

**NANYANG  
TECHNOLOGICAL  
UNIVERSITY**

**School of Computer Science and Engineering**

**Bachelor of Engineering (Computer Science)**

**Final Year Project:**

**PCSE22-0071**

**Parametrized DNN Design for Identifying the Resource Limitations  
of Edge Deep Learning Hardware**

**Name: Shin Thant Aung**

**Matric Number: U2020855A**

**Supervisor: A/P Liu Weichen**

**Mentor: Dr Hao Kong**

## Contents

List Of Figures .....	4
List Of Tables .....	5
Abstract.....	6
Acknowledgement .....	6
Chapter 1: Introduction.....	7
1.1 Background.....	7
1.2 Objectives.....	8
1.3 Organization .....	9
Chapter 2: Literature Review.....	10
2.1 Deep Neural Networks .....	10
2.2 Deep Learning Hardware .....	12
2.3 Edge Computing .....	13
2.4 Image Classification.....	14
Chapter 3: Methodology .....	15
3.1 Hardware Used.....	15
3.1.1 NVIDIA Jetson Nano .....	15
3.1.2 NVIDIA GeForce RTX 3070 .....	15
3.2 Software Used .....	16
3.2.1 Python 3.10+.....	16
3.2.2 PyTorch.....	17
3.2.3 Torchvision API .....	17
3.2 Dataset .....	18
3.2.1 ImageNet.....	18
3.2.2 Data Preprocessing .....	19
3.5 Profiling Tools .....	20
3.5.1 Torch Profiler.....	20
3.5.2 CodeCarbon.....	20
3.6 Hyperparameters.....	21
3.6.1 Floating Point Operations (FLOPs) .....	21
3.6.2 Input Shape.....	22

3.6.3 Batch Sizes .....	22
3.7 Metrics .....	22
3.7.1 Power Usage .....	22
3.7.2 Latency and Throughput .....	23
3.7.3 Memory .....	23
3.7.4 Top1 and Top5 Accuracy .....	24
3.8 Model Compression .....	24
3.8.1 Pruning .....	24
3.8.2 Torch-Pruning Tool .....	25
Chapter 4: Implementation .....	26
4.1 Overview .....	26
4.2 Energy Usage .....	27
4.3 Inference Time .....	28
4.3.1 Asynchronous Execution .....	28
4.3.2 GPU Warmup .....	29
4.4 Memory Usage .....	30
4.4.1 Allocated Memory .....	30
4.4.2 Reserved Memory .....	31
4.5 Complexity Tuning Through Compression .....	31
4.5.1 Layer Based Performance Analysis .....	32
4.5.2 Structured Pruning through Torch-Pruning .....	33
Chapter 5: Result .....	34
5.1 Hyperparameters .....	34
5.1 Complexity Tuning .....	34
5.2 Memory Footprint .....	36
5.3 Latency and Throughput Profile .....	37
5.4 Power Consumption .....	37
5.5 Result Conclusion .....	38
Chapter 6: Conclusion and Future Work .....	39
Chapter 6.1: Conclusion .....	39
Chapter 6.2: Future Work .....	40
Appendix .....	41

References .....	45
------------------	----

## List Of Figures

Figure 1 Representation of a Neural Network [1].....	10
Figure 2 Convolutional Neural Network (CNN) [2].....	11
Figure 3 Comparison of CPU versus GPU Architecture [4].....	12
Figure 4 Deep Learning Inference on Edge Device [7] .....	13
Figure 5 Illustration of VGG16 Architecture [8] .....	14
Figure 6 Code Snippet of Image Transformation Implementation .....	19
Figure 7 Code Carbon Configuration.....	20
Figure 8 Illustration of Latency vs Throughput [24].....	23
Figure 9 Unstructured versus Structured Pruning [26] .....	25
Figure 10 Grouped Parameters with Inter-dependency in Different Structures [28] .....	25
Figure 11 High-Level Structure of the Project.....	26
Figure 12 Code Snippet of CodeCarbon Usage .....	27
Figure 13 Code Snippet of Measuring Inference Time .....	28
Figure 14 GPU Warmup .....	29
Figure 15 Code Snippet of Torch Profiler Implementation .....	30
Figure 16 Reserved and Allocated Memory of Mobilenet_v2 Model Inference .....	31
Figure 17 Illustration of Layer-Based Performance Analysis.....	32
Figure 18 Pruning Strategy .....	33
Figure 19 VGG16 Models' Accuracy.....	35
Figure 20 Memory Footprint of Pruned VGG16 on Jetson Nano.....	36
Figure 21 Throughput and Latency of VGG16.....	37
Figure 22 Energy Consumption of VGG16 .....	38
Figure 23 Accuracies of Resnet50 .....	42
Figure 24 Resource Requirements of Resnet50 .....	42
Figure 25 Accuracies of MobileNet.....	43
Figure 26 Resource Requirements of MobileNet.....	44

## List Of Tables

Table 1 Specifications of Hardware Used.....	16
Table 2 PyTorch Pretrained Classification Models Used for Experiments.....	18
Table 3 Equations Used to Compute FLOPs .....	21
Table 4 Selected Hyperparameters.....	34
Table 5 VGG16 Models' Complexities .....	35
Table 6 FLOPs of Resnet50 .....	41
Table 7 FLOPs of MobileNet.....	43

## **Abstract**

Artificial intelligence has come a long way in the last several years and has made remarkable advancements, with deep learning neural networks emerging as a powerful tool for solving complex problems. Even with the technology having rapidly grown to such incredible levels thus far, one of the most significant challenges still standing is how to effectively deploy deep learning algorithms on edge devices such as mobile phones and Internet of Things (IOT) devices while preserving performance. Edge devices often face limitations in resources, such as memory and processing power, which can raise challenges for running complex deep-learning models. The fundamental goal of this project is to evaluate the resource constraints of edge devices, allowing for the deployment of deep learning algorithms without sacrificing performance. This enables tasks such as image recognition and speech processing to be carried out directly on edge devices, minimizing the need to rely on powerful cloud servers. In this final year project (FYP), a program to identify resource limitations on edge learning devices while inferencing deep learning models will be developed using Python and the deep learning framework PyTorch. Furthermore, the program will tune hyperparameters such as input shapes and batch sizes, as well as model complexity such as feature maps and weights, to achieve an ideal state that minimizes resource demands while preserving performance.

## **Acknowledgement**

First and foremost, I wish to express my deepest gratitude to my supervisor, Associate Professor Liu Weichen, and my mentor, Dr. Hao Kong, for their guidance throughout this project. I would also like to give my warmest thanks to my family for their unwavering support throughout this academic journey. Additionally, I would like to pay special thanks to my girlfriend, who has always been supportive and encouraged me in all of my endeavours. Lastly, I am grateful to Nanyang Technological University (NTU) for providing the resources needed to complete this project and my bachelor's degree.

# Chapter 1: Introduction

## 1.1 Background

The increasing demand for smart applications across various sectors, such as healthcare, smart cities, and industrial automation, can be addressed by integrating deep learning with edge computing. As a result, edge computing, which processes data locally on devices rather than transferring it to distant servers and enhances the entire process speed, has become a growing trend in technology. Edge computing is all about making devices like the Internet of Things (IoT), smartphones, and sensors smarter. However, these edge devices frequently face various resource limitations, such as limited memory, computing capacity, and energy efficiency.

In response to these challenges, developing a specialized method: a parametrized deep neural network (DNN) design methodology can be used for edge deep learning hardware. The primary goal is to tune the computational complexity and optimize the parameters of the DNN to assess its applicability on edge devices while preserving performance. To enable a wide range of applications like real-time image recognition, speech processing, and anomaly detection in resource-constrained environments, deep learning technologies on edge devices have become widely adopted. Consequently, organizations can take advantage of new opportunities for real-time analytics, predictive maintenance, and customised services without relying on cloud infrastructure by utilizing deep learning algorithms on edge devices.

## 1.2 Objectives

The aim of this project is to determine the feasibility of deploying a deep learning model on an edge device by thoroughly analysing the model's performance and resource utilisation throughout the inference process. Furthermore, this project involves fine-tuning the model's complexity and tweaking hyperparameters such as input and batch sizes to get an ideal state that fits the edge device's capabilities while maintaining adequate performance levels. This project will use convolutional neural networks (CNN) and an embedded edge device, the NVIDIA Jetson Nano, for the experiments. The results will evaluate the resource limits of edge devices for deploying deep learning models, offering vital insights into their applicability. Additionally, no study will be conducted on model training performances or performance optimization strategies because edge devices are solely intended for edge operations like inference.



## 1.3 Organization

The report is structured into six chapters, each of which focuses on key components of the project.

Chapter 1 provides an introduction, covering the background, objectives, and organization of the report.

Chapter 2 is the literature review, which includes topics such as deep neural networks, deep learning hardware, edge computing, and object classification.

Chapter 3 outlines the methodology, detailing the hardware and software used, dataset considerations, profiling tools, hyperparameters, metrics used to measure performance and resource requirements, and model compression techniques used.

In Chapter 4, the implementation of the project is discussed, including an overview, energy usage, inference time, memory usage, and complexity tuning through compression.

Chapter 5 presents results, focusing on the hyperparameters chosen for experiments, the complexity tuning results, memory footprint, latency and throughput profile, and power consumption.

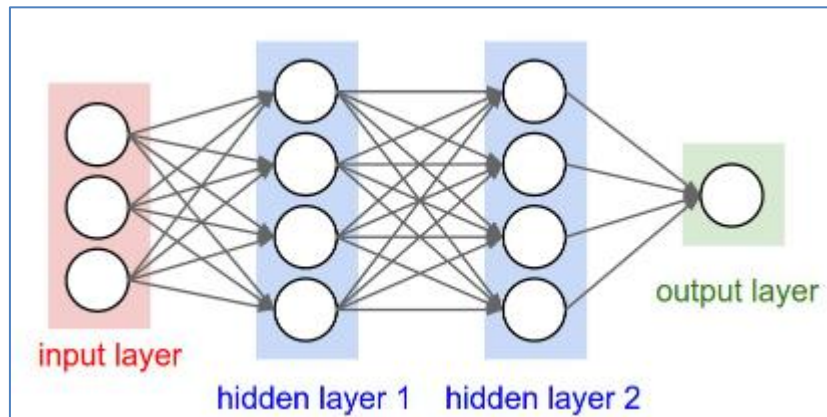
Finally, Chapter 6 concludes the report and suggests possibilities for future work.

The appendix contains additional supplementary materials.

## Chapter 2: Literature Review

### 2.1 Deep Neural Networks

Deep learning is one of the most widespread machine learning algorithms, inspired by the intricate workings of the human brain. Fundamentally, neural networks are made up of linked nodes, or neurons, structured in layers. Each node, or neuron, receives input from the prior layer and processes it before sending its output via weighted connections to the subsequent post layers. Neural networks then acquire the ability to recognize patterns and generate predictions from massive datasets through a process known as training. During the training process, optimization algorithms are used to minimize the gap between the expected and actual outputs. There are several types of neural networks, each designed for specific tasks and data types, and thus, the popularity of a neural network architecture often depends on the task and the characteristics of the data involved.



*Figure 1 Representation of a Neural Network [1]*

Convolutional neural networks (CNNs) are now recognised as one of the most notable classes of neural networks, having shown tremendous growth in popularity in recent years. Computer vision tasks, including object identification, picture segmentation, and image classification, are areas where CNNs specialize and excel with their unique architecture. CNNs consist of multiple layers, such as convolutional layers, pooling layers, and fully linked layers. These layers collaborate to extract information from input images, such as

patterns, shapes, or colours. This process allows neural networks to learn from data and make predictions.

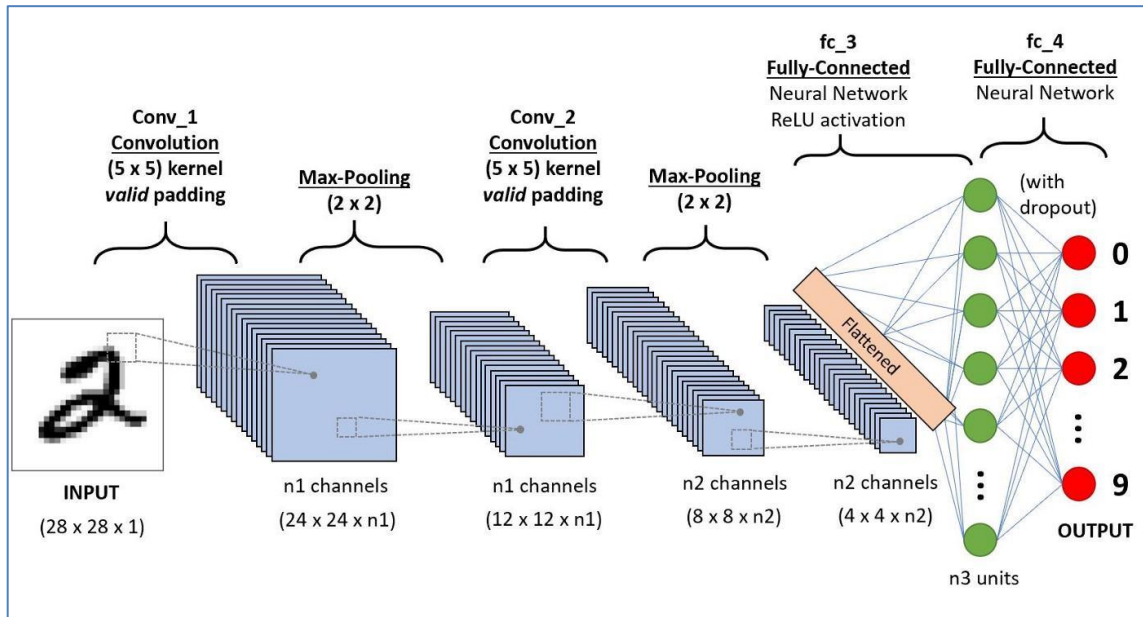


Figure 2 Convolutional Neural Network (CNN) [2]

Convolutional layers play a key role in feature extraction by applying a set of learnable filters to input pixels, producing feature maps that highlight relevant patterns and structures. These feature maps are then down-sampled through pooling layers, retaining key information while reducing spatial dimensions. Finally, fully connected layers integrate the extracted features to make predictions based on learned representations. As a result, CNNs have revolutionized computer vision and image processing, achieving state-of-the-art performance in applications like image classification, image segmentation, and object detection. Their ability to automatically learn hierarchical representations from raw pixel data makes them ideally suited for analysing and understanding visual information, driving advancements in diverse fields.

## 2.2 Deep Learning Hardware

Since a few decades ago, advances in hardware technology have always had a profound impact on neural networks and deep learning research. Deep learning applications require large amounts of simultaneous mathematical computation, and general-purpose CPUs cannot do them efficiently. Despite the widespread use of multiple cores and hyper-threading in mainstream technology, CPUs still have limits in deep learning applications. On the other hand, GPUs, FPGAs, and ASICs provide higher computational performance and reduced latency because of their specialized designs, which include several processing cores and on-chip memory [3].

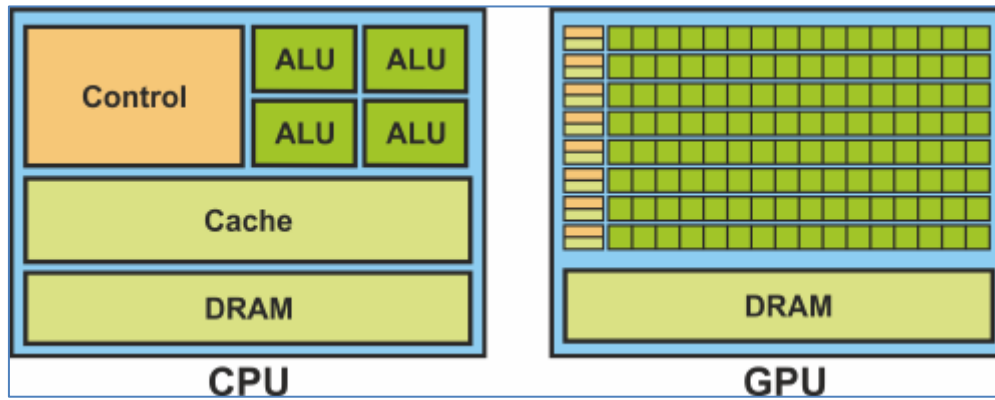


Figure 3 Comparison of CPU versus GPU Architecture [4]

There has been an increase in the use of deep learning models on edge devices, including mobile phones, IoT devices, and embedded systems. Conducting inference directly on these devices reduces latency and eliminates the need to send data to and from a cloud computing server. However, edge technology imposes several limits, including limited processing performance and battery constraints. For example, many IoT devices, such as the Raspberry Pi, have multi-core CPUs that are not well suited for deep learning inference workloads. As a result, specialized, embedded devices equipped with power-efficient GPUs that have substantial computing capabilities, such as NVIDIA Jetson Nano and Brains Pi, play critical roles in overcoming these constraints [5].

## 2.3 Edge Computing

Edge computing is a computer paradigm in which data is handled and stored closer to endpoints or edge devices rather than on centralized cloud servers [6]. This strategy aims to reduce latency, improve data security, and boost overall system efficiency by processing data locally or at neighbouring edge nodes. Edge computing involves deploying deep neural networks to the network edge which includes IoT devices, mobile phones, and even self-driving cars. These devices often have limited processing power and storage capacity, allowing them to conduct deep learning tasks locally. Nowadays, Edge computing has advanced significantly with the incorporation of parallel computing devices such as GPUs and ASICs, allowing complicated deep learning inference jobs to be executed directly on the edge.

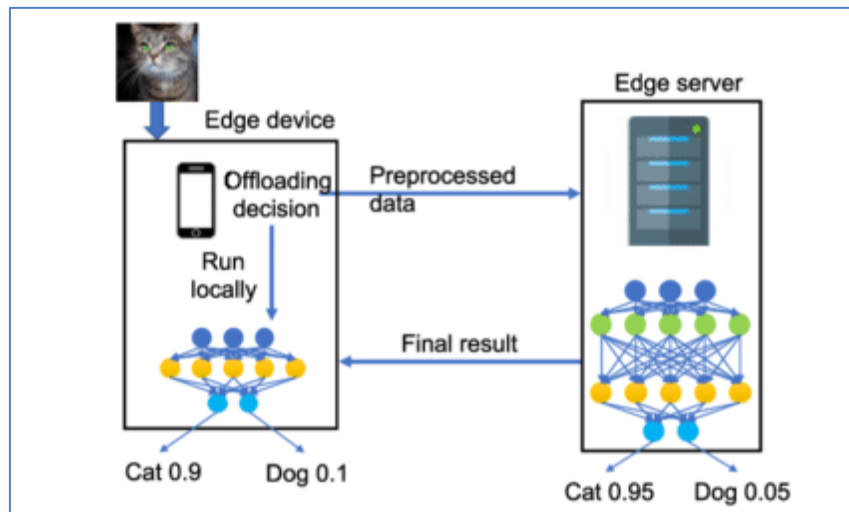


Figure 4 Deep Learning Inference on Edge Device [7]

## 2.4 Image Classification

Deep learning image classification is the process of teaching computers how to recognize and categorize objects in images. It is a crucial task in computer vision with numerous applications, from autonomous driving to medical diagnostics. Classifying images usually requires several steps to be done. Initially, a sizable dataset of labelled images is gathered, with each photo associated with a specific label or category. Subsequently, a CNN model is trained by utilizing this dataset, and the network is further trained to identify patterns and features unique to each class of objects. The CNN model that has been trained can be used to categorize unseen images into predetermined groups. When a picture is fed into a trained network, predictions about its content are obtained. This process is called inference. Each category is then given a probability score by the model, which expresses how likely the image falls into that class. The category for which the likelihood score is highest is then regarded as the image's suggested label.

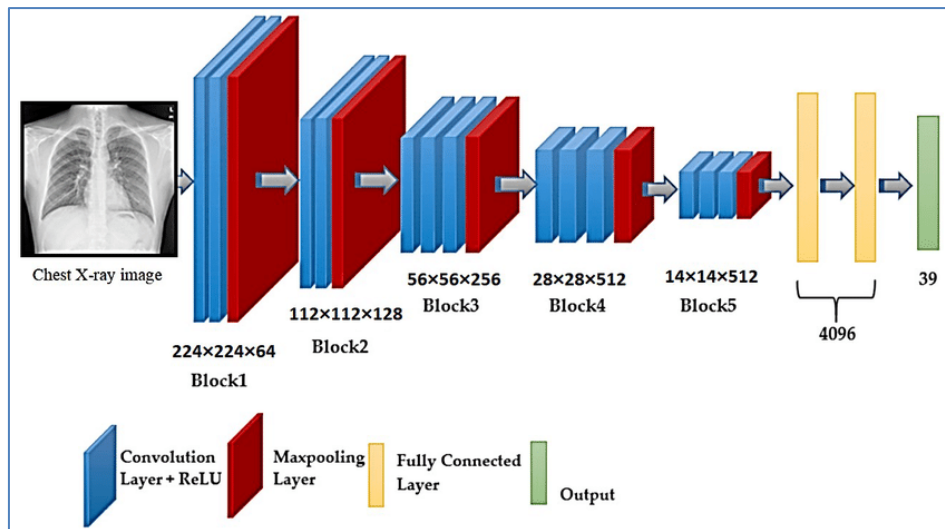


Figure 5 Illustration of VGG16 Architecture [8]

## **Chapter 3: Methodology**

### **3.1 Hardware Used**

To conduct deep learning model inference, the following hardware devices were selected, each offering unique capabilities and features.

#### **3.1.1 NVIDIA Jetson Nano**

NVIDIA Jetson Nano is equipped with a GPU based on NVIDIA's Maxwell Architecture [9] and was chosen for its powerful GPU acceleration, making it suitable for computationally intensive tasks such as computer vision. With its low power consumption, it is well-suited for edge computing applications, particularly in IoT and mobile deployments. The vibrant community support and cost-effectiveness further contribute to its suitability for the experiments.

#### **3.1.2 NVIDIA GeForce RTX 3070**

NVIDIA GeForce RTX 3070 represents a high-performance desktop GPU tailored for gaming and graphics-intensive applications [10]. Although it is not often categorized as an edge device, its exceptional computing capabilities, along with dedicated hardware for ray tracing and AI inference, make it ideal for undertaking hyperparameter tweaking, performance analysis, and optimization of deep learning models.

<b>Specifications</b>	<b>NVIDIA Jetson Nano [11]</b>	<b>NVIDIA GeForce RTX 3070 [10]</b>
<b>CPU</b>	Quad-core ARM Cortex-A57 MPCore processor	Intel(R) Core(TM) i5-10600KF CPU @ 4.10GHz 4.10 GHz
<b>GPU</b>	NVIDIA Maxwell architecture with 128 NVIDIA CUDA cores	NVIDIA 2nd Gen Ampere architecture with 5888 NVIDIA CUDA cores
<b>Memory</b>	4 GB 64-bit LPDDR4, 1600 MHz, 25.6 GB/s	8 GB GDDR6
<b>Power Requirements</b>	<b>5 - 10 Watts</b>	<b>220 - 250 Watts</b>

*Table 1 Specifications of Hardware Used*

## 3.2 Software Used

### 3.2.1 Python 3.10+

Python was used as the primary programming language for the implementation. Python version 3.10 or above was chosen for its cutting-edge capabilities, improved performance, and compatibility with the necessary libraries and frameworks for deep learning workloads.



### 3.2.2 PyTorch

PyTorch [12] is an open-source machine learning framework based on the C++ Torch Backend Library developed by Facebook's AI Research Lab (FAIR). PyTorch provides a seamless platform for building and deploying neural networks. PyTorch was selected for the following reasons:

- 1. Dynamic Computational Graphs:** Unlike static computational graphs in frameworks like TensorFlow, dynamic computational graphs are utilized in PyTorch. This allows for dynamic adjustment of the graph during runtime, making it flexible for experimentation [13].
- 2. Pythonic Interface:** PyTorch offers a Python-first approach, making its syntax closely resemble NumPy and facilitating smooth prototyping and experimentation with deep learning models for researchers.
- 3. GPU Acceleration:** PyTorch seamlessly integrates with CUDA, enabling efficient utilization of GPUs for accelerated inference and training.

### 3.2.3 Torchvision API

Torchvision is a part of PyTorch and offers a collection of datasets, pre-trained deep-learning models, and image transformations specifically designed for computer vision tasks. Torchvision also offers access to pre-trained model weights that are trained on benchmark datasets such as ImageNet [14] for object recognition and COCO dataset [15] for object detection. The following pre-trained object recognition models [16] were used for experimenting with inference on the platforms.

Model	Top1 Accuracy	Top5 Accuracy	Params	GFLOPs
MobileNet V3 Large	75.27	92.57	5.5M	0.22
VGG16	71.59	90.38	138.4M	15.47
Resnet50	80.86	95.43	25.6M	4.09

*Table 2 PyTorch Pretrained Classification Models Used for Experiments*

## 3.2 Dataset

### 3.2.1 ImageNet

ImageNet dataset [17] is a massive and structured image database designed for object classification and contains over 14 million annotated images, organized according to the WordNet hierarchy. A significant aspect of ImageNet is the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which was held from 2010 to 2017. The ILSVRC 2017 validation dataset contains 50,000 images with 1,000 object classes, providing diverse and feasible data for performance evaluation. In this project, ImageNet validation dataset from ILSVRC 2017 was utilized as the benchmark dataset to determine the accuracy of the deep neural networks.

### 3.2.2 Data Preprocessing

Data preprocessing involves transforming raw image pixels into usable and meaningful formats for downstream tasks. PyTorch provides a package for computer vision transformations, namely ‘torchvision. transforms’ [18], which has various built-in functions to preprocess images, such as resizing, normalization, data augmentation techniques, etc. The following preprocessing steps for the ImageNet validation dataset were applied:

**1. Resizing:** The images were scaled to 224 by 224 pixels, the usual input size for models trained on ImageNet. Alternatively, customized sizes were employed during hyperparameter tuning, as explained further in the Implementation section.

**2. Normalization:** The images were normalized using the mean and standard deviation precomputed from the ImageNet train dataset.

```
1  from torchvision.datasets import ImageNet
2  from torchvision import transforms
3
4
5  # Define the transformation
6  transform = transforms.Compose([
7      transforms.Resize((224, 224)),
8      transforms.ToTensor(),
9      transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
10 ])
11
12
13 # Apply the transformation to the validation dataset
14 val_dataset = ImageNet(root_dir='dataset_path', transform=transform)
```

*Figure 6 Code Snippet of Image Transformation Implementation*

## 3.5 Profiling Tools

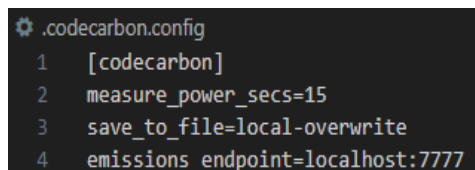
### 3.5.1 Torch Profiler

The Torch Profiler [19] stands as a robust tool offering comprehensive performance metrics for deep learning models during both training and inference, across CPU and GPU devices. Leveraging the Profiler's context manager allows for detailed profiling of resource consumption, examination of input/output shapes, and monitoring of device kernel activities associated with model operations. Notably, the PyTorch Profiler's design prioritizes efficiency through a C++ backend while maintaining user-friendliness with Python bindings.

The PyTorch Profiler played a pivotal role in pinpointing performance bottlenecks within deep learning models and uncovering resource limitations inherent to edge deep learning hardware.

### 3.5.2 CodeCarbon

CodeCarbon [20] is an open-source tool designed to estimate hardware power utilities. Its seamless integration with hardware-specific utility tools such as Intel Power Gadget [21] for CPU power measurement, pynvml [22] (a wrapper around the Nvidia Management Library NVML) for CUDA, and power metrics for Apple Silicon Chips (M1, M2) compatibility, enhances its functionality and reliability. CodeCarbon uses a scheduler that measures every 15 seconds by default to prevent significant overhead. CodeCarbon measuring intervals can be customized by creating a configuration file in the working directory as follows.



```
1 [codecarbon]
2 measure_power_secs=15
3 save_to_file=local-overwrite
4 emissions_endpoint=localhost:7777
```

*Figure 7 Code Carbon Configuration*

### 3.6 Hyperparameters

During the inference of the deep learning models, several critical hyperparameters are adjusted to trade-off between performance and resource utilization. These hyperparameters include:

#### 3.6.1 Floating Point Operations (FLOPs)

FLOPs (Floating Point Operations) are the number of floating-point mathematical computations carried out by the model during inference. This metric offers insights into the computational complexity of the model and directly influences inference time. FLOPs are derived from factors including the dimensions of the input feature maps, the size of the convolutional kernels in convolution layers, and the number of output channels. Consequently, adjusting these variables enables the tuning of FLOPs, aiding in the identification of computational bottlenecks. The following equations were used to estimate the FLOPs of the respective layers [23].

Layer Type	Equation
Convolution Layer	$2 \cdot K_1 \cdot K_2 \cdot C_{in} \cdot W_{out} \cdot H_{out} \cdot C_{out}$
Linear Layer	$2 \cdot C_{out} \cdot W_{out} \cdot H_{out}$

*Table 3 Equations Used to Compute FLOPs*

$K_1 \cdot K_2$  represents the dimensions of the convolutional kernel.  $C_{in}$  and  $C_{out}$  denote the number of input and output channels of their respective layers, while  $W_{in}$ ,  $W_{out}$  and  $H_{out}$  represent the input width, output width, and output height, respectively.

### **3.6.2 Input Shape**

The input shape refers to the dimensions of the input images fed into the deep learning model during inference. Proper adjustment of the input shape ensures compatibility of DNN models with edge devices, balancing trade-offs between accuracy and hardware constraints.

### **3.6.3 Batch Sizes**

Batch size determines the number of input samples processed simultaneously during the inference process. Large batch sizes can improve throughput by leveraging parallelism, but they may require more memory resources. Batch sizes can be determined based on the memory constraints of the edge hardware devices, maximizing computational throughput. Experimentation with various batch sizes helps to identify the optimal balance between memory utilization and inference speed.

## **3.7 Metrics**

The evaluation of deep learning models deployed on edge devices necessitates the measurement of various performance metrics to assess their efficiency and effectiveness. The following metrics are utilized to assess the performance of the deployed models:

### **3.7.1 Power Usage**

Power consumption measures the amount of electrical power consumed by the edge device during the execution of deep learning models. It is an essential metric for assessing the energy efficiency and sustainability of edge computing solutions. Power consumption

data was collected using tools such as CodeCarbon to provide insights into the energy requirements of the deployed models.

### 3.7.2 Latency and Throughput

Latency refers to the time taken for a single inference operation to be completed and is a critical metric for real-time applications where prompt responses are essential. Throughput measures the number of inference operations that can be processed per unit time, indicating the overall processing capacity of the system. These metrics were measured under different experimental conditions to assess the model's responsiveness and efficiency.

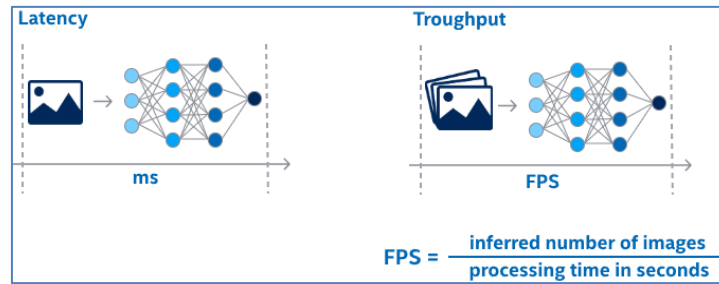


Figure 8 Illustration of Latency vs Throughput [24]

### 3.7.3 Memory

Memory usage quantifies the amount of main memory (RAM) required by the deep learning model during inference. Understanding memory usage is crucial for optimizing resource allocation and ensuring efficient utilization of available memory resources on edge devices. Memory usage data was collected and analysed to gain insights into the memory requirements of the deployed models.

### 3.7.4 Top1 and Top5 Accuracy

Top-1 accuracy is the percentage of predictions where the correct class label is the model's prediction with the highest probability, and Top-5 accuracy represents the percentage of predictions where the correct class label is among the top five highest probability predictions. It provides a broader assessment of the model's performance by considering multiple possible predictions. Both top-1 and top-5 accuracy metrics were calculated and analysed to assess the classification performance of the deployed models.

## 3.8 Model Compression

To fine-tune the model complexity and optimize the computational performance of deep learning models for deployment on edge devices, two model compression approaches pruning, and quantization are used.

### 3.8.1 Pruning

Pruning, a model compression technique, involves removing redundant connections (weights) from the neural network while maintaining its performance. The method was utilized to reduce the computational complexity of the deep learning models. Specifically, less important connections were identified and pruned based on their magnitudes or other criteria, effectively reducing the number of parameters and convolutional kernel sizes in the model. Pruning adjusted the number of FLOPs (floating point operations) required for inference, thus experimenting with computational efficiency within the acceptable range of trade-offs between model performance. There are two types of pruning techniques:

1. Structured Pruning: eliminates whole neurons, channels, and layers from the network while maintaining model structure. Predetermined criteria, such as



weighted magnitude or Taylor importance [25], are used to systematically identify and prune less important portions.

2. Unstructured Pruning: PyTorch masks zero to random individual connections or weights from the network without considering the model structure.

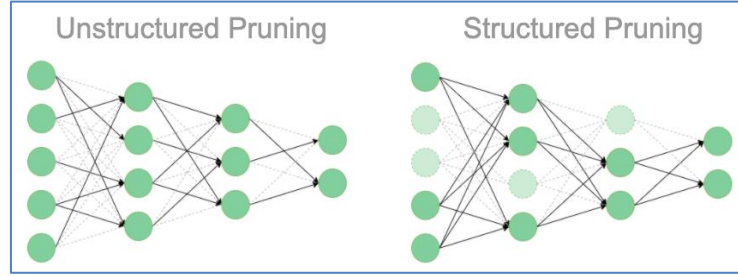


Figure 9 Unstructured versus Structured Pruning [26]

### 3.8.2 Torch-Pruning Tool

Torch-Pruning is a PyTorch pruning framework that executes structured pruning by utilizing the DepGraph [27] algorithm. In structural pruning, minimal prunable components of the neural network are called “Group”. Most groups consist of multiple layers that are interconnected and pruning needs to be conducted together to maintain structural integrity. To overcome such challenges, the DepGraph algorithm identifies the group in the network seamlessly and facilitates pruning.

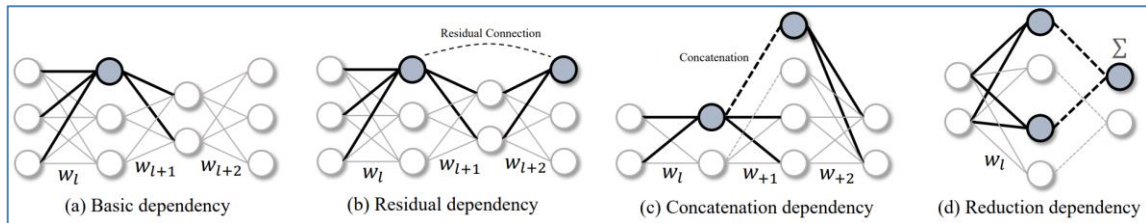


Figure 2. Grouped parameters with inter-dependency in different structures. All highlighted parameters must be pruned simultaneously.

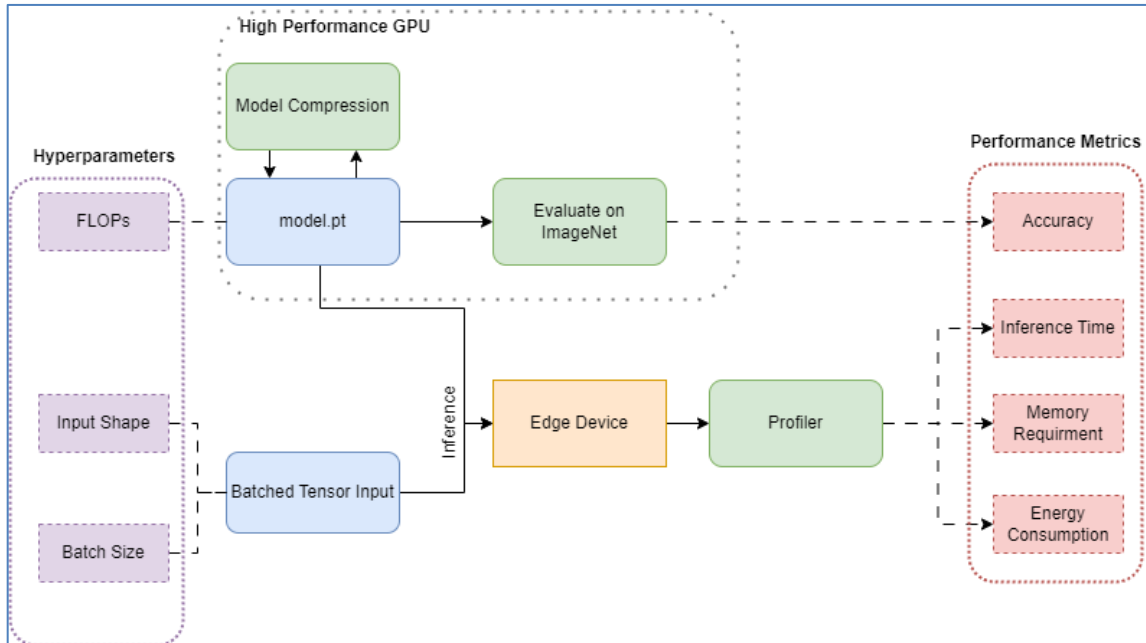
Figure 10 Grouped Parameters with Inter-dependency in Different Structures [28]

## Chapter 4: Implementation

During the implementation phase of the project, hyperparameters were fine-tuned, and deep learning models were evaluated on edge devices. The primary focus was on measuring key performance metrics, including inference time, memory utilization, and power usage. This process allowed for the adjustment of trade-offs between model performance and resource utilization. Ultimately, it facilitated the determination of the suitability of CNN models for specific edge devices.

### 4.1 Overview

The illustration *Figure 11* outlines the overarching structure of this project. Accuracy assessment and model compression were conducted independently on a desktop PC equipped with a high-performance GPU to expedite processing. The profiling of FLOPs for assessing model complexity and resource limitations was carried out during inference on the edge device.



*Figure 11 High-Level Structure of the Project*

## 4.2 Energy Usage

Energy consumption during inference can be approximated using the CodeCarbon package. The implementation of CodeCarbon in this project is illustrated in *Figure 12* below. Lines 83-85 initialize the energy tracker and commence tracking every 15 seconds, as specified in the configuration file referenced in *Figure 7*. Subsequently, lines 90-92 conclude the tracking and compute the energy consumed in kilowatt-hours (kWh) during the tracking duration.

```

79     def record_energy(self, stop:bool=False)->float:
80
81         # initiate energy tracker
82         if(not stop):
83             self._etracker = OfflineEmissionsTracker(project_name="cpu",
84                                                         log_level= 'error', save_to_file=False)
85             self._etracker.start() # start tracking
86             return
87
88         # stop energy tracker
89         if(hasattr(self, '_etracker')):
90             self._etracker.stop()
91             # calculate total energy consumed on cpu
92             energy = self._etracker._total_cpu_energy.kWh + self._etracker._total_ram_energy.kWh
93             del self._etracker
94             self.clear_cache()
95             return energy

```

*Figure 12 Code Snippet of CodeCarbon Usage*

## 4.3 Inference Time

Inference time, encompassing both latency and throughput, was measured on various edge devices on intended edge devices using PyTorch and Torch Profiler.

### 4.3.1 Asynchronous Execution

Unlike single-threaded execution, asynchronous execution in multithreaded programming, particularly on GPUs, must be carefully considered when measuring inference time. Measurements are performed on the CPU device in the process of calculating inference time. However, because GPU execution is asynchronous, the line of code that halts the timing process may be completed first. Consequently, this can lead to inaccuracies in the measured inference time [29]. To address such issues, the PyTorch library offers support for a function, `torch.cuda.synchronize()` [30], which seamlessly synchronizes CPU and GPU operations. Further details on implementation can be found in *Figure 13*, where line 39 awaits GPU synchronization, followed by line 40 for inference time calculation.

```

29     def record_time(self, stop:bool=False)->float:
30
31         if(not stop):
32             self._start = torch.cuda.Event(enable_timing=True)
33             self._start.record() # record start time
34             return
35
36         if(hasattr(self, '_start')):
37             self._end = torch.cuda.Event(enable_timing=True)
38             self._end.record() # record end time
39             torch.cuda.synchronize() # wait for gpu synchronize
40             return self._start.elapsed_time(self._end) # calculate inference time

```

*Figure 13 Code Snippet of Measuring Inference Time*

### 4.3.2 GPU Warmup

Modern GPU devices can function in different power states [29]. When the GPU is idle, it automatically reduces its power consumption to a minimum or may shut down completely. Interaction with the GPU through a program prompts the driver to initialize the GPU. This initialization process may cause a latency of approximately 3 seconds due to the exfoliating nature of the error-correcting code. It is prudent to avoid measuring such side effects, as they do not yield accurate timing and do not reflect actual production environments where GPUs are often already initialized or operating in persistence mode. Therefore, a GPU warmup procedure was established by repeatedly passing a random tensor to the model on the GPU. The code snippet depicting this implementation is provided in *Figure 14*, where lines 80 to 82 denote the GPU warmup process.

```

68  @staticmethod
69  def warm_up(model, input: torch.tensor=None) -> None:
70      model_cp = deepcopy(model)
71      device = torch.device('cuda')
72      # create dummy tensor
73      if(input is None):
74          input = torch.rand(1,3,112,112, dtype=torch.float32,
75                             requires_grad=False)
76      # transfer model and tensor to gpu
77      input = input.to(device)
78      model_cp = model_cp.to(device)
79      # start warmup
80      with model_cp.eval() and torch.no_grad():
81          for _ in tqdm(range(10), 'CUDA Warm Up..'):
82              model_cp(input)
83      del (model_cp, input)
84      CUDA.clear_cache()
85

```

*Figure 14 GPU Warmup*

## 4.4 Memory Usage

In PyTorch, memory utilization during DNN model inference includes both allocated and reserved memory, factors that need to be considered during implementation.

### 4.4.1 Allocated Memory

The allocated memory represents the actual memory utilized by PyTorch tensors and operations throughout the inference process. This memory allocation can be readily monitored using Torch Profiler. *Figure 15* illustrates the implementation details of Torch Profiler, with line 51 marking the start of profiling and line 57 marking its end. Subsequently, line 61 retrieves the allocated memory during the profiling duration.

```

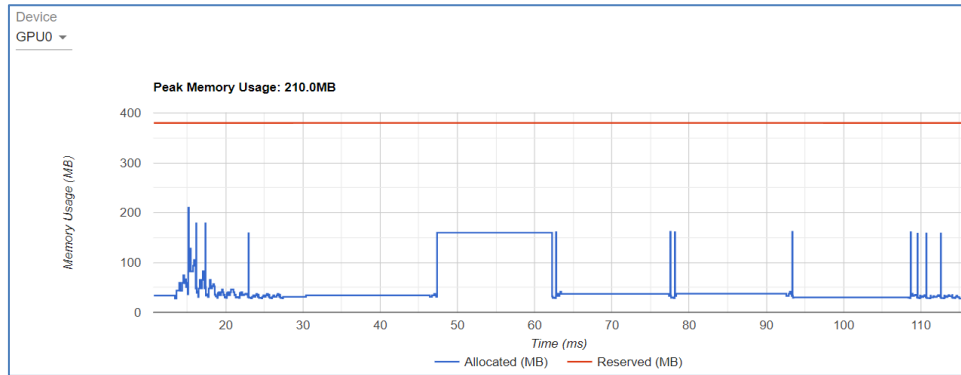
46     def record_memory(self, stop:bool=False)->list:
47         # start torch profiler
48         if(not stop and not self._is_prof_started()):
49             self._started_prof = True
50             self.clear_cache()
51             self._prof.start()
52             return
53
54         if stop:
55             # stop torch profiler
56             if(self._is_prof_started()):
57                 self._prof.stop()
58                 self._started_prof = False
59
60             # get allocated memory
61             return [self._prof.key_averages().total_average().cpu_memory_usage, 0]

```

*Figure 15 Code Snippet of Torch Profiler Implementation*

#### 4.4.2 Reserved Memory

PyTorch employs a caching memory allocator [31] to set aside memory before the operations, facilitating accelerated memory allocations and swift memory deallocation without necessitating device synchronizations. In PyTorch, reserved memory management on GPUs is transparent to users and holds significant importance in ensuring the smooth and effective execution of DNN models on GPU devices. Monitoring reserved memory during inference offers insights into the peak memory requirements of a specific DNN model.



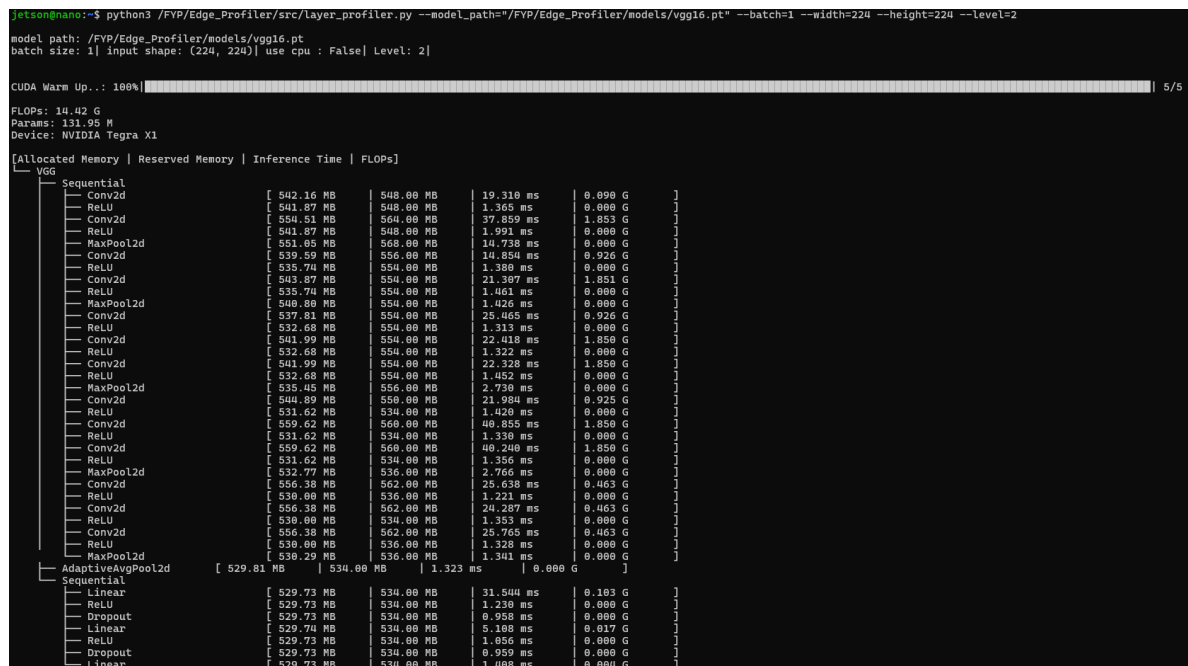
*Figure 16 Reserved and Allocated Memory of Mobilenet\_v2 Model Inference*

#### 4.5 Complexity Tuning Through Compression

The FLOPs of the model were tuned through pruning and quantization techniques. Magnitude-based or structured pruning was employed to identify and remove less important connections from the model. Quantization was implemented by converting model parameters and activations from floating-point precision to lower-precision formats.

### 4.5.1 Layer Based Performance Analysis

Analysing performance layer by layer is beneficial for identifying and pruning resource-intensive layers in neural networks. To facilitate this, PyTorch forward hooks [32] were employed. These hooks allow the registration of a function to execute whenever the forward method of a neural network is called. By implementing forward hooks in each layer, it becomes possible to analyse the performance of each layer individually, aiding in the identification of resource-intensive layers that may benefit from optimization or pruning. The performance analysis results are visualized in *Figure 17*, enabling users to identify the most resource-intensive layer and customize the layer complexity accordingly.

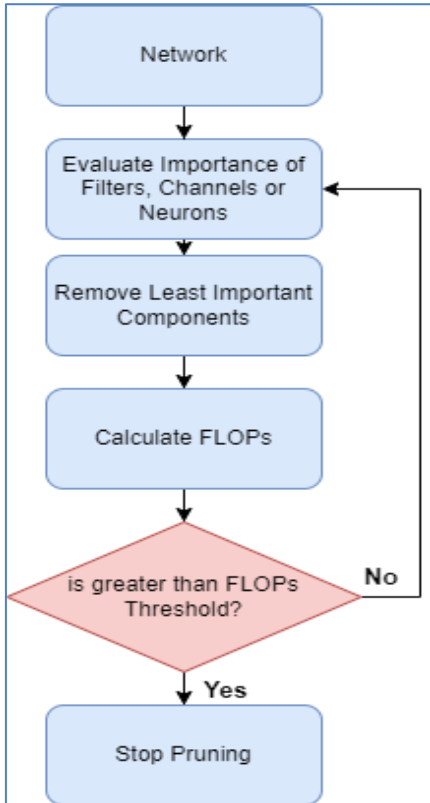


*Figure 17 Illustration of Layer-Based Performance Analysis*



#### 4.5.2 Structured Pruning through Torch-Pruning

Torch-Pruning was used to tune model complexity (FLOPs). *Figure 18* shows the high-level structure of the pruning strategy used in this study. The importance of components in the CNN is initially determined using criteria techniques such as weighted magnitude or Taylor importance [33]. The least significant components are subsequently removed, and the FLOP count is calculated. Pruning will continue until the predefined FLOP threshold is exceeded.



*Figure 18 Pruning Strategy*

## Chapter 5: Result

### 5.1 Hyperparameters

The following plausible set of hyperparameters was selected for the experiments and result demonstrations. To conserve memory and processing time, the maximum batch size was limited by 128, and input image shapes were chosen between the standard size (width: 224, height: 224) and its half to minimize the performance trade-off. The amount of model complexity (FLOPs) to be pruned is expressed as a percentage of the original model, and an approximation is used because tuning FLOPs to a precise quantity is not feasible. The devices and models used in the studies were described in the methodology section.

Hyperparameters	Values
FLOPs Prune Ratio	approx. 2.5%, approx. 5%
Batch Sizes	1, 8, 16, 32, 64, 128
Input Image Shapes (height, width)	(224, 224), (192, 192), (160, 160), (112, 112)

*Table 4 Selected Hyperparameters*

### 5.1 Complexity Tuning

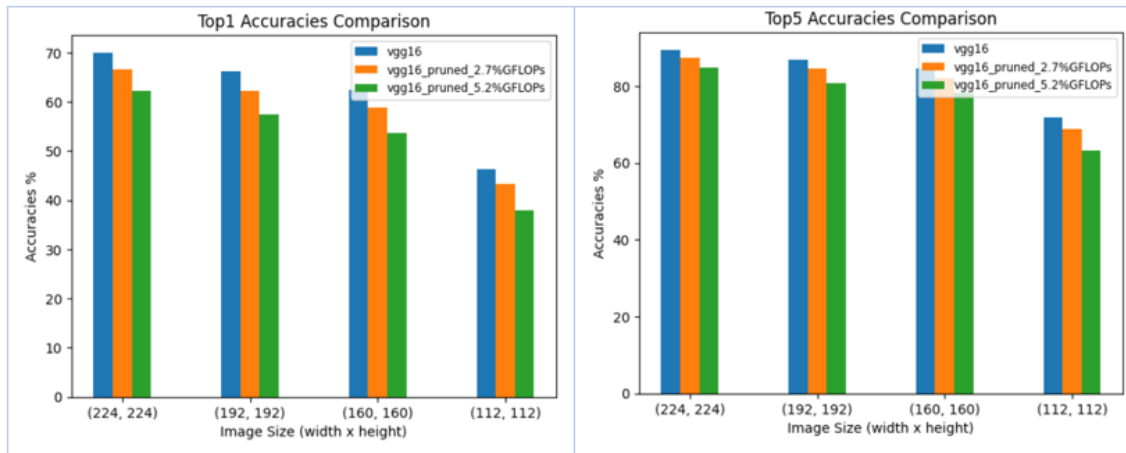
Please take note that the VGG16 model was used specifically to explain the experiment to obtain the result described in this section. Model complexity was tuned using the method described in **4.5.2 Structured Pruning through Torch-Pruning**. *Table 5* presents the Giga FLOPs (GFLOPs) of the pre-trained, 27% of FLOPs pruned, and 52% of FLOPs pruned models. The input shape corresponds to batch size, number of channels, input width, and input height). Models with higher GFLOPs are computationally intensive, and it's noted that the pruned VGG16 model exhibits the lowest resource

demand. GLOPs also increase proportionately with input shapes, as larger forms need more calculations.

Model	Input Shapes			
	(1,3,224,224)	(1,3,192,192)	(1,3,160,160)	(1,3,112,112)
vgg16	15.47 GLOPs	11.4 GFOPs	7.95 GFLOPs	3.96 GLOPs
vgg16_pruned_27%	15.07 GLOPs	11.11 GFOPs	7.75 GFLOPs	3.86 GLOPs
vgg16_pruned_52%	14.68 GLOPs	10.82 GFLOPs	7.55 GFLOPs	3.76 GLOPs

*Table 5 VGG16 Models' Complexities*

The accuracy of the models was then evaluated using ImageNet validation to compare the performance of each model. As shown in *Figure 19* below, the pruned VGG16 model maintains desirable accuracy levels close to those of the pre-trained model.



*Figure 19 VGG16 Models' Accuracy*

## 5.2 Memory Footprint

Profiling memory consumption during inference offers valuable insights into the memory limitations of a specific edge device. *Figure 20* illustrates memory consumption in megabytes (MB) per batch size in conjunction with the input shape. Notably, certain combinations of batch and input shapes were omitted due to exceeding the available memory capacity. It is crucial to consider the static memory size of the model when profiling, which is also allocated in the device's memory for inference. Based on this analysis, it can be inferred that the memory constraint of the Jetson Nano is approx. 2.6 GB for pruned VGG16.

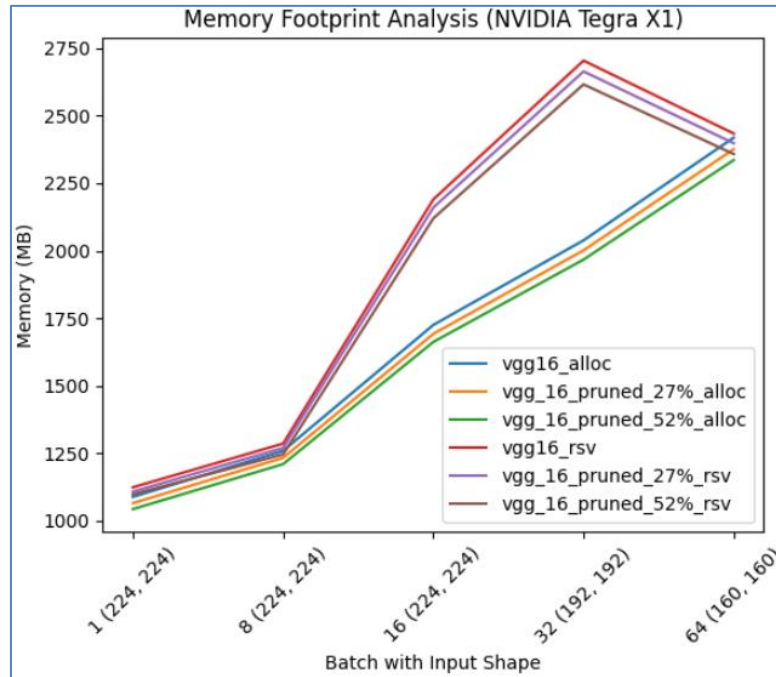


Figure 20 Memory Footprint of Pruned VGG16 on Jetson Nano

### 5.3 Latency and Throughput Profile

The achievement of low latency and high throughput is the optimal performance in real-time systems, particularly on edge devices. The transfer time involved in transferring input images from the CPU to the GPU device was also accumulated in the total inference time, whenever applicable. The left plot in Figure 21 offers insights into throughput, while the right plot portrays latency across various combinations of batch sizes with input shapes. Notably, analysis shows that all models achieved similar latency and throughput, except for pretrained VGG16, which is not a good fit for input size (192, 192).

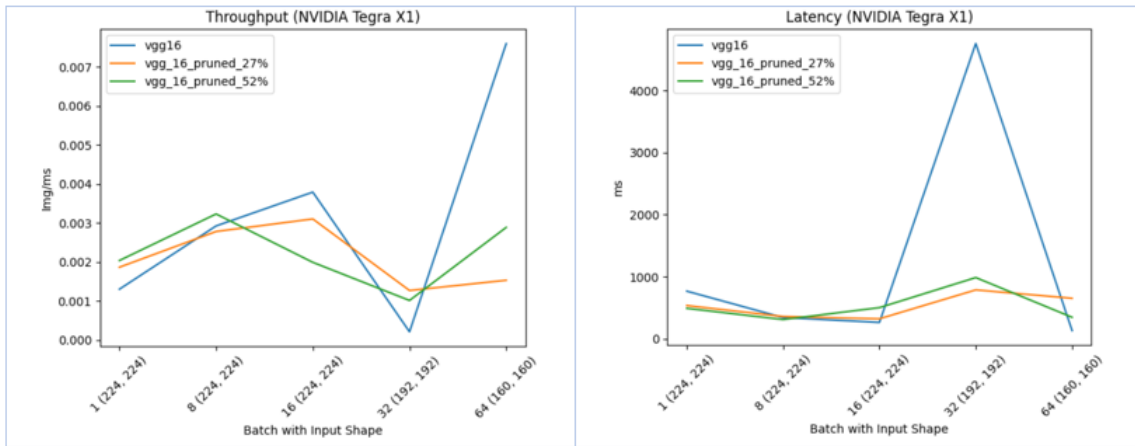


Figure 21 Throughput and Latency of VGG16

### 5.4 Power Consumption

Analysing energy consumption is crucial when deploying models on edge devices with limited battery capacities. Figure 22 illustrates energy usage in kilowatt-hours (kWh) across different batch sizes and input shapes. It is observed that the hyperparameter combo of batch size less than 16 with input size (224 x 224) consumed minimal energy.

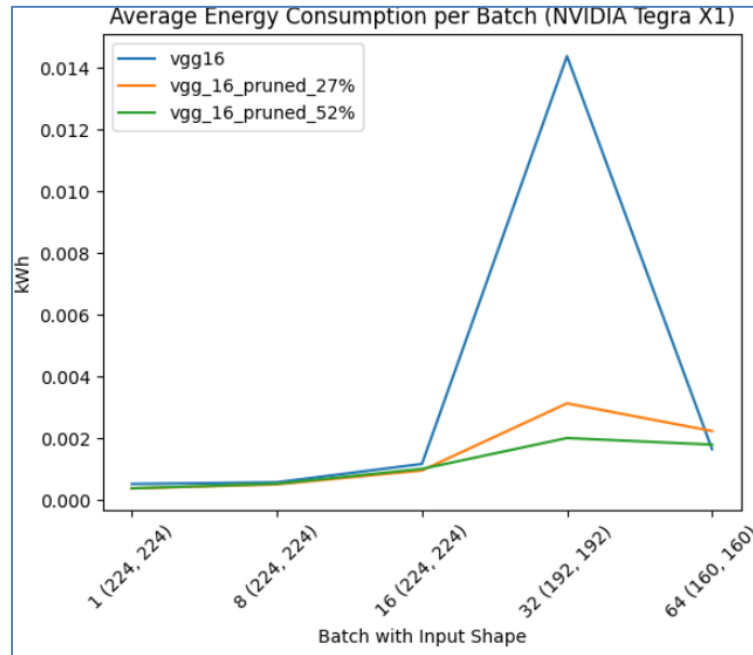


Figure 22 Energy Consumption of VGG16

## 5.5 Result Conclusion

As previously noted, both pretrained and pruned models have equal memory requirements. The hyperparameter combination of batch size 16 and picture size (224, 224) uses around 0.004 images per millisecond of throughput, 500ms of delay, and 0.001 kWh while retaining the highest accuracy across all models. To summarise, pretrained VGG16 with batch size 16 and input size (224, 224) is considered the best for inference on Jetson Nano.

## **Chapter 6: Conclusion and Future Work**

### **Chapter 6.1: Conclusion**

In essence, completing this FYP successfully was a fabulous and fruitful learning experience for the author. With the implementation of deep learning architectures on edge devices, the method of incorporating hyperparameter optimization and identifying experience. Creating a successful program that takes into account a variety of factors, such as deep learning algorithms, edge hardware architectures, and resource allocations, is an extremely difficult task. However, with focused work and patience, it is feasible to develop effective and efficient solutions. Success in any endeavour necessitates effort, hard work, and a willingness to investigate and confront diverse issues.

Furthermore, it is critical to recognize that deep learning is a fast-expanding field, with various new algorithms constantly appearing and several approaches that may produce better outcomes than those reported in this research. Despite the inherent difficulties of embarking on a deep learning project incorporating hardware, it is incredibly rewarding to walk this route and eventually fulfil the project's goals, especially under the guidance of the supervisor and mentor.

Undoubtedly, the author had various problems throughout the project. One of the most important components of any project is effective time management, particularly in the field of deep learning-related endeavours, which requires research and knowledge of various mechanisms and algorithms. Similarly, as a part-time undergraduate student juggling academic responsibilities with a full-time engineering job, the author encountered substantial difficulties in successfully managing time while working on this project. Regardless of the number of challenges the author experienced, completing this project's goal was a proud learning experience.

## **Chapter 6.2: Future Work**

Although the findings and analysis reported in this work are useful for deploying deep learning models on edge devices, there are various areas for further development in this FYP program. To boost the program's compatibility with a variety of frameworks, one possible improvement would be to enable more deep-learning frameworks, such as TensorFlow, by utilizing a common format like ONNX. Furthermore, the program may be modified to support a larger range of deep learning applications and architectures, such as rapid R-CNN for object detection and attention mechanisms in natural language processing. Finally, the integration with edge device emulators would allow targeted users to experiment with deep learning model inference and gain insights into resource ceilings without having to purchase actual edge hardware. These additional suggestions may broaden the program's variety, functionality, and accessibility in future applications.



## Appendix

Experiments were conducted on two other models, which are MobileNet and ResNet50. The following tables and figures represent the FLOPs and resource requirements of Resnet50 models. These insights not only reveal optimal hyperparameters for inference but also show the resource limits of the edge device.

Model	Input Shapes			
	(1,3,224,224)	(1,3,192,192)	(1,3,160,160)	(1,3,112,112)
resnet50	4.09 GLOPs	3.00 GFOPs	2.09 GFLOPs	1.08 GLOPs
resnet50_pruned_25%	3.99 GLOPs	2.93 GFOPs	2.04 GFLOPs	1.05 GLOPs
resnet50_pruned_52%	3.89 GLOPs	2.86 GFLOPs	1.98 GFLOPs	1.02 GLOPs

*Table 6 FLOPs of Resnet50*

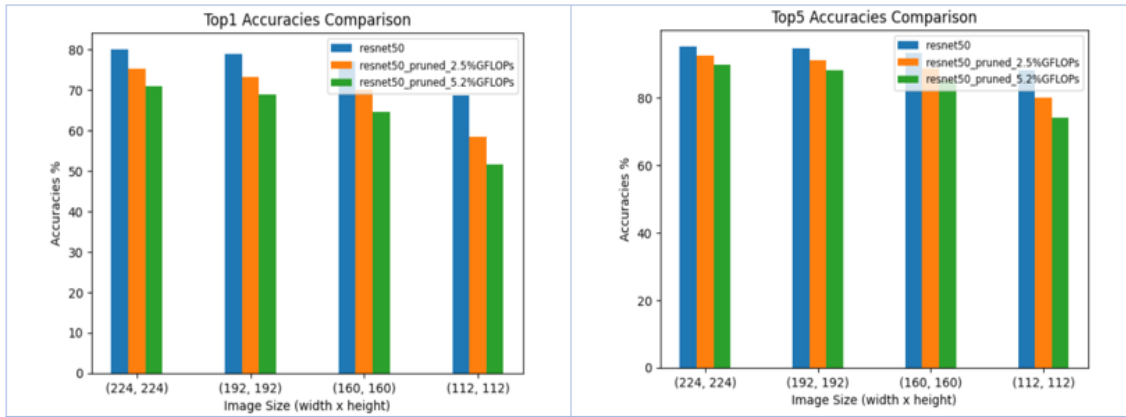


Figure 23 Accuracies of Resnet50

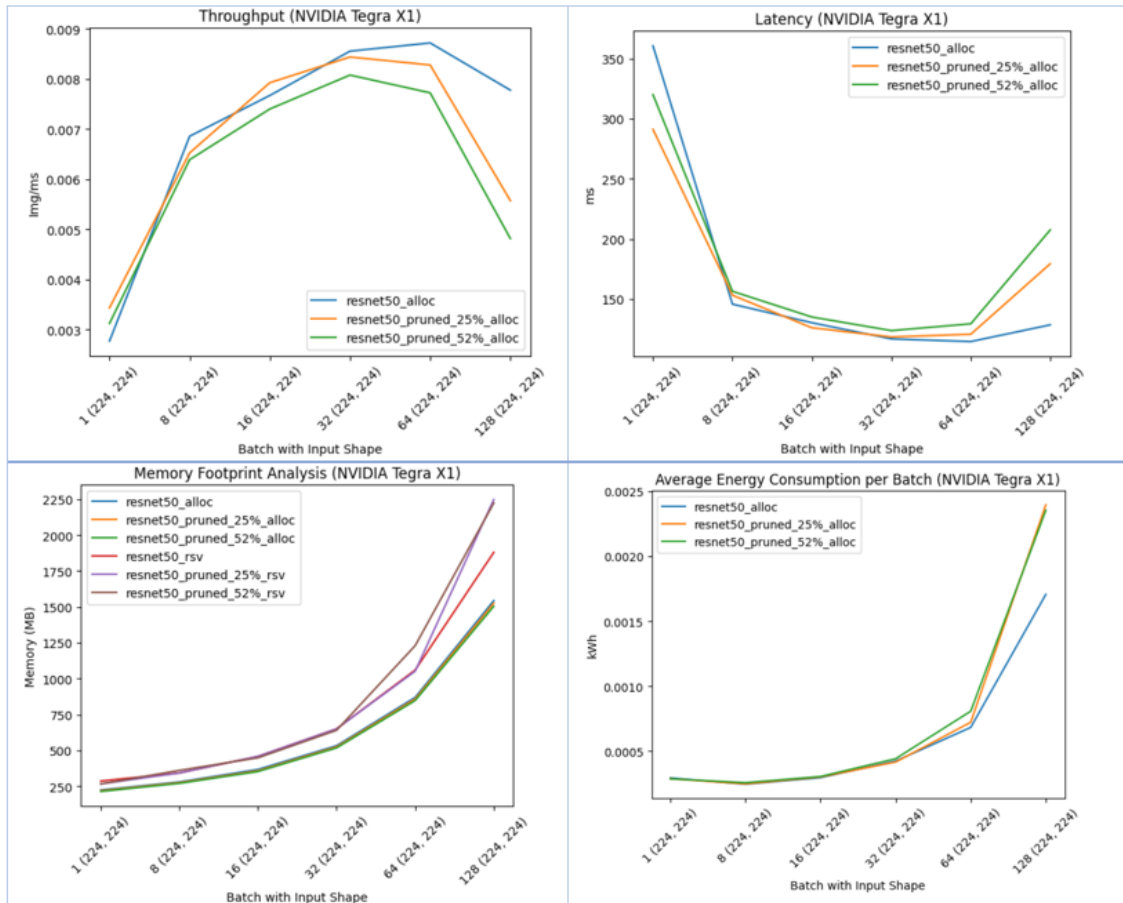
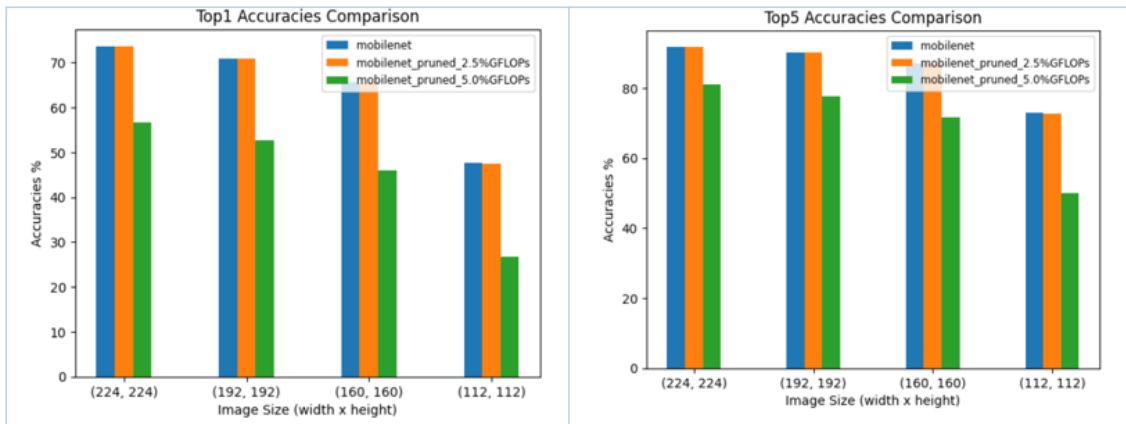


Figure 24 Resource Requirements of Resnet50

The following tables and figures represent the FLOPs and resource requirements of MobileNet models.

Model	Input Shapes			
	(1,3,224,224)	(1,3,192,192)	(1,3,160,160)	(1,3,112,112)
mobilenet	0.22 GLOPs	0.16 GFOPs	0.11 GFLOPs	0.06 GLOPs
mobilenet_pruned_25%	0.22 GLOPs	0.16 GFOPs	0.11 GFLOPs	0.06 GLOPs
mobilenet_pruned_50%	0.21 GLOPs	0.15 GFLOPs	0.11 GFLOPs	0.06 GLOPs

*Table 7 FLOPs of MobileNet*



*Figure 25 Accuracies of MobileNet*

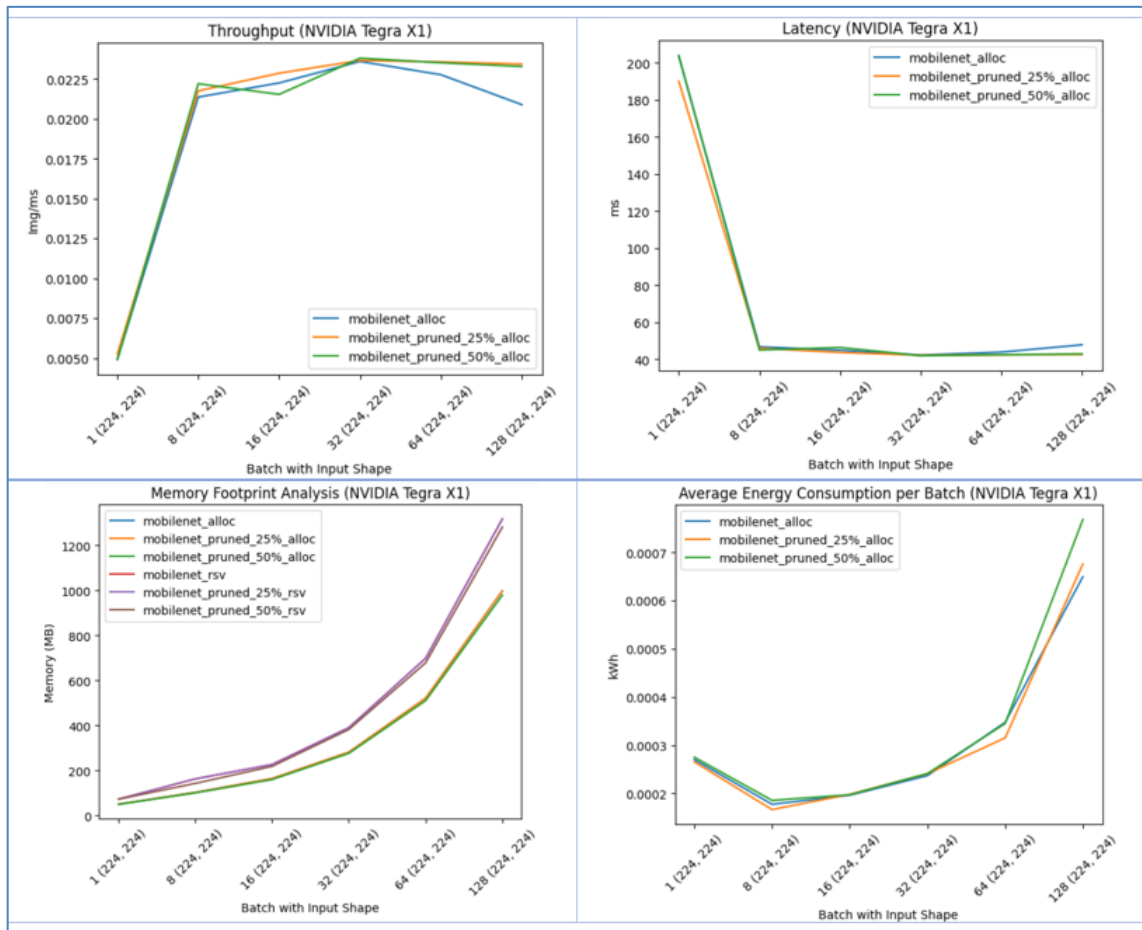


Figure 26 Resource Requirements of MobileNet

## References

- [1] Stanford - Spring 2023, “CS231n: Convolutional Neural Networks for Visual Recognition,” [Online]. Available: <https://cs231n.github.io/convolutional-networks/>.
- [2] “Convolutional Neural Networks,” [Online]. Available: <https://paperswithcode.com/methods/category/convolutional-neural-networks>.
- [3] W. G. Hatcher and W. Yu, “A Survey of Deep Learning: Platforms, Applications and Emerging Research Trends,” [Online]. Available: <https://ieeexplore.ieee.org/document/8351898>.
- [4] M. Ciznicki, K. Kurowski and A. Plaza, “GPU Implementation of JPEG2000 for Hyperspectral Image Compression,” [Online]. Available: [https://www.researchgate.net/publication/253881559\\_GPU\\_Implementation\\_of\\_JPEG2000\\_for\\_Hyperspectral\\_Image\\_Compression](https://www.researchgate.net/publication/253881559_GPU_Implementation_of_JPEG2000_for_Hyperspectral_Image_Compression).
- [5] Y. LeCun, “Deep Learning Hardware: Past, Present, and Future,” [Online]. Available: <https://ieeexplore.ieee.org/document/8662396>.
- [6] S. Voghoei, N. H. Tonekaboni, J. G. Wallace and H. R. Arabnia, “Deep Learning at the Edge,” [Online]. Available: <https://arxiv.org/abs/1910.10231>.
- [7] J. Chen and X. Ran, “Deep Learning With Edge Computing: A Review,” [Online]. Available: [https://www.researchgate.net/figure/Architectures-for-deep-learning-inference-with-edge-computing-a-On-device-computation\\_fig5\\_334489669](https://www.researchgate.net/figure/Architectures-for-deep-learning-inference-with-edge-computing-a-On-device-computation_fig5_334489669).
- [8] N. A. Baghdadi, A. S. Maklad, A. Malki and M. Deif, “Reliable Sarcoidosis Detection Using Chest X-Rays with EfficientNets and Stain Normalization Techniques,” [Online]. Available: [https://www.researchgate.net/figure/Schematic-illustration-of-VGG16-architecture-AlexNet-The-AlexNet-45-model-which-uses\\_fig3\\_360702351](https://www.researchgate.net/figure/Schematic-illustration-of-VGG16-architecture-AlexNet-The-AlexNet-45-model-which-uses_fig3_360702351).
- [9] NVIDIA, “Maxwell Architecture,” [Online]. Available: <https://developer.nvidia.com/maxwell-compute-architecture>.
- [10] NVIDIA, “GeForce RTX 3070 Family,” [Online]. Available: <https://www.nvidia.com/en-sg/geforce/graphics-cards/30-series/rtx-3070-3070ti/>.
- [11] NVIDIA, “Jetson Nano,” [Online]. Available: <https://developer.nvidia.com/embedded/jetson-nano>.

- [12] PyTorch, “PyTorch Get Started,” [Online]. Available: <https://pytorch.org/get-started/locally/>.
- [13] PyTorch, “Computational Graph,” [Online]. Available: [https://pytorch.org/tutorials/beginner/blitz/autograd\\_tutorial.html#computational-graph](https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#computational-graph).
- [14] ImageNet, “ImageNet,” [Online]. Available: <https://www.image-net.org/>.
- [15] COCO, “COCO Dataset,” [Online]. Available: <https://cocodataset.org/>.
- [16] PyTorch, “PyTorch Classification Weights,” [Online]. Available: <https://pytorch.org/vision/stable/models.html#table-of-all-available-classification-weights..>
- [17] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, “mageNet: A large-scale hierarchical image database,” [Online]. Available: <https://ieeexplore.ieee.org/document/5206848..>
- [18] PyTorch, “torchvision.transforms,” [Online]. Available: <https://pytorch.org/vision/0.9/transforms.html>.
- [19] PyTorch, “Torch Profiler,” [Online]. Available: <https://pytorch.org/docs/stable/profiler.html#torch.profiler.profile>.
- [20] CodeCarbon, “CodeCarbon,” [Online]. Available: <https://mlco2.github.io/codecarbon/>.
- [21] Intel, “Intel Power Gadget,” [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html>.
- [22] NVIDIA, “pynvml,” [Online]. Available: <https://pypi.org/project/pynvml/>.
- [23] A. Asperti, D. Evangelista and M. Marzolla, “Dissecting FLOPs along input dimensions for GreenAI cost estimations,” [Online]. Available: <https://arxiv.org/abs/2107.11949>.
- [24] OpenVINO, “OpenVINO,” [Online]. Available: <https://docs.openvino.ai/archive/2021.4/index.html>.
- [25] “Taylor Importance,” [Online]. Available: [https://openturns.github.io/openturns/latest/theory/reliability\\_sensitivity/taylor\\_importance\\_factors.html](https://openturns.github.io/openturns/latest/theory/reliability_sensitivity/taylor_importance_factors.html).
- [26] “Pruning Overview,” [Online]. Available: <https://neuralmagic.com/blog/pruning-overview/>.
- [27] Fang, G. a. Ma, X. a. Song, M. a. Mi, M. B. a. Wang and Xinchao, “Torch-Pruning,” [Online]. Available: <https://github.com/VainF/Torch-Pruning?tab=readme-ov-file>.

- [28] G. Fang, X. Ma, M. Song, M. B. Mi and X. Wang, “DepGraph: Towards Any Structural Pruning,” [Online]. Available: <https://arxiv.org/abs/2301.12900>.
- [29] A. Geifman, “The Correct Way to Measure Inference Time of Deep Neural Networks,” [Online]. Available: <https://deci.ai/blog/measure-inference-time-deep-neural-networks/>.
- [30] PyTorch, “torch.cuda.synchronize,” [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.cuda.synchronize.html>.
- [31] PyTorch, “Memory management,” [Online]. Available: <https://pytorch.org/docs/stable/notes/cuda.html#memory-management>.
- [32] PyTorch, “PyTorch Hook,” [Online]. Available: [https://pytorch.org/docs/stable/generated/torch.nn.modules.module.register\\_module\\_forward\\_hook.html](https://pytorch.org/docs/stable/generated/torch.nn.modules.module.register_module_forward_hook.html).
- [33] P. Molchanov, S. Tyree, T. Karras, T. Aila and J. Kautz, “PRUNING CONVOLUTIONAL NEURAL NETWORKS FOR RESOURCE EFFICIENT INFERENCE,” [Online]. Available: <https://arxiv.org/pdf/1611.06440.pdf>.