Tampere University

Huy Trinh Gia

ON THE EVALUATION OF NEURAL NETWORK DEPLOYMENT OPTIONS

Bachelor’s thesis

Bachelor’s thesis
Science and Engineer
Supervisor: Dr. Dat Thanh Tran and Prof. Moncef Gabbouj
April 2023

ABSTRACT

Huy Trinh Gia: On the evaluation of neural network deployment options
Bachelor's thesis
Tampere University
April 2023

It is known that machine learning inference models are already transforming computing. However, it still requires massive power usage for training multiple gigantic datasets. Furthermore, inferences can be carried out in the cloud by utilizing high-performance computing (HPC) platforms for non-time critical workflow. However, it is now commonly conducted locally on edge devices for real-time applications such as video processing, image/pattern recognition, and object detection. It is crucial that the result is generated as quickly as possible with low power and effective cost.

In this thesis, we compare multiple common neural network models' performance in terms of inference time and characterized pattern of 5 commercial edge devices: Raspberry Pi 3, Raspberry Pi 4, Jetson Nano, Jetson TX2, and Jetson AGX Xavier. These neural network models are taken mainly from 4 computer vision tasks: image classification, object detection, human pose estimation, and semantic segmentation. These models are converted from Pytorch to ONNX and TensorRT format for benchmarking. Finally, our showcase results are summed up in tables and virtualized.

Keywords: Machine Learning, Neural Network, Embedded Devices, Edge Computing, Benchmark

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

PREFACE

This thesis work was carried out at Tampere University of Finland from January 2023 to April 2023 as a Bachelor's thesis of Science and Engineering.

Above all, I would like to express my sincere and deepest gratitude toward Prof. Moncef Gabbouj for giving his valuable guidance and comment from the beginning. Especially, I am immensely grateful to my supervisor Dr. Dat Thanh Tran for continuous feedback and advice for conducting experimental work. Moreover, I feel thankful for BSc. Quoc Nguyen for his perspective ideas and code feedback. Lastly, I really appreciate the encouragement and support from family and friends whoever push my struggles, shared joy, and daily discussion.

Tampere, Finland, 1st April 2023

Huy Trinh Gia

CONTENTS

1. Introduction . . . 1
2. Theoretical Background . . . 2
2.1 Machine Learning . . . 2
2.2 Neural Networks . . . 2
2.2.1 Convolution Neural Network . . . 3
2.2.2 Multi-Branch Network . . . 3
2.2.3 Residual Networks . . . 4
2.2.4 Densely-Connected Convolutional Networks . . . 5
2.2.5 Neural Network Models for Mobile Devices . . . 6
2.3 Deep Learning Framework and Platform . . . 7
2.3.1 Edge AI . . . 7
2.3.2 ONNX . . . 7
2.3.3 TensorRT . . . 8
2.3.4 OpenVINO . . . 9
2.4 Computer Vision Application . . . 9
2.4.1 Image Classification . . . 10
2.4.2 Object Detection . . . 11
2.4.3 Human Pose Estimation . . . 13
2.4.4 Semantic Segementation . . . 14
3. Methods . . . 16
4. Evaluation . . . 22
4.1 Image Classification . . . 22
4.2 Object Detection . . . 39
4.3 Human Pose and Semantic Segmentation . . . 42
5. Conclusion . . . 46
References . . . 47

LIST OF ABBREVIATIONS

ML	Machine Learning
NN	Neural Network
AI	Artificial Intelligence
CNN	Convolution Neural Network
DL	Deep Learning
IoT	Internet of Things
CPU	Central Processing Unit
GPU	Graphic Processing Unit
TPU	Tensor Processing Unit
SBC	Single-board computer
RPi4	Raspberry Pi 4
RPi3	Raspberry Pi 3
ONNX	Open Neural Network Exchange

1. INTRODUCTION

There is increasing interest in using neural network (NN) models for edge devices such as smartphones, tablets, and smart TVs. This is because NN models can capture complex patterns and relationships in data that classical statistical models are hard to detect. Thus, their application comprises a wide range of domains, such as computer vision, natural language processing, and bioinformatics. However, deploying NN models on these devices can be challenging because they need to run on low-power hardware with minimal memory and processing capabilities.

Despite these disadvantages, their performance can still be evaluated to get the depth of understanding required to use these devices. In addition, our primary goal is to understand the challenges involving building them and optimizing their run-time. This thesis is organized as follows. Chapter 2 will briefly introduce the theoretical concept and general discussion over neural networks being deployed and the target framework. Then, the experimental setup will be described in Chapter 3, and its result will be evaluated in Chapter 4. Finally, Chapter 5 will sum up the thesis and provide a technical discussion about future research based on this thesis.

2. THEORETICAL BACKGROUND

This chapter briefly describes and explains the basic theory of models used in this benchmark, their parameter characteristics, and device information.

2.1 Machine Learning

In the past decade, machine learning has been recognized as an inherently multi-disciplinary field since its idea merged neuroscience, biology, mathematics, physics, and statistic to make computer learns. Since then, it has been modified, adapted, and evolved to get higher accuracy, more power efficiency, and integrated into more reality applications.

2.2 Neural Networks

Artificial Neural Network, whose name was inspired by biological inspiration, comprises interconnected computing nodes or neurons. A simple neural network architecture includes three layers of them (input, hidden, and output layer). A more advanced architecture, known as Deep Neural Network (DNN), can be made up of millions of artificial neurons and weights that connect between them. However, they require much more training as compared to other machine learning models.

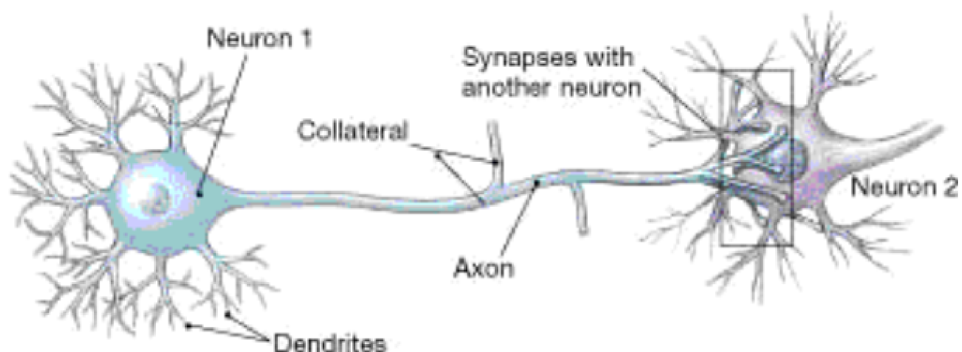

Figure 2.1. *Neuron cells of the brain.*

2.2.1 Convolution Neural Network

In mathematics, convolution 2.1 measures the sum of the overlaps of one function v and all of its shifted versions w. In terms of image classification, convolution extracts input image features using different types of filters or kernels, thus preserving the relationship between pixels and producing various operations such as box blurring, sharpening, and edge detection.

$$(v * w)(t) = \int_{-\infty}^{\infty} v(\lambda)w(t-\lambda)\,d\lambda \tag{2.1}$$

Alexnet [1] is the first CNN to win the LSVRC (Large Scale Visual Recognition Challenge) on evaluating a massive dataset of labeled images (ImageNet) and achieving higher accuracy on several visual recognition tasks. The significant feature of Alexnet is that it uses ReLU (Rectified Linear Unit) Nonlinearity as an activation layer, which achieves much faster training time than saturating ones like tanh or sigmoid.

SqueezeNet [2], as the paper's name, provides an intelligent architecture for the same accuracy as Alexnet but achieves three times faster and 500 times smaller by using a 1x1 point-wise filter instead of a 3x3 for a bottleneck layer. In addition, the number of input channels is decreased into 3x1 filters using a squeeze layer and downsample late to keep a large feature map. With fewer parameters, it can easily be fitted into limited-memory devices and transmitted over the computer network.

ShuffleNet [3] flows information across feature channels by shuffling novel channels. The author also introduces the ShuffleNet unit using this method for small network design and point-wise group convolution with channel shuffle for efficient computation. As a result, it achieves approximately 13 times speedup over Alexnet with comparable accuracy.

2.2.2 Multi-Branch Network

GoogLenet (Inception V1) [4] won the ImageNet Challenge in 2014 by strengthening Network in Network (NIiN) blocks with repeated block and cocktail of convolution kernels [4]. The key contribution to its success is that it solves the selecting convolution kernel issue in an ingenious way. While others try to identify which convolution kernels sizing from 1x1 to 11x11 suit, GoogLenet concatenated multi-branch of them to a block known as Inception block as Figure 2.2.

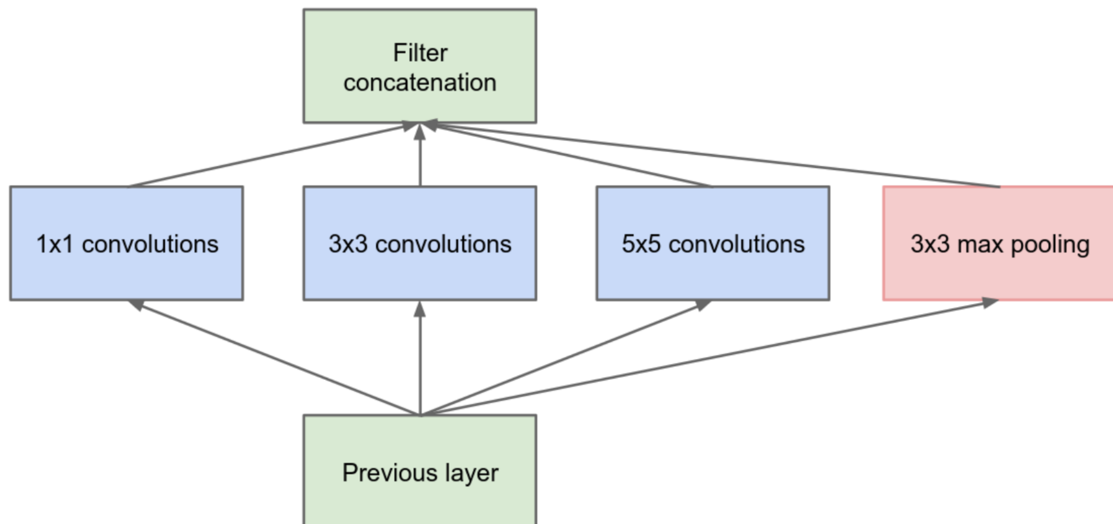

Figure 2.2. *Inception blocks (Naive Form). [4]*

Inception V3 further enhances predecessors by factorizing smaller convolutions (from 5x5 to 1x1 convolution) and asymmetric convolutions (of form nx1) by replacing 3x3 convolutions with 1x3 convolution and 3x1 convolution. In addition, they efficiently reduce the grid size and use auxiliary classifiers to address the vanishing gradient problem. Thus, as expected, Inception V3 achieves better accuracy and less computational cost compared to the previous Inception version.

2.2.3 Residual Networks

Deeper and more complex networks, with state-of-the-art networks, are growing to over a hundred layers, which presents the vanishing gradients issue in the training period. However, the Microsoft Research paper's group tries to resolve this by introducing the Deep Residual learning framework [5]. This approach lets the network fit the residual mapping by adding a "residual connection" (skip connection) as Figure 2.3 from one layer to a consecutive one, which is known as the **ResNet block**. While the regularization will skip over additional ineffective layers, the weights or kernels of useful ones would be non-zero. As a result, by stacking multiple Resnet blocks, the author can form and make tests with 100 and 1000 layers on the CIFAR-10 dataset.

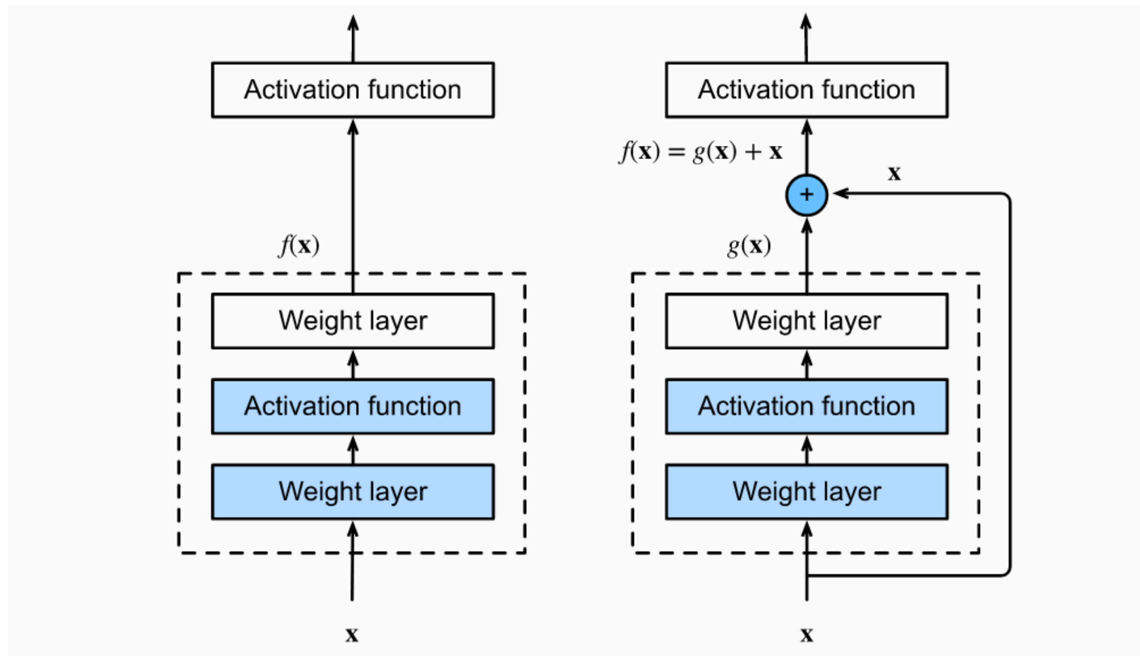

Figure 2.3. *Regular block (left) and Residual block (right). [6]*

2.2.4 Densely-Connected Convolutional Networks

Continuing from ResNet on the way to increase the depth of deep convolution networks, **DenseNet** [7] extends the rational parametrization of deep network functions. While ResNet decomposes function $f(x)$ into linear terms and more complex term and output feature maps of the layer are added together, Densenet concatenates them. In other words, it applies mapping from each current layer to an increasingly complex sequence of preceding layers, denoted as equation 2.2 [6]:

$$\mathbf{x} \rightarrow [\mathbf{x}, \mathbf{f_1}(\mathbf{x}), \mathbf{f_2}([\mathbf{x}, \mathbf{f_1}(\mathbf{x})]), \mathbf{f_3}([\mathbf{x}, \mathbf{f_1}(\mathbf{x}), \mathbf{f_2}([\mathbf{x}, \mathbf{f_1}(\mathbf{x})])]), ...] \tag{2.2}$$

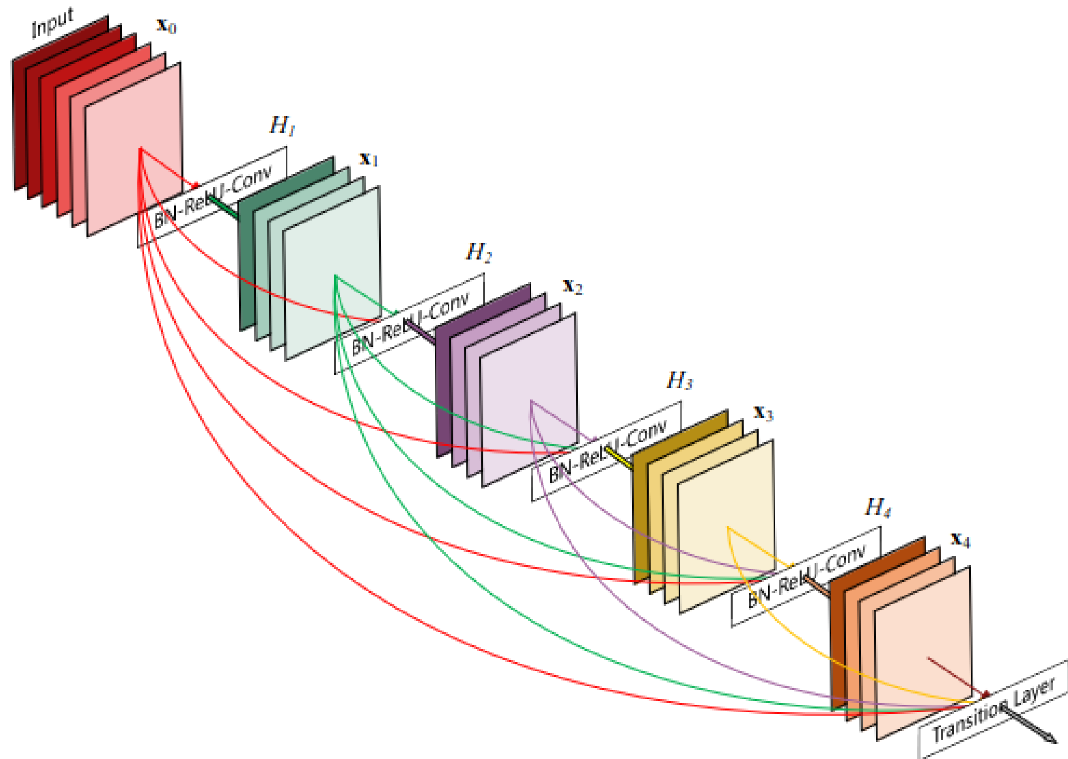

Figure 2.4. *A 5-layer dense block. [7]*

Therefore, its chain would preserve and reuses the feed-forward nature of earlier layers' features. Muck like ResNet, the DenseNet is constructed from DenseBlocks, where the dimensions of feature maps within the block remain the same, and Transition layers are laid between. The number of these filters, however, adjusts to in charge of downsampling applying a batch normalization as illustrated in Figure 2.4

2.2.5 Neural Network Models for Mobile Devices

MobileNet is a convolutional neural network architecture designed for efficient and lightweight image classification on mobile devices and embedded systems. Since MobileNet uses depthwise separable, which separates spatial and depth dimension convolution operations, it has vastly reduced the number of parameters while maintaining high accuracy. In addition, MobileNet applies a technique called channel-wise linear bottleneck for input channels before depthwise convolution operation.

Like MobileNet, **MnasNet** also uses deep-wise separable convolution operations to decrease computations and network size. Nonetheless, MnasNet incorporates squeeze-and-excitation features to achieve state-of-art performance on several benchmark datasets.

2.3 Deep Learning Framework and Platform

2.3.1 Edge AI

In general, there have been many surveys and papers working on evaluating and optimizing neural network performance for embedded devices [8] [9] [10] [11]. In machine learning, edge devices play a significant role in making decisions and predictions as close as possible to the original data sources. Thus, it is also called edge artificial intelligence or edge AI. Several types of data processing can be done on edge computing, such as visual inference, anomaly detection, environment monitoring, and multi-access edge computing (MEC). In visual inference, edges compute and perform machine inference on the video stream with high-resolution cameras. A more sophisticated neural network deployment on edge is self-driving cars, where models make inferences in almost real-time and give directions for controllers and sensors.

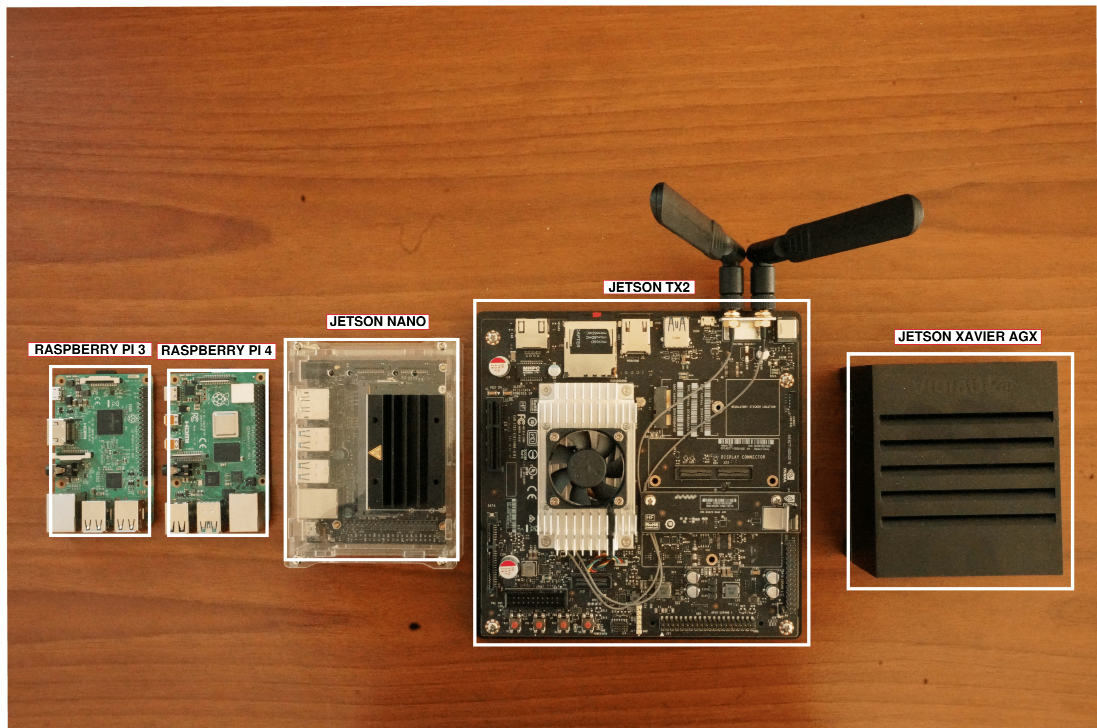

Figure 2.5. *All devices used in this thesis.*

2.3.2 ONNX

Previously, framework dependencies were a concern when deploying neural networks on embedded devices since the software distribution had to support the used framework. However, a library supporting interoperability named **ONNX** (`https://onnx.ai/`), which stands for Open Neural Network Exchange, was introduced to resolve this challenge. Several frameworks, including Pytorch, Tensorflow, Keras, SAS, Matlab, etc., support it

as the figure 2.6.

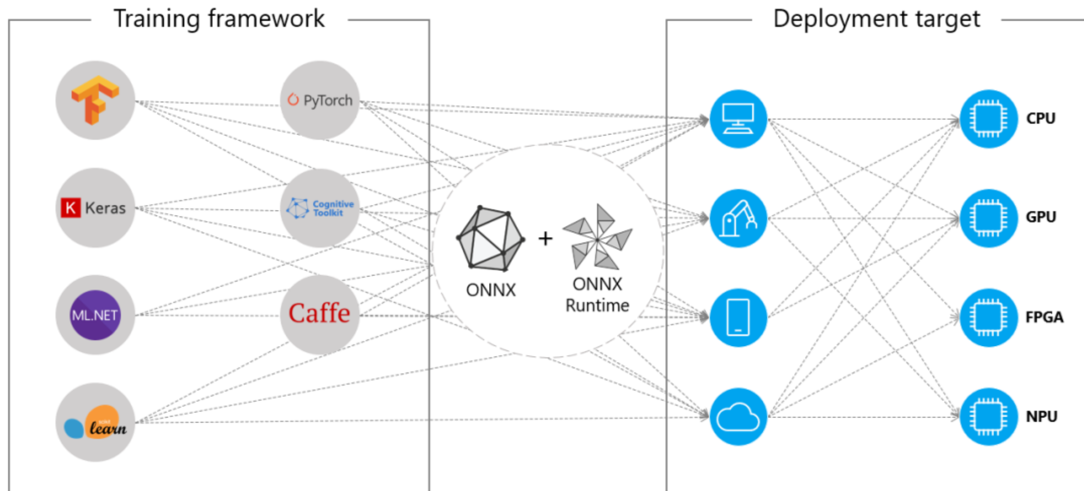

Figure 2.6. *ONNX interoperability between different frameworks. [11]*

Theoretically, any models are cross-platform-operated over various AI frameworks without worrying about compatibility. Specifically, any DL models developed by the supporting framework can be used on embedded devices as long as it is adequately converted. ONNX runtime is a high-performance inference engine that enables the efficient execution of deep learning models across various hardware platforms. By providing a cross-platform API, developers can deploy multiple training framework models on CPU, GPU, and other specialized accelerators.

2.3.3 TensorRT

In addition to ONNX runtime, NVIDIA developed TensorRT, another high-performance inference engine that targets NVIDIA GPUs. It can take advantage of the specialized architecture of NVIDIA GPU to deliver faster inference times, reducing latencies and improving throughputs. As an inference-oriented framework, TensorRT's primary goal is to optimize end-user performance and accelerate inferences using mixed-precision computation and quantization (INT8 and FP16). Furthermore, TensorRT reuses memory and fusion operations, minimizing memory footprint.

As shown in Figure 2.7, combining input and filter sizes with different weights to form a single 1x1 CBR layer can significantly reduce computational costs and memory overhead. This is especially useful for embedded devices where resources are limited.

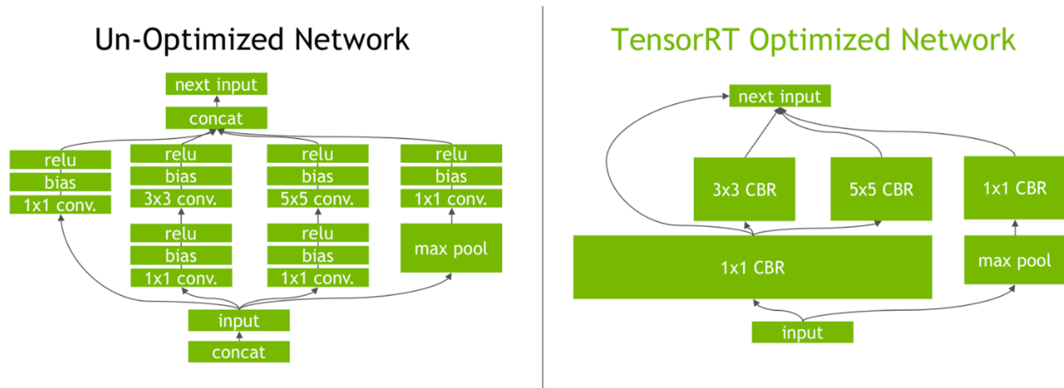

Figure 2.7. *TensorRT's layer fusion on the GoogleNet Inception module graph. [12]*

2.3.4 OpenVINO

OpenVINO is directly compatible with ONNX and used to optimize the deep learning model's neural network deployment on vision-related tasks. Specifically, it contains tools and libraries for optimizing neural networks by applying different techniques (for example, pruning, quantization, ...) to speed up inference on Intel architecture in a hardware-agnostic way. Hardware-agnostic or device-agnostic illustrates the computing capacity of various systems without requiring a specific adapting process. Thus, the OpenVINO toolkit benefits from faster inference, optimized libraries (OpenCV, e.g.), and heterogeneous-platform supported on Intel architectures (CPU, GPU, VPU, FPGA).

Figure 2.8 shows that OpenVINO takes multiple deep learning models from frameworks such as Tensorflow, Pytorch, Mxnet, Keras, ONNX, and Caffe and converts them to one standard format IR (Intermediate Representation) that can be run on any Intel devices. At the same time, it contains some fundamental tools:

Model Optimizer: Cross-platform command line tool for load-trained deep learning model into memory, building internal representation, optimizes and produces IR format, which is understood by the inference engine.
Inference Engine: The inference engine reads IR format files and runs the converted model for different hardware platforms.
Model Zoo: Openvino's model Zoo provides pre-trained optimized models (such as VGG16, Alexnet, Yolo, etc.) for object detection, image classification, image segmentation, etc.

2.4 Computer Vision Application

Computer vision is a field of study that focuses on enabling machines to interpret and understand visual data from the world around us. Specifically, it consists of the development

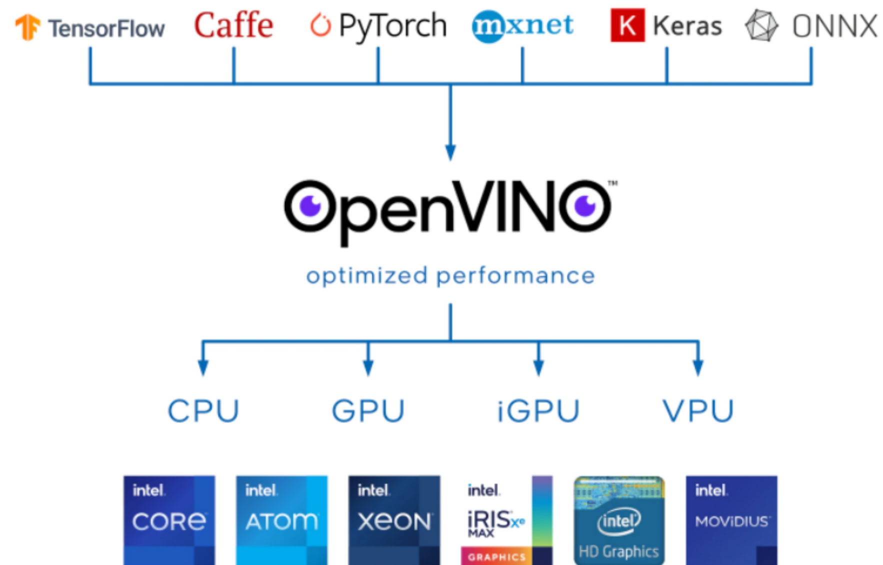

Figure 2.8. *OpenVINO interoperability. 2.8*

of algorithms and techniques for processing, analyzing, and extracting information from images and videos. Figures 2.12, 2.14, 2.16 below show the few result obtained from the benchmark for each application.

2.4.1 Image Classification

Image classification is a fundamental task in computer vision, which involves assigning a label or category to an input image based on its visual contents. In other words, its primary objective is to determine whether a specific object is present in an image and to identify its category. Convolutional neural networks (CNNs) are widely used for image classification due to their ability to extract meaningful features from images. CNNs consist of multiple layers, including an input layer, convolution and pooling layers, and a fully-connected layer near the output layer, as illustrated in figure 2.9. The convolution layers generate multiple feature maps, each presenting a specific input image feature. The extracted features from an input image serve as the backbone for other application models.

Depending on the specific application, the extracted features may be fed into different functional layers for training or inferencing. For instance, the object detection model fasterrcnn_resnet50_fpn_v2 uses the resnet50 classification model as its backbone to extract features from images. These features are then fed into a separate object detection head to predict the location and category of objects within the images.

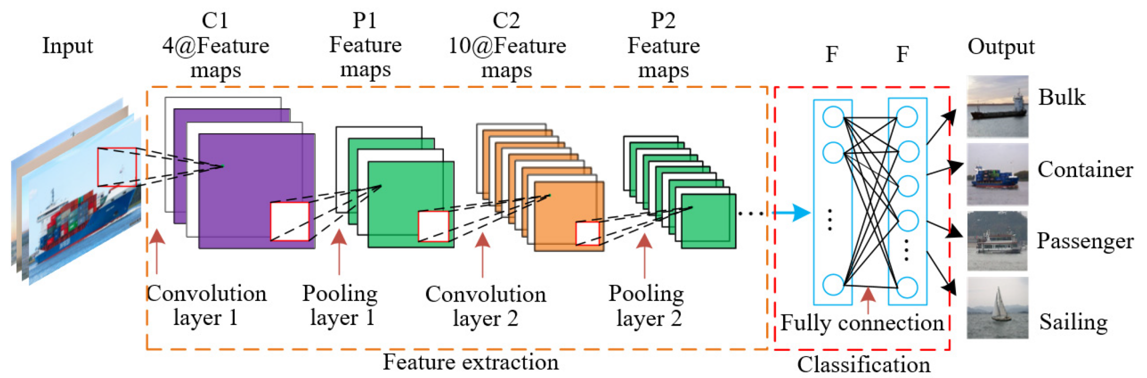

Figure 2.9. *Typical convolution neural network (CNN) structure. [13]*

2.4.2 Object Detection

Object Detection advances the image classification task by defining the actual position of the objects. Some well-known object detection applications include autonomous vehicles, surveillance systems, and robotics. One common approach to object detection is the region-based CNN (R-CNN) family of models [14].

Figure 2.10 shows models first generate region proposals, which are potential object locations within the image, and then extract features from each proposal using a CNN. These features are fed into a classifier and a regressor to predict each object's class label and bounding box coordinates. Another popular approach is the single-shot detector (SSD), which directly predicts each object's class label and bounding box coordinates without needing region proposals.

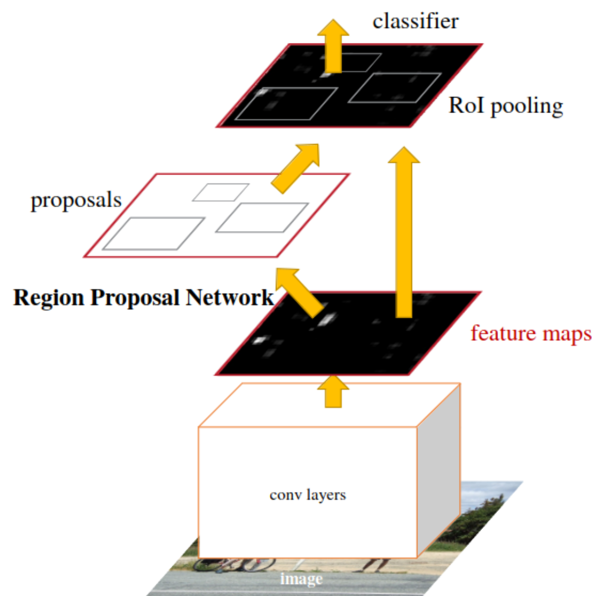

Figure 2.10. *Faster R-CNN network for object detection. [14]*

A new approach for object detection is You Only Look Once or YOLO, which uses a

single neural network and combines a multi-step process to perform both classification and prediction of bounding boxes for detected objects. By dividing the input image into a grid of cells, it predicts the class probabilities and bounding box coordinates for each object within these cells [15] as figure 2.11. The model also predicts the confidence score for each detection, reflecting the model's confidence. Its last layer of the network outputs a tensor containing the class probabilities and bounding box coordinates for each cell. One of the critical advantages of YOLO models is their speed. Since YOLO models make predictions for all objects in an image in one pass, they can achieve real-time performance even on low-end hardware.

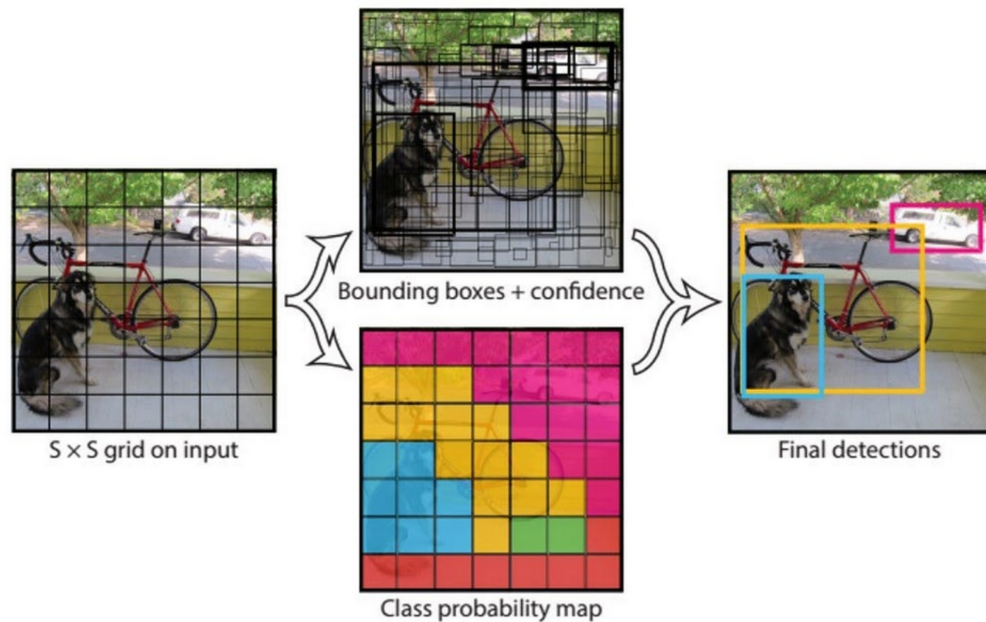

Figure 2.11. *The YOLO model system detection as a regression problem [15]*

Figure 2.12. *Object Detection Result From This Benchmark.*

2.4.3 Human Pose Estimation

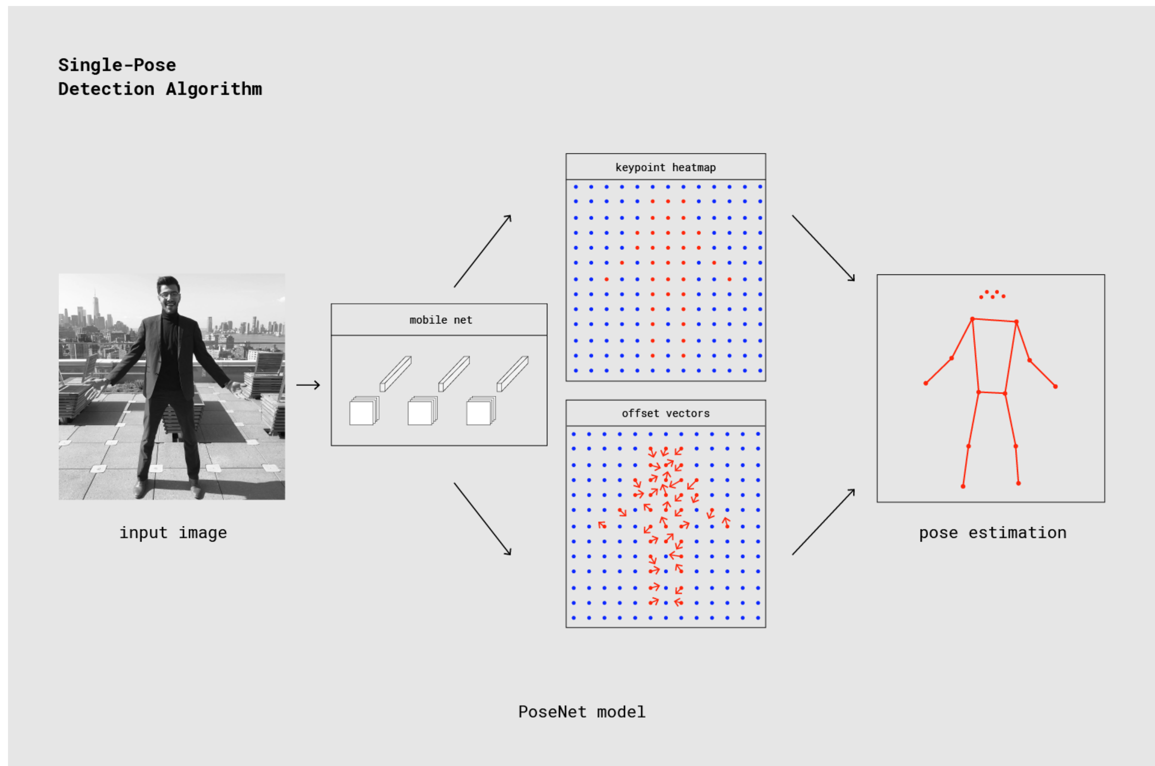

Figure 2.13. *Single Pose Detection Algorithm. [16]*

Figure 2.14. *Human Poses Result From This Benchmark.*

Human pose estimation identifies and captures the pose of human body parts like the wrist, shoulder, knees, eyes, ears, and arms as key points and represents them in 2D/3D space. Different models have different numbers of keypoints; for example, the COCO model has 18, while MPII uses 15. Computers can now understand human body language by detecting and tracking pose. Generally, the human bodies are represented in 2D and 3D planes through 3 main types of models, as figure 2.15.

This benchmark's lightweight human pose estimation model belongs to the kinematic type. The output of a typical human pose estimation model includes arrays of offset vectors of keypoints, a heat map, and a Part Affinity map. The heat map is a 2D image that reflects the likelihood of a joint being present at a specific location in an input image.

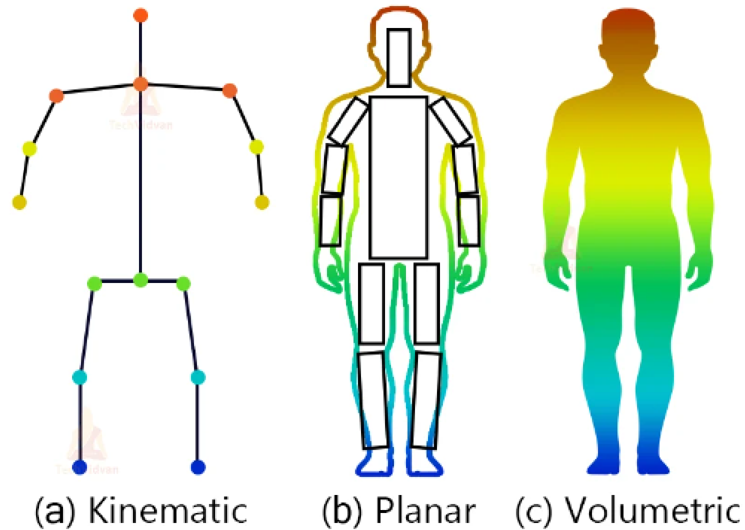

Figure 2.15. *3 types of human poses. [17]*

Each pixel in the heat map represents a location on the image, with its value indicating the probability that a joint is present at that spot. Meanwhile, the offset vector is a 2D vector that describes the displacement from the center of a bounding box to the precise location of a joint within that box. These vectors can be used to refine the area of the joints based on their estimated offsets. By combining the heat map and offset vectors, the model can accurately estimate the location of the keypoints in an image with high confidence. Ultimately, these indexes are decoded to identify the areas in the image with the highest likelihood of containing keypoints, as presented in figure 2.13.

2.4.4 Semantic Segementation

Semantic segmentation furthers image classification by treating multiple objects of the same classes or types into one entity. In other words, semantic segmentation might indicate pixel boundaries of the same object. Semantic segmentation aims to segment an image into meaningful regions, such as objects or parts of objects. Semantic segmentation is a challenging task due to its large amount of detail and fine-grained features, such as texture and shape.

A widely-used method for semantic segmentation is based on the fully convolutional network (FCN) models [18]. In this approach, the input images are first processed by the FCN's encoders, which apply a series of convolutions to reduce their size while increasing the number of channels. The encoded output is then upsampled using bilinear interpolation or a series of transpose-convolution operations. The FCN architecture is adapted to produce a dense per-pixel output that assigns a probability to each pixel for each possible class. This output takes the form of a probability map that provides a probability value for

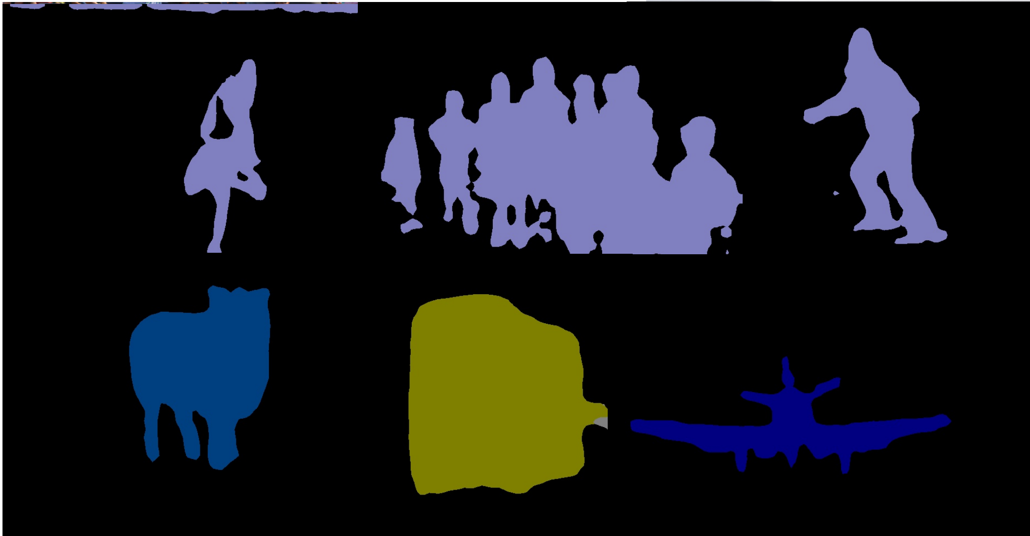

Figure 2.16. *Semantic Segmentation Result From This Benchmark.*

every pixel, indicating the likelihood of it belonging to a particular category. The benchmark model we used here **LRASPP_MobileNet_V3_Large** instantiates on this method.

Another popular approach is the U-Net architecture, which consists of a contracting path that captures the context of the input image and an expansive way that predicts the segmentation mask. U-Net models have achieved state-of-the-art performance on several semantic segmentation benchmarks.

3. METHODS

We first study the characteristics of devices used in this thesis. Raspberry Pi is a **single-board computer (SBC)**, a small powerful high-end device that is adapted to being used in a more versatile and wide range of applications from the proliferation of IoT sensors' controllers to latency-sensitivity computing paradigm like emergency-alert system, cognitive-recognition [19], human gait study [20] and augmented reality devices. With respect to RPi4, RPi 3 has limited computing capability considering data-extensive required applications. For inferencing DL tasks, RPi4, however, still relies on the CPU, which is not specialized for intensive data processing, to run simple small AI models. Therefore, it is often referred to as **General-purpose Edge Device**. In contrast, **GPU-based edge devices**, like Jetson family devices, are designed for high-performance computing tasks, particularly involving graphic processing such as image and video processing. These devices are optimized for deep learning models and neural networks using GPU or TPU for acceleration. Such characteristics and overview of these devices are described in table 3.1.

Category	IoT/Edge Devices		GPU-Based Edge Devices		
Platform	Raspberry Pi 3B (v1.2)	Raspberry Pi 4	Jetson Nano	Jetson TX2	Jetson AGX Xavier
Operating System	Raspbian Bullseye (Debian 11)	Raspbian Bullseye (Debian 11)	Ubuntu Bionic 18.04	Ubuntu Bionic 18.04	Ubuntu Bionic 18.04
CPU	4-core BCM2837 64bit @1.2GHz	4-core BCM2711 64bit @1.8GHz	4-core Cortex-A57	4-core Cortex-A57 2-core NVIDIA Denver2 64bit	8-core ARMv8.2 64bit @2.26GHz
GPU	N/A	N/A	128-core Pascal μA	256-core Pascal μA	512-core Volta 64 Tensor Cores
Memory	1GB LPDDR2	4GB LPDDR4	4GB LPDDR4	8GB LPDDR4	16GB LPDDR4

Table 3.1. *Overview of edge devices used in this thesis.*

Due to the constraint of Raspberry Pi 3, we choose simple small-sized and mobile-designed pre-trained models from 4 computer vision applications (image classification, object detection, human pose estimation, and semantic segmentation) for this benchmark.

The table 3.2 summarizes the different numbers of FLOPS, layers, and parameters between these models. Most of the models are taken from Pytorch framework (**torchhub**) and converted into ONNX format for deployments as figure 2.6. On the one hand, IMAGENET1K is a large-scale image recognition dataset that comprises over 1 million images, each labeled with one of 1000 object categories. It is widely used for training and evaluating deep neural networks for image classification tasks. On the other hand, COCO is a different dataset that targets object detection, instance segmentation, and image captioning tasks. Images in COCO are annotated with bounding boxes, segmentation masks, and captions. Similarly, PASCAL VAC (Visual Object Classes) is another dataset that includes annotated images with object class labels, object bounding boxes, and segmentation masks.

The ONNX Inference Runtimes provide a platform for inferencing ONNX models on diverse operating systems, chip architectures, and hardware accelerators, referred to as **Execution Providers**. However, the focus of this thesis will be limited to examining CPU and CUDA accelerators exclusively. Furthermore, specific models undergo conversion

to TensorRT, a commonly utilized high-performance GPU inference framework designed for NVIDIA hardware. It is noticeable that the Tensor engine file is hardware-dependent, which implies that TensorRT is tailored to the specific version (both major and minor) of the framework used to create them and the GPU on which they were generated.

Table 3.2. *Overview of DL models, * denotes model file from OpenVINO toolkit.*

Model Name	Input size (pixels)	Params (M)	GFLOPs (ops)	File Size (MB)	Trained dataset
AlexNet [21]	224	61.1	4.3	233.1	IMAGENET1K
DenseNet121 [21]	224	8	2.83	30.8	IMAGENET1K
EfficientNet_B0 [21]	224	5.3	0.39	20.5	IMAGENET1K
GoogLeNet [21]	224	6.6	1.5	49.7	IMAGENET1K
Inception_V3 [21]	299	27.2	5.71	103.9	IMAGENET1K
MNASNet0_5 [21]	224	2.2	0.1	8.6	IMAGENET1K
MNASNet0_75 [21]	224	3.2	0.21	12.3	IMAGENET1K
MNASNet0_1 [21]	224	4.4	0.31	16.9	IMAGENET1K
MNASNet1_3 [21]	224	6.3	0.53	24.2	IMAGENET1K
MobileNet_V2 [21]	224	3.5	0.3	13.6	IMAGENET1K
MobileNet_V3_Large [21]	224	5.5	0.22	21.1	IMAGENET1K
MobileNet_V3_Small [21]	224	2.5	0.06	9.8	IMAGENET1K
ShuffleNet_V2_X0_5 [21]	224	1.4	0.05	5.3	IMAGENET1K
SqueezeNet1_0 [21]	224	1.2	0.82	4.8	IMAGENET1K
ResNet18 [21]	224	11.7	1.81	44.7	IMAGENET1K
SSDLite320_MobileNet_V3_Large [22]	320	3.4	0.85	13.4	COCO
tinyYOLOv2* [23]	416	11.2	5.424	42.9	Pascal VOC
tinyYOLOv3* [24]	416	8.84	5.58	33.8	COCO
yolv5n [25]	640	1.9	4.5	3.9	PASCAL VOC
yolov5n6[25]	1280	3.2	4.6	6.9	PASCAL VOC
yolov5s [25]	640	7.2	16.5	14.1	PASCAL VOC
yolov5s6 [25]	1280	12.6	6.8	24.8	PASCAL VOC
yolox-Nano [26]	416	0.91	1.08	7.3	COCO
yolox-Tiny [26]	416	5.06	6.45	38.9	COCO
LRASPP_MobileNet_V3_Large [27]	520	3.2	2.09	12.5	COCO
Light_Weight_Human_Pose [28]	256	4.1	9	83.9	COCO

Numerous machine learning model deployment tools are widely recognized and used in both academic and industrial settings, including MLflow [29], BentoML [30], MMDetec-

tion [31]. Moreover, various methods and approaches have been explored to adapt deep learning models to devices with limited resources, such as quantization [32] and compression [33] or different light-weight frameworks have been introduced, e.g., MNN, TF Lite. Inspired by these setups, we have created small-scale deployment tools to evaluate the performance and ability to adapt to different environments and devices. Our approach involves incorporating custom ONNX from various sources, along with pre-trained models. Specifically, the tool is designed to change the opset version back to 13 if its version is higher and convert int32 when building the TensorRT engine for homogeneous purposes. Besides, the deployment code is written to enable easy scaling up and updating. The benchmark procedure is illustrated in figure 3.1 illustrate, and we can include additional models to benchmark by adding their configuration to file *modelsConfig.py* and rerun the script *convert.sh*.

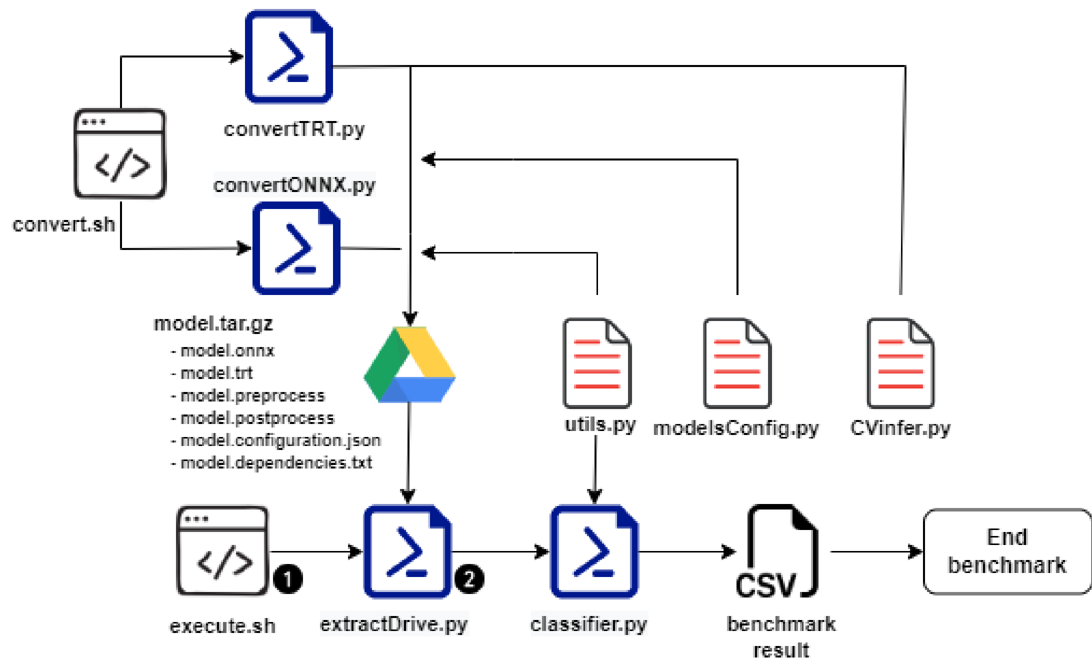

Figure 3.1. *Benchmark procedure.*

When executed, the *convert.sh* script will convert the models into a zipped file containing all artifact files and save it on Google Drive. When running the *execute.sh* script on the benchmarking device, all required files will be downloaded, the inference process will begin, and all results will be saved locally. An example of the execution is shown in figure 3.2

Figure 3.2. Screenshot of run procedure on devices.

The author of this thesis aims to establish an identical environment setup for all devices. All devices will be connected to the same WiFi router via ethernet ports while disconnecting from peripherals such as GPIO, keyboard, and monitor. A configuration file will store the IP addresses to enable well as the package versions generated by pip. The primary laptop will be the primary controller in this setup as Figure 3.3.

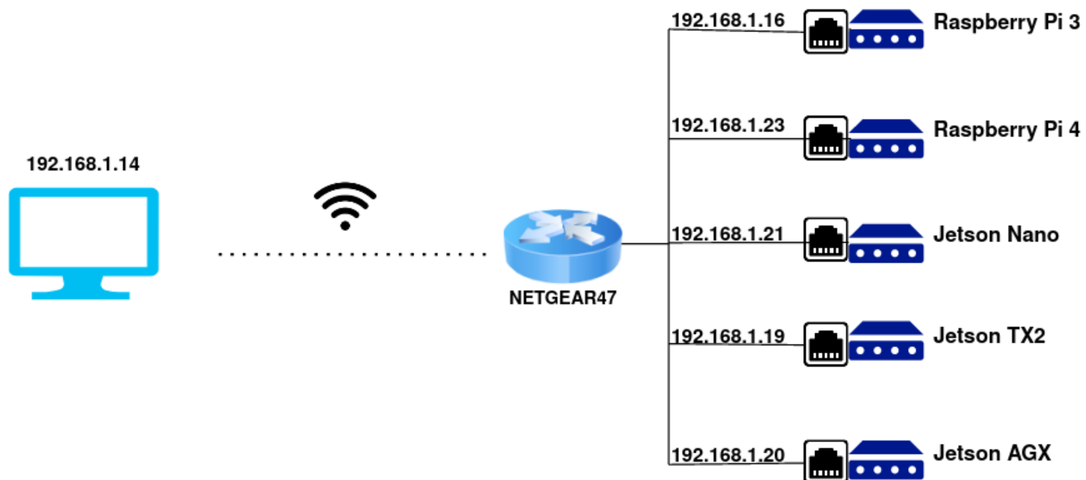

Figure 3.3. Experiment setup diagram.

The experiment comprises two parts: measuring with and without the cache's warm-up. To elaborate, the models are provided with 100 dummy inputs or images to load the cache before actual inference benchmarking for the cache warm-up. Additionally, there will be a 10-minute rest period between each model's runtime to reduce heat from the CPU/GPU.

Only two cases (in CPU) are demonstrated for Raspberry Pi 3/4, while four cases are presented for the Jetson family. Moreover, a reset will occur before each application runs to clear cache memory and background processes. However, in the case of no-cache loading, there will be no rest time between models, and they will run straight through. Finally, the results are compared and illustrated in chapter 4

4. EVALUATION

In the previous chapter, the author already presents the experiment setup and benchmark processes. The discussion and elaboration of the results will be presented in this chapter. The models have been inferred from 5000 images from the COCO dataset, allowing the sequence inference time to provide information about each image. This thesis does not measure accuracy, but several steps are defined to ensure the model functions appropriately. For instance, when converting to the ONNX model, a function is limited to verify the result. This function has two test cases: one for randomly generated input and another for 20 test images taken from the COCO dataset (excluding the 5000 images used for the benchmark). Additionally, a test file will download the zip file from our drive and verify the result to ensure that links are loaded correctly. From this point onward, unless explicitly specified, the terms CPU and GPU/CUDA will be used interchangeably to refer to the ONNX execution providers running on either the CPU or GPU. All tables' reported average frames per second (FPS) account for the entire process, including preprocessing, inference, and postprocessing time. The models that achieved the highest FPS are highlighted in bold.

4.1 Image Classification

Initially, we execute ONNX models for image classification, and then we analyze the resulting patterns across different devices, which are demonstrated in the upcoming tables and figures.

The tables 4.1 and 4.2 demonstrate image classification models running on the ONNX CPU execution provider. It can be seen that **Shufflenet_v2_x0_5** achieve the highest frames per second with 14.2 and 30.9 on both pi3 and pi4, respectively. In figures 4.1 and 4.2, it has been observed that the mean difference in inference time standard deviation for most models between Raspberry Pi 4 and Raspberry Pi 3 is 1.476%. However, the **Shufflenet_v2_x0_5** model has a much smaller mean difference of only 0.189%.

Table 4.1. *Image classification models inference result on Raspberry Pi 3.*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Alexnet	41.624	274.035	0.599	3.168
Googlenet	46.938	416.764	0.641	2.160
Efficientnet_b0	43.403	428.874	0.603	2.119
Densenet121	48.296	966.394	0.658	0.985
Squeezenet1_0	44.664	239.799	0.644	3.518
Inception_v3	60.985	1577.825	0.670	0.610
Shufflenet_v2_x0_5	**7.913**	**32.321**	**0.552**	**14.208**
Mobilenet_v2	42.259	222.914	0.601	3.775
Mobilenet_v3_small	39.559	87.033	0.571	7.896
Mobilenet_v3_large	41.074	216.333	0.588	3.885
Mnasnet0_5	40.334	132.673	0.578	5.785
Mnasnet0_75	41.592	218.910	0.593	3.841
Mnasnet1_0	42.202	270.652	0.596	3.201
Mnasnet1_3	43.215	389.693	0.610	2.311
Resnet18	47.003	493.082	0.648	1.853

Table 4.2. *Image classification models inference result on Raspberry Pi 4.*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Alexnet	16.989	155.015	0.308	5.804
Googlenet	16.865	231.728	0.308	4.018
Efficientnet_b0	16.830	200.982	0.309	4.585
Densenet121	16.926	519.852	0.311	1.862
Squeezenet1_0	16.915	131.797	0.341	6.710
Inception_v3	21.637	887.624	0.310	1.099
Shufflenet_v2_x0_5	**16.827**	**15.268**	**0.309**	**30.926**
Mobilenet_v2	16.845	100.459	0.308	8.503
Mobilenet_v3_small	16.845	36.336	0.307	18.711
Mobilenet_v3_large	16.841	95.321	0.307	8.892
Mnasnet0_5	16.887	54.421	0.309	13.970
Mnasnet0_75	16.908	91.436	0.310	9.205
Mnasnet1_0	17.200	117.334	0.311	7.417
Mnasnet1_3	16.962	169.964	0.309	5.341
Resnet18	16.844	262.382	0.313	3.578

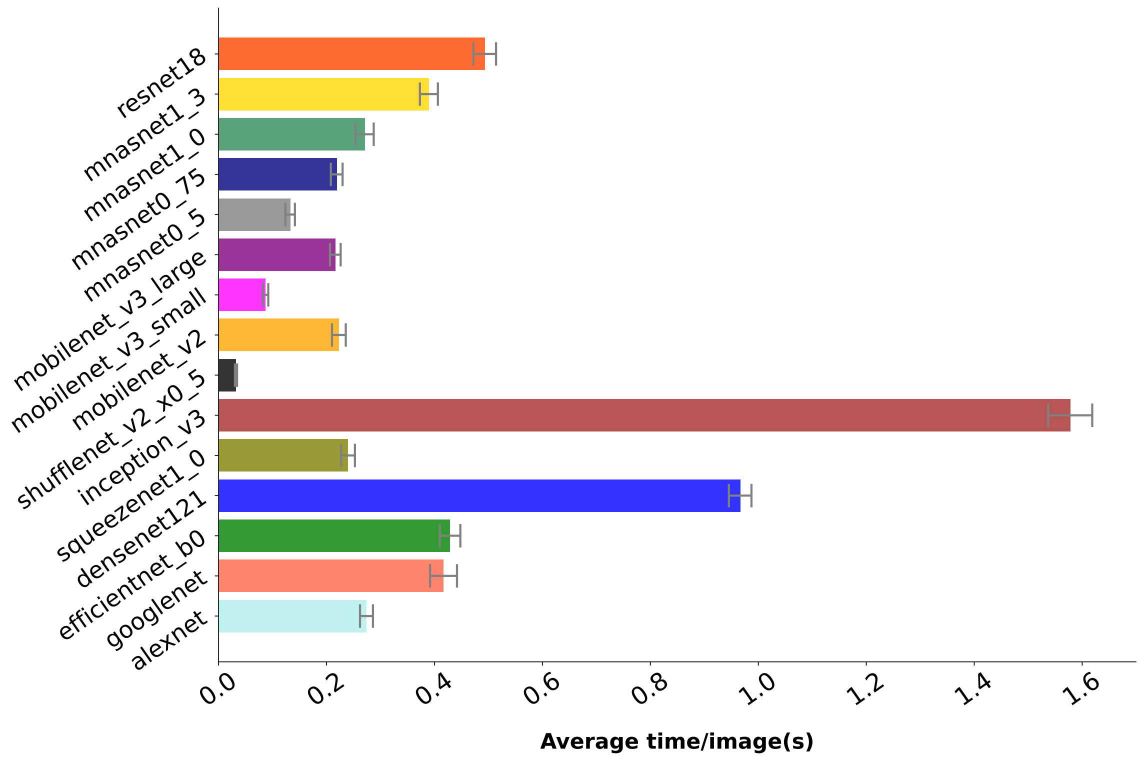

Figure 4.1. *Inference time of image classification models on Pi 3 (CPU).*

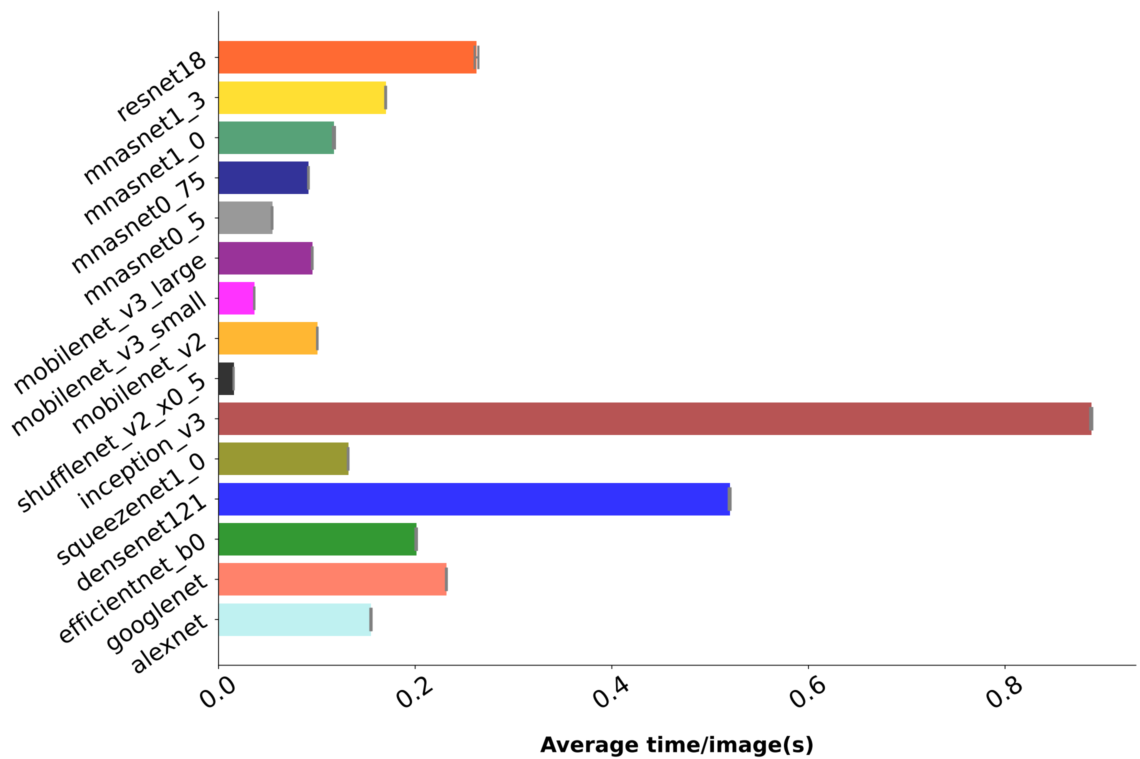

Figure 4.2. *Inference time of image classification models on Pi 4 (CPU).*

The next 4 figures (figure 4.3 - figure 4.6) demonstrate that the inference time for 5000 images is more consistent on Raspberry Pi 4 than Raspberry Pi 3. This can be attributed

to the difference in hardware capabilities of the two devices. Raspberry Pi 4 has a more powerful processor and higher RAM, allowing it to perform computations faster and more efficiently. This increased processing power results in faster inference times and lower standard deviations. In addition, the better thermal management of Raspberry Pi 4 might prevent the processor from overheating and slowing down during prolonged use, providing an additional advantage.

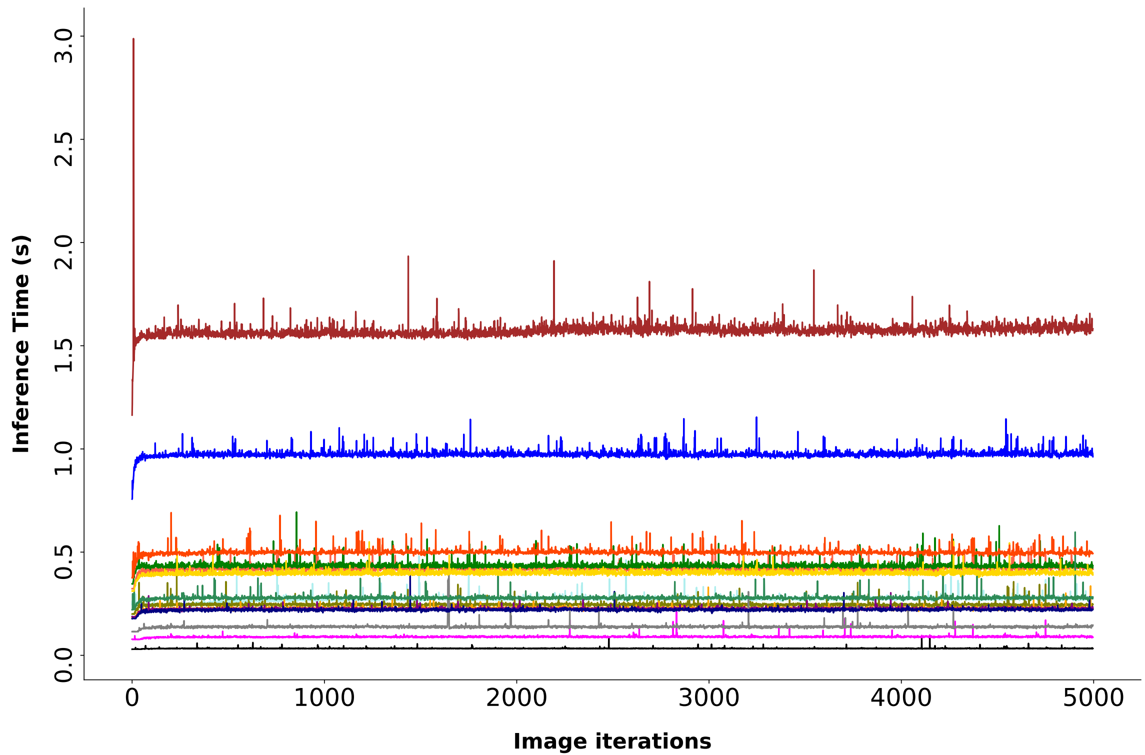

Figure 4.3. *No-warmup image classification models on Pi 3 (CPU).*

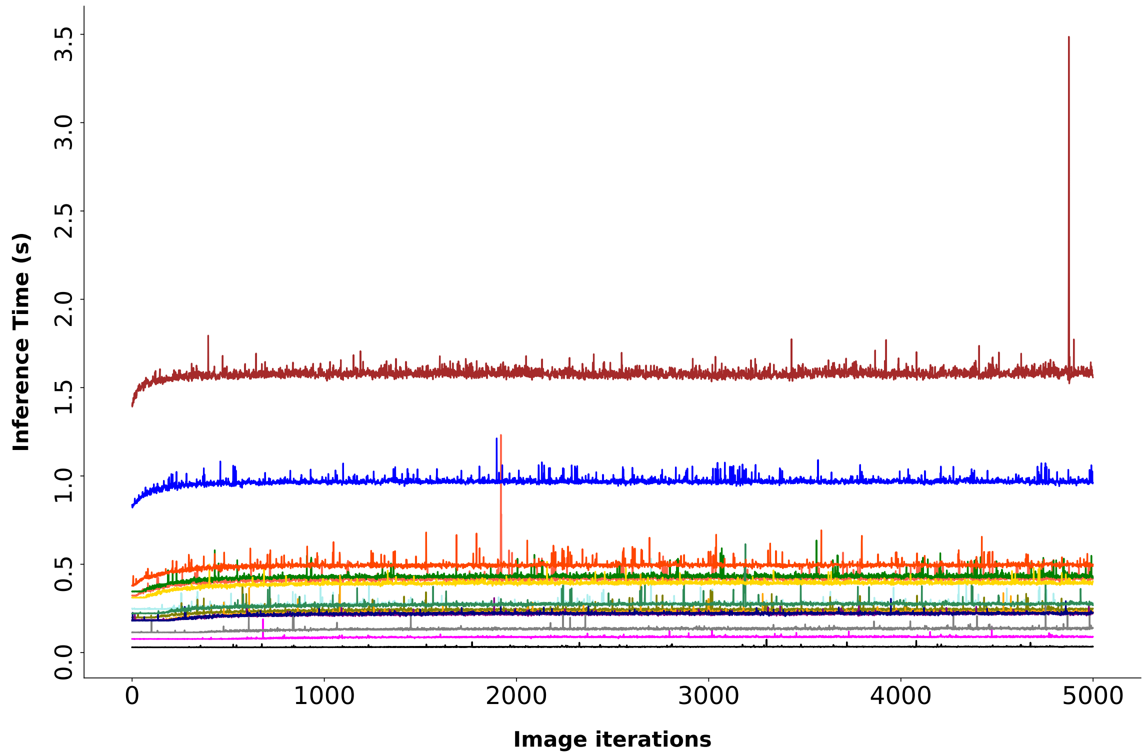

Figure 4.4. *Warmup image classification models on Pi 3 (CPU).*

One additional point to consider is the presence of spices at the beginning of some no-warmup models, particularly in large models like **inception_v3** for Pi devices. This phenomenon occurs because the models have not yet been adapted to the specific task when benchmarking starts. In other words, the model may produce suboptimal results at the beginning of the benchmark because it is still "learning" and "adjusting" its parameters to better fit the data. These "spiky" or inconsistent results are not very clear for Pi 4, 4.5, 4.6, Jetson TX2, and Jetson Nano. However, they are more significant for Jetson Xavier, especially in large models like **inception_v3** and **densenet121**, as shown in figure 4.10.

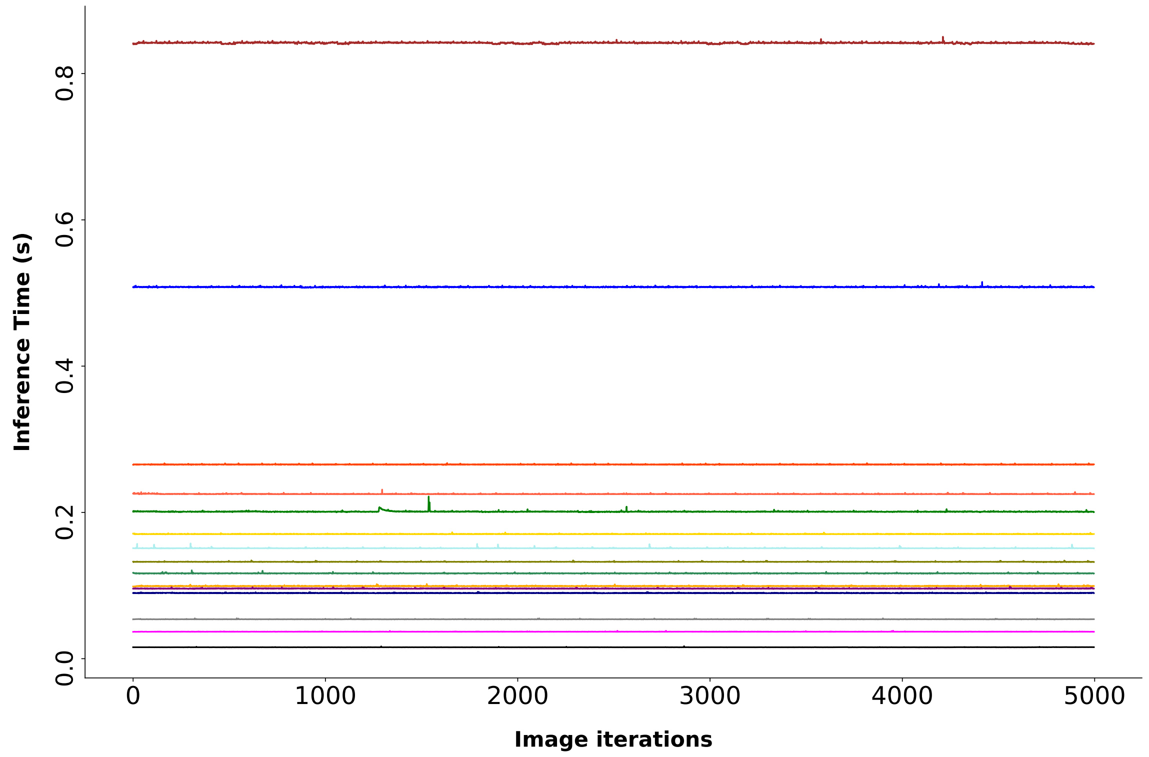

Figure 4.5. *No-warmup image classification models on Pi 4 (CPU).*

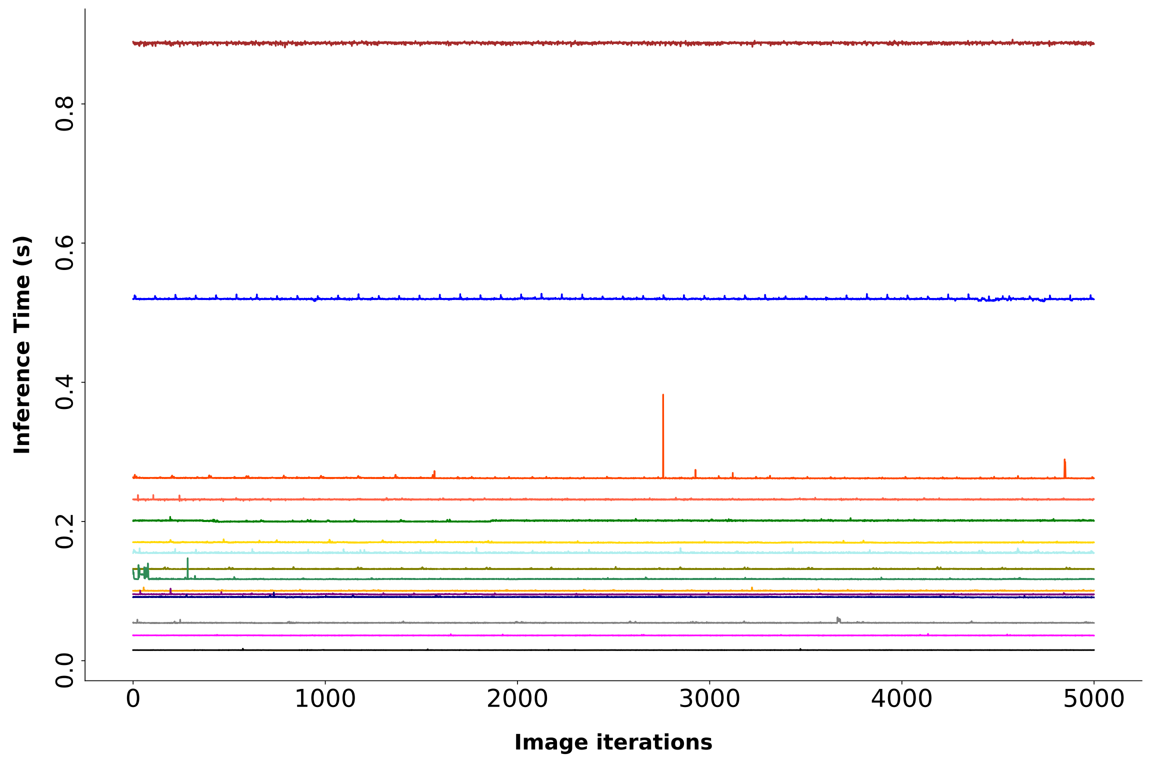

Figure 4.6. *Warmup image classification models on Pi 4 (CPU).*

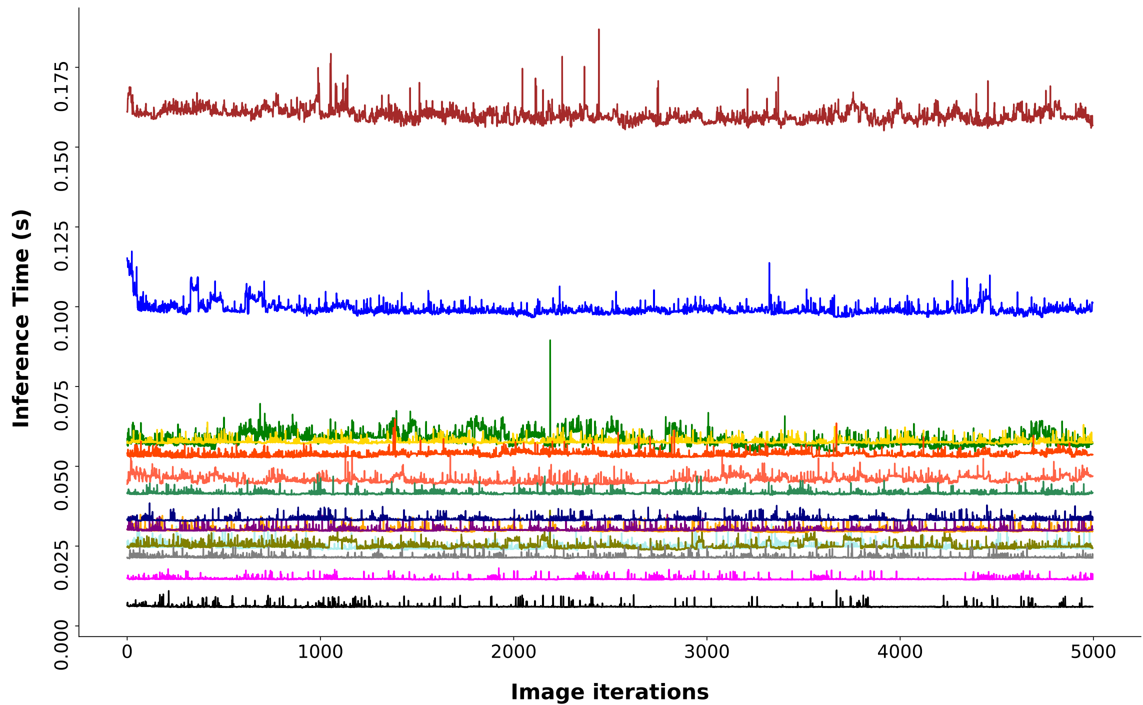

Figure 4.7. *No-warmup image classification models on Jetson Xavier.*

The "spice effect" on Raspberry Pi 3 behaves quite differently than other devices. For instance, on devices like the Jetson Xavier (as seen in figure 4.10), the inference time initially spikes at the higher beginning point than the mean and then gradually decreases over time. However, on the Raspberry Pi 3, the inference time initially spikes at the beginning lower point and continues to soar. One possible reason for the difference in performance between the Raspberry Pi 3 and the Jetson Xavier when running a classification model is that the Raspberry Pi 3 may require additional memory and resources during the initial run to load the model's parameters from the disk. This initial loading process may result in slower inference times compared to the Jetson Xavier, which has a more powerful processor and greater memory capacity, allowing it to allocate the necessary resources more efficiently. However, after running the model several times, the Raspberry Pi 3's memory cache becomes more optimized for the model's parameters, leading to more stable inference times. Alternatively, the Raspberry Pi 4 has several enhancements over the Raspberry Pi 3 that enable it to achieve more consistent inference times without requiring a warm-up period.

As mentioned in chapter 3, we run 100 images for cache warmup before benchmarking. However, it can be seen from figure 4.4 that raspberry pi 3 needs more than 100 images for the warmup processes to get rid of these spices completely.

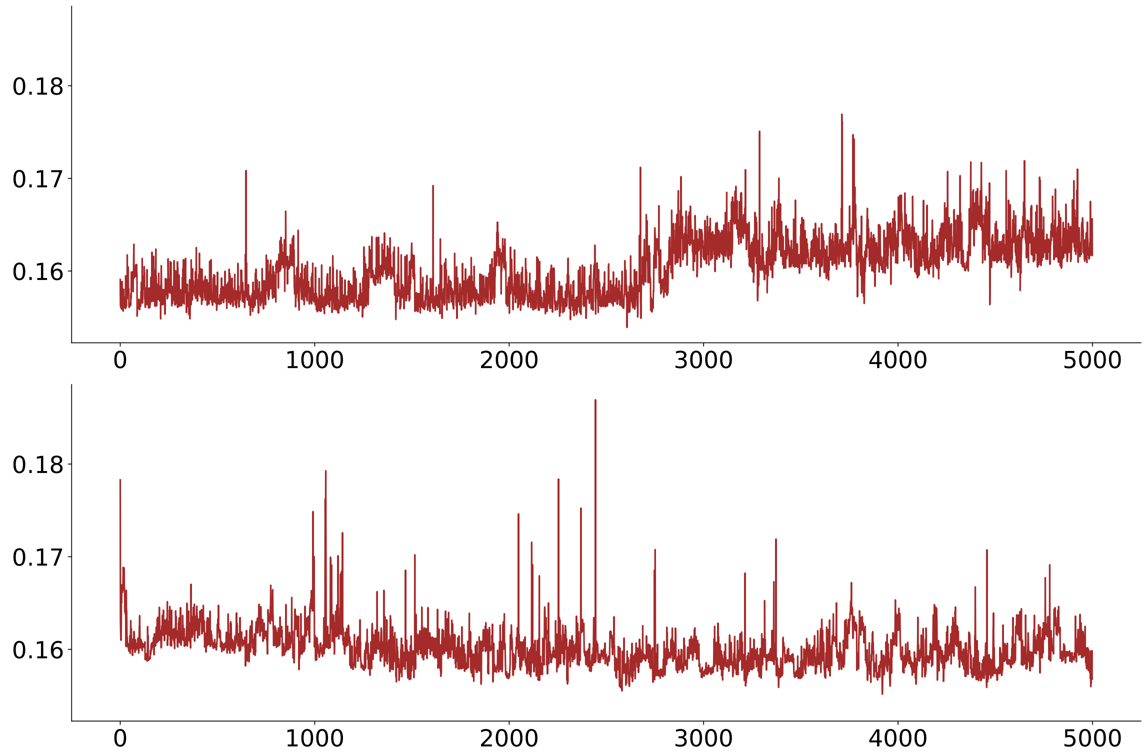

Figure 4.8. *Warmup and no warmup for model Inception_v3 on Jetson Xavier.*

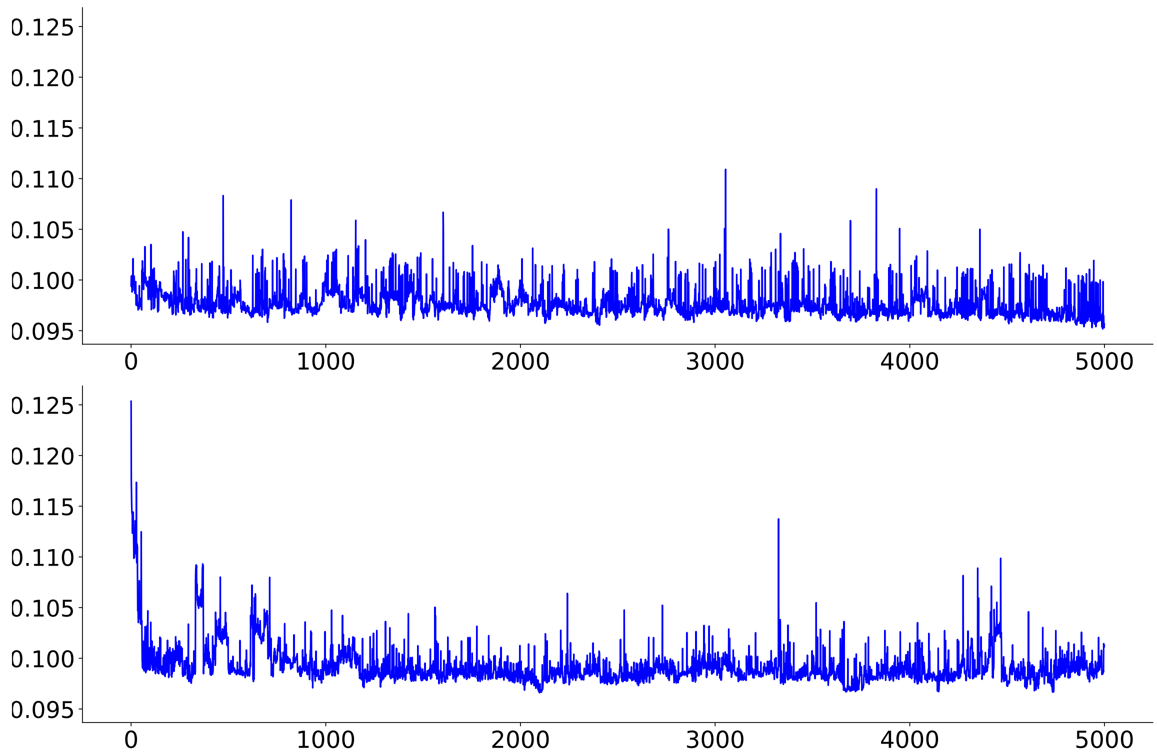

Figure 4.9. *Warmup and no warmup for model DenseNet121 on Jetson Xavier.*

Both figure 4.8 and 4.9 indicates that the difference in spice phenomenon compared to the mean in the DenseNet121 and Inception_v3 models is approximately 0.02 seconds.

However, the DenseNet121 model demonstrates a more stable inference process due to its architecture, and its fewer parameters mean that it requires less computation and memory.

The purpose of warmup is to allow models gradually adjust their parameters and biases based on tasks they will perform rather than start with random initialization or not well-suited tasks.

When comparing the Raspberry Pi 4 ARM Cortex-A72 processor to the one found in the Jetson Nano, it is clear that the Raspberry Pi has a slightly upgraded processor, as seen from the close average frame per second results in both tables 4.2 and 4.3. However, the GPU in the Jetson Nano outperforms the Pi 4 chip in the domain of video encoding and decoding, thus enhancing these performance tasks. Additionally, the Jetson Nano has to share the same memory between its CPU and GPU, which gives the Raspberry Pi an advantage regarding memory availability.

Table 4.3. *Image classification models inference result on Jetson Nano (CPU).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Alexnet	19.215	135.715	0.289	6.443
Googlenet	19.256	228.747	0.292	4.028
Efficientnet_b0	19.317	204.279	0.289	4.467
Densenet121	19.170	467.791	0.289	2.052
Squeezenet1_0	19.273	132.870	0.295	6.561
Inception_v3	25.944	841.300	0.287	1.153
Shufflenet_v2_x0_5	**19.301**	**16.260**	**0.288**	**27.955**
Mobilenet_v2	19.204	95.569	0.285	8.692
Mobilenet_v3_small	19.314	38.201	0.290	17.313
Mobilenet_v3_large	19.138	89.598	0.287	9.174
Mnasnet0_5	19.249	55.944	0.292	13.254
Mnasnet0_75	19.247	92.519	0.291	8.925
Mnasnet1_0	19.281	119.055	0.292	7.215
Mnasnet1_3	19.087	173.776	0.284	5.178
Resnet18	19.164	267.207	0.284	3.489

Table 4.4. *Image classification models inference result on Jetson Nano (CUDA).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Alexnet	19.218	29.528	0.602	20.299
Googlenet	19.275	36.249	0.483	17.872
Efficientnet_b0	19.313	39.926	0.417	16.776
Densenet121	19.254	83.473	0.468	9.693
Squeezenet1_0	19.277	25.779	0.489	21.989
Inception_v3	26.076	182.877	1.262	4.757
Shufflenet_v2_x0_5	**19.317**	**22.131**	**0.345**	**24.079**
Mobilenet_v2	19.218	26.054	0.485	22.375
Mobilenet_v3_small	19.296	19.942	0.430	25.294
Mobilenet_v3_large	19.277	22.665	0.430	23.629
Mnasnet0_5	19.268	22.567	0.376	23.732
Mnasnet0_75	19.232	23.097	0.370	23.469
Mnasnet1_0	19.227	26.655	0.400	21.651
Mnasnet1_3	19.232	36.145	0.490	17.912
Resnet18	19.234	30.677	0.448	19.882

The table 4.4 and 4.6 indicates that running with CUDA execution provider can vastly accelerate the inference process. Most of the models are 3-4 times faster when running with GPU, especially large models like Resnet18. In constrast, **Shufflenet_v2_x0_5** ONNX model does not benefit from running with GPU, which achieves around 24 FPS compared to nearly 28 FPS on CPU, due to its architecture.

Table 4.5. *Image classification models inference result on Jetson TX2 (CPU).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Alexnet	14.528	115.703	0.293	7.663
Googlenet	14.525	179.667	0.292	5.143
Efficientnet_b0	14.527	153.314	0.293	5.948
Densenet121	14.365	371.680	0.293	2.588
Squeezenet1_0	14.851	103.662	0.292	8.422
Inception_v3	19.099	644.653	0.295	1.506
Shufflenet_v2_x0_5	**15.341**	**12.710**	**0.295**	**35.656**
Mobilenet_v2	14.483	72.590	0.291	11.450
Mobilenet_v3	15.537	27.715	0.295	23.089
Mobilenet_v3_large	14.908	69.467	0.297	11.816
Mnasnet0_5	15.762	43.192	0.273	16.920
Mnasnet0_75	14.496	68.457	0.293	12.029
Mnasnet1_0	14.764	86.599	0.295	9.842
Mnasnet1_3	15.076	128.970	0.295	6.930
Resnet18	14.588	204.559	0.296	4.558

Table 4.6. *Image classification models inference result on Jetson TX2 (CUDA).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Alexnet	14.291	18.963	0.631	29.698
Googlenet	14.375	24.029	0.478	26.171
Efficientnet_b0	14.383	23.914	0.507	25.938
Densenet121	14.400	53.502	0.537	14.808
Squeezenet1_0	14.357	18.722	0.536	30.274
Inception_v3	19.299	105.940	0.790	8.004
Shufflenet_v2_x0_5	**14.811**	**17.700**	**0.879**	**30.054**
Mobilenet_v2	14.367	21.069	0.658	27.748
Mobilenet_v3_small	15.116	15.906	0.855	31.526
Mobilenet_v3_large	14.379	20.573	0.785	28.055
Mnasnet0_5	15.605	18.041	0.702	29.251
Mnasnet0_75	14.423	21.655	0.837	27.144
Mnasnet1_0	14.289	21.459	0.723	27.505
Mnasnet1_3	14.409	22.445	0.814	26.583
Resnet18	14.327	20.951	0.858	27.731

To clarify, most models running on the Jetson TX2 GPU can achieve nearly real-time performance at around 30 frames per second (FPS). Similarly, the Jetson Xavier running

on its CPU can also achieve this level of performance.

Table 4.7. *Image classification models inference result on Jetson Xavier (CPU).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Alexnet	4.514	25.938	0.387	32.473
Googlenet	4.548	45.244	0.369	19.950
Efficientnet_b0	4.541	58.319	0.366	15.829
Densenet121	4.522	97.630	0.363	9.757
Squeezenet1_0	4.514	24.588	0.375	33.971
Inception_v3	6.624	160.159	0.378	5.984
Shufflenet_v2_x0_5	**4.346**	**5.942**	**0.322**	**94.441**
Mobilenet_v2	4.300	30.280	0.367	28.631
Mobilenet_v3_small	4.420	15.546	0.345	49.280
Mobilenet_v3_large	4.302	31.224	0.371	27.865
Mnasnet0_5	5.089	20.218	0.321	39.106
Mnasnet0_75	4.292	32.988	0.352	26.582
Mnasnet1_0	4.528	41.145	0.364	21.731
Mnasnet1_3	4.333	57.146	0.359	16.176
Resnet18	4.545	54.461	0.378	16.846

Table 4.8. *Image classification models inference result on Jetson Xavier (CUDA).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Alexnet	4.649	16.037	0.459	47.381
Googlenet	4.530	14.117	0.391	54.288
Efficientnet_b0	4.628	14.794	0.389	51.908
Densenet121	4.486	19.766	0.424	40.606
Squeezenet1_0	4.485	11.933	0.393	59.784
Inception_v3	6.151	29.822	0.440	27.469
Shufflenet_v2_x0_5	**4.489**	**7.636**	**0.315**	**83.623**
Mobilenet_v2	4.386	10.883	0.372	69.463
Mobilenet_v3_small	4.437	7.578	0.346	82.311
Mobilenet_v3_large	4.379	10.822	0.370	65.857
Mnasnet0_5	4.640	9.302	0.379	70.439
Mnasnet0_75	4.373	9.851	0.386	71.236
Mnasnet1_0	4.530	13.445	0.398	54.531
Mnasnet1_3	4.409	11.331	0.379	64.857
Resnet18	4.500	19.245	0.463	41.361

Table 4.9. *Image classification models inference result on Jetson Xavier (TensorRT).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Alexnet	4.520	13.161	0.321	55.625
Googlenet	4.491	13.284	0.303	55.458
Efficientnet_b0	4.519	16.562	0.344	47.272
Densenet121	4.565	22.439	0.338	37.782
Squeezenet1_0	4.511	8.023	0.294	78.110
Inception_v3	6.270	20.100	0.341	38.298
Shufflenet_v2_x0_5	**4.431**	**4.057**	**0.300**	**114.293**
Mobilenet_v2	4.342	8.783	0.304	77.014
Mobilenet_v3_small	4.398	5.830	0.299	97.599
Mobilenet_v3_large	4.394	7.405	0.308	88.271
Mnasnet0_5	4.532	6.434	0.292	91.179
Mnasnet0_75	4.364	5.975	0.297	98.396
Mnasnet1_0	4.587	10.114	0.290	68.088
Mnasnet1_3	4.375	12.241	0.291	60.921
Resnet18	4.493	12.628	0.282	58.374

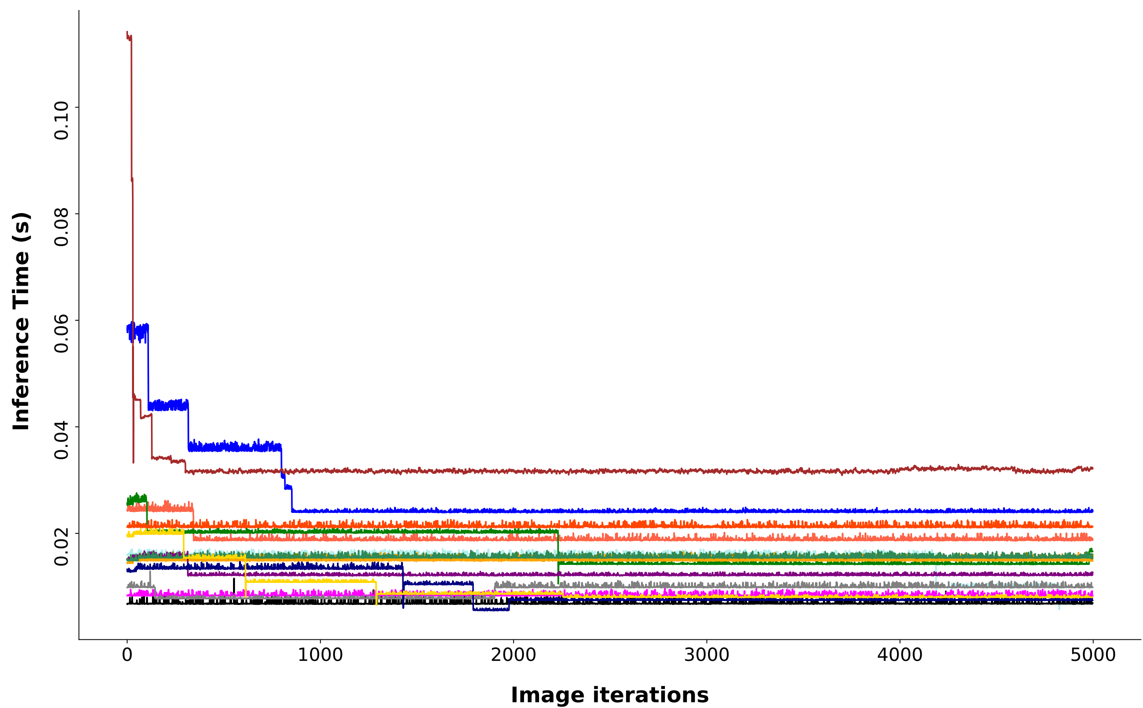

Figure 4.10. *No-warmup image classification models on Jetson Xavier (GPU).*

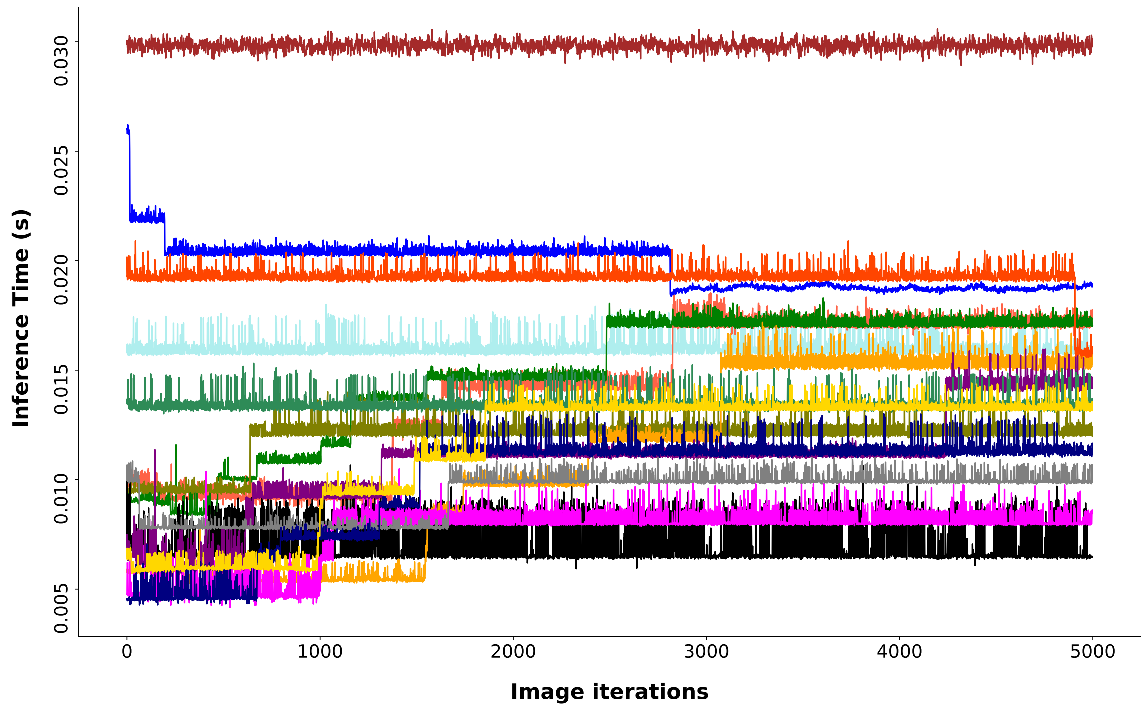

Figure 4.11. *Warmup image classification models on Jetson Xavier (GPU).*

Figure 4.10 and 4.11 reveal an interesting pattern in the "step pattern" for some mobile neural networks, such as mnasnet and mobilenet, when running CUDA execution provider on Jetson Xavier. This phenomenon is also observable in Jetson TX2. Specifically, when comparing warmup models to no-warmup models, there is an increasing step-up in inference time for warmup models on Jetson Xavier. In contrast, some no-warmup models exhibit a step-down. One possible explanation for these observations is that the Jetsons experiences thermal throttling, which result in the processor alternating between running at full speed and slowing down to regulate its temperature, ultimately slowing down to prevent overheating

Figure 4.12 depicts the distribution of the standard deviation of inference time for classification models on Jetson TX2 using boxplots.

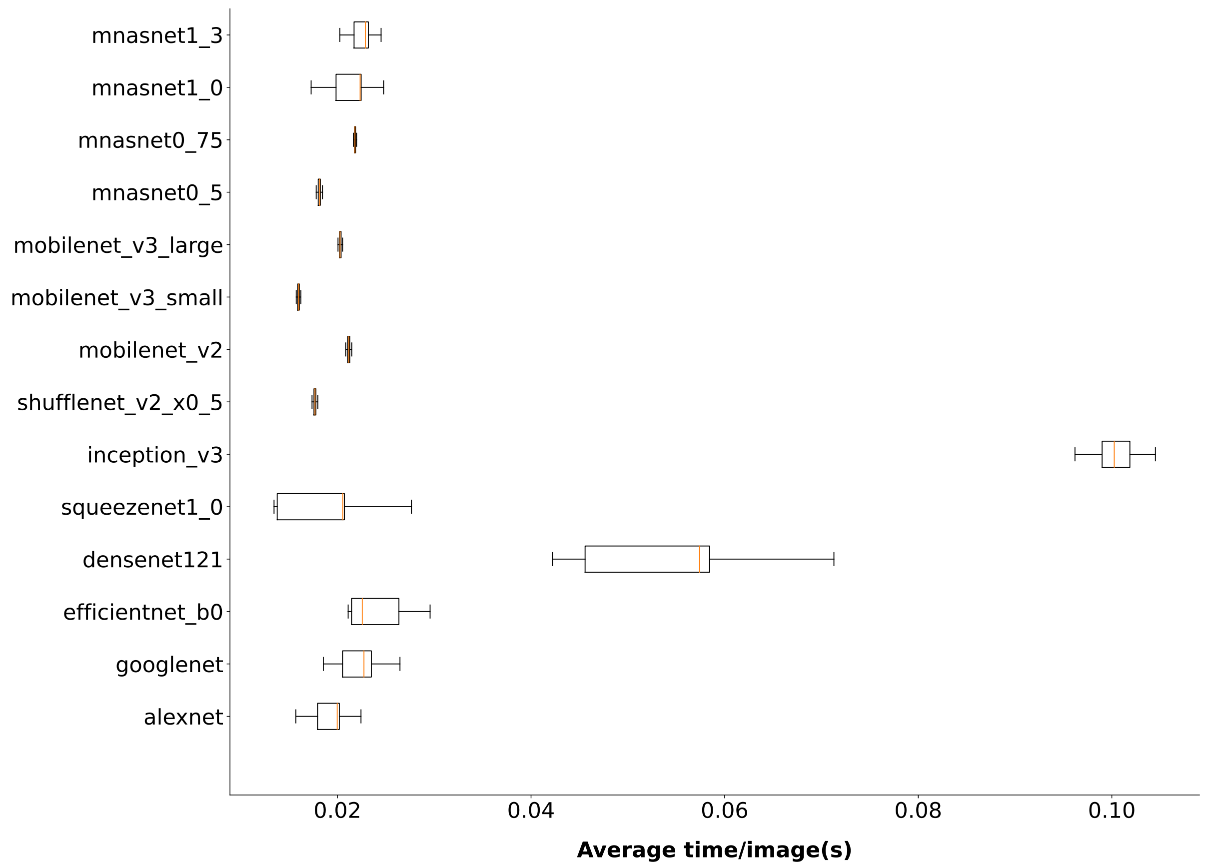

Figure 4.12. *Standard deviation inference time of image classification models on Jetson TX2 (GPU).*

The figure suggests that neural networks designed for mobile devices exhibit lower variability in inference time compared to state-of-the-art models like AlexNet and GoogLeNet, especially when running on GPU. However, this difference is not observed on the CPU.

Finally, only TensorRT models are executed on Jetson Xavier and demonstrated as figure 4.13.

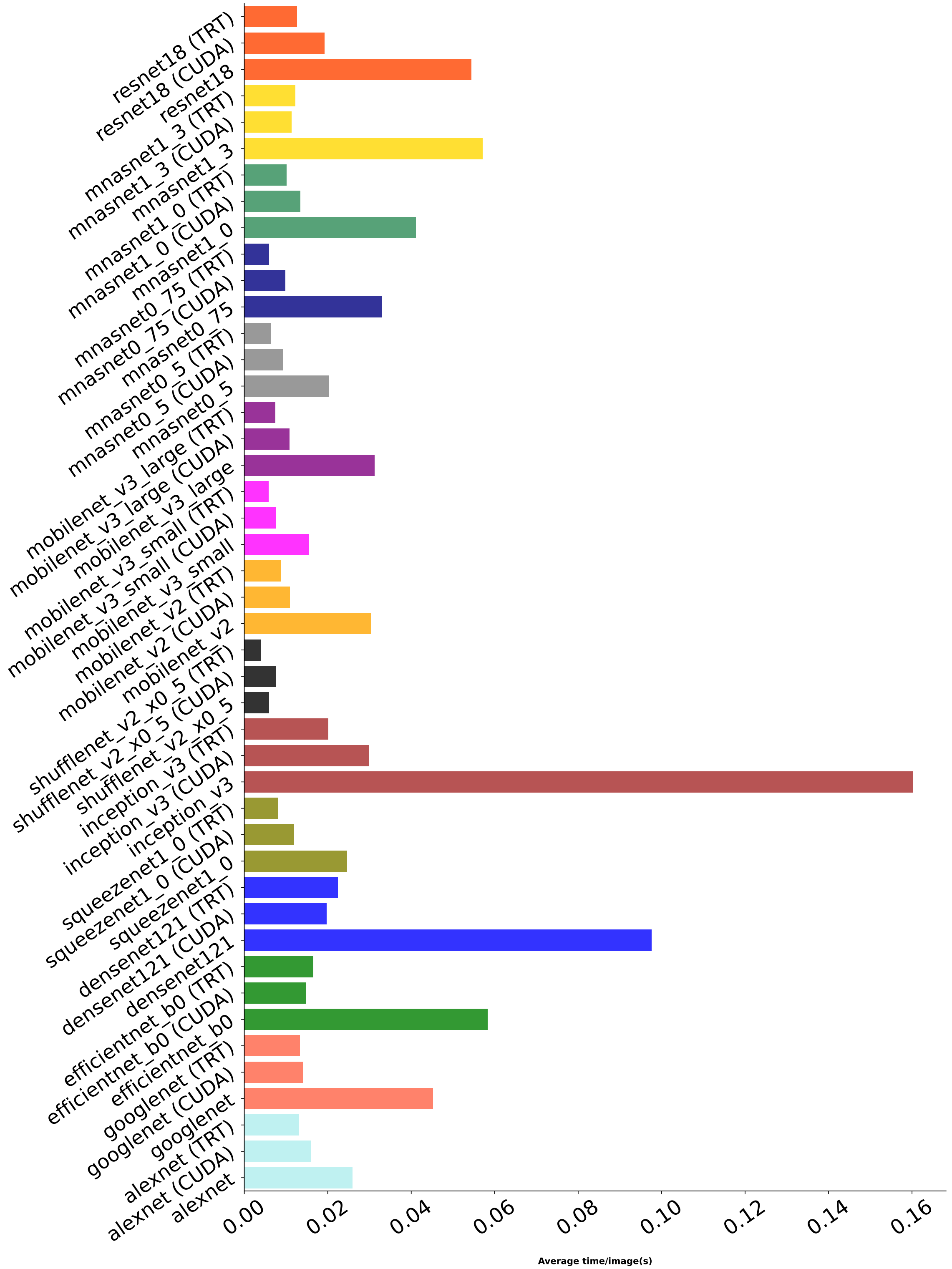

Figure 4.13. *Inference time of image classification models on Jetson Xavier (TensorRT).*

In addition to the benchmarking models we introduced earlier, we experimented with larger models containing more layers to determine how much they could benefit from TensorRT. We used different variations of ResNet and ResNext models with varying numbers of layers, cardinality, and network width to make our comparisons consistent. The

TensorRT engine was found to accelerate the inference time of most models significantly. From the results presented in table 4.10, it can be observed that the inference time of ResNet and ResNeXt models is inversely proportional to the number of layers, indicating that models with more layers experience a greater speed up when using TensorRT.

Table 4.10. *CUDA and TensoRT speed up for image classification models comparison.*

Model Name	GPU Speed Up (Times)	TensorRT Speed Up (Times)	Number of layers
Alexnet	1.617	1.971	8
Googlenet	3.205	3.406	22
Efficientnet_b0	3.942	3.521	215
Densenet121	4.939	4.351	121
Squeezenet1_0	2.061	3.065	18
Inception_v3	5.370	7.968	159
Shufflenet_v2_x0_5	0.778	1.464	69
Mobilenet_v2	2.782	3.448	18
Mobilenet_v3_small	2.051	2.666	17
Mobilenet_v3_large	2.885	4.216	25
Mnasnet0_5	2.173	3.143	17
Mnasnet0_75	3.349	5.521	21
Mnasnet1_0	3.060	4.068	23
Mnasnet1_3	5.044	4.669	25
Resnet18	2.830	4.313	18
Resnet50	4.594	6.095	50
Resnet101	**8.286**	**12.280**	**101**
Resnet152	**8.135**	**12.796**	**152**
Resnext50_32x4d	5.988	6.899	50
Resnext101_32x8d	9.090	11.478	101

For instance, ResNet101 and ResNet152 models can achieve up to **12.28** and **12.79** times faster inference times, respectively, compared to running on a CPU. In contrast, ResNet18 only experiences a speed-up of 4.3 times. The ResNeXt_101_32x8d also achieves a speed up of **11.47** times compared to ResNeXt50_32x4d, which only experiences a speed up of around 6.9 times.

However, DenseNet121 is an exception, as it achieves a faster inference time when running on a GPU, with a speedup of 4.9 times compared to TensorRT's 4.351 times. Additionally, EfficientNet and neural networks designed for mobile devices such as MobileNet_v2 and Mnasnet0_5 do not benefit significantly from using the TensorRT engine.

4.2 Object Detection

We tried to run a large object detection model (ssdlite320_mobilenet_v3_large) on raspberry pi 3 and 4, and it took more than 4 hours to infer 2500 images which are not applicable. Thus, we investigate YOLO object detection family models, and the results are summed up in next following tables.

Table 4.11. *Object Detection models inference result on Raspberry Pi 4.*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
tinyYOLOv2	8.877	521.919	130.187	1.513
tinyYOLOv3	21.525	417.499	2.656	2.264
yolov5n	8.996	634.392	15.537	1.519
yolov5n6	14.264	560.697	35.361	1.643
yolov5s	8.990	1436.215	11.419	0.687
yolov5s6	13.280	1490.459	49.739	0.644
yolox_nano	**4.366**	**217.079**	**6.268**	**4.392**
yolox_tiny	4.311	561.717	6.423	1.747
yolox_s	14.069	2185.671	17.666	0.451

Table 4.12. *Object Detection models inference result on Raspberry Pi 3.*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
tinyYOLOv2	28.120	852.261	361.271	0.806
tinyYOLOv3**	-	-	-	-
yolov5n	29.537	962.417	17.672	0.992
yolov5n6	38.500	975.024	85.725	0.916
yolov5s	32.039	2672.667	30.441	0.366
yolov5s6	41.984	2706.808	129.851	0.348
yolox_nano	**12.426**	**394.160**	**16.181**	**2.375**
yolox_tiny	13.875	1046.580	18.496	0.928
yolox_s	39.820	3948.442	51.590	0.248

tinyYOLOv3 in table 4.12 inference process was killed by the kernel due to insufficient memory despite we ran several times. yolox_nano achieves the highest performance over all devices with a maximum of 36 FPS when running on GPU on Jetson Xavier. It's worth noting that for jetson-family devices, only the inference time is accelerated, and not the pre and post-processing times.

Table 4.13. *Object Detection models inference result on Jetson Nano (CPU).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
tinyYOLOv2	10.064	517.452	146.291	1.485
tinyYOLOv3	24.207	413.046	2.439	2.276
yolov5n	9.022	526.215	6.565	1.848
yolov5n6	14.424	548.351	31.516	1.687
yolov5s	9.013	1420.627	10.420	0.695
yolov5s6	13.462	1476.449	44.284	0.653
yolox_nano	**4.018**	**218.423**	**6.601**	**4.367**
yolox_tiny	3.963	572.359	6.829	1.716
yolox_s	4.761	2149.700	12.889	0.462

Table 4.14. *Object Detection models inference result on Jetson Nano (GPU).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
tinyYOLOv2	10.039	68.002	147.409	4.438
tinyYOLOv3	24.184	103.632	2.457	7.700
yolov5n	9.027	108.765	6.529	8.087
yolov5n6	15.293	117.973	31.256	6.243
yolov5s	8.989	214.626	10.416	4.286
yolov5s6	14.352	224.544	43.402	3.598
yolox_nano	**3.966**	**46.261**	**6.919**	**17.511**
yolox_tiny	4.015	82.275	8.654	10.540
yolox_s	4.907	244.337	15.482	3.778

Table 4.15. *Object Detection models inference result on Jetson TX2 (CPU).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
tinyYOLOv2	7.840	390.402	115.501	1.947
tinyYOLOv3	18.779	309.998	1.941	3.024
yolov5n	6.758	409.145	5.060	2.376
yolov5n6	10.597	423.225	23.877	2.190
yolov5s	6.779	1103.224	8.058	0.894
yolov5s6	9.614	1132.361	33.435	0.851
yolox_nano	**3.036**	**171.925**	**5.651**	**5.537**
yolox_tiny	3.061	439.083	5.861	2.232
yolox_s	4.065	1630.156	10.893	0.608

The results indicate that using the CPU execution provider on Jetson Xavier yields comparable or better performance for lightweight object detection models compared to Jetson Nano GPU. This is particularly evident with the tinyYOLOv2, tinyYOLOv3, yolov5n, and yolov5n6 models.

Table 4.16. *Object Detection models inference result on Jetson TX2 (GPU).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
tinyYOLOv2	7.538	30.497	117.326	6.440
tinyYOLOv3	18.821	60.232	1.981	12.364
yolov5n	6.721	65.554	5.104	13.028
yolov5n6	10.480	69.197	23.583	10.040
yolov5s	6.727	121.852	8.272	7.341
yolov5s6	10.324	124.593	33.280	6.091
yolox_nano	**3.140**	**30.717**	6.114	**25.049**
yolox_tiny	3.141	43.823	6.392	18.758
yolox_s	3.943	122.938	15.613	7.019

Table 4.17. *Object Detection models inference result on Jetson Xavier (CPU).*

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
tinyYOLOv2	2.952	100.082	79.102	5.492
tinyYOLOv3	5.893	75.495	1.094	12.130
yolov5n	1.876	124.036	2.364	7.806
yolov5n6	2.743	126.772	11.079	7.150
yolov5s	1.778	280.266	3.614	3.502
yolov5s6	3.013	290.624	15.422	3.242
yolox_nano	**1.470**	**44.706**	**4.033**	**19.932**
yolox_tiny	1.449	105.031	4.420	9.025
yolox_s	1.809	375.859	7.123	2.599

Table 4.18. Object Detection models inference result on Jetson Xavier (GPU).

Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
tinyYOLOv2	4.564	22.231	71.633	10.190
tinyYOLOv3	6.362	46.246	1.317	18.670
yolov5n	2.252	28.790	2.493	30.263
yolov5n6	2.681	31.475	10.090	23.520
yolov5s	1.973	37.336	3.813	23.414
yolov5s6	2.868	39.155	14.226	18.499
yolox_nano	**1.535**	**20.468**	**5.404**	**36.755**
yolox_tiny	1.557	21.848	5.021	35.434
yolox_s	1.751	37.588	6.970	21.831

The image shown in Figure 4.14 summarizes the table results. It confirms that the YOLO models achieve more significant FPS on the CUDA ONNX runtime compared to the CPU ONNX runtime corresponding to each own device.

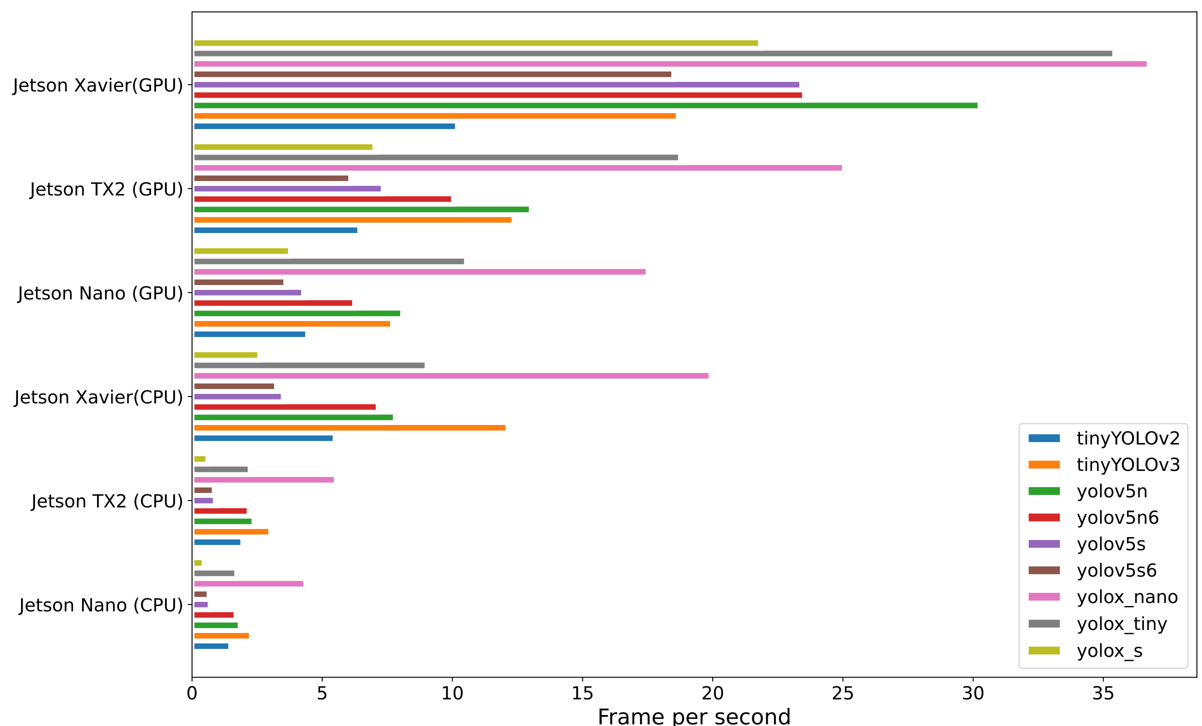

Figure 4.14. Object Detection Models Comparison for all devices.

4.3 Human Pose and Semantic Segmentation

This section will depict all devices' human pose estimation and semantic segmentation inference results.

Table 4.19. *Human Pose Estimation inference results on all devices using stride 8.*

Device	Execution Provider	Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Raspberry Pi 3	CPU	Light_Weight_Human_Pose	35.908	1143.095	308.078	0.673
Raspberry Pi 4	CPU	Light_Weight_Human_Pose	12.502	663.061	132.066	1.239
Jetson Nano	CPU	Light_Weight_Human_Pose	11.513	635.252	98.210	1.343
	GPU	Light_Weight_Human_Pose	11.727	210.265	105.230	3.065
Jetson TX2	CPU	Light_Weight_Human_Pose	9.607	480.743	74.769	1.771
	GPU	Light_Weight_Human_Pose	9.152	61.591	80.494	6.668
Jetson Xavier	CPU	Light_Weight_Human_Pose	3.483	132.261	41.144	5.671
	GPU	Light_Weight_Human_Pose	3.285	20.517	43.198	**15.529**

The output stride parameter determines how much the output is reduced in size relative to the input image, affecting the size of layers and the model's output 4.15. While a smaller stride offers higher accuracy, a higher output stride leads to a lower resolution for network layers and outputs, thus resulting in faster performance. To evaluate the performance of a lightweight human pose model, we first ran it with a small stride (8) and recorded its runtime. Despite being designed as a lightweight estimation model, the workload is still heavy when deployed on Raspberry Pi devices. For this application, the result on Jetson TX2 experienced a significant improvement of 3.7 times when running on GPU compared to solely on the CPU.

Additionally, we conducted experiments with a larger stride (32) to improve speed, and the results are presented in Table 4.20.

Table 4.20. *Human Pose Estimation inference results on all devices using stride 32.*

Device	Execution Provider	Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Raspberry Pi 3	CPU	Light_Weight_Human_Pose	34.368	1093.708	294.381	0.704
Raspberry Pi 4	CPU	Light_Weight_Human_Pose	12.415	424.252	129.707	1.767
Jetson Nano	CPU	Light_Weight_Human_Pose	11.668	648.277	101.041	1.317
	GPU	Light_Weight_Human_Pose	11.915	96.387	106.677	4.686
Jetson TX2	CPU	Light_Weight_Human_Pose	9.767	484.956	75.056	1.756
	GPU	Light_Weight_Human_Pose	9.089	146.329	85.175	4.170
Jetson Xavier	CPU	Light_Weight_Human_Pose	3.709	133.267	40.476	5.649
	GPU	Light_Weight_Human_Pose	3.137	18.923	41.012	**16.246**

The results depicted in plot 4.16 indicate that increasing the stride size to 32 does not improve inference speed on the CPU, but it does improve inference speed on the GPU of

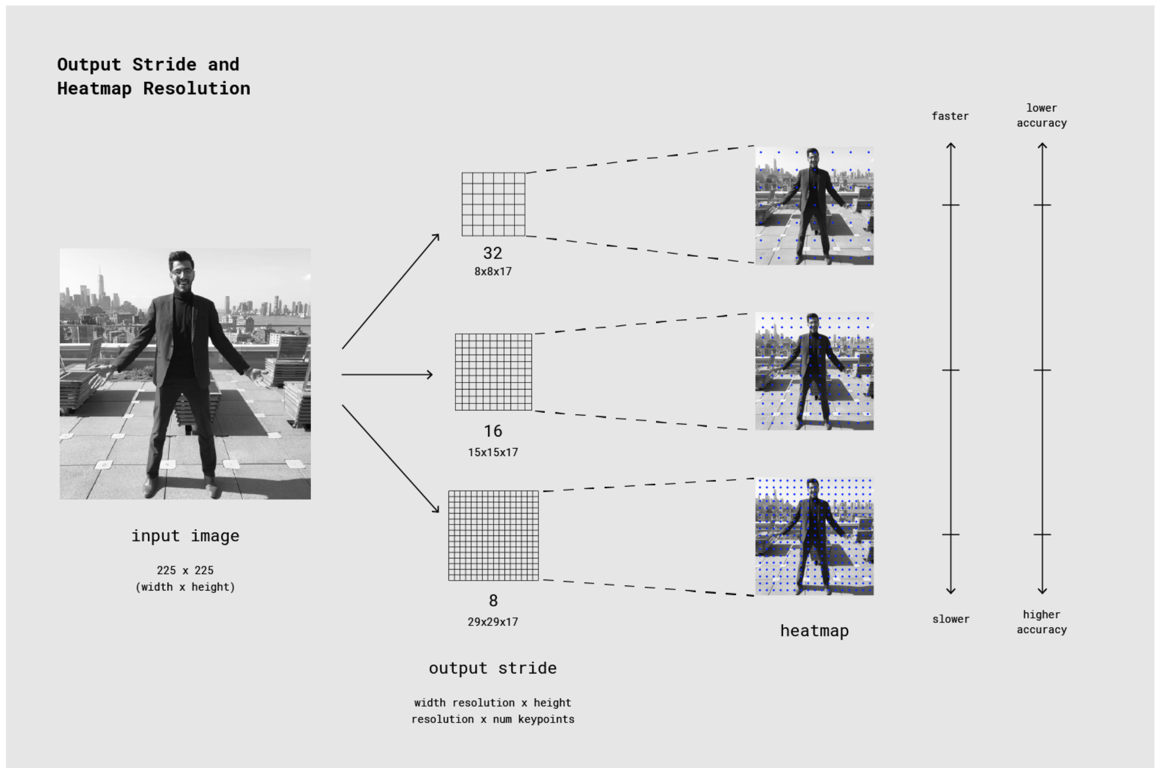

Figure 4.15. *Output Stride and Heatmap Resolution. [16]*

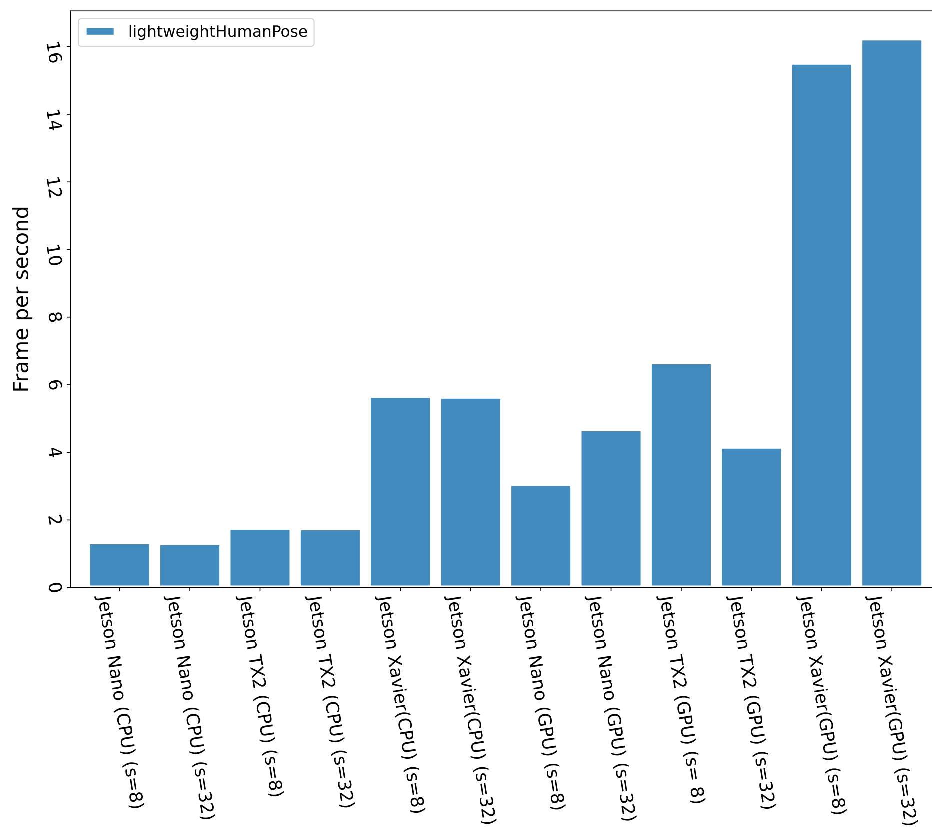

Figure 4.16. *Human pose estimation result for jetson devices.*

both Jetson Nano and Jetson Xavier.

Table 4.21. *Semantic Segmentation inference results on all devices.*

Device	Execution Provider	Model Name	Avg. Preprocess Time (ms)	Avg. Inference Time (ms)	Avg. Postprocess Time (ms)	Avg. FPS
Raspberry Pi 3	CPU	LRASPP_MobileNet_V3_Large	-	-	-	-
Raspberry Pi 4	CPU	LRASPP_MobileNet_V3_Large	-	-	-	-
Jetson Nano	CPU	LRASPP_MobileNet_V3_Large	69.244	666.211	76.330	1.232
	GPU	LRASPP_MobileNet_V3_Large	71.246	166.550	91.285	3.058
Jetson TX2	CPU	LRASPP_MobileNet_V3_Large	52.811	496.342	62.253	1.636
	GPU	LRASPP_MobileNet_V3_Large	51.636	97.000	62.302	4.742
Jetson Xavier	CPU	LRASPP_MobileNet_V3_Large	20.254	192.213	27.987	4.164
	GPU	LRASPP_MobileNet_V3_Large	20.776	37.519	28.539	**11.568**

The kernel terminated the inferencing model **lraspp_mobilenet_v3_large** on raspberry pi because of out-of-memory errors. Before ending abruptly, the inference could only process up to approximately 500 images on the pi3 and 4000 images on the pi4. Both preprocessing and postprocessing times show similar performance on both CPU and GPU for jetson devices. However, when using the CUDA execution provider, the inference time is significantly accelerated, resulting in a nearly three times increase in FPS.

5. CONCLUSION

In conclusion, this thesis has explored the evaluation of multiple neural networks on embedded edge devices and served as an artifact benchmark for reference purposes. The study has highlighted the increasing need for efficient and accurate machine learning models that can operate on edge devices, given their low latency advantages, reduced bandwidth consumption, and improved privacy. The experimental results have demonstrated that evaluating multiple neural networks on embedded edge devices with limited computational resources is possible. The evaluation methodology used in this thesis has shown that the performance of different neural network models can vary significantly depending on the hardware platform and the dataset used for testing.

This thesis has some limitations. Specifically, the benchmark was conducted on a single device for each category, which may not accurately reflect the full range of hardware variability. Therefore, future work should aim to run the benchmark on multiple devices of the same type and calculate the average performance to obtain a more comprehensive perspective of the models' real-world performance across different hardware. This approach would also mitigate the impact of hardware factory production errors and provide a more reliable evaluation of the models. In order to obtain more reliable results, measuring power consumption and its relationship with runtime is fundamental.

In the future, all the processes should be fully automatic. Currently, the process of uploading or downloading scripts and artifact files is carried out atmanually, which is time-consuming. The scripts can be automatically uploaded/downloaded to Google Drive when there are local updates on the laptop. In addition, the scripts should be cross-device compatible, both hardware and software. In general, package manager pip already resolves environment dependencies, but some packages still need manual installation or even source compilation if packages are not already available. Finally, converting to ONNX format with different opset versions should also be considered.

REFERENCES

[1] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C. Burges, L. Bottou and K. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[2] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J. and Keutzer, K. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*. 2016. arXiv: 1602.07360 [cs.CV].

[3] Zhang, X., Zhou, X., Lin, M. and Sun, J. *ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices*. 2017. arXiv: 1707.01083 [cs.CV].

[4] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A. *Going Deeper with Convolutions*. 2014. arXiv: 1409.4842 [cs.CV].

[5] He, K., Zhang, X., Ren, S. and Sun, J. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[6] Zhang, A., Lipton, Z. C., Li, M. and Smola, A. J. *Dive into Deep Learning*. 2023. arXiv: 2106.11342 [cs.LG].

[7] Huang, G., Liu, Z., Maaten, L. van der and Weinberger, K. Q. *Densely Connected Convolutional Networks*. 2018. arXiv: 1608.06993 [cs.CV].

[8] Abdelouahab, K., Pelcat, M., Serot, J. and Berry, F. *Accelerating CNN inference on FPGAs: A Survey*. 2018. arXiv: 1806.01683 [cs.DC].

[9] Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V. and Wang, Y. Optimizing CNN Model Inference on CPUs. *USENIX Annual Technical Conference*. 2018.

[10] Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V. and Wang, Y. Optimizing CNN Model Inference on CPUs. *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 1025–1040. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/liu-yizhi.

[11] Cantero, D., Esnaola-Gonzalez, I., Miguel-Alonso, J. and Jauregi, E. Benchmarking Object Detection Deep Learning Models in Embedded Devices. *Sensors* 22.11 (2022). ISSN: 1424-8220. DOI: 10.3390/s22114205. URL: https://www.mdpi.com/1424-8220/22/11/4205.

[12] Shashank Prasanna, P. K. and Milletari, F. 2017. URL: https://developer.nvidia.com/blog/tensorrt-3-faster-tensorflow-inference/.

[13] Ren, Y., Yang, J., Zhang, Q. and Guo, Z. Multi-Feature Fusion with Convolutional Neural Network for Ship Classification in Optical Images. *Applied Sciences* 9.20 (2019). ISSN: 2076-3417. DOI: 10.3390/app9204209. URL: https://www.mdpi.com/2076-3417/9/20/4209.

[14] Ren, S., He, K., Girshick, R. and Sun, J. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2016. arXiv: 1506.01497 [cs.CV].

[15] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV].

[16] *Real-time Human Pose Estimation in the Browser with TensorFlow.js*. URL: https://blog.tensorflow.org/2018/05/real-time-human-pose-estimation-in.html.

[17] *A Comprehensive Guide on Human Pose Estimation*. Accessed: 2010-09-30. URL: https://www.analyticsvidhya.com/blog/2022/01/a-comprehensive-guide-on-human-pose-estimation/.

[18] Long, J., Shelhamer, E. and Darrell, T. *Fully Convolutional Networks for Semantic Segmentation*. 2015. arXiv: 1411.4038 [cs.CV].

[19] Kim, J., Lee, J., Son, S. and Kim, J. An Application on Visual Cognitive Assistance system for Braille Block Recognition Based on Raspberry Pi System. *International Journal on Advanced Science, Engineering and Information Technology* 11 (Dec. 2021), p. 2527. DOI: 10.18517/ijaseit.11.6.13771.

[20] Davidson, P., Trinh, H., Vekki, S. and Müller, P. Surrogate Modelling for Oxygen Uptake Prediction Using LSTM Neural Network. *Sensors* 23.4 (2023). ISSN: 1424-8220. DOI: 10.3390/s23042249. URL: https://www.mdpi.com/1424-8220/23/4/2249.

[21] Pytorch. *Pytorch: Models and pre-trained weights for Classification*. URL: https://pytorch.org/vision/stable/models.html#table-of-all-available-classification-weights.

[22] Pytorch. *Pytorch: Models and pre-trained weights for Object Detection*. URL: https://pytorch.org/vision/stable/models.html#object-detection.

[23] (ONNX), O. N. N. E. *Github: onnx/models/vision/object_detection_segmentation/tiny-yolov2/*. URL: https://github.com/onnx/models/tree/main/vision/object_detection_segmentation/tiny-yolov2.

[24] (ONNX), O. N. N. E. *Github: onnx/models/vision/object_detection_segmentation/tiny-yolov3/*. URL: https://github.com/onnx/models/tree/main/vision/object_detection_segmentation/tiny-yolov3.

[25] *Github: ultralytics/yolov5*. URL: https://github.com/ultralytics/yolov5#pretrained-checkpoints.

[26] Megvii-BaseDetection. *Github: Megvii-BaseDetection /YOLOX*. URL: https://github.com/Megvii-BaseDetection/YOLOX.

[27] Pytorch. *Pytorch: Models and pre-trained weights for Semantic Segmentation*. URL: https://pytorch.org/vision/stable/models.html#semantic-segmentation.

[28] Osokin, D. *Real-time 2D Multi-Person Pose Estimation on CPU: Lightweight OpenPose*. 2018. arXiv: 1811.12004 [cs.CV].

[29] MLflow. *An open source platform for the machine learning lifecycle*. URL: https://mlflow.org/.

[30] bentoml. *Unified Model Serving Framework*. URL: https://www.bentoml.com/.

[31] open-mmlab. *OpenMMLab Detection Toolbox and Benchmark*. URL: https://mmdetection.readthedocs.io/en/latest/.

[32] Wang, K., Liu, Z., Lin, Y., Lin, J. and Han, S. HAQ: Hardware-Aware Automated Quantization. *CoRR* abs/1811.08886 (2018). arXiv: 1811.08886. URL: http://arxiv.org/abs/1811.08886.

[33] He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J. and Han, S. *AMC: AutoML for Model Compression and Acceleration on Mobile Devices*. 2019. arXiv: 1802.03494 [cs.CV].