing, Robotics Automation, Autonomous Navigation

RESUMO

Kamimura, A. H. **Estudo e implementação de sistema embarcado de processamento de imagens em robôs autônomos de inspeção**. 2024. 54 p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2024.

Este estudo apresenta uma análise abrangente de modelos de visão computacional aplicados a cenários embarcados, que podem ser utilizados para pesquisas futuras. Inclui um breve contexto com os modelos mais comumente utilizados e métricas de comparação. Um dos principais desafios na implementação de tais modelos é que, na maioria dos casos, eles não são projetados para dispositivos embarcados, pois são otimizados para precisão, sem preocupação com a velocidade de execução, memória ou complexidade do modelo. Para abordar essa questão, o estudo explora em detalhes o equilíbrio entre as estratégias de quantização de modelos e delegação de predição. Como resultado, obtemos uma solução híbrida de detecção de objetos e estimativa de profundidade monocular utilizando uma arquitetura backbone mobilenet-v2, com um tempo de inferência de 0,008 segundos por quadro, capaz de processamento em tempo real, operando na NPU do Verdin i.MX8M Plus. Código disponível em: https://github.com/AllanKamimura/ComputerVisionPipelines

Palavras-chave: Visão Computacional, Sistemas Embarcados, Detecção de Objetos, Estimativa de Profundidade, Processamento em Tempo Real, Computação de Borda, Automação Robótica, Navegação Autônoma

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

UAV	Unmanned Aerial Vehicles
AI	Artificial Inteligence
YOLO	You Only Look Once
SSD	Single Shot MultiBox Detector
OS	Operational System
GPU	Graphical Processing Unit
CPU	Central Processing Unit
NPU	Neural Processing Unit
TPU	Tensor Processing Unit
FPGA	Field-Programmable Gate Array
IoU	Intersection Over Union
RMSE	Root Mean Squared Error
mAP	Mean Average Precision
FlOps	Floating Point Operations
MAC	Multiply-Accumulate Operations
FPS	Frames per Second
LIDAR	Light Detection and Ranging
CNN	Convolutional Neural Network
NMS	Non Max Suppression
RGB	Red, Green, Blue
BGR	Blue, Green, Red

CONTENTS

1 INTRODUCTION

This chapter presents the motivation and objectives of this study, highlighting the relevance of the topic discussed and providing the foundation for the theoretical framework and experimental investigations discussed in the subsequent chapters.

1.1 Motivation

Recent developments in the fields of embedded systems and AI has led to significant breakthroughs in robotics automation, particularly in sectors requiring high-level computer vision tasks, such as infrastructure maintenance, quality control, and agriculture monitoring. Autonomous inspection robots equipped with advanced sensing capabilities have become increasingly valuable for performing repetitive, hazardous, or other repetitive tasks that challenge human operators. Some examples include autonomous UAVs used for power line inspections, border control, and crop analysis; mobile ground robots that assess industrial machinery or inspect construction sites; and underwater robots that monitor submerged infrastructure such as oil rigs and pipelines. The versatility of these robots relies heavily on their capacity to interpret their surroundings accurately and on their mobility to navigate the environment in real-time.

While algorithms for object detection and depth estimation are well-established in the industry they are frequently deployed on standard computing platforms. However, implementing them on embedded devices presents unique challenges because unlike conventional PCs, which offer virtually unlimited computational power, memory space, high-end GPUs, and full access to cloud resources; embedded devices are typically compact, lightweight, and optimized for energy efficiency. This constrained design makes them ideal for mobile and autonomous systems like inspection robots but imposes significant limitations on the development of real time applications that rely on computation-heavy models.

Moreover, embedded devices are designed to work autonomously in real-world environments where consistent internet connectivity may not be available, so they cannot offload high-processing tasks to the cloud. Furthermore, to lessen the down-time spent on battery recharging on swaps, energy efficiency should also be a key aspect. As a result, models that are easily deployed on a PC require significant modification to run on embedded devices, including quantization, pruning, and optimization techniques aimed at reducing model size and computational requirements.

1.2 Objectives

The aim of this thesis is to assess the feasibility of implementing object detection and depth estimation algorithms on embedded ARM-based systems, which are often used in low-power autonomous devices like drones and small robots. The end goal is to achieve real-time inference for these computer vision tasks on constrained hardware, by applying optimization techniques such as model quantization and reduction. By ensuring these models can operate within the limited processing power and energy budgets of embedded devices, this research will make complex perception capabilities accessible for autonomous inspection applications.

Additionally, to evaluate the system's practical application, a second objective is to test the embedded solution in a real-life inspection scenario with a camera mounted on top of a robot to capture its surroundings. This setup will simulate operational conditions, such as varying lighting, distance to objects, and complex backgrounds, that are typical in industrial or field inspection tasks. The aim is to confirm the system's suitability for deployment in operational inspection tasks where autonomous decision-making is essential.

1.3 Chapters Overview

The structure of this thesis is as follows: first, the theory behind object detection and depth estimation models and a review of related work; then, a detailed explanation of the adaptation process for embedded systems, covering model selection, optimization strategies, and the implementation. Finally, the proposed solution will be validated through experiments in simulated inspection scenarios, assessing the trade-offs between performance, accuracy, and computational efficiency. The results aim to advance the feasibility of deploying complex AI models on embedded devices, making autonomous inspection robots more effective and resilient in real-world applications.

2 RELATED WORK

Early work on embedded computer vision used handcrafted *FPGA* and classical algorithms to achieve near real-time performance in tasks such as Stereo Vision (Honegger; al., 2014), Background Subtraction (Khan; al., 2016), and Object Detection (Zhao; al., 2016). More recently, significant advances have been made by leveraging the use of more widely available hardware, such as embedded *GPUs* (Otterness, 2017; Wofk *et al.*, 2019), mobile *GPUs* (Howard; al., 2017; Yang *et al.*, 2018), and *TPUs* (Sun; al., 2021; Mohammadi; al., 2023; Alqahtani1; al., 2024). These works explore some of the adaptations required to run neural networks on embedded devices; however, they do not include the use of *NPUs*.

The study by (Khalili; al., 2024) proposes the use of an object detection model to plan the path of the robot manipulator. While this brings focus to real-world application development, it lacks the trade-off analysis of different hardware and algorithms. On the other hand, (Alqahtani1; al., 2024) provides thorough benchmarking, but without focusing on real-world applications. With this in mind, the approach of this work is to find a middle ground.

3 METHODOLOGY

This chapter describes the approach and techniques employed to conduct this study. It begins with the initial selection of devices and their setup for testing, followed by an explanation of the models used and the metrics applied to evaluate them. Finally, it details how the comparison results were analysed to draw meaningful conclusions.

3.1 Hardware

For the choice of hardware to be tested and compared, the criteria were to find a group of devices that are readily available in the market, affordable, and with close release dates. Table 1 shows the overall characteristics of each device.

Table 1 – Comparison of Embedded Devices

Device	**Release Year**	**Price (USD)**	**RAM**	**Camera**
Raspberry Pi 4	2020	55	4 GB	PiCamera 5MP
Verdin i.MX8M Plus	2020	300	4 GB	AR0521 CSI 5MP
NVidia Jetson Nano	2019	99	4 GB	AC340 Warrior 1080p

During this market research, another group of devices showed great potential, but was discarded because it was either too expensive for the scope of this work or was not readily available. However, this can be used for reference for future works.

Table 2 – Comparison of Embedded Devices for Future Works

Device	**Release Year**	**Price (USD)**	**RAM (GB)**
Raspberry Pi 5 + AI Kit	2024	150	8
Aquila AM69	2024	1100	32
NVIDIA Jetson Orin Nano Super	2023	249	8

3.2 Setup and Configuration

Table 3 shows the basic setup for each device. The reason for *NVidia Jetson* using a different runtime is that *NVidia* don't support *LiteRT* nor *TFLite*. Also, using their proprietary runtime *TensorRT* requires changes to the underlying structure of the model, which, in turn, would require the models to be re-evaluated.

Table 3 – Embedded Devices with Base OS and Runtime Information

Device	**Base OS**	**Linux Kernel**	**Runtime**
Raspberry Pi 4	Raspbian OS Bookworm	6.6	ai-edge-litert-
Verdin i.MX8M Plus	Torizon OS 6.8.0	5.15	ai-edge-litert
NVidia Jetson Nano	JetPack 4.6.1	4.9	tensorflow

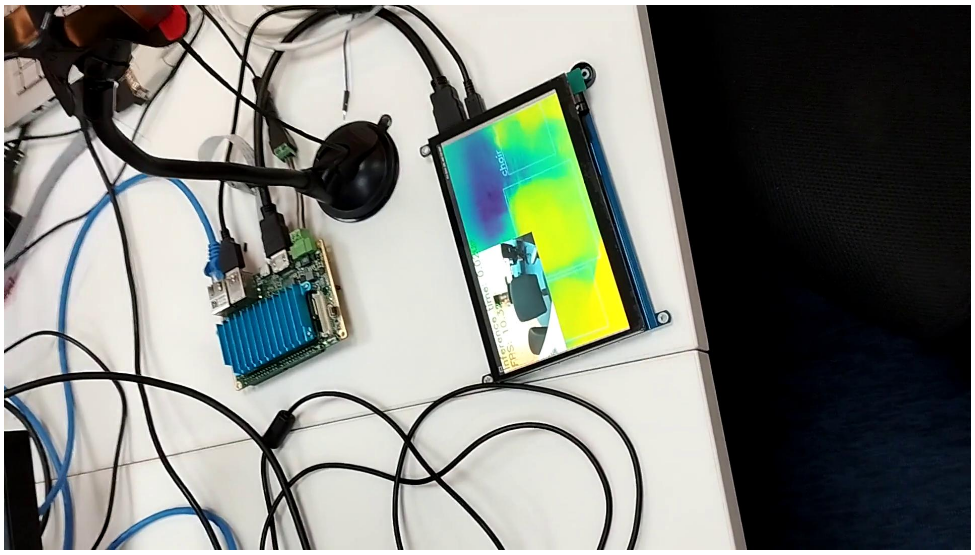

Figure 1 – Verdin i.MX8M Plus setup

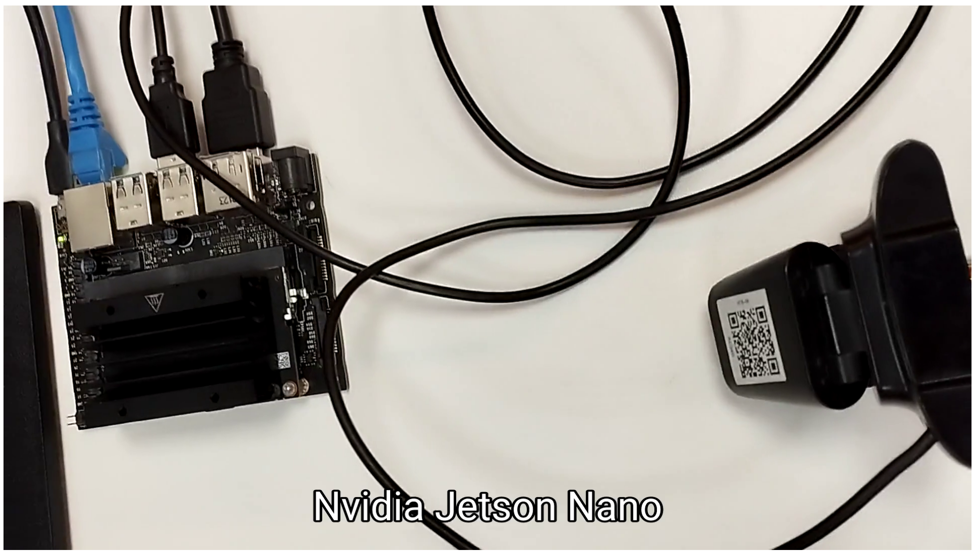

Figure 2 – NVidia Jetson Nano setup

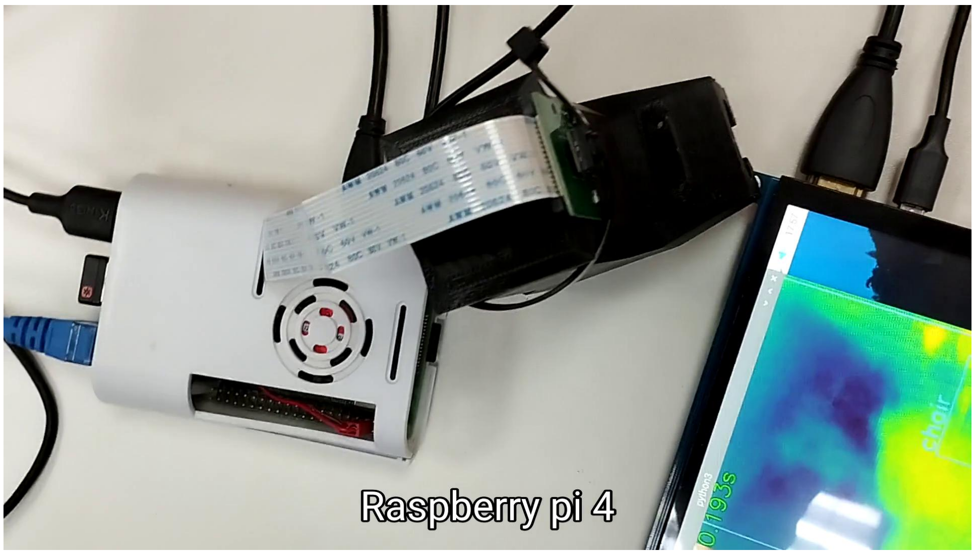

Figure 3 – Raspberry Pi 4 setup

The image processing pipeline uses *GStreamer*[1] to capture the images, process it and display the results on a screen. The image pre/post processing are handled by openCV [2] library. For the inference runtime, a custom build of LiteRT[3] and of TensorFlow [4] are used. Figure 4 summarizes this pipeline.

It can be divided into 2 parts: the first part is a queue that accumulates frames from the camera and transforms the raw pixels into images and serves it to an appsink. When the new frame signal is triggered, the image goes through the prediction pipeline, and the resulting post-processed output is shown on a display with a waylandsink.

[1] https://gstreamer.freedesktop.org
[2] https://opencv.org/
[3] https://github.com/AllanKamimura/LiteRT/blob/main/gen/litert_pip/python3/dist/ai_-edge_litert_nightly-1.0.1.dev20241024-cp311-cp311-linux_aarch64.whl
[4] https://developer.download.nvidia.com/compute/redist/jp/v461/tensorflow/

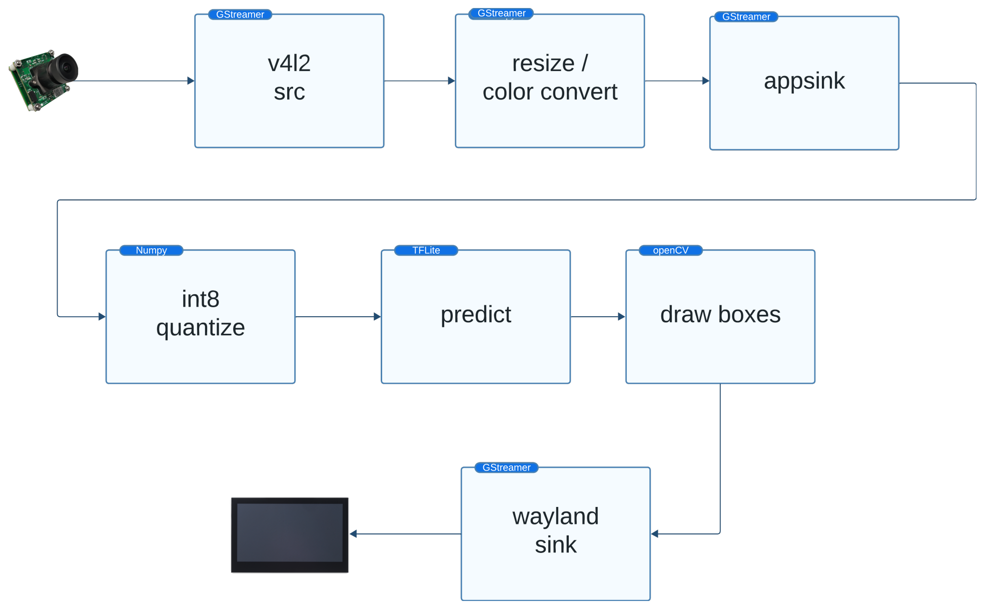

Figure 4 – Data Pipeline (Source: Author)

The benchmarks are taken under good illumination and with a batch size of 1. After a 1 minute warm-up with the model running, the time is measured as the average of 400 predictions.

The complete scripts related to this benchmark can be found at the author's *GitHub* repository **Computer Vision Pipelines**[5]

3.3 Metrics and Models Descriptors

When comparing two models or hardware, it's common to use a set of metrics and descriptors. Some authors are ambiguous about what they are measuring, most notably the FPS and inference FPS. This section tries to bring some clarity to this underlying difference as well as give a short background about other commonly used metrics.

- **Inference Time:** This measures how long it takes for the model to run inference at a single frame input. Lower is better.
- **Inference FPS:** The mathematical reciprocal of the inference time. It measures how many frames the model can process in a single second. Higher is better.
- **Latency:** Also referred to as total time, this is a more practical time measurement, as it takes into account the entire end-to-end workflow: camera capture, image conversion, image pre-processing, the inference, post-processing, and image display. Lower is better.

[5] https://github.com/AllanKamimura/ComputerVisionPipelines

- **FPS:** The reciprocal of the latency. The actual number of frames we are going to see on the screen. Higher is better.
- **MAC:** It measures the computational complexity of a model. It's the counting of multiple accumulated operations and is used to compare model efficiency. After the pruning and quantization steps, this number will be higher than the actual value. Higher is more complex.
- **FlOps:** It measures the computational complexity of a model. It's the counting of floating-point operations and is used to compare model efficiency. After pruning, this number will be higher than the actual value; and after quantization steps, the model will be in integer format. Higher is more complex.
- **Params:** The total count of parameters (weights and biases) in the model. After the pruning and quantization steps, this number will be higher than the actual value. Higher is more complex.
- **Delta1%:** This metric evaluates the accuracy of depth estimation models by measuring the percentage of pixels where the predicted depth is within a specific threshold (commonly 1.25 times) of the ground truth depth. A higher Delta1 percentage indicates that a greater proportion of predictions are "close enough" to the actual depth. Higher is better.
- **RMSE:** This metric is used when the output of the model is an array of continuous values, which is usually the case for depth estimation tasks. [Add mathematical definition]. Lower is better.
- **IoU:** This metric measures the overlap between the predicted output and the ground truth. It is commonly used in segmentation and object detection tasks to evaluate the accuracy of predicted masks or boxes. The IoU score ranges from 0 to 1, where 1 indicates perfect overlap.
- **mAP:** This metric evaluates the performance of object detection models by calculating the average precision for each class and then taking the mean across all classes. It considers various thresholds of IoU. Higher is better.
- **Precision:** This metric measures the proportion of true positive predictions among all positive predictions (both true and false). It is used in classification and object detection tasks. Higher is better.
- **Recall (Sensitivity):** This metric measures the proportion of true positive predictions among all actual positive cases. It indicates how well the model identifies positive cases. Higher is better.

- **Average Precision:** This metric calculates the area under the precision-recall curve for a specific class. It is used in object detection to evaluate performance per class before computing mAP. Higher is better.

- **Input size:** In the context of computer vision, it's the resolution of the image. Usually, more detailed images increase the model's performance at the cost of higher inference time and model complexity.

3.4 Models and Datasets

The datasets are collections of data used to train and validate machine learning models.

The models are quantized in *INT8* format. Details about the quantization process are in the runtime section (6), while details about the models implementation are in section (5.3).

3.4.1 Object Detection

Table 4 – Comparison of Object Detection Models

Dataset	**Model Name**	**Input Size**	**GigaMACS**	**mAP**
COCO	Yolov_11n	640x640	3.241	0.538
COCO	Yolov_5nu	640x640	3.862	0.421
COCO	Yolov_11s	320x320	2.683	0.516
COCO	Yolov_5su	320x320	2.999	0.486
COCO	ssd_mobilenet_v2	300x300	0.728	0.222

- **COCO** (Lin *et al.*, 2014): It's the most widely used dataset for object detection and segmentation. It consists of 2.5 million labeled images classified into 91 classes of common objects. In this work, the COCO2017 is used.

- **ImageNet Large Scale** (Russakovsky *et al.*, 2015): It was the first large-scale standardized dataset of 14,197,122 images labeled into 1,000 classes. Most of the early successful CNN models started from this competition .

3.4.2 Depth Estimation

Table 5 – Comparison of Depth Estimation Models

Dataset	**Model Name**	**Input Size**	**GigaMACs**	**Delta1%**
NYUDepthV2	Fast Depth	224x224	0.3664	77.1
NYUDepthV2	MiDaS-small (v2.1)	192x256	3.446	86.67

- **KITTI-Depth** (Zhou *et al.*, 2018): It's a dataset of 94,000 LIDAR point clouds that were matched with stereo vision depth estimation .
- **NYU-Depth V2** (Silberman *et al.*, 2012): It's a dataset of 2,347 images with depth estimation and semantic segmentation collected using a Kinect .

4 OBJECT DETECTION

4.1 Introduction

This chapter begins with a brief contextualization of object detection. Then it presents a general idea about the SSD and YOLO model architectures. Lastly, it presents the NMS algorithm.

4.2 Context

Object Detection is a computer vision task that involves identifying and locating objects within an image. The localization information is usually a bounding box around the object, while the classification is a predefined class label. Commonly used in applications like surveillance, autonomous vehicles, robotics, and augmented reality, object detection relies on convolutional neural networks (CNNs) to recognize and distinguish objects under various conditions and environments. 5 shows an example:

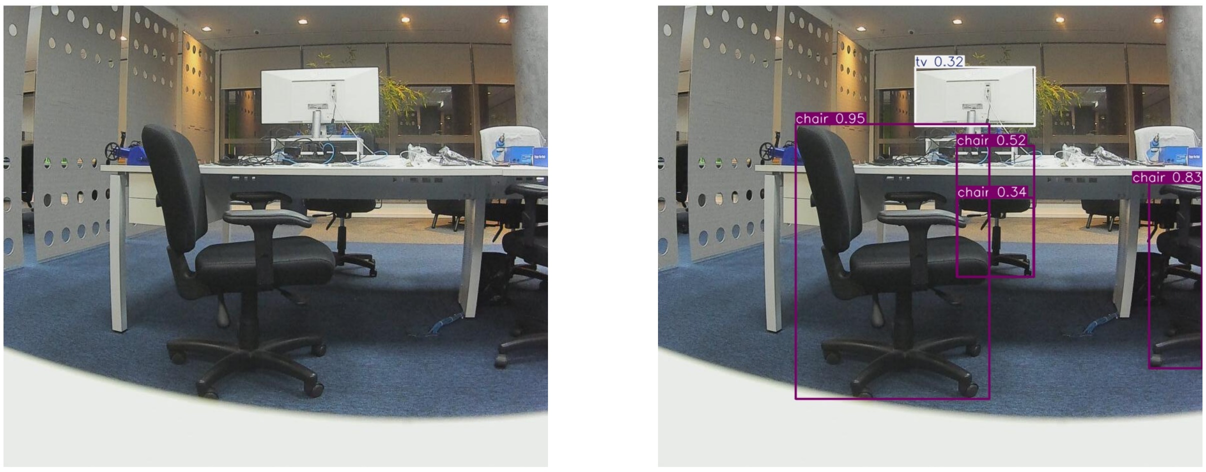

Figure 5 – Comparison of Original and Predicted Images (Source: Author)

The object detection models usually consists of three parts:

- **Backbone Model**: A general-purpose feature extract model with no specific task
- **Bounding Box**: A localization + classification model
- **Box Fusion**: Usually a NMS algorithm

4.3 Implementation Overview

The core idea of the object detection model is to use a cell based grid classification. First it defines a default bounding box centered on each cell and tries to encapsulate the object by offsetting the size of each box. Then the object inside this box is classified into predefined categories.

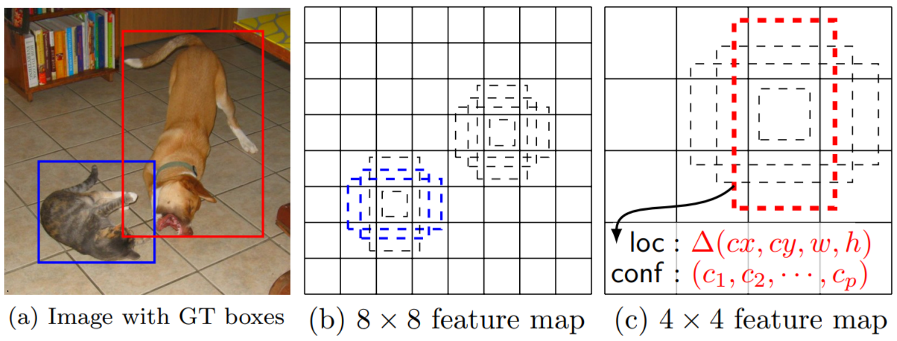
loc : $\Delta(cx, cy, w, h)$
conf : $(c_1, c_2, \cdots, c_p)$
(a) Image with GT boxes
(b) 8×8 feature map
(c) 4×4 feature map

Figure 6 – Bounding Boxes Prediction (Source: SSD: Single Shot MultiBox Detector)

The figure 6 illustrates this idea. The output of the backbone model, the feature map, condense semantic information in a lower spatial resolution, which is used for the classification and box estimation.

(Liu *et al.*, 2016) suggests SSD, a box detector that uses the feature maps at different stages of the backbone, and, thus with different resolutions, the estimation can "naturally" handle objects of different sizes.

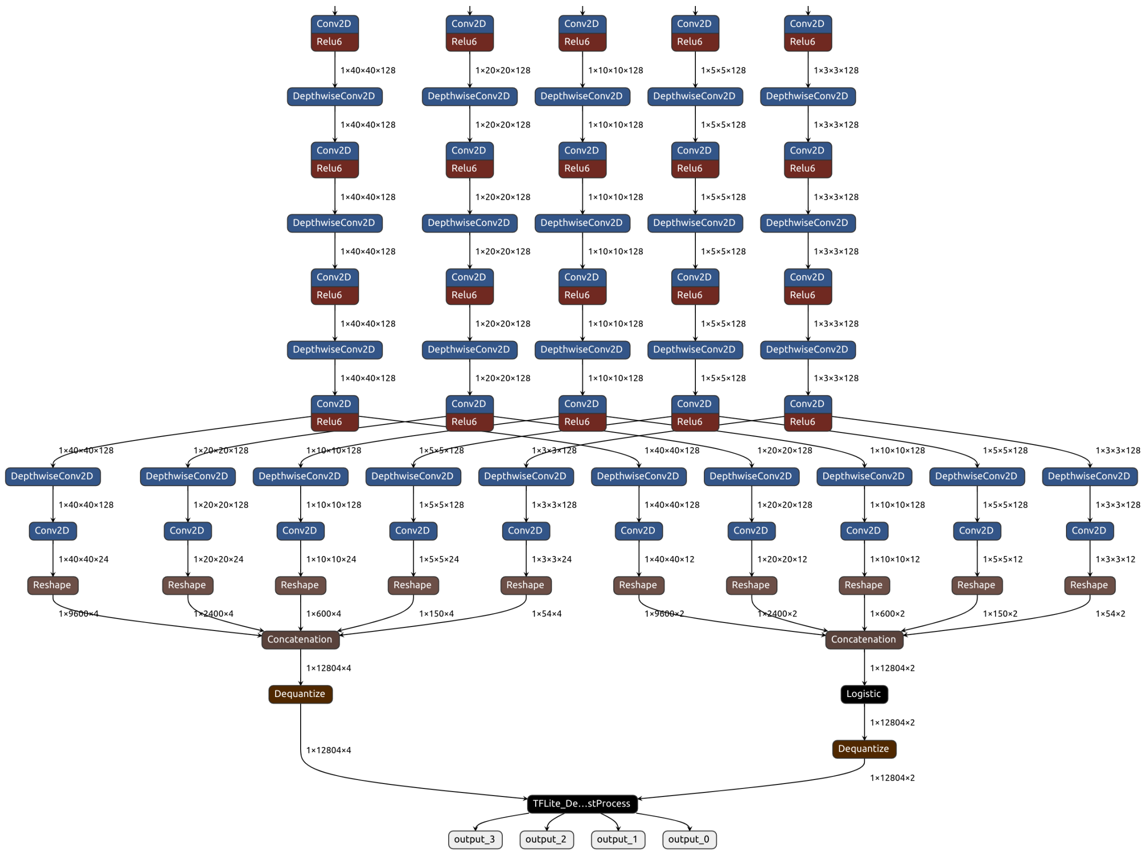

Figure 7 – The SSD model Architecture (Source: Author)

Figure 7 shows this: from top to down, we can see 5 branches of feature maps with different resolutions. At the end, the box predictions at different scales are concatenated into a final prediction. This same process is repeated for the class prediction. This multi staged prediction was later reproduced by other object detection algorithms such as the YOLO.

Also in (Liu *et al.*, 2016), the backbone model used is the VGG-16 (Simonyan; Zisserman, 2015), but in this work, we use the implementation from (Yu *et al.*, 2020) that uses the mobilenetv2 (Howard *et al.*, 2017) as the backbone for a better computational performance.

The YOLO (Redmon *et al.*, 2016) model uses a similar approach, but with one key difference, both the label classification and the box regression comes from the same convolutional layer, as illustrated on figure 8. From top to down, there is 3 main branches of different resolutions, the final layer concatenates all the predictions into a final output. It has the advantage of sharing weights, which makes the overall model more compact, light-weighted and faster.

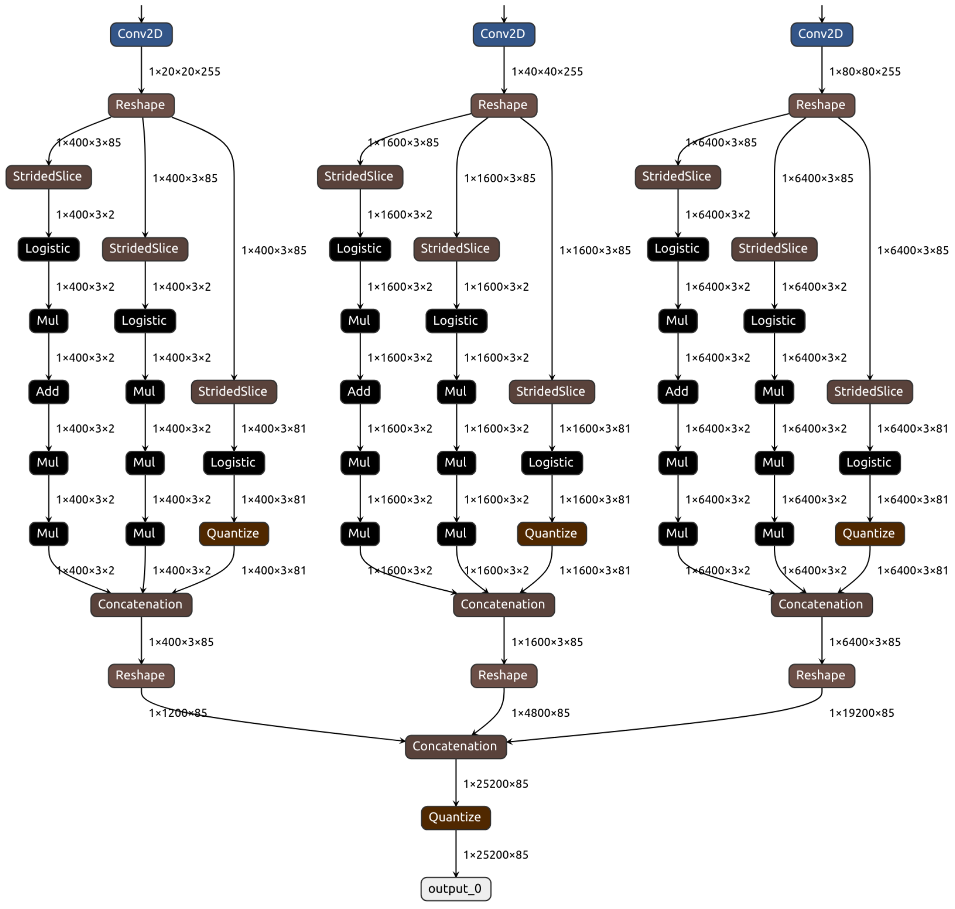

Figure 8 – The YOLO model Architecture (Source: Author)

In this work we are using the Ultralytics (Jocher, 2020) (Jocher; Qiu, 2024) implementation of the algorithm. While those models are open source, there's actually little information available about the design concepts and decisions.

4.4 Non-Max Suppression (NMS)

The use of the cell grid and the different scales can lead to redundant bounding boxes. The NMS is a post-processing technique that merges boxes with the same class and a high IoU into a single box that represents the object, thus avoiding duplicated boxes. The figure 9 shows an example of object detection with redundant boxes. By eliminating duplicates, NMS enhances both the performance and the interpretability of the outputs, which are crucial for autonomous robots.

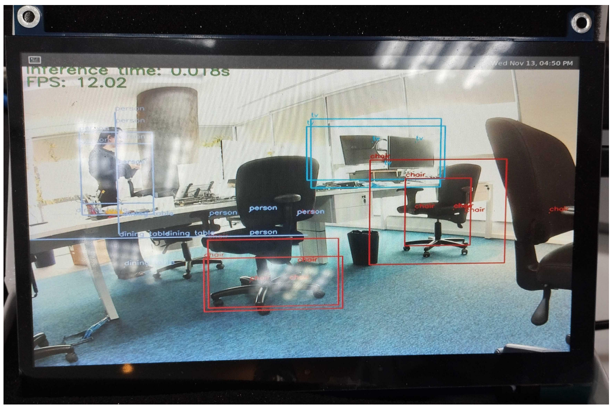

Figure 9 – Prediction without NMS (Source: Author)

Algorithm 1 Non-Maximum Suppression

```
Require: List of boxes $B$, scores $S$, and thresholds $T_{IoU}$, $T_S$
Ensure: Selected boxes
 1: Sort $B$ by $S$ in descending order
 2: Remove from B Where S < $T_S$
 3: Initialize $R \leftarrow \emptyset$
 4: Initialize $Q \leftarrow \emptyset$
 5: while $B$ is not empty do
 6:     $M \leftarrow$ box with highest score in $B$
 7:     Add $M$ to $R$
 8:     Add $s_M$ to $Q$
 9:     Remove $M$ from $B$
10:     for each $b \in B$ do
11:         if IoU$(M, b) > T_{IoU}$ then
12:             Remove $b$ from $B$
13:         end if
14:     end for
15: end while
16: return $R$, $Q$
```

The pseudo-code 1 shows the vanilla NMS algorithm, which is $O(n^2)$ with the number of boxes above the score threshold.

5 MONOCULAR DEPTH ESTIMATION

5.1 Introduction

This chapter gives a short context about the monocular depth estimation task. It also explains, in the high level, how this problem is modeled as a deep learning task.

5.2 Context

Depth perception is a key aspect of robotic tasks such as localization, mapping and obstacle detection and it typically relies on stereo-vision or external sensors like LiDAR and structured light. Those solutions, however, may be prohibitively expensive, both monetary and computationally, for large scale embedded projects. On the other hand, Monocular Depth Estimation predicts depth information from just a single image. This poses quite a challenge, for the problem is highly under-constrained and the model must rely on subtle visual cues and prior context. Figure 10 shows an example of a depth prediction: the brighter colors indicates that the object is closer to the camera.

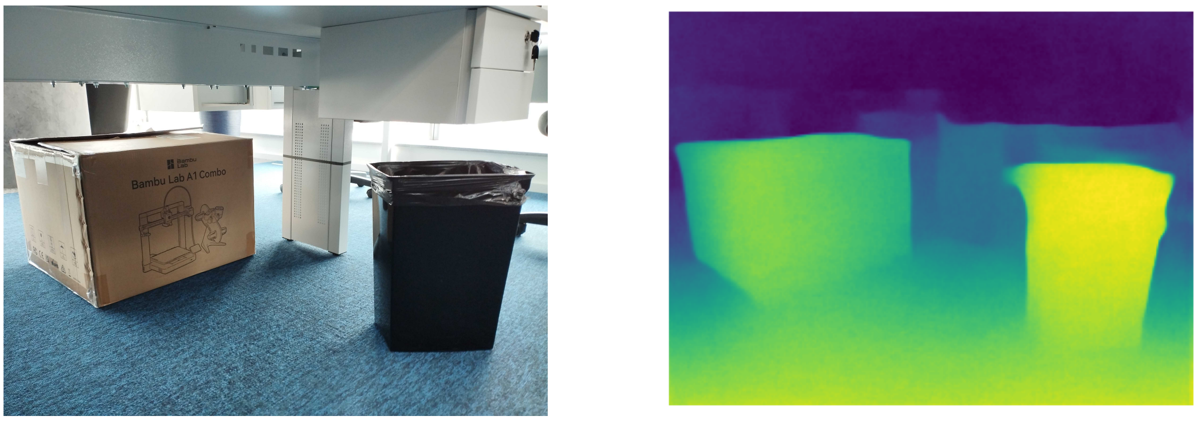

Figure 10 – Comparison of Original and Predicted Images (Source: Author)

5.3 Implementation Overview

Depth estimation is an image-to-image task in which the output is a pixel map where the pixel values are the predicted distances. Those type of models usually use the encoder-decoder architecture: the model progressively reduces the spatial dimensions (downsampling) to extract meaningful features while recovering spatial resolution (upsampling) to make pixel-level predictions. The higher dimension direct connection is often called a skip connection. Figure 11 illustrates the U-net (Ronneberger; Fischer; Brox, 2015) architecture, which is the mostly widely know encoder-decoder model.

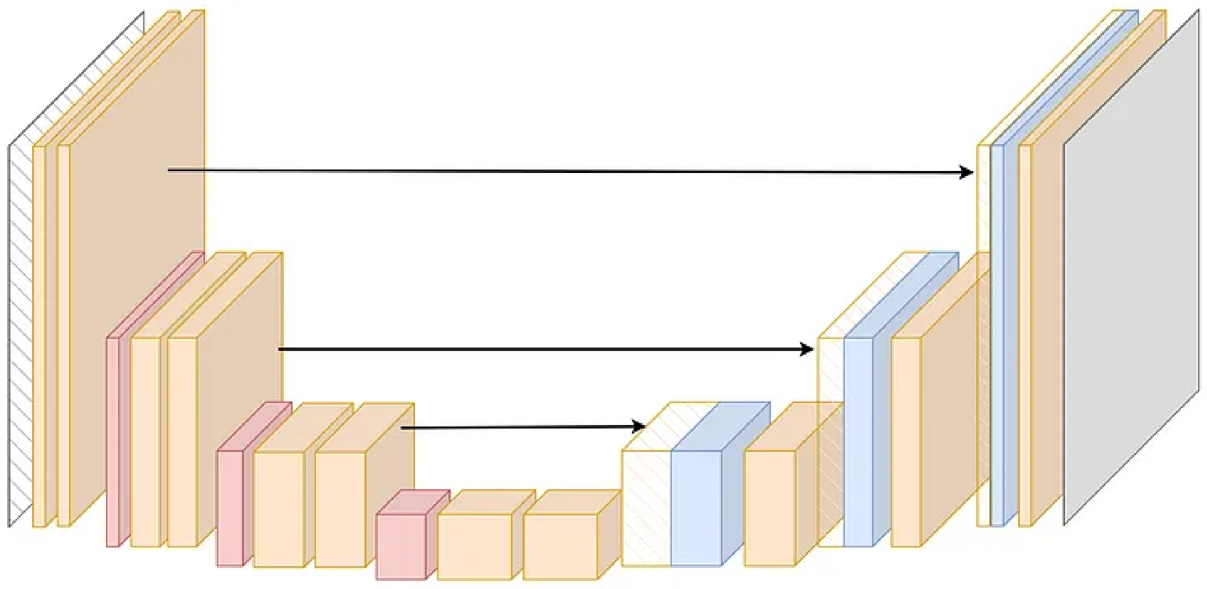

Figure 11 – Illustration of the U-Net Architecture (Source: Towards Data Science)

From left to right, the encoder extracts high-level spatial and contextual features from the input image, translating the spatial information into feature channels. The decoder reconstructs the depth map from the encoder's feature representation. The arrows are called skip connections and indicates that the latter layers have access to both spatial information and high level features.

(Ranftl *et al.*, 2019) proposes the MiDaS[1] model, which uses a Resnet (He *et al.*, 2015) encoder trained on ImageNet (Russakovsky *et al.*, 2015). The main contribution of those authors, however, is not from the model architecture, but on the training part. They propose the use of a custom dataset agnostic loss function which allows the integration of training data of different sources and formats, such as depth maps, cloud points, and stereo-graphical movies.

(Wofk *et al.*, 2019) proposes the FastDepth[2], a model optimized for inference on embedded devices. The main idea is to use the MobileNet (Howard *et al.*, 2017) as the encoder. This model uses the Depthwise Separable Convolutions Layers (Chollet, 2016), which the authors agrees that it enhances computational efficiency and reduces model size. The FastDepth's authors also make use of a harware aware model pruning (Yang *et al.*, 2018) to further optimize the network for runtime inference.

1 MiDaS: Robust Monocular Depth Estimation
2 FastDepth: Fast Monocular Depth Estimation on Embedded Systems

6 EDGE RUNTIME

6.1 Introduction

This chapter presents some details about running computer vision models on embedded devices. It gives an overview about the high level of the NPU and introduces 2 keys components: the quantization step and the delegates runtime.

6.2 Context

Most modern embedded devices include dedicated hardware accelerators for heavy computational tasks. As opposed to a common CPU, which is designed for general purpose tasks, NPUs, GPUs and TPUs are optimized to perform a specific task.

The NPU, used by the *Verdin i.MX8M Plus*, is a piece of hardware optmized to INT8 and INT16 Multiply-Accumulate operations. Figure 12 shows the high level of the NPU. The Vision Engine is optimized for fixed function pixel operations, such as blur and filtering, while the neural engine, for convolutional operations using dynamic kernels. The interface between the host system and the hardware accelerator is handled by the *OpenVX™* API with NN Extensions.

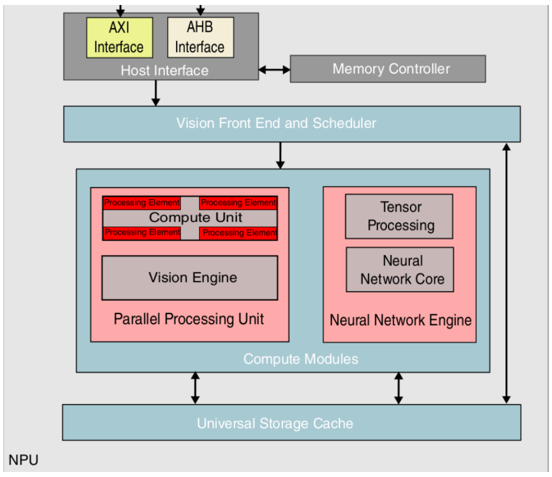

Figure 12 – NPU Overview (Source: *NXP i.MX8M* PLUS Product Page)

On the other hand, the Jenson Nano GPU uses the Maxwell architecture with 128 NVIDIA *CUDA®* cores. It works with float16 and *float32*, and while it doesn't have any special neural network accelerator, it still is a lot faster than a CPU given the parallel capacity.

6.3 Quantization

Model quantization is a machine learning optimization technique that reduces the precision of a deep learning model's parameters. The primary goal is to minimize the model's size, accelerate inference, and lower power consumption, making it especially useful for resource-constrained devices like embedded systems.

A common quantization process involves converting 32-bit floating-point representations to 8-bit integers. This results in up to a 4x reduction in model size, with a trade-off in precision. Despite the slight loss in accuracy, quantization significantly enhances inference performance by improving memory efficiency (e.g., reduced bandwidth and better cache utilization) and leveraging hardware-optimized integer operations, which leads to faster execution and reduced latency.

(Jacob *et al.*, 2017) proposes a schema to translate *float32* into a *int8* space and to efficiently handle the most common operations in the limited quantized space. The scale and zero point can be defined per tensor of per tensor channel.

$$real\_value = (int8\_value - zero\_point) \times scale \tag{6.1}$$

Usually, the layers quantization is handled by conversion tools such as Edge AI Torch [1], *TensorFlow* Model Optimization [2] or *ONNX* Quantization [3], when exporting the model to runtime format such as *.tflite* or *.trt*.

On the practical side, one point of attention is how to integrate the preprocessing function, the input quantization and the model. While images are usually represented using *uin8* RGB or BGR, in general computer vision models, it is common to either scale ($[0, 1]$) or normalize ($[-0.5, 0.5]$) the inputs.

6.4 Delegates

As the name suggests, delegates are the mechanism used by runtime frameworks, such as *TFLite* or *LiteRT*, to assign instructions to the NPU, offloading the CPU. This is one of the biggest challenges of running embedded models. The delegate works as a bridge interface between a specific runtime framework and a specific hardware low level API, therefore, for each combination of runtime framework and hardware accelerator we need a specific set of delegations.

The TIM-VX[4]is a software integration module from *VeriSilicon* that creates bindings

[1] https://github.com/google-ai-edge/ai-edge-torch
[2] https://www.tensorflow.org/model_optimization
[3] https://github.com/microsoft/onnxruntime/tree/main/onnxruntime/python/tools
[4] https://github.com/nxp-imx/tim-vx-imx

from the most common operations from, for instance, TFLite to calls, using the *OpenVX™* API, to the *NXP i.MX8M Plus* NPU.

This introduces two key limitations to the type of deep learning models we can be executed on the NPU:

- Only a subset of operations used by general machine learning frameworks (*TensorFlow* and *PyTorch*), are available for the runtime framework (*TFLite*)[5]
- Only a subset of the runtime operations are covered by the delegations.

The TIM-VX module does have support for compound custom operators, that is, new operations that can be described from built-in base operators. The Figure 13 shows an overview of how to implement an RNN Cell based on the Fully Connected Layer and the Tanh operators. The support for new operators, however, are out of the scope of this work.

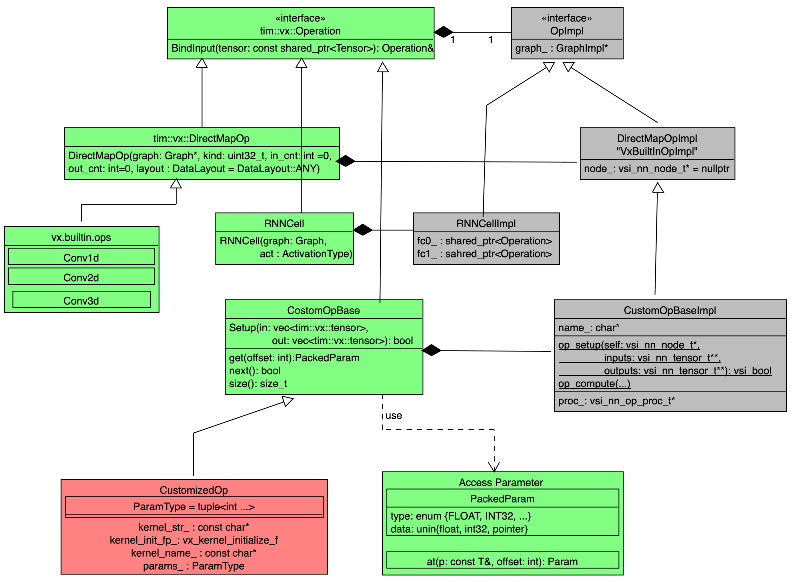
«interface»
tim::vx::Operation
BindInput(tensor: const shared_ptr<Tensor>): Operation&
1
1
«interface»
OpImpl
graph_ : GraphImpl*
tim::vx::DirectMapOp
DirectMapOp(graph: Graph*, kind: uint32_t, in_cnt: int =0, out_cnt: int=0, layout : DataLayout = DataLayout::ANY)
DirectMapOpImpl
"VxBuiltInOpImpl"
node_: vsi_nn_node_t* = nullptr
vx.builtin.ops
Conv1d
Conv2d
Conv3d
RNNCell
RNNCell(graph: Graph, act : ActivationType)
RNNCellImpl
fc0_ : shared_ptr<Operation>
fc1_ : sahred_ptr<Operation>
CostomOpBase
Setup(in: vec<tim::vx::tensor>, out: vec<tim::vx::tensor>): bool
get(offset: int):PackedParam
next(): bool
size(): size_t
CustomOpBaseImpl
name_: char*
op_setup(self: vsi_nn_node_t*, inputs: vsi_nn_tensor_t**, outputs: vsi_nn_tensor_t**): vsi_bool
op_compute(...)
proc_: vsi_nn_op_proc_t*
use
CustomizedOp
ParamType = tuple<int ...>
kernel_str_ : const char*
kernel_init_fp_: vx_kernel_initialize_f
kernel_name_ : const char*
params_ : ParamType
Access Parameter
PackedParam
type: enum {FLOAT, INT32, ...}
data: unin{float, int32, pointer}
at(p: const T&, offset: int): Param

Figure 13 – TIM-VX Custom Operator

[5] Refer to: i.MX Machine Learning User Guide.

7 RESULTS

Table 6 – Models Benchmark

Device	Model	Inference Time (ms)	FPS
Jetson	FastDepth	31.0 ± 5.0	8.48 ± 0.4
Raspi	FastDepth	68.0 ± 0.5	8.45 ± 0.04
Verdin	FastDepth	10.0 ± 0.1	13.2 ± 0.2
Jetson	Midas	not run	not run
Raspi	Midas	402.0 ± 0.7	2.26 ± 0.02
Verdin	Midas	159.0 ± 20.0	4.86 ± 0.01
Jetson	SSD	215.0 ± 10.0	3.52 ± 0.2
Raspi	SSD	125.0 ± 2.0	6.03 ± 0.04
Verdin	SSD	8.0 ± 0.0	13.3 ± 0.3
Jetson	Yolo11n_640	92.0 ± 4.0	3.89 ± 0.4
Raspi	Yolo11n_640	461.0 ± 0.5	1.8 ± 0.02
Verdin	Yolo11n_640	117.0 ± 0.4	5.82 ± 0.03
Jetson	Yolo11s_320	113.0 ± 6.0	5.13 ± 0.2
Raspi	Yolo11s_320	314.0 ± 0.4	2.73 ± 0.02
Verdin	Yolo11s_320	49.0 ± 0.0	10.6 ± 0.06
Jetson	Yolov5nu_640	47.0 ± 6.0	3.87 ± 0.2
Raspi	Yolov5nu_640	538.0 ± 0.6	2.43 ± 0.02
Verdin	Yolov5nu_640	66.0 ± 0.2	6.83 ± 0.1
Jetson	Yolov5su_320	47.0 ± 2.0	6.23 ± 0.1
Raspi	Yolov5su_320	361.0 ± 0.5	2.45 ± 0.02
Verdin	Yolov5su_320	31.0 ± 0.0	13.2 ± 0.09

Following the steps described in 3.2, table 6 shows a comparison between the different models (include the tasks columns) for each device, the highlighted row shows the fastest inference time. From this objectives, we want to choose an object detection and a depth estimation algorithms that can be used at the same time; and, a hardware that can deliver real-time processing.

The NVidia Jetson was not able to run the Midas model because of an incompatibility between the model and the version of tensorflow runtime.

For the YOLO models, when comparing versions 5 and 11, overall we observed an increase in the inference time between models of the same input and model sizes. When comparing the Toradex Verdin with the NVidia Jetson, we observed that for Verdin is better for smaller input sizes, while for the 640 pixels version, the Jetson is faster.

The choice for the final setup is described in table 7.

Table 7 – Selected Model Setup

Device	Object Detection	Depth Estimation	Inference Time (ms)	FPS
Verdin-iMX8MP	SSD_MobileNet_V2	FastDepth	19.2 ± 0.4	12.40 ± 0.19

One of the reasons to choose the SSD model over the YOLO is that the first share the same backbone model as the fastdepth model. This can improve the runtime inference speed, as part of the calculation can be used for both models. The resulting prediction can be seen on Figure 14.

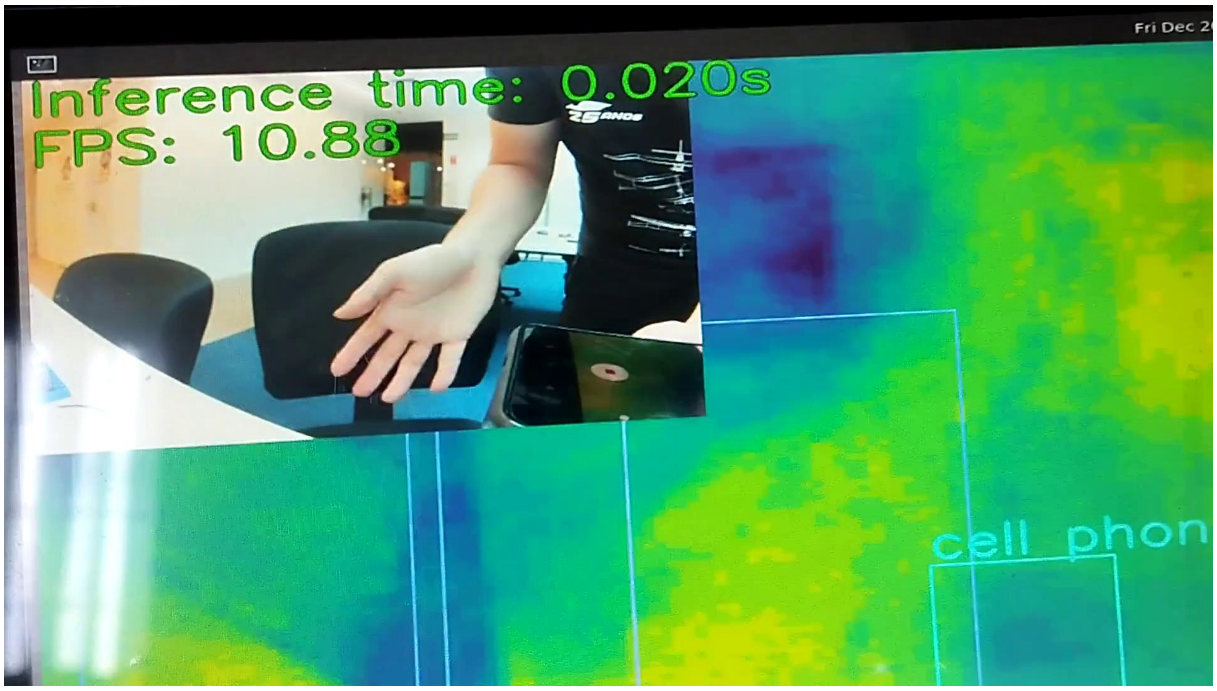

Figure 14 – Evaluation Test Result

8 CONCLUSION

In this work we evaluate the feasibility of using object detection and monocular depth estimation models in embedded devices. After a carefull study of computer vision models and evaluation metrics, the final setup is obtained through empirical benchmarking. This setup is assembled and evaluated in a real world environment.

REFERENCES

ALQAHTANI1, D.; AL. et. Comprehensive benchmarking on tpus. **arXiv preprint arXiv:2409.16808**, 2024. Disponível em: https://arxiv.org/pdf/2409.16808.

CHOLLET, F. Xception: Deep learning with depthwise separable convolutions. **CoRR**, abs/1610.02357, 2016. Disponível em: http://arxiv.org/abs/1610.02357.

HE, K. *et al.* Deep residual learning for image recognition. **CoRR**, abs/1512.03385, 2015. Disponível em: http://arxiv.org/abs/1512.03385.

HONEGGER, D.; AL. et. Stereo vision for uavs. **IROS 2014**, 2014. Disponível em: https://people.inf.ethz.ch/pomarc/pubs/HoneggerIROS14.pdf.

HOWARD, A. G.; AL. et. Mobilenets: Efficient convolutional neural networks for mobile vision applications. **arXiv preprint arXiv:1704.04861**, 2017. Disponível em: https://arxiv.org/pdf/1704.04861.

HOWARD, A. G. *et al.* Mobilenets: Efficient convolutional neural networks for mobile vision applications. **CoRR**, abs/1704.04861, 2017. Disponível em: http://arxiv.org/abs/1704.04861.

JACOB, B. *et al.* Quantization and training of neural networks for efficient integer-arithmetic-only inference. **CoRR**, abs/1712.05877, 2017. Disponível em: http://arxiv.org/abs/1712.05877.

JOCHER, G. **Ultralytics YOLOv5**. 2020. Disponível em: https://github.com/ultralytics/yolov5.

JOCHER, G.; QIU, J. **Ultralytics YOLO11**. 2024. Disponível em: https://github.com/ultralytics/ultralytics.

KHALILI, E.; AL. et. Object detection for robotic manipulators. **arXiv preprint arXiv:2409.06693**, 2024. Disponível em: https://arxiv.org/abs/2409.06693.

KHAN, M.; AL. et. Background subtraction on fpga. **IEEE Transactions on VLSI Systems**, 2016. Disponível em: https://colab.ws/articles/10.1109/tvlsi.2016.2567485.

LIN, T.-Y. *et al.* Microsoft coco: Common objects in context. **arXiv preprint arXiv:1405.0312**, 2014. Disponível em: https://arxiv.org/abs/1405.0312.

LIU, W. *et al.* Ssd: Single shot multibox detector. *In*: _____. **Computer Vision – ECCV 2016**. Springer International Publishing, 2016. p. 21–37. ISBN 9783319464480. Disponível em: http://dx.doi.org/10.1007/978-3-319-46448-0_2.

MOHAMMADI, M.; AL. et. Tpu adaptations for neural networks. **arXiv preprint arXiv:2305.15422**, 2023. Disponível em: https://arxiv.org/pdf/2305.15422.

OTTERNESS, N. e. a. Embedded gpu for object detection. **IEEE Embedded Systems**, 2017. Disponível em: https://ieeexplore.ieee.org/document/7939053.

RANFTL, R. *et al.* Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. **arXiv preprint arXiv:1907.01341**, 2019. Disponível em: https://arxiv.org/abs/1907.01341.

REDMON, J. *et al.* **You Only Look Once: Unified, Real-Time Object Detection**. 2016. Disponível em: https://arxiv.org/abs/1506.02640.

RONNEBERGER, O.; FISCHER, P.; BROX, T. U-net: Convolutional networks for biomedical image segmentation. *In*: **Medical Image Computing and Computer-Assisted Intervention (MICCAI)**. Springer, 2015. v. 9351, p. 234–241. Disponível em: https://arxiv.org/abs/1505.04597.

RUSSAKOVSKY, O. *et al.* Imagenet large scale visual recognition challenge. **International Journal of Computer Vision**, Springer, v. 115, n. 3, p. 211–252, 2015. Disponível em: https://arxiv.org/abs/1409.0575.

SILBERMAN, N. *et al.* Indoor segmentation and support inference from rgbd images. **arXiv preprint arXiv:1208.6106**, 2012. Disponível em: https://arxiv.org/abs/1208.6106.

SIMONYAN, K.; ZISSERMAN, A. **Very Deep Convolutional Networks for Large-Scale Image Recognition**. 2015. Disponível em: https://arxiv.org/abs/1409.1556.

SUN, Y.; AL. et. Google tpu: Hardware for scalable neural networks. **arXiv preprint arXiv:2108.13732**, 2021. Disponível em: https://arxiv.org/pdf/2108.13732.

WOFK, D. *et al.* Fastdepth: Fast monocular depth estimation on embedded systems. *In*: **Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)**. [*S.l.: s.n.*], 2019. p. 6101–6108. Disponível em: https://arxiv.org/abs/1903.03273.

YANG, T.-J. *et al.* **NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications**. 2018. Disponível em: https://arxiv.org/abs/1804.03230.

YU, H. *et al.* **TensorFlow Model Garden**. 2020. https://github.com/tensorflow/models.

ZHAO, R.; AL. et. Object detection with embedded systems. *In*: **Proceedings of Computer Vision Conference**. [*S.l.: s.n.*], 2016. Disponível em: https://link.springer.com/chapter/10.1007/978-3-319-56258-2_22.

ZHOU, W. *et al.* Learning depth from monocular videos using direct methods. **arXiv preprint arXiv:1811.01791**, 2018. Disponível em: https://arxiv.org/abs/1811.01791.