Object Detection Networks at the Edge: Hardware Optimization and Intelligent Transportation Systems Applications

A Thesis Submitted to the

College of Graduate and Postdoctoral Studies

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

in the Department of Electrical and Computer Engineering

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

By

Juan Fernando Yépez Rodríguez

© Juan Fernando Yépez Rodríguez, October 2021. All rights reserved. Unless otherwise noted, copyright of the material in this thesis belongs to the author.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Electrical and Computer Engineering
3B48 Engineering Building
University of Saskatchewan
57 Campus Drive
Saskatoon, Saskatchewan S7N 5A9
Canada

Or

Dean of College of Graduate and Postdoctoral Studies
116 Thorvaldson Building
University of Saskatchewan
110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

In recent years, deep learning (DL) and especially Convolutional Neural Networks (CNNs) have become a key component of many computer vision systems and applications due to their demonstrated capability to accurately process visual information. Object detection is one of the most important and challenging problems capable of being solved by DL; in general, higher object detection accuracy can be achieved by DL compared to other techniques. However, DL tends to require expensive GPUs or cloud-based services (the latter requiring a high internet bandwidth, latency, and other associated costs), making DL applications traditionally very expensive to implement in practice. This thesis emphasizes the optimization of DL computation for object detection and proposes designs of real-time Intelligent Transportation Systems (ITS) applications at the edge using hardware accelerators.

To optimize DL computation, a novel stride 2 Winograd method is proposed for deep neural network (DNN) inference optimization. The proposed method provides new algorithms that trade expensive multiplications for cheap additions, thereby increasing efficiency by vastly decreasing computational complexity. The proposed algorithms support 1D, 2D, and 3D input for CNNs. Additionally, a novel Processing Element (PE) is proposed to process stride one and two convolution in the same FPGA module. These algorithms, implemented using a GPU and an FPGA, are demonstrated to provide better efficiency compared to regular convolution implementations for a variety of kernels.

Additionally, three ITS applications are proposed. The first application is a License Plate Localization (LPL) system constructed using an architecture comprised of bottleneck depth-separable convolutions with inverted residuals. The second proposed application is a novel two stage real-time deep CNN recognition system for decals issued by the Commercial Vehicle Safety Alliance (CVSA). The third proposed application is a novel three stage real-time deep learning-based edge system for hazardous materials (HAZMATs) recognition. The designed custom object detection architectures for ITS applications are capable of highly accurate real-time prediction on edge computing

devices (Intel, Google, and/or NVIDIA), thus providing enormous cost and performance advantages compared to current implementations.

Acknowledgements

The research works presented in this thesis are sponsored by the Natural Sciences and Engineering Research Council (NSERC) of Canada, Mitacs of Canada, International Road Dynamics (IRD) Inc., and the Department of Electrical and Computer Engineering at the University of Saskatchewan.

The compilation of this thesis would have been impossible without the love and support I received from my lovely wife, Lucia. She stayed by my side during the many arduous and sleepless nights. She was always my source of encouragement in difficult situations.

Thanks to God for the gift of life and special thanks for the lovely gift of my children, Anthony and Juan Andrés. They have been my source of motivation and joy each day.

I would also like to acknowledge and appreciate the guidance and constructive role played by my supervisor, Dr. Seok-Bum Ko. Without his thoughtful insights, comments and patience with me, this task would have been difficult to accomplish. He assisted me in seeing many new perspectives in every draft that I presented to him. This has translated my view and broadened my scope regarding this area. Thank you.

Sincere thanks to all my colleagues for assisting me throughout my PhD program. I also extend my gratitude to all the ECE faculties for the important lessons regarding this particular discipline.

I am also indebted to my family and friends who were so helpful. All they did may be too much to mention on this piece. Special thanks to my mother Cecilia, my sister Andrea, and my brother Christian for their unwavering support. Last but not least, thanks to my father, Juan, who is not with me physically but spiritually.

Thank you all so much.

Contents

Permission to Use i
Abstract ii
Acknowledgements iv
Contents v
List of Tables ix
List of Figures xi
List of Abbreviations xiii
Part I Preface 1
Chapter 1: Introduction 2
1.1 Intelligent Transportation Systems 2
1.2 Motivation of Research Works 4
1.3 Research Objectives 5
1.4 Overview of Research Works 6
1.5 Summary of Contribution 8
Chapter 2: Background 12
2.1 Machine Learning and Deep Learning 12
2.2 Convolutional Neural Networks 13
2.2.1 Stride in Convolutional Neural Networks 15
2.2.2 Padding in Convolutional Neural Networks 16
2.3 Optimized Layers for Convolutional Neural Networks 17
2.3.1 Deep Residual Learning 17
2.3.2 Depthwise Separable Convolution 18
2.3.3 Inverted Bottleneck Networks 19
2.3.4 Fused Inverted Bottleneck Layers (Expansion) 19
2.3.5 Tucker Decomposition 19
2.3.6 Linear Bottlenecks 20
2.3.7 FFT Based Convolution 22
2.3.8 Winograd Algorithm 22
2.3.9 Winograd Works 25
2.4 Object Detection 26

2.4.1 Multistage Object Detection Systems 28
2.4.2 Video Object Detection Systems 29
Part II Hardware Optimization for Convolutional Neural Networks 30
Chapter 3: Stride 2 Winograd for Convolutional Neural Networks 32
3.1 Introduction 32
3.2 Proposed Winograd with Stride 2 34
3.2.1 One-dimension 34
3.2.2 Two-dimensions 39
3.2.2.1 Using Kernel 3×3 39
3.2.2.2 Using Kernel 5×5 41
3.2.2.3 Using Kernel 7×7 43
3.2.3 Three-dimensions 46
3.3 CNN Architectures with Layer Stride > 1 50
3.4 GPU Implementation 51
3.5 FPGA Implementation 53
3.5.1 CNN Architecture 53
3.5.2 FPGA Implementation 54
3.5.3 Memory Access 55
3.5.4 Proposed PE Architecture 56
3.5.4.1 Input Tile From Registers 57
3.5.4.2 Splitter Block 57
3.5.4.3 Input Transform Block 58
3.5.4.4 Filter Transform Block 59
3.5.4.5 Multiplication Block 59
3.5.4.6 Inverse Transform 59
3.5.4.7 Accumulator Block 59
3.5.5 Parallelization 60
3.5.6 Results 61
3.6 Summary 64
Part III Applications for Intelligent Transportation Systems 66
Chapter 4: Deep Learning-based Embedded License Plate Localization System 68

4.1 Introduction 68
4.2 Proposed Solution 70
4.2.1 Neural Network Description 70
4.2.2 Training Process 74
4.2.3 Inference Process 78
4.2.4 Proposed DL LPL Algorithm 79
4.3 Results 82
4.3.1 Dataset Information 82
4.3.2 Comparisons to Public Libraries 83
4.3.2.1 The Caltech Dataset 83
4.3.2.2 The University of Zagreb Database 85
4.3.2.3 The NTUA MediaLab Dataset 86
4.3.3 Comparisons to Popular DL Object Detection Frameworks 87
4.3.4 Comparisons to other DL LPL Frameworks 88
4.3.5 Real-life Real-time Testing 89
4.4 Summary 90
Chapter 5: Real-time CVSA Decals Recognition System Using Deep Convolutional Neural Network Architectures 92
5.1 Introduction 93
5.2 Proposed Architecture 95
5.2.1 Windshield Detection, CVSA Decal Detection, and Colour Classification 97
5.2.2 Digit and Corner-cut Detection and Classification 98
5.3 Labelling and Training 99
5.3.1 First Stage 101
5.3.2 Second stage 103
5.4 Real-time Prediction 106
5.5 Results 109
5.5.1 Model Comparison 109
5.5.2 Hardware Accelerators Result Testing 112
5.6 Summary 115
Chapter 6: Real-Time Deep Learning-based Edge System for HAZMAT Recognition 117
6.1 Introduction 118

6.2 Proposed Solution 120
6.2.1 HAZMAT Placard Localization and Classification 120
6.2.2 UN/NA Number Localization and Class Recognition 122
6.2.3 UN/NA Number Recognition... 123
6.3 Dataset Description 124
6.3.1 Dataset 124
6.3.2 Dataset Augmentations... 127
6.4 Training Methodology 128
6.4.1 Model and Training Environment. 128
6.4.2 Models Training 129
6.5 Real-time Prediction Methodology 130
6.6 Results.. 133
6.6.1 Testing Environments... 133
6.6.2 Performance Metric 134
6.6.3 Stage 1 Training Results Summary 135
6.6.4 Stage 1 General Model Comparison 135
6.6.5 Stage 2 and 3 Model Results 138
6.6.6 Edge Hardware Deployment and Real-Time Results... 139
6.6.7 Comparison to Others' Works... 141
6.7 Discussion... 143
6.7.1 General 143
6.7.2 Placard Localization (Stage 1) Deployment Results 143
6.8 Summary... 144
Part IV Conclusion 146
Chapter 7: Conclusions and Future Work 147
7.1 Conclusions... 147
7.2 Future Work.. 151
References 152

List of Tables

2.1 Summary of object detection components 27

3.1 Architecture layers using stride >1 51

3.2 Comparisons of the Regular convolution and Winograd Stride Two 52

3.3 Performance Comparison using stride two in GPU 52

3.4 Modified VGG-16 Architecture with convolution stride two 53

3.5 Resource Utilization of different PEs for kernel=3×3 on INTEL ARRIA-10 60

3.6 Resource Utilization of CNN accelerator on INTEL ARRIA-10 62

3.7 Performance comparison with state-of-the-art FPGA accelerators 63

4.1 Bottleneck Residual Block Structure 71

4.2 License plate localization neural network used as feature extractor 74

4.3 Public datasets used to train the deep learning model 82

4.4 Comparison of LPL algorithms on the Caltech Cars 1999 (rear) 2 dataset 84

4.5 Comparison of license plate localization algorithms on the University of Zagreb plate detection, recognition, and automated storage dataset 85

4.6 Set specifics of the NTUA Medialab LPR Database 86

4.7 Comparison of License Plate Localization Algorithms on the NTUA Medialab LPR Database 87

4.8 The proposed license plate localization deep learning architecture compared to popular DL architectures 88

4.9 The proposed license plate localization deep learning architecture compared to other deep learning license plate localization architectures 88

5.1 First stage backbone 98

5.2 Second stage backbone 99

5.3 Summary of model training and results 111

5.4 Hardware accelerator benchmark 113

6.1 Neural Network Architecture for UN/NA Number Detection and Class Identification Feature Extractor 123

6.2 Modified ResNet-18 Backbone for UN/NA Number Recognition 124

6.3 Histogram of Number of Placards Per Image 125

6.4 Dataset Class-Subset Breakdown 127

6.5 Summary of Model Training and Results for Stage 1 134

6.6 Summary of Model Results for Stage 2 138
6.7 Summary of Model Results for Stage 3 138
6.8 Results: Real-time Prediction Speed (FPS) 140
6.9 Per-Class comparisons 142

List of Figures

2.1 Neocognition 14

2.2 Convolution with stride equal to 1 and 2 15

2.3 Convolution with padding to 1 16

2.4 (a) IBN layer, (b) Fused layer, and (c) Tucker layer 20

2.5 Process of the 3D Winograd algorithm 24

2.6 (a) Image classification , (b) object detection 26

3.1 Proposed convolution with stride=2 for kernel=3 35

3.2 Proposed convolution with stride=2 for kernel=5 36

3.3 Proposed convolution with stride=2 for kernel=7 37

3.4 Proposed convolution with stride=2 for kernel=3×3 40

3.5 Proposed convolution with stride=2 for kernel=5×5 42

3.6 Proposed convolution with stride=2 for kernel=7×7 44

3.7 Proposed convolution with stride=2 for kernel=3×3×3 47

3.8 Deep learning architecture 55

3.9 Proposed PE architecture 56

3.10 Splitter the input in based on the stride selected 58

3.11 Input transform block 58

3.12 Evaluation results of the original VGG-16 architecture compared to the proposed modified VGG-16 architecture 61

4.1 Standard vs. depthwise separable convolution operations to demonstrate the effectiveness of depthwise separable convolution over standard 72

4.2 Architecture of the proposed license plate localization method. 75

4.3 Images of plate regions on vehicles labelled using labelImg 76

4.4 Program flow of proposed multi-threading video capture with motion detection then inference 79

4.5 Frames of a vehicle travelling at highway speed passing under a camera setup 81

4.6 Different obstructions on license plates in images from the NTUA Medialab dataset 82

4.7 Images of vehicles and their corresponding respective localized license plates generated by the proposed method 83

4.8 Real-time can detect one or multiple license plates at the same time 90

5.1 CVSA decal color types 93

5.2 Trucks with CVSA decals on different highways 100
5.3 Windshield and CVSA decal labelled using the LabelImg program 102
5.4 Digit and corner-cut labelled in a CVSA decal 103
5.5 (a) 7 spots filled, (b) 5 spots filled, and (c) 3 spots filled 105
5.6 Real-time CVSA Decals Recognition Systems (CDRS) 108
5.7 Comparison of accuracy (mAP) vs. processing time by stage 110
5.8 DeepStream pipeline for the real-time CDRS on the Jetson Xavier 114
6.1 HAZMAT recognition system 120
6.2 UN/NA number recognition 123
6.3 Example images from the dataset 126
6.4 Processing pipeline as deployed on the Jetsons, possible using Nvidia libraries 131
6.5 Example test set inferences of the quantized SSDlite + Custom model 137

List of Abbreviations

AI	Artificial Intelligence
ALPR	Automatic License Plate Recognition
AMQP	Advanced Message Queuing Protocol
ANN	Artificial Neural Network
AP	Average Precision
ASCII	American Standard Code for Information Interchange
CDRS	CVSA Decals Recognition Systems
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CuDNN	NVIDIA CUDA Deep Neural Network Library
CV	Computer Vision
CVSA	Commercial Vehicle Safety Alliance
DL	Deep Learning
DNN	Deep Neural Network
DSP	Digital Signal Processing
FLOPS	Floating Point Operations Per Second
FPGA	Field-Programmable Gate Array
FPN	Feature Pyramid Network
FPS	Frames per Second
GHz	Gigahertz
GOPS	Giga Operations Per Second
GPU	Graphics Processing Unit
HAZMAT	Hazardous materials
IBN	Inverted Bottleneck Network
IoT	Internet of things

IoU	Intersection over Union
ITS	Intelligent Transportation System
LiDAR	Light Detection and Ranging
LPL	License Plate Localization
LSTM	Long Short-Term Memory
mAP	Mean Average Precision
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
NA	North American
NAS	Neural Architecture Search
NTUA	National Technical University of Athens
OCR	Optical Character Recognition
OpenCL	Open Computing Language
PASCAL	Pattern Analysis, Statistical Modelling and Computational Learning
PE	Processing Element
RAM	Random Access Memory
R-CNN	Region-based Convolutional Neural Network
ReLU	Rectified Linear Unit
R-FCN	Region-based Fully Convolutional Network
RNN	Recurrent Neural Network
ROI	Region of Interest
RPN	Region Proposal Network
SSD	Single Shot Detector
SSVD	Single Shot Video Object Detector
UN	United Nations
USDOT	United States Department of Transportation
VGG	Visual Geometry Group
VOD	Video Object Detection

WMFAs	Winograd Minimal Filtering Algorithms
YOLO	You Only Look Once

Part I

Preface

Chapter 1

Introduction

This chapter presents intelligent transportation systems (ITS), artificial intelligence in the context of ITS, and the importance of optimizing network architectures and hardware. Developing optimized real-time ITS on low-cost devices may increase the utilization of these systems around the world; this has motivated the research works proposed in this thesis. Section 1.1 presents ITS. The motivations of the research works are presented in Section 1.2. Section 1.3 presents the overview of the research works. The contributions of these research works are summarized in Section 1.4.

1.1 Intelligent Transportation Systems

Intelligent Transportation Systems (ITS), officially defined at the 1994 World Congress in Paris, encompass all technology-driven applications within the broad field of transportation designed to improve the general driving experience. ITS incorporates a wide variety of technologies (e.g. within the fields of telecommunication, ICT, networks, automation, and sensors), methods (e.g. data-driven, measurement and statistics, algorithms, computer vision, and artificial intelligence (AI)), and management techniques [1]. ITS objectives include protecting the health and safety of traffic actors and pedestrians, safeguarding the natural environment and its resources, and increasing the effectiveness and streamlining the efficiency of transportation in generals, supply chains, cities and highways, and the broader field of ITS itself [2].

Innovative ITS services produce useful gains within transportation, traffic management, and traffic control. A well-designed system will communicate with applicable users or other systems to encourage safer and more coordinated utilization of transport channels [1]. ITS aims to improve traffic planners and road users' safety, mobility, productivity, and environmental performance [2].

Traditional ITS approaches use dedicated hardware such as inductive loop detectors, radar detectors, and laser detectors to locate vehicles and determine their speeds and characteristics, but such equipment may incur high maintenance and installation costs. Compared to these traditional sensor-based approaches, video cameras are more advantageous in terms of cost and flexibility [3].

Video cameras have long since been deployed for traffic surveillance because they provide important contextual information for human consumption and understanding [4]. Due to decreasing costs (Moore's Law), the number and coverage of road cameras has dramatically increased in recent years, resulting in broad accessibility of image/video data; this has led to the feasibility of camera image-based object detection—an incredibly promising new technique for large-scale traffic data analysis. Video analysis within the scope of smart transportation public safety has led to research and advancements in both academia and industry [4].

Compared with the "traditional" transportation system, the most significant characteristic of ITS is the integration of data-driven approaches including AI [5]. Leveraging AI for transportation may help the sector increase passenger safety, reduce traffic congestion and accidents, lessen carbon emissions, and minimize overall financial expenses for industries, governments, companies, and consumers.

AI can be defined as a technology that allows machines to learn from experience, oftentimes guided by human knowledge. Machines with AI capabilities can learn to mimic humans, automate manual tasks, create aesthetically-pleasing works, and practice continual learning—just like humans [6]. Computer/AI-driven automation frees human operators from repetitive and time-consuming tasks in a way that may also lessen (human) operator fatigue and stimulate higher and more deterministic accuracy. Moreover, specific AI-powered systems may exhibit human-level accuracy and/or continuous learning with new experiences, which may indicate the potential for AI-powered machines to self-improve and perform tasks that require critical-thinking or higher-level functions—possibly without human intervention, depending on task complexity [6]. Governments, the transportation sector, and academic institutions are making significant investments in

this relatively new and near limitless AI field [5]. Novel general or custom-tailored AI-based applications are designed with an emphasis on improving speed and accuracy, but also reliability, efficiency, robustness, quality, and—where applicable—safety.

1.2 Motivation of Research Works

Recent and rapid AI developments have provided unprecedented opportunities to revolutionize different fields, industries, and businesses, including within the transport sector. Intelligent Transportation Systems (ITS) have seen a rise in application development through the implementation of AI methods [5]. One of the most promising areas of research within AI is computer vision.

Computer vision enables computers to process visual data and extract information from that data at a baseline level [7]. Visual input/output tends to be highly interpretable by humans, which can be useful for the design and implementation of certain systems. The general goal of computer vision is to teach computers how to identify, classify, and categorize the visual world as humans do. Although the field of computer vision has existed for some time, the advent of deep learning to computer vision pipelines has led to a revolution in vision technologies and systems [6].

Deep learning (DL) has made computer vision algorithms highly effective for real-world applications [6]. One famous DL architecture originally designed for image data is the convolutional neural network (CNN) [8]. CNNs have made computer vision feasible and relatively inexpensive for industrial applications, leading to high industry investment and adoption, especially for task automation.

In general, DL models consist of many of layers and parameters. For example, Deep Neural Networks (DNN) and Convolutional Neural Networks (CNN) require upwards of millions (sometimes into the billions) of interconnected units and parameters [8]. As a result, running most current DL applications requires high-performance computers with expensive GPUs, or centralized servers and/or cloud-based systems [9]. This can be a

huge limitation in terms of system implementation cost, e.g. on highways or remote locations where the Internet/remote access is limited.

Rather than sending real-time video to the cloud—which can be very expensive in terms of time and data consumption—the main goal of this thesis is the implementation of ITS systems on edge devices. This minimizes both data transmission time and system latency, as well as reduces the cost of data transmission, computation, and storage. Additionally, processing data at the edge can preserve the privacy of users and the integrity of the raw data itself, because uploading the video/images is unnecessary [9].

Due to the limited memory and computational resources of edge devices, power-efficient computation can be designed by reducing the number of hardware-expensive computations (i.e. multiplications) for convolutions making the development of custom and more specialized DL architectures becomes necessary.

1.3 Research Objectives

The main objective of this present study is to unify recent advancements in deep learning architectures with the design of optimized computing hardware, with an emphasis on the development of new and novel applications with the scope of ITS on edge devices.

The specific objectives of this thesis are as follows:

1. Formulate algorithms to enhance the calculation of convolution stride 2—particularly important for deep learning object detection systems—using fewer hardware resources.
2. Design a processing element capable of supporting both stride 1 and 2 convolution, to enhance FPGA efficiency.
3. Benchmark the DSP efficiency of the processing elements implemented on an FPGA.

4. Develop ITS applications based on deep learning architectures that use convolution strides 1 and 2.
5. Design custom optimized deep learning models for specific detection/recognition tasks in each system.
6. Implement and evaluate the ITS applications on available edge devices and hardware accelerators.

1.4 Overview of Research Works

In this thesis, based on the motivation discussed in Section 1.2, optimized CNN algorithms and architectures capable of achieving a high convolution speed are proposed. The algorithms for 1-D, 2-D, and 3-D convolution reduce the number of multiplications when a convolution of stride 2 is performed. A Processing Element (PE) for a 3×3 kernel, compatible with both stride 1 and 2 convolutions is introduced. Therefore, three ITS applications are presented: a License plate localization (LPL) system, a CVSA decal recognition system, and a HAZMAT recognition system. The applications are capable of deployment in real-time in complex outdoor environments. The whole thesis is composed of four parts with seven chapter shown as follows:

- Part I Preface includes:
  - Chapter 1 *Introduction*: presents the importance of deep learning and the motivations, the overview, and the contributions of the research works.
  - Chapter 2 *Background*: introduces the background information required to present the proposed research works.
- Part II Hardware Optimization for Convolutional Neural Networks includes:
  - Chapter 3 *Stride 2 Winograd for Convolutional Neural Networks*: present a novel method to apply the Winograd algorithm to a stride (shift-displacement of a kernel over an input) of two. This method is valid for one, two, or three dimensions. In this chapter, new Winograd versions compatible with a kernel of size three, five, and seven were introduced. The algorithms were

successfully implemented on an NVIDIA K20c GPU and an Intel Arria-10 FPGA. The proposed implementation uses a novel Processing Element (PE) able to perform two Winograd stride one, or one Winograd stride two, operations per clock cycle. Compared to regular convolutions and other designs, the proposed implementations provide fast convolution for stride one and two and high DSP efficiencies.

- Part III Intelligent Transportation Systems Applications include:
  - Chapter 4 *Deep Learning-based Embedded License Plate Localization System*: presents a novel neural network architecture for license plate localization (LPL) based on an inverted residual structure where the shortcut connections are between the linear bottleneck layers. The proposed deep learning (DL) solution was tested against three popular international research databases and achieves state-of-the-art results, proving that the proposed model is accurate and robust. Across those databases, the proposed model surpasses other recent LPL work, including DL-based methods, in terms of accuracy and speed. The proposed architecture is shown to have significant speedup and computational efficiency gains over other DL models, and to have fast per-image localization processing times sufficient for applications deployed on expensive and commodity hardware alike. Using a novel multi-threading video capture with motion detection then inference algorithm, computational efficiency is increased, thus dropping less frames overall and allowing for increased performance. Repeated tests show the proposed method is well-suited to real-time and highly accurate LPL, regardless of hardware.
  - Chapter 5 *Real-time CVSA Decals Recognition System Using Deep Convolutional Neural Network Architectures*: presents a 2-step automatic Commercial Vehicle Safety Alliance (CVSA) decal recognition system using deep convolutional neural network architectures. The MobileDet architecture was used as a baseline for the proposed system and customized to better suit the system's tasks. The first step localizes a vehicle's windshield and the CVSA decal within, and classifies the decal colour. The CVSA decal is

cropped and used as input to the second stage, which localizes and classifies a digit and the corner-cut of a CVSA decal. The custom architectures reduce processing time and exceed accuracies relative to pre-trained architectures. The proposed model was implemented on different edge hardware accelerators, and the performance on each – in terms of high inference speed, real-time video processing, and high mean average precision – was contrasted.

- Chapter 6 *Real-Time Deep Learning-based Edge System for HAZMAT Recognition*: presents a 3-stages cascading system using deep learning networks. The first network localizes and classifies the HAZMAT placard. If the placard contains a United Nations (UN) / North American (NA) number, the second network localizes that number and identifies the nature of the substance. The third network recognizes the UN/NA number. For both the first and second stage, an SSDlite object detection network was developed using custom backbones based on MobileDet. For the third stage, a segmentation-free UN/NA number recognition network was developed using a lightweight sequence classification model. The system was deployed on a variety of AI edge hardware accelerators from vendors like NVIDIA, Google, and Intel, and performance differences among the accelerators were subsequently compared. For each stage, a detailed comparison with other networks was provided.

- Part IV Conclusions includes:
  - Chapter 7 Conclusions *and Future Work*: includes the conclusions of all presented research works and the plan for futures works.

1.5 Summary of Contribution

In this thesis, a novel set of algorithms are introduced to calculate the convolution when using a kernel sliding by two units about the input; this convolution is called "convolution stride 2". The presented algorithms of convolution stride 2 are based on the conventional Winograd Minimal Filtering Algorithm, which is only formulated for convolutions of stride 1. The proposed novel algorithms calculate the convolution stride

2 results using less multiplications than conventional convolution. They work in one, two and three dimensions using respective kernels of size 3, 5, 7, 3×3, 5×5, 7×7, and 3×3×3. The convolution stride 2 downsamples the feature maps while preserving spatial information via feature learning. As further explained in Part II, this characteristic is particularly important for object detection systems. A novel Processing Element (PE) is presented that is able to perform two Winograd stride one, or one Winograd stride two, operations per clock cycle. The convolution stride 2 was implemented on an NVIDIA K20c GPU and an Intel Arria-10 FPGA.

Furthermore, three deep learning-based systems for ITS are presented. The first is an embedded license plate localization system, the second is a real-time CVSA decal recognition system, and the third is a real-time edge system for HAZMAT recognition. For each system, custom object detection architectures that use a mix of stride 1 and 2 convolution are designed. The proposed architectures are optimized to achieve high accuracies and low processing times. This feature makes the systems suitable to the implementation on varying edge devices. To evaluate the speed of the proposed systems, various hardware accelerators were used, including the Nvidia Jetsons (Nano and Xavier), Intel's Neural Compute Sticks (versions 1 and 2), and Google's Coral USB accelerator. Each ITS application’s deep learning architectures and corresponding accuracies and prediction speeds are analyzed in detail in their respective chapters. To highlight the advantages and improvements of the proposed systems over existing methods, the systems are compared against related works (where applicable).

In this thesis, the computational efficiency for CNNs is improved—especially those architectures designed for object detection—by providing new algorithms to accelerate layers with a convolutional stride of 2; these novel algorithms are introduced in Chapter 3. Using this research, custom object detection architectures capable of real-time prediction on edge computing devices were designed for ITS applications, , thus providing enormous cost and performance advantages compared to current implementations: Chapter 4 presents a license plate localization system, Chapter 5 a real-time recognition system for Commercial Vehicle Safety Alliance (CVSA) decals, and

Chapter 6 a real-time edge system for recognizing hazardous material (HAZMAT) placards.

Below is the list of publications, arranged according to the order of appearance in this thesis:

- Chapter 3 *Stride 2 Winograd for Convolutional Neural Networks:*
    - **J. Yepez** and S. Ko, "Stride 2 1-D, 2-D, and 3-D Winograd for Convolutional Neural Networks", in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 28, n0. 4, pp. 853-863, April 2020.

- Chapter 4 *Deep Learning-based Embedded License Plate Localization System:*
    - **J. Yepez**, R. D. Castro-Zunti and S. Ko, "Deep learning-based embedded license plate localisation system", in IET Intelligent Transport Systems, vol. 13, no.10, pp. 1569-1578, 10 2019.

- Chapter 5 *Real-time CVSA Decals Recognition System Using Deep Convolutional Neural Network Architectures:*
    - **J. Yepez**, R. D. Castro-Zunti and S. Ko, "Real-time CVSA Decals Recognition System Using Deep Convolutional Neural Network Architectures", in IET Intelligent Transport Systems 1–13 (2021), https://doi.org/10.1049/itr2.12103.

- Chapter 6 Real-Time Deep Learning-based Edge System for HAZMAT Recognition*:*
    - **J. Yepez**, R. D. Castro-Zunti and S. Ko, "Real-Time Deep Learning-based Edge System for HAZMAT Recognition", under review Springer Machine Vision and Applications, submitted July 2021.

- Other publications that are not included in this thesis:

- **J. Yepez** and S. Ko, "IoT-Based Intelligent Residential Kitchen Fire Prevention System", in J. Electr. Eng. Technol. 15, pp. 2823-2832, August 2020.
- R.D. Castro-Zunti, **J. Yepez** and S. Ko, "License plate segmentation and recognition system using deep learning and OpenVINO", in IET Intelligent Transport Systems, vol. 14, no. 2, pp. 119-126, 2 2020.
- **J. Yepez**, X. Shi, and S. Ko, "An FPGA-based Closed-loop Approach of Angular Displacement for a Resolver-to-Digital- Converter", 2018 IEEE International Symposium on Circuits and Systems (ISCAS). Florence, 2018, pp. 1-4
- A. Dinh, M. Bayati, M. Bhatti, **J. Yepez**, and J. Zhexin, "Design and Implementation of a Wireless Wearable Band for Gait Analysis," in 6th International Conference on the Development of Biomedical Engineering in Vietnam (BME6), 2018, pp. 693-698.
- Z. Jiang, **J. Yepez**, S. An, and S. Ko, "Fast, accurate and robust retinal vessel segmentation system," Biocybern. Biomed. Eng., pp. 1-10, 2017.
- X. Shi, J. Dai, X. Luo, **J. Yepez**, and S. Ko, "Foreground-Background Separation Guided by Statistical Features of Surveillance Video," IEEE/IEIE ICCE-Asia, pp. 3-6, 2016.

Chapter 2

Background

This chapter present the background information of the proposed works in this thesis. Machine and deep learning concepts are presented in Section 2.1 Section 2.2 presents the convolutional neural networks. Section 2.3 presents optimized layers for convolution. Section 2.4 presents object detection works in the literature.

2.1 Machine Learning and Deep Learning

Machine learning (ML) can be described as an application of artificial intelligence (AI) that allows a system to enhance the accuracy of its algorithm without explicit changes to the algorithm by a (human) programmer [10]. A ML apparatus makes predictions based on sets of data, and thus a wealth of data is usually required for a ML algorithm to be effective. The predictions can include images where the content is divided into different categories, one of the most important applications of deep learning, currently common in machine learning systems.

Deep Learning (DL) and deep neural networks extend the idea of ML by offering a method of multiple layers of predictions [6], with input to successive layers being the output of one or more preceding layers. DL plays an active role in research pertaining to neural networks, computer vision, and pattern recognition, with its success attributable to improved hardware and graphical processing unit (GPU) capabilities, and the advent of accessible big data for training.

The concept of distributed representation of data forms the basis of DL. Distributed representation assumes that many factors, and the complex spatial relationships between their properties, constitute data. These relationships form distinct and perceivable patterns on many levels of abstraction. The depth of a neural network refers to the number of layers through which data passes, with more layers representing increasing.

2.2 Convolutional Neural Networks

A frequently used application of DL is the convolution neural network (CNN). CNN apparatus take inspiration from and are modelled after the ways in which the human mind learns [11], and they have proved incredibly successful at solving problems in computer vision, such as object recognition [12].

The first work on neural networks began in the late 1940s. The Canadian neuropsychologist O.D. Hebb was the pioneer in the computer simulation of neurons. One of the first training algorithms in the field was Hebbian learning.

One important precursor to the convolution neural network topologies is the neocognitron by Kunihiko Fukushima used for handwritten characters in 1980. The proposed topology has many similarities with the modern layout; every layer in the network increases the complexity of the recognized feature.

CNNs are invariant to small amounts of shift, scale, and distortion, and can extract complex features in high-dimensional space; layers that perform feature extractions are called convolutional layers [11].

A notable precursor to modern convolutional neural network topologies is the neocognitron architecture for classifying handwritten characters, designed by Fukushima

[13]. Figure 2.1 the general structure of neocognitron; as can be seen, it has many similarities to modern CNNs, and each network layer increases in complexity to learn a feature desirable to the recognition task.

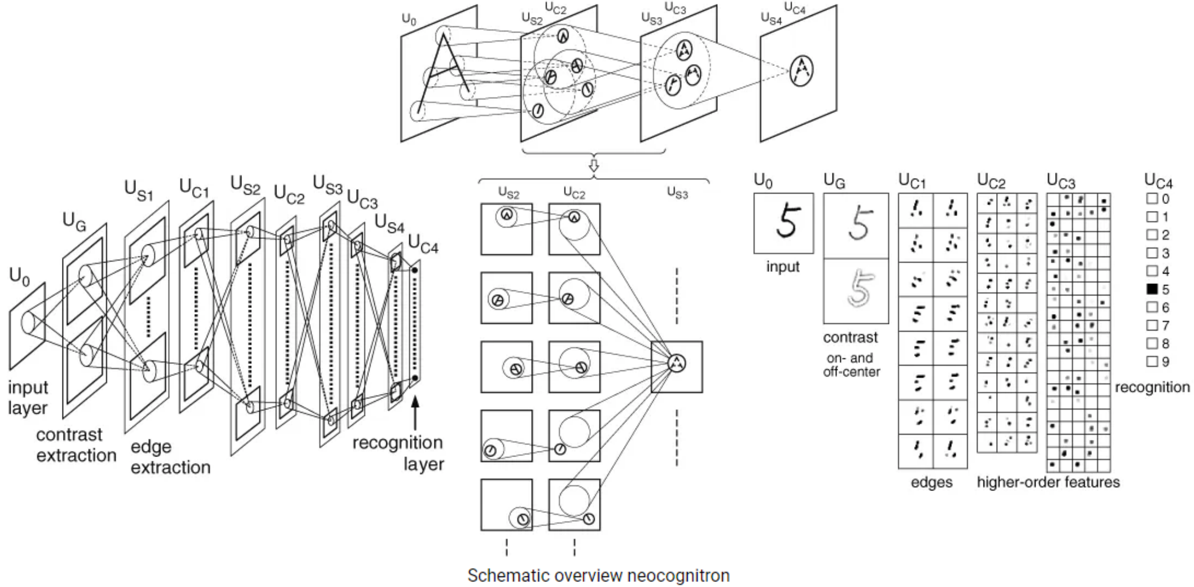


Figure 2.1: Neocognition [13] where U_0 is the input layer, U_G is the layer comprised of contrast-extracting cells, U_S layers are feature-extracting cells, and U_C layers are recognition cells.

Like the neocognitron architecture, convolutional layers extract corners, edges, endpoints, and other visual and hyperdimensional features, organizing them into output feature maps. A hierarchy of convolutional layers may be arranged, and feature maps are extracted from image data via repeated convolutions of the data with either varying filters/kernels or results obtained from earlier layers.

Input to the first layer in a CNN is the data the network is to analyze. Input to the next layer, and for all other layers, is the output feature map generated by the previous layer. The number of layers, the quantity of which is referred to as the depth of the CNN, can potentially reach hundreds—creating a need for massive and efficient computation.

With the conventional convolutional algorithm, each element in the output is computed individually by multiplying and accumulating the corresponding kernel and the input data.

2.2.1 Stride in Convolutional Neural Networks

The stride controls how the kernel convolves around the input. When the stride is one, the kernel is shifted over the input one element at a time. The stride is normally set in such a way that the output volume is an integer.

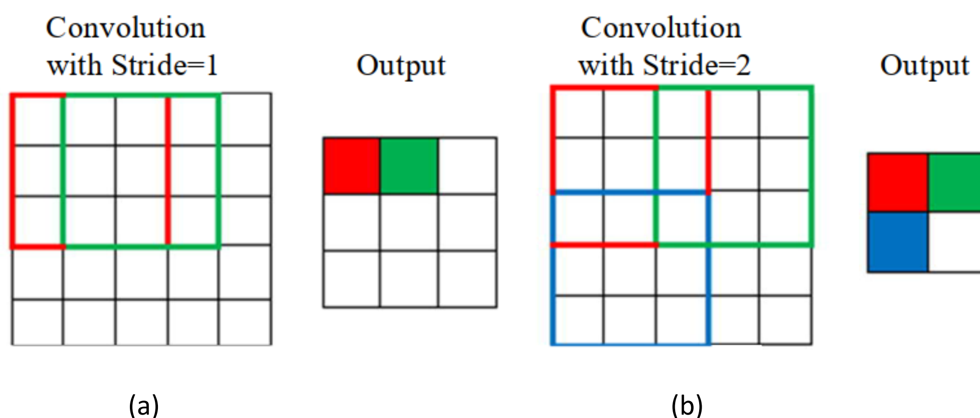


Figure 2.2: Convolution with stride equal to 1 and 2

Figure 2.2 (a) shows a 5×5 input with a 3×3 kernel. With stride one, the output matrix generated is size 3×3. A stride of one is normally used to extract the maximum number of features, as it provides the maximum overlapping between the kernel and input—but at maximum computational complexity. Figure 2.2 (b) shows the kernel shifting by two units over the input, generating an output 2×2 matrix. Generally, when the stride is bigger than one, the receptive fields overlap less. A smaller output is produced. If the stride were three, there would be issues with spacing, as the receptive field would not fit around the input as an integer.

2.2.2 Padding in Convolutional Neural Networks

Padding defines how an image's border is processed, and has a direct effect on the spatial dimensions of the output shape [14]. For example, to achieve a shape with dimensions equal to the input image, padding (e.g. with the average pixel value, with mirroring the image's borders, or typically with 0s) about the input boundaries is necessary. Conversely, unpadded convolution performs the convolution operation only on the pixels of the input image, i.e. without the addition of a border, causing the output dimension size to be smaller than that of the input (by mathematics surrounding convolution).

Figure 2.3 shows a 2D convolution using a kernel size of 3, stride 1, and padding of 1.

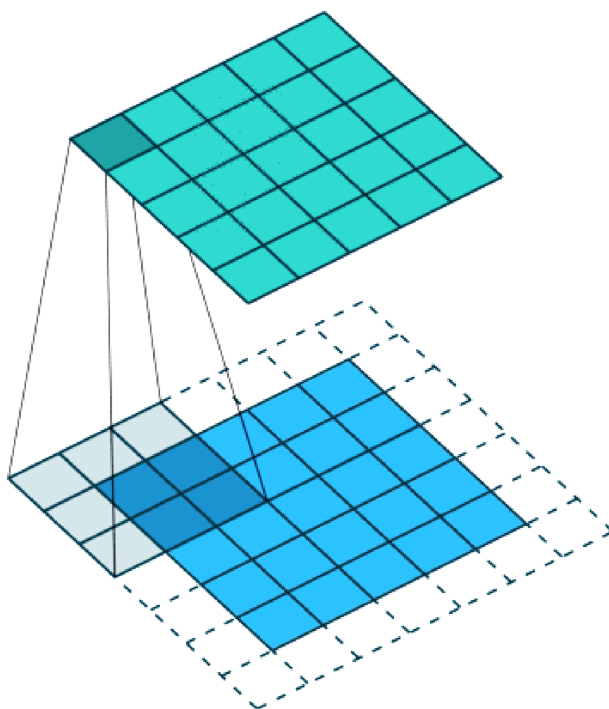


Figure 2.3: Convolution with padding of 1

For an input image with size i, kernel size k, padding p, and stride s, the corresponding size of the output shape o resulting from the convolution is:

$$o = lower_bound\left(\frac{i+2p-k}{s}\right) + 1 \tag{2.1}$$

2.3 Optimized Layers for Convolutional Neural Networks

In addition to using Winograd method to optimize convolutions, in the literature some layers can be found that have been shown to improve model precision and reduce inference time. The layers used in this thesis are presented below.

2.3.1 Deep Residual Learning

DL using residual blocks—first defined in ResNet [7] by researchers at Microsoft—is based on the idea that an approximate complex function is tantamount to an exact residual function (a function plus an error), and thus nonlinear layers in DL that approximate complex functions closer approximate residual functions [7].

Thus, a residual block may be modelled as:

$$\mathbf{y} = F(\mathbf{x}, \{W_i\}) + \mathbf{x} \tag{2.2}$$

Where x is the input vector and y the output vector; F is the function trying to be learned; and x also represents the error to the residual function represented by y. Thus, between stages, element-wise addition occurs between the learned F and the input x; this addition is referred to as a shortcut [7].

These blocks may be combined to effectively create and train very deep neural networks with fewer FLOPS than previous deep learning models [7], and [15].

The original ResNet used a residual block whose channels progressively narrow between input and output via successive convolutions before expansion at the next block layer stage, with a shortcut used to connect successive (wide) layer inputs [7].

2.3.2 Depthwise Separable Convolution

A normal convolution layer may be separated into two or more steps such that applying each step individually lessens overall computation time without tangibly compromising accuracy: this is called a separable convolution [16].

Generally, computation time and expense is reduced in a separable convolution by conducting the same or a similar convolutional operation while using less parameters than a non-separated convolution. Using less parameters has the additional benefit of making the model less prone to overfitting [16].

A depthwise separable convolution is a two-step process involving a depthwise convolution followed by a pointwise convolution. A depthwise convolution performs filtering over multiple channels while allowing the channels to remain separate. This is done by convolving each separate channel with a filter to produce separate outputs before combining/stacking those outputs [15].

A pointwise convolution applies a 1×1 spatial filter across the input channels which identifies features using linear combinations of the inputs.

Depthwise separable convolution reduces computation compared to standard layers by a factor of $k^2dj/(k^2 + dj)$, where k is the kernel size and dj is the dimension of the output channel layers. In the proposed DL apparatus, the convolutions use a kernel of 3×3. Thus, the computational cost is almost 9 times smaller than that of standard convolutions

2.3.3 Inverted Bottleneck Networks

Bottlenecks are introduced in ResNet [12] to reduce the processing time of large convolutions over high-dimensional feature maps. A bottleneck layer does compression, as the feature maps are first projected to have less channels and then projected back during the block's final convolutional layer; both projections are implemented as 1×1 convolutions. It is generally useful to allow fine-grained control over the channel sizes in each layer because of their strong influence on the inference latency.

2.3.4 Fused Inverted Bottleneck Layers (Expansion)

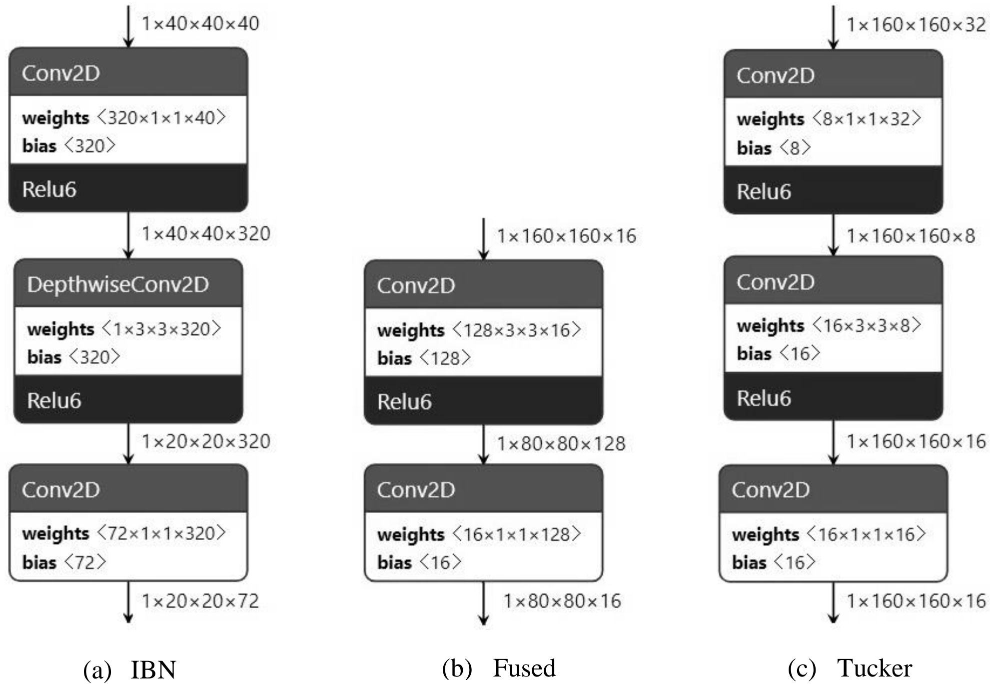
The depthwise-separable convolution is a critical element of an inverted bottleneck [16]. The idea behind the depthwise-separable convolution is to replace an expensive full convolution with a combination of a depthwise convolution (spatial dimension) and a 1 x 1 pointwise convolution (channel dimension). In this layer, a regular convolution replaces the IBN depthwise-separable by fusing its first 1 x 1 (which usually comes with an expansion ratio) and its subsequent K x K depthwise convolution into a single K x K regular convolution. The full convolution allows expansion of the channel size, with the expansion ratio determined by the algorithm.

2.3.5 Tucker Decomposition

The Tucker layer is a compression block that involves a sequence of three operations: a 1 x 1 convolution; a K x K regular convolution; and a 1 x 1 convolution. Figure 2.4 (c). Combining the first 1 x 1 pointwise convolution and the second K x K regular convolution as

one K x K regular convolution gives the fused inverted bottleneck layer in Figure 2.4 (b). The IBN structure shown in Figure 2.4 (a) is equivalent to the sequential structure of approximate evaluation of a regular convolution by using CP decomposition [17]. Thus, all the layers can be linked to Tucker/CP decomposition.

Regular, pointwise, and depthwise convolutions can be perform using these functions. In this thesis, these blocks were used to customize the backbones of each stage according to the type of objects to identify.



(a) IBN (b) Fused (c) Tucker

Figure 2.4: **(a)** IBN layer, **(b)** Fused layer, and **(c)** Tucker layer

2.3.6 Linear Bottlenecks

A bottleneck layer refers to a low-dimensional representation of relevant information to a convolutional layer, and may be used to predict features and for classification [18].

Non-linear activation functions are used in neural networks because many successive matrix multiplications often cannot be simplified to one numerical operation. These activation functions collapse channel information between successive convolutional layers and thus allow the construction of networks of depth [15].

A frequently used non-linear activation function is the rectified linear unit (ReLU), defined as follows:

$$ReLU(x) = \begin{cases} x, & if\ x > 0 \\ 0, & if\ x \leq 0 \end{cases} \qquad (2.3)$$

Applying (1) to data passing through successive convolutional layers within a neural network creates inherent information loss between those layers, which can be combatted by increasing the network's channels and thus its capacity.

For any n-dimensional CNN layer Li there forms a set of activation tensors T corresponding to an abstraction of relevant information; it has been believed that for any Li there exists some low-dimensional manifold M ∈ Rn which entirely embeds T.

It can be shown that if M exists in a higher-dimensional activation space, a ReLU operation which produces non-zero M is congruous with a linear transformation about the ReLU's inputs, and thus the ReLU operation may preserve information relating to the inputs or M.

Thus, with the assumption that M is low-dimensional and exists in a higher-dimensional activation space, the bottleneck may be treated as a linear operation by simply not applying a non-linear activation function to it [15].

2.3.7 FFT Based Convolution

By the convolution theorem, convolution can be performed using Fourier transforms:

$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g)) \tag{2.4}$$

$\mathcal{F}$ and $\mathcal{F}^{-1}$ are Fourier and inverse Fourier transforms [19]. In the discrete case, f and g must have the same number of elements, which can be accomplished by padding zeros to the shorter signal. The discrete Fourier transform (DFT) results in a circular convolution, from which the correct result of the convolution can be extracted by taking the last $|f| - |g| + 1 = m$ elements.

FFT-based convolution is more efficient in practice when there are long numbers of convolutions [20], e.g. N>100, where the savings become enormous compared to ″direct” convolution; this is because direct convolution requires on the order of N^2 operations (multiplications and additions), whereas FFT-based convolution requires on the order of $N \times lg(N)$ operations, where $lg(N)$ denotes the logarithm-base-2 of N.

2.3.8 Winograd Algorithm

The conventional convolution algorithm is simple to implement, but it is not efficient. An efficient alternative convolution method can be realized via the Winograd minimal filtering algorithm [21].

In the case of a size four input data vector and size three kernel vector, conventional convolution requires six multiplications to generate the final result, whereas the Winograd algorithm requires only four multiplications.

The one-dimensional convolution using the Winograd algorithm can be formulated using the transformation matrices A, B, and G, input data d, and kernel g as follows:

$$Y = A^T\left[\left[(Gg) \cdot (B^T d)\right]\right] \tag{2.5}$$

[21] introduces the matrices for F(2,3) as follows:

$$g = [g_0 \quad g_1 \quad g_2]^T$$

$$d = [d_0 \quad d_1 \quad d_2 \quad d_3]^T \qquad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \qquad B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{2.6}$$

These matrices are also compatible for 2D. In the two-dimensional Winograd algorithm, F(m×m, r×r), the output size is m×m, the kernel size is r×r, and the input size is n×n, where n = m + r – 1. The Winograd algorithm for 2D can be written in matrix form as follows:

$$Y = A^T[(GgG^T) \cdot (B^T dB)]A \tag{2.7}$$

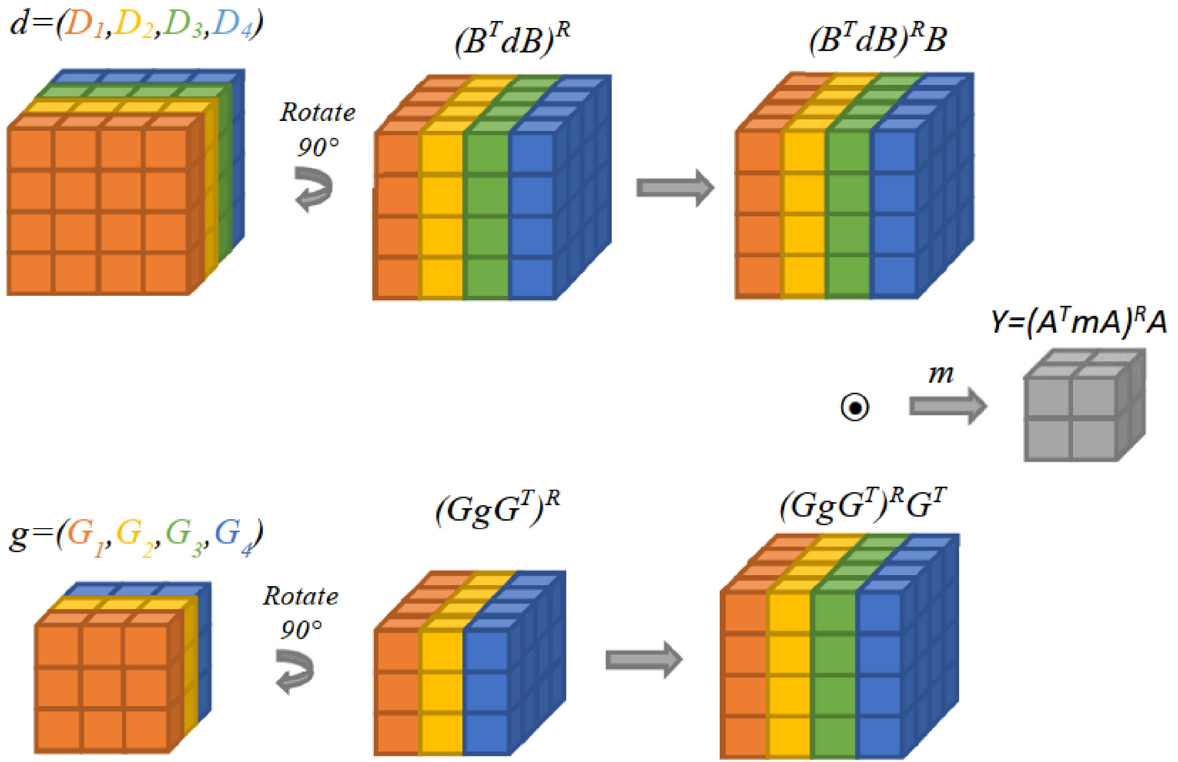


Figure 2.5: Process of the 3D Winograd algorithm

In [22], Shen et al. propose a method for the 3D Winograd algorithm F(m×m×m, r×r×r), which can be represented with equation (4):

$$Y = [A^T[(GgG^T)^R G^T \cdot (B^T dB)^R B]A]^R A \tag{2.8}$$

where R represents rotating the transformed image or filter tiles 90 degrees clockwise.

Figure 2.5 illustrates the process to obtain the convolution for F(2×2×2, 3×3×3) using the 3D Winograd algorithm. First, the 3D matrix should be split into channels. For each channel, a 2D Winograd transformation is applied to the kernel and the feature map. The new 3D matrices containing the results should be rotated 90 degrees clockwise. After this, an additional transformation using the transpose is required. At this point, the dot multiplications are performed. The results can be separated into channels. Utilizing these separate channels, the 2D transformation is applied using A and rotated 90 degrees clockwise. Then, the matrix

is multiplied using AT. Following these steps, the convolution result is achieved in a 2×2×2 matrix.

2.3.9 Winograd Works

Researchers have developed many approaches to reducing the computational cost for CNNs. In [11], an algorithm improvement is proposed through the analysis of the algebraic properties of CNNs. The algorithm achieves 47% reduction in computation without affecting the accuracy. In [21], the authors use Winograd's minimal filtering algorithms (WMFA) to develop new algorithms for stride one convolution. This algorithm is implemented for 2D on a GPU and lessens multiplications by a factor of 2.25, thus achieving better performance than the cuDNN library. In [23] and [24] the authors implement 2D WMFA on an FPGA. WMFA uses fewer multiplications and little extra memory. In [25], an FPGA implementation using OpenCL is presented. This implementation uses DSP in parallel to process 1D Winograd F(4, 3) while using the entire FPGA's processing capacity for a more efficient implementation. In [22], an architecture for accelerating 3D CNNs is presented, which is 13× faster than regular convolution. In [26], an instruction-driven CNN accelerator is proposed which supports the Winograd algorithm and cross-layer scheduling; their accelerator achieves 7× speedup compared to another cross-layer accelerator [27] on the same platform. In [23], a Winograd algorithm implementation is presented; their design uses a line buffer structure to reuse feature map data, and achieves 2940 GOPS for VGG16 on a Xilinx ZCU102 platform. The authors in [28] proposes an instruction-driven accelerator. However, the performance is limited for the large data transfer between on-chip and off-chip memory. In [29], a WMFA implementation

is proposed using an optimal algorithm to determine the fusion and algorithm strategy for each layer. The implementation achieves 660 GOPs of VGG-16.

2.4 Object Detection

Object detection, a topic within computer vision, encompasses aspects of image classification and localization [30]. Image classification determines what object classes are present within an image, and localization refers to finding those objects within the image (typically via the coordinates of its enclosing "bounding" box); this can be seen in Figure 2.6.

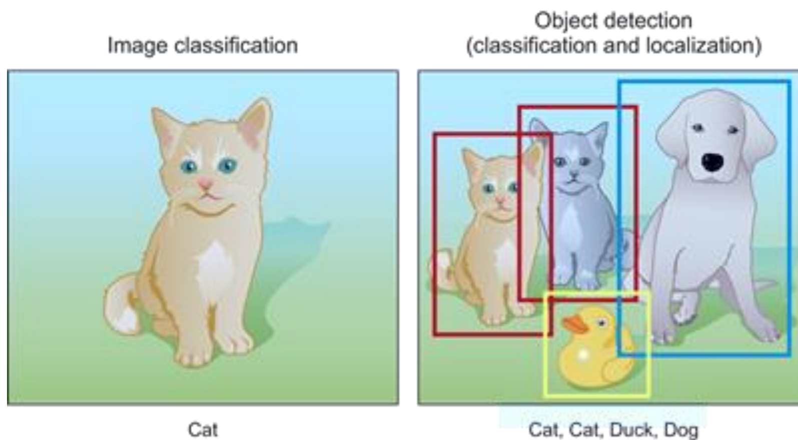


Figure 2.6: **(a)** Image Classification, **(b)** Object detection

Within the context of CNNs, convolution acts as a filter. The output of a CNN layer (the "activation map") represents the response of the filter (the "kernel")—indicating the location and strength of the feature, if detected—about the input. A CNN layer will utilize upwards hundreds of filters, thereby producing a large output activation map, where each filter represents an appropriate learned feature (e.g. a line, blob, or something hyperdimensional, e.g. for the purpose of discriminating different classes). During the learning process ("training"), a model is repeatedly exposed to a wealth of varied and representative data, and the internal model filter parameters are iteratively updated via small adjustments governed by a calculus-based algorithm called "backpropagation".

Table 2.1: Summary of object detection components

Heads	Backbones	Necks
R-CNN [40]	EfficientNet [32]	FPN [50]
Fast R-CNN [41]	VGG [7]	BiFPN [46]
Faster R-CNN [42]	ResNet [33]	NAS-FPN [51]
R-FCN [43]	DenseNet [34]	
Libra R-CNN [44]	SqueezeNet† [35]	
RepPoints [45]	MobileNet† [15], [16], [36]	
EfficientDet* [46]	ShuffleNet† [37]	
YOLO* [47]	MobileDet† [39]	
SSD* [79]		
RetinaNet* [49]		

* Refers to one-stage object detectors.
† Refers to backbones designed for implementation on CPU platforms and/or edge devices.

Modern object detectors are comprised of two parts, a “backbone” feature extractor that gives a feature map representation of the input, and a “head” used to predict classes and bounding boxes of found objects. Recent object detectors additionally insert layers between the backbone and the head to collect feature maps from different stages; such layers are called “necks”. Table 2.1 shows a summary of which models could be classified as which object detection network components.

Backbones are models, oftentimes pre-trained on ImageNet [31], designed for image classification and repurposed for object detection. Backbones like EfficientNet [32], VGG [7], ResNet [33], and DenseNet [34] are powerful feature extractors that usually require a GPU for reasonable processing time. For CPU platforms, SqueezeNet [35], MobileNet [15], [16], [36], or ShuffleNet [37] are more suitable. Neural architecture search (NAS) [38] has been used to build efficient backbones for object detection with high average precision (AP), e.g. for MobileDet [39].

The head can generally be categorized as either one- or two-stage. Two-stage object detectors have a ROI proposal step which finds possible places within an image where objects

are located. Popular two-stage object detectors include R-CNN [40], fast R-CNN [41], faster R-CNN [42], R-FCN [43], Libra R-CNN [44], and RepPoints [45]. One-stage object detectors require only a single pass through the network and do not use a separate ROI proposal step. Popular one-stage detectors include EfficientDet [46], YOLO [47], SSD [48], and RetinaNet [49].

Neck networks include the Feature Pyramid Network (FPN) [50], BiFPN [46], and NAS-FPN [51].

2.4.1 Multistage Object Detection Systems

In a multistage object detection pipeline system, an object's bounding box is localized, and its class identified, using different networks. König et al. [52] present a multi-stage reinforcement learning approach for detecting objects within an image. The authors' approach is comprised of a zoom stage and a refinement stage, uses aspect-ratio modifying actions, and is trained via a combination of three different reward metrics. Wang et al. [53] present a multi-stage 3D object detection network architecture that takes LIDAR points and images as inputs. They utilize a cascade-enhanced detector for small classes, a 3D region proposal subnet, and a second stage detector subnet to achieve high-precision oriented 3D bounding box prediction.

Yonetsu et al. [54] design a two-stage system for license plate detection in complex scenes using YOLOv2. The first stage detects cars and the second detects license plates within the cars' regions. Zhang et al. [55] use two-stage deep neural networks for license plate localization in unconstrained scenes. In the first stage, a CNN is used to extract local character features. In the second stage, a recurrent neural network (RNN) connects the fine-scale proposals to obtain the whole license plate. An improved faster R-CNN with a two-stage detection system for small object detection is presented by Cao et al. [56]. In the proposal stage, they achieve bounding box regression via an improved loss function based on Intersection over Union (IoU), and RoI pooling is enhanced via bilinear interpolation. They also use multi-scale convolution feature fusion so feature maps contain more information to detect small objects.

2.4.2 Video Object Detection Systems

Video object detection (VOD) detects objects within a video data stream rather than from a static image. Wang et al. [57] introduced an object detection system for compressed videos using a motion aided memory network (MMNet). Their system takes advantage of both motion vectors and residual errors in video streams, needing only to run a complete recognition network for I-frames. Their faster speed measured on a NVIDIA Titan X Pascal GPU is 55 FPS which is 3x times faster than single image R-FCN and 10x times faster than the high-performance detector MANet, with a minor accuracy loss. Deng et al. [57] introduced single shot video object detector (SSVD). The authors proposed enhancing per-frame features through aggregation of neighbouring frames using an FPN as a backbone network. SSVD estimates the motion and aggregates the nearby features along the motion path. SSVD achieves 79.2% mAP on ImageNet VID, and processes one frame in 85 ms on an Nvidia Titan X Pascal GPU. Given their success in natural language processing tasks, long short-term memory (LSTM) networks have been used in VOD systems by incorporating frame sequence information. Both [57] and [58] achieve high accuracy on ImageNet VID using an expensive GPU. However, their proposed layers are based on FPNs which are unsuitable for most edge devices in terms of computational complexity and processing time. For edge devices, Flow&LSTM [59] achieved the highest accuracy of 75.5%. Looking Fast and Slow [60] achieved relatively high speed (23.5 FPS) on a Pixel 3 phone, but had lower accuracy (58.9%).

Part II

Hardware Optimization for Convolutional Neural Networks

Part II of this thesis consists of hardware optimization for Convolutional Neural Networks (CNNs). Although a model can achieve the same accuracy result regardless of whether its underlying system is optimized, an optimized hardware system will generally use less resources which improves processing time, efficiency, and energy consumption. Processing an input through a CNN requires billions of multiplications; therefore, reducing the number of multiplications (ideally without affecting the CNN's accuracy) is important to improving the total efficiency/speed of the system.

Winograd minimal filtering algorithms (WMFAs) take advantage of overlapping computations between adjacent windows to reduce the number of multiplications required for convolution; WMFAs do so by trading multiplications for addition. Given that the hardware required for multiplication is complex and large compared to that of a simple adder, the multiplication–addition tradeoff proposed by Winograd is desirable. However, the original Winograd algorithm only applies when using a stride of 1, where stride is defined as the element-wise shift displacement of a kernel over an input along a particular axis. In object detection, a convolutional layer of stride 2 is particularly important because it allows the layer to downsample the input (for more efficient processing of latter model layers) while preserving learned spatial information (which is by definition important for object detection).

Part II of this thesis is comprised of Chapter 3, which proposes a novel Winograd algorithm for stride 2 convolutional. The formulated Winograd algorithms are compatible with stride 2 for 1-D(imension), 2-D, and 3-D convolutional layers. Furthermore, Chapter 3 presents the design of a novel processing element (PE) for an FPGA implementation. Compared to other Winograd implementations available in academic literature, the proposed PE unit supports both stride 1 and 2 convolutions. The proposed PE is implemented on an FPGA and benchmarked for DSP efficiency using the popular VGG-16 network architecture.

The PE and concomitant WMFA algorithms proposed in Part II may support the efficient implementation of models proposed in Part III of this thesis.

Chapter 3

Stride 2 Winograd for Convolutional Neural Networks[1]

This chapter present the motivations to design a Winograd stride 2 in Section 3.1. Section 3.2 presents novel Winograd stride 2 algorithms for 1-D, 2-D, and 3-D using 3×3, 5×5, and 7×7 kernels. Section 3.3 presents the CNN architectures that uses layer stride >1. The GPU implementation is shown in Section 3.4. Section 3.5 shows the design, implementation, and results of the CNN accelerator and compares the proposed method to prior works. Section 3.6 concludes the chapter.

3.1 Introduction

Convolutional Neural Networks (CNNs) are widely used in many deep learning systems. CNNs have shown state-of-the-art accuracy in a variety of interdisciplinary research, including image classification [61]–[63], object detection [64]–[66], and speech recognition [67], [68], leading to their widespread adoption.

To reduce memory bandwidth requirements, recent research into CNN hardware acceleration has focused on increasing parallelism, reducing bits via quantization, or using fixed points rather than floating points [69], [70].

The Winograd minimal filtering algorithms, capable of being used for any stride [71], take advantage of overlapping computations between adjacent windows [21] to reduce

[1]The content of this chapter is originally published in IEEE Transactions on Very Large Scale Integration (VLSI) Systems [113]. The manuscript has been reformatted for inclusion in this thesis.

Juan Yépez (JY), and Seok-Bum Ko (SK) designed the study. JY designed the algorithms, developed and optimized the processing elements, implemented the algorithms on a GPU and a FPGA, and performed logic synthesis and results analysis. JY prepared the manuscript with contributions from SK to the manuscript structure, readability and analysis and discussion of the results.

the number of multiplications required for convolution, trading multiplication for addition. Given that the hardware required for multiplication is complex and large compared to that of a simple adder, the multiplication-addition tradeoff proposed by Winograd is desirable.

Modern CNN architectures replace pooling layers with strided convolutions for downsampling [72], where stride is defined as the element-wise shift-displacement of a kernel over an input along a particular axis [73]. Convolutional layers learn feature properties during training; conversely, pooling is a fixed downsampling operation and pooling layers have no trainable weights. A convolutional layer with stride >1 is advantageous in that it has trainable parameters and downsamples.

Recent architectures (e.g. the MobileNet family) use increasingly more layers with stride >1. Therefore, it is important to develop methods to process these layers efficiently. In this chapter, a novel way to use Winograd stride one algorithms to produce the effect of stride two is introduced. Algorithms for 3×3 and 5×5 kernels for 1D and 2D cases are proposed. These algorithms require special cases of the Winograd algorithm that are described in this chapter. Therefore, the main contributions of this chapter are as follows:

• Novel Winograd algorithms with stride two for 1D, 2D, and 3D convolutions. These algorithms reduce the multiplication complexity of convolution. A quantitative analysis of the number of multiplications and additions required by this algorithm was provided.

• The matrices for 1D Winograd F(2,2) and F(2,4), for 2D Winograd F(2×2,2×3), F(2×2,3×2), F(2×2,3×4), and F(2×2,4×3) were determined. These versions are required to solve the proposed Winograd with stride two. To obtain the values of these respective matrices, the popular Winograd F(2,3) is referenced and the Chinese Remainder Theorem [74] is used. A quantitative analysis of these matrices is performed, comparing them with regular convolutions.

• The proposed algorithm was implemented on an NVIDIA K20c GPU. It shows 1.44x improvement for a 3×3 kernel, 2.04x improvement for a 5×5 kernel, 2.42x improvement for a 7×7 kernel, and 1.73x improvement for a 3×3×3 kernel.

• A CNN accelerator was implemented on an Intel Arria-10 FPGA for both an original and modified VGG-16 architecture, which achieved Digital Signal Processor (DSP) efficiencies of 1.22 GOPS/DSPs and 1.33 GOPS/DSPs, respectively. For the accelerator, a Winograd Processing Element (PE) for a 3×3 kernel was proposed, compatible for both stride one and two convolutions. The combined one- and two-stride Winograd PE uses only 146 and 100 more LUTs and registers, respectively, and the same number of DSPs (32), as a PE for two Winograd stride one calculations. It also uses 25 less DSPs than would be required by independent PEs for calculating two Winograd stride one, and one Winograd stride two, operations.

3.2 Proposed Winograd with Stride 2

3.2.1 One-dimension

In one-dimensional convolution with stride two, odd positions of the input are multiplied with odd positions of the kernel, and even-position input elements are multiplied with even-position kernel elements; no multiplication between odd-position and even-position elements occurs. Thus, the input and kernel elements can be separated into two groups: odd and even. Using these groupings, it is possible to convert a convolution of stride two into two convolutions of stride one. Figure 3.1 shows the procedure for a size five input vector and a size three kernel vector.

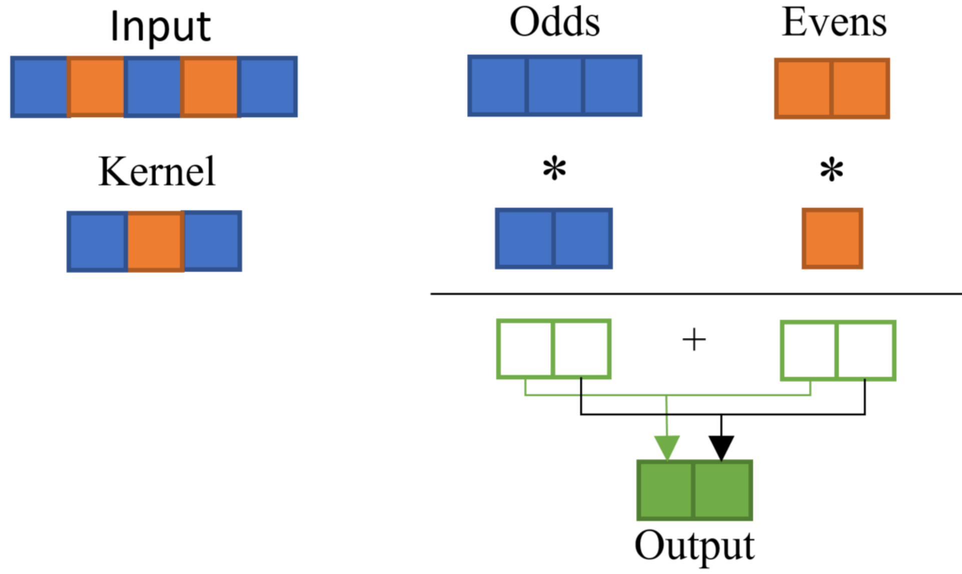


Figure 3.1: Proposed convolution with stride=2 for kernel=3

The even group contains two elements for the input and one element for the kernel. This cannot be further simplified. However, the odd group contains three elements for the input and two elements for the kernel, which can be calculated using the F(2,2) Winograd.

There is no available F(2,2) Winograd based on matrices. However, F(2,2) can be determined using the popular F(2,3) Winograd. In these matrices, the values of the variables g2 and d3 are zero when using F(2,2) because the fourth row for both G and BT are zero and can be deleted. Similarly, the fourth column of BT and AT and the third column of G can also be eliminated. Therefore, the matrices for F(2,2) are as follows:

$$g = [g_0 \quad g_1]^T \qquad d = [d_0 \quad d_1 \quad d_2]^T \tag{3.1}$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \\ 0 & -1 & 1 \end{bmatrix} \; G = \begin{bmatrix} 1 & 0 \\ 1/2 & 1/2 \\ 1/2 & -1/2 \end{bmatrix} \; A^T = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \end{bmatrix}$$

The minimal filtering algorithm for computing m outputs with an r-tap kernel, which is called F(m, r), requires a number of multiplications defined by:

$$\mu(F(m, r)) = m + r - 1 \tag{3.2}$$

The F(2,2) algorithm uses just three multiplications and is therefore minimal by the formula $\mu(F(2, 2)) = 2 + 2 - 1 = 3$. The algorithm also uses three additions involving the data, two additions and two multiplications by a constant involving the kernel, and three additions to reduce the products in the final result, whereas six is required for regular convolution.

To use a size five kernel with stride two, the procedure is similar to using a size three kernel: there is no multiplication between odd-position and even-position elements, and elements are separated into respective odd- and even-position groups. For a size five kernel, the input vector contains seven elements. The odd group contains four input elements and three kernel elements, allowing the Winograd F(2,3) to be applied. The even group contains three input and two kernel elements, and Winograd F(2,2) is used. Figure 3.2 shows the procedure for a size seven input vector and a size five kernel vector.

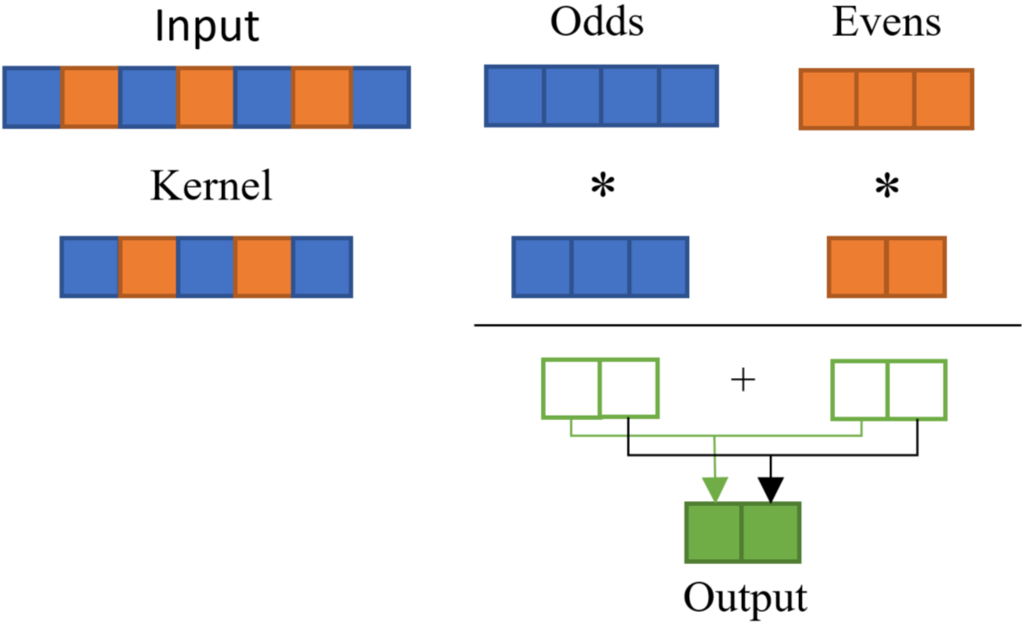


Figure 3.2: Proposed convolution with stride=2 for kernel=5

The odd-position group uses four multiplications ($\mu(F(2,3))=4$), and the even-position group uses three multiplications ($\mu(F(2,2))=3$). Therefore, the total number of multiplications is seven, whereas ten would be required for regular convolution.

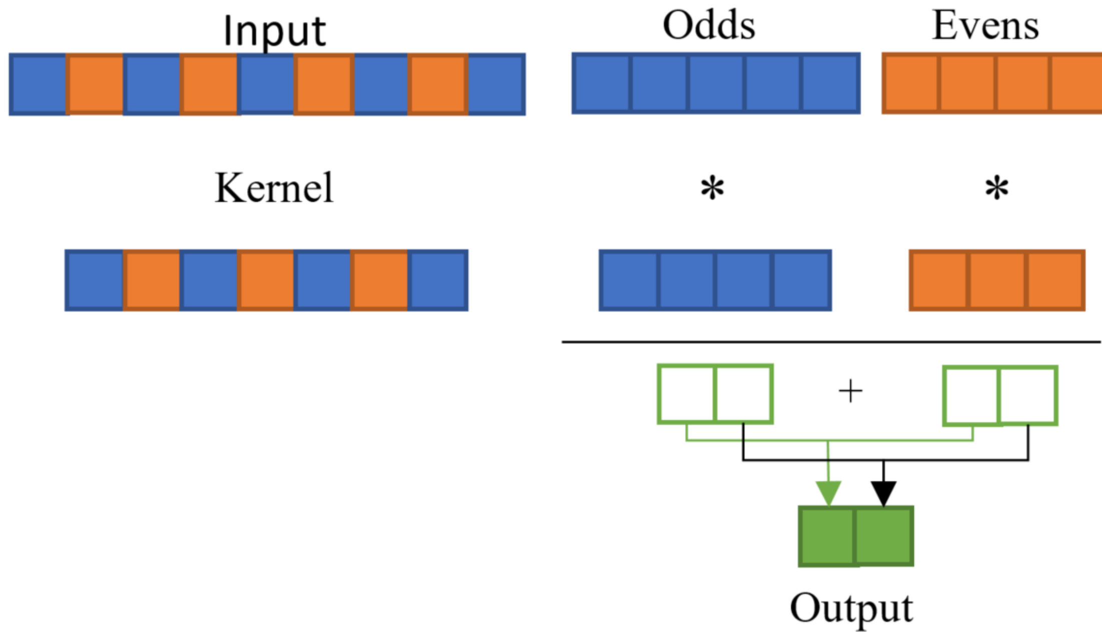


Figure 3.3: Proposed convolution with stride=2 for kernel=7

For a size seven kernel, the even group contains four input and three kernel elements, and the popular Winograd F(2,3) will be used. The odd group contains five input elements and four kernel elements, allowing the Winograd F(2,4) to be applied. Figure 3.3 shows the procedure for a size nine input vector and a size seven kernel vector.

Because there is no available F(2,4) Winograd based on matrices, F(2,4) was derived using the same technique provide by Winograd.

The four-element kernel g and five-element data d are represented as polynomials in the following:

$$g(x) = g_3x^3 + g_2x^2 + g_1x + g_0 \tag{3.3}$$

$$d(x) = d_4x^4 + d_3x^3 + d_2x^2 + d_1x + d_0$$

The lineal convolution $g * d$ is:

$$y(x) = g(x)d(x) \tag{3.4}$$

The polynomial m(x) can be written in terms of

$$y(x) = g(x)d(x) \bmod \mathrm{m(x)} \tag{3.5}$$

Applying the Winograd analysis, the equations can be represented as matrices, as shown equation 6:

$$g = [g_0 \quad g_1 \quad g_2 \quad g_3]^T$$

$$d = [d_0 \quad d_1 \quad d_2 \quad d_3 \quad d_4]^T \tag{3.6}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & 1 \end{bmatrix}$$

$$B^T = \begin{bmatrix} 2 & -1 & -2 & 1 & 0 \\ 0 & -2 & -1 & 1 & 0 \\ 0 & 2 & -3 & 1 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1/2 & 0 & 0 & 0 \\ -1/2 & -1/2 & -1/2 & -1/2 \\ -1/6 & 1/6 & -1/6 & 1/6 \\ 1/6 & 1/3 & 2/3 & 4/3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The odd-position group uses five multiplications ($\mu(F(2,4))=5$), and the even-position group uses four multiplications ($\mu(F(2,3))=4$). Therefore, the total number of multiplications is nine, whereas 14 would be required for regular convolution.

The practical implementation for the one-dimensional stride two Winograd is shown in Algorithm 1.

Algorithm 1 Winograd-Stride2(*input, kernel*)

```
W=input.width()
for i=0 to W do
  if (i % 2 == 1) then
    G1=input(i/2)              Odds Group.
     if (j≤ kernel.width()) then
        K1=kernel(i/2)
  else
    G2=input((i-1)/2)          Evens Group.
     if (j≤ kernel.width()) then
        K2=kernel((i-1)/2)
  output=Winograd(G1,K1)+ Winograd(G2,K2)
```

3.2.2 Two-dimensions

3.2.2.1 Using Kernel 3×3

To apply the Winograd with stride two in two dimensions, the elements of the input and the kernel should be separated into four groups. The first group comprises the elements at row-odd, column-odd indices of the input and kernel; the second group comprises the elements at row-odd, column-even indices of the input and kernel; the third group comprises the elements at row-even, column-odd indices of the input and kernel; the last group comprises the remaining four elements of the input and one element of the kernel. Figure 3.4 shows the groups with a 3×3 kernel.

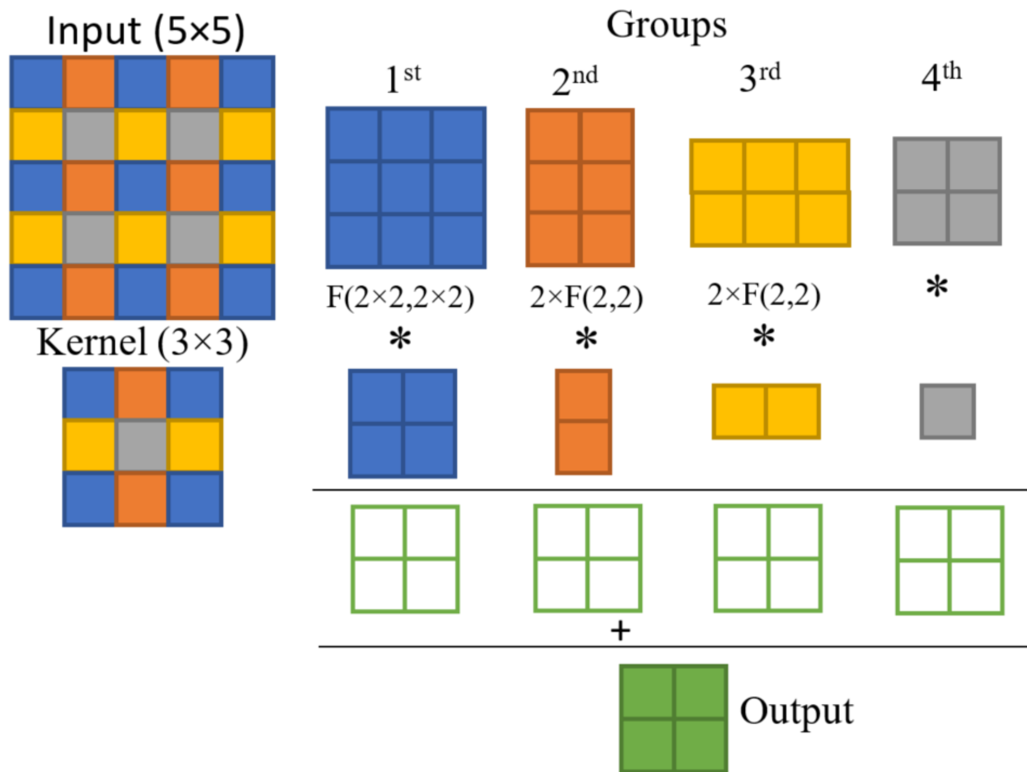


Figure 3.4: Proposed convolution with stride=2 for kernel=3×3

The first group contains 3×3 elements for input and 2×2 elements for the kernel. In this case, the Winograd F(2×2,2×2) can be used; the matrices for 2D are the same as those for one-dimensional F(2,2).

Two consecutive Winograd F(2,2) operations can be used for the second and the third groups.

The fourth group cannot be further simplified, so regular multiplications are used. Each group's convolution produces a 2×2 matrix, and the final value of this process is simply the sum of these intermediate matrices.

The number of multiplications per-group are as follows: nine multiplications for the first group; six multiplications for the second; six multiplications for the third; and four multiplications for the last. The total multiplications used is 25, whereas 36 is used with regular convolution.

3.2.2.2 Using Kernel 5×5

For two dimensions with a 5×5 kernel, like the 3×3 kernel case, the elements of the (presumably 7×7) input and the kernel can be separated into four groups: the elements of the row-odd, column-odd indices of the input and kernel; the elements of the row-odd, column-even indices; the elements of the row-even, column-odd indices; and the remainder. Figure 3.5 shows the groups with a 5×5 kernel.

The first group contains 4×4 elements for input and 3×3 elements for the kernel, which can be implemented using the popular Winograd F(2×2, 3×3).

The second group has 4×3 input elements and 3×2 kernel elements. This special case can be implemented using a combination of the matrices of the different methods of Winograd. The matrices of F(2,3) are used for $A^T, G,$ and B^T, and the matrices of F(2,2) are used for $G^T, B,$ and A.

The third group has 3×4 input elements and 2×3 kernel elements. This is a swapped version of the second group's case, and matrices of F(2,2) are used for $A^T, G,$ and B^T, and the matrices of F(2,3) are used for $G^T, B,$ and A.

The fourth group contains 3×3 input elements with a 2×2 kernel, and uses Winograd F(2×2,2×2).

The result of the convolution of the input 7×7 with kernel 5×5 will be the sum of the intermediate 2×2 matrices produced via the respective convolutions of each group.

The numbers of multiplications per group using this algorithm are as follows: 16 for the first group; 12 in each of the second and third; and nine multiplications for the last. The total is 49 multiplications, whereas 100 would be used by regular convolution.

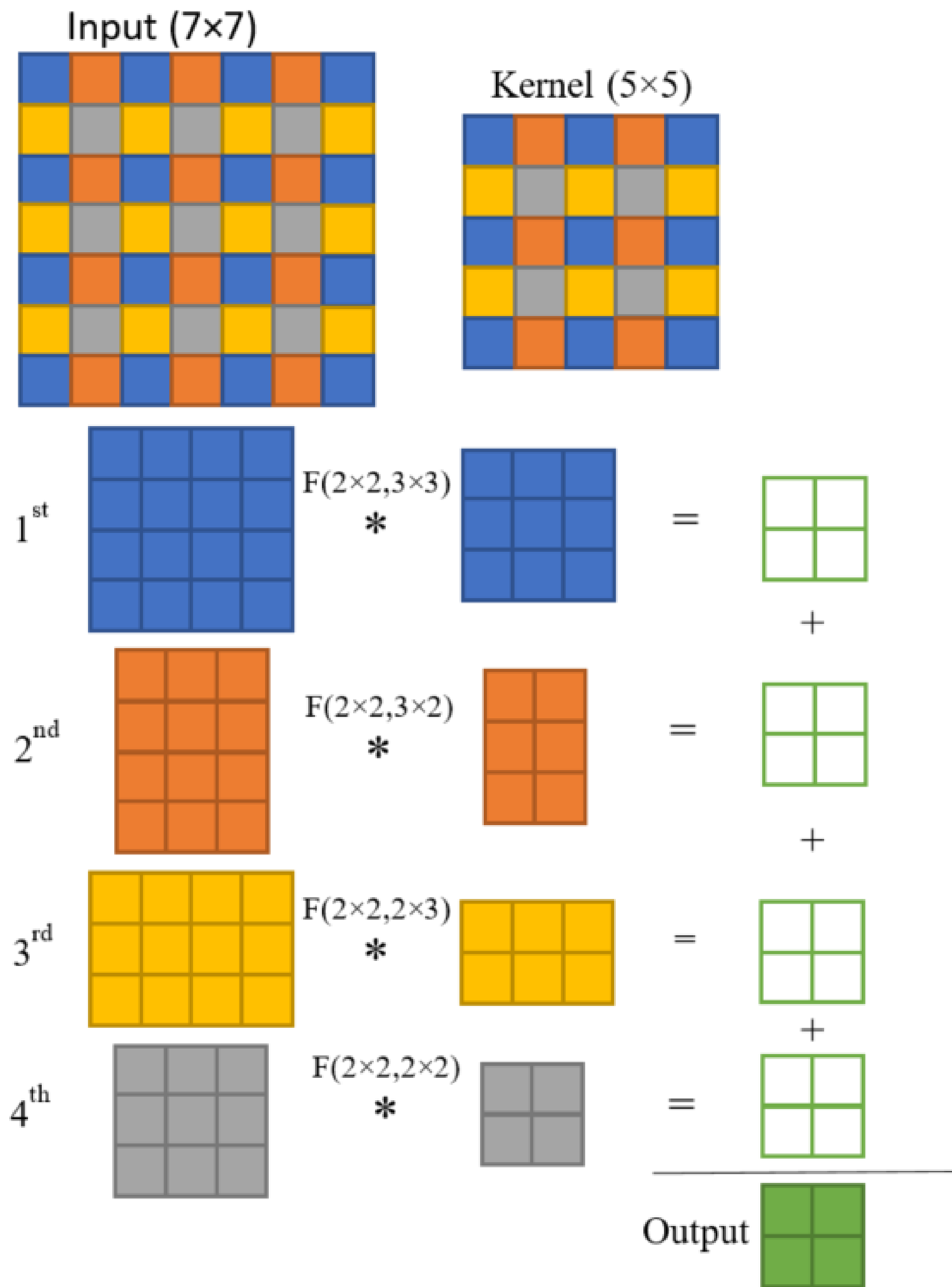


Figure 3.5: Proposed convolution with stride=2 for kernel=5×5

3.2.2.3 Using Kernel 7×7

There are popular architectures where the size 7×7 kernel with stride two is use at the first layer. Commonly a kernel size seven could be operate with FFT. However, the stride two make the invalid the FFT operation. Due the procedure of separated the elements into odd- and even-position reduce the size of kernel for the operation. It makes possible the use of Winograd in an efficient matter.

For two dimensions with a 7×7 kernel, like the case of previous kernels, the input and the kernel can be separated into four groups: the elements of the row-odd, column-odd indices of the input and kernel; the elements of the row-odd, column-even indices; the elements of the row-even, column-odd indices; and the remainder. Figure 3.6 shows the groups with a 7×7 kernel.

The first group contains 5×5 elements for input and 4×4 elements for the kernel, which can be implemented using the proposed method F(2,4) which was described in the last part of section 3.2.

The second group has 5×4 input elements and 4×3 kernel elements. This special case can be implemented using a combination of the matrices of the different methods of Winograd. The matrices of F(2,4) are used for A^T, G, and B^T, and the matrices of F(2,3) are used for G^T, B, and A.

The third group has 4×5 input elements and 3×4 kernel elements. This is a swapped version of the second group's case. The matrices of F(2,3) are used for A^T, G, and B^T, and the matrices of F(2,4) are used for G^T, B, and A.

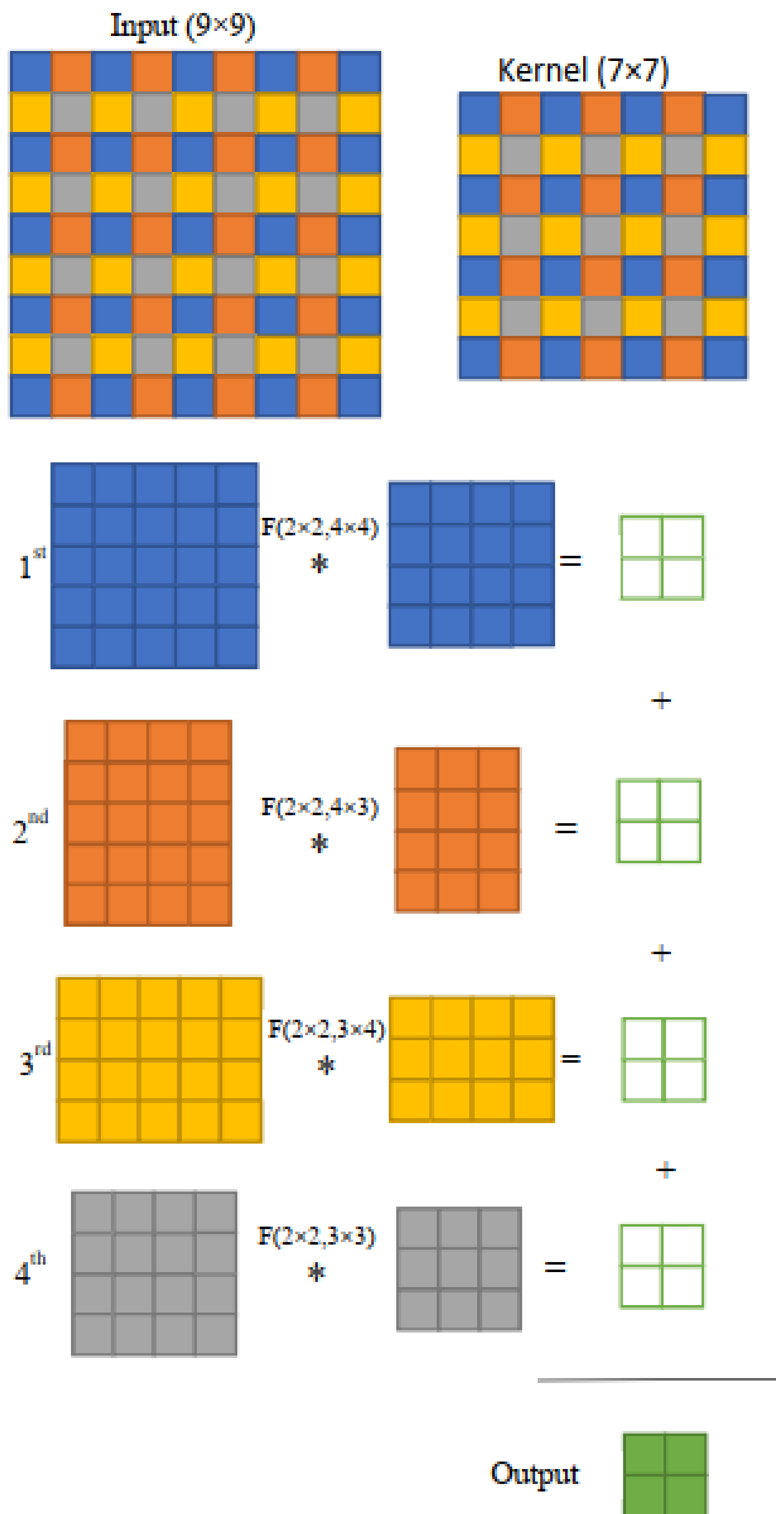


Figure 3.6: Proposed convolution with stride=2 for kernel=7×7

The fourth group contains 4×4 elements for input and 3×3 elements for the kernel, which can be implemented using the popular Winograd F(2×2, 3×3).

The result of the convolution of the input 9×9 with kernel 7×7 is the sum of the intermediate 2×2 matrices produced via the respective convolutions of each group. The number of multiplications per group using this algorithm are as follows: 25 for the first group; 20 in each of the second and third; and 16 multiplications for the last. The total is 81 multiplications, whereas 196 would be used by regular convolution.

The practical implementation for two-dimensional stride two Winograd is shown in Algorithm 2.

Algorithm 2 Winograd2D-Stride2(*input, kernel*)

```
W=input.width()
H=input.height()
for i=0 to W do
  for j=0 to H do
    if (i % 2 == 1) then
      if (j % 2 == 1) then
        G1=input(i/2, j/2)              Group 1
        if (j≤ kernel.width()) then
          K1=kernel(i/2, j/2)
      else
        G2=input(i/2, (j-1)/2)          Group 2
        if (j≤ kernel.width()) then
          K2=kernel(i/2, (j-1)/2)
    else
      if (j % 2 == 1) then
        G3=input((i-1)/2, j/2)          Group 3
        if (j≤ kernel.width()) then
          K3=kernel((i-1)/2, j/2)
      else
        G4=input((i-1)/2, (j-1)/2)      Group 4
```

```
if (j≤ kernel.width()) then
    K4=kernel((i-1)/2, (j-1)/2)
output=Winograd(G1,K1)+Winograd(G2,K2)+
    Winograd(G3,K3)+Winograd(G4,K4)
```

3.2.3 Three-dimensions

To apply Winograd with stride two in three dimensions using a 3×3×3 kernel, the elements of the input and the kernel should be in eight groups. The first group comprises the elements at row-odd, column-odd, channel-odd of the input and kernel; the second group comprises the elements at row-odd, column-even, channel-odd indices of the input and kernel; the third group comprises the elements at row-even, column-odd, channel-odd indices of the input and kernel; the fourth group comprises the elements at row-odd, column-odd, channel-even indices of the input and kernel; the fifth group comprises the elements at row-even, column-even, channel-odd indices of the input and kernel; the sixth group comprises the elements at row-odd, column-even, channel-even indices of the input and kernel; the seventh group comprises the elements at row-even, column-odd, channel-even indices of the input and kernel form the seventh group; the last group comprises the remaining four elements of the input and one element of the kernel. Figure 3.7 shows the groups with a 3×3×3 kernel.

The first group contains 3×3×3 elements for input and 2×2×2 elements for the kernel. In this case, the Winograd F(2×2×2,2×2×2) can be used; the matrices for 3D will be the same as those for one-dimensional F(2,2).

Two consecutive 2D Winograd F(2×2,2×2) operations can be used for the second, third, and fourth groups.

Four consecutive 1D Winograd F(2,2) operations can be used for the fifth, the sixth, and the seventh groups.

The eighth group cannot be further simplified, so regular multiplications are used. Each group's convolution produces a 2×2×2 matrix, and the final value of this process is simply the sum of these intermediate matrices.

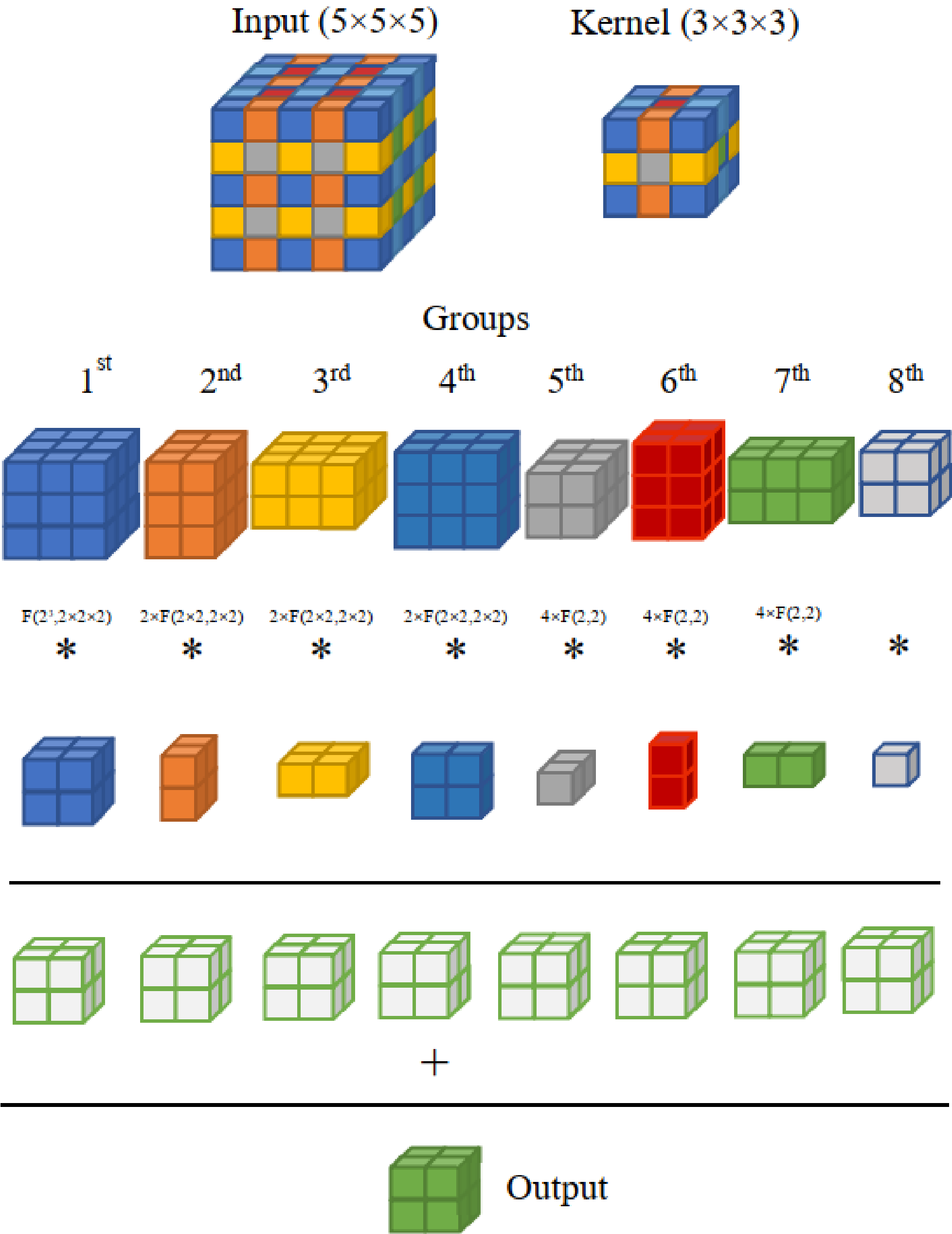


Figure 3.7: Proposed convolution with stride=2 for kernel=3×3×3

The number of multiplications per-group are as follows: 27 multiplications for the first group; 18 multiplications in each of the second, third, and fourth; 12 multiplications in each of the fifth, sixth, and seventh; and 8 multiplications for the last. The total multiplications are 125, whereas 216 would be used with the regular convolution.

The practical implementation for three-dimensional stride two Winograd is shown in Algorithm 3.

Because 5×5 and 7×7 kernels are commonly used only for the input layer (assuming an RGB color image), it is impractical to extend the stride two methodology to 3D kernels other than 3×3×3; therefore a 5×5×3 or 7×7×3 kernel was not implemented.

Algorithm 3 Winograd3D-Stride2(*input, kernel*)

```
W=input.width()
H=input.height()
D=input.depth()
for i=0 to W do
  for j=0 to H do
    for k=0 to D do
      if (i % 2 == 1) then
        if (j % 2 == 1) then
          if (k % 2 == 1) then
            G1=input(i/2, j/2, k/2)              Group 1
            if (k≤ kernel.width()) then
              K1=kernel(i/2, j/2, k/2)
          else
            G2=input(i/2, j/2, (j-1)/2)          Group 2
            if (k≤ kernel.width()) then
              K2=kernel(i/2, (j-1)/2)
        else
          if (k % 2 == 1) then
            G3=input(i/2, (j-1)/2, k/2)          Group 3
```

```
            if (k≤ kernel.width()) then
               K3=kernel(i/2, (j-1)/2, k/2)
         else
            G4=input(i/2, (j-1)/2, (k-1)/2)      Group 4
            if (k≤ kernel.width()) then
               K4=kernel(i/2, (j-1)/2, (k-1)/2)
     else
       if (j % 2 == 1) then
         if (k % 2 == 1) then
            G5=input((i-1)/2, j/2, k/2)          Group 5
            if (k≤ kernel.width()) then
               K5=kernel((i-1)/2, j/2, k/2)
         else
            G6=input((i-1)/2, j/2, (j-1)/2)      Group 6
            if (k≤ kernel.width()) then
               K6=kernel(i/2, (j-1)/2)
       else
         if (k % 2 == 1) then
            G7=input((i-1)/2, (j-1)/2, k/2)      Group 7
            if (k≤ kernel.width()) then
               K7=kernel((i-1)/2, (j-1)/2, k/2)
         else
            G8=input((i-1)/2, (j-1)/2, (k-1)/2) Group 8
            if (k≤ kernel.width()) then
               K8=kernel((i-1)/2, (j-1)/2, (k-1)/2)
 output=Winograd(G1,K1)+Winograd(G2,K2)+
Winograd(G3,K3)+Winograd(G4,K4)+ Winograd(G5,K5)+
Winograd(G6,K6)+Winograd(G7,K7)+ Winograd(G8,K8)
```

3.3 CNN Architectures with Layer Stride > 1

Modern CNN architectures use convolutional layers for feature learning followed by max-pooling layers to downsample feature maps. However, max-pooling is a fixed operation that trades spatial structure knowledge for improved computational efficiency in future layers. Spatial knowledge of where regions of interest are in an image is not important for classification tasks because the goal of classification is recognizing whether a certain region of interest (or features thereof) merely exists in an image. Conversely, spatial knowledge is foundational to the task of object detection, where the positional information of regions of interest is paramount. For such tasks where spatial information is important, the use of strided convolution has found success because it improves the computational efficiency of future layers while preserving spatial information. In [72], the pooling layers are replaced by an additional convolutional layer with stride two; results show performance stabilization and even accuracy improvement compared to the base model. This concept has led to the adoption of >1 stride convolutional layers in modern CNN architectures, and the use of stride two is a popular choice. Table 3.1 highlights the stride >1 layers in recent architectures and their utilized kernel sizes.

AlexNet [75], the first CNN to use ReLU non-linearity and whose win in the ILSVRC 2012 image recognition challenge, sparked the current DL revolution. It uses an 11×11 kernel with stride four. However, with large kernels like 11×11, the Winograd algorithm is outperformed by other methods (e.g., FB-FFT) [21].

ZFNet [12], GoogLeNet [76], ResNet [12], SqueezeNet [77], and YOLO [47] use a 7×7 kernel with stride two in the first layer, allowing 1D and 2D Winograd to be used. 3D Winograd for a 7×7×7 kernel is unavailable for the first layer due this have only three input channels. However, the 3D Winograd can be used for later layers when the input depth is larger than 7.

ResNet [78], YOLO [47], SSD [79], MobileNet [16], and MobileNetV2 [80] use a 3×3 kernel. MobileNetV3 [81] uses three and two layers with 3×3 and 5×5 kernel,

respectively. The convolutions with stride two can be applied using the proposed Winograd algorithms.

Table 3.1: Architecture layers using stride >1

Architecture	Convolutional Layer with stride >1		
	Kernels	**Stride**	**Number of layers**
AlexNet [75]	11×11	4	1
ZFNet [78]	7×7	2	1
GoogLeNet [76]	7×7	2	1
ResNet [12]	7×7, 3×3	2	1, 3
SqueezeNet [77]	7×7	2	1
YOLO [47]	7×7, 3×3	2	1, 1
SSD [79]	3×3	2	2
MobileNets [16]	3×3	2	6
MobileNetV2 [80]	3×3	2	5
MobileNetV3 [81]	5×5, 3×3	2	2, 3

3.4 GPU Implementation

The proposed method for 1D and 2D stride two Winograd were tested on a NVIDIA K20c GPU, and tested it using several convolutions with 3×3, 5×5, and 7x7 kernel sizes. For 3D, a 3×3×3 kernel was used. The proposed implementation was compared against regular stride two convolutions. A comparison of the numbers of multiplications and additions for all applicable dimensions is shown in Table 3.2.

Table 3.2: Comparisons of the Regular convolution and Winograd Stride Two

Algorithms	Regular Convolution		Proposed Winograd stride 2	
	muls	adds	muls	adds
F(2,3)	6	6	5	11
F(2,5)	10	10	7	21
F(2,7)	14	14	9	27
F(2×2, 3×3)	36	36	25	77
F(2×2, 5×5)	100	100	49	137
F(2×2, 7×7)	196	196	81	243
F(2×2×2, 3×3×3)	216	216	125	419

The implemented program uses the same settings from the stride two layers of MobileNet (an input size of 224×224). The program recorded the GPU processing times and generated the average processing times for regular convolution and for the proposed algorithms. The results are shown in Table 3.3.

Table 3.3: Performance Comparison using stride two in GPU

Kernel	Processing time		
	Regular convolution	Proposed Winograd Stride 2	Speedup
3×3	8.09ms	5.61ms	1.44x
5×5	11.21ms	5.49ms	2.04x
7×7	13.21ms	5.46ms	2.42x
3×3×3	15.45ms	8.93ms	1.73x

Compared to regular convolution, the proposed method is 1.44x, 2.04x, 2.42x, and 1.73x faster for the respective 3×3, 5×5, 7×7, and 3×3×3 kernels. Furthermore, the results

of the proposed algorithms were tested with single precision (fp32) data. The results show the same values as direct convolution, indicating that the mathematical transformations do not lose precision and therefore will affect neither multiplication accuracy nor neural network performance.

3.5 FPGA Implementation

3.5.1 CNN Architecture

In this chapter, a CNN accelerator design was proposed based on stride one and two Winograd for FPGAs. The accelerator implements the VGG-16 architecture [7]. Because VGG-16 does not contain any layers of stride two, a modified VGG-16 architecture was proposed using a similar methodology to that implemented by [72] for a custom network, in which one convolution and one max pooling layer are replaced by one convolutional layer of stride two. The modified architecture can be seen in Table 3.4.

Table 3.4: Modified VGG-16 Architecture with convolution stride two

Layer	Type/Stride	Filter Shape	Input size
Conv1-1	Conv/S1	3×3×64	224×224×3
Conv1-2	Conv/S2	3×3×64	224×224×64
Conv2-1	Conv/S1	3×3×128	112×112×64
Conv2-2	Conv/S2	3×3×128	112×112×128
Conv3-1	Conv/S1	3×3×256	56×56×128
Conv3-2	Conv/S1	3×3×256	56×56×256
Conv3-3	Conv/S2	3×3×512	56×56×256
Conv4-1	Conv/S1	3×3×512	28×28×512
Conv4-2	Conv/S1	3×3×512	28×28×512
Conv4-3	Conv/S2	3×3×512	28×28×512
Conv5-1	Conv/S1	3×3×512	14×14×512
Conv5-2	Conv/S1	3×3×512	14×14×512
Conv5-3	Conv/S2	3×3×512	14×14×512
Flatten	Flatten	-	7×7×512
FC2	Dense	4096	1×25088
FC2	Dense	4096	1×4096
Predictions	Dense	1000	1×4096

The original VGG-16 model was trained on the ImageNet database [31], which contains 1M images over 1000 classes; this allows the network to learn varied and representative features for many types of images. Moreover, building and training a model without pre-initialized weights is not always feasible due to the inherent time or computational restraints of training.

Therefore, the weights for the ImageNet pre-trained VGG-16 model were used, which are available online [82], for transfer learning—where the trained model weights are reused to initialize training a model for a different task. The weights were reused of the original VGG-16 to train the modified architecture using stride two, which has the same number of parameters as the original VGG-16.

Non-modified layers preserve the (frozen) weights from the original model. Only the added convolution stride two layers were trained, or 5,494,208 of 138,357,544 total parameters; thus, the model's training time is reduced. After only two hundred epochs, the proposed modified VGG-16 architecture has less computational cost than the original VGG-16; this is because stride 2 convolution requires less multiplications than stride 1, and a stride 1 convolution requires an additional pooling layer for downsampling. For example, the output of layer Conv5-3 from the modified VGG-16 has a size of $7 \times 7 \times 512$, while that from the original VGG-16 is $14 \times 14 \times 512$; a max-pooling layer of stride 2 is required for the layer shape to become $7 \times 7 \times 512$ in the stride 1 case. Thus, when compared to the original architecture, four times fewer multiplications are required for the convolutional layers in the proposed stride 2-modified VGG-16 architecture, which improves the throughput significantly whilst maintaining high accuracy.

3.5.2 FPGA Implementation

Both the original and the modified VGG-16 architectures were accelerated on an Intel Arria-10 FPGA with 1150K logic elements, 1518 DSP blocks, and 2131 M20K RAMs, at a clock speed of 250 MHz. A 16-bit fixed point precision to evaluate the proposed

design was used. Figure 3.8 shows the architecture which contains a host interface, DDR memory, on-chip buffer, sequencer, and processing elements (PE).

The host interface receives instructions and input data from and sends results to a host. The sequencer decodes instructions from which the appropriate components receive operating commands. Host data and PE processing results are saved in DDR memory. An on-chip buffer stores the data to be processed in the current operation. All computations are performed within PEs.

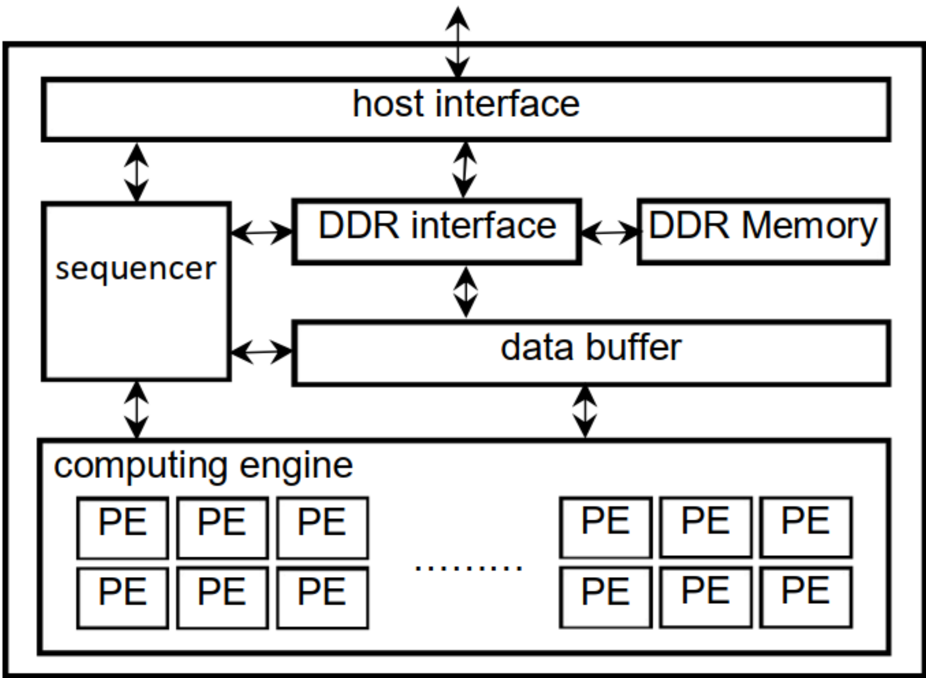


Figure 3.8: Deep learning architecture

3.5.3 Memory Access

Due to the limited on-chip buffer and the large amount of data required by CNNs, the accelerator is not able to transfer all the data from external memory into on-chip buffer. That is why the input and the kernel data are divided into several groups, allowing for data reuse in both the input and kernel. Multiple kernel groups are stored in the buffer and processed in parallel with the same input group. The partial results from the PEs are accumulated into the output buffer until the convolutional result is generated. Then, the

results are moved to external memory. After this process, a new input group is loaded and the same kernels can be reused. In the proposed design, double buffers are used to overlap data communication and reduce memory access delay in reading and writing data to memory.

3.5.4 Proposed PE Architecture

A PE was designed to accommodate two Winograd F(2×2, 3×3) stride one operations. As each Winograd uses 16 multiplications, 32 multipliers are required for each PE. Some logic elements are added to the PE to make it compatible with the proposed Winograd stride two, without using additional DSPs. Because VGG-16 uses the same filter size (3×3) for all convolutional layers, the PEs can be reused.

For headings 1 through 8 below, please refer to Section 3.2.2 and Figure 3.9 for information on Winograd stride 2 groups.

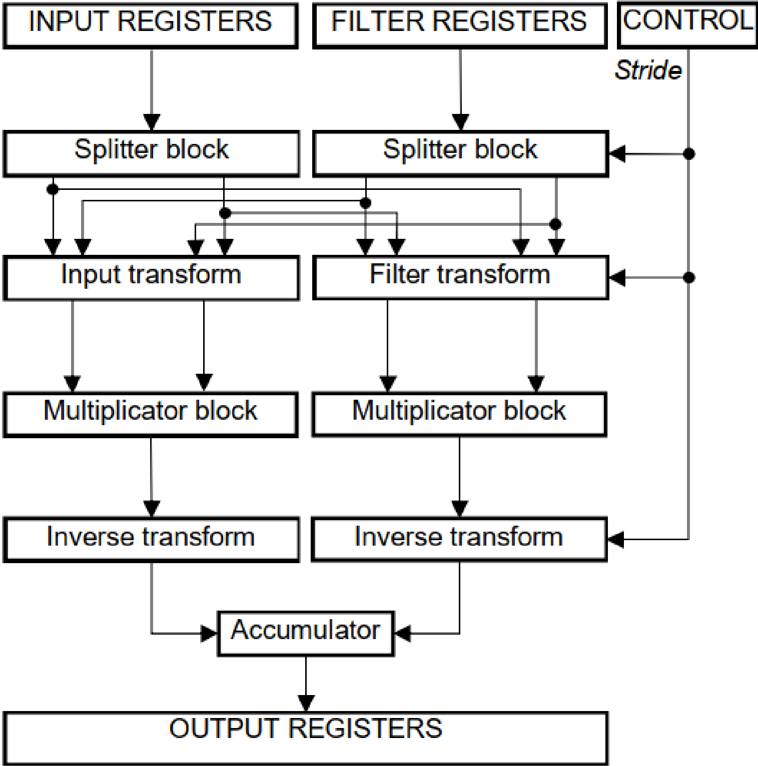


Figure 3.9: Proposed PE architecture

3.5.4.1 Input Tile From Registers

The input tile size of the proposed design is 5×6, where the PE may process two Winograd stride one or one Winograd stride two in one clock cycle.

For Winograd stride one, the PE allows two Winograd stride one in parallel, 32 multiplications are performed, and 32 multipliers are needed.

The proposed PE allows one Winograd stride two, a process requiring 25 multiplications. If the PE used regular convolution stride two instead of the proposed Winograd, it would require 36 multipliers.

3.5.4.2 Splitter Block

The control signal "stride" splits the input data based on the type of stride to process (one or two), i.e., how the input is used depends on the stride (as shown in Figure 3.9).

For stride one, the block splits a 4×6 array from the input tile into two 4×4 arrays which are sent to the Transform block; this is possible because two columns have the same input for both Winograd modules.

For stride two, a 5×5 input array is used, and the splitters split the input into four groups. Group 1 is sent to the input transform block and Groups 2-4 are rearranged to generate another input for the input transform block.

Figure 3.10 shows how the splitter block works. The 5×6 array (black) represents the input tile. If stride one is selected, the elements within the red and green squares will be the two 4×4 outputs of the splitter. The elements within the 5×5 yellow square will be the output of the splitter if stride two is chosen.

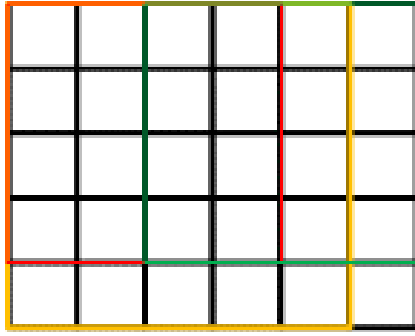


Figure 3.10: Splitter the input in based on the stride selected

3.5.4.3 Input Transform Block

The input transform block contains two parts. The first part uses the 2D transform of Winograd stride one, which can also process the Group 1 of Winograd stride two. The second part transforms the 2D Winograd stride one when the control signal "stride" is low; when the control signal is high, this part performs four 1D Winograd transforms for Groups 2 and 3 of Winograd stride two, and also four multiplications for Group 4. This is shown in Figure 3.11.

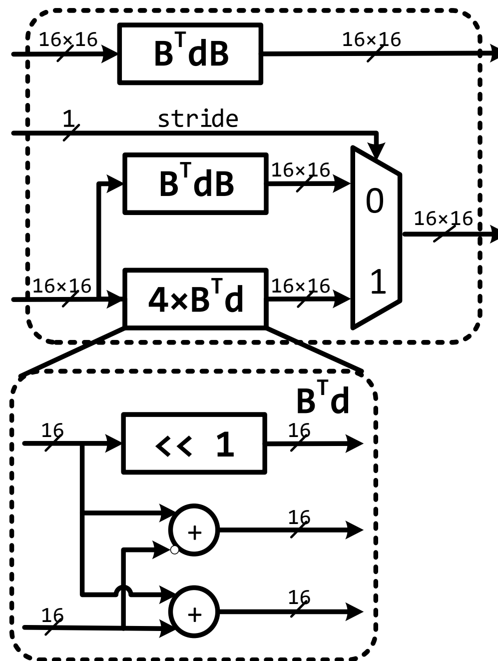


Figure 3.11: Input transform block

3.5.4.4 Filter Transform Block

Our PE has one filter transform module controlled by the control signal "stride." If the signal level is low, a 2D Winograd transform is processed into the kernel and the same output is sent to the DSP module for multiplications; if the signal is high, the outputs are arranged according to the kernel described in Section 3.2.2.

3.5.4.5 Multiplication Block

To perform the convolution operations, multiplications between the input transforms and their corresponding kernel transforms are necessary. This block requires 32 multiplications for the whole process. Similar to [23], an array of DSPs was used to perform the multiplications. The outputs of this block are two 4×4 arrays.

3.5.4.6 Inverse Transform

After the multiplications an inverse transform is applied to the results. The first part uses the inverse 2D transform of Winograd stride one. The second part transforms an inverse 2D Winograd stride one when the control signal "stride" is low; when the control signal is high, this part performs four inverse 1D Winograd transformations for Groups 2 and 3 of Winograd stride two.

3.5.4.7 Accumulator Block

For stride two, the results of the inverse transform for each Group (1-4) are summed and sent to the register. For the case of stride one, the results skip this block and they are sent to the register directly.

Table 3.5 compares the FPGA's resource utilization for the proposed Winograd PE compatible with stride one and two against both a PE capable of two Winograd stride one operations, and a PE calculating one stride two Winograd operation. Although the

proposed PE uses more LUTs and registers than the other Winograd modules, the proposed PE can perform Winograd operations with both strides—unlike the other PEs. Compared to the two Winograd stride one PE, the PE requires only 146 and 100 more LUTs and registers, respectively. Moreover, the proposed PE, with the same DSP usage as that of the two Winograd stride one, allows the additional processing of Winograd stride two. DSPs are a critical resource for FPGAs, and the proposed design uses only 32 DSPs; compare this to the combined 57 required by the independent two Winograd stride one PE and the Winograd stride two PE.

Table 3.5: Resource Utilization of different PEs for kernel=3×3 on INTEL ARRIA-10

Resource	LUTs	Registers	DSPs
Winograd stride two	584	304	25
Two Winograd stride one	930	487	32
Proposed method	1076	587	32

3.5.5 Parallelization

CNN architectures (including VGG-16, ResNet, MobileNet and others) use an input image size of 224×224. Each network block, comprised of convolutional layers and oftentimes ending with a pooling layer, downsamples the feature map by half before outputting a new volume to be used as input to the next block. Thus, the last feature map of these networks prior to the fully connected or prediction layers has a width and height of 7×7.

To process the feature maps efficiently, seven PEs were arranged in a single block to parallelize convolution. This allows for a 28×5 input tile for each block. Inside the block the PEs can read and share the input tile, thereby avoiding multiple memory reads for the same data. The proposed CNN accelerator uses six of these blocks. Although the blocks need to share an input tile and kernel data from the same buffer, connecting multiple blocks directly to the buffer may cause the system to experience issues in timing closure,

especially with a high clock rate. To avoid this issue, the blocks were organized in a systolic array architecture [83]. The proposed design array consists of three rows and two columns, and the rows read input data while the columns read kernel data. Local buffers are used to cache the input and kernel data from each block. Thus, every time a block receives a new data tile, the previous data is also delivered. Using a double buffer in each block also allows computation and data delivery to occur simultaneously. Preliminary results from each block are accumulated in a local buffer until the entire convolution is complete, after which the results can be written to external memory.

3.5.6 Results

In the proposed implementation, the Winograd algorithm was evaluated for both stride one and two using the original and the proposed modified version of VGG-16.

Figure 3.12 shows the results of all convolutional layers in the FPGA using the original and the modified VGG-16 networks. The average performance in the modified VGG-16 network is 8.9% higher than the original VGG-16, owing to higher GOPS (and a higher peak of 2463 GOPS) during stride two operations.

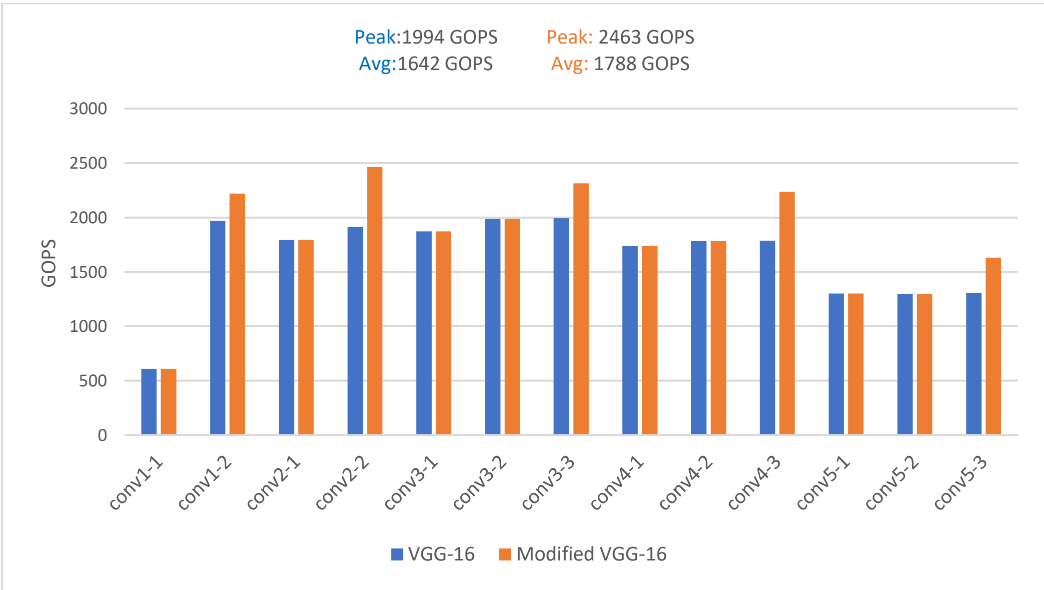


Figure 3.12: Evaluation results of the original VGG-16 architecture compared to the proposed modified VGG-16 architecture

The performance of the first layer (conv1-1) is the worst. This is because the first layer has only three input channels and a large input size which is divided into many groups, taking more time to initialize the elements. The proposed parallelization scheme is efficient when the size of the input feature map and the output layer depth (like in the case of conv2) are well-balanced. However, when the input feature maps are very small, e.g. the lattermost layer (conv5), inefficient memory access patterns cause throughput reduction.

Table 3.6 shows the resource utilization of the proposed CNN accelerator on an Intel Arria-10 FPGA clocked at 250 MHz. The Winograd algorithms reduce the required number of DSPs per convolution; having fewer DSPs per PE allows for more PEs, which enables the possibility of greater parallelism, and multiple blocks can perform multiple convolutions in parallel.

Table 3.6: Resource Utilization of CNN accelerator on INTEL ARRIA-10

Resource	LUTs	M20K RAMs	DSPs
Available	1150K	2131	1518
Used	181K	1310	1344
Utilization %	15.7%	61.5%	88.5%

The proposed implementation uses 88.5% of the DSPs available on the FPGA to achieve high throughput. The Winograd algorithm uses more LUTs than regular convolution to compensate for the reduced number of multiplications. However, even with that increase, the proposed design only used 15.7% of the FPGA's available LUT. Because the proposed design stores groups of input and kernel data in on-chip memory, 61.5% of the available RAM is used. In summary, by using Winograd, the available DSPs can be used efficiently to improve the performance of the accelerator.

Table 3.7 compares the proposed method's overall performance and DSP efficiency against previous FPGA CNN acceleration works. Because one multiplication operation consumes only one DSP and the additional operation does not use a DSP, these

Table 3.7: Performance comparison with state-of-the-art FPGA accelerators

	[28] 2016	[26] 2017	[23] 2017	[29] 2017	[24] 2018	[22] 2018	Proposed method	
Platform	Zynq XC7Z045	VirtexV X690T	MPSOC ZCU102	Zynq ZC706	VCU440	VCU440	Arria-10	Arria-10
CNN	VGG-16	VGG-16	VGG-16	VGG-16	VGG-16	VGG-16	VGG-16	Modified VGG-16
Freq(MHz)	150	200	200	100	200	200	250	250
Precision	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed
Used DSPs	780	2048	2304	725	756	1376	1344	1344
Performance (GOPS)	137	1467	3044	660	943	821	1642	1788
Efficiency (GOPS/DSPs)	0.18	0.716	1.32	0.91	1.2	0.6	1.22	1.33

implementations used 16-bit fixed point data. Using 16-bit fixed-point data is more practical for an FPGA, and the results show that this numeric conversion contributes to only 0.4% accuracy loss [28].

Overall performance throughput depends on the number of resources used by the FPGA. DSP efficiency (GOPS/DSPs) is a fair comparison metric for overall performance because these works were implemented using different FPGA platforms with different numbers of available DSPs.

Our modified VGG-16 implementation achieves 1.33 GOPS/DSPs of DSP efficiency, which outperforms other implementations. The modified VGG-16 network uses convolutional layers of stride two, where one stride two convolution is used instead of one stride one convolution and one max-pooling layer. The PEs in the proposed implementation can be used for stride one or two. Thus, the proposed design has a better efficiency when it is used in architectures that contain stride one and two layers.

For standard VGG-16 implementations which contains only stride one convolutional layers, the proposed work has the second highest DSP efficiency at 1.22 GOPS/DSPs,

behind only [23]. However, the Winograd algorithm in [23] is F(4×4,3×3) which uses an input tile of 6×6 elements; although a larger Winograd input tile leads to a higher throughput, the precision is reduced due to a truncation of intermediate values because the transformation matrices cannot be simply converted to shift operations [26]. To perform strides 1 and 2 Winograd, multiple Winograd F(2×2,3×3) is used which requires division by 2. This division is performed using shift operations with an additional bit; this avoids accuracy reduction when using Winograd when compared to conventional convolution.

3.6 Summary

In this chapter, new algorithms for convolutional neural networks with stride two based on 1D, 2D, and 3D Winograd minimal filtering algorithms were introduced. These algorithms separate the input and the kernel into odd-position and even-position groups, and stride two output is equivalent to the summation of many stride one convolutions. This idea allows for greater efficiency and advantages for CNN architectures that use stride two.

These groups have different array sizes and can be computed using the popular Winograd F(2×3) and the proposed Winograd F(2,2), F(2,4), F(2×2,2×3), F(2×2,3×2), F(2×2,4×4), F(2×2,4×3), or F(2×2,3×4). These new Winograd versions, derived from the popular Winograd F(2,3), decrease computational complexity and increase efficiency by trading expensive multiplications for cheap additions. The GPU implementations of 1D, 2D, and 3D stride two Winograd are tested on several stride two layers and show the proposed method is 1.44x, 2.04x, 2.42x, and 1.73x faster for the respective 3×3, 5×5, 7×7, and 3×3×3 kernels.

A Winograd PE was designed which can process stride one and two in the same module. The combined PE uses the same number of DSPs (32) as a PE for two Winograd stride one calculations, and 25 less DSPs than that required by having independent PEs to calculate two Winograd stride one, and one Winograd stride two, operations. This

optimization allows an increase in the number of total PEs. Using a systolic array with a double buffer, greater parallelism was enabled in the design.

Stride two convolutional layers produce smaller output volumes while still extracting feature maps. This was demonstrated with the implementation of the proposed modified VGG-16 architecture where one stride two convolutional layer is used instead of one stride one convolutional layer followed by one max-pooling layer. The proposed implementation achieves DSP efficiencies of 1.22 GOPS/DSPs and 1.33 GOPS/DSPs for the original and modified VGG-16 architectures, respectively. Because the proposed design can effectively handle both stride one and two layers within the same architecture, the proposed method is a new technique for emerging architectures which contain layers using stride one and/or two, and those increasing the numbers of stride two layers used.

Part III

Applications for Intelligent Transportation Systems

Part III of this thesis present three Intelligent Transportation Systems (ITS). The first system, presented in Chapter 4, is a deep learning-based embedded license plate localization (LPL) system. In Chapter 5, a real-time Commercial Vehicle Safety Alliance (CVSA) decal recognition system (CDRS) using deep convolutional neural network architectures is proposed. The final system, presented in Chapter 6, is a real-time deep learning-based edge system for HAZMAT recognition.

The systems use custom deep learning architectures designed for the specific detection/recognition task. The convolutional layers within the architectures use a mix of stride 1 and 2—the importance of the latter stride to object detection convolutional layers being explained in Part II.

The LPL system is a one stage (single shot) system. The CDRS system uses two stages (detection/classification then fine-grained classification), and the HAZMAT recognition system is a three-stage system (detection/classification and two fine-grained classification models). A detailed comparison with other networks is provided for each designed model stage. Each designed system is compared against similar models using the same dataset, or directly against related work within academic literature (where such work exists).

When this research was conceived, there were no commercially available hardware edge accelerators. Therefore, the FPGA design presented in Part II was planned to provide the underlying hardware framework for the ITS applications proposed in this part. However, the FPGA has considerable costs to rewrite code for varying custom deep learning architectures. Because companies like Google, Intel, and NVIDIA released their own hardware accelerators during the research period, the research pivoted to incorporate such commercial edge systems and hardware accelerators instead of FPGA designs; the applications presented in Part III are evaluated using these commercially available devices. The edge devices are low-cost and enable the integrated model architectures to achieve high speed and accuracy. Models are evaluated on the edge devices that existed and/or were in widespread use at the time the models were created. Edge devices evaluated in Part III include the Raspberry Pi, the Intel Neural Compute Stick, the Google Coral USB accelerator, the NVIDIA Jetsons, and/or combinations thereof.

Chapter 4

Deep Learning-based Embedded License Plate Localization System[1]

This chapter presents a novel neural network architecture for license plate localization (LPL) based on an inverted residual structure where the shortcut connections are between the linear bottleneck layers. The proposed deep learning (DL) solution was tested against three popular international research databases and achieves state-of-the-art results, proving that the model is accurate and robust. Across those databases, the proposed model surpasses other recent LPL work, including DL-based methods, in terms of accuracy and speed. Using a novel multi-threading video capture with motion detection then inference algorithm, the computational efficiency was increased and drop less frames overall, allowing for increased performance. Repeated tests show the proposed method is well-suited to real-time and highly accurate LPL, regardless of hardware. Section 4.1 presents the motivation to design an embedded LPL system; Section 4.2 describes the proposed DL LPL solution; Section 4.3 provides comparisons and the outcomes of the tests; and Section 4.4 concludes this chapter.

4.1 Introduction

Automatic License Plate Recognition (ALPR; also ANPR for “name” or “number” plates) refers to the capturing and processing of license plate information via

[1]The content of this chapter is originally published in IET Intelligent Transport Systems [107]. The manuscript has been reformatted for inclusion in this thesis.

Juan Yépez (JY), Riel Castro-Zunti (RC), and Seok-Bum Ko (SK) designed the study. JY designed the network architecture, trained and tested the models, designed the motion detection system, implemented the system on the edge devices, and provided results analysis. RC helped annotated the images from dataset and proofreading the manuscript. JY prepared the manuscript with contributions from SK to the manuscript structure, readability and analysis and discussion of the results

computational or algorithmic means. ALPR is a subfield of Intelligent Transportation Systems (ITS) [84], which includes services aimed at improving the driver experience.

Applications of ALPR are many and varied, and include the following: law enforcement tools, including for stolen or unlicensed vehicles; toll booths and issuing fares or tickets; and driver-to-driver communication. ALPR may also play a key role if or when driverless vehicles become conventional and ubiquitous, as information linked to license plates may become incorporated into algorithms that calculate the most ethical or advantageous action in an emergency or life-or-death situation.

An ALPR solution should minimize processing time and maximize accuracy, especially where the captured image may be blurry, rotated, obscured, or otherwise distorted. Minimizing processing time is especially important in "real-time" applications, such as red light or police traffic cameras.

There are generally five sequential steps in an ALPR solution: image capture; vehicle detection; license plate localization (LPL); plate character segmentation; and optical character recognition (OCR) of the segmented characters [85]. However, some systems use pre-processed images from elsewhere, and thus are designed only to achieve LPL, character segmentation, and OCR [86].

This chapter focuses on the research, results, and conclusions for LPL using a bottleneck depth-separable convolution with inverted residuals deep learning architecture. Although the proposed solution is designed for GPU usage, it is demonstrated that the system runs reasonably well on low-power, low-cost devices such as CPUs, embedded systems, smartphones, tablets, and other personal mobile devices [15].

The contributions of this chapter are, in order of appearance:

1) An accurate and robust deep learning (DL) architecture for LPL based off depthwise separable convolutions and residual linear bottlenecks.

2) A multi-threading video capture with motion detection then inference algorithm to increase computational efficiency by only allowing the DL architecture to run when (vehicle) motion is detected; this method drops less frames overall.

3) Overall systems testing in a real-world high-speed highway environment showing 99.77% plate region localization over 898 vehicles that passed beneath the video capture setup.

4.2 Proposed Solution

In this thesis a neural network architecture for license plate localization using bottleneck depth-separable convolution with inverted residuals was proposed. This architecture uses a single step for localization, in contrast to [87]–[89] and [90], where two or more steps are required.

4.2.1 Neural Network Description

The neural network used for LPL is based on SSD architecture [48]. The original feature extractor used in SSD is VGG-16 [48]. VGG-16 consists of 13 convolutional layers followed by three fully connected layers and is very appealing because of its uniform architecture [91]. However, VGG consists of about 140 million parameters [91], making a system using it computationally complex and thus requiring a powerful GPU to run effectively and within an acceptable timeframe. In this work, a different feature extractor involving blocks based on linear bottleneck depth-separable convolution with residuals was proposed.

Combining the versatility of depthwise separable convolutions [15] with the underlying ideas of relevant information extraction, abstraction, and accumulation inherent in linear bottlenecks could provide an accurate and fast LPL solution.

Depthwise separable convolutions decrease computation and processing time with little to no reduction in overall accuracy, and any accuracy reduction is compensated with

the use of linear bottlenecks due to their demonstrated in-practice performance gains, especially for classification tasks.

See Figure 4.1 for a visual comparison between standard and depthwise separable convolution operations. In Figure 4.1 (a), 128 filters with value of 3×3 pass over 3 input channels to generate 128 output channels, in the process the filters move 5×5 times. There are 128×3×3 (filters) × 3 (input channels) ×5×5 (movement) = 86,400 multiplications. Depthwise separable convolution consists of two parts, depthwise convolution and pointwise convolution. Depthwise convolution is shown in Figure 4.1 (b); during this convolution one 3×3 filter passes over 1 input channel 3 times, in the process utilizing 3×3 (filter) × 3 (input channels) ×5×5 (movement) = 675 multiplications. Subsequently during pointwise convolution, shown in Figure 4.2 (c), 128 filters with value of 1×1 (filter) × 3 (input channels) passes over the result of the depthwise convolution to produce 128 output channels, in the process utilizing 128×1×1 (filters) × 3 (input channels) ×5×5 (movement) = 9,600 multiplications. Thus, the entire depthwise separable convolution operation in Figures 4.1 (b) and 4.1 (c) approximates the traditional convolution in Figure 4.1 (a) but uses only 10,275 multiplications; this is 88.1% less parameters than the standard convolution in Figure 4.1 (a). One can easily note the computational efficiency of depthwise separable over standard convolution operations as o gets large.

Residual blocks with shortcut connections were utilized between successive layers, as in ResNet [12], to increase accuracy and further decrease required parameters. However, rather than use shortcut connections to connect (wide) layer inputs to blocks, shortcuts between the (narrow) bottleneck sub-blocks were applied—using the idea that the most

Table 4.1: Bottleneck Residual Block Structure

Input	Operator	Output
$h \times w \times k$	1×1 conv2d, ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3×3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	Linear 1×1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

h = height, w = width, k = channels, s = stride.

relevant information is contained in the bottlenecks—and it is those bottlenecks that should be given as input to a block. This forms the basis of the inverted residual block. The structure of this block is shown in Table 4.1.

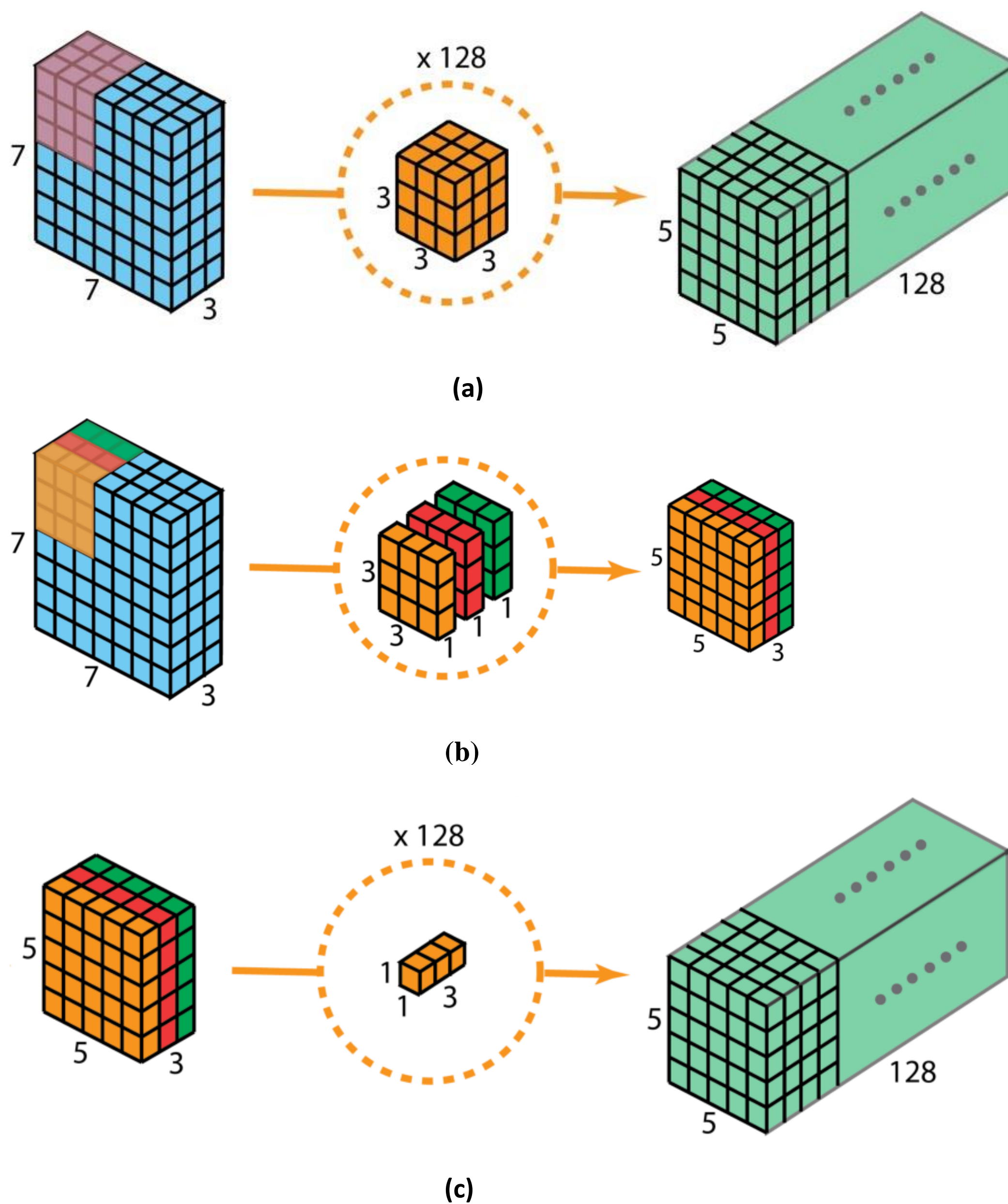


Figure 4.1: Standard vs. Depthwise separable convolution operations to demonstrate the effectiveness of depthwise separable convolution over standard. The 7×7 input with 3 channels and 128 filters are for illustrative purposes, and arbitrary. **(a)** Standard convolution. Depthwise separable convolution uses less parameters than standard convolution because separates the process in 2 parts: **(b)** Depthwise convolution. **(b)** Pointwise convolution.

Instead of using convolutions to successively narrow layers within a block as in ResNet, the bottleneck input layer (by a 1×1 convolution) was first expanded and then narrowed using a depthwise separable convolution, which lowers the number of parameters compared to the regular residual block.

Note that the ReLU6 non-linear activation function is defined as follows:

$$ReLU6(x) = \begin{cases} 6,\ if\ x > 6 \\ x,\ if\ 0 < x \leq 6 \\ 0,\ if\ x \leq 0 \end{cases} \tag{4.1}$$

ReLU6 has been shown to perform well in low-precision applications [4].

The inverted residual improves the ability of a gradient to backpropagate across multiple layers, similar to a classical residual connection but using a linear bottleneck. The bottleneck residual block can be treated as a single operation, and the amount of memory used depends on the size of the bottleneck rather than the size of tensors, making this method more memory efficient.

The feature extractor architecture contains the initial fully convolutional layer with 32 filters, followed by 19 bottleneck residual blocks. The kernel size is 3×3, for non-linearity ReLU6 is used, and dropout and batch normalization are utilized during training. A detailed architecture for the feature extractor can be found in Table 4.2.

The neural network allows scale-invariant license plate prediction because layers decrease in size progressively and the convolutional model for localization is different for each feature layer.

One difference in the proposed model compared to original SSD is the input image size; in original SSD, this could be either 300×300 or 512×512, but in the proposed system the size is reduced to 224×224, which increases speed but can decrease accuracy. However, using residual bottlenecks allows high accuracy with less parameters and multiplication operations in the inference process—inference is the stage in which a trained model is used to make predictions on input data and samples (see section 4.2.3).

Table 4.2: License plate localization neural network used as feature extractor, * Layers that are used as input for the detection block.

Input Size	Operator	Repetition	Kernel
300×300×3	Conv2d	1	3×3
150×150×32	Bottleneck	1	3×3
150×150×16	Bottleneck	2	3×3
75×75×24	Bottleneck	3	3×3
38×38×32	Bottleneck	4	3×3
19×19×64	Bottleneck	3	3×3
*19×19×576	Bottleneck	3	3×3
*10×10×1280	Bottleneck	1	3×3
*5×5×512	Conv2d	2	3×3
*3×3×256	Conv2d	2	3×3
*2×2×256	Conv2d	2	3×3
*1×1×128	Conv2d	1	3×3

The first layer of the modified SSD is attached to the expansion of layer 15 with output stride of 16. The second and the rest of the layers are attached to the top of the last layer, which has an output stride of 32. Finally, all layers are attached to the feature map of the output. A modified SSD was used in which all standard convolutions in the prediction layers are replaced with depthwise separable convolutions. The modified SSD is faster than other object detection variants while offering comparable accuracy [30], [48], making it a useful architecture for all devices and especially those with low computational complexity. The proposed architecture is shown in Figure 4.2.

4.2.2 Training Process

Training is the phase in which a network learns patterns from given data. In training, each layer of data is assigned some random weights and a classifier runs a “forward pass” (propagating weights and calculating outputs) through the data, predicting the class labels

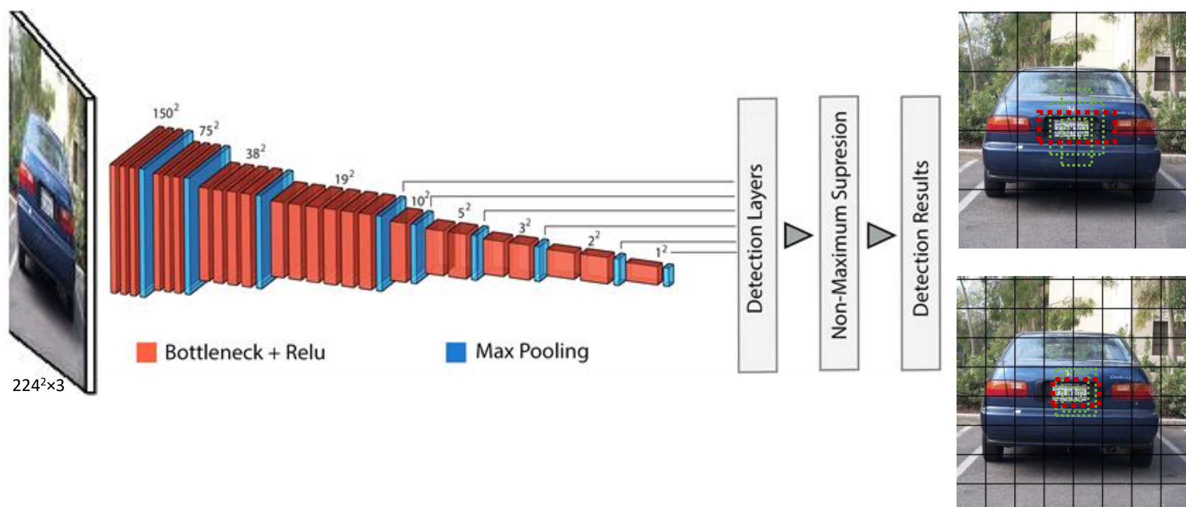


Figure 4.2: Architecture of the proposed license plate localization method. This network is based on Single Shot MultiBox Detector, but differs in that the proposed architecture uses depthwise separable convolution, as opposed to standard convolutions, and layers of linear bottlenecks with inverted residuals. Note that the bottleneck layers contain the depthwise separable convolutions, which are comprised of a depthwise convolution followed by a pointwise convolution. For more information on depthwise separable convolution. Car images from the Caltech dataset [96].

and scores using those weights. The class scores are then compared against the actual labels and an error is computed via a L1 loss function for prior box learning and a log Softmax of cross-entropy for class learning. This error is then backpropagated through the network and weights are updated accordingly via some algorithm such as gradient descent. One complete pass through all training samples is called an epoch.

Training is computationally very expensive. To make training faster, data is divided into batches, and weights are updated after each batch; this method takes less time to converge, and fewer epochs are needed to construct an adequate DL model from the data.

Training was performed using a single class: the license plate. A supervised learning apparatus was utilized with all license plate regions labelled/annotated using labelImg [92]. Rectangular bounding boxes were drawn about a plate region to minimize irrelevant background area around the plate. From a labelled image, labelImg creates an XML file that describes the location of a plate within the image and indicates its class. This process was used for each image of the training dataset.

See Figure 4.3 for examples of labelled images.

Figure 4.3: Images of plate regions on vehicles labelled using labelImg [92] (with emphasis added by manually drawing yellow boxes around the labelled plate region). From each of these, a respective XML file will be generated for record creation and training/testing. The first row contains images from the Caltech dataset [96], the second row from the University of Zagreb dataset [100], and the third from the NTUA Medialab dataset [102].

Labelled data is then split into train and test sets: 80% of images are randomly taken for training, and 20% for testing. These images are joined in respective training and testing record files.

Setup and configuration of the hyperparameters are made. The most important hyperparameters to tune are learning rate, optimizer, batch size, and epochs.

During an iterative process called training, a model adjusts its weights to attempt to achieve the minimum loss in feature space as determined by some loss function [93]; the magnitude of the weight adjustments is governed by the learning rate hyperparameter [94]. A higher learning rate enables the model to learn faster, but it may miss the global minimum loss and only (unstably) reach some local minimum surrounding the global. A lower learning rate gives a better chance to find the minimum loss, but the model may get "stuck" at a local minimum. As lower learning rate typically requires more epochs for convergence, meaning exhausting more time and memory capacity resources. The optimizer is responsible for changing the learning rate and weights of neurons in the neural network to reach the minimum loss, typically via a process called backpropagation. The optimizer is important to achieve the highest possible accuracy or minimum loss. A common solution is to start with a higher learning rate that progressively decays throughout the training cycle [94].

Batch size refers to the number of training examples utilized in one iteration and subsequent weight adjustment [95]. When the batch size is too low, the network weights are too frequently adjusted on too little information, meaning the model cannot effectively learn from the data or (within a reasonable period if at all) converge to a final state. This negatively impacts total training time and accuracy.

An epoch refers to one whole processing (and subsequent backpropagation) of the entire training dataset by the model [17]. Multiple epochs of training are typically required. Training with too few epochs may result in under fitting because the neural network has not seen the data enough times to learn relevant patterns [95]. Conversely, training with too many epochs may lead to overfitting, where the model begins to

associate “feature noise” within the training set as relevant to the task; this means the model can predict the training data very well but cannot sufficiently generalize to unseen data. A validation set independent from the training set can help overcome this problem; the validation set is iteratively used to evaluate the model after a certain number of training epochs, and where validation set performance starts to decrease indicates the point at which the model is likely starting to become overfit.

The hyperparameters are as follows: a batch size of 24, an initial learning rate of 0.004, the Adam optimizer, and a momentum optimizer value of 0.9.

Our model was trained using a Tesla K40c GPU on a computer whose operating system is Ubuntu 18.04 LTS. The framework was Tensorflow v. 1.8 running on Python 3.6.5. The model was trained across 200,000 iterations. Finally, a graph file is exported from the new trained model containing weights used in inference.

4.2.3 Inference Process

Training demands high computational throughput; thus, it is most often performed by GPUs, given their massive parallelism, simple control flow, and energy efficiency. For inference, however, the paramount performance goal is latency. To minimize the network’s end-to-end response time, inference typically batches a much smaller number of inputs than training, as automated services relying on inference are required to respond in near real-time.

Inference can be sped up by using a GPU as opposed to a CPU as GPUs perform vector and matrix manipulation much faster. A DL solution requires many multiplication operations to produce inference; thus, DL typically requires a GPU with many TFLOPS to increase parallel processing and reduce processing time to a duration adequate and sufficient for normal application operation.

Our trained model is compared with other works using a Tesla K40c GPU (see section 4.3.2). However, the proposed LPL model obtains high performance at a lower price, designed to be fast and accurate even when running on commodity hardware.

4.2.4 Proposed DL LPL Algorithm

An algorithm that can run sufficiently fast and efficient on computers using only a CPU was defined, as well as low-cost devices such as embedded systems, smartphones, tablets, and other personal mobile devices. This algorithm consists of three stages:

- Video input multithreading
- Motion detection
- DL inference

The diagram for the proposed algorithm is shown in Figure 4.4.

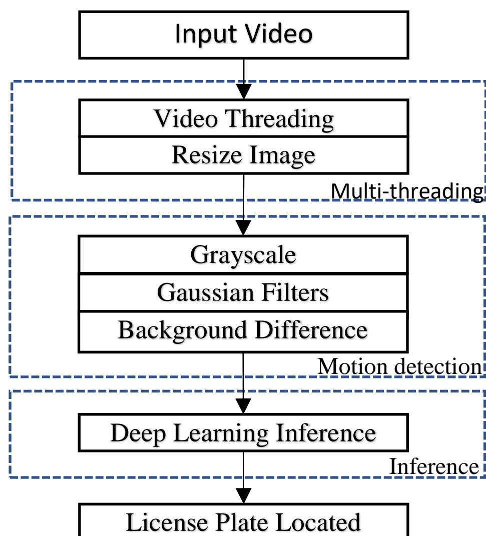


Figure 4.4: Program flow of proposed multi-threading video capture with motion detection then inference

Normally, accessing a network/USB camera is a blocking operation; the main code is blocked until the frame is read from the camera device and returned to the main script. This can cause an input/output bottleneck, and thus reducing the time to capture each frame is especially important when a real-time application is required. An LPL system

cannot waste any time between captures because a fast-travelling vehicle, such as on a highway, could cross in that moment.

Recognizing this importance, an I/O thread separate from the main script was defined that allows frames to be read continuously from the camera; frames are read and buffered from the I/O thread while the main thread processes the current frame. Once the main thread has finished processing its frame, it simply grabs the current frame from the I/O thread. Thus, LPL is achieved without having to wait for blocking I/O operations; this makes the system more efficient when used on CPUs, and the overall frames per second (FPS) is increased.

Since LPL must occur in the presence of a vehicle, a motion detection process was developed which detects when a car is in the frame; only after the vehicle completely leaves the bottom of the frame will the LPL object detection DL apparatus run, to capture the vehicle's rear license plate. This makes the overall algorithm more efficient and increases the system's maximum FPS, as LPL inference is performed only when required. An example of when to start detection is shown in Figure 4.5.

The motion detection process utilizes the underlying assumption that the background of consecutive captured video frames is largely static and immutable. Therefore, the background can be modelled and supervised for substantial changes. When such a change occurs, it is detected and corresponds to movement in the video capture. Because movement is invariant to color, the image is converted to grayscale and softened via Gaussian blur.

Finally, Gaussian smoothing is applied to the average pixel intensities, which softens high frequency noise.

The algorithm takes the first frame of the video as the static background, though theoretically any image could be imported for use as the background. The difference between the background frame and the new subsequent frames of the video transmission is calculated; this difference is a simple subtraction, taking the absolute value of corresponding pixel intensity differences:

$$\Delta frame = |\, background - current\ frame \,| \tag{4.2}$$

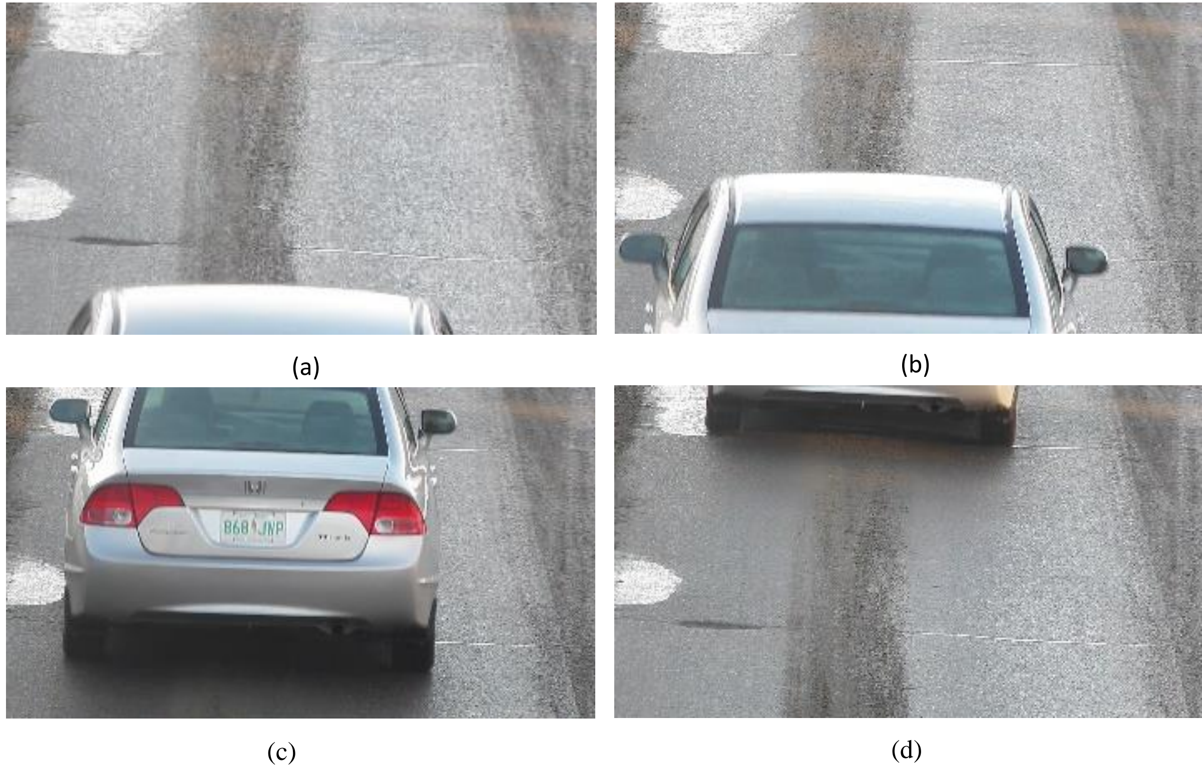


Figure 4.5: (a-d) Frames of a vehicle travelling at highway speed passing under a camera setup. Vehicle presence would be detected in **(a)** via background subtraction against the (otherwise vehicle-clear) road. Deep learning license plate localization inference would occur in **(c)**, when the vehicle is detected to have left the bottom of the frame and while the rear license plate is still visible; this minimizes the total number of frames over which inference occurs, thereby increasing the possible maximal frames per second.

This subtraction causes the background in a subsequent frame to appear black while regions that contain non-background, such as moving vehicles, are white. This implies that larger frame deltas indicate that movement is occurring in the image. The Δframe was adjust to reveal the regions of the image that have significant changes in pixel intensity values.

The final step of this algorithm is inference, which is performed with the deep neural network model using a feed-forward algorithm that operates on each applicable video frame separately. The algorithm begins at the input layer and progressively moves forward layer by layer. At each layer the feed-forward algorithm updates the state of each unit; this process terminates once all units in the output layer are updated. The inferred class corresponds to the output layer unit with the largest state, the data within the

localization bounding box region, and the confidence value of the output layer in its prediction of whether the region contains a valid license plate.

4.3 Results

4.3.1 Dataset Information

Our solution involves a DL model with supervised learning trained using the datasets in Table 4.3. One such publicly available dataset may also be referred to as a public library.

Table 4.3: Public datasets used to train the deep learning models

Public Dataset	# of Images in Set	Image Size (px)
Caltech Cars 1999 (Rear) 2 [96]	126	896×592
University of Zagreb License Plate Detection, Recognition, and Automated Storage [100]	510	640×480
University of Athens (NTUA) Medialab LPR Database [102]	571	1792×1312, 800×600, 640×480

In total there are 1207 sample images from different scenes and different countries. Many plates exhibit small amounts of rotation and skew. Others exhibit over or underexposure. Several plates are blurry, partially obscured by shadows, dirty, or otherwise distorted. These unideal conditions rarely occur in isolation; see Figure 4.6.

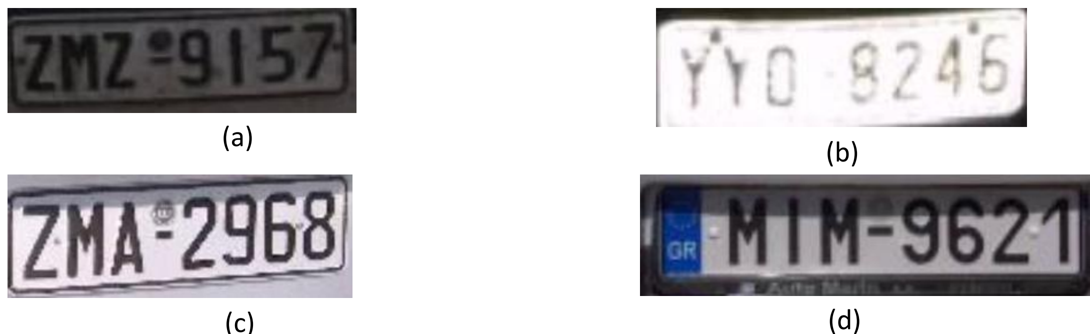


Figure 4.6: Different obstructions on license plates in images from the NTUA Medialab dataset [41]. **(a)** Rotated, Underexposed Plate. **(b)** Blurry, Overexposed Plate. **(c)** Skewed Plate. **(d)** Partially Shadow-covered Plate.

4.3.2 Comparisons to Public Libraries

Our tests were conducted using a Tesla K40c GPU on a computer whose operating system was Ubuntu 18.04 LTS. Experiments were implemented in Python 3.6.5 using the libraries Tensorflow v. 1.8 and OpenCV v. 3.4.0.

Tests were conducted using an intersection over union (IoU) threshold of 0.5.

Localized plate regions from all the public libraries described in section 4.3.1 are shown in Figure 4.7.

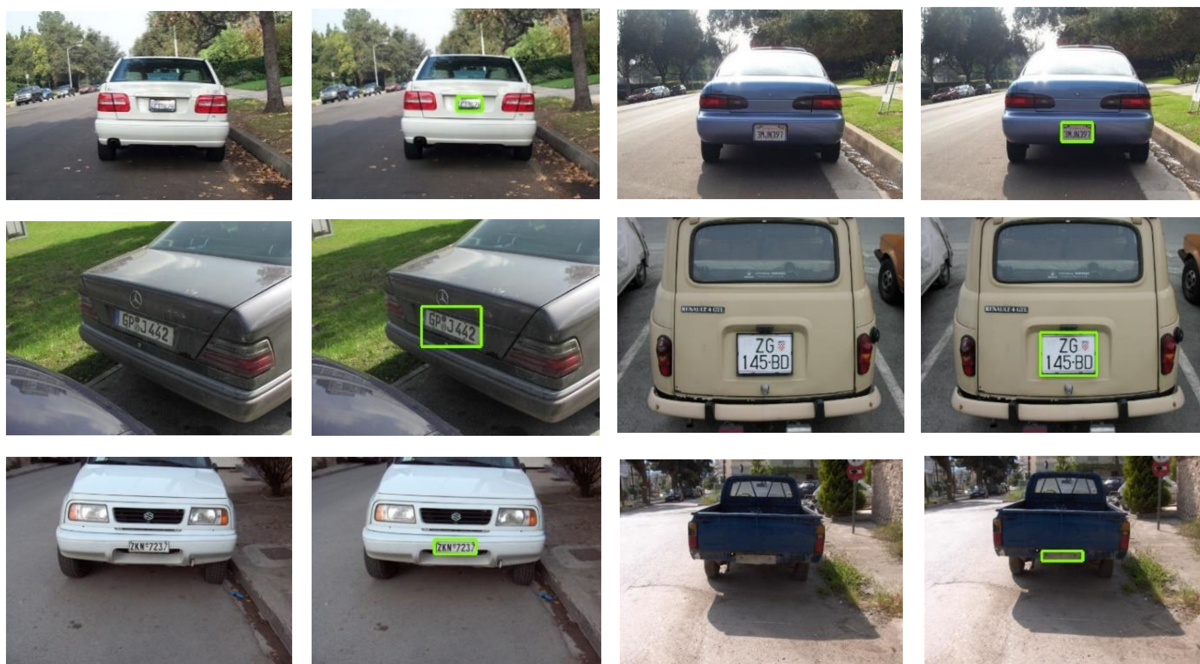


Figure 4.7: Images of vehicles and their corresponding respective localized license plates generated by the proposed method. The first row contains images from the Caltech dataset [96], the second row from the University of Zagreb dataset [100], and the third from the NTUA Medialab dataset [102].

4.3.2.1 The Caltech Dataset

The Caltech Cars 1999 (Rear) 2 [96] public library is comprised of images of vehicles in Caltech parking lots. The plates are American and 120 are Californian; few other U.S. states are represented. All vehicles were photographed in daylight. Most plates are clearly visible, exhibit little rotation or skew, and are generally human-readable; though many

Table 4.4: Comparison of license plate localization algorithms on the Caltech Cars 1999 (rear) 2 dataset [96]

Description of System/Algorithm	Correct Detection %	Processing Time Per Image (s)
Proposed system	98.4	0.02
Faster R-CNN for vehicle detection + CNN Classifier [87]	98.39	None given
Faster R-CNN + RPN [97]	98.04	0.279 (estimated)
Line Density Filter + SVM-based Classifier [98]	91.27	< 0.042 (estimated)
Feature Extraction + Principal Visual Word [99]	84.4	7.19

plates are somewhat unfocused, and may have unideal exposure. Most license plates have a width-height ratio of approximately 2-2.5. A typical plate region occupies ≤ 1% of the total image, and is located on a vehicle that occupies approximately 20-40% of the image; the remainder of the image is (meaningless) background. Most plates contain darker alphanumeric characters on a lighter background. See Table 4.4 for comparisons.

Our method has an accuracy of 98.4% and is 0.01% more accurate than the next closest [87]. Although [87] does not give a processing time, it is likely near the time given by [97] due to the similarities between their respective neural network processes—and the system localizes almost 14x faster than [97], which also uses a GPU (though their listed time was for a different dataset as no time was given for Caltech [96]). Reference [97] is also 0.36% less accurate than the proposed method. Moreover, [87] and [97] use Faster R-CNN as an object detector whereas the proposed solution uses a modified SSD. There are numerous advantages of SSD over Faster R-CNN, including hard negative mining to reduce negative bounding boxes during inference, and a concatenation of low- and high-level discriminative features.

Compared to [98], which uses a CPU, the proposed system localizes about 2x faster—and it is 7.13% more accurate, meaning the proposed system correctly localizes about 9 more plates than [98].

Compared to [99], which uses a CPU, the proposed system localizes about 360x faster and it is 14% more accurate, meaning the proposed system recognizes about 18 more plates than [99].

4.3.2.2 The University of Zagreb Database

The University of Zagreb Plate Detection, Recognition, and Automated Storage [100] public library is comprised of images of cars with Croatian license plates, though a small minority of other European plates are represented. Many are clearly visible and are generally human-readable; but some exhibit rotation, skew, blurs, shadows, and other unideal conditions. Some images have been captured in low-light conditions. Most license plates have a width-height ratio of approximately 5, and several have a ratio of approximately 1.46. A typical plate region occupies approximately 1-2% of the total image, and is located on a vehicle that, with few exceptions, occupies the majority, and in some cases the entirety, of the image; any remaining image space contains (meaningless) background. Most plates contain darker alphanumeric characters on a lighter background. See Table 4.5 for comparisons.

Table 4.5: Comparison of license plate localization algorithms on the University of Zagreb plate detection, recognition, and automated storage dataset [100]

Description of System/Algorithm	Correct Detection %	Processing Time Per Image (s)
Proposed system	97.83	0.02
Corner-point Detection + Linear Discriminant Analysis-based Classifier [101]	92.8	0.12 (estimated)

Compared to [101], which uses a CPU, the system localizes 6x faster. The proposed system has an accuracy of 97.83%; this is 5.03% more accurate than [101], meaning the proposed system correctly localizes about 26 more plates.

4.3.2.3 The NTUA MediaLab Dataset

The NTUA Medialab LPR Database [102] is comprised of images of cars with Greek license plates. Most license plates have a width-height ratio of approximately 3.5-4.8. A typical focused plate region occupies < 1% to approximately 3% of the total image, though some plate regions occupy as large as 5-9%; a plate region is located on a vehicle that occupies anywhere from approximately 10% to the entirety of the total image; remaining image space may contain other vehicles with visible (but not necessarily readable) plates. Most plates contain darker alphanumeric characters on a lighter background. At the time of training and testing, the NTUA Medialab LPR database was split across 8 sample sets; for detailed set information see Table 4.6. See Table 4.7 for comparisons.

Table 4.6: Set specifics of the NTUA Medialab LPR Database [102]

Set #	# of Images in Set	Description
1	136	Color images captured in daylight, with an easily visible plate region
2	122	Zoomed view, with large and easily visible clear plate regions on a non-complex background
3	49	Images with plate regions obscured by shadows, and some with unideal illumination
4	67	Similar to Set 1
5	7	Blurred images
6	3	Color images captured at night using an external flash
7	26	Complex images with plate regions obscured by shadow
8	161	Complex images with plate regions obscured by shadow and dirt

The proposed system has an accuracy of 99.8% and is 1.35% more accurate than [103]. The system correctly localizes 2 more plates than [103] from Set 4, and 6 more plates than [103] from Set 8 (which contains complex images). Reference [103] utilizes a CPU and processes an image in 0.2 s, the same amount of time it took the system during the tests.

Table 4.7: Comparison of License Plate Localization Algorithms on the NTUA Medialab LPR Database [102]

Description of System/Algorithm	Correct Detection Percentage By Set # (%)								Correct Detection %	Processing Time Per Image (s)
	1	2	3	4	5	6	7	8		
Proposed system	100	100	100	100	100	100	100	99.37	99.8	0.02
Morphological Operations [3]	100	100	100	97	100	100	100	95.65	98.45	0.02
Connected Component Analysis [25]	92.02	82.48	88.73	87.24	74	90.84	N/A	N/A	89.45 (sets 1 through 6 only)	0.035

Compared to [88], which uses a CPU, the proposed system is 1.75x faster and is 10.35% more accurate—and the accuracy gain over [88] would likely be even more had the results of [88] included Sets 7 and 8, which featured more complex images than the other sets against which [88] was tested.

The proposed system misses localizing only one plate from the NTUA Medialab LPR Database [102], from Set 8.

4.3.3 Comparisons to Popular DL Object Detection Frameworks

The proposed LPL model was also tested for speed and efficiency using only a CPU, giving us a rough metric for compatibility with low computational complexity devices like smartphones and embedded systems.

The DL solution was against two well-known DL architectures, SSD and YOLOv2. YOLOv2, particularly over its predecessor, is aimed at real-time object detection tasks [104].

The tests were conducted using an Intel Core i7-2620M CPU (2.7 GHz) with 8 GiB of RAM on a computer whose operating system was Ubuntu 16.04 LTS. Experiments

were implemented in Python 3.6.5 using the libraries Tensorflow v. 1.8 and OpenCV v. 3.4.0.

Results are summarized in Table 4.8. The proposed architecture is 24x faster than, uses 6.1% of the parameters of, and uses 21.9x fewer multiply-add (MAdd) operations compared to YOLOv2. Against standard SSD [48], the proposed system is 7.5x faster, uses 8.6% of the parameters, and 44x fewer MAdds.

Table 4.8: The proposed license plate localization deep learning architecture compared to popular DL architectures

Network	Parameters	MAdd	CPU Time Per Image (s)
Proposed system	**3.1M**	**0.8B**	**0.2**
SSD [48]	36.1M	35.2B	1.5
YOLOv2 [104]	50.7M	17.5B	4.8

4.3.4 Comparisons to other DL LPL Frameworks

Because some DL LPL systems test using private databases, comparisons of accuracy and processing time (as in section 4.3.2) cannot be directly made.

One such example is [90]. As of August 2018, one database used to test the accuracy of [90] is no longer available, and the other [98] is semi-private and for use on a case-by-case basis only. Thus, the model was compared against [90] in terms of its neural network computational complexity. The tests were conducted using the same computer setup in section 4.3.3. Results are summarized in Table 4.9.

Table 4.9: The proposed license plate localization deep learning architecture compared to other deep learning license plate localization architectures

Network	Parameters	MAdd	CPU Time Per Image (s)
Proposed system	**3.1M**	**0.8B**	**0.2**
ALMD-YOLO [34]	59.57M	1.81B	0.67

The proposed architecture is 3.35x faster than, uses 5.2% of the parameters of, and uses 2.26x fewer multiply-add (MAdd) operations compared to the ALMD-YOLO architecture described in [90]. Theoretically, the proposed model should perform more efficiently than [90], especially on devices with lower computational complexity.

4.3.5 Real-life Real-time Testing

License plate recognition systems often require connection to a server to localize license plates from a video feed. When the server is stationed at a remote location, high bandwidth per-camera is required, increasing the cost of the system. To mitigate this, a real-time processing system at the edge was proposed using the low-cost embedded system Raspberry Pi 3 and an Intel Neural Compute Stick 2 (NCS2).

The Raspberry Pi 3 device with a quad core 1.2GHz processing power chip and 1 GB of RAM. The NCS2 enables a CNN to be deployed on a low-power chip, thereby enabling real-time inference for license plate detection without requiring a connection to the cloud or a large processing server. The NCS2 allows 1 trillion operations per seconds (TOPS), and the embedded system was used for testing. The camera utilized was a Samsung S5K2L1 with a 12 MP resolution and a sensor of 1.4 µm; 1/2.6" and 10x optical zoom was used. The camera streams a 60 FPS video to the Raspberry Pi 3.

Using the algorithm proposed in section 4.2.4, 99.77% localization accuracy was achieved when testing with 898 vehicles. The system can run at an average of 13 FPS. Testing was conducted during summer, winter and snowy conditions, and some license plates are covered with mud and dust; this is itself a testament to the robustness of the proposed architecture, given that the training images used were captured in non-winter conditions. The test occurred on a highway with average vehicle speeds in excess of 80 km/h, or about 50 mph.

The proposed DL design is based on multibox detection and it can thus detect multiple license plates simultaneously with the same processing time as with a single plate; see Figure 4.8.

Figure 4.8: Real-time can detect one or multiple license plates at the same time.

Some ALPR systems are susceptible to the angle of the license plate within an image. However, using DL methods greatly mitigates this via the rotationally-invariant nature of feature-map outputs from CNNs, and therefore an angled plate or camera will not generally be a problem.

A hyperlink to the videos showing the proposed system operating in a real-life real-time high-speed highway environment can be found at https://youtu.be/7eyfGCW_UwQ

4.4 Summary

In this chapter, a deep learning apparatus for LPL was proposed. The LPL system uses inverted residual blocks, linear bottlenecks, and depthwise separable convolutions. It requires only 3.1M parameters and 0.8B MAdds. Using a GPU, per-image plate localization takes only 20 ms of processing time.

The proposed system was trained using the Caltech Cars 1999 (Rear) 2 library, the University of Zagreb License Plate Detection, Recognition, and Automated Storage library, and the NTUA Medialab LPR Database. Testing achieved 98.4%, 97.83%, and 99.8% correct detection accuracy over those same respective datasets.

Plates regions were correctly identified despite rotations, skew, under or overexposure, blur, obstruction by shadows or dirt, and/or other distortions. This shows that the proposed model is robust against non-ideal plate captures and environmental conditions, and reliable even in images with complex backgrounds.

Using the proposed multi-threading video capture with motion detection then inference algorithm, frames are read and buffered through a multithreading process when the system detects motion, and sent to the DL model for LPL inference. The proposed DL model and algorithm allows DL-based LPL to be efficient, accurate, and well-suited for real-time applications on low-computational devices; LPL is achieved within an acceptable time frame (77 ms) when an embedded system and neural compute stick is used. The algorithm was tested on highway conditions with vehicles traveling in excess of 80 km/h and achieved 99.77% localization over 898 test vehicles.

As both the localization processing time and neural network parameter count are relatively low in the current system, especially compared to other deep learning solutions, the framework may be easily implemented on portable devices and devices with low computational power. This should make ITS applications using ALPR less expensive from both a hardware and consumer standpoint, facilitate and expedite the advancement and creation of ITS technology, and make ITS applications more accessible for a greater percentage of society.

Chapter 5

Real-time CVSA Decals Recognition System Using Deep Convolutional Neural Network Architectures[1]

The Commercial Vehicle Safety Alliance (CVSA) aims to achieve uniformity, compatibility and reciprocity of commercial motor vehicle inspections and enforcement by certified inspectors dedicated to driver and vehicle safety. Commercial vehicles that pass a CVSA inspection are eligible for a decal representing a commitment to safety. In this chapter, a 2-step automatic CVSA decal recognition system using deep convolutional neural network architectures was proposed. The first step localizes a vehicle's windshield and the CVSA decal within, and classifies the decal colour. The CVSA decal is cropped and used as input to the second stage, which localizes and classifies a digit and the corner-cut of a CVSA decal. With the corner-cut, colour, and digit, the system can determine the decal's date of issue. This chapter is structured as follow Section 5.1 contains introduction; Section 5.2 describes the proposed architecture; Section 5.3 describes the labelling and training process for the proposed solution; Section 5.4 describes real-time processing using different platforms; results are in Section 5.5; and Section 5.6 concludes this chapter.

[1]The content of this chapter is originally published in IET Intelligent Transport Systems. The manuscript has been reformatted for inclusion in this thesis.

Juan Yépez (JY), Riel Castro-Zunti (RC), Younhee Choi (YC), and Seok-Bum Ko (SK) designed the study. JY designed the network architecture, trained and tested the models, implemented the system on the edge devices, and provided results analysis. RC helped annotated the images from dataset and proofreading the manuscript. JY prepared the manuscript with contributions from YC and SK to the manuscript structure, readability and analysis and discussion of the results

5.1 Introduction

4 million commercial vehicle inspections are conducted every year throughout North America to ensure the safe operations of vehicles on the road [105]. Specially trained safety inspectors in each state, province, and territory inspect commercial vehicles based on procedures, policy, and criteria developed by the Commercial Vehicle Safety Alliance (CVSA), a non-profit association that operates throughout the U.S., Canada, and Mexico and regulates and improves commercial vehicle safety standards [105].

The CVSA was established to encourage the collaboration of law enforcement, government, and industry to promote an environment free of commercial vehicle crashes, deaths, and injuries. This would be achieved by establishing effective transportation safety standards for motor carriers, drivers, vehicles, and inspectors through compliance, education, awareness, training, and enforcement programs.

A vehicle that passes inspection is issued a decal that is typically affixed to the windshield of a commercial vehicle. This decal contains information that reveals the date of issue.

The decal's color, as seen in Figure 5.1, indicates the calendar quarter in which the commercial vehicle was last inspected [105]:

Figure 5.1: CVSA decal color types.

- Green represents January through March.
- Yellow represents April through June.
- Orange represents July through September.
- White represents October through December.

The decal may have one or two corners cut, indicating the month of issuance [105]. The year of issuance is represented on the decal by a single digit (e.g., 2018 is indicated by the number "8"). The CVSA trademark is below the year.

Despite the CVSA's reach throughout North America, there are no commercial automatic CVSA decal recognition systems. In recent years, convolutional neural networks (CNNs) have brought impressive improvements to object detection projects [30], [106], [107]. Object detection locates and classifies regions of interest (ROIs) within images. However, detecting small objects remains challenging: for a neural network to have fast prediction time, it must have a relatively low input resolution (e.g. 300×300 px). Images input to the network must be downscaled to fit the network, meaning the loss of details important for object recognition further down the system's pipeline; this makes a one stage system impractical. A larger neural network input size may improve detection of small objects, but the subsequent processing time increase generally renders such systems too slow for real-time processing on commodity hardware. Though multiple networks can be chained, with one specifically focusing on small objects of interest, each stage increases network latency, thereby decreasing efficiency [108]. MobileDet [39] is the current state-of-the-art feature extractor for object detection models on mobile devices; thus, we chose it as the baseline/backbone and customized it to the tasks.

In this thesis, the following are proposed and presented:

A novel two-stage automatic real-time edge CVSA Decal Recognition System (CDRS). A two object detection architectures customized for detection and recognition tasks was designed: the first architecture focuses on small RoIs, specifically decals in the context of a vehicle's windshield; the second architecture provides fast prediction for digit

and corner-cut recognition. The two-stage system effectively reduces the number of false positives whilst achieving high accuracy.

A novel “7-spots” method for video stream prediction operating at the CDRS’s second stage. When a truck enters the field of view of a camera setup, it is relatively far away, making its CVSA decal in the first few frames appear small and blurry. The 7-spots method provides better performance by evaluating the last 7 detected decal images per-truck—when the decal is closer to the camera and thus larger and sharper—and doing majority voting. Moreover, this method reduces false positives by assuming <2 consecutive detections to be noise. It is also more efficient because the CDRS predicts only up to 7 decals (as one composite image) rather than all detected decals.

A performance comparison of the CDRS implemented using several popular edge hardware accelerators—with vendors including Nvidia, Intel, and Google—and demonstrated real-time performance therein. With high frames per second (FPS), the proposed CDRS detects trucks, decals, and provides decal information.

5.2 Proposed Architecture

The resolution of an input to an object detection network is oftentimes much smaller than the original image. Several object detection architectures, e.g. SSD [48], use an input size of 300 x 300 pixels despite the original image being 1280 x 720 pixels (high definition (HD)) or higher. Although an input with small resolution reduces the parameters and thus the processing time, details specifically from small objects are lost.

According to the MS COCO benchmark [109], objects with a resolution less than 32 x 32 pixels are defined as small, between 32 x 32 and 96 x 96 as medium, and greater than 96 x 96 pixels as large. Object detection networks struggle to detect small objects [110] because small objects have less pixels and cover a smaller area of the input. Fast and accurate CVSA decal detection is a difficult task because decals are small, and any input reduction would cause severe degradation in decal image quality.

This issue may be potentially mitigated by dividing detection into multiple stages that are processed sequentially [108]. E.g. detection pipelines that treat object proposals independently and predict bounding box locations and their classification scores separately [111]. Although such architectures have achieved good detection performance, this methodology is generally limited in that it causes delays and difficulties that are unacceptable and irreconcilable for real-time applications (>30 FPS) [112].

If a conventional multi-stage methodology is applied for the proposed CDRS, the system would require 5 stages: in the first stage, trucks or windshields are detected; in the second stage, the CVSA decal is detected; the third would classify the colour; the fourth would classify the corner cut; finally, digit detection and classification. The output of the first stage would be a bounding box that would be cropped and sent to the second stage, whose output would be the decal which would be cropped and used as input to stages three through five. After all stages are processed, the system can identify the month and year of the CVSA decal.

Though a CDRS system can be created using a conventional multi-stage methodology, the resultant five-stage design—with separate networks for localization (truck and decal) and each CVSA attribute (colour, digit, and corner cut)—is marred with inefficiencies. Having a new network at each stage increases both the total system software size and the system latency; such issues are compounded when using a hardware accelerator.

The proposed two-stage system uses custom task-tailored architectures based on a MobileDet backbone. The number and types of layers are selected after training and evaluating several backbone configurations using NAS [39]. NAS has demonstrated a superior ability to learn models that are both accurate and efficient on a specific hardware platform. The proposed models are hardware accelerator compatible and can run in parallel to reduce processing time compared to a five-stage system. MobileDet collects the feature maps at six different endpoints; two are consumed by the head, being processed and concatenated in parallel to generate location and classification values. The stages are as follows:

- Windshield detection, CVSA decal detection, and colour classification.
- Digit and edge detection and classification.

5.2.1 Windshield Detection, CVSA Decal Detection, and Colour Classification

The objective of the first stage is to mitigate the problem of the small input size. Thus, the standard SSD resolution was increased from 300 x 300 to 320 x 320 px. Although this cannot overcome the problem, it allows for better decal details.

The first layer of the customized backbone contains a CNN with stride 2; this downsamples the input without a max pooling layer and requires less computation than stride one [113].

Next, several Fused layers was used with stride one or two, and kernels 3×3 or 5×5. Tucker and Fused layers use regular convolutions better suited to feature extraction, especially important in the backbone's early stages; depthwise convolutions are less efficient at this task [39]. Finally, several IBN layers was used before the endpoints to improve sensitivity to small objects. Table 5.1 shows the first stage backbone. Each line describes a layer with kernel k, expansion e, repeated n times, with residual r, and stride s. C4 and C5 are endpoints into the head.

Because the truck is large and can occupy many consecutive video frames, the CDRS does not detect the truck itself; instead, the first network stage detects both the vehicle windshield and the CVSA decal. Thus, the approach differs from Yonetsu et al. [54] that detects the vehicle in their first stage; by detecting the windshield, the proposed CDRS implicitly determines the presence of a truck. This helps the system reduce latency compared to [54]. Additionally, the CDRS's first stage determines the colour of the CVSA decal, streamlining the design and improving its functionality. The proposed backbone has five classes: the windshield; and four for decals of varying colours that indicate the calendar quarter. Though small object colour classification is generally difficult, high

Table 5.1: First stage backbone

Input	Layer	k	E	N	R	s
320^2 x 3	Conv	3×3	N/A	1	No	2
160^2 x 32	Tucker	3×3	0.25-0.75	1	No	1
160^2 x 16	Fused	3×3	8	1	No	2
80^2 x 16	Fused	3×3	4	1	Yes	1
80^2 x 16	Fused	3×3	8	1	Yes	1
80^2 x 16	Fused	3×3	4	1	Yes	1
80^2 x 16	Fused	5×5	8	1	No	2
40^2 x 40	Fused	3×3	4	3	Yes	1
40^2 x 40	IBN	3×3	8	1	No	2
20^2 x 72	IBN	3×3	8	1	Yes	1
20^2 x 72	Fused	3×3	4	2	Yes	1
20^2 x 72	IBN	5×5	8	1	No	1
20^2 x 96	IBN	5×5	8	1	Yes	1
20^2 x 96 (C4)	IBN	3×3	8	2	Yes	1
20^2 x 96	IBN	5×5	8	1	No	2
10^2 x 120	IBN	3×3	8	1	Yes	1
10^2 x 120	IBN	5×5	4	1	Yes	1
10^2 x 120	IBN	3×3	8	1	Yes	1
$10^2 \times$ 120 (C5)	IBN	5×5	8	1	Yes	1

mAP results for this task are achievable because decals only have four colours that are relatively easily differentiated.

5.2.2 Digit and Corner-cut Detection and Classification

The output of the first stage is the localized decal as cropped from the originally inputted frame. This decal crop is the input to the first stage, which detects and classifies the decal's digit and corner-cut. The locations and sizes of the digit and corner-cut are generally standardized and are large relative to the whole decal. Thus, the backbone can be simpler and focus on larger objects.

Table 5.2 shows the second stage backbone. The first 3 layers of the customized backbone are similar to the first stage, to extract the most important features from the input image. Repeated layers were removed, and no layers have a residual function; this discards details for small objects, which is not necessary in this stage. The proposed architecture is thus much faster than its baseline with no precision loss; this illustrates the parameter-wise superiority of slimmer custom architectures compared to deep general ones. This backbone has 7 layers, compared to 23 in that of first stage. The proposed network localizes the top part of the decal and identifies the corner-cut, which could be one of three classes: no corner cuts; one corner is cut; and both corners are cut. The remaining classes correspond to digit recognition (0 through 9).

Table 5.2: Second stage backbone

Input	Layer	k	E	S
320^2 x 3	Conv	3×3	N/A	2
160^2 x 32	Tucker	3×3	0.25-0.75	1
160^2 x 16	Fused	3×3	8	2
80^2 x 16	Fused	5×5	8	2
40^2 x 40	IBN	3×3	8	2
20^2 x 72 (C4)	IBN	5×5	8	1
20^2 x 96 (C5)	IBN	5×5	8	2

5.3 Labelling and Training

The dataset used in this work was provided by International Road Dynamics Inc. (IRD) [114]. The dataset contains 5869 still images and one 30-minute video with 151 trucks (17 with the absence of a CVSA decal) recorded at a North American commercial vehicle check stop. Figure 5.2 shows the two main types of images collected. Both Figure 5.2 (a) and Figure 5.2 (b) show the front part of the truck and were captured in daylight conditions. The CVSA decals have few rotations or skews. Several CVSA decals are blurry, making it difficult to read their digits.

(a)

(b)

Figure 5.2: Trucks with CVSA decals on different highways.

Because the dataset contained a relatively small number of images, it was explicitly inflated using data augmentation techniques. Histogram equalization of the Y channel in YUV colour space was performed, different zoom levels were (e.g. 2x, 4x, etc.) applied to the decals, and decals were translated to other areas within the image. Dataset augmentation brought the dataset to 6083 images for training and testing. This augmentation increases representation and may contribute to less overfitting. A random 80% of the images for training and 20% for testing were used. Images were resized to 320×320 px for training and testing.

The graphical image annotation program "LabelImg" [92] was used to draw a bounding boxes around each ROI and assign each box a class label; a sample annotation can be observed in Figure 5.3. Annotations are saved as PASCAL VOC-conforming XML files and are used to generate TFRecords, which store a sequence of binary records in a way that allows for the efficient import (from TensorFlow's perspective) of annotated image data.

The networks were trained from scratch to 50,000 steps using TensorFlow 1.15 on a Tesla K40c GPU in an Ubuntu 18.04 LTS computer. First and second stage networks were trained with the same hyperparameters: a batch size of 16; categorical cross-entropy loss; and a stochastic gradient descent optimizer with an initial learning rate of 0.8, lowered to 0.013 after 15,000 steps, with 0.9 momentum, 0.97 decay, and 0.001 epsilon.

Finally, because CVSA decals are typically much smaller than objects in the COCO dataset [109], the scale of the smallest anchor boxes was reduced to 2 in the first network.

5.3.1 First Stage

In the first stage, the truck's windshields and the CVSA decals from the augmented dataset were labelled. The colour of the decals that represent the calendar quarter is also annotated. The annotated dataset was used to train the network shown in Table 5.1. Once the proposed network is trained, the resulting model can predict windshields and decals at the same time.

Figure 5.3: Windshield and CVSA decal labelled using the LabelImg program.

If bounding boxes for a vehicle's windshield and a CVSA decal are predicted via the inference process, their locations are used to determine if the decal is located within the windshield. If they are, the predicted CVSA decal is assumed to be valid. Else—as is the case for objects located elsewhere on the truck/road that are visually similar to decals—the prediction is discarded. The decal's predicted bounding box is cropped from the original image and inputted to the second stage.

Because a decal occupies less than 1% of the input image area, lane markings or other vehicle decorations can be confused as CVSA decals. Confirming that a predicted CVSA decal is within a vehicle's windshield is an effective measure to reduce the number of potential false positives.

Detecting a windshield is a practical way to determine the presence of a vehicle from streaming video: if a windshield is detected, a vehicle must also be there. The CDRS saves each frame that contains a detected windshield in an image array. If a CVSA decal is also detected, a "Found" status is set therein; else, it is set to "Not_found". After the truck passes (i.e. the end of consecutive frames containing a windshield), the CDRS saves (as

an image file) the "Found" frame located closest to the middle of the array; empirically, this middle image was determined to contain the windshield centred in the frame, which provides the best view of the truck (for validation purposes and/or future work).

5.3.2 Second stage

For the second stage, the year-identifying digit and corner cut were annotated from cropped decals provided by the first stage. These annotated images were used to train the network shown in Table 5.2. Figure 5.4 shows a sample second stage annotation.

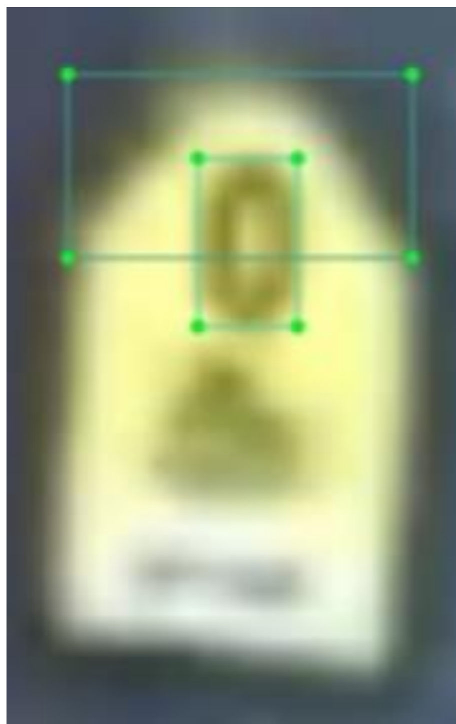


Figure 5.4: Digit and corner-cut labelled in a CVSA decal.

Integrating custom digit detection into the second stage has two main benefits over utilizing a third network specifically for optical character recognition (OCR, e.g. tesseract [115]). First, as aforementioned, additional networks produce a greater software model file size and additional latency. Second, as can be noted in Figs. 4 and 5, the digit appears almost blurred due to its small resolution; a network custom-trained on such

representations could handle blurred images more effectively than a generic OCR network trained without such considerations.

From the detected information, the date of decal issuance can be found. Using Figure 5.4 as an example, the system can use the information of the 0 digit, the 2 top corners cut, and the yellow decal colour to determine that the decal was issued in April 2020.

The trained model can successfully and practically predict still images. However, when a video stream is processed, the same decal is detected multiple times in consecutive frames, leading to potentially redundant prediction computation. This leads to wasted resources and long processing times, especially when a CPU is used or when the model is deployed unoptimized on commodity hardware. A naive solution is to choose only one decal to be processed, but this could be a problem if the selected decal is blurry or otherwise unideal; this tends to happen during the first frames that a decal appears in the video, when the truck is far and the decal is very small.

To mitigate the video stream processing time and turn the redundancy into an asset, a model was proposed that receives an input image comprised of up to 7 consecutive decal detections. Each decal in the input image has a size of 60×95 px. Once a decal is detected in the first stage, one of the 7 “spots” in the composite decal image are filled. If another decal is detected within the next 15 frames, this decal occupies the second spot; the procedure continues in this way, potentially overwriting earlier spots, until no decal is detected after 15 frames. If at least 2 spots are filled, the 7-spots image is inputted to the second stage model; else, the detection is assumed a false positive and is not predicted, which further reduces the potential for false positives. Note that 7 spots and 15 frames were chosen as thresholds by measuring the average frames a decal was present and the average frames between decals in the 30-minute video in the dataset. Figure 5.5 shows some example 7-spots input images.

Figure 5.5 (a) shows a CVSA decal issued in April 2020 which was detected at least seven times; its detections occupy all available spots. Figure 5.5 (b) shows a CVSA decal

issue in January 2020 where only five detections are found. Figure 5.5 (c) shows a CVSA decal issue in October 2019 with only three detections.

To train the model used for video stream prediction, 51 7-spots images was used, like those shown in Figure 5.5, collected from the 30-minute video. Additionally, 5869 7-spots images were synthesized using decals from still images by randomly duplicating the decal to fill 3 to 7 spots. These images are used to train the network shown in Table 5.2. Detection and classification are performed over the filled spots, and the inference time is the same regardless of the number of spots filled. A majority vote over detections is used to determine a decal's final qualities. If there is a tie, the value from the last decal is selected.

Figure 5.5: **(a)** 7 spots filled, **(b)** 5 spots filled, and **(c)** 3 spots filled.

5.4 Real-time Prediction

The trained custom models was exported as frozen inference graph files (.pb). However, frozen graphs are optimized for GPU deployment, rather than for all platforms. Thus, to improve prediction speed, the models were converted to the appropriate native framework for each hardware accelerator.

The Google Coral platform uses TensorFlow Lite (TFLite), the lightweight version of TensorFlow specifically designed for mobile platforms and embedded devices. It provides lower latency and a smaller binary size but tends to degrade accuracy—ideally, this degradation is negligible or offers a competitive speed-accuracy trade-off. TFLite supports a set of core operators tuned for mobile platforms, the desired medium for inference. First, models must be pruned [116] and quantized from FP32 to INT8 Then, models are converted to TFLite file format. Next, the Edge TPU compiler was used to rearrange layer weights from the TFLite file to a new compatible format. The compiler shows the layers that can perform inference on the Coral vs. the CPU.

The Edge TPU has 8 MB of SRAM. A small amount of the RAM is reserved for the model's inference executable, and the remaining space is used to cache the model's parameter data. This enables faster prediction speed compared to fetching the parameter data from external memory.

For Intel's Movidius-based hardware, their Open-VINO toolkit was used which facilitates both the optimization of a deep learning model from a framework and the deployment of the model onto Intel hardware using an inference engine. It enables deep learning inference at the edge and supports heterogeneous execution across a variety of computer vision accelerators: CPUs, GPUs, Intel Movidius NCSs, and FPGAs. It supports more deep learning models out of the box than Google Coral.

The Intel Inference Engine (IE) enables the deployment of the Tensorflow-trained models. Rather than using the original model for inference, the IE uses its Intermediate Representation (IR), which is optimized for execution on endpoint target devices. The

Intel Model Optimization Tool was used to generate the IR, comprised of two files (.xml and .bin) for each trained model. The USB stick-based Intel NCS and NCS2 supports Half Precision Floating Point (FP16) [117].

NVIDIA's Jetson Nano and Jetson Xavier, small AI computers for developers, were benchmarked. The models were deployed in two ways: using regular TensorFlow (with GPU support); and first optimizing them using NVIDIA's TensorRT framework. TensorRT includes a deep learning inference optimizer and runtime engine for production deployment. TensorRT optimizes the original model by combining layers, optimizing kernels, pruning, and quantization. The models can be quantized using FP16 or INT8. Depending on the available resources of the Jetson, the framework runtime engine generates a file which improves latency, throughput, power, efficiency, and memory consumption.

The Jetson Nano is a standalone computer and does not require additional hardware. The Coral USB accelerator and Intel's Movidius hardware requires a computer for proper operation; they were paired with a Raspberry Pi 4 (RPi4) single-board computer. The RPi4 has 4GB of LPDDR4 RAM, a 1.5 GHz Broadcom quad-core processor, two USB 3.0 and two USB 2.0 ports, two micro-HDMI video outputs, a gigabit Ethernet jack, and radios for 802.11ac Wi-Fi and Bluetooth 5.0. The USB 3.0 port was used to achieve maximum speed with the accelerators.

The high-level programming language Python is very popular for data science, deep learning training, and model deployment. Python can be run interactively—a big advantage for vision or image processing applications. However, compared to C/C++, Python programs typically run slower, especially when a single board computer like the RPi4 or Jetson is used. Furthermore, library bindings for Python are usually less mature than C/C++. For those reasons, in this thesis C/C++ was used for model prediction.

To further increase the inference speed, the open-source pipeline-based multimedia framework "GStreamer" was used, able to link a variety of media processing systems in complex workflows. Using GStreamer, a system can be built that reads frames in different

formats and even from different sources in parallel, process them, and export them to a file or stream them over a network. For the Jetsons, NVIDIA developed plugins for GStreamer including inferencing using TensorRT and encoding/decoding video streams using the hardware accelerator (NVDEC/NVENC). DeepStream is an NVIDIA plugin for GStreamer, and part of their GStreamer analytics SDK. DeepStream allows the entire pipeline to be processed on the GPU, with zero memory copy between the CPU and GPU; this makes the entire pipeline faster and more efficient.

Figure 5.6 shows a screenshot of the proposed two-stage with 7-spots system as implemented on a Google Coral USB Accelerator attached to a RPi4. As can be noted in the top left corner, the video stream is being processed in real time.

Figure 5.6: Real-time CVSA Decals Recognition Systems (CDRS)

5.5 Results

5.5.1 Model Comparison

The proposed CDRS method consists of two stages. The upper half of Table 5.3 compares the complete proposed custom edge solution against those formed by state-of-the-art detection architectures at the edge. Whereas the solution uses a different architecture at each stage, state-of-the-art comparison work is trained using the same model architecture at each stage. I.e., at each stage, the solution uses a different model with a different number of parameters, MAdds, file size, and prediction time; but, for other architectures, these values are the same at each stage. Thus, for reference, the bottommost 2 rows of Table 5.3 shows the values of each of the custom stages independently. Also shown in Table 5.3 are Stage Two results from both single (still image prediction) and 7-spots (VOD prediction) models.

The model was evaluated using Mean Average Precision (mAP) with a prediction Intersection Over Union (IoU) ≥ 0.5 (@0.5), a commonly used benchmark in object detection e.g. for the PASCAL VOC challenge [118]. Figure 5.7 shows mAP@0.5 comparison results. The proposed Stage One model achieves the highest mAP@0.5—98.5%—due to added layers for greater representational power for small objects. Although this means the Stage One model has more parameters and a larger size compared to other SSDLite-based models (though less than MobileNetV1 that uses SSD), it requires only 0.3 ms more time prediction time than MobileDet EdgeTPU, which has an inference time of 7.3 ms (half of 14.6 at each stage). This speed-mAP trade-off is very fair and ensures the model's competitiveness for mobile and edge device implementations.

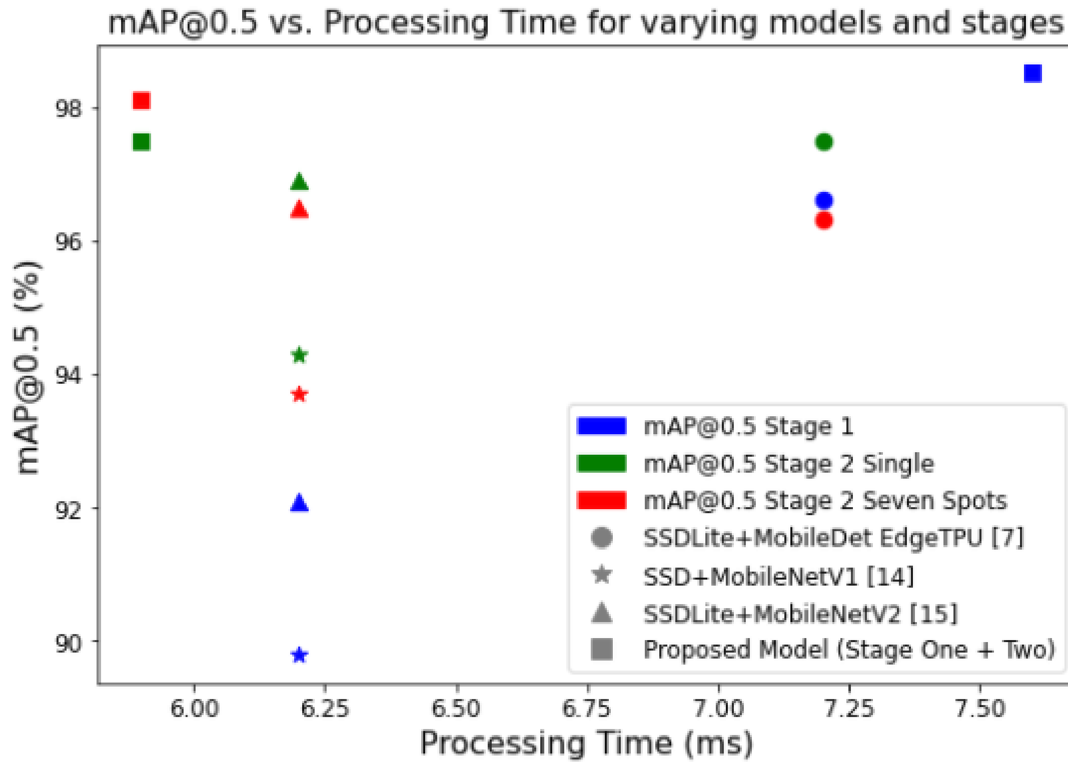


Figure 5.7: Comparison of accuracy (mAP) vs. processing time by stage

Table 5.3 also shows that the proposed Stage Two models achieve 97.5% mAP@0.5 for single image and 98.1% mAP@0.5 for VOD (7-spots) prediction. The propose model has the same single image accuracy as MobileDet Edge TPU but uses 17.4% less parameters for that stage. Moreover, the proposed model is 0.6% more accurate than SSDLite+MobileNetV2 for Stage Two 7-spots. The proposed model has 130,000 to 5.33M less parameters, corresponding to a 31.6-41.6% lower MB file size than other models. Though SSD+MobileNetV1 and SSDLite+MobileNet V2 have slightly lower overall prediction times, the proposed model is 5.4-7.7% more accurate in Stage One and 0.4-3% more accurate for Stage Two single image prediction.

The file size of a regular MobileDet model is 5.04 MB. Using this architecture at each stage, the file size would be 10.08 MB, as seen in Table 5.3. Similarly, a two-stage SSD+MobileNetV1 occupies 11.04 MB and SSDLite+MobileNetV2 would occupy 9.42 MB. Even worse, a hypothetical conventional multi-stage 5-stage system based on

MobileDet would have a file size of 25.20 MB. All these exceed the ~8 MB of SRAM in the Coral USB Accelerator; unlike the PCI accelerator variant that shares the host computer's memory, the USB accelerator slowdown would be particularly impactful because loading from external memory would be required. Conversely, the custom models together occupy only 6.64 MB, which fits within the SRAM and thus makes the processing time faster than other networks. This shows the custom model's superiority to other work when deployed on the Coral USB Accelerator, and potentially similar SRAM-limited edge hardware.

Table 5.3: Summary of model training and results

Model	**Input Image (W×H×3)**	**MAdds (B)**	**Params (M)**	**File Size (MB)**	**mAP @0.5 Stage One (%)**	**mAP@0.5 Stage Two (%)**		**Per-Image Inference Time (ms)***
						Single	**7-spots**	
SSDLite + MobileDet EdgeTPU [39]	320×320	3.06	8.40	10.08	96.6	**97.5**	96.3	14.6
SSD + MobileNetV1 [16]	300×300	2.40	13.60	11.04	89.8	94.3	93.7	12.4
SSDLite + MobileNetV2 [80]	300×300	**1.60**	8.60	9.42	92.1	96.9	96.5	12.4
Proposed Model (Stage One + Two)	320×320	2.15	**8.27**	**6.44**	**98.5**	**97.5**	**98.1**	13.5
Stage One Only	320×320	1.77	4.80	5.20	98.5	—	—	7.6
Stage Two Only	320×320	0.38	3.47	1.44	—	97.5	98.1	5.9

* Refers to time on the Google Coral. Note that where the File Size is >8MB, the Per-Image Inference Time is << total Processing Time due to SRAM fetching.

5.5.2 Hardware Accelerators Result Testing

Table 5.4 shows the average prediction times of the system measured using different hardware implementations. Each system was tested using a 30-minute video with frame resolution 1280 x 720 pixels, h.264 compression, a bit rate of 1385 kbps, and streamed at 60 FPS. The proposed CDRS was implemented on the Jetson Nano and Jetson AGX Xavier using both standard TensorFlow model files and TensorRT (TRT) files. A RPi4 was also paired with the hardware accelerators Google Coral, NCS, and NCS2. Finally, results were provided of the RPi4 alone using both the standard TensorFlow model files and the optimized TFLite files, the latter built using C++ and OpenCV with VideoCore 6, NEON register, and FP16 support.

Implemented on an Nvidia Jetson AGX Xavier with DeepStream, the proposed system achieved 173.31 FPS—higher than other platforms. The Tensor cores and the GStreamer APIs allow the entire pipeline to run on the AGX Xavier's Volta GPU, lessening prediction time. TensorRT was used for quantization. Also, the system deployed was tested on the AGX Xavier's FP16 deep learning accelerator (DLA); this achieved 77.10 FPS. Although the performance is less than using the AGX Xavier's GPU, it is higher than other hardware accelerators. Finally, the model was tested with FP32 using TensorFlow directly (i.e. no TensorRT optimization); the performance was comparatively poor at 17.54 FPS.

Using the two DLAs and GPU in parallel, the Jetson Xavier can achieve 327.51 FPS. This was tested by adding the stream mux plugin to the DeepStream pipeline, allowing the processing of multiple inputs sources simultaneously—four 60 FPS video streams were used as input. An object tracker plugin was used to run the primary and secondary detectors in parallel. Also, two probe plugins was added: one to control the detection using metadata, and the other to save images of the detected trucks and decals using OpenCV. Finally, a tiler plugin was added to show multiple inputs on the same screen. Figure 5.8 shows the pipeline used for the CDRS on the Jetson Xavier.

Table 5.4: Hardware accelerator benchmark

Device	Bits	Stage One			Stage Two (7-spots)			FPS*
		Pre-process (ms)	Inference (ms)	Post-process (ms)	Pre-process (ms)	Inference (ms)	Post-process (ms)	
NVIDIA Jetson Xavier (GPU)	INT8	**2.30**	**1.24**	**2.23**	**2.11**	**1.16**	**2.15**	**173.31**
NVIDIA Jetson Xavier (DLA)	FP16	**2.30**	7.94	**2.23**	**2.11**	6.16	**2.15**	77.10
Coral USB Accelerator	INT8	4.35	7.64	4.45	4.88	5.91	3.89	60.83
NVIDIA Jetson Nano (TRT)	FP16	3.50	29.6	3.10	3.38	23.9	2.96	27.62
Intel NCS2	FP16	5.85	29.19	5.45	5.34	28.98	4.72	24.70
NVIDIA Jetson Xavier	FP32	4.75	48.75	3.51	4.31	39.91	3.18	17.54
Intel NCS	FP16	6.19	107.06	5.73	5.56	88.48	4.86	8.43
NVIDIA Jetson Nano	FP32	6.87	105.21	6.54	6.25	84.85	5.78	8.37
Raspberry Pi 4 (TF Lite)	FP16	6.51	144.57	5.91	5.87	120.48	5.10	6.37
Raspberry Pi 4	FP32	9.07	260.26	8.45	8.34	218.70	7.19	3.60

* Because Stage Two runs only once per truck, the average speed is limited by Stage One which acts as the pipeline's bottleneck.

The Jetson AGX Xavier has the best performance for the proposed system, and can run four simultaneous inputs real-time and without lag. However, the AGX Xavier's commercial price is US $699, making it the most expensive solution.

The RPi4 with the Google Coral USB accelerator achieves 60.83 FPS. The Coral USB accelerator and the RPi4 cost US $59.99 and US $55, respectively—a very competitive price to inference speed compared to the other options.

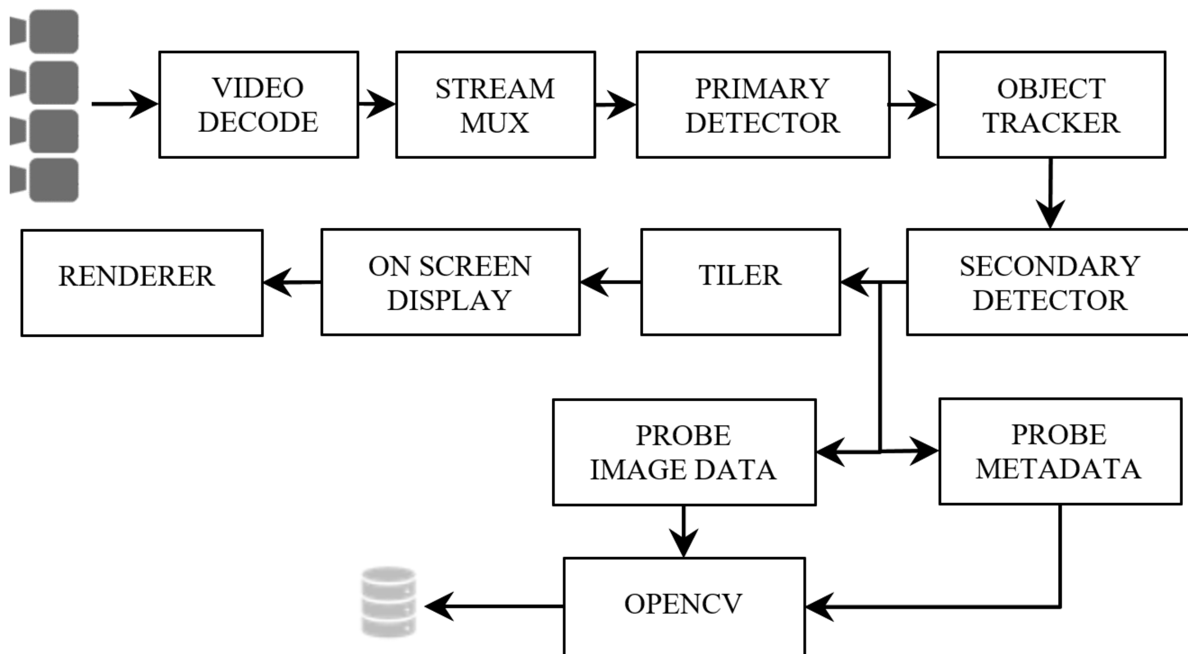


Figure 5.8: DeepStream pipeline for the real-time CDRS on the Jetson Xavier

The Nvidia Nano has a price (US $99) lower than using the RPi4 with any USB hardware accelerator. On this system the models achieve 27.62 FPS. Thus, it is better and less expensive than the NCS2 on the RPi4 with a cost of US $69 and US $55, respectively. However, the Intel NCS has a slightly higher prediction speed (8.37 FPS) than the unoptimized Jetson Nano system. This is an example of the importance of optimizing the platform to the available resources of the accelerator.

As presumed, the RPi4 alone has the worst prediction speed (3.60 FPS). However, optimizing the TFLite model file with the NEON register and FP16 support increases the inference speed by 77%—close to the same inference speed as the unoptimized Jetson Nano.

Deep learning prediction is the most time-consuming process through the proposed system pipeline. The second stage 7-spots method for video stream prediction enhances the efficiency of a video-input system because only one image (the 7-spots composite decal image) per truck is predicted. The alternative would be predicting each detected decal image individually, which can be upwards 12 decals per truck. Thus, the 7-spots

method reduces the overall time to generate results after a truck passes beneath the camera. Because Stage Two is less computationally complex, faster, and occurs only once per truck, it can be run entirely on the CPU, thus allowing the GPU's compute capacity to be focused on predicting the high FPS video input (and thus high required throughput) of Stage One. Running each stage on separate hardware creates effective parallelism; nevertheless, Stage One predicts every frame and becomes the system's bottleneck, limiting overall FPS.

The proposed CVSA decal recognition system was successfully deployed in Cordelia, California, using a Jetson Xavier. The system was equipped with a camera with shutter speed 1/5000s; a high shutter speed reduces the decal's blur in the video stream, especially important for vehicles travelling highway speeds. For the system to work in low light and at night, an illuminator was installed; special consideration to the lighting angle was given to avoid blinding drivers. The proposed system, deployable on a variety of edge devices, is a lower cost solution compared to systems that require high bandwidth internet connectivity for cloud/server processing. Additionally—free of the risks associated with uploading sensitive information to third party cloud platforms—the solution is safer and more privacy-protecting. Finally, the proposed system can easily be integrated to other systems through IoT messaging protocols like MQTT or AMQP.

5.6 Summary

In this chapter, the first real-time two-stage CVSA decal recognition system using deep convolutional neural networks was presented. The first stage—custom-tailored to locate small objects, such as decals—localizes a vehicle's windshield and the decal therein, and determines the decal's colour; this method lowers the system's false positive rate by removing decal candidates not within a windshield. Because a truck can be present in multiple frames, the method is also more robust than systems that first simply detect the truck. The next stage—customized with far fewer parameters because the task is easier—localizes the decal's digit and corner-cut. This, along with the colour, can determine the decal's date of issue, thereby recognizing the CVSA decal. The proposed

system can predict from still images and/or a video stream. The custom architectures demonstrated high average precision: 98.5% mAP for Stage One and 97.5% mAP for Stage Two single image. A second stage 7-spots model was presented to predict multiple frames of the same CVSA decal from a video stream, using majority voting to provide a more accurate result. This model achieved 98.1% mAP; though this is 0.6% more than SSDLite+MobileNetV2 (the best model), the proposed model has an inference time of 5.9 ms, 0.3 ms faster than SSDLite+MobileNetV2 at that stage. The CDRS was evaluated using hardware accelerators from different vendors like Intel, Google, and Nvidia. Finally, it is showed that the proposed model's stages can run in parallel and achieve an inference speed of 173.31 FPS on the Jetson AGX Xavier and 60.83 FPS on the inexpensive RPi4 + Google Coral system. The Jetson AGX Xavier is recommended for ultra-high-speed time-critical applications, and the RPi4 + Google Coral system for "fast enough" consumer-grade tasks.

Chapter 6

Real-Time Deep Learning-based Edge System for HAZMAT Recognition[1]

Hazardous materials (HAZMATs) are commonly transported by commercial vehicles. The nature of a HAZMAT is indicated on a vehicle by a specific placard, usually on the vehicle's front or sides. To the best of the author's knowledge, the proposed system is the first real-time deep learning-based edge HAZMAT placard recognition system for complex outdoor environments. A three-stage cascading system using deep learning networks was designed. The first network localizes and classifies the HAZMAT placard. If the placard contains a United Nations (UN) / North American (NA) number, the second network localizes that number and identifies the nature of the substance. The third network recognizes the UN/NA number. This chapter is structured as follows: Section 6.1 contains the introduction; Section 6.2 describes the models and proposed solutions; Section 6.3 covers the dataset and utilized training augmentations; Section 6.4 details the methodology, including training environment, models and model selection/development, and training parameters; Section 6.5 elaborates on the methodology for real-time deployment on edge hardware; results are given in Section 6.6, including accuracy, processing time, model deployment on edge systems, and performance comparisons against other models and related work; Section 6.7 discusses the implications and limitations of the results and possible real-world ramifications of the research; and Section 6.8 concludes this chapter.

---

[1]The content of this chapter is originally published in Springer Machine Vision and Applications. The manuscript has been reformatted for inclusion in this thesis.

Juan Yépez (JY), Riel Castro-Zunti (RC), Younhee Choi (YC), and Seok-Bum Ko (SK) designed the study. JY designed the network architectures, trained and tested the models for stage two and three, implemented the system on the edge devices, and provided results analysis. RC designed the network architecture and trained stage one, annotated the images from dataset, and proofreading the manuscript. JY prepared the manuscript with contributions from YC and SK to the manuscript structure, readability and analysis and discussion of the results.

6.1 Introduction

Hazardous materials (HAZMATs), defined by the United States Department of Transportation (USDOT), are "capable of posing an unreasonable risk to health, safety, and property when transported in commerce" [119]. The U.S. Department of Labor Occupational Safety and Health Administration (OSHA) further categorizes HAZMATs, which includes physical hazards (e.g. explosives and flammables) and health hazards (e.g. acute toxicity and skin corrosion) [119]. The HAZMAT standards set by the OSHA follow the United Nations Globally Harmonized System of Classification and Labelling of Chemicals (GHS), the universal standard describing the appearance and purpose of HAZMAT symbols, and for what chemicals a certain symbol is valid [120].

Although regulations that govern where and how a symbol should be placed on a vehicle differ by country (e.g. [119] for the U.S. and [121] for Canada), typical placarding includes a vehicle's sides and ends. This consistency allows for the automation of HAZMAT recognition in (semi-)controlled (e.g. check stop) and on-road environments using well-placed sensors, like cameras. Such a system in this context falls within Intelligent Transportation Systems (ITS), services that improve the driver experience and the safety of everyone on the road [107].

There are many applications of HAZMAT recognition: streamlining commercial vehicle check stops; dangerous goods tracking; emergency management, e.g. when a vehicle carrying hazardous materials crashes and the public must be notified; and, in future end-to-end smart cities and highways, i.e. where trucks themselves are (near) driverless. The task has also found niche development via the World RoboCup Rescue League; robots enrolled in the competition must be able to detect hazards, including HAZMAT placards [122].

This chapter details the research, results, discussions, and conclusions during the development of a fast and accurate real-time end-to-end deep learning (DL) HAZMAT recognition edge solution. It is an emerging system capable of deployment in real-world, complex, outdoor environments. The following contributions were proposed:

(1) A speed-accuracy assessment of popular single-pass DL object detection architectures over a dataset of HAZMAT placards on commercial vehicles acquired at a check stop.

(2) A custom placard detection and classification model adapted from that which had the best speed-accuracy trade-off (the "baseline"); the proposed model better suits the problem and dataset by customizing the baseline's latter layers. The models demonstrate to be robust in a variety of circumstances.

(3) An end-to-end HAZMAT placard recognition system that uses a pipeline of three stages: localization via the model achieved in contribution 2; localization and recognition of the HAZMAT class digit and localization of the United Nations (UN) / North American (NA) number via a similar custom network; and segmentation-free UN/NA number recognition.

(4) Quantize-aware training results.

(5) Processing time results, of the proposed models deployed on varying edge hardware, including from Nvidia, Intel, and Google. The end-to-end system's real-time (> 30 FPS) performance is demonstrated.

The research bridges a knowledge gap in specifically DL-based and complex-environment HAZMAT recognition, as well as provides a reasonable and realistic framework for fast, low-power edge DL deployment at check stops and elsewhere.

6.2 Proposed Solution

A three-stage HAZMAT recognition system was proposed, as shown in Figure 6.1. The pipeline involves detecting and classifying the placards in the frame using an object detection deep learning model; using a second deep learning object detection model to localize and recognize the placard's class digit, and localize the UN/NA number, if present; finally, recognizing the UN/NA number via a lightweight sequence classification model.

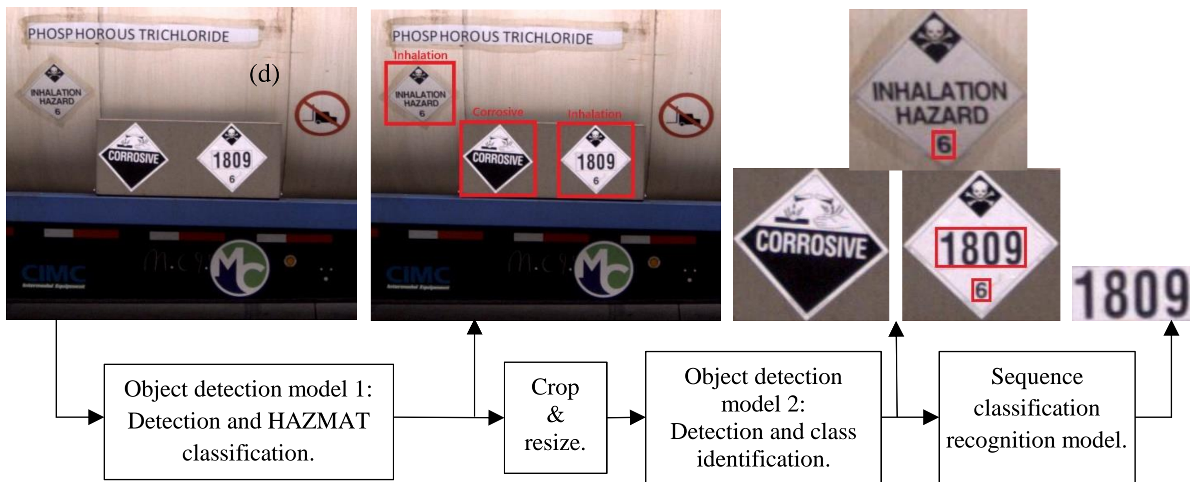


Figure 6.1: HAZMAT recognition system.

6.2.1 HAZMAT Placard Localization and Classification

An object detection architecture is used in the first stage to detect and classify HAZMAT placards. Although there are a variety of object detection architectures that can provide high accuracy, most require an expensive GPU or cloud services to process the input images or video. ITS applications (such as HAZMAT recognition) tend to be deployed on the road, and may be deployed in rural areas or where internet access is limited; this complicates systems that may otherwise rely on local servers or cloud services for data processing. Therefore—especially to reduce costs whilst preserving real-time functionality—the implementation of ITS applications via edge computing is critical. Processing image/video inputs at the edge enables the transmission of only HAZMAT

class/substance information, rather than a whole (expensive) video stream, to the destination server or cloud center. For this system stage, an edge-capable custom object detection architecture able to run in real time (> 30 FPS) on various edge hardware is proposed.

After evaluating a variety of object detection architectures, the edge deployment-ready SSDlite + MobileDet EdgeTPU model was chosen to have the best trade-off between speed and full precision validation set mAP@0.5. The architecture was subsequently adapted to produce a higher-performing model over the validation set. Moreover, validation mAP@0.5 results for the quantization-aware-trained SSDlite + MobileDet EdgeTPU model lacked in comparison to its full precision counterpart; because quantization is essential for some edge hardware, developing a custom feature extractor with better quantization performance over the dataset was paramount.

Like MobileDet [39], the design uses inverted bottleneck (IBN), fused, and tucker blocks. Many blocks employ residual ("skip") connections in which a block's final output is the sum of its input and its last layer's output. Stride 2 convolution is used for downsampling. Though information on network blocks can be found in [39], and the reader is referred to [39] for visual representations, a short summary is presented below.

IBN. A pointwise and depthwise convolution, each with RELU6 activation, followed by a "linear bottleneck" (a pointwise with identity activation) for depth changes.

Fused. A regular convolution with RELU6 activation followed by a linear bottleneck. IBN and Fused blocks have an "expansion" coefficient (E) which scales the within-block feature map depth to be $E\times$ the input channels (i.e. the depth of the output of the previous block). The block's linear bottleneck tapers this expansion to the desired output depth.

Tucker. A pointwise convolution with RELU6 activation, whose depth is the number of input channels scaled by a coefficient denoted the "input rank" (SIR), followed by a regular convolution with RELU6 activation, whose depth is the final output depth scaled by an "output rank" coefficient (SOR), followed by a linear bottleneck. SIR and SOR are

< 1. The resultant depths from scaling by SIR or SOR (collectively denoted S) is found in (1), where Cin and Cout are the numbers of pre- and post-scaled filters, respectively.

$$C_{\text{out}}(C_{\text{in}}, S) = \max(8 \times \left\lfloor \left\lfloor \frac{C_{\text{in}} \times S}{8} \right\rfloor + 0.5 \right\rfloor, 8) \tag{6.1}$$

Using NAS [123] with the HAZMAT dataset, the final custom architecture was achieved. It is paired with the SSDlite object detection network.

6.2.2 UN/NA Number Localization and Class Recognition

The input to the second object detection model is the HAZMAT placard(s) cropped from the localization information generated by the first model. It detects the presence of a UN/NA number and/or the bottom class digit. In Fig. 2, the vehicle has three HAZMAT placards, two "Inhalation Hazard" placards and one "Corrosive" placard.

Because the Corrosive placard contains no class digit or UN/NA number, no further processing is required. One Inhalation Hazard placard has only the bottom class digit, so only further prediction/classification is required in the second stage. The final placard has both the bottom class digit and a UN/NA number; these would be localized and the class digit recognized by the second-stage model, and the UN/NA number would be recognized by the third-stage model.

The localization and recognition task for the second-stage model is less complicated than that for which the first-stage model is designed. This is because a placard's bottom digit and potential UN/NA number are large relative to the input image (the placard), and the locations within the placard are relatively uniform. Moreover, there are less classes to recognize (9 for the digits + 1 for the UN/NA number), meaning there is a potentially greater distance between classes in feature-space. Thus, the feature extractor ("backbone") for this stage can be simpler, which decreases prediction time.

After evaluating several backbone configurations via NAS, the types and organization of layers were selected. The first 3 layers of the customized backbone, similar to the first

stage, extract the most important features from the input image. Repeated layers were removed, and no layers have a residual function. This backbone has only 8 layers, compared to 22 in that of first stage. Table 6.1 shows the backbone architecture.

Table 6.1: Neural Network Architecture for UN/NA Number Detection and Class Identification Feature Extractor

Operator	Scaling Factor	Output Channels	Kernel	Stride
Conv2d+RELU6	N/A	32	3×3	2
Tucker	S_{IR}=0.25, S_{OR}=0.75	16	3×3	1
Fused	E=8	16	3×3	2
Fused	E=8	40	5×5	2
IBN	E=8	72	3×3	2
IBN	E=8	96	5×5	1
IBN*	E=8	120	5×5	1
IBN*	E=8	384	5×5	1

* denotes the block's output is an endpoint into the SSDlite architecture

6.2.3 UN/NA Number Recognition

Recognizing a placard's UN/NA number can be performed via several approaches, such as the popular text-from-image technique of Optical Character Recognition (OCR) [115]. However, because each character should be individually recognized lest false positive detections be generated from noise between characters, OCR approaches require high computation.

For UN/NA number recognition, a sequence classification network iwith a fine-tuned ResNet-18 backbone is utilized. The network takes as input the UN/NA number image cropped from the previous stage, extracts the image features using the backbone, and produces a sequence output with the sequence classifier. The UN/NA number is then decoded via a variable-length sequence decoder driven by connectionist temporal classification (CTC). The proposed architecture is based on the segmentation-free license

Figure 6.2: UN/NA number recognition

plate recognition model LPRNet [124], which uses a SqueezeNet [77] and Inception [125] with an input size of 96×48×3; the proposed architecture uses a different backbone and input size. The third-stage architecture block diagram is shown in Figure 6.2.

Table 6.2: Modified ResNet-18 Backbone for UN/NA Number Recognition

Operator	Repetition	Output Channels	Kernel	Stride
Conv2	1	64	7×7	2
Conv2	2	64	3×3	1
Conv2	2	128	3×3	1
Max pool	1	128	3×3	2
Conv2	2	256	3×3	1
Conv2	2	512	3×3	1

The input of the proposed network is an RGB image of size 96×48×3 (width, height, channel). Because the input's spatial size is small, the ResNet-18 backbone was modified to have only one max pooling layer instead of 5; otherwise, the features would be too small to be extracted. The modified ResNet-18 architecture is presented in Table 6.2.

CTC is "alignment-free"; it uses probabilities so the input image and output characters need not be perfectly aligned. The CTC decoder generates the UN/NA number sequence based on a greedy decoding method [126], a straightforward approach where the digit selected is that which has the highest probability.

6.3 Dataset Description

6.3.1 Dataset

The dataset consists of 2093 1440×1080 square px images captured at a commercial vehicle check stop in the United States and represent a statistically random sample of commercial vehicle HAZMAT placards at the check stop over the capture period. The remaining two images were photographed elsewhere, in daytime settings with relatively ideal conditions, to ensure each class had enough members to be distributed across train, test, and validate sets. To the best of the author's knowledge, the proposed HAZMAT

dataset is the first of its kind. It is comprised primarily of images taken at vehicle check stops, is the first whose images are taken outside against complex backgrounds, and contains a sufficient supply of night time and low-light images—about 19% of the dataset was captured at night.

Thus, available and/or public datasets could not suffice to achieve the objective. Example dataset images can be found in Figure 6.3.

Each image contains a HAZMAT placard located on a vehicle. Taken in a real-world setting, images and thus placards are subject to varying illumination and other environmental effects, as well as having a reasonably complex background. Using labelImg [92], 2229 HAZMAT placards were labelled over 15 classes; a placards-per-image histogram is found in Table 6.3. On average a placard was (169±28)×(165±27) square px and comprised only 1.84±0.60% of the entire image.

Table 6.3: Histogram of Number of Placards Per Image

1	2	3	4	5+	Total
1611	177	49	19	8	1864

The images were divided into a 60%-20%-20% train-validate-test subset split based on the HAZMAT class; where the number of class samples was <5 (rendering a 60%-20%-20% split ineffectual), an equal split was used. To ensure members of each class were sufficiently represented within each subset, images containing HAZMAT classes with the lowest numbers of samples were sorted first; images were then removed from those awaiting subset demarcation, to prevent duplicating the same image across multiple sets where >1 HAZMAT classes were present. Subset demarcation was otherwise random. Nevertheless, for these reasons, classes may not conform exactly to the 60%-20%-20% ratio ideal. The final training, validate, and test sets contained 1117, 373, and 374 images, respectively. Table 6.4 contains final subset counts per class. Images were resized to 500×500 square px for training.

(a)

(b)

(c)

(d)

(e)

(f)

Figure 6.3: Example images from the dataset. (a) A prime specimen. (b) A specimen captured at night; note the absence of red color in the placard. (c) A specimen with the potential for many false positive detections. (d) A specimen with unideal lighting, and some placard rotations, skews, and occlusion. (e) A specimen whose bounding box annotations would overlap; note the identical bottom number ("2") in each placard despite different hazard class natures. (f) A prime specimen located on the front (rather than side) of a commercial vehicle; the license plate has been obfuscated in this figure for privacy concerns.

Table 6.4: Dataset Class-Subset Breakdown

	Number of Samples in Subset		
Hazard Class	**Train**	**Validate**	**Test**
Corrosive	375	127	129
Dangerous	26	9	8
Environmental	11	4	4
Explosives	7	2	2
Flammable	411	142	141
Flammable Solid	1	1	1
Hot	127	37	47
Inhalation	40	14	15
Miscellaneous	23	8	10
Non-Flammable	199	68	69
Organic	1	1	1
Oxidizer	51	14	18
Poison	7	2	3
Radioactive	4	1	2
Toxic	40	12	14
Total	**1323**	**442**	**464**

6.3.2 Dataset Augmentations

The models were trained with the following per-epoch dataset augmentations:

1) Horizontal flips (with 0.5 probability).
2) Scaling (between 0.8 and 1.25×).
3) Conversion to greyscale (with 0.5 probability), due to the presence of greyscale (night-time) images in the dataset.
4) Contrast adjustment (gain between 0.8 and 1.25×).
5) Brightness adjustment (bias up to ±20% of bit representation maximum).
6) Bounding box jitter (up to ±5% of box area).

This was found to contribute to better results than not using augmentations, likely because this expanded the diversity of images seen by the proposed model and thus the model's representational power. Once the mAP@0.5 over the validation set saturated, training continued without per-epoch augmentations 1, 4, and 5; this method can be seen as a narrowing ("fine-tuning") of the learned features to actual instances, and contributed to a higher final validation mAP@0.5 for some models [95].

Preliminary tests misclassified or failed to localize relatively small and large placards. To mitigate this, the training set was explicitly inflated by scaling down each image (0.5× and 0.75×), and scaling up (1.5×, 2×, 2.5× and 3×) on each placard from the placard's geometric center. The number of additional training images generated via this method for each image is $4n + 2$, n being the number of placards in the image. Though each placard was focused on when performing >1× zooming, bounding boxes of other placards if present were included in the annotation of the final scaled image if the box overlapped the resultant image by ≥45%. A downscaled image kept the original size by replicating the right and bottom border pixels.

6.4 Training Methodology

6.4.1 Model and Training Environment.

To train the proposed Stage 1 models, Tensorflow (TF) 1.15.0 with GPU support in Python 3.7.7 was used within Anaconda 4.8.4. The host computer, running Windows 10, had an AMD Ryzen Threadripper 3.8 GHz 8-core processor with 32 GB of DDR4 RAM, and an Nvidia GTX 1080 ti GPU with 11 GB of DDR5 memory. A model's batch size was the largest multiple of 8 that could be achieved using the GPU. SSD- (including SSDlite-) based models were trained using TF's Object Detection API (ODAPI) [127]. YOLOv3 and its Tiny variant were trained using Keras [128] 2.3.0 running over the TF GPU environment.

With this configuration, each training epoch took approximately 30 seconds (though this was ultimately subject to the nature and size of the model being trained).

6.4.2 Models Training

First, current "off-the-shelf" models were assessed in terms of their accuracy and processing times. Because a fast and accurate model for Coral edge prediction was desired, the selected architecture had to have quantize-capable, and could not contain Coral-incompatible, layers. Though this excludes FPNs and YOLOv3, they were trained as a benchmark.

SSD models were trained with in-place batch normalization, 0.9 momentum, and a momentum optimizer that included cosine decay LR scheduling [129], as seen in (2): LR_{init} is the initial LR; x is the (integer) current step; and x_{max} is the number of steps until the LR is 0. Each optimizer had a warmup period [130], described in (3), in which the LR increased linearly from a lower LR (LR_{warm}) to the LR_{init} specified in the cosine decay scheduler; in (3), x_{max} is the number of steps until the LR is the LR_{init} of the cosine decay scheduler. For brevity later in this chapter, (2) and (3) are shown with x omitted, and LR_{init} is not shown in (3) because it is the same as that in (2).

$$CosDecay(LR_{\text{init}}, x, x_{\text{max}}) = LR_{\text{init}} \times \frac{1+\cos(\frac{x\pi}{x_{\text{max}}})}{2} \tag{6.2}$$

$$WarmUp(LR_{\text{warm}}, x, x_{\text{max}}, LR_{\text{init}}) = LR_{\text{warm}} + x\,\frac{LR_{\text{init}}-LR_{\text{warm}}}{x_{\text{max}}} \tag{6.3}$$

A cosine decay [129] for SSD was used because it was found to outperform other LR schedulers, especially exponential decay.

When training SSDlite + MobileDet EdgeTPU from scratch, x_{max} was chosen to be between 2-2.5E5 as a rough middle between the respective short- and long-schedule of 5E4 and 4E5 steps described in [39]; other models trained from scratch followed this approach. The SSD-based models transfer learned using preinitialized weights were

empirically determined to converge around 4E4 steps, and hence the decision to train to a maximum of 5E4 steps. Despite differing hyperparameters, model training was fair because, generally, a model's default LR_{init} was used LR_{init} x_{max}; the only exception was SSDlite + MobileNet EdgeTPU LR_{init} led to unstable training, so it was reduced to 0.5.

YOLOv3 and its tiny variant were trained for a maximum of 100 epochs with 3 warmup epochs (with an LR_{warm} of 0 and LR_{init} of 1E-4). A Keras ReduceLROnPlateau callback was used, monitoring training loss, with a reduction factor of 0.1, patience of 2 epochs, and no minimum LR. Additionally, a Keras EarlyStopping callback was used, monitoring training loss, with a patience of 5 epochs and minimum improvement of 0.01. YOLO anchors were generated specifically for the training set (9 for full and 6 for tiny). This configuration was the default for the model. Moreover, it is noteworthy that better results were obtained monitoring training than validation loss.

A homogenous 8-bit quantization-aware training for quantization-capable models was employed. Models were trained with full precision for 10,000 steps, and quantized training thereafter.

6.5 Real-time Prediction Methodology

As detailed in Section 6.2, each stage's architectural backbone was customized according to the task complexity and the number of classes; for example, it is much easier to detect one of 15 classes than one of 80, as are present in the COCO dataset [131]. The proposed streamlined backbones reduce unnecessary complexity in the internal and output layers. For greater efficiency gains, and to achieve real-time (> 30 FPS) prediction on (typically) computationally-constrained edge hardware, the models were implemented using the GStreamer framework [132].

GStreamer uses plugins, data flow, and media type handling/negotiation. Plugins are shared libraries that are dynamically loaded at runtime and can be independently extended and upgraded. When arranged and linked together, plugins form the processing pipeline that defines the data flow for a streaming media application. GStreamer eliminates

performance bottlenecks by more efficiently utilizing the limited hardware resources of an edge system. For example, GStreamer can encode/decode streaming input video in hardware, which is considerably faster than using software.

GStreamer can be used on the Raspberry Pi or on the Jetsons. In the latter case, GStreamer has plugins that leverage Nvidia GPU capabilities allowing the entire pipeline (including DL prediction) to be processed on the GPU, with zero memory copy between the CPU and GPU; this makes the entire pipeline faster and more efficient. Nvidia refers to this GStreamer integration plugin as “DeepStream”.

Figure 6.4 shows how modular plugins were connected to form the processing pipeline of the proposed HAZMAT recognition system when deployed on the Jetsons. Each plugin represents a functional block. Hardware-accelerated plugins interact with underlying hardware (where applicable) to deliver maximum performance.

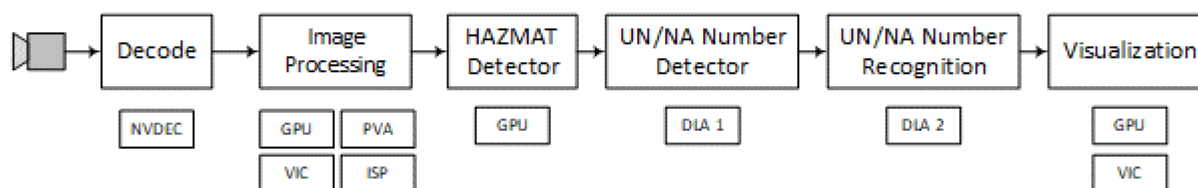


Figure 6.4: Processing pipeline as deployed on the Jetsons, possible using Nvidia libraries.

The Nvidia Jetsons utilize accelerators designed to augment the functionality of the GPU and CPU, thereby providing greater flexibility and a more efficient implementation of common algorithms according to the hardware characteristics.

NVDEC is graphics card feature that performs video decoding, a compute-intensive task traditionally done by the CPU. It is part of Nvidia’s Video Codec SDK [133].

The Programmable Vision Accelerator (PVA) [134] is capable of real-time decoding/encoding streams from multiple cameras (side, front, inside) in a high dynamic range (up to 1.8 GPIX/s).

The Nvidia Video Image Compositor (VIC) [135] implements 2D image and video operations, e.g. scaling, blending, rotation, video post-processing, and advanced capture-time denoising.

The Nvidia Image Signal Processor (ISP) [136] processes data from the Video Input Subsystem, or raw data directly, to remove artifacts from sensors, the camera lens, and color-space conversion.

The open source Nvidia Deep Learning Accelerator (NVDLA) [137] allows for the design of deep learning inference accelerators. It has a modular, scalable, and configurable architecture for better integration and portability, with many low-power and IoT devices supported.

For Jetson deployment, neural network models are converted to UFF (Universal Framework Format) and used to generate a TensorRT execution engine according to the system's available resources.

Also, the system on an NCS2 and a Google Coral accelerator was deployed. For the NCS2 and the Google Coral accelerators, the USB-interface dongles were used; because a host computer is required in these cases, a low-cost Raspberry Pi 4 (RPi4) was used.

The RPi4 is single-board edge computer with a Broadcom quad-core 1.5 GHz processor and 4 GB of LPDDR4 RAM. The (open source) 64-bit Raspberry Pi operating system based on Debian was used. OpenCV 4.4 was built from source with GStreamer compatibility. Also, the TFLite's C++ API libraries were built. For testing the performance, the frozen models (.pb) with FP32 were run, and the converted TFLite models (.tflite) with INT8.

To run the model on the NCS2, the model optimizer migrates the neural network model into a half-precision (16-bit floating point, FP16) intermediate representation (IR) (BIN/XML files). This IR is an abstraction of the model that can subsequently be deployed on OpenVINO-capable CPUs, GPUs, FPGAs, or (as in the proposed case) NCSs using the library's Inference Engine.

To run a quantized model on the Google Coral, it must be converted from TF frozen format to TFLite. Next, Coral's Edge TPU compiler rearranges layer weights from the TFLite file into a new Coral-compatible format. The compiler shows which layers can be ran on the Coral, and which default to the CPU.

6.6 Results

6.6.1 Testing Environments

In Table 6.5, processing times for non-quantized models were measured on the training setup using Python's time.clock(). They are shown as GPU / CPU pairs. "Gross" refers to the network's prediction time; net refers to the time with expenses, i.e. import, resizing, and marking.

In Table 6.5, processing times for quantized models are the theoretical Coral prediction time and measured using OpenCV-Python's [138] getTickCount().

Processing times for specific edge hardware deployments can be found in Section 6.6.8. On each platform, the time was measured using Python's time library and the highest resolution clock available. Further information can be found in that section.

Models were tested using no dataset augmentations and with a batch size of 1.

Table 6.5: Summary of Model Training and Results for Stage 1

Model	Input Image (W×H×3)	Params × 10^6	Batch Size	Train Steps × 10^3	mAP@ 0.5 (Test)	Per-Image Proc. Time (ms)	
						Gross	Net
SSD + MobileNetV1 FPN	640×640	11.0	8	38.1	**0.9964**	49 / 479	70 / 500
SSD + MobileNetV2 MNAS FPN	320×320	**2.36**	24	34.6†	0.8140	**30** / **66**	**51** / **87**
YOLOv3	416×416	61.7	8	173	0.7980	87 / 1175	119 / 1242
SSDlite + Custom	320×320	3.48	24	231†	**0.8930**	**17** / 57	**36** / 76
SSDlite + MobileDet EdgeTPU	320×320	3.38	24	210†	0.8861	19 / 58	39 / 79
SSDlite + MobileDet EdgeTPU	320×320	3.38	24	170	0.8704	19 / 58	39 / 79
SSD + MobileNetV1	300×300	5.70	24	34.8	0.8184	18 / 46	39 / 67
SSDlite + MobileNet V2	300×300	3.22	24	38.1	0.8375	19 / 48	40 / 69
SSDlite + MobileNet V3 Small	320×320	**1.05**	24	196	0.8513	20 / **26**	41 / **47**
YOLOv3 Tiny	416×416	8.71	24	180	0.5658	44 / 202	73 / 244
SSDlite + Custom*	320×320	3.48	16	231†	**0.8404**	6.6	—
SSDlite + MobileDet EdgeTPU*	320×320	3.38	16	214†	0.8285	7.1	—
SSDlite + MobileDet EdgeTPU*	320×320	3.38	16	160	0.7728	7.1	—
SSD + MobileNetV1*	300×300	5.70	16	31.7†	0.7801	**6.2**	—
SSDlite+MobileNetV2*	300×300	4.80	16	39.1†	0.8357	**6.2**	—
SSDlite + MobileNet EdgeTPU*	320×320	**3.12**	16	196	0.7814	7.3	—

* denotes the model is quantized, and its Proc. Time refers to processing time on the Coral rather than a GPU / CPU
† denotes the weight model at that step was achieved by discontinuing dataset augmentations as described in Section 6.3.2.

6.6.2 Performance Metric

To assess the trained models, the mAP@0.5—popular for object detection and classification was used. It can be defined as the class-wise mean of the average precision, which itself is the area under an interpolated precision-recall curve, at an intersection over union of 50%. mAP@0.5 was calculated using the TF ODAPI [127]. Although COCO mAP was used, the mAP@0.5 challenge metric for COCO is the same as for PASCALVOC [131]; thus, for more information, the reader was referred to the PASCALVOC challenge paper [118].

The F1-scores for some classes are listed, e.g. in a comparison in Table 6.9. F1-score can be defined in terms of precision and recall, which themselves can be defined in terms of true positive (TP), false positive (FP), and false negative (FN).

6.6.3 Stage 1 Training Results Summary

Stage 1 training results are found in Table 6.5, divided into 3 categories by dashed lines. The first category (from the top of Table 6.5) contains “benchmark” models trained for results purposes but impractical for edge usage due to large processing times or INT8-quantize-incapable layers. The second category has models whose quantized versions could theoretically be run on edge hardware, trained to gauge speed-accuracy trade-offs. The third category contains INT8-quantized models. The final selected weights for SSD-based models were those that had the highest COCO mAP@0.5 over the validation set, which often occurred just before or slightly after the mAP generally plateaued. The selected weights for the YOLO models were those achieved after the EarlyStopping callback was triggered.

6.6.4 Stage 1 General Model Comparison

Categories are described in the previous subsection and are in reference to Table 6.5.

From Category 1—and overall—SSD + MobileNetV1 FPN attained the highest test set mAP@0.5 of 0.9964. However, the (net) processing time is comparatively slow at 68 ms on a GPU or an unusably slow 505 ms on a CPU. The model illustrates a speed-accuracy trade-off where the processing time required is inhibitive to any real-time system. Although prediction at different scales is advantageous, the comparatively more modest test set mAP@0.5 of 0.8140 for SSD + MobileNetV2 MNAS FPN hints that its accuracy may be from having many more weights and a larger image input size (640 vs. 320 px)—specifically, higher resolution images on which the initial weight model was trained. Nevertheless, because FPNs contain INT8-quantize-incapable layers, further discussion for the purposes is moot. YOLOv3, with almost 62M parameters and high GPU

and CPU processing times—coupled with its test set mAP@0.5 of 0.7980—make it ill-suited to the objective.

From Category 2, the proposed model beat SSD-based MobileNet implementations by between 0.042 and 0.075. The model had the best GPU speed, surpassing other SSD-based object detectors by 1-3 ms and YOLOv3 Tiny by 27 ms. Though MobileNet-based models had lower CPU processing time, the proposed model has the best speed-accuracy trade-off for high-compute hardware (e.g. GPUs and presumably TPUs). Given YOLOv3 Tiny was slower than the SSD-based models and had relatively poor mAP@0.5, it was deemed unsuitable for the task and was not tested further.

Category 3 contains information on edge models and is thus most pertinent to the objective. Validation set performance for quantized SSDlite + MobileDet EdgeTPU lacked compared to its non-quantized version. The validation mAP@0.5 was less than that of some other models; this motivated to adapt the architecture to find a higher-performing model with a similar processing time. The custom quantized network achieved a test set mAP@0.5 of 0.8404. The model beat the test set mAP@0.5 for quantized SSD-based MobileNet models by between 0.005 and 0.060. Models in Category 3 had near equivalent Coral gross processing times— an impressive 6-7 ms. Though the custom model could achieve a theoretical 151 FPS on the Coral with a powerful computer, there are hardware limitations using the RPi4 and an IP camera; see Section 6.6.7 for a discussion in a real-time context.

YOLOv3, and especially its tiny variant, were marred by false positives, and hence their comparatively lower mAP@0.5. Examples of potential false positives can be found in Figure 6.5 (c).

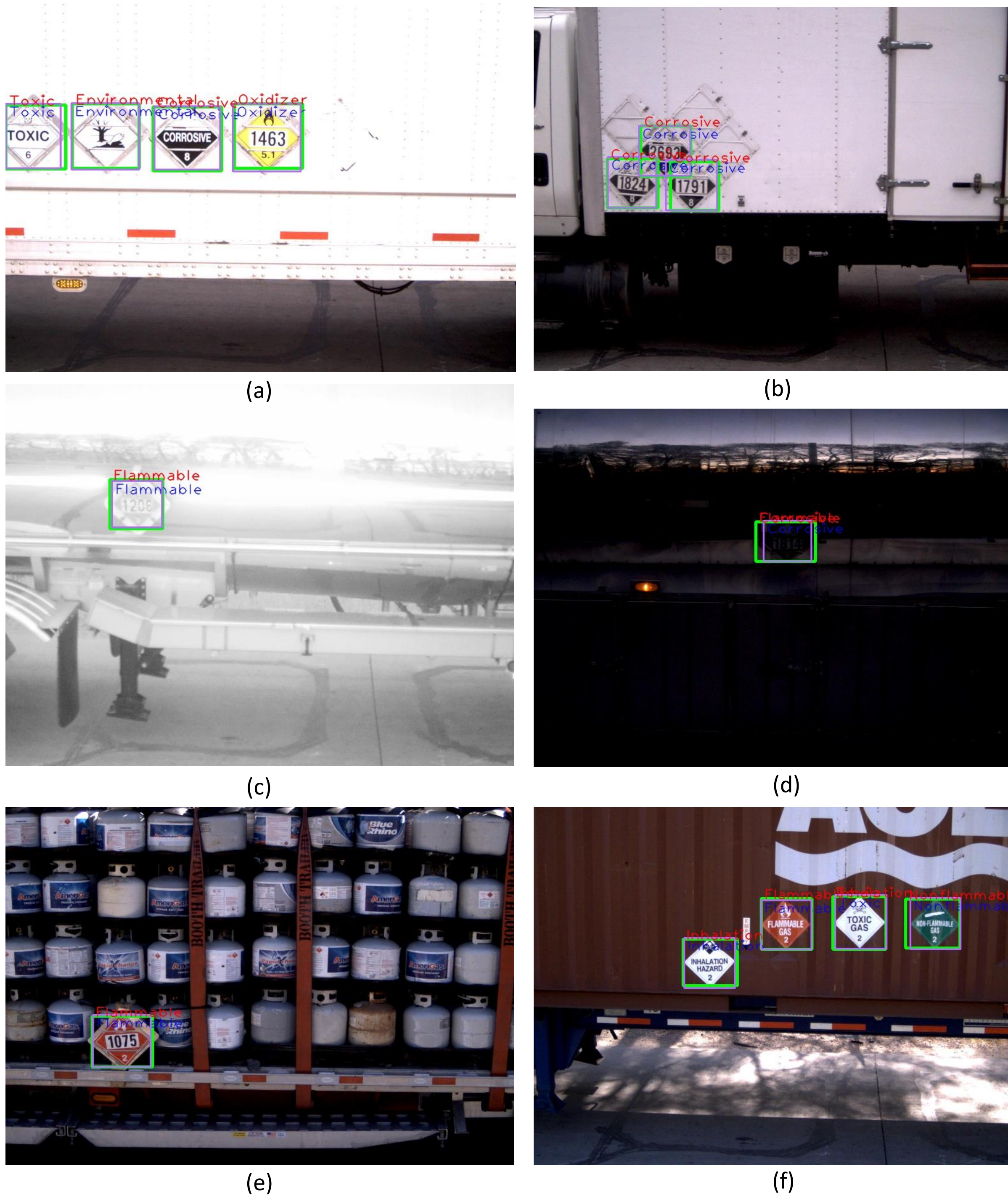


Figure 6.5: Example test set inferences of the quantized SSDlite + Custom model with detection boxes in green, ground truth boxes in mauve, predicted classes in red, and ground truth classes in blue. (a) Correctly identified prime specimens. (b) Correctly identified specimens whose bounding boxes would overlap, beside correctly unidentified potential false positives. (c) A correctly identified specimen within a highly irregular background. (d) A correctly identified overexposed night capture specimen. (e) An underexposed late day captured specimen; though correctly identified as Corrosive, there is a duplicate detection mistaking it for Flammable. (f) Specimens that are correctly recognized, but there is a duplicate detection on the Toxic placard mistaking it for Inhalation.

Output test set results images from the quantized SSDlite + Custom model can be found in Figure 6.5.

6.6.5 Stage 2 and 3 Model Results

Table 6.6 shows a summary of the model results for Stage 2, wherein the UN/NA number and/or the bottom class digit are localized. The proposed Stage 2 model achieved 0.982 mAP@0.5 prediction accuracy—the same as SSDlite+ MobileDet Edge TPU, but the proposed model uses 36.7% less parameters. Moreover, the proposed model has 0.007 greater mAP@0.5 than SSDlite+ MobileNetV2. The Stage 2 model file size is 3.5× smaller than that of SSDlite+MobileDet EdgeTPU. The results show that customizing backbones for a specific task creates accurate models with smaller sizes, meaning faster loading and processing times; this makes such models better suited to edge devices with small memory capacities.

Table 6.6: Summary of Model Results for Stage 2

Model	Input Image (W×H×3)	Params × 10^6	File Size (MB)	mAP@0.5
SSDlite+MobileDet EdgeTPU	320×320	3.38	5.04	0.982
SSD+MobileNetV1	300×300	5.70	5.52	0.971
SSDlite+MobileNetV2	300×300	3.22	4.71	0.975
Proposed Model	320×320	2.14	1.44	0.982

Table 6.7 shows a summary of the model results for Stage 3, where the UN/NA number is extracted as a character string. The proposed model is compared with LPRNet [124], originally designed to recognize license plate numbers. Because license plates and cropped UN/NA numbers are visually similar, LPRNet achieved a high mAP@0.5 of 0.950. However, the proposed changes to the model's backbone and input size improve

Table 6.7: Summary of Model Results for Stage 3

Model	Input Image (W×H×3)	Backbone	mAP@0.5
LPRNet	94×24	SqueezeNet	0.950
Proposed Model	96×48	Customized ResNet-18	0.991

the mAP@0.5 to 0.991. This highlights the versatility of segmentation-free models with CTC decoding algorithms for different applications.

6.6.6 Edge Hardware Deployment and Real-Time Results

To assess the real-time suitability of the proposed system, the following implementations were tested of each custom model:

- On the RPi4 alone, using TFLite.
- On the RPi4 + NCS2 using OpenVINO's IR files.
- On the RPi4 + Coral USB Accelerator using TFLite.
- On the Jetsons (Nano and Xavier) using the converted TensorRT files.

As explained in Section 6.5, all implementations used GStreamer where appropriate. On the RPi4, the TFLite runtime library was installed which provides the minimum code required for prediction with Python; this saves disk space, important for an edge device. Additionally, the USB accelerators were connected to the RPi4's USB 3.0 ports for maximum data transfer speed.

Deployment processing time results for all stages are shown in Table 6.8. However, Stage 1 (initial placard localization) is focused upon because it is the most compute-intensive task for a video stream, requiring continuous processing of each frame; conversely, Stage 2 need only run if a placard is identified in Stage 1, and Stage 3 would only run if a UN/NA number is identified in Stage 2. Moreover, for cases where the pipeline can be deployed on different hardware components—for example, deploying the Stage 1 model on the Xavier's GPU and stages 2 and 3 on its DLA—Stage 1, being the largest and thus slowest model, will be the system's bottleneck.

For testing, an IP camera was connected to the edge device via the local network. The test camera offered a stream video with resolution 640 × 360 px at 60 FPS with a bitrate

of 385 kbps using h.265 compression. However, for the FP16 and INT8 models, the high-performance Xavier could process the video so fast that the camera FPS was the bottleneck; thus, to test these models on the Xavier, a video was streamed with resolution 1920 × 1080 px at 120 FPS with a bit rate of 9845 kbps using h.264 compression. The model's prediction execution time was measured from input to output. Processes that could be easily parallelized, e.g. image capturing and scaling, were not taken into account. Overclocking was not used.

When running on the RPi4 alone, The quantize-aware-trained INT8 custom model achieved an average of 17.20 FPS—a substantial increase compared to the conventional full precision model with an average of just 2.07 FPS.

As expected, using the RPi4 + the USB accelerators further increase the achievable FPS of the custom model. The NCS2 (using FP16) and the Coral USB accelerator (using INT8) achieved, respectively, 2.2× and 2.7× faster prediction than the INT8-quantized model running on the RPi4 alone. For the NCS2, a FP16 model was generated from the full precision model using OpenVINO's post-training FP16 quantization/optimization.

Via TensorRT, three models were deployed on the Jetsons: the trained FP32 custom model; a post-training FP16 quantization of the FP32 custom model, generated using TensorRT; and the quantize-aware-trained INT8 model.

Table 6.8: Results: Real-time Prediction Speed (FPS)

Platform	Platform Cost (USD)	Stage 1			Stage 2			Stage 3		
		FP32	FP16	INT8	FP32	FP16	INT8	FP32	FP16	INT8
Raspberry Pi 4 (RPi4)	$55	2.07	N/A	17.20	2.53	N/A	21.02	3.26	N/A	27.10
RPi 4 + NCS2	$124	N/A	37.12	N/A	N/A	39.85	N/A	N/A	43.13	N/A
RPi 4 + Coral USB	$115	N/A	N/A	46.42	N/A	N/A	59.28	N/A	N/A	76.42
Jetson Nano	$99	25.12	28.89	N/A	29.12	33.49	N/A	32.12	37.04	N/A
Jetson Xavier (DLA)	~$700	N/A	46.74	N/A	N/A	77.74	N/A	N/A	94.14	N/A
Jetson Xavier (GPU)	~$700	58.13	89.75	104.49	71.14	121.20	172.51	95.80	152.11	250.49

On the Jetson Xavier, the FP32 model achieves a prediction speed of 58.13 FPS—2.3× faster than the model running on the Nano, and an impressive 28.1× faster than the full precision model running on the RPi4 alone. When reduced to FP16 or INT8, the system's bottleneck becomes the input stream (60 FPS). The theoretical processing speed (from processing a video, as aforementioned) of the FP16 model is 89.75 FPS, 2.4× faster than the RPi4 + NCS2 and 3.1× faster than the Jetson Nano. The Xavier supports INT8 prediction using a Volta GPU; the INT8 model achieves a superior 104.49 FPS, 2.3× faster than the RPi4 + Coral USB accelerator and 6.1× faster than RPi4 alone.

A further deployment discussion, especially in terms of unit cost, can be found in Section 6.7.3.

6.6.7 Comparison to Others' Works

Reference [139] uses a system and dataset designed to integrate depth information into the final placard detection and recognition. Thus, the intricacies of their system are out of scope of this work. However, they used a YOLOv3 Tiny object detector as their object detection model. As seen in Table 6.5, the mAP@0.5 of the YOLOv3 Tiny model trained on the HAZMAT dataset proposed in this thesis is 0.5658, 0.2746 less than the quantized SSDlite + Custom model. The proposed model also has 60% less parameters.

In [140] an attention + SIFT method was proposed with a dataset of 600 images: 1 placard per image × 25 images per class and background type × 8 classes × 3 types of backgrounds—OSB, woodchip, and brick, with brick the most difficult for [140] due to shadows and illumination effects. Each set of 25 images per-class per-background contains 5 images captured at each of the azimuths -45°, -30°, 0°, 30°, and 45°. The training set consisted of the 400 images in their dataset with OSB and woodchip backgrounds; the 80 negative azimuth brick images were used as the validation set; and the remaining 120 brick images comprised the test set. Training weights were initialized with the weight model that achieved the highest test set mAP@0.5 over the dataset. Though every test set placard was recognized (i.e. no FNs), there were 1 misclassified

duplicate and 1 FP (class Explosive). In Table 6.9 shows per-class F1-scores vs. the accuracies of [140]. Like [140], CPU processing time is listed. The proposed model is 21-91% faster and generally more accurate than [140]. The proposed model has also been shown to handle more complex backgrounds and can detect ≥1 placards in an image. Overall, it shows the model's superiority to keypoint detection.

The performance of the model transfer learned using the dataset in [140] indicates potential improvements with more per-class data, e.g. Organic, and an overall better balanced dataset. This would seem to be a better avenue to higher accuracy than, for example, resampling the dataset. The results on [140] also show the proposed model's capabilities to correctly predict non-0° signs.

The system cannot be compared against other related works because, at this time, they neither supply public dataset results nor do they make the intricacies of their algorithms known.

Table 6.9: Per-Class comparisons against [140] over dataset in [140].

Class	**Proposed system**		[140]	
	F1-Score (%)	**Proc. Time (ms)**	**Accuracy (%)**	**Proc. Time (ms)**
Combustible	100	79	~85	~650
Dangerous When Wet	96.7	79	~90	~250
Explosive	96.7	79	100	~100
Flammable	100	79	~75	~450
Non-Flammable	96.7	79	~65	~900
Organic	100	79	~90	~550
Oxidizer	100	79	~75	~700
Radioactive	100	79	~80	~600

6.7 Discussion

6.7.1 General

In this chapter the first automatic HAZMAT detection system for commercial vehicles at check stops was designed, with a modest per-unit cost. The research may streamline and revolutionize the ground shipping of hazardous materials, which has implications for governments, institutions, and corporations across the fields of supply chain management, emergency management, intelligent transportation systems, shipping, smart cities, and Industry 4.0.

6.7.2 Placard Localization (Stage 1) Deployment Results

Deployment on the Jetson Xavier had the fastest Stage 1 processing speed of 58.13 FPS for the full precision model, and 104.49 FPS for the INT8 quantize-aware-trained model. However, this performance comes at a steep cost, with approximately only 0.08 FPS per $ (USD) for FP32 and 0.149 FPS per $ (USD) for INT8. Nevertheless, the Jetson Xavier is the platform of choice for extremely critical high-accuracy and time-sensitive applications.

The Jetson Nano achieves a relatively good Stage 1 full precision performance at 25.12 FPS, or 0.25 FPS per $ (USD). This would be the platform of choice for low-cost consumer-grade applications that require higher accuracy than that afforded by quantization, but are not highly time-sensitive and not necessarily real-time.

The RPi4 + Google Coral system achieves good INT8-quantized performance at 46.42 FPS, or 0.40 FPS per $ (USD). However, the INT8 model achieved 0.0526 less mAP@0.5, making this system the platform of choice for low-cost consumer-grade applications that require higher time-sensitivity but can tolerate a slight drop in accuracy.

Even the INT8-quantized performance of the RPi4 alone at 17.20 FPS, or 0.31 FPS per $ (USD), is reasonable. It could still be competitive in ultra-low-cost consumer-grade applications that require neither high time sensitivity nor the highest achievable accuracy.

6.8 Summary

A fast and accurate 3-stage HAZMAT recognition edge system was developed by creating a DL-based solution and demonstrating its real-time implementations on edge inference accelerator hardware. The proposed emerging intelligent system is envisaged to be deployed at vehicle check stops or integrated into a larger edge system capable of recognizing many commercial vehicle features.

In Stage 1, a placard is localized from an input image or video stream. SSDlite + MobileDet EdgeTPU presents the best speed-accuracy trade-off. The model's test was set mAP@0.5 to 0.8861 using custom parameters. From SSDlite + MobileDet EdgeTPU, a custom model was developed. The custom model achieved a test set mAP@0.5 of 0.8930. The custom model was retrained in an 8-bit integer quantize-aware fashion. This model achieved a test set mAP@0.5 of 0.8404, 0.0676 higher than vanilla quantized SSDlite + MobileDet EdgeTPU. The proposed model can detect ≥1 placards per image with the same processing time, and is robust to placard irregularities, illumination, and complex backgrounds.

From detected HAZMAT placards in Stage 1, Stage 2 recognizes the bottom HAZMAT class digit and determines the presence of a UN/NA number for more specific substance information. The model—comprised of a highly simplified custom SSDlite + MobileDet EdgeTPU architecture trained achieved a mAP@0.5 of 0.982. Although the custom model achieved the same accuracy as the architecture from which it was derived, it does so with 36.7% less parameters and a 3.5× smaller file size.

If a UN/NA number is found in Stage 2, it is inputted into the Stage 3 model. Stage 3 uses a ResNet-18 backbone, Sequence Classifier, and CTC decoder to recognize text

strings in a way that does not need aligned inputs and outputs during training. It achieves a digit recognition mAP@0.5 of 0.991, 0.041 more than LPRNet.

The Stage 1 FP32/INT8 models can achieve 58.13/104.49 FPS on a Jetson Xavier. The proposed INT8-quantized model achieves 46.42 FPS on the Coral USB accelerator paired with a RPi4 and a 60 FPS IP camera. Meanwhile, the FP32/INT8 Stage 2 model can achieve 71.14/172.51 FPS on the Xavier, and the INT8 model achieves 59.28 FPS on the Coral setup. Finally, the proposed Stage 3 model can achieve 95.80/250.49 FPS on the Xavier, and the INT8 model achieves 76.42 FPS on the Coral setup. Though performance is ultimately limited by Stage 1, the results nonetheless demonstrate the power of developing and pairing custom and/or quantized networks with edge accelerators for accurate and real-time deep learning inference.

Part IV

Conclusion

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In recent years, Intelligent Transportation Systems (ITS) have led to substantial advancements in road safety and traffic efficiency. Paired with Artificial Intelligence (AI)—and specifically its state-of-the-art subfield Deep Learning (DL)—ITS applications can model complex systems with high accuracy; however, such applications still require significant computational power. Therefore, much processing of data takes place in on-premises data centers or cloud-based infrastructure. However, with the arrival of powerful, low-energy consumption DL accelerators, computations can now be executed on devices at the edge. These devices can facilitate the implementation of ITS applications on highways or remote areas.

The main challenge associated with implementing and running deep learning algorithms on edge devices is its limited memory and computational resources. This forces the systems designer to consider trade-offs between speed and accuracy; it is difficult to achieve both simultaneously using a low-complexity device, such as most inexpensive and commercially available edge devices. This thesis addresses the challenge by providing new algorithms to accelerate convolution stride two, especially important for object detection applications. Additionally, three accurate ITS applications are designed and deployed on edge devices and are shown to achieve real-time prediction.

Chapter I introduces ITS in greater depth, and provides the objectives and motivations behind the proposed thesis and a summary of contributions. Chapter II describes machine learning (ML) and DL, Convolutional Neural Networks (CNNs), optimization techniques for CNNs, and object detection fundamentals. Each proposed work has been divided into one of two categories: (1) hardware optimization for convolutional neural networks; and (2) ITS applications.

Most modern DL-based object detection networks utilize CNN architectures. Hardware optimization of DL architectures—detailed in Part II Chapter III—begins with the formulation of new Winograd minimal filtering algorithms for accelerating CNN architectures containing layers with stride 2. The algorithms—able to decrease computational complexity and increase efficiency by trading expensive multiplications for cheap additions—optimize 1D, 2D, and 3D convolutions. All proposed Winograd stride 2 algorithms were implemented on an NVIDIA K20c GPU. Results show that the algorithms contribute to a speedup of 1.44x, 2.04x, 2.42x, and 1.73x for respective 3×3, 5×5, 7×7, and 3×3×3 kernels. Additionally, a novel Processing Element (PE) for FPGAs is designed that can process stride one and stride one convolutions. The novel PE uses the same number of DSPs (32) as a PE able to perform two Winograd stride one calculations, and 25 less DSPs than what would be required by having independent PEs that perform two Winograd stride one and one Winograd stride two operations. Using a systolic array, a larger number of PEs can be used, improving the system's efficiency via greater parallelism. Finally, an implementation of the proposed PE was tested via integration into a modified VGG-16 architecture where one stride two convolutional layer was used instead of one stride one convolutional layers followed by one max-pooling layer, as in the original model architecture. The novel implementation achieves DSP efficiencies of 1.22 GOPS/DSPs and 1.33 GOPS/DSPs for the original and modified VGG-16 architectures, respectively.

Three ITS applications are proposed in Part III of this thesis. Chapter IV presents a license plate localization (LPL) system, Chapter V a real-time Commercial Vehicle Safety Alliance (CVSA) decal recognition edge computing system, and Chapter VI a real-time edge system for recognizing cropped hazardous material (HAZMAT) placards.

The LPL system was designed using inverted residual blocks, linear bottlenecks, and depthwise separable convolutions. The system was trained using three popular publicly available datasets: Caltech Cars 1999 (Rear) 2; the University of Zagreb License Plate Detection, Recognition, and Automated Storage library; and the NTUA Medialab LPR Database. Tested on those datasets, the proposed system achieved 98.4%, 97.83%, and

99.8% accuracy, respectively. Using an NVIDIA K20c GPU, the processing time was only 20 ms per image, regardless of the number of license plates in the image. Additionally, an algorithm for multi-threaded video capture with motion detection was proposed; this allowed the DL prediction stage—the most computationally-complex part of the system—to only be running when a vehicle is detected, thus contributing to greater efficiency and real-time suitability for computationally non-complex devices such as smartphones. This has the potential to make ALPR less expensive from a hardware and thus consumer standpoint, facilitating and expediting the advancement and creation of ITS technology, thereby making LPL more accessible for a greater percentage of society.

The second application presented in this thesis is the first two-stage real-time edge-based DL system for the detection and recognition of decals issued by the Commercial Vehicle Safety Alliance (CVSA). The first stage was designed to locate small objects; it localizes a vehicle's windshield and the decal therein, and determines the decal's color. The second stage localizes and recognizes the decal's digit and determines its corner cut. The decal's date of issue can be determined using its digit, corner cut, and color. The proposed architectures demonstrated high average precision: 98.5% mean average precision (mAP)@0.5(IoU) for Stage One and 97.5% mAP@0.5 for Stage Two when the input is a single frame. A Stage Two model was proposed for enhancing accuracy via redundancy when using a video stream input; in this model, up to 7 images of the same decal over multiple frames are collated and simultaneously predicted, and majority voting is used to determine the final decal characteristics. This "7-spots" model achieved 98.1% mAP@0.5; the model's mAP@0.5 is 0.6% better than SSDLite+MobileNetV2 (the best off-the-shelf model), and the proposed model has an inference time of 5.9 ms, 0.3 ms faster than SSDLite+MobileNetV2. The results were evaluated using hardware accelerators from different vendors like Intel, Google, and NVIDIA. Finally, the model's stages can run in parallel (though limited by Stage 1), and the proposed system achieved an inference speed of 173.31 FPS when deployed on an NVIDIA Jetson AGX Xavier and 60.83 FPS on an inexpensive RPi4 + Google Coral system.

The third proposed application is a novel real-time deep learning-based edge System for Hazardous materials (HAZMATs) recognition. The HAZMAT standards set by the OSHA comply with the United Nations Globally Harmonized System of Classification and Labelling of Chemicals (GHS), the universal standard describing the appearance and purpose of HAZMAT symbols, and for what chemicals a certain symbol is valid. HAZMAT symbols are diamond-shape decals. The system will be able to localize, classify 15 HAZMAT symbols in real-time, and extract the text present in the placard for further elaboration of the material or danger.

A fast and accurate 3-stage HAZMAT recognition edge system was presented. Custom models were designed for each stage. Stage 1 localizes a placard from an input image or video stream. The proposed custom model for stage 1 achieved a test set mAP@0.5 of 0.8930. The model can detect ≥1 placards per image with the same processing time, and is robust to placard irregularities, illumination, and complex backgrounds. Stage 2 utilizes the cropped HAZMAT placard and recognizes the bottom HAZMAT class digit and determines the presence of a UN/NA number. The model for the second stage achieved a mAP@0.5 of 0.982. Stage 1 and stage 2 achieved the same accuracy as the MobileDet architecture, but it does so with 36.7% less parameters and a 3.5× smaller file size. The third stage recognize the UN/NA number if it is found in Stage 2. This stage uses a ResNet-18 backbone, Sequence Classifier, and CTC decoder to recognize text strings in a way that does not need aligned inputs and outputs during training. It achieves a digit recognition mAP@0.5 of 0.991, 0.041 more than LPRNet. On a Jetson AGX Xavier, the Stage 1 models can achieve 104.49 FPS, the Stage 2 model can achieve 172.51 FPS, and the Stage 3 model can achieve 250.49 FPS.

7.2 Future Work

In the future, the research works presented in this thesis may be expanded.

In this research work, a processing element (PE) capable of handling both stride one and stride two Winograd operations was implemented on an FPGA. This implementation and the proposed PE could serve as a baseline for accelerating other neural network architectures, particularly those which natively use stride two convolutional layers, e.g. the MobileNet family. In addition, the proposed Winograd algorithms could be extended to efficiently compute convolutions for greater strides, e.g. stride 3 or 4, when and if architectures containing such strides become commonplace.

Future work might include implementing and benchmarking Winograd algorithms on modern CPUs and comparing their performance against FFT-based convolution. FFT-based convolutions show satisfactory performance on modern CPUs (which have larger caches but smaller memory bandwidths than modern GPUs).

The proposed LPL system could be deployed and evaluated on multiple types of low-complexity computers and devices, example smartphones, tablets, embedded system, etc. Corresponding accuracies, hardware utilization, and processing speeds could be compared.

For the real-time recognition system proposed in Chapters V and VI, further decrease the system's processing time is planned by interleaving predictions with tracking algorithms like IOU, KLT, and NVDCF. Thus, once an object is detected via the CNN model, the system could simply follow the object with a tracking algorithm, with a new DL prediction occurring in regular but less-frequent intervals. This would lessen the number of frames requiring compute-expensive DL prediction, thereby freeing resources that could be, for example, used to simultaneously process video streams from multiple cameras (multi-streaming inputs).

References

[1] E. Stawiarska and P. Sobczak, "The Impact of Intelligent Transportation System Implementations on the Sustainable Growth of Passenger Transport in EU Regions," *Sustainability*, vol. 10, no. 5, 2018, doi: 10.3390/su10051318.

[2] X. Xu *et al.*, "ITS-frame: A framework for multi-aspect analysis in the field of intelligent transportation systems," *IEEE Trans. Intell. Transp. Syst.*, vol. 20, no. 8, pp. 2893–2902, 2019, doi: 10.1109/TITS.2018.2868840.

[3] S. Sunkari, R. Parker, H. Charara, T. Palekar, and D. Middleston, "Evaluation of Cost-Effective Technologies for Advance Detection," in *Security*, 2005, vol. 7, no. 2.

[4] D. Kim and S. Park, "An intelligent collaboration framework between edge camera and video analysis system," in *2018 International Conference on Electronics, Information, and Communication (ICEIC)*, 2018, pp. 1–3, doi: 10.23919/ELINFOCOM.2018.8330653.

[5] R. Abduljabbar, H. Dia, S. Liyanage, and S. A. Bagloee, "Applications of Artificial Intelligence in Transport : An Overview," 2019, doi: 10.3390/su11010189.

[6] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–27, 2009, doi: 10.1561/2200000006.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016, vol. 2016-Decem, pp. 770–778, doi: 10.1109/CVPR.2016.90.

[8] H.-C. Shin *et al.*, "Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning," *IEEE Trans. Med. Imaging*, vol. 35, no. 5, pp. 1285–1298, 2016, doi: 10.1109/TMI.2016.2528162.

[9] L. Baresi, D. Filgueira Mendonça, and M. Garriga, "Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture," in *Service-Oriented and Cloud Computing*, 2017, pp. 196–210.

[10] X.-D. Zhang, "Machine Learning," in *A Matrix Algebra Approach to Artificial Intelligence*, Singapore: Springer Singapore, 2020, pp. 223–440.

[11] J. Cong and B. Xiao, "Minimizing Computation in Convolutional Neural Networks," in *Artificial Neural Networks and Machine Learning -- ICANN 2014*, 2014, pp. 281–290.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and*

Pattern Recognition, 2016, vol. 2016-Decem, pp. 770–778, doi: 10.1109/CVPR.2016.90.

[13] K. Fukushima, "Neocognitron for handwritten digit recognition," *Neurocomputing*, vol. 51, pp. 161–180, 2003, doi: 10.1016/S0925-2312(02)00614-8.

[14] A. D. Nguyen, S. Choi, W. Kim, S. Ahn, J. Kim, and S. Lee, "Distribution Padding in Convolutional Neural Networks," *Proc. - Int. Conf. Image Process. ICIP*, vol. 2019-September, pp. 4275–4279, 2019, doi: 10.1109/ICIP.2019.8803537.

[15] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520, doi: 10.1109/CVPR.2018.00474.

[16] A. G. Howard *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv Prepr. arXiv1704.04861*, 2017, doi: arXiv:1704.04861.

[17] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned CP-decomposition," in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015, pp. 1–11.

[18] D. Yu and M. L. Seltzer, "Improved bottleneck features using pretrained deep neural networks," *Proc. Annu. Conf. Int. Speech Commun. Assoc. INTERSPEECH*, no. August, pp. 237–240, 2011.

[19] M. S. Mohammed, S. L. Matilia, and L. Nozal, "Fast 2D convolution filter based on look up table FFT," in *[1992] Proceedings of the IEEE International Symposium on Industrial Electronics*, 1992, pp. 446–449 vol.1, doi: 10.1109/ISIE.1992.279637.

[20] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, "Accelerating Convolutional Neural Network with FFT on Embedded Hardware," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 26, no. 9, pp. 1737–1749, 2018, doi: 10.1109/TVLSI.2018.2825145.

[21] A. Lavin and S. Gray, "Fast Algorithms for Convolutional Neural Networks," *2016 IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 4013–4021, 2016, doi: 10.1109/CVPR.2016.435.

[22] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, "Towards a Uniform Template-based Architecture for Accelerating 2D and 3D CNNs on FPGA," *Proc. 2018 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays - FPGA '18*, pp. 97–106, 2018, doi: 10.1145/3174243.3174257.

[23] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," *Proc. - IEEE 25th Annu. Int. Symp. Field-Programmable Cust. Comput. Mach. FCCM 2017*, pp. 101–108, 2017, doi: 10.1109/FCCM.2017.64.

[24] Y. Huang, J. Shen, Z. Wang, M. Wen, and C. Zhang, "A High-efficiency FPGA-based Accelerator for Convolutional Neural Networks using Winograd Algorithm," *J. Phys. Conf. Ser.*, vol. 1026, no. 1, 2018, doi: 10.1088/1742-6596/1026/1/012019.

[25] U. Aydonat, S. O. Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL Deep Learning Accelerator on Arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 55–64.

[26] J. Yu *et al.*, "Instruction Driven Cross-Layer CNN Accelerator with Winograd Transformation on FPGA," in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 227–230.

[27] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," *Proc. Annu. Int. Symp. Microarchitecture, MICRO*, vol. 2016-Decem, 2016, doi: 10.1109/MICRO.2016.7783725.

[28] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," *FPGA 2016 - Proc. 2016 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, pp. 26–35, 2016, doi: 10.1145/2847263.2847265.

[29] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y. W. Tai, "Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs," *Proc. - Des. Autom. Conf.*, vol. Part 12828, pp. 1–6, 2017, doi: 10.1145/3061639.3062244.

[30] Z. Q. Zhao, P. Zheng, S. T. Xu, and X. Wu, "Object Detection with Deep Learning: A Review," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 30, no. 11, pp. 3212–3232, 2019, doi: 10.1109/TNNLS.2018.2876865.

[31] L. Fei-Fei, J. Deng, and K. Li, "ImageNet: Constructing a large-scale image database," *J. Vis.*, vol. 9, no. 8, pp. 1037–1037, 2010, doi: 10.1167/9.8.1037.

[32] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *36th International Conference on Machine Learning, ICML 2019*, 2019, vol. 2019-June, pp. 10691–10700.

[33] S. Wu, S. Zhong, and Y. Liu, "Deep residual learning for image steganalysis," in *Multimedia Tools and Applications*, 2017, pp. 1–17, doi: 10.1007/s11042-017-4440-4.

[34] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," *Proc. - 30th IEEE Conf. Comput. Vis. Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 2261–2269, 2017, doi: 10.1109/CVPR.2017.243.

[35] A. Gholami *et al.*, "SqueezeNext: Hardware-aware neural network design," 2018, doi: 10.1109/CVPRW.2018.00215.

[36] A. Howard *et al.*, "Searching for mobileNetV3," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, vol. 2019-Octob, pp. 1314–1324, doi: 10.1109/ICCV.2019.00140.

[37] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6848–6856, doi: 10.1109/CVPR.2018.00716.

[38] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2019, pp. 1–16.

[39] Y. Xiong *et al.*, "MobileDets: Searching for Object Detection Architectures for Mobile Accelerators," 2020, [Online]. Available: http://arxiv.org/abs/2004.14525.

[40] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2014, pp. 580–587, doi: 10.1109/CVPR.2014.81.

[41] R. Girshick, "Fast R-CNN," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, vol. 2015 Inter, pp. 1440–1448, doi: 10.1109/ICCV.2015.169.

[42] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, 2017, doi: 10.1109/TPAMI.2016.2577031.

[43] J. Dai, Y. Li, K. He, and J. Sun, "R-FCN: Object detection via region-based fully convolutional networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 379–387.

[44] J. Pang, K. Chen, J. Shi, H. Feng, W. Ouyang, and D. Lin, "Libra R-CNN: Towards balanced learning for object detection," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2019, vol. 2019-June, pp. 821–830, doi: 10.1109/CVPR.2019.00091.

[45] Z. Yang, S. Liu, H. Hu, L. Wang, and S. Lin, "Reppoints: Point set representation for object detection," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, vol. 2019-Octob, pp. 9656–9665, doi: 10.1109/ICCV.2019.00975.

[46] M. Tan, R. Pang, and Q. V. Le, "EfficientDet: Scalable and efficient object detection," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2020, pp. 10778–10787, doi: 10.1109/CVPR42600.2020.01079.

[47] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788, doi: 10.1109/CVPR.2016.91.

[48] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, vol. 9905 LN, pp. 21–37, doi:10.1007/978-3-319-46448-0_2.

[49] T. Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal Loss for Dense Object Detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 2, pp. 318–327, 2020, doi: 10.1109/TPAMI.2018.2858826.

[50] T. Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017, vol. 2017-Janua, pp. 936–944, doi: 10.1109/CVPR.2017.106.

[51] G. Ghiasi, T. Y. Lin, and Q. V. Le, "NAS-FPN: Learning scalable feature pyramid architecture for object detection," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2019-June, pp. 7029–7038, 2019, doi: 10.1109/CVPR.2019.00720.

[52] J. König, S. Malberg, M. Martens, S. Niehaus, A. Krohn-Grimberghe, and A. Ramaswamy, "Multi-stage Reinforcement Learning for Object Detection," in *Advances in Intelligent Systems and Computing*, 2020, vol. 943, pp. 178–191, doi: 10.1007/978-3-030-17795-9_13.

[53] J. Wang, M. Zhu, D. Sun, B. Wang, W. Gao, and H. Wei, "MCF3D: Multi-Stage Complementary Fusion for Multi-Sensor 3D Object Detection," *IEEE Access*, vol. 7, pp. 90801–90814, 2019, doi: 10.1109/ACCESS.2019.2927012.

[54] S. Yonetsu, Y. Iwamoto, and Y. W. Chen, "Two-Stage YOLOv2 for Accurate License-Plate Detection in Complex Scenes," in *2019 IEEE International Conference on Consumer Electronics, ICCE 2019*, 2019, pp. 1–4, doi: 10.1109/ICCE.2019.8661944.

[55] J. Zhang, Y. Li, T. Li, L. Xun, and C. Shan, "License Plate Localization in Unconstrained Scenes Using a Two-Stage CNN-RNN," *IEEE Sens. J.*, vol. 19, no. 13, pp. 5256–5265, 2019, doi: 10.1109/JSEN.2019.2900257.

[56] C. Cao *et al.*, "An Improved Faster R-CNN for Small Object Detection," *IEEE Access*, vol. 7, pp. 106838–106846, 2019, doi: 10.1109/ACCESS.2019.2932731.

[57] S. Wang, H. Lu, and Z. Deng, "Fast object detection in compressed video," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, vol. 2019-Octob, pp. 7103–7112, doi: 10.1109/ICCV.2019.00720.

[58] J. Deng, Y. Pan, T. Yao, W. Zhou, H. Li, and T. Mei, "Single Shot Video Object Detector," *IEEE Trans. Multimed.*, vol. 1, no. c, pp. 1–1, 2020, doi: 10.1109/tmm.2020.2990070.

[59] C. Zhang and J. Kim, "Modeling Long- and Short-Term Temporal Context for Video Object Detection," *Proc. - Int. Conf. Image Process. ICIP*, vol. 2019-Septe, pp. 71–75, 2019, doi: 10.1109/ICIP.2019.8802920.

[60] M. Liu, M. Zhu, M. White, Y. Li, and D. Kalenichenko, "Looking Fast and Slow: Memory-Guided Mobile Video Object Detection," *arXiv*, 2019, [Online]. Available: http://arxiv.org/abs/1903.10172.

[61] A. Romero, C. Gatta, G. Camps-valls, and S. Member, “Unsupervised Deep Feature Extraction for Remote Sensing Image Classification,” *IEEE Trans. Geosci. Remote Sens.*, vol. 54, no. 3, pp. 1349–1362, 2016.

[62] S. Chen and C. Tao, “PolSAR Image Classification Using Polarimetric-Feature-Driven Deep Convolutional Neural Network,” *IEEE Geosci. Remote Sens. Lett.*, vol. 15, no. 4, pp. 627–631, 2018.

[63] L. Jiao, S. Member, and F. Liu, “Wishart Deep Stacking Network for Fast POLSAR Image Classification,” *IEEE Trans. Image Process.*, vol. 25, no. 7, pp. 3273–3286, 2016.

[64] S. D. H. Forests, Y. Yu, H. Guan, and Z. Ji, “Rotation-Invariant Object Detection in High-Resolution Satellite Imagery Using Superpixel-Based Deep Hough Forests,” *IEEE Geosci. Remote Sens. Lett.*, vol. 12, no. 11, pp. 2183–2187, 2015.

[65] T. Le, “Video Salient Object Detection Using Spatiotemporal Deep Features,” *IEEE Trans. Image Process.*, vol. 27, no. 10, pp. 5002–5015, 2018, doi: 10.1109/TIP.2018.2849860.

[66] J. Han, D. Zhang, X. Hu, L. Guo, J. Ren, and F. Wu, “Background Prior-Based Salient Object Detection via Deep Reconstruction Residual,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 8, pp. 1309–1321, 2015.

[67] B. Wu *et al.*, “An End-to-End Deep Learning Approach to Simultaneous Speech Dereverberation and Acoustic Modeling for Robust Speech Recognition,” *IEEE J. Sel. Top. Signal Process.*, vol. 11, no. 8, pp. 1289–1300, 2017.

[68] P. Zhou, H. Jiang, S. Member, L. Dai, Y. Hu, and Q. Liu, “State-Clustering Based Multiple Deep Neural Networks Modeling Approach for Speech Recognition,” *IEEE/ACM Trans. Audio, Speech, Lang. Process.*, vol. 23, no. 4, pp. 631–642, 2015.

[69] S. Park, J. Park, S. Member, and K. Bong, “An Energy-Efficient and Scalable Deep Learning / Inference Processor With Tetra-Parallel MIMD Architecture for Big Data Applications,” *IEEE Trans. Biomed. Circuits Syst.*, vol. 9, no. 6, pp. 838–848, 2015.

[70] M. Alam, L. S. Vidyaratne, and K. M. Iftekharuddin, “Robust Facial Expression Recognition,” *IEEE Trans. Neural Networks Learn. Syst.*, vol. 29, no. 10, pp. 4905–4916, 2018, doi: 10.1109/TNNLS.2017.2776248.

[71] S. Winograd, *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, 1980.

[72] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller, “Striving for Simplicity: The All Convolutional Net,” 2015, [Online]. Available: http://arxiv.org/abs/1412.6806.

[73] J. Hannink, T. Kautz, C. F. Pasluosta, J. Barth, and S. Sch, “Mobile Stride Length

Estimation With Deep Convolutional Neural Networks," *IEEE J. Biomed. Heal. Informatics*, vol. 22, no. 2, pp. 354–362, 2018.

[74] C. Ding, D. Pei, and A. Salomaa, *Chinese remainder theorem: applications in computing, coding, cryptography*. USA: World Scientific Publishing Co., Inc., 1996.

[75] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[76] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[77] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," in *International Conference on Learning Representations ICLR 2017*, 2017, pp. 1–13, doi: 10.1007/978-3-319-24553-9.

[78] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks BT - Computer Vision – ECCV 2014," 2014, pp. 818–833.

[79] W. Liu *et al.*, "SSD: Single Shot MultiBox Detector BT - Computer Vision – ECCV," 2016, pp. 21–37.

[80] M. Sandler, M. Zhu, A. Zhmoginov, and C. V Apr, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," *arXiv:1801.04381v3*, 2018.

[81] A. Howard *et al.*, "Searching for MobileNetV3," 2019, [Online]. Available: http://arxiv.org/abs/1905.02244.

[82] "Pre-trained models." https://github.com/tensorflow/models/tree/master/research/.

[83] X. Wei *et al.*, "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs," *Proc. - Des. Autom. Conf.*, vol. Part 12828, 2017, doi: 10.1145/3061639.3062207.

[84] Y. R. Wang, W. H. Lin, and S. J. Horng, "A sliding window technique for efficient license plate localization based on discrete wavelet transform," *Expert Syst. Appl.*, vol. 38, no. 4, pp. 3142–3146, 2011, doi: 10.1016/j.eswa.2010.08.106.

[85] S. Zhu, S. Dianat, and L. K. Mestha, "End-to-end system of license plate localization and recognition," *J. Electron. Imaging*, vol. 24, no. 2, p. 023020, 2015, doi: 10.1117/1.JEI.24.2.023020.

[86] S. Du, M. Ibrahim, M. Shehata, and W. Badawy, "Automatic license plate recognition (ALPR): A state-of-the-art review," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 23, no. 2, pp. 311–325, 2013, doi: 10.1109/TCSVT.2012.2203741.

[87] S. G. Kim, H. G. Jeon, and H. I. Koo, "Deep-learning-based license plate detection

method using vehicle region extraction," *Electron. Lett.*, vol. 53, no. 15, pp. 1034–1036, 2017, doi: 10.1049/el.2017.1373.

[88] Q. Fu, Y. Shen, and Z. Guo, "License Plate Detection Using Deep Cascaded Convolutional Neural Networks in Complex Scenes," in *Neural Information Processing*, 2017, pp. 696–706.

[89] A. Naimi, Y. Kessentini, and M. Hammami, "Multi-nation and Multi-norm License Plates Detection in Real Traffic Surveillance Environment Using Deep Learning," in *Neural Information Processing*, 2016, pp. 462–469.

[90] L. Xie, T. Ahmad, L. Jin, Y. Liu, and S. Zhang, "A New CNN-Based Method for Multi-Directional Car License Plate Detection," *IEEE Trans. Intell. Transp. Syst.*, vol. 19, no. 2, pp. 507–517, 2018, doi: 10.1109/TITS.2017.2784093.

[91] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," pp. 1–14, 2014, [Online]. Available: http://arxiv.org/abs/1409.1556.

[92] Tzutalin, "LabelImg," 2015. https://github.com/tzutalin/labelImg.

[93] W. Wu, Y. Liu, W. Zeng, M. Guo, C. Wang, and X. Liu, "Effective constructing training sets for object detection," in *2013 IEEE International Conference on Image Processing*, 2013, pp. 3377–3380, doi: 10.1109/ICIP.2013.6738696.

[94] J. Konar, P. Khandelwal, and R. Tripathi, "Comparison of Various Learning Rate Scheduling Techniques on Convolutional Neural Network," in *2020 IEEE International Students' Conference on Electrical,Electronics and Computer Science (SCEECS)*, 2020, pp. 1–5, doi: 10.1109/SCEECS48394.2020.94.

[95] A. Chamarty, "Fine-Tuning of Learning Rate for Improvement of Object Detection Accuracy," in *2020 IEEE India Council International Subsections Conference (INDISCON)*, 2020, pp. 135–141, doi: 10.1109/INDISCON50162.2020.00038.

[96] "Caltech: 'Computational vision: archive.'" http://www.vision.caltech.edu/html-files/archive.html.

[97] H. Li, P. Wang, and C. Shen, "Toward End-to-End Car License Plate Detection and Recognition With Deep Neural Networks," *IEEE Trans. Intell. Transp. Syst.*, vol. 20, no. 3, pp. 1126–1136, 2019, doi: 10.1109/TITS.2018.2847291.

[98] Y. Yuan, W. Zou, Y. Zhao, X. Wang, X. Hu, and N. Komodakis, "A Robust and Efficient Approach to License Plate Detection," vol. 26, no. 3, pp. 1102–1114, 2017, doi: 10.1109/TIP.2016.2631901.

[99] W. Zhou, H. Li, Y. Lu, and Q. Tian, "Principal Visual Word Discovery for Automatic License Plate Detection," *IEEE Trans. Image Process.*, vol. 21, no. 9, pp. 4269–4279, 2012, doi: 10.1109/TIP.2012.2199506.

[100] "Kalafatić, Z.: 'License plate detection, recognition.'" http://www.zemris.fer.hr/projects/LicensePlates/english/.

[101] M. R. Asif *et al.*, "Efficient method for vehicle license plate identification based on learning a morphological feature," *IET Intell. Transp. Syst.*, vol. 10, no. 6, pp. 389–395, 2016, doi: 10.1049/iet-its.2015.0064.

[102] MediaLab of National Technical University of Athens, "Medialab Lpr Database." http://www.medialab.ntua.gr/research/LPRdatabase.html.

[103] S. Mao, X. Huang, and M. Wang, "An adaptive method for Chinese license plate location," *Proc. World Congr. Intell. Control Autom.*, pp. 6173–6177, 2010, doi: 10.1109/WCICA.2010.5554434.

[104] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *CoRR*, vol. abs/1612.0, 2016.

[105] A. Slavich and J. E. Daust, "Commercial Vehicle Safety Alliance (CVSA)/Department of Energy (DOE) cooperative agreement final report," doi: 10.2172/758961.

[106] R. D. Castro-Zunti, J. Yépez, and S. B. Ko, "License plate segmentation and recognition system using deep learning and OpenVINO," *IET Intell. Transp. Syst.*, vol. 14, no. 2, pp. 119–126, 2020, doi: 10.1049/iet-its.2019.0481.

[107] J. Yépez, R. D. Castro-Zunti, and S.-B. Ko, "Deep learning-based embedded license plate localisation system," *IET Intell. Transp. Syst.*, vol. 13, no. 10, pp. 1569–1578, 2019, doi: 10.1049/iet-its.2019.0082.

[108] Z. Cai and N. Vasconcelos, "Cascade R-CNN: High Quality Object Detection and Instance Segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, pp. 1–1, 2019, doi: 10.1109/tpami.2019.2956516.

[109] T. Y. Lin *et al.*, "Microsoft COCO: Common objects in context," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014, vol. 8693 LNCS, no. PART 5, pp. 740–755, doi: 10.1007/978-3-319-10602-1_48.

[110] P. Fang and Y. Shi, "Small object detection using context information fusion in faster R-CNN," in *2018 IEEE 4th International Conference on Computer and Communications, ICCC 2018*, 2018, pp. 1537–1540, doi: 10.1109/CompComm.2018.8780579.

[111] J. Li *et al.*, "Multistage Object Detection with Group Recursive Learning," *IEEE Trans. Multimed.*, vol. 20, no. 7, pp. 1645–1655, 2018, doi: 10.1109/TMM.2017.2772796.

[112] L. Puglia, M. Vigliar, and G. Raiconi, "Real-Time Low-Power FPGA Architecture for Stereo Vision," *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 64, no. 11, pp. 1307–1311, 2017, doi: 10.1109/TCSII.2017.2691675.

[113] J. Yepez and S. B. Ko, “Stride 2 1-D, 2-D, and 3-D Winograd for Convolutional Neural Networks,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 28, no. 4, pp. 853–863, 2020, doi: 10.1109/TVLSI.2019.2961602.

[114] “International Road Dynamics Inc.,” 2020. https://www.irdinc.com.

[115] R. Mittal and A. Garg, “Text extraction using OCR: A Systematic Review,” in *2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA)*, 2020, pp. 357–362, doi: 10.1109/ICIRCA48905.2020.9183326.

[116] M. Asadikouhanjani, H. Zhang, L. Gopalakrishnan, H. J. Lee, and S. B. Ko, “A Real-Time Architecture for Pruning the Effectual Computations in Deep Neural Networks,” *IEEE Trans. Circuits Syst. I Regul. Pap.*, pp. 1–12, 2021, doi: 10.1109/TCSI.2021.3060945.

[117] H. Halawa, H. A. Abdelhafez, A. Boktor, and M. Ripeanu, “NVIDIA jetson platform characterization,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, vol. 10417 LNCS, pp. 92–105, doi: 10.1007/978-3-319-64203-1_7.

[118] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The pascal visual object classes (VOC) challenge,” *Int. J. Comput. Vis.*, vol. 88, no. 2, pp. 303–338, 2010, doi: 10.1007/s11263-009-0275-4.

[119] I. U. Ahmed, S. M. Gaweesh, and M. M. Ahmed, “Exploration of Hazardous Material Truck Crashes on Wyoming’s Interstate Roads using a Novel Hamiltonian Monte Carlo Markov Chain Bayesian Inference,” *Transp. Res. Rec.*, vol. 2674, no. 9, pp. 661–675, 2020, doi: 10.1177/0361198120931103.

[120] *Globally harmonized system of classification and labelling of chemicals (GHS)*. New York and Geneva, 2019.

[121] E. Goforth, M. Ezzeldin, W. El-Dakhakhni, L. Wiebe, and M. Mohamed, “Network-of-Networks Framework for Multimodal Hazmat Transportation Risk Mitigation: Application to Used Nuclear Fuel Transportation,” *J. Hazardous, Toxic, Radioact. Waste*, vol. 24, p. 4020016, 2020, doi: 10.1061/(ASCE)HZ.2153-5515.0000493.

[122] “RoboCup Rescue Rulebook,” *RoboCup Rescue*, 2019. https://rrl.robocup.org/wp-content/uploads/2019/06/rrl_rulebook_2019_v2.4.pdf.

[123] B. Chen *et al.*, “MnasFPN: Learning latency-aware pyramid architecture for object detection on mobile devices,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 13604–13613, 2020, doi: 10.1109/CVPR42600.2020.01362.

[124] S. Zherzdev and A. Gruzdev, “LPRNet: License Plate Recognition via Deep Neural Networks,” pp. 1–6, 2018, [Online]. Available: http://arxiv.org/abs/1806.10447.

[125] X. Zhang, S. Huang, X. Zhang, W. Wang, Q. Wang, and D. Yang, “Residual Inception:

A New Module Combining Modified Residual with Inception to Improve Network Performance," in *2018 25th IEEE International Conference on Image Processing (ICIP)*, 2018, pp. 3039–3043, doi: 10.1109/ICIP.2018.8451515.

[126] M.-J. Chae *et al.*, "Convolutional Sequence to Sequence Model with Non-Sequential Greedy Decoding for Grapheme to Phoneme Conversion," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018, pp. 2486–2490, doi: 10.1109/ICASSP.2018.8462678.

[127] S. Xie *et al.*, "Speed/accuracy trade-offs for modern convolutional object detectors," *arXiv Prepr. arXiv1512.03385*, 2015.

[128] J. Moolayil, "An Introduction to Deep Learning and Keras," in *Learn Keras for Deep Neural Networks: A Fast-Track Approach to Modern Deep Learning with Python*, Berkeley, CA: Apress, 2019, pp. 1–16.

[129] I. Loshchilov and F. Hutter, "SGDR: Stochastic gradient descent with warm restarts," *5th Int. Conf. Learn. Represent. ICLR 2017 - Conf. Track Proc.*, pp. 1–16, 2017.

[130] P. Goyal *et al.*, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," 2017, [Online]. Available: http://arxiv.org/abs/1706.02677.

[131] T.-Y. Lin *et al.*, "Microsoft COCO: Common Objects in Context," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, 2015, doi: 10.1109/CVPR.2014.471.

[132] W. Taymans, S. Baker, A. Wingo, R. S. Bultje, and S. Kost, "GStreamer application development manual," 2013.

[133] A. Patait and E. Young, "High performance video encoding with NVIDIA GPUs," 2016.

[134] N. Capodieci, R. Cavicchioli, I. S. Olmedo, M. Solieri, and M. Bertogna, "Contending memory in heterogeneous SoCs: Evolution in NVIDIA Tegra embedded platforms," in *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2020, pp. 1–10, doi: 10.1109/RTCSA50079.2020.9203722.

[135] M. Ditty, A. Karandikar, and D. Reed, "NVIDIA ' S XAVIER SOC Xavier — Designed for the next wave of Autonomous Machines," in *Proc. IEEE Hot Chips Symp. (HCS)*, 2018, pp. 1–17.

[136] M. Toksvig, P. Sriram, J. Matheson, B. Cabral, and B. Smith, "NVIDIA Tegra," in *2008 IEEE Hot Chips 20 Symposium (HCS)*, 2008, pp. 1–28, doi: 10.1109/HOTCHIPS.2008.7476540.

[137] F. Farshchi, Q. Huang, and H. Yun, "Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim," in *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*,

2019, pp. 21–25, doi: 10.1109/EMC249363.2019.00012.

[138] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, “Real-Time Computer Vision with OpenCV,” *Commun. ACM*, vol. 55, no. 6, pp. 61–69, Jun. 2012, doi: 10.1145/2184319.2184337.

[139] J. Cai, J. Hou, Y. Lu, H. Chen, L. Kneip, and S. Schwertfeger, “Improving CNN-based Planar Object Detection with Geometric Prior Knowledge,” in *2020 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 2020, pp. 387–393, doi: 10.1109/SSRR50563.2020.9292601.

[140] M. A. Mohamed, J. Tünnermann, and B. Mertsching, “Seeing Signs of Danger: Attention-Accelerated Hazmat Label Detection,” in *2018 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 2018, pp. 1–6, doi: 10.1109/SSRR.2018.8468639.