NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

MODEL OPTIMIZATION FOR VEHICLE RECOGNITION ON EDGE DEVICES

by

Zong Long Goh

June 2021

Thesis Advisor: Gurminder Singh
Co-Advisor: Marko Orescanin

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2021	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
MODEL OPTIMIZATION FOR VEHICLE RECOGNITION ON EDGE DEVICES

5. FUNDING NUMBERS

6. AUTHOR(S) Zong Long Goh

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Video surveillance is commonly used for the protection of military installations. Within video surveillance, artificial intelligence (AI) techniques are often incorporated for object recognition and motion tracking. Network communication and stable power are usually required to operate such systems. Hence, they are not often deployed in remote areas where stable network connectivity and power supply cannot be supported. The emergence of lightweight edge devices with low power requirements and high processing power to run AI, however, has offered an avenue to deploy AI in remote areas. Thus, the focus shifts to the type of AI used in video surveillance systems. One approach is machine learning (ML), in which the ML models need to be trained and optimized within network and power constraints while maintaining good inference performance.

This research explores ML for vehicle recognition via transfer learning of various state-of-the-art convolutional neural network models. Also, we study the effects of applying optimization techniques, pruning, and quantization, to improve performance and allow for deployment of the models on an edge device, the Raspberry Pi 4. This study found that the MobileNet model, when trained on a vehicle's dataset and optimized with post-training weights pruning and full integer quantization, achieves an inference accuracy of 81.88% with a latency of 132 ms and a compressed model size of 3.44 MB, making it viable for real-time inference applications.

14. SUBJECT TERMS convolutional neural networks, vehicle classification, vehicle recognition, surveillance, model optimization, edge device, quantization, pruning, transfer learning, data augmentation	**15. NUMBER OF PAGES** 75
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

MODEL OPTIMIZATION FOR VEHICLE RECOGNITION ON EDGE DEVICES

Zong Long Goh
Civilian, Ministry of Defence, Singapore
BCE, Nanyang Technological University, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2021

Approved by: Gurminder Singh
Advisor

Marko Orescanin
Co-Advisor

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Video surveillance is commonly used for the protection of military installations. Within video surveillance, artificial intelligence (AI) techniques are often incorporated for object recognition and motion tracking. Network communication and stable power are usually required to operate such systems. Hence, they are not often deployed in remote areas where stable network connectivity and power supply cannot be supported. The emergence of lightweight edge devices with low power requirements and high processing power to run AI, however, has offered an avenue to deploy AI in remote areas. Thus, the focus shifts to the type of AI used in video surveillance systems. One approach is machine learning (ML), in which the ML models need to be trained and optimized within network and power constraints while maintaining good inference performance.

This research explores ML for vehicle recognition via transfer learning of various state-of-the-art convolutional neural network models. Also, we study the effects of applying optimization techniques, pruning, and quantization, to improve performance and allow for deployment of the models on an edge device, the Raspberry Pi 4. This study found that the MobileNet model, when trained on a vehicle's dataset and optimized with post-training weights pruning and full integer quantization, achieves an inference accuracy of 81.88% with a latency of 132 ms and a compressed model size of 3.44 MB, making it viable for real-time inference applications.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
A. MOTIVATION....2
B. OBJECTIVES....2
C. CONTRIBUTIONS....2
D. THESIS ORGANIZATION....3

II. BACKGROUND AND RELATED WORKS....5
A. RELATED RESEARCH....5
B. DEEP LEARNING....6
1. Feed Forward Neural Network....6
2. Convolutional Neural Networks....7
3. Training Techniques....12
4. Development Frameworks....14
C. DEEP LEARNING ON EDGE DEVICES....15
D. CHAPTER SUMMARY....16

III. METHODOLOGY....17
A. SYSTEM ARCHITECTURE....17
B. DATASET....19
C. CLASSIFIER ARCHITECTURE....22
1. Design....22
2. Pre-trained Model Selection....25
3. Top Layers....26
D. MODEL OPTIMIZATION FOR PERFORMANCE....28
1. Pruning....28
2. Quantization....29
E. TRAINING AND TESTING SETUP....30
F. CHAPTER SUMMARY....33

IV. RESULTS AND ANALYSIS....35
A. OVERVIEW....35
B. MODEL PERFORMANCE....36
1. Effects of Optimization....36
2. Model Selection for Vehicle Recognition....44
C. CHAPTER SUMMARY....45

V. CONCLUSIONS AND FUTURE WORK 47
A. SUMMARY 47
B. FUTURE WORK 48

LIST OF REFERENCES 51

INITIAL DISTRIBUTION LIST 57

LIST OF FIGURES

Figure 1. Simple Feed Forward Neural Network. Adapted from [8]. 7

Figure 2. CNN Sequence for Vehicle Recognition. Source: [10]. 8

Figure 3. Convolutional Layer Illustration. Adapted from [14]. 9

Figure 4. Zero Padding. Source: [17]. 9

Figure 5. Activation Functions. Source: [20]. 10

Figure 6. Pooling Layer Illustration. Adapted from [14]. 11

Figure 7. Fully Connected Layer Illustration. Adapted from [14]. 11

Figure 8. How TensorFlow Lite Works. Source: [6]. 15

Figure 9. System Architecture. 18

Figure 10. Example Images from Stanford Cars Dataset. Source: [34]. 20

Figure 11. Example Images with Data Augmentation. Adapted from [34]. 22

Figure 12. Keras Classifier Architecture. 23

Figure 13. TF Lite Classifier Architecture. 23

Figure 14. Experimental Classifier. 24

Figure 15. Classifier Top Layers. 26

Figure 16. Initial MobileNetV2 Training and Validation Accuracy. 27

Figure 17. Improved MobileNetV2 Training and Validation Accuracy. 28

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Available Keras Application Models. Source: [32]. 25

Table 2. Keras Models Training Parameters 31

Table 3. TF Lite Models Training Parameters 32

Table 4. Models Test Parameters 33

Table 5. Pruning on Keras Models 37

Table 6. Quantized Keras Applications Models (Compressed Size) 39

Table 7. Quantized TF Lite Models (Compressed Size) 41

Table 8. Quantized Keras Applications Models (Latency) 42

Table 9. Quantized TF Lite Models (Latency) 43

Table 10. Keras and TF Lite Models with Above 80% Accuracy 45

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
CNN	Convolutional Neural Networks
Colab	Colaboratory
CONV	Convolutional
CPU	Central Processing Unit
CV	Computer Vision
DNN	Deep Neural Network
FNN	Feed Forward Neural Network
GPU	Graphical Processing Unit
ML	Machine Learning
NLP	Natural Language Processing
POOL	Pooling
RNN	Recurrent Neural Network
RPi	Raspberry Pi
TF	TensorFlow
TFMOT	TensorFlow Model Optimization Toolkit
TPU	Tensor Processing Unit

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to my thesis advisor, Dr. Gurminder Singh, for accepting me as his thesis student and offering insightful advice on getting my thesis done on time. I am grateful for his patience and support that allowed me to complete my thesis through the extraordinary times of the Covid-19 pandemic.

I also want to thank my thesis co-advisor, Dr. Marko Orescanin, for his time and expert guidance on the topic of machine learning. He has offered me exceptional advice on getting me started on the subject and helped me navigate through the many unknowns.

Lastly, I am eternally thankful to my loved ones for all their love, sacrifice, and unwavering support throughout my time at the Naval Postgraduate School.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Video surveillance is an essential tool for the security and protection of a facility. Be it highly secure environments such as military bases, airports, power plants, or environments with less stringent security requirements such as in schools and shopping malls, video surveillance is prevalent. Systems deployed for surveillance purposes have long relied on recorded videos for post-incident reviews. Several cameras are typically deployed in a facility to recognize and track an object of interest. Often, manual reviews of copious amounts of footage are required to identify a target and track it across different camera views to deduce a suspect's motive. This task is time-consuming, attention-intensive, error-prone, and laborious.

In order to achieve real-time incident detection and reduce the laborious task of manual post-incident reviews, Artificial Intelligence (AI) techniques are commonly used for object recognition, detection, and motion tracking scenarios. One such application is to use deep learning methods to recognize and identify objects, such as a human or a vehicle, and build an understanding of the object's behavior and its path of approach across a series of cameras [1]. This understanding can be achieved by the correlation between learned information from the parsed image/video capture of each camera. Yet, the deployment of such intelligent camera nodes constantly communicating and running deep learning algorithms for object recognition and detection usually requires high network communication bandwidth and a stable supply of power. Consequently, this method is not commonly deployed in remote areas where the power supply may be low and inconsistent, leading to surveillance outages and poor network connectivity and bandwidth.

The emergence of lightweight edge devices has offered an avenue to apply distributed deep learning methods for video surveillance. Drawing low power and being able to run on batteries, these devices enable the deployment of intelligent surveillance to areas with an inconsistent power supply. With that problem solved, the only concern left for this video surveillance approach is the training and optimization of machine learning (ML) models to run within the network bandwidth and power supply constraints while maintaining reasonably good inference accuracy.

A. MOTIVATION

Seeking to enhance surveillance by incorporating deep learning methods in a distributed learning fashion and deploying the system to remote areas with low network bandwidth and low or inconsistent power supply, we recognize the advantages of running video surveillance systems on edge devices in such scenarios. Hence, we think it is crucial to study the resultant effects of training and optimizing ML image classification models for deployment on edge devices.

B. OBJECTIVES

This research explores training deep learning models for vehicle recognition and studies the resultant effects of applying specific model optimization techniques, pruning and quantization, on the models to eventually identify the best performing model for vehicle recognition on a lightweight edge device. The models, originally pre-trained for generic image classification, are transfer learned on a vehicles dataset and optimized to allow the models to run on an edge device. The models' performance is evaluated based on three main criteria: model compressed size, inference latency, and accuracy.

C. CONTRIBUTIONS

This work makes three key contributions to achieving the deployment of deep learning methods for vehicle recognition on edge devices. Ultimately, the work is done in support of a greater goal of having a distributed learning surveillance system for installation security in remote areas.

The three key contributions of this work are:

1. The identification of pre-trained deep learning models suitable to be trained for deployment on an edge device when transfer learned on an vehicles dataset.
2. The demonstration of the application and improvements gained from applying pruning and quantization optimization techniques on the models.

3. The identification of the best-performing model, trained and optimized to run on an edge device, evaluated based on compressed model size, inference latency, and accuracy.

D. THESIS ORGANIZATION

Following this chapter, this thesis is organized as follows:

Chapter II discusses related works on the application of deep learning for vehicle recognition. It provides background on deep learning concepts and available optimization techniques. It also covers the development framework and hardware used in this study.

Chapter III covers the implementation approach to train and optimize the pre-trained models. It discusses the challenges faced and methods used to improve training and inference accuracy.

Chapter IV presents the experimental results and analysis of the various optimization techniques applied to the models when run on an edge device.

Chapter V summarizes and concludes the study. Areas for possible future research are also discussed.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND AND RELATED WORKS

This study explores the employment of Convolutional Neural Networks (CNN) to identify cars by make, model, and year. The objective is to train, optimize, and evaluate CNN models, enabling inference on small, low-power edge devices. We examine various models, efficient training techniques, and optimization methods towards achieving the objective. This chapter reviews related research on applying deep learning for vehicle recognition and examines concepts in deep learning and training techniques to optimize the model for execution on the edge devices.

A. RELATED RESEARCH

Deep learning has been extensively used in the field of object detection. One of the most common deep learning networks used for object detection is the CNN. Jerry Wei [2] describes CNN as robust models that are easy to control and train and seldom overfit when trained on large image datasets. Significant computational power, however, is required to train CNNs on high-resolution images [2].

With much ongoing application and research, CNNs turn out to be well-suited for vehicle detection and recognition tasks. Xingcheng Luo et al. [3]. used a large image dataset and increased the layers in AlexNet to achieve vehicle and facial recognition accuracy of up to 97.51% and 91.22%, respectively. Hyo Jong Lee et al. [4]. extracted frontal views of vehicle images and fed them into SqueezeNet for training and testing. Albeit running on a desktop Central Processing Unit (CPU) with a powerful Graphical Processing Unit (GPU) setup, the study managed to achieve a 96.3% recognition accuracy with the inference tasks running at a mean of 108 ms. Their model also required less than 5 MB of space, making it broadly viable for real-time inference applications.

The proven success of applying CNN to vehicle recognition tasks and the advancement of small embedded systems, such as dedicated GPU boards and Tensor Processing Unit (TPU) accelerators coupled with lightweight micro-computers, has led to the exploration of running vehicle recognition tasks on such devices. Sanghyeop Lee et al. [5]. ran AlexNet on a dedicated NVIDIA Jetson TX1 board system for vehicle license plate

recognition and achieved accuracy of 95.24%. In a study [6] on objection detection and tracking, a MobileNet TensorFlow (TF) model was converted to a TF Lite object detection model and was run on a Raspberry Pi (RPi) with Coral Edge TPU USB Accelerator, achieving tracking speeds of 24 fps.

The ability to achieve automated vehicle identification through the classification of make, model, year, color, and identification of the license plate is crucial for security and traffic surveillance systems. As such, many studies apply deep learning models, CNNs in particular, on vehicle detection and recognition. Yet most studies execute deep learning model training and inference on systems with powerful CPUs and GPUs. This study focuses on furthering the research of employing CNN on a lightweight device, such as an RPi, through transfer learning on pre-trained models and evaluates various optimization techniques such as pruning and quantization to find the optimum configuration for vehicle recognition.

B. DEEP LEARNING

AI includes ML, where, through experience, machines acquire skills necessary to complete a specific task without human involvement. Deep learning is a subspecialty of ML, and it models the human brain's biological neural network. The result is an artificial neural network, and most advancements of AI in recent years revolve around CNN. This section reviews various concepts related to CNN.

1. Feed Forward Neural Network

A Feed Forward Neural Network (FNN) is a typical neural network that consists of many connected nodes, called neurons, arranged in layers. Input neurons are activated by sensors perceiving the environment or by input data like images, and neurons in subsequent layers are activated via weighted connections from active neurons of the prior layer [7]. Due to its simplicity, the most common model in artificial neural networks is the FNN, where signals propagate in one direction from input to output nodes.

Figure 1 depicts a simple model of an FNN, consisting of neurons aggregated into layers; input data could be training data or environmental data, which is fed into the first

layer, putting the data through an activation function and passing the output on to the hidden layer. The process repeats from the hidden layer to the output layer, eventually producing a set of weights that define the network model. It can then produce a prediction result based on the set of trained weights and an input.

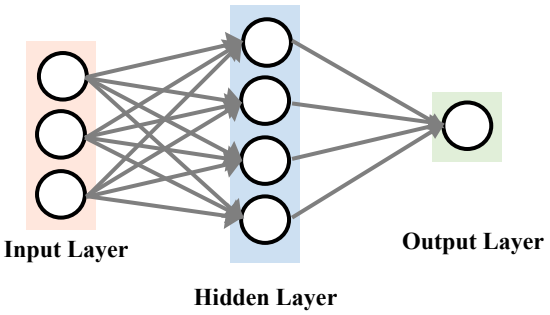

Figure 1. Simple Feed Forward Neural Network. Adapted from [8].

2. Convolutional Neural Networks

An FNN that works particularly well on classifying images is the CNN. It is built upon a sequence of layers of neurons that have learnable weights and biases. A dot product is performed on some inputs, sometimes followed by a non-linearity function, and each neuron's output is fed into the following layers [9]. CNN architectures typically consist of four main layers: Convolutional Layer, Activation Layer, Pooling Layer, and Fully Connected Layer. They are stacked to form a complete CNN architecture.

Figure 2 shows a CNN sequence applied to the task of vehicle recognition. We shall review the essential layers for a CNN in the following sections.

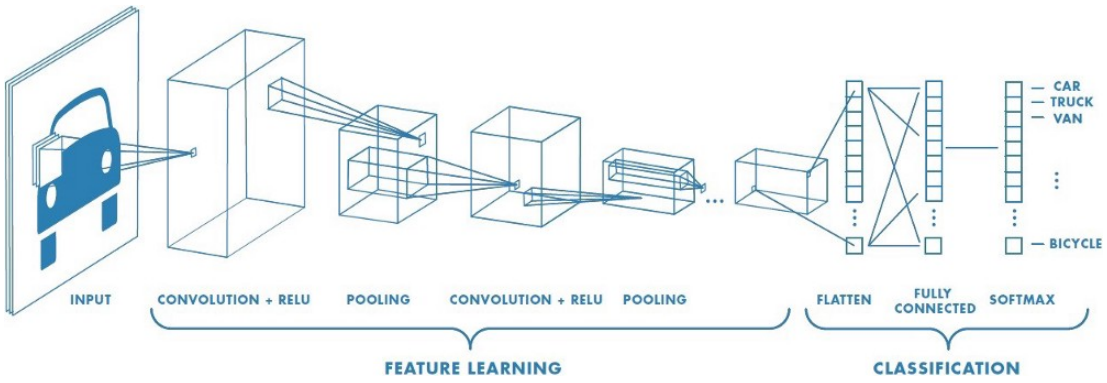

Figure 2. CNN Sequence for Vehicle Recognition. Source: [10].

a. Convolutional Layer

The Convolutional (CONV) layer is the critical layer of a CNN and also the most computationally intensive layer [11]. The CONV layer uses filters, which are the "neurons" of the layer. A filter can be applied to any object in an image; for images of vehicles, a filter could be associated with distinguishing car logos. The logo filter would indicate how strongly the logo appears in the image, noting the count and location of appearances [12]. At the construction of a CNN, filter values are randomly specified, and they will be continuously updated as the network is trained. It is improbable that two identical filters are produced unless the number of chosen filters is extremely large [13].

Adapted from [14], Figure 3 depicts a filter scanning across the entire input layer, moving one pixel at a time where each position can possibly activate a neuron. The output is then collected and forms a feature map. If the CONV layer is an input layer, then the input will be pixel values, for example, 0–255. However, when a CONV layer is deeper in the CNN architecture, a feature map derived from the previous layer will be its input [14].

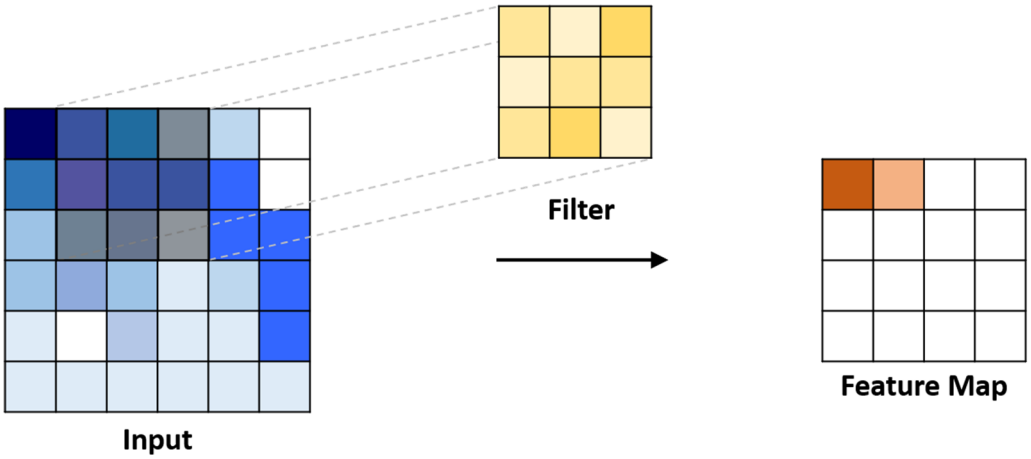

Figure 3. Convolutional Layer Illustration. Adapted from [14].

As shown in Figure 4, when the size of the previous layer cannot be cleanly divided by the filter boundaries and the stride length, a technique used is to insert pad values to serve as mock inputs, called zero padding [15]. Full padding is applied to ensure that all pixels are visited the same number of times by the filter, and it increases the size of the output. The same padding, on the other hand, ensures the output is the same size as the input [16].

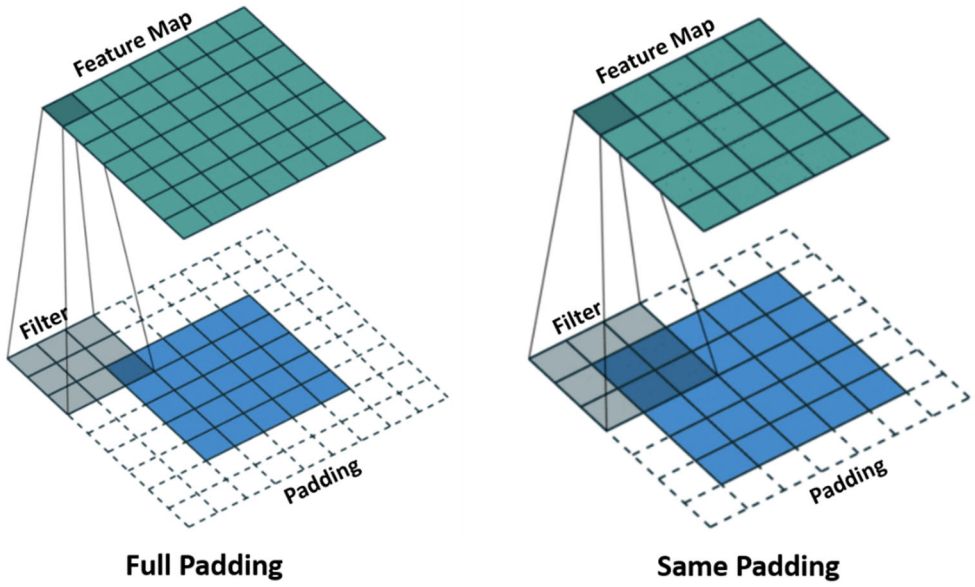

Figure 4. Zero Padding. Source: [17].

b. Activation Layer

The activation layer is placed at the end of a neural network or between CONV layers. The activation layer decides whether a neuron activates or "fires." It essentially does a non-linear transformation on the input signal, and the transformed output is then sent to the next layer as input [18]. A widely used activation function is the ReLU function. As it does not activate all neurons simultaneously, its advantage over other activation functions is computation efficiency.

Figure 5 shows that the ReLU function converts all negative inputs to zero, and unlike Sigmoid and tanh functions, it does not saturate at the positive range. Further, [19] shares that ReLU enables models to converge six times faster than applying tanh and Sigmoid functions.

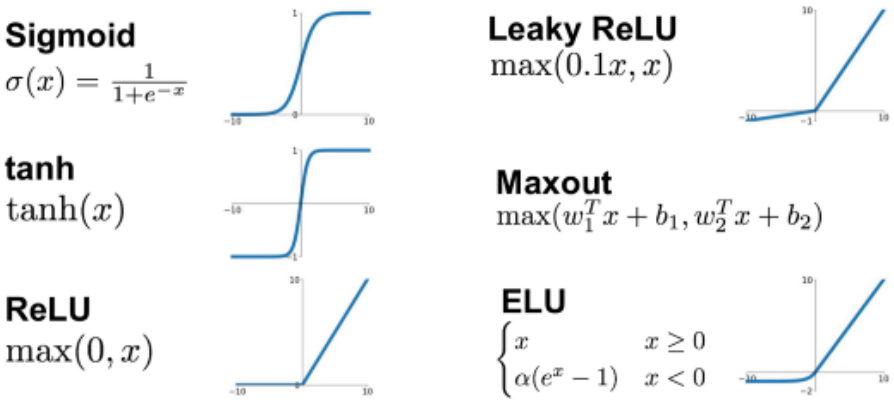

Figure 5. Activation Functions. Source: [20].

c. Pooling Layer

Pooling (POOL) layers are commonly placed after one or more CONV layers and are used to reduce the dimensions of the previous layer's feature map. The use of POOL layers can be considered a technique to reduce dimensionality and generalize feature representations, reducing overfitting [21]. POOL layers are often simple and perform a specific function. Figure 6 depicts the process of taking the maximum value or average value in a filter region, named Max Pooling or Average Pooling, respectively.

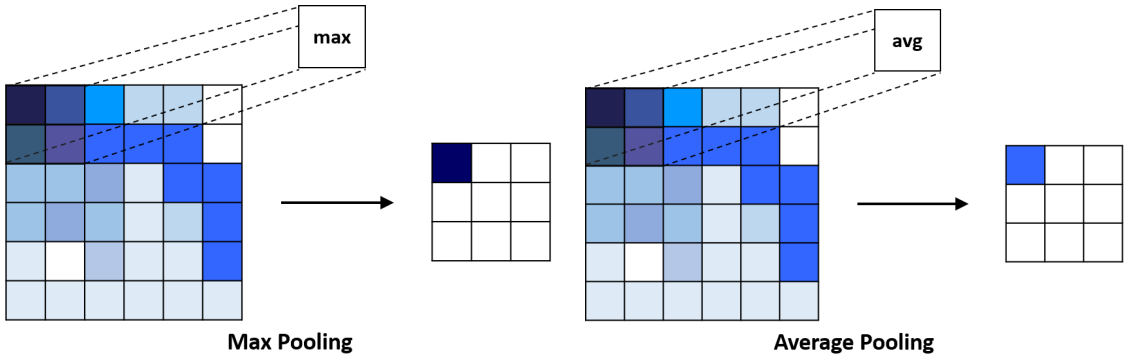

Figure 6. Pooling Layer Illustration. Adapted from [14].

d. Fully Connected Layer

After features extraction and consolidation by the CONV and POOL layers, respectively, fully connected layers are inserted at the end of the CNN to produce predictions for the given input. A non-linear function, such as the Softmax activation, is commonly used in a fully connected layer to generate predictions as output [17]. Figure 7 depicts an example of a feature map being flattened and fed into the fully connected layers to generate an inference output.

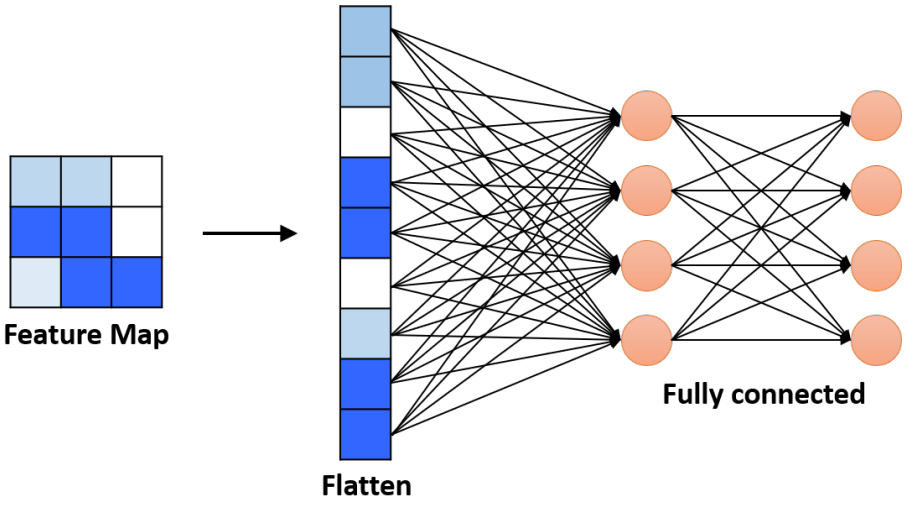

Figure 7. Fully Connected Layer Illustration. Adapted from [14].

3. Training Techniques

The advancement of deep learning networks, especially CNNs, has enabled their wide usage in multiple AI fields such as Computer Vision (CV), Natural Language Processing (NLP), and audio processing. At present, AI implementations have not fully proliferated in consumer-level applications and have limited capabilities when running on resource-constrained devices. Hence, reducing the gap between high-powered proprietary implementations and consumer-level applications is an increasingly active research topic [22].

a. Transfer Learning

Training a Deep Neural Network (DNN) from scratch is an extremely time-consuming and resource-intensive task. It is especially so for complex object recognition tasks. As increasingly different networks are trained for various tasks, Lorien Pratt [23] sought to avoid separate training and re-training of networks for similar purposes and instead built on previously trained network results. By reusing previously trained models, transfer learning can considerably speed up the learning process.

Wei Zhao [12] demonstrated training a CNN with the MNIST handwritten digital dataset, then transferred the learned model to train on a vehicle logo dataset by adding a fully connected layer and Softmax activation. The studied CNN recognized vehicle logos with higher efficiency and accuracy compared to directly training a CNN on the vehicle logo dataset.

b. Network Quantization

A way to reduce computational demands and increase power efficiency is through quantization. Quantization involves transforming an ML model into an approximated representation with available lower precision operations [24]. In deep learning, quantization generally refers to converting floating-point values to fixed point integers, for example, the numbers 0, 1, 2, …, 255 for pixel color or tone representation of a digital image.

In most cases, the primary source of latency when running a DNN is caused by the transfer of the network weights and data between the main memory and the processing cores. Reducing the data from 32-bit floating-point values to 16-bit floating-point values or 8-bit integers increases the efficiency of computing and hardware compatibility and reduces memory, power, and network bandwidth utilization.

Studies [25, 26, 27] have shown that the loss in accuracy is still manageable with variations in applying quantization during and post training. There are also variations of quantization types; each has its own set of pros and cons.

1. Reduced Float – reduction of 32-bit float to 16-bit float reduces complexity and generally produces negligible accuracy loss.
2. Hybrid – reduction of specific 32-bit float to 8-bit integer parameters, for example, 8-bit integer weights and 32-bit float biases and activation, achieving 10% to 50% faster execution on CNN models.
3. Integer – only contains 16-bit and 8-bit integers, enabling support for running on ML accelerators.

c. Network Pruning

Network Pruning is another optimization technique where redundant neurons and connections are removed, enabling a model to be compressed more efficiently. In one study [28], Hao Li et al. suggest that "filters with smallest weights tend to produce feature maps with weak activations compared to other filters in that layer." By pruning off the smallest filters instead of the same number of random or largest filters, it is possible to enable better optimization results [28].

Similar work in [29], which pruned filters with weights very close to zero and removed the feature maps completely from the layer, showed that a CNN model was still able to operate efficiently with minimal accuracy loss even when 76% of feature maps were removed. On the other hand, [29] also highlights that the percentage of channels a model can afford to lose before it starts losing significant accuracy varies from model to model but concluded that a significant portion of CNN parameters does not play an important

role. The concept of pruning is potentially very valuable in the application of ML on lightweight embedded devices.

4. Development Frameworks

In recent years, we have seen accelerated advancement in AI development and applications. Once a very specialized field of information technology, AI has become a much more manageable and widely applied technology due to the development of multiple libraries and frameworks. Development frameworks for AI come in various programming languages such as C, C++, C#, and Python. A group of frameworks by Google, TensorFlow (TF), TF Lite, and Keras utilizes the Python language and is one of the most mature for deployment in embedded applications. As this study concerns deployment of deep learning on lightweight edge devices, we review the Google frameworks in this section.

a. TensorFlow

TF is an open-source library widely used for large-scale ML. TF applications are built with Python, providing a convenient front-end application programming interface (API), while the backend processes execute in high-performance C++. TF can train and run a wide range of DNNs such as handwritten digit classification, image recognition, word embeddings, Recurrent Neural Networks (RNN), and NLP. TF 2.0, released in October 2019, incorporated the Keras API for model training and provided support for distributed training [30].

b. TensorFlow Lite

An extension to TF, TF Lite is an open-source deep learning framework for on-device inference. It enables the deployment of models on various platforms such as mobile devices running Android and IOS and micro computing devices like Raspberry Pi.

Figure 8 shows the TF Lite process that would enable DNN models to be converted, compressed, or optimized by quantization and executed on lightweight devices such as mobile and embedded devices.

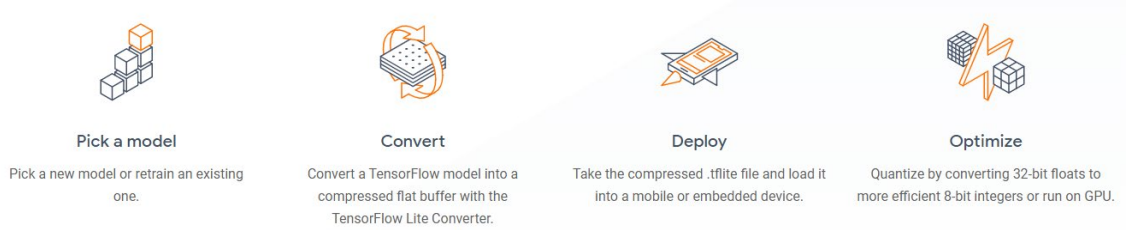

Figure 8. How TensorFlow Lite Works. Source: [6].

c. Keras

Keras is a wrapper to the TF framework, designed specifically for easy deployment of DNNs. Keras allows for easy and fast prototyping as well as seamless performance on CPU and GPU. Furthermore, DNN models developed on Keras can be converted to TF Lite to be optimized for deployment on lightweight edge devices.

C. DEEP LEARNING ON EDGE DEVICES

Applying AI on edge devices is an emerging paradigm that combines AI, Internet of Things (IoT), and Edge Computing technologies. As the name implies, it pushes "computing tasks and services from the network core to the network edge" [31]. With the advancement of robust and low power consumption IoT devices, advanced ML models can now be executed on edge devices such as robots and video cameras. By processing data on the edges, less data will be transmitted, hence reducing network communication overhead.

In this research, we explore running deep learning models for vehicle recognition on an edge device, the Raspberry Pi (RPi). It is a low-cost, versatile microcomputer used mainly for educational purposes and projects where features like portability and low power consumption are desired. In general, the RPi is incapable of training large datasets or complex ML models, and further optimizations on trained models are required to enable inference or prediction tasks on it. At the time of this study, the latest version features a 64-bit Quad-core ARM processor and up to 8 GB of RAM, making it adequately capable of running optimized deep learning models for vehicle recognition.

D. CHAPTER SUMMARY

Chapter II has discussed the background study of components required to train and optimize neural network models for inference on small, low-power edge devices. The chapter also covered related research on applying deep learning for vehicle recognition and examined concepts in deep learning and training techniques to optimize the models for execution on the edge devices.

Next, Chapter III describes implementing a classifier architecture and sharing the parameters required to fulfill our research objective.

III. METHODOLOGY

This chapter discusses the design and implementation of an ML system using various training and optimization techniques. It describes the dataset, data pre-processing methods, classifier architecture, and techniques applied to optimize inference performance.

A. SYSTEM ARCHITECTURE

This system aims to train, optimize, and identify the best performing model for vehicle recognition from a selected set of pre-trained Keras Applications models [32] and TF Lite models [33]. Inference tests are conducted on a lightweight edge device, the RPi 4, and results are evaluated in this study. The pre-trained models are re-trained on the Stanford Cars dataset [34] with transfer learning then optimized by various model optimization techniques. The eventual goal is to measure and compare the model size reduction and on-device inference performance (accuracy and latency) of the optimized models. All training and optimization of the models is conducted using the NVIDIA Tesla P100 and V100 GPUs on the Google Colaboratory (Colab) platform. All inference performance tests are conducted on a Raspberry Pi Model 4B, 4 GB, running ARM architecture Linux-based 32-bit Raspberry Pi OS.

Figure 9 depicts key components of the system architecture and flow. Starting with the classifier architecture, it consists of input data pre-processing, model training operations, and optimization. Only one dataset is used for training and test inputs, whereas multiple pre-trained models are fitted in the classifier, and variants of optimized models for each pre-trained model are produced. The original model size and its compressed size are recorded. Additionally, inference accuracy and latency of tests conducted on the RPi are also recorded. They are subsequently evaluated to identify the best model to perform vehicle recognition on the RPi 4.

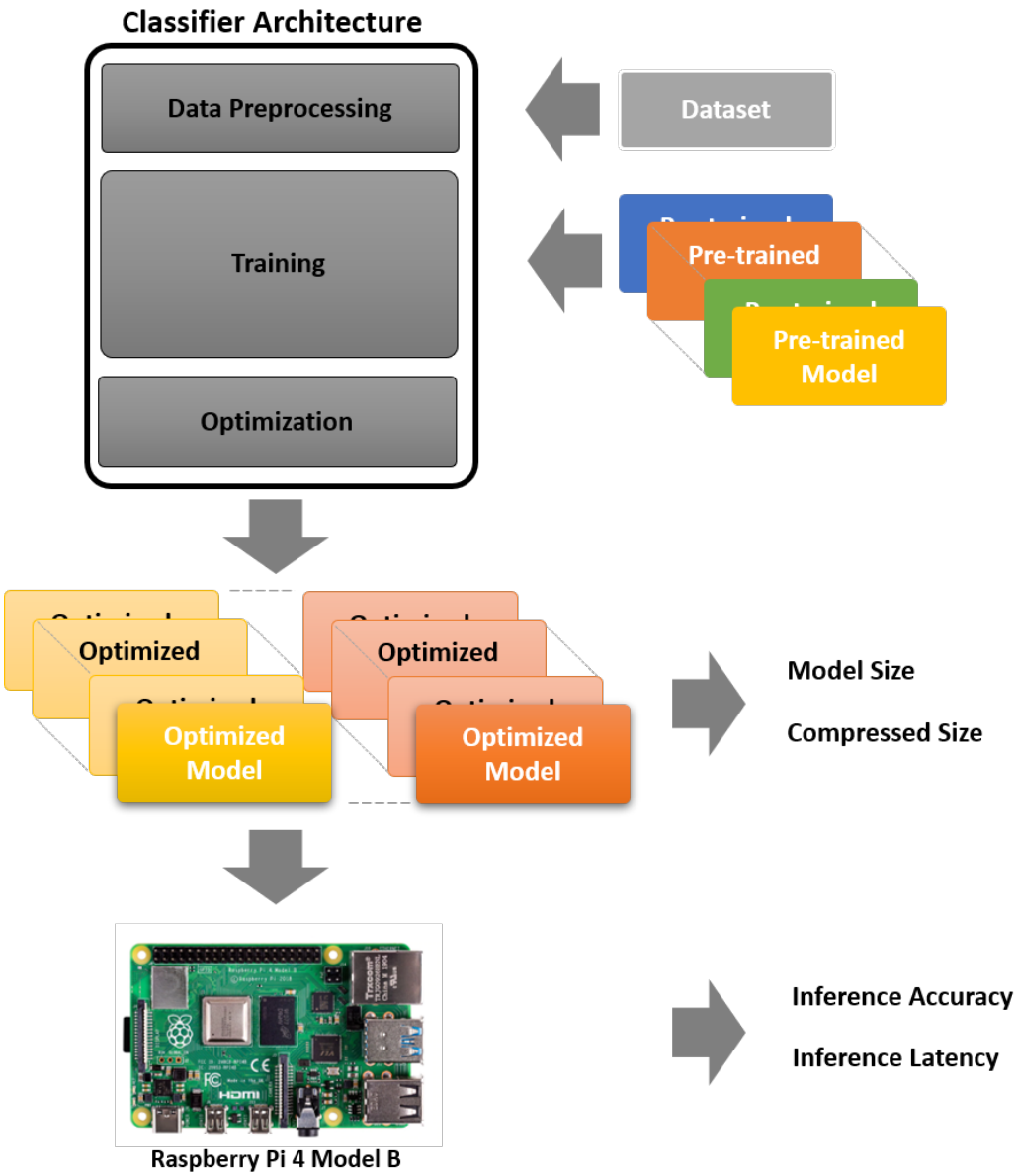

Figure 9. System Architecture.

The dataset encompasses a training set and a test set. For the entire setup, the training set is further split into a training set and a validation set with a proportion of 80% and 20%, respectively, while the test set is used for testing all trained and optimized models.

The pre-trained models selected for the tests are Keras models, MobileNet, MobileNetV2, NASNetMobile, DenseNet121, and DenseNet169, and TF Lite models,

EfficientNetLite0 through 4. The same configuration of top layers is added to all Keras models, and default top layers provided by TF Lite Model Maker are applied for all TF Lite models. Re-training the pre-trained models involves loading weights trained on ImageNet as initial weights and training the full model (the pre-trained and top layers) on a vehicles dataset. Models are trained using the Adam optimizer with a learning rate of 0.0001. As the TF Lite models do not converge well with the learning rate of 0.0001, a rate of 0.001 is used instead.

Training times were a maximum of 200 epochs with early stopping imposed for 20 epochs if there was no improvement in validation accuracy. The early stopping strategy is employed in [35] as regularization for overfitting. On every improvement of validation accuracy in training, a model checkpoint is saved. Doing so ensures that the best accuracy model is saved and used in subsequent transfer learning and optimization steps.

Pruning is applied to the Keras models, and the resultant model size and its compressed size are recorded. TF Lite models, particularly the EfficientNetLite models, do not support pruning at the time of this study. Therefore, they do not go through the pruning process. Subsequently, the transfer learned models are quantized using various techniques mentioned in section D.2. The resultant model sizes are recorded and tested for inference performance on the RPi. Finally, the results are tabulated and analyzed in the next chapter.

Evaluation criteria to find the best performing model for the task of vehicle recognition by make, model, and year, include model accuracy, model size (compressed and uncompressed), and inference latency when run on an RPi 4.

B. DATASET

The Stanford Cars dataset [34] was used for the entirety of this study. It consists of 16,185 images labeled with 196 classes of vehicles by make, model, and year. The dataset has a proportion of 8,144 images for training and 8,041 images for testing. It is one of the more comprehensive publicly available datasets and is widely used as a benchmark dataset for ML research on vehicle recognition.

As shown in Figure 10, images are of various sizes, and each contains a vehicle in the foreground taken from various angles against a random background. The quality of images also varies from professionally taken shots to low-quality screenshots taken off online advertisements.

The distribution of images per vehicle class in the dataset has an average of 83 images per class, with a minimum of 48 images and a maximum of 138 images per class [36]. Initial trials on training the models did not yield good accuracy performance. The models tended to overfit, which can be attributed to the small number of images per class and overall small dataset. Thus, data pre-processing is introduced to counter the effects of the lack of training samples and overfitting.

Figure 10. Example Images from Stanford Cars Dataset. Source: [34].

a. Pre-processing

The data pre-processing techniques used in this implementation serve two purposes. One is to ensure uniformity of image size and to normalize the input scale received by the model. Another is to use data augmentation and create more variation in images for each class of vehicles, effectively increasing the diversity of the dataset.

(1) Rescale and Resize

All Keras models selected in this study accept an input image size of 224 x 224 pixels with three channels of RGB colors per pixel. Thus, all input images must be resized to 224 x 224 pixels and have the RGB values normalized by dividing each channel by a factor of 255.0.

For the TF Lite models, each variant of the EfficientNetLite models accepts different fixed test image sizes for inference. From EfficientLite0 to EfficientLite4, the models accept 224 x 224, 240 x 240, 260 x 260, 280 x 280, and 300 x 300 pixels, respectively. Test images have to be resized accordingly, while training images are resized by default when data is input into TF Lite Model Maker.

(2) Data Augmentation

For Keras models, a variety of data augmentation techniques have been used to increase the diversity of the training set. In this study, random but realistic transformations were applied by building custom pipelines with TF's "tf.image" library. The augmentation methods used in this experiment include random horizontal flip, resize by crop or pad, random crop, random contrast, brightness, saturation, and hue.

Figure 11 shows examples of vehicle images with their RGB channels rescaled, image size resized to 224 x 224 pixels, and augmentation applied. Augmentation is typically applied only on the training set and not on the validation and test sets.

As for TF Lite models, the default data augmentation option is applied for training.

Figure 11. Example Images with Data Augmentation. Adapted from [34].

C. CLASSIFIER ARCHITECTURE

Building the classifier for transfer learning consists of three components: designing the classifier, selecting pre-trained models for our experiment, and adding appropriate top layers with suitable parameters.

1. Design

The classifier is built for three objectives. First, it pre-processes the input data. Next, it trains and produces machine-learned models for vehicle recognition with hyperparameters tuning to speed up and improve the training process and accuracy. Lastly, it applies optimization techniques to reduce the model size and improve inference speed for execution on the RPi.

A selected set of pre-trained models designed for resource-constrained devices is evaluated in this study, and these models are discussed in the next section.

Figure 12 depicts the components of the Keras Classifier Architecture. Pruning is applied to all fully connected layers of the top layers only, and the stages, namely Pre-trained Model and Quantization, vary with the options listed in the figure.

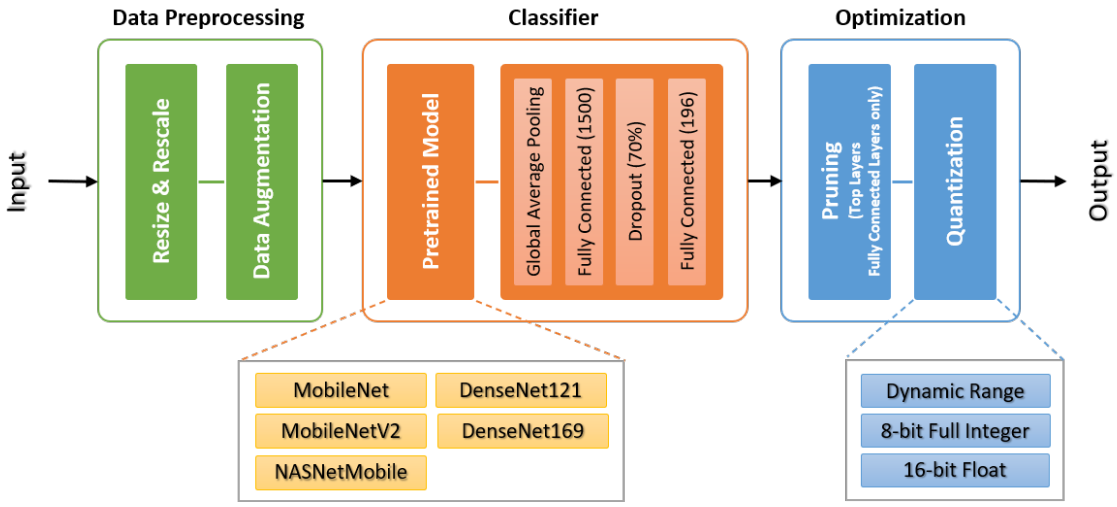

Figure 12. Keras Classifier Architecture.

Figure 13 depicts the components of the TF Lite Classifier Architecture available with the TF Lite Model Maker. Data preprocessing and top layers are handled by default by the classifier. Like the Keras Classifier, the Pre-trained Model and Quantization stages vary with the options listed in the figure.

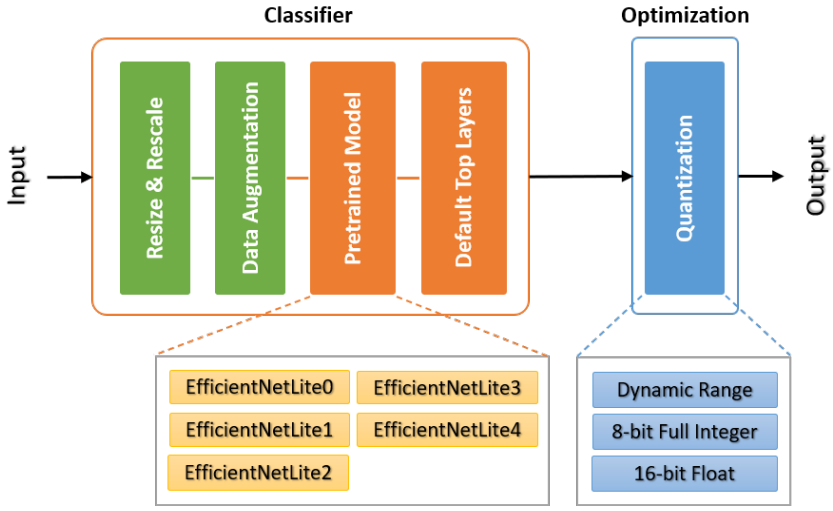

Figure 13. TF Lite Classifier Architecture.

a. Improving Classification

Figure 14 shows an experimental Keras classifier architecture built with the MobileNetV2 model as a base model. The top layers were stacked with a Global Average Pooling layer and two fully connected (dense) layers of a dimensionality of 1,500 and 196. A Dropout layer of 50% separated the dense layers to prevent overfitting. Weights trained on the ImageNet dataset were preloaded to the network, and the Adam optimizer with a learning rate of 0.0001 was applied for training. The base model was frozen, with only the top layers trained.

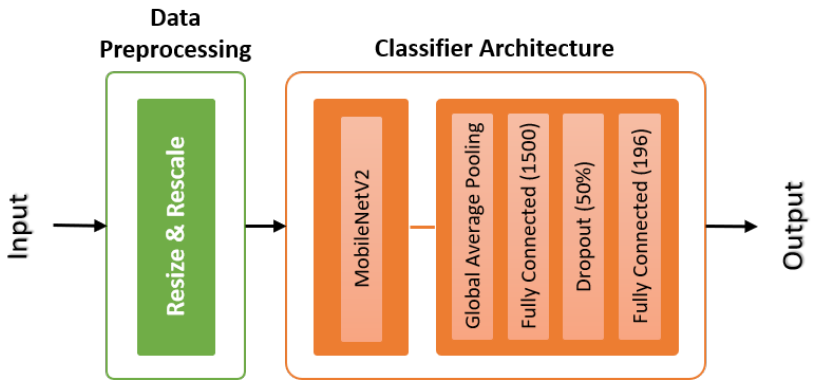

Figure 14. Experimental Classifier.

After more than 70 epochs of training, the model only managed to attain an accuracy of approximately 41% on both the validation and test sets. It was hypothesized that the poor accuracy was caused by a variety of factors, such as:

1. Small dataset size for each class. The size of the Stanford Cars dataset was relatively small for each of the 196 classes of vehicles.
2. Low variation between designs. Vehicles of the same model usually have minor variations between them, making them hard to classify.
3. Weights not optimized for vehicle classification. The proportion of vehicle image classes in ImageNet is unbalanced compared to other classes. There are 317K images of vehicles versus 1,567K images of animals (mammals and birds) and 414K images of objects [37].

The factors identified mainly point to issues with the dataset image diversity and insufficiently trained weights on vehicle images. The issues were eventually resolved through data augmentation and re-training the entire model instead of only the top layers.

2. Pre-trained Model Selection

Keras API [32] provides a list of deep learning models with pre-trained weights that can be used for prediction, feature extraction, and fine-tuning in transfer learning. Extracted from the Keras Applications, Table 1 lists the models trained on the ImageNet dataset.

Table 1. Available Keras Application Models. Source: [32].

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-
EfficientNetB0	29 MB	-	-	5,330,571	-
EfficientNetB1	31 MB	-	-	7,856,239	-
EfficientNetB2	36 MB	-	-	9,177,569	-
EfficientNetB3	48 MB	-	-	12,320,535	-
EfficientNetB4	75 MB	-	-	19,466,823	-
EfficientNetB5	118 MB	-	-	30,562,527	-
EfficientNetB6	166 MB	-	-	43,265,143	-
EfficientNetB7	256 MB	-	-	66,658,687	-

Since this study concerns finding a best performing vehicle recognition model to run on the RPi, we base the model selection criteria on the size and number of parameters. The size affects the transmission of trained models across devices, while the number of parameters reflects the complexity and, in turn, affects inference speed. Using the attributes of models trained on ImageNet as a reference, we limit the model size to less than 100 MB and 20 million parameters. The Keras models are highlighted in blue in Table 1, while models highlighted in green have their corresponding TF Lite models evaluated in this study.

3. Top Layers

A usual approach for transfer learning is to add a pooling layer followed by fully connected layers on top of the base model. The final output layer should have an output size equivalent to the output classes; for example, 196 classes of vehicles will require 196 output nodes. The fully connected layers should provide sufficient learnable space for the new model.

Figure 15 shows the eventual top layers added to the base (pre-trained) model. A Global Average Pool layer is added first, followed by a Fully Connected layer of size 1,500 for a sizeable learning space. A Dropout layer of 70% is added to help with overfitting, and finally, an output layer of size 196 to provide inference classification for 196 classes of vehicles. "SoftMax" is applied as the activation function for the output layer.

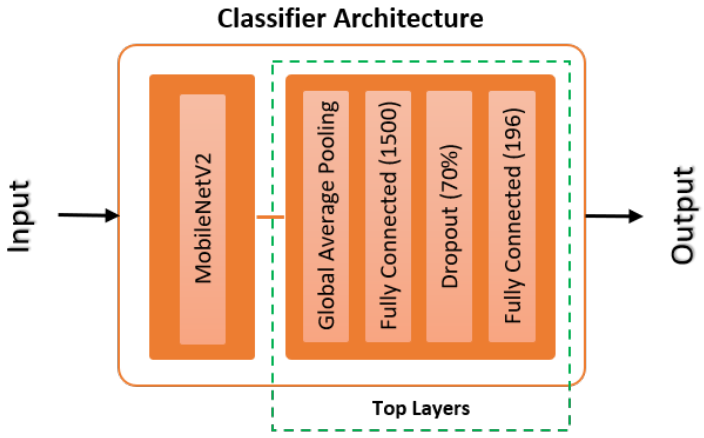

Figure 15. Classifier Top Layers.

An initial training test was done with the base model loaded with pre-trained weights from ImageNet, and the classifier was trained only on its top layers. The accuracy achieved was an average of 41%, which is still far from reaching the approximately 71% accuracy when MobileNetV2 was trained on ImageNet. Figure 16 depicts the initial training versus validation accuracy. Validation accuracy remained at approximately 41% while the training accuracy increased to 90%, a sign of overfitting.

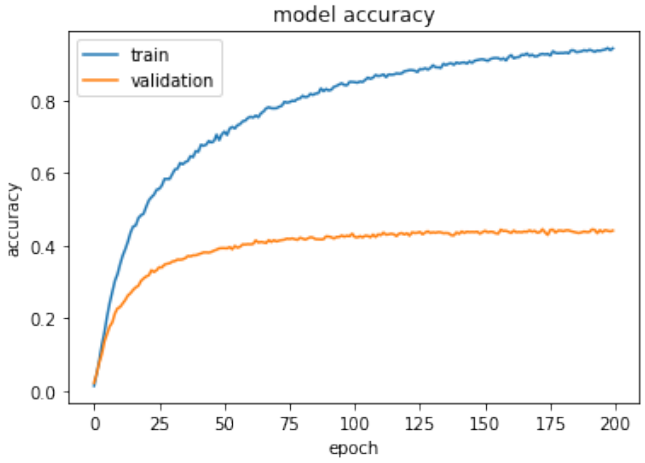

Figure 16. Initial MobileNetV2 Training and Validation Accuracy.

To improve the accuracy, a significant change was to train the entire classifier on the dataset instead of only training the top layers. Since only the inference performance on the RPi is evaluated, there was no concern about training time as it is done off-device for this study. Data augmentation also helped in improving the validation accuracy of the model. Figure 17 shows the improved validation accuracy of approximately 75%.

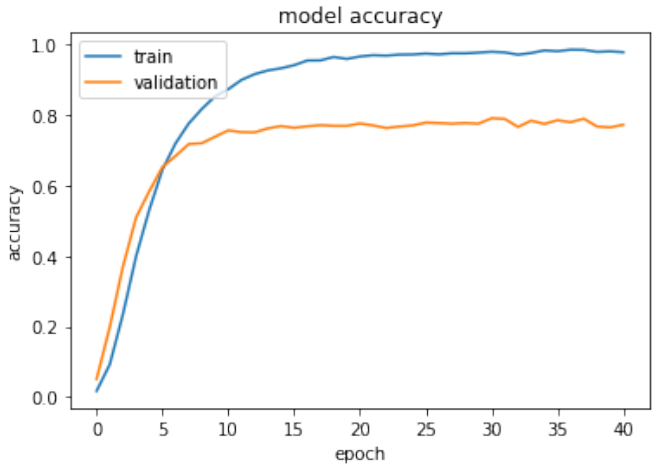

Figure 17. Improved MobileNetV2 Training and Validation Accuracy.

D. MODEL OPTIMIZATION FOR PERFORMANCE

In this study, we look at optimization techniques to increase inference efficiency as it is a concern when conducting inference on resource-constrained devices. Particularly on devices like the RPi, model size and computation efficiency are significant concerns when factoring for latency, memory utilization, and power consumption.

By combining pruning and quantization techniques in the TensorFlow Model Optimization Toolkit (TFMOT), this study intends to minimize the complexity of ML models and reduce their size while maintaining inference accuracy.

1. Pruning

Pruning essentially removes parameters from a model, without having a critical impact on its predictions. This technique is used to gradually zero out parameters during the training process to achieve model sparsity [38]. When sparse, a model can be easily compressed, and the zeroed-out parameters can be skipped to improve inference latency. Enabling effective compression makes pruning a helpful technique in reducing model download size and shortens transmission time. On the other hand, it is noted in [39] that pruned models generally remain the same size on disk and have a similar latency at runtime as unpruned models.

It is also noted that pruning a model may negatively affect accuracy [40]. As such, in this study, pruning is done post-training and restricted to only the fully connected (dense) layers in the top layers to minimize disruption to the critical layers of the pre-trained models. Pruning is applied by further training the Keras models with the TFMOT sparsity library, having the model's low magnitude parameters pruned off. Pruning is not applied to TF Lite EfficientNetLite models as it is not supported at the time of this study.

2. Quantization

Two forms of quantization are available in the TFMOT, post-training quantization and quantization-aware training. Post-training quantization is more straightforward to implement, but quantization-aware training often offers better model accuracy. Quantization-aware training is only available to a limited set of Keras models [41]; thus, this study explores post-training quantization only.

Post-training quantization converts a trained model's weights and activation outputs from 32-bit floating-point numbers to either 16-bit floating-point or 8-bit integers, depending on requirements. The result is a smaller model and increased inference speed, critical to resource-constrained devices like the RPi and required by integer-only accelerators such as the Edge Tensor Processing Unit (TPU). In general, 16-bit floats are recommended for inference on GPU acceleration and the 8-bit integer for CPU executions.

Three post-training quantization techniques available for TF Lite models are explored in this study.

a. Dynamic Range Quantization

Dynamic quantization involves converting only model weights from floating-point to 8-bit integer precision. Depending on the inference operation, activations (outputs of intermediate layers) can be dynamically quantized to 8-bit, and computations are performed with 8-bit weights and activations. Although doing so can achieve near fixed-point inference latencies, the outputs will remain in floating point precision; thus, latencies will still be higher than fixed-point integer computation [42]. In this study, dynamic quantization is achieved with the "Optimize" flag on the TF Lite converter library.

b. 16-bit Float Quantization

16-bit float quantization converts all weights from 32-bit to 16-bit floating-point precision. This technique effectively reduces accuracy loss and results in two times reduction in model size since all weights are reduced to half of their original size. However, latency improvement can only be achieved when inference operations are done on GPUs that natively operate on 16-bit float data. The weights are dynamically "dequantized" to 32-bit floats when operating on CPUs [42]. Nonetheless, this study evaluates the accuracy and size reduction properties of this quantization scheme.

c. 8-bit Full Integer Quantization

Full integer quantization involves converting "32-bit floating-point numbers (such as weights and activation outputs) to the nearest 8-bit fixed-point numbers" [43]. The range of floating-point parameters in a model, however, needs to be calibrated or known before a conversion can be done. Model weights and biases are constant parameters, but variable parameters like model inputs, activations, and model output cannot be calibrated unless the input data is put through a few inference cycles. TF Lite converter allows for such calibration using a representative dataset of 100 to 500 samples [42]. In order to study the latency improvements, full integer quantization is enforced for all parameters, including the input and output data types in this study.

E. TRAINING AND TESTING SETUP

The setup leverages Keras (for Keras Applications models) and TF Lite Model Maker (for TF Lite models) frameworks to optimize selected pre-trained models on the Stanford Cars dataset [34]. All training and optimizations of the models are done on the Google Colaboratory (Colab) platform with GPU, and High RAM settings turned on.

Parameters used for optimizing the performance of the Keras models are given in Table 2:

Table 2. Keras Models Training Parameters

Dataset	
Stanford Cars training set	80% training. 20 % validation
Data Augmentation	
Resizing and Rescaling ***(applied to entire dataset)***	Cast image values to 32-bit float, Rescale RGB by 255, Resize images to 224x224 pixels
Image Augmentation ***(applied only on training split)***	Random flip left and right, Resize by crop or pad, Random cropping, Random contrast, Random brightness, Random saturation, Random hue
Model Training	
Base Models	MobileNet, MobileNetV2, NASNetMobile, DenseNet121, DenseNet169
Optimizer	Adam
Learning rate	0.0001
Training epoch	200
Early stopping patience	10
Pre-train weights	ImageNet
Top Layers	Global Average Pooling Dense (1500, "ReLU" activation") Dropout (0.7) Dense (196, "SoftMax" activation)
Training	Entire model (Base + Top layers)
Pruning	
Learning rate	0.0001
Training epoch	30
Sparsity function	Polynomial Decay (Initial sparsity 0.5, Final sparsity 0.8)
Layers trained	Only Dense layers
Quantization	
Dynamic	Optimizations - tf.lite.Optimize.DEFAULT
Float16	Optimizations - tf.lite.Optimize.DEFAULT Target spec – tf.float16
Integer8	Optimizations - tf.lite.Optimize.DEFAULT Representative data – 400 images from training set Target spec - tf.lite.OpsSet.TFLITE_BUILTINS_INT8
Output	
Type	TF Lite model

Parameters used for optimizing the performance of the TF Lite models on the TF Lite Model Maker are given in Table 3:

Table 3. TF Lite Models Training Parameters

Dataset	
Stanford Cars training set	80% training. 20 % validation
Model Training	
Base Models	EfficientNetLite0, EfficientNetLite1, EfficientNetLite2, EfficientNetLite3, EfficientNetLite4
Data Augmentation	True
Data shuffle	True
Learning rate	0.001
Training epoch	200
Dropout Rate	0.7
Layers trained	Entire model
Pruning (not supported)	
Quantization	
Dynamic	Optimizations – tf.lite.Optimize.DEFAULT
Float16	Optimizations – tf.lite.Optimize.DEFAULT
Integer8	Optimizations – tf.lite.Optimize.DEFAULT Representative data – training set
Output	
Type	TF Lite model

All generated TF Lite models are tested on a Raspberry Pi 4 Model B, 4 GB computer. It runs an ARM architecture Linux-based 32-bit Raspberry Pi OS with Python 3.7 and TF 2.4.0-rc2 library and dependencies.

Inference tests are coded in Python, and the test parameters are given in Table 4:

Table 4. Models Test Parameters

Dataset	
Stanford Cars testing set	100% testing
Data Augmentation	
Rescaling	Cast image values to 32-bit float, Rescale RGB by 255
Input image size	• 224 x 224 pixels for MobileNet, MobileNetV2, NASNetMobile, DenseNet121, DenseNet169, EfficientNetLite0 • 240 x 240 pixels for EfficientNetLite1 • 260 x 260 pixels for EfficientNetLite2 • 280 x 280 pixels for EfficientNetLite3 • 300 x 300 pixels for EfficientNetLite4
Test Readings	
Inference latency	Average of total inference process time / total images in testing set
Accuracy	Correct classification / total classifications made
Model file size	Model file sizes and compressed file sizes both collected upon generation from training

F. CHAPTER SUMMARY

This chapter discussed the entire process of implementing a classifier architecture to train, optimize, and test different state-of-the-art Keras Applications and TF Lite models for vehicle recognition. It also covered the necessary steps and optimization required to improve overall model performance.

Chapter IV analyzes the results obtained through training and testing the generated models with the different optimization techniques applied.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. RESULTS AND ANALYSIS

This chapter presents the results and analysis of tests conducted on the selected Keras and TF Lite models trained on the Stanford Cars dataset [34] using the implemented system architecture described in the previous chapter. It provides an overview of primary findings for this study and the models' performance analysis and comparisons.

A. OVERVIEW

As noted earlier, TF model optimizations enable a model to run more efficiently and be deployed on edge devices such as the Raspberry Pi. This study demonstrates the effects of optimization on the selected models and attempts to find a model best suited to run vehicle recognition on the RPi, based on the evaluation criteria of compressed model size, inference accuracy, and latency.

The primary findings in this study include the following:

1. When applied to the models, both pruning and quantization reduce the model size by more than a factor of 2. At the same time, accuracy is maintained within the range of plus or minus 2.5% from the baseline models' accuracy.
2. Pruning only the fully connected (dense) layers of the added top layers drastically reduces the model sizes by up to 4.57 times and improves accuracy slightly across all models.
3. Selected quantization techniques, when applied to the models, performed as intended. Dynamic and Integer 8-bit quantization methods reduce weights and activations from 32 bits to 8 bits, reducing the model sizes by up to 4.39 times. Float 16-bit quantization reduces model weights and activations from 32 bits to 16 bits, reducing the model sizes by up to 2 times.

4. Through prioritization of the collected metrics, MobileNet pruned and quantized to full 8-bit integer is identified as the model best suited for vehicle recognition on the RPi.

B. MODEL PERFORMANCE

This section presents the results of testing the CNN models transfer learned on the Stanford Cars dataset [34], on an RPi. Readings gathered from the tests include Top-1 accuracy, model original size, compressed model size, and inference latency.

1. Effects of Optimization

Optimization techniques, pruning, and quantization, were applied post-training on the models. It should be noted, however, pruning was not supported for the EfficientNetLite models on TF Lite Model Maker at the time of this study. Thus, it was only applied to the Keras Application models.

a. Evaluating Pruned Models

The Keras Applications models MobileNet, MobileNetV2, NASNetMobile, DenseNet121, and DenseNet169 were selected as they meet the selection criteria of within 100 MB and 20 million parameters when trained on the ImageNet dataset.

Table 5 tabulates each model's accuracy, original size, compressed size, and factor of size reduction on its compressed base Keras model to its compressed pruned TF Lite model. All Keras models were trained and evaluated for accuracy on the Google Colaboratory (Colab) platform. In contrast, the TF Lite models were converted from the Keras models on Colab, then evaluated for accuracy on the Raspberry Pi. When considering model transmission between server and edge devices or between edge devices, transferring compressed versions of the model is considered a more efficient use of network bandwidth. Thus, the compressed model sizes were used for comparison in this analysis.

The tabulated compressed sizes demonstrated a significant size reduction ranging from 3.18x to 4.57x after pruning and converting to TF Lite. We also noted that a conversion of the pruned models from Keras to TF Lite does not affect the accuracy of the

models when tested on separate platforms. Moreover, there is further model size reduction across all models in the range of 0.2 to 1.4 MB after the model conversion from Keras to TF Lite.

The accuracy of the pruned models is also observed to improve with an increase in the range of 0.18% to 2.32%. As we learned from [37], if done incorrectly, pruning might negatively affect model accuracy. We found that pruning only the fully connected (dense) layers of the top layers can drastically reduce model size and improve accuracy across all models. Based on the findings, model pruning of dense layers should always be considered for deployment efficiency.

Table 5. Pruning on Keras Models

	Generated Model	*Top-1 Accuracy*	*Size (MB)*	*Compressed Size (MB)*	*Size Reduction*
MobileNet	*Base (Keras)*	81.21%	58.10	**52.86**	**3.97x**
	Pruned (Keras)	82.19%	19.46	13.50	
	Pruned (TF Lite)	82.19%	19.20	**13.31**	
MobileNetV2	*Base (Keras)*	77.34%	51.60	**46.78**	**4.57x**
	Pruned (Keras)	77.52%	17.35	10.49	
	Pruned (TF Lite)	77.52%	16.91	**10.24**	
NASNetMobile	*Base (Keras)*	72.89%	72.97	**64.77**	**3.80x**
	Pruned (Keras)	73.62%	24.67	17.30	
	Pruned (TF Lite)	73.62%	23.47	**17.06**	
DenseNet121	*Base (Keras)*	83.56%	102.38	**89.51**	**3.34x**
	Pruned (Keras)	85.36%	34.49	27.14	
	Pruned (TF Lite)	85.36%	33.59	**26.80**	
DenseNet169	*Base (Keras)*	83.19%	177.53	**151.95**	**3.18x**
	Pruned (Keras)	85.51%	59.78	48.30	
	Pruned (TF Lite)	85.51%	58.38	**47.75**	

Size reduction is derived from the size comparison between the respective networks Base (Keras) model and Pruned (TF Lite) model.

b. Evaluating Quantized Models

With pruning shown to reduce the model size drastically, we now evaluate the effects of model quantization.

First, we look at the effects of quantization on the size of the models. Table 6 tabulates the test results recorded for three types of quantization done on the Keras models

that were pruned and converted to TF Lite. All optimized TF Lite models were ported over and tested on the RPi. Across all quantized models, further size reductions of between 1.38x to 4.39x are observed. The size reductions are significant, considering there was already a 3.18x to 4.57x reduction from the original Keras models to the pruned TF Lite models.

Dynamic quantization offers a good model size reduction of 2.63x to 4.25x. Top-1 accuracy of the models generally held up well, with only minor drops of 0.17%, 0.25%, 0.27%, and 0.01% on the MobileNet, MobileNetV2, NASNetMobile, and DenseNet121 models, respectively. And there is a negligible accuracy increase of 0.03% on the DenseNet169 model.

Float 16-bit quantization is observed to produce a moderate size reduction of 1.38x to 2.00x, yet it maintains the models' accuracy very well, with a slight drop of 0.2% on only the MobileNet and NASNetMobile models. The size reduction demonstrated the expected behavior of cutting weights and activations of 32 bits to 16 bits, effectively reducing the model size by approximately half. The model size is not expected to reduce by an absolute half as it is dependent on a model's structure and composition.

Lastly, Integer 8-bit quantization offered size reductions similar to those from dynamic quantization. Compressed model size reductions range between 2.56x and 4.39x from the pruned TF Lite models. Similarly, model size is not expected to reduce by an absolute quarter as it largely depends on a model's structure and composition. It is also observed to have a slightly larger negative impact on accuracy compared to dynamic quantization. The accuracy drops are 0.31%, 0.48%, 0.65%, 2.39%, and 0.41% for MobileNet, MobileNetV2, NASNetMobile, DenseNet121, and DenseNet169 models, respectively.

The tests demonstrated that model quantization can further reduce model size while maintaining accuracy, with a majority of the drops at less than 1%.

Table 6. Quantized Keras Applications Models (Compressed Size)

	Optimized Model	*Top-1 Accuracy*	*Size (MB)*	*Compressed Size (MB)*	*Size Reduction*
MobileNet	*Base (Keras)*	81.21%	58.10	52.86	-
	Pruned (TF Lite)	82.19%	19.20	**13.31**	-
	Dynamic Quan.	82.02%	5.06	**5.06**	**2.63x**
	Float 16bit Quan.	82.17%	9.62	**9.62**	**1.38x**
	Int 8bit Quan.	81.88%	5.19	**5.19**	**2.56x**
MobileNetV2	*Base (Keras)*	77.34%	51.60	46.78	-
	Pruned (TF Lite)	77.52%	16.91	10.24	-
	Dynamic Quan.	77.27%	4.64	**2.88**	**3.57x**
	Float 16bit Quan.	77.52%	8.49	**5.34**	**1.92x**
	Int 8bit Quan.	77.04%	4.84	**2.92**	**3.51x**
NASNetMobile	*Base (Keras)*	72.89%	72.97	64.77	-
	Pruned (TF Lite)	73.05%	23.47	**17.06**	-
	Dynamic Quan.	73.13%	6.81	**4.80**	**3.55x**
	Float 16bit Quan.	73.03%	11.94	**8.74**	**1.95x**
	Int 8bit Quan.	72.40%	7.01	**4.75**	**3.59x**
DenseNet121	*Base (Keras)*	83.56%	102.38	89.51	-
	Pruned (TF Lite)	85.36%	33.59	**26.80**	-
	Dynamic Quan.	85.35%	8.90	**6.43**	**4.17x**
	Float 16bit Quan.	85.36%	16.89	**13.48**	**1.99x**
	Int 8bit Quan.	82.97%	8.84	**6.27**	**4.27x**
DenseNet169	*Base (Keras)*	83.19%	177.53	151.95	-
	Pruned (TF Lite)	85.51%	58.38	**47.75**	-
	Dynamic Quan.	85.54%	15.42	**11.24**	**4.25x**
	Float 16bit Quan.	85.51%	29.32	**23.90**	**2.00x**
	Int 8bit Quan.	85.10%	15.20	**10.90**	**4.39x**

Size reduction is derived from the size comparison between the respective networks' individual quantized models and their Pruned (TF Lite) model. The Base (Keras) model results are inserted for reference only.

Table 7 tabulates the test results recorded for the three types of quantization done on the EfficientNetLite TF Lite models. Since pruning is not supported on TF Lite models, comparisons were made against the base EfficientNetLite TF Lite models re-trained on the Stanford Cars dataset [34]. All trained and quantized models were tested on an RPi.

The quantized EfficientNetLite models' compressed size reductions range from 1.92x to 4.38x, which has a similar upper bound but better lower bound than the Keras models.

Dynamic quantization generated models with a size reduction range of 3.55x to 4.25x demonstrated a better lower bound and similar upper bound when compared against

the dynamic quantization on the Keras models. With slight drops in accuracy of 0.23%, 0.05%, 0.01%, and 0.21% for EfficientNetLite0, 1, 2, and 3 respectively, and a minuscule increase in accuracy of 0.01% for EfficientNetLite4, we observe minimal impact on the models' Top-1 accuracy.

Float 16-bit quantization produced models with a size reduction of 1.92x to 2.00x, which is again in line with the reduction of weights and activations from 32 bits to 16 bits, effectively cutting the model size by approximately half. Interestingly, besides EfficientNetLite2, which has an accuracy drop of 0.01%, the rest of the EfficientNetLite models have slight improvements in accuracy. The accuracy improvements are 0.01%, 0.03%, 0.03%, and 0.01% for EfficientNetLite0, 1, 3, and 4, respectively.

Similar to Dynamic quantization, Int 8-bit quantization on the EfficientNetLite models produced a size reduction range of 3.51x to 4.38x. It performed better than Integer 8-bit quantization on the Keras models for both lower and upper bounds. Accuracy, however, experienced a slight drop across all the models except the EfficientNetLite3. Top-1 accuracy drops are 0.15%, 0.16%, 0.14% and 0.16% for EfficientNetLite0, 1, 2 and 4, respectively. The EfficentNetLite3 model has its accuracy improved by 0.08%.

Summarizing the results, quantization on the EfficientNetLite models produced models with size reduction very close to their respective intended specifications, and accuracy is maintained well, with drops of less than 0.23%.

Table 7. Quantized TF Lite Models (Compressed Size)

	Optimized Model	*Top-1 Accuracy*	*Size (MB)*	*Compressed Size (MB)*	*Size Reduction*
EfficientNetLite0	*Base (TF Lite)*	77.83%	19.20	**19.20**	-
	Dynamic Quan.	77.60%	5.06	**5.06**	**3.79x**
	Float 16bit Quan.	77.84%	9.62	**9.62**	**2.00x**
	Int 8bit Quan.	77.68%	5.19	**5.19**	**3.70x**
EfficientNetLite1	*Base (TF Lite)*	79.59%	16.91	**10.24**	-
	Dynamic Quan.	79.54%	4.64	**2.88**	**3.56x**
	Float 16bit Quan.	79.62%	8.49	**5.34**	**1.92x**
	Int 8bit Quan.	79.43%	4.84	**2.92**	**3.51x**
EfficientNetLite2	*Base (TF Lite)*	80.09%	23.47	**17.06**	-
	Dynamic Quan.	80.08%	6.81	**4.80**	**3.55x**
	Float 16bit Quan.	80.08%	11.94	**8.74**	**1.95x**
	Int 8bit Quan.	79.95%	7.01	**4.75**	**3.59x**
EfficientNetLite3	*Base (TF Lite)*	81.02%	33.59	**26.80**	-
	Dynamic Quan.	80.81%	8.90	**6.43**	**4.17x**
	Float 16bit Quan.	81.05%	16.89	**13.48**	**1.99x**
	Int 8bit Quan.	81.10%	8.84	**6.27**	**4.27x**
EfficientNetLite4	*Base (TF Lite)*	84.43%	58.38	**47.75**	-
	Dynamic Quan.	84.44%	15.42	**11.24**	**4.25x**
	Float 16bit Quan.	84.44%	29.32	**23.90**	**2.00x**
	Int 8bit Quan.	84.27%	15.20	**10.90**	**4.38x**

Size reduction is derived from the size comparison between the respective networks' individual quantized models and their Base (TF Lite) model.

Next, we examine the inference latency of optimized models. Latency readings are obtained using the average of the total time taken for inference operations on the entire test set. Table 8 tabulates the inference latency for the optimized Keras models. It is observed that quantization does not necessarily improve inference speeds.

Dynamic quantization on MobileNetV2, DenseNet121, and DenseNet169 resulted in higher latencies than the baseline pruned models. The increase could be attributed to the time spent when activations of the intermediate layers are dynamically quantized during inference. The latency increments on MobileNetV2, DenseNet121, and DenseNet169 are 5 ms, 17 ms, and 20 ms, respectively. Latency improvements on MobileNet and NASNetMobile are 3 ms and 2 ms, respectively, which are negligible considering the latencies are in the hundreds of milliseconds range.

Across all models, except for the NASNetMobile, Float 16-bit quantization ran with the same or negligible differences in latency compared to the respective pruned models. This performance is expected since the true advantage of Float 16-bit quantization is only realized when the inference operations are running on GPUs that natively operate on 16-bit computations. Thus, Float 16-bit quantization affected the models minimally when they were running on the RPi's 32-bit CPU.

All models optimized with Integer 8-bit quantization generated the lowest latency among the optimization methods, albeit with a slight decrement in accuracy on all models. Latency improvements are 38 ms, 24 ms, 8 ms, 143 ms, and 169 ms for MobileNet, MobileNetV2, NASNetMobile, DenseNet121, and DenseNet169, respectively. The significant latency improvements demonstrate the advantage of Integer 8-bit quantization on lightweight edge devices like the RPi.

Table 8. Quantized Keras Applications Models (Latency)

	Optimized Model	***Top-1 Accuracy***	***Latency (ms)***
MobileNet	*Pruned (TF Lite)*	82.19%	170
	Dynamic Quan.	82.02%	167
	Float 16bit Quan.	82.17%	170
	Int 8bit Quan.	81.88%	132
MobileNetV2	*Pruned (TF Lite)*	77.52%	147
	Dynamic Quan.	77.27%	152
	Float 16bit Quan.	77.52%	132
	Int 8bit Quan.	77.04%	123
NASNetMobile	*Pruned (TF Lite)*	73.05%	314
	Dynamic Quan.	73.13%	310
	Float 16bit Quan.	73.03%	311
	Int 8bit Quan.	72.40%	306
DenseNet121	*Pruned (TF Lite)*	85.36%	710
	Dynamic Quan.	85.35%	727
	Float 16bit Quan.	85.36%	656
	Int 8bit Quan.	82.97%	567
DenseNet169	*Pruned (TF Lite)*	85.51%	833
	Dynamic Quan.	85.54%	853
	Float 16bit Quan.	85.51%	827
	Int 8bit Quan.	85.10%	664

Table 9 records the inference latency for the optimized EfficientNetLite models. It is observed that the Top-1 accuracy of the models increases with each variant of

EfficientNetLite, 0 to 4. However, at the same time, inference latency also increases significantly. This is due to the complexity increase across the EfficientNetLite models.

Dynamic quantization resulted in higher latencies across all models, which similarly is due to activations being dynamically quantized to 8 bits during the inference operations. Latency increments are 14 ms, 6 ms, 15 ms, 49 ms, and 96 ms for EfficientNetLite0 to 4, respectively.

Float 16-bit quantization has very minimal impact on the EfficientNetLite0 and 1 models. Both accuracy and latency differences are negligible. For EfficientNetLite2 and 3, latency improvements are 36 ms and 45 ms, respectively. There is a slight increase in latency of 5 ms on EfficientNet4. Negligible accuracy changes of less than 0.03% are observed.

Lastly, Integer 8-bit quantization vastly improves latency with minor impact on accuracy across all models. Latency improvements are 27 ms, 58 ms, 75 ms, 97 ms, and 139 ms for EfficientNetLite0 to 4, respectively.

Table 9. Quantized TF Lite Models (Latency)

	Optimized Model	***Top-1 Accuracy***	***Latency (ms)***
EfficientNetLite0	*Base (TF Lite)*	77.83%	157
	Dynamic Quan.	77.60%	171
	Float 16bit Quan.	77.84%	157
	Int 8bit Quan.	77.68%	130
EfficientNetLite1	*Base (TF Lite)*	79.59%	254
	Dynamic Quan.	79.54%	260
	Float 16bit Quan.	79.62%	255
	Int 8bit Quan.	79.43%	196
EfficientNetLite2	*Base (TF Lite)*	80.09%	337
	Dynamic Quan.	80.08%	352
	Float 16bit Quan.	80.08%	301
	Int 8bit Quan.	79.95%	262
EfficientNetLite3	*Base (TF Lite)*	81.02%	490
	Dynamic Quan.	80.81%	539
	Float 16bit Quan.	81.05%	445
	Int 8bit Quan.	81.10%	393
EfficientNetLite4	*Base (TF Lite)*	84.43%	789
	Dynamic Quan.	84.44%	885
	Float 16bit Quan.	84.44%	794
	Int 8bit Quan.	84.27%	650

2. Model Selection for Vehicle Recognition

There is no straightforward way of determining a model best suited for vehicle recognition on the RPi with the metrics collected on inference accuracy, latency, and compressed model size. Nonetheless, we can formulate some prioritization principles to reach a decision.

From a security system implementation perspective, inference accuracy must be the top priority. Recognizing a vehicle incorrectly may hamper investigations or, in the worst case, wrongly accuse a suspect due to incorrect information. Similarly, to support real-time video surveillance, the next metric we should prioritize is the inference latency of the CNN. The latency has to be as low as possible and preferably below the threshold of human awareness of 200 ms as described in [44]. Model size is prioritized as the least important metric simply because models can be split or retransmitted after a failed attempt, and we have already capped the model size to under 100 MB from the initial pre-trained model selection phase.

From the test results generated in Tables 6, 7, 8, and 9, we observe that the inference accuracy ranges between 72.89% and 85.54%. Since accuracy is the top priority, we filter out the less accurate models of lower than 80% accuracy.

Presented in Table 10 are models that generate inference accuracy of above 80%. We can now sieve out MobileNet, highlighted in green borders, as the only model that fulfills both prioritized criteria of above 80% accuracy and lower than 200 ms latency. It also seems that any quantized versions of the MobileNet model are viable for vehicle recognition. On the other hand, when we consider accuracy to be rounded to 82% across all MobileNet variants and ignore the negligible compressed size difference between Float 16-bit and Integer 8-bit, the latter offered the lowest inference latency. Hence, in this case MobileNet with Integer 8-bit quantization is the model best suited for vehicle recognition on the RPi.

Table 10. Keras and TF Lite Models with Above 80% Accuracy

	Optimized Model	*Top-1 Accuracy*	*Latency (ms)*	*Compressed Size (MB)*
MobileNet	*Pruned (TF Lite)*	82.19%	170	13.31
	Dynamic Quan.	82.02%	167	5.06
	Float 16bit Quan.	82.17%	170	9.62
	Int 8bit Quan.	81.88%	132	5.19
DenseNet121	*Pruned (TF Lite)*	85.36%	710	26.80
	Dynamic Quan.	85.35%	727	6.43
	Float 16bit Quan.	85.36%	656	13.48
	Int 8bit Quan.	82.97%	567	6.27
DenseNet169	*Pruned (TF Lite)*	85.51%	833	47.75
	Dynamic Quan.	85.54%	853	11.24
	Float 16bit Quan.	85.51%	827	23.90
	Int 8bit Quan.	85.10%	664	10.90
EfficientNetLite2	*Base (TF Lite)*	80.09%	337	17.06
	Dynamic Quan.	80.08%	352	4.80
	Float 16bit Quan.	80.08%	301	8.74
	Int 8bit Quan.	79.95%	262	4.75
EfficientNetLite3	*Base (TF Lite)*	81.02%	490	26.80
	Dynamic Quan.	80.81%	539	6.43
	Float 16bit Quan.	81.05%	445	13.48
	Int 8bit Quan.	81.10%	393	6.27
EfficientNetLite4	*Base (TF Lite)*	84.43%	789	47.75
	Dynamic Quan.	84.44%	885	11.24
	Float 16bit Quan.	84.44%	794	23.90
	Int 8bit Quan.	84.27%	650	10.90

C. CHAPTER SUMMARY

This chapter shared the primary findings of this study and discussed the effects of pruning and quantization on the selected Keras and TF Lite models. It also described the selection methodology to find the model best suited for vehicle recognition on the RPi.

In the next chapter, we conclude the study and discuss possible future work.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND FUTURE WORK

This research focused on training deep learning models for vehicle recognition, studying the resultant effects of applying model optimization techniques, pruning and quantization, and eventually identifying the best performing model for vehicle recognition on a lightweight and low-power edge device, the Raspberry Pi.

Two classifier architectures were developed on the Google Colaboratory (Colab) platform to perform transfer learning of selected Keras and TensorFlow Lite models on the Stanford Cars dataset [34]. The models were then optimized with post-training pruning and quantization and evaluated with a Python script on an RPi 4. Metrics including model accuracy, latency, and model size were collected for analysis and evaluation.

Our research analyzed the performance of the models trained with transfer learning from ImageNet checkpoints [32] and optimized with pruning and quantization. By prioritizing the metrics based on the usage scenario, we have identified the MobileNet model, pruned and quantized to Integer 8-bit, as the model best suited to run vehicle recognition on the RPi.

A. SUMMARY

This research explored transfer learning of state-of-the-art deep learning models pre-trained on ImageNet, with the Stanford Cars dataset [34] for the task of vehicle recognition. Referencing the list of Keras Application models trained on ImageNet in Table 1 (Chapter III, Section C.2), we proposed a model selection criteria of model size within 100 MB and 20 million parameters. Models with larger sizes or a higher number of parameters are expected not to perform well on resource-constrained devices such as the RPi in inference latency and compressed model size.

Initially, top layers were added to the Keras models, and only the top layers were trained on the cars dataset in the transfer learning process. This approach, however, had led to a poor accuracy score of 41% on the MobileNetV2 model. The issue was resolved by first augmenting the training data with selected random effects and training the entire model on the data instead of only training the added top layers.

The research also demonstrated the effects of optimization techniques, pruning and quantization, on the transfer learned models. It was observed that applying both pruning and quantization reduces all model sizes by more than half of the baseline models. At the same time, accuracy is maintained within the range of plus or minus 2.5% from the baseline models' accuracy. Specifically, pruning only the fully connected (dense) layers of the added top layers was found to drastically reduce the model sizes by a range of 3.18x to 4.57x and improve accuracy slightly, in the range of 0.18% and 2.32%, across all models. In addition, all quantization methods, when applied to the models, are observed to perform as intended. Dynamic and Integer 8-bit quantization reduced model sizes by up to 4.39x, and Float 16-bit quantization reduced model sizes by up to 2.00x, on top of the size reduction achieved via pruning. In latency, Dynamic quantization caused some latency increments, whereas Integer 8-bit quantization provided significant latency improvements, and Float 16-bit quantization caused a negligible impact on latency for most models.

With the models optimized, we found no straightforward way to determine the best performing model simply by looking at the inference accuracy, latency, and compressed model sizes. Hence, principles to prioritize the metrics were formulated to aid the selection process. The MobileNet model, pruned and quantized to 8-bit Integer, is identified as the model best suited for vehicle recognition on the RPi.

B. FUTURE WORK

While there is success in applying the optimization techniques, pruning and quantization, and identifying the best performing model, this research also presents several challenges that warrant future work.

As mentioned, the dataset used in this study is a widely used benchmark dataset for machine learning research on vehicle recognition. Nevertheless, the small number of images per class and overall small dataset size require much data pre-processing to achieve an acceptable accuracy score. It is also observed that the dataset contains car models manufactured before 2013, which is considered outdated for practical usage of vehicle recognition today. Future work in this area should collect real-world images of the latest vehicles, constantly update the dataset, and re-train the network on the new data. A good

source for capturing such data is at the facility of interest, which presents applicable real-world data. Alternatively, vehicle images by make, model, and year can be collected by scraping images off the web, then annotating and adding them to the current dataset to maintain relevance and improve diversity.

Although re-training and optimizing the pre-trained models with the existing cars dataset achieved reasonable accuracy levels, to push the boundaries of improving performance, further research into customizing existing models or developing lightweight custom models specifically to run vehicle recognition tasks on resource-constrained devices or environments should be explored.

Another area for research is to combine federated learning with the identified transfer learned model to enable distributed learning across edge devices. Doing so enables the training tasks to be shifted to the edge devices and allows constant updating of the model as new data is captured on the edge devices.

Furthermore, additional work on license plate recognition should be included to achieve a complete vehicle recognition system for security at military bases and facilities. This approach, however, would add latency when both recognition tasks run sequentially. One way to resolve that issue would be to explore training an ML model capable of multi-tasking via dual head output.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] H. T. Ozdemir and K. C. Lee, "Threat-detection in a distributed multi-camera surveillance system," U.S. Patent No. 8,760,519 B2, Jun. 24, 2014. [Online]. Available: https://patents.google.com/patent/US20080198231

[2] J. Wei, "AlexNet: The architecture that challenged CNNs," Towards Data Science. Accessed Nov.15, 2020. [Online]. Available: https://towardsdatascience.com/alexnet-the-architecture-that-challenged-cnns-e406d5297951

[3] X. Luo, R. Shen, J. Hu, J. Deng, L. Hu, and Q. Guan, "A deep convolution neural network model for vehicle recognition and face recognition," *Procedia Computer Science*, vol. 107, pp. 715–720, Dec. 2017.

[4] H. J. Lee, I. Ullah, W. Wan, Y. Gao, and Z. Fang, "Real-time vehicle make and model recognition with the residual SqueezeNet architecture," *Sensors*, vol. 19, no. 5, p. 982, Feb. 26, 2019.

[5] S. Lee, K. Son, H. Kim, and J. Park, "Car plate recognition based on CNN using embedded system with GPU," *2017 10th International Conference on Human System Interactions (HSI)*, 2017, pp. 239–241, doi: 10.1109/HSI.2017.8005037.

[6] L. Johnson, "Real-time object tracking with TensorFlow, Raspberry Pi, and Pan-Tilt HAT," Towards Data Science. Accessed Nov. 15, 2020. [Online]. Available: https://towardsdatascience.com/real-time-object-tracking-with-tensorflow-raspberry-pi-and-pan-tilt-hat-2aeaef47e134

[7] J. Schmidhuber, "Deep learning in neural networks: an overview," *Neural Networks*, vol. 61, pp. 88–117, Jan. 2015.

[8] J. McGonagle et al., "Feedforward neural networks," Brilliant. Accessed Nov. 15, 2020. [Online]. Available: https://brilliant.org/wiki/feedforward-neural-networks/

[9] "Convolutional neural network," eLtronics villa. Accessed Nov. 15, 2020. [Online]. Available: https://medium.com/@eltronicsvilla17/convolutional-neural-network-1a02f472a90c

[10] "An intuitive guide to convolutional neural networks," FreeCodeCamp. Accessed Nov. 15, 2020. [Online]. Available: https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/

[11] "Convolutional layer," class notes for CS231n: Convolutional Neural Networks for Visual Recognition, Dept. of Comp. Sci., Stanford University, Palo Alto, CA, USA, spring 2021. Accessed Nov. 15, 2020. [Online]. Available: https://cs231n.github.io/convolutional-networks/#conv

[12] W. Zhao, "Research on the transfer learning of the vehicle logo recognition," *AIP Conference Proceedings*, vol. 1864, p. 020058, Aug. 2017.

[13] M. Stewart, "Simple introduction to convolutional neural networks," Towards Data Science. Accessed Nov. 15, 2020. [Online]. Available: https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac

[14] S. Amidi and A. Amidi, "Convolutional neural networks cheatsheet," Stanford. Accessed Nov. 15, 2020. [Online]. Available: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

[15] I. Shafkat, "Intuitively Understanding Convolutions for Deep Learning," Towards Data Science. Accessed Nov. 15, 2020. [Online]. Available: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

[16] A. Kumar, S. Sarkar, and C. Pradhan, "Malaria disease detection using CNN technique with SGD, RMSprop and ADAM optimizers," in *Deep Learning Techniques for Biomedical and Health Informatics*, S. Dash, B. Acharya, M. Mittal et al., Eds. Cham, Switzerland: Springer, Nov. 2019, pp. 211–230. [Online]. https://doi.org/10.1007/978-3-030-33966-1_11

[17] S. Saha, "A comprehensive guide to convolutional neural networks," Towards Data Science. Accessed Feb. 26, 2021. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[18] D. Gupta, "Fundamentals of deep learning – activation functions and when to use them?" Analytics Vidhya. Accessed Nov. 13, 2020. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them

[19] M. Labs, "Secret sauce behind the beauty of deep learning: Beginners guide to activation functions," Towards Data Science. Accessed Nov. 13, 2020. [Online]. Available: https://towardsdatascience.com/secret-sauce-behind-the-beauty-of-deep-learning-beginners-guide-to-activation-functions-a8e23a57d046

[20] S. Jadon, "Introduction to different activation functions for deep learning," Medium. Accessed Nov. 13, 2020. [Online]. Available: https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

[21] "Introduction to pooling layer, " Geeks for Geeks. Accessed Nov. 15, 2020. [Online]. Available: https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/

[22] L. Guerra, B. Zhuang, I. Reid and T. Drummond, *Automatic pruning for quantized neural networks*. *ArXiv*, abs/2002.00523v1, Feb. 2020. [Online]. https://arxiv.org/abs/2002.00523

[23] L. Y. Pratt, "Discriminability-based transfer between neural networks," in *Proceedings of the 5th International Conference on Neural Information Processing Systems (NIPS'92)*, 1992, pp. 204–211.

[24] "Quantization," TensorFlow. Accessed Mar. 9, 2021. [Online]. Available: https://www.tensorflow.org/lite/performance/model_optimization#quantization

[25] R. Zhao, Y. Hu, J. Dotzel, C. D. Sa, and Z. Zhang, *Improving neural network quantization without retraining using outlier channel splitting*. *ArXiv*, abs/1901.09504v3, May 2019. [Online]. https://arxiv.org/abs/1901.09504

[26] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713, doi: 10.1109/CVPR.2018.00286.

[27] D. Zhang, J. Yang, D. Ye, and G. Hua, "LQ-Nets: Learned quantization for highly accurate and compact deep neural networks," in *Computer Vision – ECCV 2018,* V. Ferrari, M. Hebert, C. Sminchisescu, Y. Weiss, Eds. ECCV 2018. Lecture Notes in Computer Science, Springer, Cham, vol 11212, Oct. 2018.

[28] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, *Pruning filters for efficient ConvNets*. *ArXiv*, abs/1608.08710v3, Aug. 2016. [Online]. https://arxiv.org/abs/1608.08710

[29] A. Chen, "Pruning convolutional neural networks," Towards Data Science. Accessed Nov.12, 2020. [Online]. Available: https://towardsdatascience.com/pruning-convolutional-neural-networks-cae7986cbba8

[30] S. Yegulalp, "What is TensorFlow? The machine learning library explained," InfoWorld. Accessed Nov. 13, 2020. [Online]. Available: https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html

[31] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," in *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, Aug. 2019, doi: 10.1109/JPROC.2019.2918951.

[32] “Keras applications,” Keras. Accessed Mar. 1, 2021. [Online]. Available: https://keras.io/api/applications/

[33] “TensorFlow lite model maker,” TensorFlow. Accessed Mar. 1, 2021. [Online]. Available: https://www.tensorflow.org/lite/guide/model_maker

[34] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, “3D object representations for fine-grained categorization,” *2013 IEEE International Conference on Computer Vision Workshops*, Sydney, NSW, Australia, 2013, pp. 554–561, doi: 10.1109/ICCVW.2013.77.

[35] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA, MIT Press, 2016.

[36] N. Benavides and C. Tae, “Fine-grained image classification for vehicle makes and models using convolutional neural networks.” Accessed Mar. 1, 2021. [Online]. Available: http://cs230.stanford.edu/projects_spring_2019/reports/18681590.pdfns/

[37] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” *2009 IEEE Conference on Computer Vision and Pattern Recognition*, Miami, FL, USA, 2009, pp. 248–255, doi: 10.1109/CVPR.2009.5206848.

[38] “Model optimization,” TensorFlow. Accessed Mar. 9, 2021. [Online]. Available: https://www.tensorflow.org/model_optimization/guide/pruning#overview

[39] “Trim insignificant weights,” TensorFlow. Accessed Mar. 9, 2021. [Online]. Available: https://www.tensorflow.org/lite/performance/model_optimization#pruning

[40] “Pruning comprehensive guide,” TensorFlow, Accessed Mar. 9, 2021. [Online]. Available: https://www.tensorflow.org/model_optimization/guide/pruning/comprehensive_guide

[41] “Quantization aware training,” TensorFlow. Accessed Mar. 9, 2021. [Online]. Available: https://www.tensorflow.org/model_optimization/guide/quantization/training#general_support_matrix

[42] “Post-training quantization,” TensorFlow. Accessed Mar. 9, 2021. [Online]. Available: https://www.tensorflow.org/lite/performance/post_training_quantization

[43] “TensorFlow models on the Edge TPU,” Coral. Accessed Mar. 9, 2021. [Online]. Available: https://coral.ai/docs/edgetpu/models-intro/#quantization

[44] “Upgrade to superhuman reflexes without feeling like a robot,” IEEE Spectrum. Accessed Apr. 27, 2021. [Online]. Available: https://spectrum.ieee.org/the-human-os/biomedical/devices/enabling-superhuman-reflexes-without-feeling-like-a-robot

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California