MACHINE LEARNING AND DEEP LEARNING ALGORITHMS IN THERMAL IMAGING VEHICLE PERCEPTION SYSTEMS

MACHINE LEARNING AND DEEP LEARNING ALGORITHMS IN THERMAL IMAGING VEHICLE PERCEPTION SYSTEMS

BY

JIAHONG DONG, B.Eng.Mgmt

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

© Copyright by Jiahong Dong, May 2021

All Rights Reserved

Master of Applied Science (2021)	McMaster University
(Computing and Software)	Hamilton, Ontario, Canada

TITLE: Machine Learning and Deep Learning Algorithms in Thermal Imaging Vehicle Perception Systems

AUTHOR: Jiahong Dong
B.Eng.Mgmt (Software Engineering & Management),
McMaster University, Hamilton, Canada

SUPERVISOR: Dr. Martin Von. Mohrenschildt & Dr. Saeid Habibi

NUMBER OF PAGES: xix, 180

Abstract

Modern Advanced Driver Assistant Systems (ADAS) focus more on daytime driving and primarily use daylight cameras as the main vision sources to detect, classify, and track objects. However, evidence has proved that autonomous driving using such a setup is compromised in the dark, and consequently, resulting in accidents. The hypothesis is that adding an infrared camera to the existing ADAS will boost the detection rate and accuracy, and further enhance the overall safety. This thesis investigates how well a standalone infrared camera performs onboard vehicle perception tasks such as object detection and classification using both machine learning and deep learning algorithms. Given a custom labeled infrared driving dataset that contains 4 classes of objects, "People", "Vehicle", "Bicycle", and "Animal", multiple attempts and improvements of training a supervised learning model, namely the linear multi-class Support Vector Machine (SVM) have been made by using various image preprocessing and feature extraction methods to detect the objects. During training, hard example mining is used to reduce the number of false classifications. This SVM employs a One-Against-All (OAA) styled approach and uses the image pyramid technique to enable multi-scale detection. On the deep learning side, a Convolutional Neural Network (CNN) based state-of-the-art detector, the YOLOv4 family including the full-sized and tiny YOLOv4 has been selected, trained, and tested at different

input sizes using the same dataset. Labeling format conversion is performed to make this work. The results show that using bilateral filtering with the Histogram of Oriented Gradients (HOG) feature to train an SVM is preferable and is more accurate than the YOLOv4 family. However, the YOLOv4 networks are significantly faster. Overall, a standalone infrared camera cannot provide dominant detection results, but it can definitely supply useful information to the ADAS and complement other sensory devices for improved safety.

"I could either watch it happen or be a part of it."

– *Elon Musk*

Acknowledgements

First and foremost, I would like to show my sincere gratitude for the support and generosity of my supervisors, Dr. Martin von Mohrenschildt and Dr. Saeid Habibi. Dr. Martin von Mohrenschildt's mathematical background and his programming expertise have prepared me well for this research. He started my journey to AI and provided me with an opportunity to seek my academic goals as a Master's student. His technical and mental support was essential to the completion of this thesis. I also greatly appreciate Dr. Saeid Habibi for all the inspiration, advice, and affirmation throughout my research. Without Dr. Saeid Habibi's time, ideas, and the platform he provided me with at the CMHT, I would not be able to practice my knowledge and be where I am now. Words cannot express my feelings, nor my thanks for all their help, their thoughtfulness and encouragement have been and will always be my biggest motivation.

My sincere thanks also go to Mr. Cam Fisher and my colleagues at the CMHT. They have been an integral part of my research and made my journey enjoyable and fascinating. Without their coordination, collaboration, and technical support, this research would not have been possible.

Last but not least, I would like to thank my parents and families for their love and all the mental and financial support throughout my overseas studies.

Special thanks to all my friends in Canada, this journey would not have been possible without your accompaniment.

Contents

Abstract iii

Acknowledgements vi

List of Figures xi

List of Tables xv

Abbreviations xvi

1 Introduction 1

1.1 Contributions 2

1.2 The McMaster Center for Mechatronics and Hybrid Technologies (CMHT) 4

1.3 Preliminary Remarks 5

1.4 The Levels of Autonomy 6

1.5 The History of Autonomous Vehicles 7

1.6 Some Recent Autonomous Accidents 10

1.7 The Sensors 11

1.8 The Latest Trend of Autonomous Vehicle 17

1.9 Thesis Outline 18

2 Literature Review **20**
2.1 Thermal Imaging . . . 20
2.2 The Algorithms . . . 21

3 Camera Calibration **61**
3.1 The Camera Parameters . . . 62
3.2 The Implementation . . . 65
3.3 A Creative Calibration Method for Infrared Cameras . . . 66
3.4 The Calibration Result . . . 70

4 A You Only Look Once (YOLO) Based Approach **72**
4.1 Image Preprocessing . . . 73
4.2 Data Augmentation . . . 74
4.3 Feature Extraction . . . 79
4.4 Transfer Learning . . . 79
4.5 Dataset Preparation and Labelling . . . 80
4.6 Training a YOLOv4 Network . . . 84
4.7 Training a Tiny-YOLOv4 Network . . . 86
4.8 Running YOLOv4 and Tiny-YOLOv4 . . . 88

5 A Support Vector Machine (SVM) Based Approach **92**
5.1 Color Space Conversion . . . 93
5.2 Noise Reduction . . . 94
5.3 Filtering Application Results . . . 100
5.4 Feature Extraction . . . 103
5.5 Method Selection . . . 120

5.6 Dataset Preparation and Labelling 121
5.7 Decision Making Using Windowing 123
5.8 Training a Support Vector Machine 127
5.9 Running the SVM on Full Images and Comparing with the YOLOv4 141

6 Conclusion and Future Work 149

6.1 Conclusion 149
6.2 Future Work 154
6.3 Closing Remark 159

A Python Code 160

Bibliography 166

List of Figures

1.1 The experiment vehicle at the CMHT 5

1.2 The American Wonder in 1925 8

1.3 Tesla's front-facing camera module 12

1.4 The FLIR A65 thermal camera 13

1.5 BOSCH's 77GHz front radar sensor 14

1.6 The Luminar LiDAR on the TRI's research vehicle 15

1.7 The ultrasonic parking sensor on modern vehicles 15

1.8 A typical GPS satnav system on modern vehicles 16

1.9 Baidu's autonomous vehicle is delivering food and supplies to a hospital 18

2.1 Intersection over Union 23

2.2 The components of a linear binary SVM 31

2.3 The hinge loss 33

2.4 The operation on each neuron 37

2.5 The binary step (green) function and its gradient (red) 40

2.6 The linear function (green) and its gradient (red) 41

2.7 The sigmoid function (green) and its gradient (red) 42

2.8 The Tanh function (green) and its gradient (red) 43

2.9 The ReLu (green), Leaky ReLu (purple), PReLu (red), and ELu (blue) function . . . 44
2.10 The movement of a 3×3 RGB kernel . . . 46
2.11 The architecture of LeNet-5 . . . 48
2.12 The architecture of AlexNet . . . 49
2.13 GoogLeNet's inception module . . . 50
2.14 The architecture of VGG-16 . . . 51
2.15 The architecture of ResNet . . . 51
2.16 The architrcture of YOLOv1 . . . 54
2.17 The prediction of a bounding box in YOLOv2 . . . 56
2.18 The Feature Pyramid Network . . . 57
2.19 The CSPDarknet-53 . . . 58
2.20 The FPN and PAN . . . 59
2.21 The YOLOv4's architecture . . . 60
3.1 The ideal pinhole camera . . . 62
3.2 The pixel skew . . . 63
3.3 Heating the checkerboard requires careful timing . . . 66
3.4 The checkerboard used in this experiment . . . 67
3.5 An example of a properly heated checkerboard . . . 68
3.6 The orientation of the images during calibration . . . 69
3.7 The detected corners on the checkerboard . . . 69
3.8 The world coordinates . . . 71
4.1 The CutMix . . . 75
4.2 The Mosaic data augmentation . . . 77

4.3 Vehicles from different viewpoints in an infrared camera 81
4.4 An example of a labeled image frame 82
4.5 Running the YOLOv4 and Tiny-YOLOv4 on an image 89
5.1 Grayscaling an image 94
5.2 The Gaussian distribution 96
5.3 The effect of each denoising method 102
5.4 Applying the Sobel edge detector 105
5.5 The partitions of the reduced gradient direction 107
5.6 Using the Canny edge detector without hysteresis thresholding 108
5.7 Properly applying the Canny edge detector with hysteresis thresholding 109
5.8 Applying the Roberts operator 110
5.9 Applying the Prewitt operator 111
5.10 The second-order derivative crosses zero at the edge 112
5.11 The results of using the Laplacian operator 113
5.12 Applying the LOG operator 115
5.13 Resizing the image patch to 1:2 aspect ratio 116
5.14 Dividing the image patch in to 8×8 cells 118
5.15 The 9-bin histogram 118
5.16 Applying the HOG descriptor 120
5.17 Visualizing the dataset after all bounding boxes have been collected. . 123
5.18 Cropping a patch from the original image 124
5.19 The windowing process 125
5.20 When objects appear at different scales in the window 126
5.21 The image pyramid 127

5.22 An image in the positive set 129

5.23 An image in the negative set 129

5.24 An example of the negative images 130

5.25 The K-means clustering results 131

5.26 The overall SVM training procedure 134

5.27 Before the Non-Max suppression. 143

5.28 After the Non-Max suppression 144

5.29 Running the SVM on full-sized 640×512 images. 145

A.1 The code snippet for the YOLO-OpenCV labelling format conversion 161

A.2 The code snippet for the hard example mining process 162

A.3 The code snippet for image preprocessing 163

A.4 The code snippet for feature extraction 164

A.5 The code snippet for training the SVM 165

List of Tables

4.1 The average processing time of each filtering method at different network sizes. . . . 74
4.2 The training results of a 320×320 full YOLOv4 network. . . . 85
4.3 The training results of a 416×416 full YOLOv4 network. . . . 85
4.4 The training results of a 608×608 full YOLOv4 network. . . . 85
4.5 The training results of a 320×320 tiny-YOLOv4 network. . . . 87
4.6 The training results of a 416×416 tiny-YOLOv4 network. . . . 87
4.7 The training results of a 608×608 tiny-YOLOv4 network. . . . 87
4.8 The average processing time (also converted to FPS) of each YOLOv4 network. . . . 91
5.1 Training on a balanced and an unbalanced dataset. . . . 133
5.2 The testing results of the unified linear binary SVM. . . . 135
5.3 The precision of each linear binary SVM in the second attempt. . . . 138
5.4 The recall of each linear binary SVM in the second attempt. . . . 139
5.5 The precision of each linear binary SVM in the third attempt. . . . 140
5.6 The recall of each linear binary SVM in the third attempt. . . . 141
5.7 Comparing the SVM with the YOLOv4 and tiny YOLOv4 network. . . . 146

Abbreviations

Abbreviations

ADAS	Advanced Driver Assistance Systems
MARC	McMaster Automotive Resource Centre
DL	Deep Learning
DNN	Deep Neural Network
ML	Machine Learning
SVM	Support Vector Machine
SVR	Support Vector Regression
OAA SVM	One-Against-All Support Vector Machine
OAO SVM	One-Against-One Support Vector Machine
YOLO	You Only Look Once
CMHT	Center for Mechatronics and Hybrid Technologies

ACES	Autonomous driving, Connected vehicles, Electrification, and Shared mobility
WHO	World Health Organization
SAE	Society of Automotive Engineers International
NHTSA	The United States National Highway Traffic Safety Administration
DARPA	Defense Advanced Research Projects Agency
VIAC	VisLab Intercontinental Autonomous Challenge
NTSB	National Transportation Safety Board
GPS	Global Positioning System
LiDAR	Light Detection and Ranging devices
FOV	Field of View
IR	Infrared
TRI	Toyota Research Institute
PNT	Positioning, Navigation, and Timing
ETA	Estimated Time of Arrival
EM	Electromagnetic
AP	Average Precision
mAP	mean Average Precision

IoU Intersection over Union

TP True Positive

TN True Negative

FP False Positive

FN False Negative

MS COCO Microsoft Common Objects in Context

CIFAR Canadian Institute For Advanced Researech

PASCAL VOC Pattern Analysis, Statistical Modelling and Computational Learning Visual Object Classes

ILSVRC ImageNet Large Scale Visual Recognition Competition

HOG Histogram of Oriented Gradients

RGB Red, Green, and Blue

RBF Gaussian Radial Based Function

SSE Sum Squared Error

MSE Mean Squared Error

ME Mean Error

MAE Mean Absolute Error

RMSE	Root Mean Squared Error
ReLU	Rectified Linear Unit
PReLu	Parameterised Rectified Linear Unit
ELu	Exponential Linear Unit
SGD	Stochastic Gradient Descent
RPN	Region Proposal Network
R-CNN	Region Based Convolutional Neural Network
FPN	Feature Pyramid Network
CSP	Cross Stage Partial
SPP	Spatial Pyramid Pooling
PAN	Path Aggregation Network
GAN	Generative Adversarial Network
SAT	Self-Adversarial Training
OpenCV	Open Source Computer Vision
BFLOPS	Billion Floating Point Operations Per Second
NMS	Non-Maxima Suppression

Chapter 1

Introduction

The autonomous vehicle, self-driving vehicle, and driverless vehicle usually refer to vehicles equipped with Advanced Driver-Assistance Systems (ADAS) that would not require a human driver to operate. These vehicles must be able to detect and classify objects in their vicinity, provide distance measurements, and operate using an autopilot. Sensors are one of the key components of ADAS, and ADAS offers safety features such as adaptive cruise control, autonomous emergency braking, lane-keeping assist, lane departure warning, blind-spot monitoring, steering assist, parking assist, and cross-traffic assist.

Modern ADAS heavily rely on regular day-light monocular or stereo cameras to coordinate the onboard object detection and classification tasks. However, daylight cameras have been proved unhelpful working in the dark or under foggy conditions which drastically reduces the robustness of the ADAS and makes detecting and classifying objects under these conditions nearly impossible. Although thermal cameras excel at nighttime operations, they are only available as options on some production vehicles, but not an essential part of the ADAS.

Moreover, almost all state-of-the-art detectors take the Deep Learning (DL) approach and use Deep Neural Networks (DNN) as the backbone. Traditional Machine Learning (ML) algorithms seem to be absent in the competition of state-of-the-art detectors. Therefore, in this thesis, at the McMaster Automotive Resource Centre (MARC), it is aimed to use both ML and DL algorithms to build detectors and study why traditional ML algorithms become less competent at performing perception tasks. Additionally, this thesis is also aimed to study the feasibility of infrared cameras becoming an integral part of the ADAS by using a standalone infrared camera to perform onboard object detection and classification.

1.1 Contributions

In this thesis, both machine learning and deep learning algorithms have been studied in regard to object detection and classification using a vehicle-mounted infrared camera. The resulting system was deployed on a vehicle and tested on the recordings. The real-time performance will be tested in the future. The major contribution to this research includes:

- Background research: During the background research, more than 100 published literature and articles on autonomous vehicles/systems from renowned online sources have been reviewed including the history, the technology used, the instrumentation, the system's architecture, the software algorithms, and finally, the methodology.
- Camera preparation and calibration: Reading and exporting data from the FLIR A65 thermal camera generally requires a proprietary application, and this

makes synchronizing with other sensors impossible. In addition, the recordings were in a format (.seq) that cannot be directly opened and viewed by most computers. Therefore, researches and experiments have been conducted here to build a driver that can convert recording frames into PNG or JPG format. Calibrating the camera is another challenging task. Many attempts have been made to calibrate the FLIR A65 camera using existing methods from published literature. However, most of these attempts have failed due to cold temperatures. After some research and based on the findings, a new and creative infrared camera calibration method has been proposed.

- Dataset preparation: Creating a custom dataset is another important step in this project. From some previously recorded data, 8,192 frames with objects have been extracted and labeled. The object classes are people, vehicle, bicycle, and animal. The labeling process is the most boring, and time-consuming part of this project. Because how objects are labeled can directly affect the performance of a trained model and thus, carefully placing a bounding box around an object with the most appropriate amount of spacing requires a lot of patience.

- Training a Support Vector Machine (SVM): When training the SVM model, the architecture and processing pipeline have been thoroughly researched and analyzed. The architecture involves the type and number of SVM required. The processing pipeline involves the selection of image pre-processing and feature extraction techniques. Selecting the optimal pre-processing and feature extraction combo requires a significant amount of coding as each option needed to be implemented and later combined to evaluate the performance. When tuning the SVM parameters, experiments are conducted on a trial-and-error basis which

also requires numerous amounts of coding. 6 image preprocessing and 7 feature extraction methods are implemented and evaluated in this process.

- Training a You Only Look Once (YOLO) network: Each version of the YOLO (from v1 to v4) and the improvements over the previous versions has been studied. Because YOLO does not use any image preprocessing method other than resizing. Therefore, the impact of additional image preprocessing methods on the processing time has been evaluated. Moreover, 3 full-sized YOLO networks (320×320, 416×416, and 608×608) and the corresponding Tiny-YOLO networks have been trained.
- Performance evaluation: The per class precision from the SVM and YOLO has been combined and compared. The discussion on what can be further improved has been carried out.

1.2 The McMaster Center for Mechatronics and Hybrid Technologies (CMHT)

The McMaster Center for Mechatronics and Hybrid Technologies (CMHT) is a world-class research and development center focused on advanced automotive technology. CMHT utilizes a state-of-the-art research facility exploring Autonomous driving, Connected vehicles, Electrification, and Shared mobility (ACES) solutions in the automotive world. CMHT has developed an in-house car detection and tracking technology which has led to an experimental setup for on-road driving, data collection, and real-time detection and tracking. All experiments in this thesis were conducted at the

CMHT with the help of the autonomous team. All sensors including a stereo vision daylight camera, a Velodyne LiDAR, and a FLIR A65 thermal imaging camera were mounted on an electric Ford Focus in Figure 1.1.

Figure 1.1: The experiment vehicle at the CMHT.

1.3 Preliminary Remarks

Driving is an essential part of human life and most people enjoy driving, so why do we need autonomous vehicles? World Health Organization (WHO) stated that every year there are approximately 1.35 million people who die in road traffic crashes, and 20 to 50 million people suffer non-fatal injuries, [1]. Canada reported 1,922 fatalities and 152,847 injuries caused by motor vehicle collisions in 2018, [2]. Although some collisions can be avoided by improving road infrastructure or frequently checking the vehicles' status, it is estimated that around 90% of all motor vehicle collisions are due to human error such as speeding, distracted driving, and the use of psychoactive

substances, [3][4][5]. In addition to putting lives at risk, traffic accidents can also affect the global economy. According to WHO, "Road traffic crashes cost most counties 3% of their gross domestic product". Imagine the societal impact if this problem could be eliminated by autonomous vehicles. The potential benefits and savings are very significant. In economic terms only, Morgan Stanley estimated that autonomous vehicles can contribute to over $5.6 trillion global savings annually, [6].

1.4 The Levels of Autonomy

Society of Automotive Engineers International (SAE) has set six levels of driving automation from level 0 to level 5, [7].

- Level 0: Momentary assistance and warning features such as automatic emergency braking, blind-spot warning, and lane departure warning are supported. The features supported in this category are only limited to providing warnings and momentary assistance. A human driver is in complete control of the vehicle and must constantly supervise the supported features.

- Level 1: In addition to all the features supported in level 0, steering, brake, or acceleration assistance including lane-centering or adaptive cruise control are supported. A human driver must constantly control the vehicle and supervise the supported features.

- Level 2: In addition to level 1, level 2 provides both steering and brake/acceleration assistance. Lane centering and adaptive cruise control must be supported at the same time. A human driver must constantly control the vehicle and supervise the supported features.

- Level 3: In Level 3, the ADAS can drive the vehicle under limited conditions if all required conditions are met. However a human driver must be available to take control when prompted to do so.
- Level 4: Starting from level 4, pedals, steering wheel, and human driver are not needed. The autonomous vehicles in level 4 can operate autonomously under limited conditions if all required conditions are met. Local driverless taxi is one example of level 4 autonomous.
- Level 5: Unlike the vehicles in level 4 that can only operate under limited conditions, level 5 autonomous vehicles can operate in all conditions.

The United States National Highway Traffic Safety Administration (NHTSA) also defined its levels of automated driving systems, but it was soon abandoned to comply with the SAE standard.

1.5 The History of Autonomous Vehicles

Before the SAE standards, automated vehicles were guided by human vision and remote control. Similar to today's radio-controlled cars and roller coasters, they are either actively controlled by humans or only move on a pre-defined path (similar to the steering and brake/acceleration assist in SAE level 1). The first radio-controlled "autonomous" vehicle, called the "American Wonder" (see Figure 1.2) was invented in 1925, [8]. This early attempt of the autonomous vehicle has antennas installed and uses radio signal to control its acceleration and braking. During the road test, the operator remotely controlled the "American Wonder" from the vehicle behind.

In 1939, Norman Bel Geddes took a step further. Instead of remotely controlling the vehicle himself, he embedded the radio-controlled circuits in the roadway, [8]. Later in 1958, RCA Labs and the State of Nebraska successfully demonstrated their autonomous vehicle using a specially designed infrastructure: coils in the pavement and roadside installed lights. The driver was able to know if he got too close to the vehicle ahead by listening to a bell ring and looking at a radio-controlled light installed in the vehicle and on the side of the road, [8]. These prototypes are not truely autonomous by today's SAE standard. Instead, they are trying to develop driver-assist features. These early attempts rely on roadway features, either by embedding radio-controlled circuits or magnetic cables under the pavement.

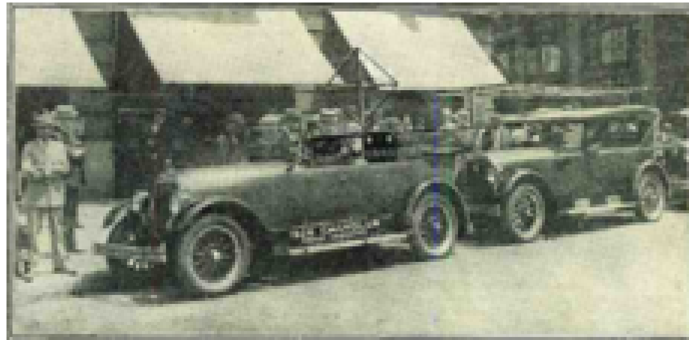

Figure 1.2: The American Wonder in 1925.

The first autonomous vehicle by today's standard was tested in 1986, a vision-guided Mercedes van built by Ernst Dickmanns and his team at Bundeswehr University Munich, Germany, [8]. The van was able to detect road markings and travel at 63 kilometers per hour on a closed highway. Dickmanns' team later put the equipment on two Mercedes 500 SELs and named them the UniBwM VaMP and Daimler VITA-2. These vehicles were demonstrated in France on a three-laned public highway in 1994, [8]. They were able to travel autonomously at 130 kilometers per hour and

switch lanes autonomously. In 1995, a re-engineered version of Dickmanns' S-class traveled autonomously more than 1,700 kilometers from Bavaria to Denmark on the German Autobahn at an astonishing speed of 175 kilometers per hour, [9].

On March 13, 2004, the Defense Advanced Research Projects Agency (DARPA) launched the Grand Challenge to facilitate robotic development, [10]. There were 15 autonomous vehicles involved in this competition to navigate a 228km route across the Nevada desert. Unfortunately, none of the vehicles finished. The second Grand Challenge was held on October 8th, 2005. A Volkswagen Touareg from Stanford University managed to finish first. The third Grand Challenge known as "the urban challenge" was held on November 3, 2007, and required the vehicles to operate autonomously and finish an urban course in under 6 minutes. All vehicles were required to drive in traffic and perform complex maneuvers such as merging, passing, parking, and negotiating intersections. 6 teams managed to finish.

During the Expo 2010 Shanghai China, the VisLab Intercontinental Autonomous Challenge (VIAC) was launched, [11]. Four autonomous vehicles were involved in the 13,000-kilometer challenge from Parma Italy to Shanghai China. These vehicles demonstrated the ability to follow a pre-defined route autonomously with almost no human intervention, and they proved that it is possible to transport goods using such vehicles between two continents. This was considered one of the main milestones in Robotics.

In October 2015, Tesla Motors released its software update 10 that allows the owner to "summon" the vehicle and allows the vehicle to park itself, [12]. This was ahead of all other production vehicles at the time. During the same year, Delphi Automotive's autonomous vehicle traveled 3,500 miles from San Francisco to New

York, and it became the first autonomous vehicle that demonstrated its capability of traveling coast-to-coast, [13].

In 2017, SAE International and General Motors launched the AutoDrive Challenge, [14]. The goal of this competition was to develop and demonstrate a level 4 autonomous vehicle by SAE standards that navigates an urban driving course autonomously. Each team was given an electric Chevrolet Bolt and tasked to convert it into an autonomous vehicle. In the end, University of Toronto's self-driving car team, "aUToronto" won the competition, [15].

1.6 Some Recent Autonomous Accidents

On January 20, 2016, in Hebei China, a Tesla Model S with autopilot engaged was traveling on the highway and crashed into a truck. The 23 years old driver was not alerted by the system and died in the accident, [16]. This is not the first known fatal autonomous car accident.

On March 18, 2018, Uber's autonomous vehicle, a Volvo XC90 hit and killed a 49 years old female cyclist at night in Tempe, Arizona, [17]. Instead of using Volvo's factory automatic emergency braking system, Uber equipped the vehicle with a level 3 autonomous system and disabled the factory system to prevent potential software conflict. While the victim was walking her bike across the street, the vehicle was traveling at 40 miles per hour in its fully autonomous mode. According to the post-crash investigation, the system classified the victim as an unknown object and crashed into her, [17].

Later on March 28, 2018, a Tesla Model X crashed into a roadside barrier in Mountain View, California. Six seconds before the impact, the driver was alerted

by the hands-on warning. However, it was reported that the 38 years old driver was distracted by his cellphone and died soon afterward. Two years later, the National Transportation Safety Board (NTSB) published its investigation regarding the accident. The involved Tesla vehicle was not designed to detect the crash barrier and accelerated into the barrier, [18].

These autonomous vehicles all have rather high-leveled autonomous features, level 2-3 for Tesla, and level 3 for the Volvo. However, they all failed to detect the objects ahead. To prove whether autonomous vehicles are safe to operate on the road, they must be tested for hundreds of millions more miles, [19]. As of February 2018, Google's autonomous vehicles have been road-tested for 5 million miles, [20].

1.7 The Sensors

Autonomous vehicles are usually equipped with five types of sensors including cameras, Radio Detecting and Ranging (RADAR), ultrasonic sensors, Global Positioning Systems (GPS), and Light Detection and Ranging devices (LiDAR). Good sensors are the prerequisite of a reliable autonomous system.

Modern autonomous vehicles use mono or stereo cameras to produce a 360° view of their surroundings because cameras can provide most of the details needed by a human driver from outside the vehicle. The 360° camera system normally consists of 4 to 6 cameras located around the vehicle including both front and rear-facing cameras. The ADAS normally runs algorithms on the front-facing cameras for the majority of object detection, classification, and distance measurement tasks. Depending on the application and cameras' specification, a camera's working distance and field of view (FOV) may vary. For example, a typical front-facing camera can detect objects as far

as 250 meters ahead, a medium-range front-facing camera uses 70° - 120° of horizontal FOV, and a long-range front-facing camera may use 35° of horizontal FOV, [21]. Figure 1.3 shows the front-facing cameras on a Tesla vehicle which have a maximum detection range of 250m, [22].

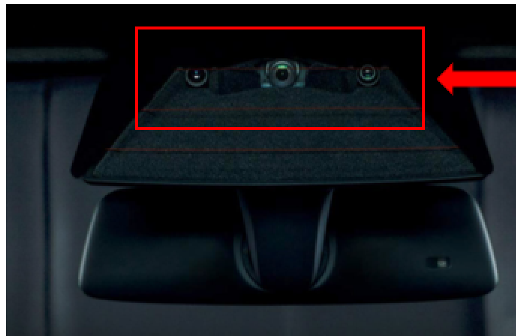

Figure 1.3: Tesla's front-facing camera module

Thermal or infrared (IR) cameras use a sensor that is sensitive to infrared radiation also known as heat. Any object above absolute zero emits infrared radiation, so IR cameras can detect most objects that regular daylight cameras fail to detect at night. The wavelength of infrared radiation is longer than visible light which makes it scatter less when traveling through airborne substances, [23]. Consequently, IR cameras are useful for night-time driving and detecting live objects behind fog, dust, and gas. Figure 1.4 shows the FLIR A65 IR camera that was used in this research.

Figure 1.4: The FLIR A65 thermal camera

Radar is commonly used for detecting and localizing objects using high-speed radio waves, [24]. Radar calculates the distance by measuring the time it takes radio waves to bounce off the object and travels back to the radar receiver. Because radar waves can pass through substances like fog and dust, the radar sensor plays an important role in detecting objects.

The current radar systems deployed on autonomous vehicles work either in the 24 GHz or 77 GHz frequency band, [21]. The 77GHz radars are more accurate but the 24 GHz radars are small and compact. Blind-spot detection, lane change assist, rear-end collision warning, automatic emergency braking, adaptive cruise control, and park assist are application domains of using radar. All Tesla vehicles have a long-range radar that can detect objects up to 160m ahead, [22]. The ADAS usually need multiple radars facing different angles for improved accuracy. Figure 1.5 shows a 77GHz front radar sensor manufactured by BOSCH that has a maximum detection range of 210m.

Figure 1.5: BOSCH's 77GHz front radar sensor

LiDAR is a remote sensing method primarily used for distance measurement, [25]. LiDAR generates a 3D animated point cloud representation of an object by illuminating the target with laser pulses and calculating the distance based on the laser reflection. In general, autonomous vehicles use two types of LiDAR, Mechanical LiDARs and solid-state LiDARs. The mechanical LiDAR uses a rotating mechanism to generate a 360° FOV, whereas the solid-state LiDAR has no moving parts and a narrower FOV.

Depending on the applications, automotive LiDARs can detect objects at a maximum distance of 70m on average, and some premium LiDARs can even detect objects at a maximum distance of 250m. Because solid-state LiDARs are cheaper, automakers usually use multiple solid-state LiDARs placed at different angles to generate a 360° view.

In 2018, Toyota Research Institute (TRI) introduced their latest automated driving research vehicle version 3.0, a Lexus LS600hL was equipped with four long-range solid-state LiDARs by Luminar for a maximum of 200 meters detection range in every

direction, [26]. Figure 1.6 shows the Luminar LiDAR on the TRI's research vehicle.

Figure 1.6: The Luminar LiDAR on the TRI's research vehicle.

The ultrasonic sensor emits high-pitched sound waves and calculates the distance by measuring how long the reflection wave takes to get back to the sensor, [27]. Ultrasonic sensors can generally detect objects up to 9-11 meters away. Therefore, this type of sensor is only suitable for close-range applications. To get accurate detection with ultrasonic sensors, objects must be made of materials that can easily reflect ultrasonic waves. Because sound waves can be absorbed by materials like fabric, the ultrasonic sensors on modern vehicles are only used for low-level and short-range safety features such as blind-spot warning and parking assist. Modern vehicles commonly use 4 to 12 ultrasonic sensors for wide FOV coverage. Figure 1.7 shows the ultrasonic parking sensors that can be found on almost all modern vehicles.

Figure 1.7: The ultrasonic parking sensor on modern vehicles

GPS is a satellite-based system that was originally developed by the U.S. government for military use and consisted of 24 satellites orbiting the Earth, [28]. But nowadays, GPS is commonly used on modern vehicles for positioning, navigation, and timing (PNT). Positioning is the act of accurately calculating the current position including longitude, latitude, and altitude. Navigation is for setting up a course from one place to another, and finally, timing is the ability to acquire the current local time.

The GPS on a vehicle usually works in conjunction with a compass and forms a system called satellite navigation (satnav). The satnav marks the vehicle's current position on a 2D or 3D digital map and indicates the current traveling direction and speed, estimated time of arrival (ETA), and highlights the optimal route to the destination. Some advanced satnav systems can even provide traffic information such as the locations of the traffic signals, signs, and speed cameras. Figure 1.8 shows a typical GPS satnav system on modern vehicles.

Figure 1.8: A typical GPS satnav system on modern vehicles.

1.8 The Latest Trend of Autonomous Vehicle

The latest autonomous vehicles are usually equipped with a variety of sensors and utilize sensor fusion technologies to combine sensors' output. Sensor fusion has been exercised by many automakers on their autonomous vehicles. Uber used a total of 7 LiDAR, 7 Radar, and 20 cameras on its Ford Fusion. It later reduced the sensors to 1 LiDAR, 10 Radar, and 7 cameras on its Volvo XC90, [29]. Tesla used 8 cameras, 12 ultrasonic sensors, and 1 radar on all of its vehicles, [22]. Baidu's latest autonomous vehicle equipped with its Apollo 6.0 platform, used two front-facing cameras, 4 LiDARs, and 2 radar, [30].

In recent years, autonomous vehicles have demonstrated their capabilities in both industrial and recreational areas. Autonomous vehicles are being used for public transport, freight transport, mining, retail, manufacturing, and agriculture. During the COVID-19 coronavirus pandemic, new opportunities and challenges for autonomous vehicles are emerging. Baidu leveraged autonomous vehicles to help to fight the pandemic. 104 autonomous vehicles from Baidu were deployed in 17 cities across China, [31]. These autonomous vehicles were used to help frontline anti-epidemic work such as cleaning, disinfecting, and transportation. Figure 1.9 shows the autonomous vehicle from Baidu that is being used to deliver food and supplies to the Beijing Haidian Hospital. On September 26, 2019, Baidu launched its autonomous taxi service in Changsha, China. an initial fleet of 45 vehicles was deployed at a trial area covering major residential and business districts, [32].

Figure 1.9: Baidu's autonomous vehicle is delivering food and supplies to a hospital

1.9 Thesis Outline

This thesis is structured as follows:

- Chapter 1 is the current chapter and provides a brief introduction and objectives of this thesis along with a background review of autonomous vehicles.
- Chapter 2 presents a literature review on thermal imaging, the differences between ML and DL, object detection and classification from both ML and DL perspectives, the metrics, and some popular ML and CNN architectures.
- Chapter 3 discusses the methods and results of calibrating the FLIR A65 infrared camera and proposes a new method for infrared camera calibration.
- Chapter 4 elucidates a YOLO approach for detecting and classifying objects including how to label a dataset, how to train the YOLO, and how altering the network size will affect the overall performance.

- Chapter 5 explains detecting objects using an SVM approach including evaluating different combinations of image pre-processing and feature extraction methods, how to prepare training samples, and how to effectively train the classifier. A discussion on how the SVM compares to the YOLOv4, and what can be improved will also be given at the end of this chapter.
- Chapter 6 provides a summary of all the work that has been done in this thesis and proposes potential areas of future research.

Chapter 2

Literature Review

In this thesis, an thermal imaging infrared camera was used to capture data that facilitates the training of a Machine Learning (ML) and a Deep Learning (DL) model on detecting and classifying objects. The literature regarding the technology and algorithms used will be reviewed in this chapter.

2.1 Thermal Imaging

Unlike regular daylight cameras and human eyes perceive visible light, thermal imaging relies on heat. Heat, also known as infrared or thermal energy is a part of the Electromagnetic (EM) spectrum. Ranking the wavelength in the EM spectrum from the shortest to the longest, the EM spectrum consists of radio, microwave, infrared, visible light, ultraviolet, X-ray, and gamma-ray, [33]. Due to the difference in radiation wavelength, a regular daylight camera and human eyes cannot detect heat and therefore making heat-radiating objects such as living things and mechanical equipment invisible in the dark. However, the optical systems on thermal imaging cameras

are built to detect IR energy, so they can easily detect the heat being absorbed, reflected, and transmitted, and therefore work perfectly for detecting such objects in the dark. The thermal imaging technology was originally developed and vastly used in military applications, but because it is so useful, nowadays it has been extensively used by many other applications such as firefighting and temperature monitoring, [34].

2.2 The Algorithms

Determining if an object is in the image is called detection, and determining what is in the image is called classification. Classification with localization outlines the boundary of the object. In this section, the algorithms for object detection and classification will be explained from the classic ML and modern DL perspectives. Training a model using either the ML or DL approach requires an annotated dataset to be split into a training, a validation, and a testing set. The training set is where the model learns from, the testing set is where the model evaluates its performance, and the validation set is used to tune the model's hyperparameters. In general, there are many options on how much data goes into each set, some commonly used ratios are 60% - 80% for training, 15% - 20% for testing, and 15% - 20% for validation, [35].

2.2.1 Metrics

When evaluating a model, some terminologies, namely precision, average precision (AP), mean average precision (mAP), recall, intersection over union (IoU), and F - β score are commonly used, [36]. It is necessary to understand the meaning of these

terms in order to tell how well a model is performing.

Among a given dataset, the objects in which the model is designed to classify are called the positive class. Conversely, the undesired objects are categorized as the negative class. The classification results can be categorized into True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). TP and TN are two expected outcomes when the model correctly classifies the positive and the negative class. FP and FN are two unexpected outcomes when the model incorrectly classifies the positive and the negative class. Given TP, TN, FP, and FN, precision and recall can be calculated respectively as:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Precision tells how accurate the model is in classification, and recall measures the level of correct true classification. It is worth mentioning that both precision and recall are per-class measurements which means the same model will yield a different precision and recall when predicting another class. Precision and recall both ranging between 0 and 1 and they are often inversely related. There is usually a trade-off between precision and recall. If the precision is high, the recall is likely to be low.

Intersection over Union (IoU) measures the overlap between the ground truth and the prediction. The ground truth usually means the manually labeled bounding box in the dataset, and the prediction means the bounding box generated by the model. The IoU is calculated as:

$$IoU = \frac{Area\,of\,Overlap}{Area\,of\,Union}$$

The higher the IoU score, the better a model is in localizing the ground truth. A good prediction usually has an IoU score greater than 0.5. The PASCAL VOC metric uses 0.5 as the IoU threshold. Figure 2.1 is a graphical depiction of the IoU where the green box represents the prediction and the red box is the ground truth.

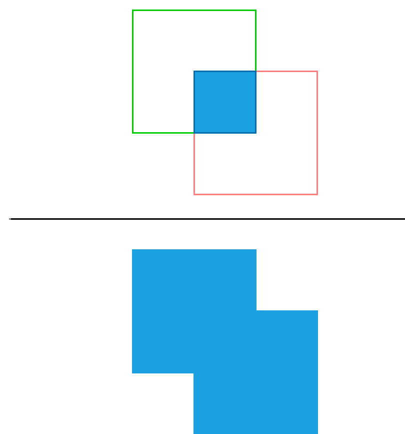

Figure 2.1: Intersection(top) over Union(bottom)

Average precision is the area under the precision curve and it is the value calculated from multiple IoU thresholds for a class. For example, when calculating the precision at both 0.5 and 0.75 IoU, AP is the average of those two. Mean average precision is used to average the AP from multiple classes. The Microsoft Common Objects in Context (MS COCO) dataset uses notations like AP@[.5 : .95], AP50, and AP75 to evaluate precision. The number following AP is the IoU score used to calculate AP. For example, AP50 means calculating the precision at 0.5 IoU, and AP@[.5 : .95] means taking the precision from 0.5 to 0.95 IoU with a step size of 0.05.

The F-β score is a harmonic mean of precision and recall and it is calculated as:

$$F - \beta\ score = (1 + \beta^2) \cdot \frac{precision \cdot recall}{\beta^2 \cdot precision + recall}$$

The β represents the balance between precision and recall. When $\beta = 1$, precision and recall are equally weighted. It becomes precision-oriented when $\beta < 1$ and recall-oriented when $\beta > 1$.

2.2.2 Machine Learning and Its Types

Machine Learning is the study of computer algorithms that improve automatically through experience, [37]. In image processing, ML offers effective methods for applications like target acquisition, feature extraction, and image segmentation. Based on the learning style, ML algorithms can be categorized into supervised, unsupervised, semi-supervised, and reinforcement learning.

Supervised learning can be further divided into classification and regression problems. The major difference between classification and regression problems is that classification problems predict a class label and regression problems predict a real number. Using supervised learning to build an image classifier is an example of the classification problem. The model learns from a labeled image dataset, and every image has a label that specifies the class that it belongs to. When feeding a new image to the model, it predicts the corresponding class label. Weather forecasting is an example of using supervised learning to solve regression problems. Given a set of labeled data, the model predicts the probability of certain weather conditions, i.e. the probability of precipitation on next Monday.

Unsupervised learning, on the other hand, does not require labeled data and is

commonly used for clustering. The k-means clustering algorithm [38] is an example of using unsupervised learning that partitions the initial data points into k clusters by adjusting the centroid (arithmetic mean of each cluster).

Semi-supervised learning is a hybrid of supervised and unsupervised learning and uses a mixture of labeled and unlabeled data during learning. Generally, the dataset in semi-supervised learning largely contains unlabeled samples with only a few labeled samples. For example, a semi-supervised model may use unsupervised learning first to cluster the data, then use supervised learning to label the unlabeled data.

Finally, a reinforcement learning model develops new tactics based on the experiences accumulated in the past. Instead of using a fixed dataset, reinforcement learning models move towards a goal by "reward" or "penalize" themselves.

2.2.3 Classic Machine Learning Methods

The classic machine learning methods in computer vision usually start by acquiring images, then preprocessing the images and extract features, and finally makes a decision through a classifier. This sequence is also known as the vision pipeline.

Image Acquisition

Image acquisition is the act of collecting images using a sensing device and transforming them into an array of numerical data that can be later processed by the computer. Nowadays, a common approach for training a machine learning model is to use challenging datasets that are publicly available on the internet. The sizes of these datasets are usually enormous. For example, the MS COCO dataset contains over 1.5 million images from 91 common object categories, [39]. The ImageNet

dataset contains more than 1.28 million training images, 50,000 validation images, and 100,000 test images that span 1,000 object classes, [40]. Learning from existing datasets like the MS COCO or ImageNet is more than enough to build a classifier that can successfully classify common objects, but some applications require researchers to prepare a custom dataset.

Image Preprocessing

Images might be taken using various optics with different sizes and aspect ratios or resolutions, hence it is often challenging to analyze raw images. Feeding a model with raw images may yield a longer training time and poor performance. Image preprocessing is the fundamental step in all image processing applications that converts raw images into a clean dataset. This step is always needed regardless of whether a machine learning or a deep learning model is being trained.

Some algorithms are designed to take images in a unified size, so raw images need to be resized. For example, the Histogram of Oriented Gradient (HOG) feature descriptor is commonly used as a gradient-based feature extractor that extracts features from images, [41]. As proposed by the original author, the HOG takes images in 64×128 resolution or 1:2 aspect ratio. Therefore, if one wants to use HOG as the feature extractor, raw images need to be resized to the aforementioned resolution or aspect ratio. Otherwise, it may cause unexpected behaviors.

Additionally, images may be noisy. In photography terms, image noise is the randomness of color or luminance fluctuation due to photons or the electronic sensors, [42]. Using such noisy images to train a classifier generates undesirable results. A common approach for noise suppression is to use a technique called filtering also

known as denoising or smoothing. Examples are average filtering [43], Gaussian filtering [44], bilateral filtering [45], median filtering [46], and non-local means denoising, [47]. These filtering techniques will be explained in greater detail in Chapter 5.2.

Moreover, manipulating pixel values such as normalizing (making the values fall under a certain range, usually 0-1), zero-centering (making the distribution of pixel values centered at 0), and standardizing the pixel values (transforming the distribution of pixel values to be a standard Gaussian) are generally beneficial for the training, [48].

Converting color images into grayscale is another commonly used preprocessing technique, [49]. In some applications, color information may not be of interest, and therefore reducing the red, green, and blue (RGB) channels to a single gray-scale channel can significantly reduce the computational complexity as using a single color channel eliminates unnecessary pixel-value-wise operations.

Finally, when there is not enough data available in the dataset, a preprocessing technique called data augmentation can be used, [50]. This can be simply done by adding a filter to the image, slightly rotating or distorting the image, or covering up a small portion of the image. Data augmentation increases the diversity of the data without collecting new data.

Feature Extraction

Images are matrices of numeric values. Given a color (RGB) image, each pixel is represented by three numeric values that correspond to its red, green, and blue components. When training a classifier model, the model usually does not need to learn on the whole image. For example, when building a dog detector, the background is

irrelevant in regards to teaching the model what the dogs are. Therefore, processing three numeric values at each pixel location in the background is unnecessary and a huge waste of computing resources.

A common approach to remove irrelevant information is feature extraction. Features are descriptive or informative patches in an image that distinguish one object from another, [51]. Common vision features are color, texture, and shape. To extract color features, the color space needs to be first defined, e.g., RGB. Then the color features can be extracted using methods such as color histogram [52], color moments [53], and Color Coherence Vector (CCV), [54]. Color histogram and color moments measure the color distribution in an image, and the color coherence vector localizes regions of similar colors in an image.

Textural features are another useful characteristic of an object and they can only be measured by a group of pixels. The methods for extracting textural features can be broadly categorized into spatial extraction and spectral extraction, [51]. Spatial extraction extracts textural features by computing the statistics of the pixel values in a local region, and the spectral method computes textural features in the frequency domain. Gabor filter is commonly used for extracting this type of feature, [55].

The shape features can be extracted globally or regionally using either a contour-based or a region-based method, [51]. The contour-based method works simply by detecting the boundary or edge of the shape. Extracting edge features is traditionally done by calculating the difference in adjacent pixel values. If the change in pixel intensity is drastic enough and greater than some thresholds, then it can be concluded that the pixel belongs to the edge. Applying a specially designed kernel over the image in both horizontal and vertical directions and performing convolution on the

pixel values can also achieve this. Some commonly used kernel detectors are Canny [56], Sobel [57], and Prewitt, [58].

Unlike contour-based methods that work on adjacent pixels, region-based methods extract features from a region of pixels. The contour-based edge extractor will be explained in greater detail with examples later in this thesis.

The Classifier

In image classification, the main objective is to accurately identify the features present in an image. Up to this point, the images are pre-processed, and all features are extracted. It is time for the classifier to make a final decision.

To train a classifier, the model requires sample-and-label pairs. The samples are the training images and are usually denoted as x_i. The labels are annotations for the training images and are usually denoted as y_i. For example, when training a dog detector, the x_is are the dog images, and the y_is are the text string "dog" or the numeric value "1" depending on the actual implementation. Given training data pairs (x_i, y_i), where x_i is the ith image, and y_i is the corresponding label. In this case, y_1 means it is a positive sample that contains the desired object (i.e. a dog image), and y_{-1} means it is a negative sample that does not contain the desired object (i.e. a cat image). A binary classifier $f(x)$ is trained such that

$$f(x_i) \begin{cases} \geq 0, & y_i = +1 \\ < 0, & y_i = -1 \end{cases} \tag{2.2.1}$$

When the classifier correctly classifies the positive and negative class, $y_i f(x_i) > 0$. Otherwise, when objects are misclassified, $y_i f(x_i) < 0$. For example, when the classifier

yields $f(x_i) < 0$ for $y = +1$, it implies the objects is misclassified. A linear classifier has the form:

$$f(x) = \mathbf{w}^T x + b \tag{2.2.2}$$

where w is the weight vector and b is the bias scalar, but only the vector w is needed for classifying new data.

Given a set of labeled data (x_i, y_i), numerous methods can be used to train the classifier. The Support Vector Machine (SVM) is a supervised learning model commonly used for classification problems in many areas of study, [59]. A variant of the SVM, namely the Support Vector Regression (SVR) is commonly used for regression analyses, [60]. The SVM can be broadly categorized into linear and non-linear SVMs based on the shape of the decision boundary, and linear SVMs can be further divided into hard-margin and soft-margin SVMs based on margin constraints. The main objective of linear SVMs is to find the optimal decision boundary or hyperplane in a linearly separable dataset by minimizing the cost (loss) function, in other words, penalizing misclassifications until it produces the largest margin. The cost function is essentially a measurement of error and tells how well the model is performing.

Linear binary SVMs are designed to separate two classes only, a positive and a negative class having labels +1 and -1 (or 0) respectively. Figure 2.2 shows the components of a linear binary SVM in a graphical way including the data points, support vectors, hyperplane, and margin.

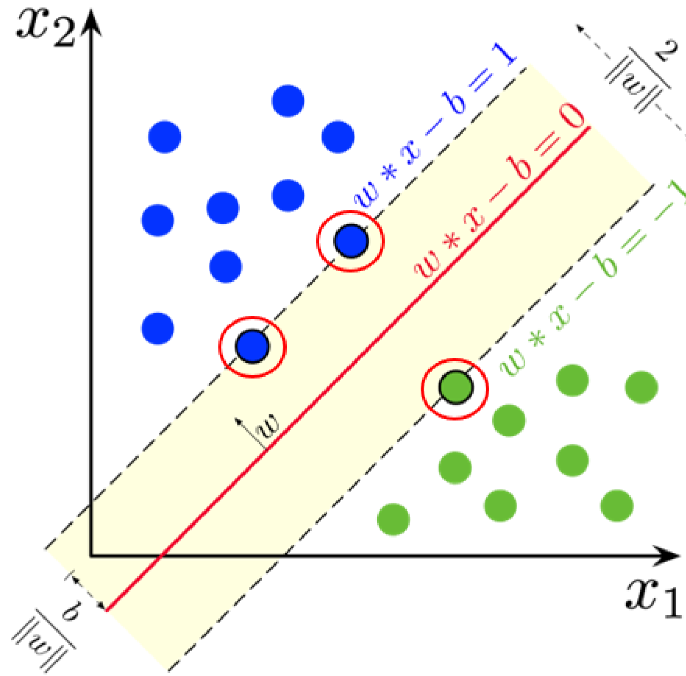

Figure 2.2: The components of a linear binary SVM.

A hyperplane (the red, blue, and green line in Figure 2.2) is the decision boundary that separates the classes, and the red line is called the optimal hyperplane. All data points on the same side of the hyperplane belong to the same class. Depending on the number of classes in the application, a binary or a multi-class SVM can be used. The data points located closest to the hyperplane in red circles are called the support vectors and they are the hardest to classify. Finally, the distance between the blue and green line is called the margin, and it is what needs to be maximized. For a hard-margin SVM, the data points that lie between the margin belong to neither of the classes and therefore there are no misclassifications. The decision function used in the hard-margin binary SVM is:

$$y_i = \mathbf{w}^T x_i + b \tag{2.2.3}$$

where w is the weight, and b is the bias. When $y_i(\mathbf{w}^T x_i + b) \geq 1$, the classification is correct. Otherwise, it is incorrect. Technically, no misclassifications mean there are no loss functions in hard-margin SVMs. Because the margin is calculated as $\frac{2}{\|\mathbf{w}\|}$, one just need to minimize $\|\mathbf{w}\|$ in order to maximize the margin.

For soft-margin SVMs, a non-negative slack variable ξ is introduced to loosen the margin constraints. The problem becomes minimizing:

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}\xi_i \tag{2.2.4}$$

such that

$$y_i(\mathbf{w}^T x_i + b) \geq 1 - \xi_i \tag{2.2.5}$$

where C is a regularization parameter that determines the tradeoff between the maximization of the margin and minimization of the classification error. When $\xi_i \leq 1$, it is correctly classified. Otherwise, misclassified. Soft-margin SVMs use the hinge loss in Figure 2.3 as the cost function, [61].

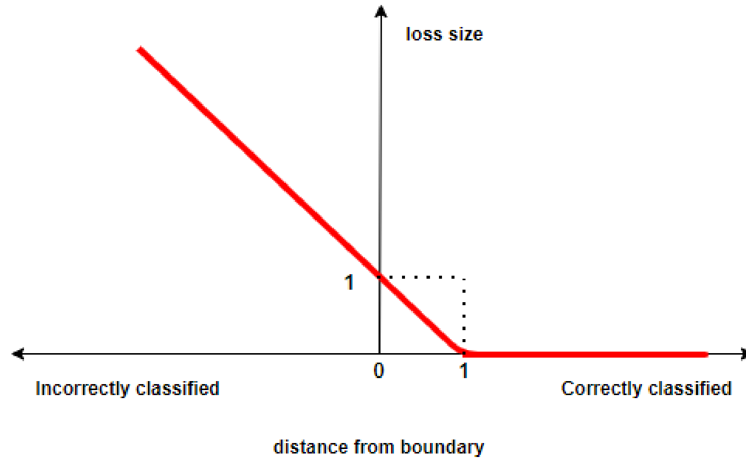

Figure 2.3: The hinge loss.

Note that in Figure 2.3, the horizontal axis represents the distance to boundary and the vertical axis represents the loss. The hinge loss crosses the horizontal axis at 1 which means any data points at a distance greater than or equal to 1 incurs no loss. The hinge loss crosses the vertical axis at 1 which means any data points that sits right on the boundary incurs a loss of 1. Furthermore, the data points that are correctly classified have a smaller loss, and incorrectly classified data points have a bigger loss. Mathematically, the hinge loss can be written as:

$$l = max(0, 1 - y_i(\mathbf{w}^T x_i)) \tag{2.2.6}$$

To obtain the optimized hyperplane in a soft-margin SVM, minimize the following cost equation with gradient descent, [62].

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} max(0, 1 - y_i(\mathbf{w}^T x_i)) \tag{2.2.7}$$

This particular type of SVM that uses "C" ranging from zero to infinity as the regularization parameter is usually called the "C-SVM". Another type of SVM that uses "nu" ranging between $[0, 1]$ is called the "nu-SVM".

When the dataset is not linearly separable, kernel functions must be used. Common kernel functions are polynomial, sigmoid, and Gaussian Radial Based Function (RBF). Kernel functions transform the data to a higher-dimensional kernel space where data becomes linearly separable, [63]. Instead of using the regular x_is, kernel SVMs apply a function f on x_is for the rest of the calculation. Therefore, the kernel SVM formulates the problem as:

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}\xi_i \tag{2.2.8}$$

such that

$$y_i(\mathbf{w}^T f(x_i) + b) \geq 1 - \xi_i \tag{2.2.9}$$

The cost function becomes:

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} max(0, 1 - y^i(\mathbf{w}^T f(x^i))) \tag{2.2.10}$$

Entending binary SVM to multi-class classification, two commonly used heuristic approaches are the One-Against-One (OAO) and One-Against-All (OAA) methods, [64]. In general, the OAO reformulates the multi-class classification problem into $K(K-1)$ binary classification problems, where K is the number of classes. For example, when there exist three classes denoted as "A", "B", and "C", a total of three binary classification problems are created including "A vs B", "A vs C", and

"B vs C". The OAA, on the other hand, also assigns a binary classifier to each class, but only generates K amount of binary classification problems. The corresponding binary classification problems in OAA are "A vs Not A", "B vs Not B", and "C vs Not C".

2.2.4 Modern Deep Learning Methods

Deep learning is a subset of machine learning and it is based largely on Artificial Neural Networks (ANN). Like the human brain, there are many neurons in the ANN that each perform some operation and forward the result layer by layer and finally make a decision.

The development of the ANNs can date back to the 1940s. Warren McCulloch and Walter Pitts started the topic by creating a computational model for neural networks, [65]. The first ANN called the perceptron was invented in 1958 by Frank Rosenblatt, [66]. The perceptron was designed for image recognition, and it used a single-layer architecture meaning that there was only one layer between the input and the output layer. However, scientists named Marvin Minsky and Seymour Papert published a book and proved that the ANN in general, is not possible because the single-layer perceptron cannot learn the XOR function, [67]. ANNs were then abandoned in the late 1960s and remained controversial until the mid-1980s.

Modern ANN architectures can be divided into feedforward neural network and Recurrent Neural Network (RNN), and every modern ANN has a hierarchical structure that consists of an input layer, an output layer, and some hidden layers that reside in the middle. In a feedforward neural network, each neuron in one layer will only connect to the neurons in the next layer. The information only travels in the

forward direction and follows the input-hidden-output flow. Such neural networks only consider the current input and do not memorize the past input. Therefore, feed-forward neural networks are not good at predicting. Whereas in RNNs, the input of a neuron can be fed back to itself and makes the neuron predicts the output based on both its current input and the output from the previous layer. Therefore, by using this feedback mechanism, RNN can make predictions based on its short-term memory, [68].

ANNs have two learnable parameters, the weight and bias denoted as w and b. The weights decide how much influence the input will have on the output, and the biases are constants being added to the next layer as an additional input. The process of getting the output from the input and calculating the error is called the foward pass or forward propagation. Going backwards and use the error to update the weights and biases is called the back propagation. Given a set of input x_i where $i = 1 \ldots N$. In the forward propagation, weights are randomly initialized to small numbers such as 0.1 or it can be initialized by using the Xavier [69] or Kaiming method [70], and the biases can be simply initialized to zero. The Xavier initialization randomly chooses the weight from a uniform distribution between

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} \tag{2.2.11}$$

where n_i is the number of incoming connections, and n_{i+1} is the number of outgoing connections. The Kaiming method is a modification of the Xavier specifically optimized for the ReLu activation function that chooses the weight from a zero-mean Gaussian distribution whose standard deviation is $\sqrt{2/n_l}$. The n_l in the Kaiming method can be calculated as k^2c, where k is the layer's filter size and c is the number

of input channels.

Each layer in the ANN does the following operation:

$$y = f(\sum(\boldsymbol{w_i}^T x_i) + b) \tag{2.2.12}$$

where y is the output being fed to the next layer, $\sum(\boldsymbol{w_i}^T x_i)$ is the summation of every weight and input pair in the ith layer, and f is the activation function. Figure 2.4 is the graphical representation of 2.2.12 with 3 inputs.

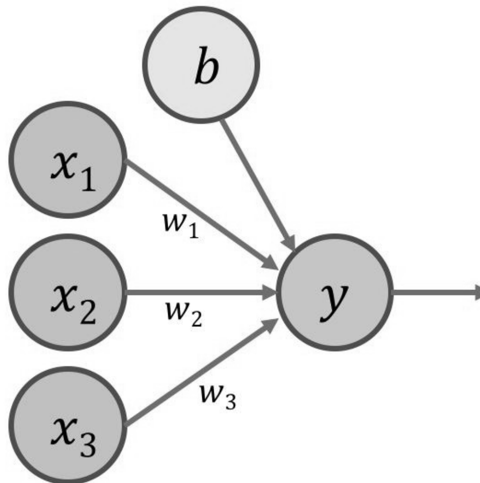

Figure 2.4: The operation on each neuron.

The error also known as the cost is measured by comparing the network-predicted output with the desired output using the cost function:

$$C = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.2.13}$$

where C is the cost, n is the number of neurons in that layer, y_i is the ith desired output, and $\hat{y}_i$ is the ith network-predicted output calculated using 2.2.12. This

particular cost function is called the Mean Squared Error (MSE), and it is one of the most used cost functions for regression problems, [71]. Many variants of this cost function can also be used depend on the application. For example, the Sum Squared Error (SSE), Mean Error (ME), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE), [72].

The goal of the back propagation is to optimize the weights and biases. During the back propagation, the data is usually divided into 16 or 32 mini-batches, and the partial derivatives (gradients) of the cost with respect to the weights and biases in each mini-batch are calculated. The partial derivative with respect to the weights and biases are calculated using the chain rule:

$$\frac{\partial C}{\partial w^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial w^L} \tag{2.2.14}$$

$$\frac{\partial C}{\partial b^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial b^L} \tag{2.2.15}$$

where L is the last layer in the neural network, $z^L = w^L * x + b$, and $a^L = f(z^L)$ with f being the activation function. The weight and bias can then be updated using:

$$w^l = w^l - \alpha * \frac{\partial C}{\partial w^l} \tag{2.2.16}$$

$$b^l = b^l - \alpha * \frac{\partial C}{\partial b^l} \tag{2.2.17}$$

where l is the current layer, and α is the learning rate that controls how much the weight and bias are updated each time.

Back propagation tries to optimize the weights and biases and ultimately minimize the cost. To minimize the cost, stochastic gradient descent (SGD) is used, [73]. SGD

takes a mini-batch of the data and update the weight and bias based on the average gradient from the mini-batch. Moving in the opposite direction of the gradient will reach the local minima. The goal of SGD is to find the global minima, but it is not guaranteed. Sometimes, it will get stuck at a local minima. When a global or local minima is reached, the cost function is minimized, and the neural network is converged. When SGD fails to locate the minima, the neural network is diverged.

2.2.5 Activation Functions

Activation functions decide whether a neuron should be "fired" or not, and it can be categorized into linear and non-linear functions. When the neurons are activated by only linear functions, the neural network will only perform linear transformation using the input, weight, and bias. Therefore, the neural network will not be able to learn the complex patterns from the data. To address this issue, non-linearity needs to be introduced. Some commonly used activation functions are binary step function, linear function, sigmoid function, Tanh function, Rectified Linear Unit (ReLU), Leaky ReLU, Parameterised ReLU (PReLu), and Exponential Linear Unit (ELu), [74].

The binary step function is a threshold-based activation function. The neuron will only be activated when the input is greater than the threshold. This activation function is useful when training a binary classifier. However, it fails when there are multiple classes need to be classified. Another problem with the binary step function is that its gradient is zero. Therefore, the weights and biases will not be updated during back propagation. Figure 2.5 shows the binary step function and its gradient.

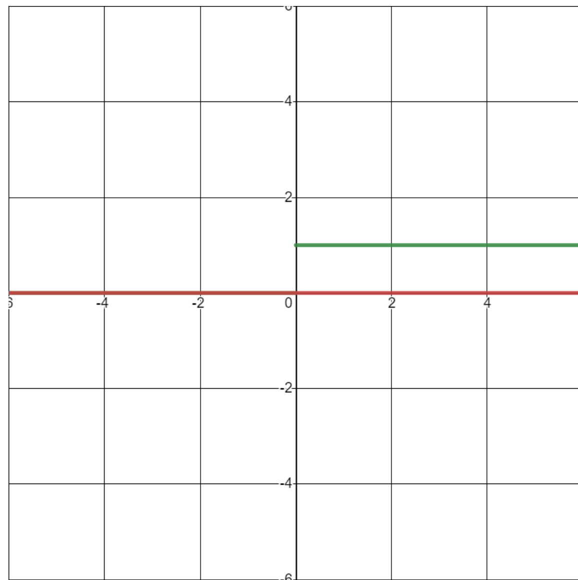

Figure 2.5: The binary step (green) function and its gradient (red).

To address the gradient problem, a linear activation function can be used instead. The activation in a linear function is proportional to the input, and the gradient is a constant. Although a constant gradient will update the weights and biases, it is completely irrelavant to the input. This implies that the weights and biases will always be updated by the same amount in each iteration, and therefore, the neural network will not perfrom well. Figure 2.6 shows the linear function and its gradient.

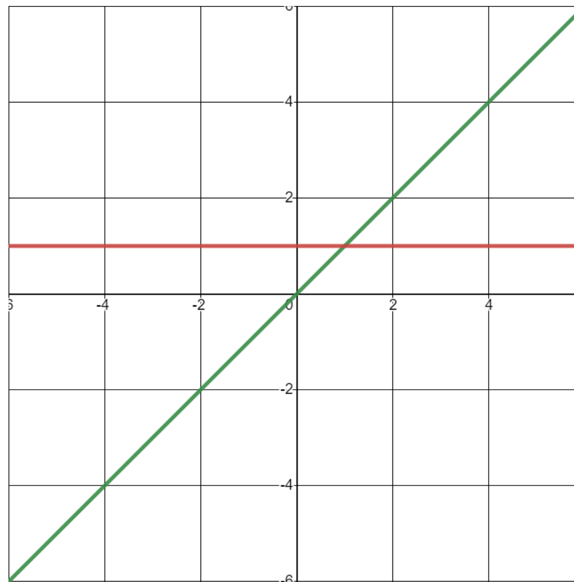

Figure 2.6: The linear function (green) and its gradient (red).

The sigmoid function is one of the most used non-linear activation function, and it is continuously differentiable. However, the gradient becomes close to zero when the input of the sigmoid function gets larger or smaller. When the gradient is close to zero, the neural network struggles to learn as the weights and biases barely update. This problem is known as the vanishing gradient. Figure 2.7(right) shows the vanishing gradient problem at both left and right ends of the graph. Additionaly, the sigmoid function is not symmetric around the origin which causes the output all have the same sign.

Figure 2.7: The sigmoid function (green) and its gradient (red).

The Tanh function looks similar to the sigmoid, but it is scaled, shifted, and symmetric around the origin. The Tanh function is usually preferred over the sigmoid function because the Tanh is zero-centered. The output in the Tanh function ranging from -1 to 1. Therefore, the Tanh function solves the problem where sigmoid's output are all of the same sign. However, the gradient of the Tanh function still approaches zero at both ends, so it will also encounter the vanishing gradient problem. Figure 2.8 shows the Tanh function and its gradient.

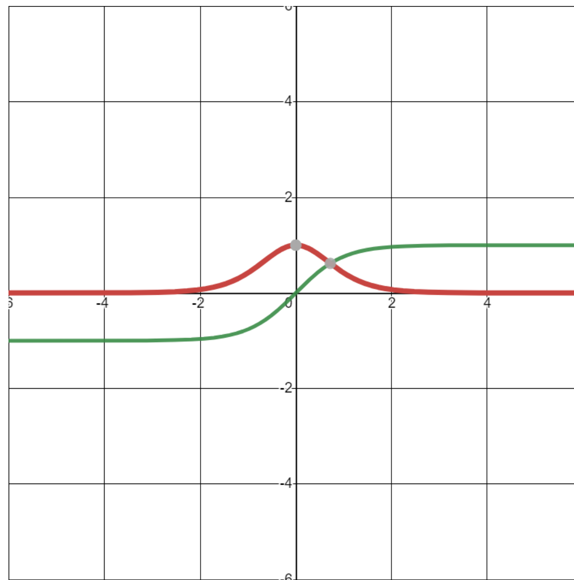

Figure 2.8: The Tanh function (green) and its gradient (red).

The ReLu function also introduce non-linearity into the nerual network, and it is one of the most popular activation functions used in the deep learning realm. The output of the ReLu function is linear for all non-negative inputs and zero for all negative inputs. Because the gradient can be either zero or a constant, such neural network is sparsely activated. A sparsely activated neural network largely mimics the biological neural netwotk in which neurons are not activated all at once. Each neuron has its "role" and is activated by different signal. The sparsity also makes the neural network faster. However, some neurons may never be activated due to the zero gradient, and therefore, they are considered "dead". This is known as the "dying ReLu" problem. This problem is likely to happen when the learning rate is too high or there is a large negative bias.

The Leaky ReLu and PReLu can be used to solve the "dying" problem. The Leaky ReLu and PReLu use a linear function with a coefficient of 0.01 and a respectively

for all the negative inputs. Therefore, the gradient is no longer zero. ELu is another ReLu's variant that uses an exponential function for all the negative inputs. Figure 2.9 shows the ReLu, Leaky ReLu, PReLu, and ELu function.

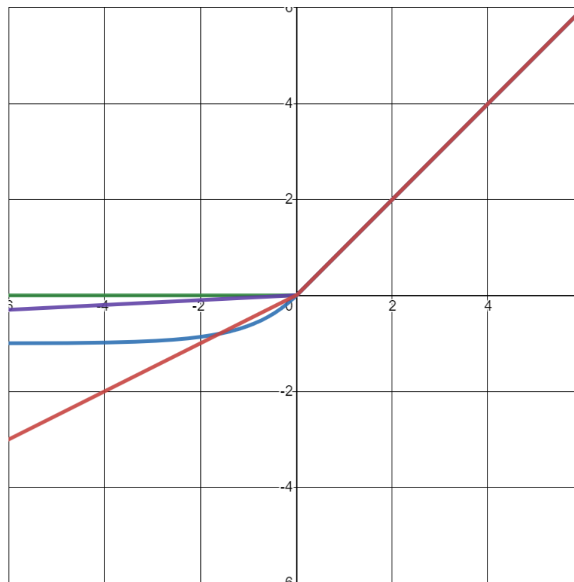

Figure 2.9: The ReLu (green), Leaky ReLu (purple), PReLu (red), and ELu (blue) function.

2.2.6 Convolutional Neural Networks

The major difference between a traditional ANN and a Convolutional Neural Network (CNN) is that the CNN largely mimics the biological neural network. ANN and CNN are structured differently. A traditional ANN usually uses entirely fully connected layers, but usually, only the last layer in a CNN is fully connected. In a CNN, instead of connecting each neuron in one layer to every neuron in the next layer, convolutional layers and pooling layers are used to learn features from the input image. The convolutional layers are the heart of the CNN and use multiple kernels or filters that perform convolution operations at a spatial location on the image.

A kernel or filter is a small $n \times m \times d$ matrix that moves through the full depth of the input volume. $n \times m$ defines the receptive field, more specifically, the width and height of the kernel in pixels, and d is the depth of the input image. Depending on the application, the kernels may be different for each depth level. Squared kernels such as 3×3 and 5×5 are commonly used. For example, when a 32×32 RGB CIFAR-10 image is used, the input volume is therefore $32 \times 32 \times 3$ (32 width, 32 height, and 3 channels).

During the forward pass, the kernel is convolved across the width and height of the input volume. The dot product between each element of the kernel and the corresponding value under the kernel is computed and added. After all entries at the current kernel position have been convolved, the kernel will move to the next location by its stride value. Figure 2.10 is a graphical representation of a $3 \times 3 \times 3$ (3×3 RGB) kernel's movement. Sliding a kernel across the input image will produce a 2-dimensional feature map that summarizes the response of the kernel at each spatial location. A convolutional layer can have multiple kernels that look for different visual features. Stacking these feature maps along the depth dimension gives a 3-dimensional feature volume, and this feature volume will be passed deeper into the network, layer by layer.

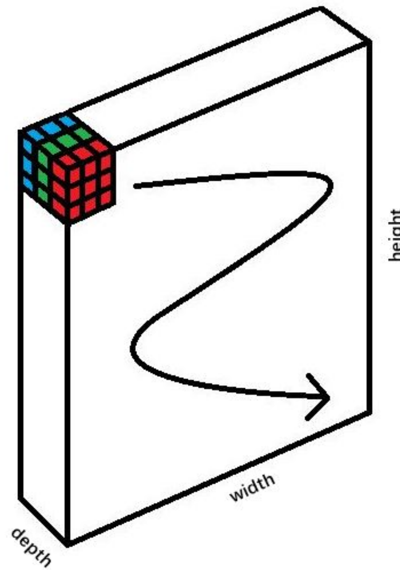

Figure 2.10: The movement of a $3 \times 3 \times 3$ (3×3 RGB) kernel.

In general, the size of the output feature volume can be calculated as:

$$(W - F + 2P)/S + 1 \tag{2.2.18}$$

where W is the input size (e.g., $W = 5$ when a 5×5 image is used), F is the kernel size, P is the number of zero-padding, and S is the stride. When applying the kernels, the information on the edges may be lost. Zero-padding is used to solve this problem and control the dimension of the output volume, expecially when the input dimension needs to be preserved.

Another potential problem with the feature map produced by the convolutional layers is that it is position-reliant. A small movement of the feature will produce a new set of feature maps. To solve this problem, a technique called downsampling is commonly used. Downsampling creates a lower resolution of the feature map while maintaining the overall features. In CNN, pooling layers are introduced to

do downsampling, and are usually placed between convolutional layers. The pooling operation applies a window on the feature map, and depending on the pooling method, a value within the pooling window will be extracted and placed in the new feature map. The most used pooling windows are of size 2×2 with a stride value of 2. By using this setup, the size of the feature map will be cut in half along the width and height. Two commonly used pooling methods are max pooling and average pooling. Max pooling and average pooling calculate the maximum value and the average value within the pooling window, respectively. Another benefit of using pooling layers is to reduce computational complexity. Pooling layers do not introduce additional parameters, instead, by reducing the spatial size of the feature map, the number of parameters is reduced. For example, when using max pooling with a 2×2 window and a stride of 2, 75% of the activation is discarded since max pooling takes only 1 value from the 4-valued window.

The feature volume produced by the last convolutional and pooling layer is then flattened into a long vector, and fed into the fully connected layer at the end of the CNN. Usually, the last layer of the fully connected layer is a softmax layer which is commonly used for multiclass classification problems. It transform the input into values between 0 and 1 that represent the probability of the input belonging to each individual class. The mathematical expression of the softmax function is:

$$P(y = j|x) = \frac{e^{\boldsymbol{w_j}^T x + b_j}}{\sum_{k=1}^{K} e^{\boldsymbol{w_k}^T x + b_k}} \quad \text{for} \quad j = 1 \dots K \tag{2.2.19}$$

Some Famous CNN Architectures

There are a number of famous CNN architectures including the LeNet, AlexNet, Zeiler & Fergus Net (ZF Net), GoogLeNet, VGGNet, and Residual Network (ResNet). LeNet was developed by Yann LeCun in the 1990s, and it was originally used for document (digit) recognition, [75]. LeNet has 3 convolutional layers that use the Tanh activation function and 1 fully connected layer. Between the convolutional layers, sub-sampling or pooling layers are used. LeNet's pooling mechanism first adds all the values in the pooling window, then multiplies a trainable coefficient, and finally adds a trainable bias. The output is then calculated using a sigmoid function. The pooling process uses a 2-strided 2×2 window, and it reduces the size of the feature map by half. Figure 2.11 shows the architecture of LeNet. LeNet was considered the first successful applciation of CNN in reading characters.

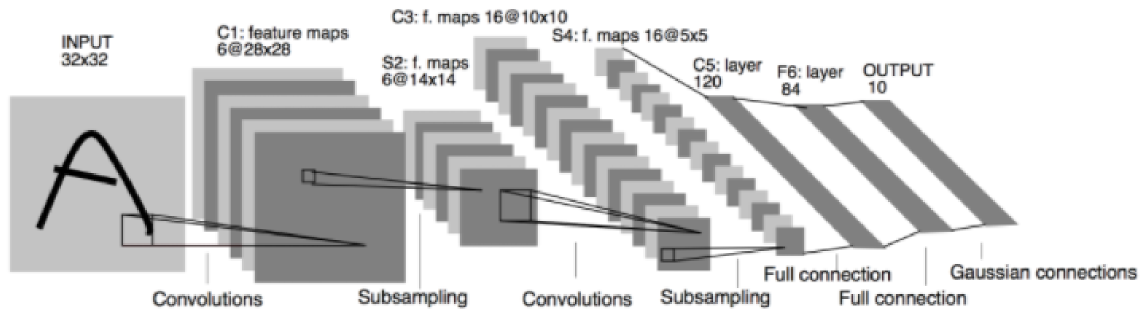

Figure 2.11: The architecture of the LeNet-5, [75].

AlexNet was developed by Alex Krizhevsky et al., and it was the winner of the 2012 ImageNet Large Scale Visual Recognition Competition (ILSVRC), [76]. AlexNet has a similar structure to LeNet, but deeper. 5 consecutive convolutional layers that use ReLu activation function stack on top of each other following three fully connected

layers. 3 Max pooling layers are used between the convolutional layers. Figure 2.12 shows the architecture of AlexNet. It is worth mentioning that AlexNet was the first architecture that popularized the use of CNN in computer vision.

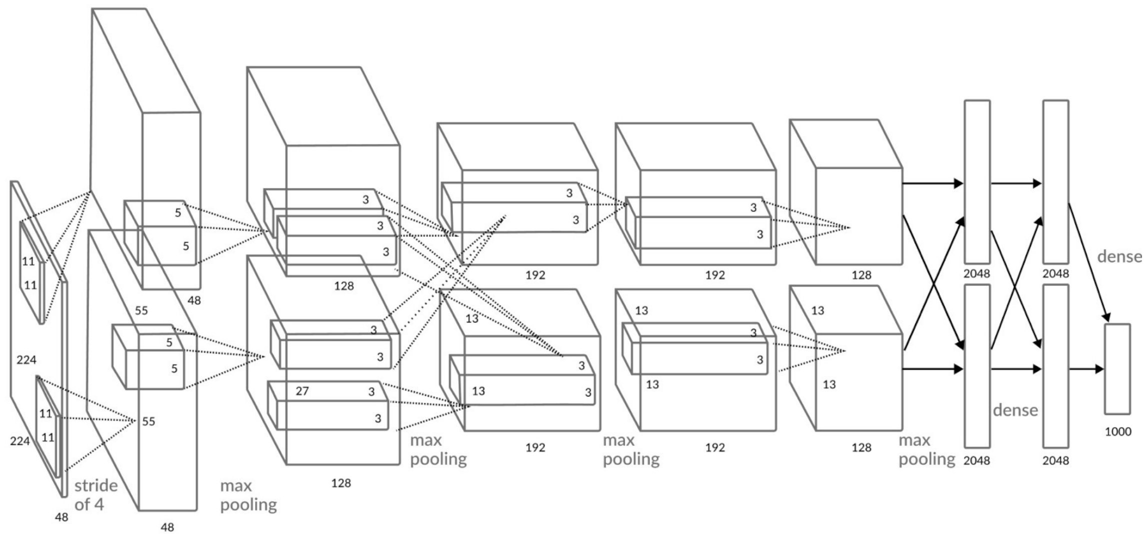

Figure 2.12: The architecture of the AlexNet with two GPUs, [76].

ZF Net was the 2013 ILSVRC winner, and it was an improvement over AlexNet by tuning the hyperparameters, [77]. The kernel size in the first layer was reduced to 7×7 (was 11×11 in AlexNet), and the stride size of the second layer is reduced to 2 (was 4 in AlexNet). These changes enable ZF Net to retain more information in the first two layers compare to AlexNet.

GoogLeNet was the 2014 ILSVRC winner and it was developed by Szegedy et al. from Google, [78]. GoogLeNet uses inception modules to reduce computational complexity. The inception module works by performing convolution operation using the ReLu activation function on the input with three different sized kernels (1×1, 3×3, 5×5). Additionally, 3×3 max pooling is performed within the inception

module. The inception module then concatenates the result from all components and feeds it to the next layer. GoogLeNet has a total of 22 layers excluding pooling layers. Figure 2.13 shows the inception module used in GoogLeNet. Szegedy also introduced a concept called batch normalization where the data are equally divided into many subsets called mini-batches. The mean and standard deviation of each mini-batch is used to normalize a layer's input and therefore speed up the training.

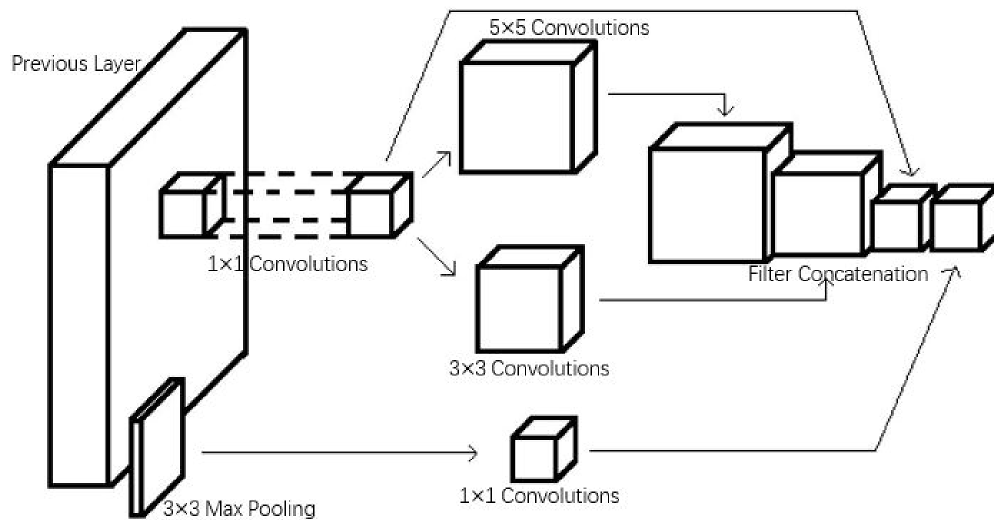

Figure 2.13: GoogLeNet's inception module, [78].

VGGNet was developed by Karen Simonyan and Andrew Zisserman and placed second in the 2014 ILSVRC, [79]. VGGNet has 13 convolutional layers that use the ReLu activation function following 3 fully connected layers. VGGNet showed that good performance can be achieved by increasing the depth of the architecture. The second version of VGGNet (VGG-16) has 16 convoluitonal layers and 3 fully connected layers. Figure 2.14 shows the architecture of VGG-16.

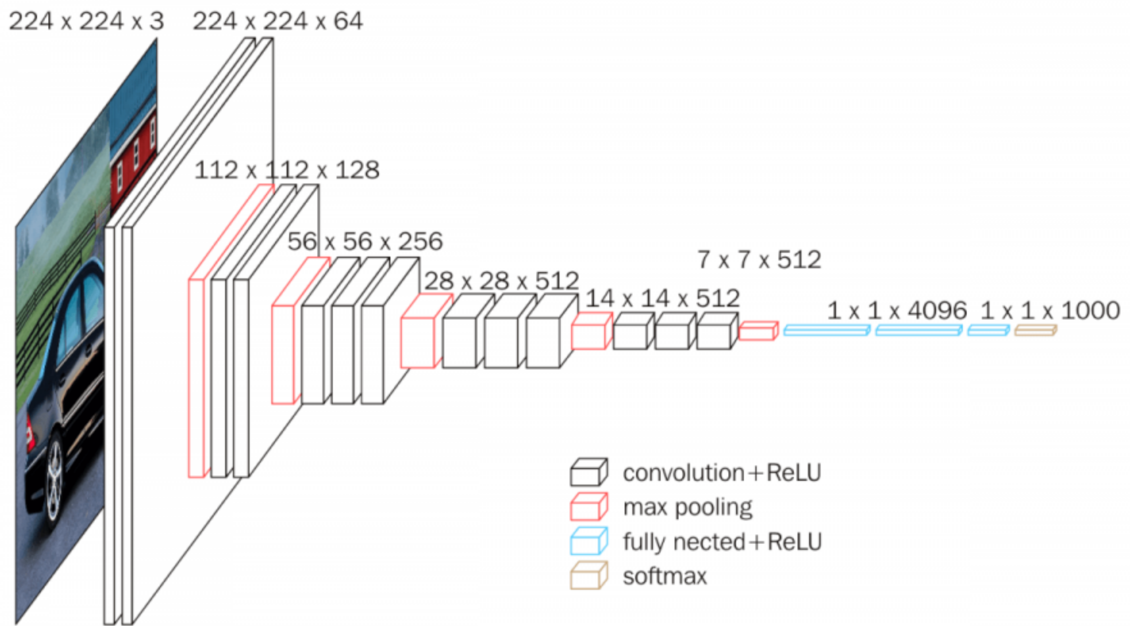

Figure 2.14: The architecture of VGG-16, [79].

ResNet was developed by He et al., and it was the 2015 ILSVRC winner, [80]. ResNet heavily uses batch normalization and features skip connections where a layer's output is fed 2-3 layers ahead. Figure 5.3 shows the architecture of ResNet and its skip connection mechanism.

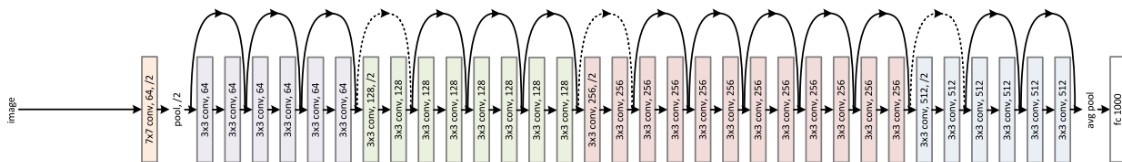

Figure 2.15: The architecture of the ResNet, [80].

You Only Look Once

You Only Look Once (YOLO) was developed by Joseph Redmon et al. and it was inspired by GoogLeNet, [81]. YOLO replaces GoogLeNet's inception modules by

1×1 reduction layers following 3×3 convolutional layers. The backbone of the first YOLO or YOLOv1 was made of 24 convolutional layers that use the leaky ReLu activation function following 2 fully connected layers as shown in Figure 2.16. The final layer predicts both class probabilities and bounding box coordinates using a linear activation function.

Before YOLO, Region Proposal Network (RPN) such as Region-Based Convolutional Neural Network (R-CNN) [82] and its variants, fast R-CNN [83] and faster R-CNN [84] are commonly used for image classification problems. RPNs use a two stage process where region proposals or the potential identifiable regions are extracted from the image, and then a CNN is used to classify these region proposals. Whereas in YOLOv1, an image is divided in to a $S \times S$ grid and a CNN is used to classify and localize objects from each of the S^2 grid cells simultaneously. Each grid cell is responsible to predict B bounding boxes. Each bounding box can be defined as (x, y, w, h, c), where x and y are the coordinates that represent the center of the bounding box relative to the boundary of the grid cell, w and h are the width and height of the bounding box relative to the entire image, and c is the confidence score defined as $c = P_r(Object) * IoU$. When a bounding box contains no object, c is zero, otherwise, it is equal to the IoU between the bounding box and the ground truth. The x, y, w, h are all normalized to values between 0 and 1. Additionally, if an object presents in a grid cell, that grid cell also predicts C conditional class probabilities denoted as $P_r(Class_i|Object)$. The class-specific confidence score for each bounding box denoted as $P_r(Class_i) * IoU$ can then be calculated using:

$$P_r(Class_i) * IoU = P_r(Class_i|Object) * P_r(Object) * IoU \tag{2.2.20}$$

There are multiple bounding boxes within each grid cell, but only the one with the highest IoU is used to calculate the loss. YOLOv1 uses the sum squared error to calculated the loss, and the loss function has three components, the localization loss, the classification loss, and the confidence loss, [81]. The localization loss measures the error in the bounding boxes' orientation (x, y, w, h), and it is calculated as:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{w}_i - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h}_i - \sqrt{\hat{h}_i} \right)^2 \right]$$

Because YOLOv1 does not weigh localization error equally with classification error, so $\lambda_{coord} = 5$ is introduced to increase the significance of localization error. The $\mathbb{1}_{ij}^{obj}$ equals 1 when the jth bounding box in grid cell i is responsible for the prediction, otherwise, it equals 0. The confidence loss is calculated as:

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left(C_i - \hat{C}_i \right)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} \left(C_i - \hat{C}_i \right)^2 \tag{2.2.21}$$

when a grid cell contains no object, the confidence is 0. YOLOv1 wants to decrease the significance of these grid cells on the loss and therefore the $\lambda_{noobj} = 0.5$ is used. The $\mathbb{1}_{ij}^{noobj}$ equals 1 when there is no object within a grid cell, otherwise it equals to 0. The classification loss can be calculated as:

$$\sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \tag{2.2.22}$$

The $\mathbb{1}_{i}^{obj}$ equals 1 when there exists an object in the ith grid cell, otherwise 0.

Although YOLOv1's one-stage process makes it one of the fastest architectures for real-time object detection and classification, there are limitations. Each grid cell only produces 1 classification regardless the number of bounding boxes generated. Therefore, YOLOv1 struggles to detect and classify objects that appear in groups. As YOLOv1 generates bounding boxes based on the training data, when applying YOLOv1 on new data with different aspect ratios or configurations, the result is not idea. Because the input image is downsampled multiple times using the downsampling or pooling layers, relative coarse features are extracted for training. Finally, a small error has the same significance in both big and small bounding boxes. The loss function should consider a small error in a small box more important than a small error in a big box.

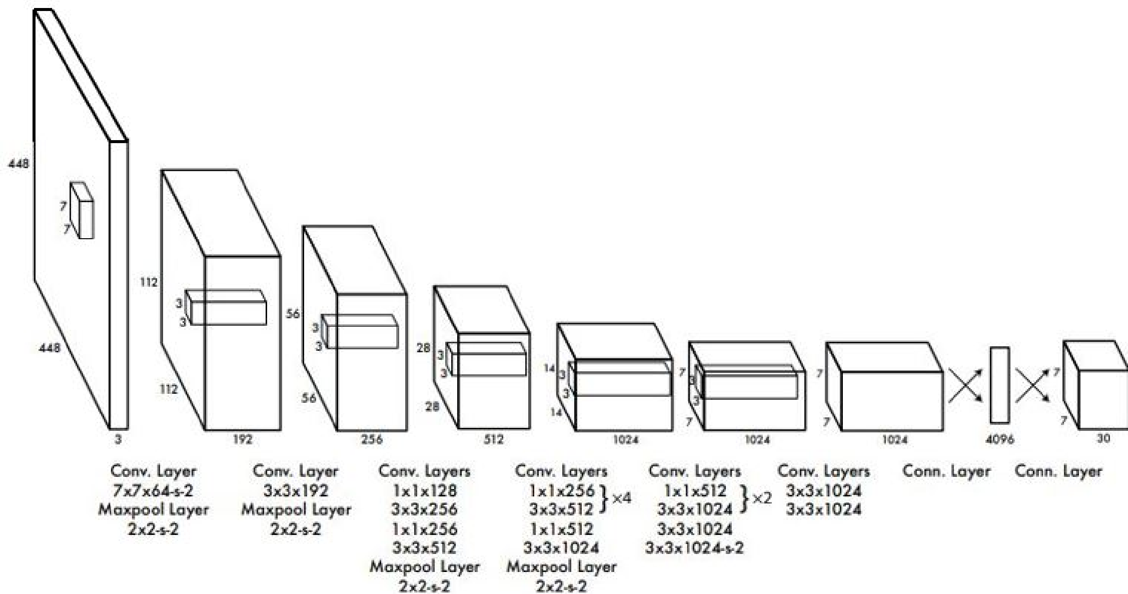

Figure 2.16: The architrcture of YOLOv1, [81].

To remedy YOLOv1's limitations and further improve the speed and accuracy, YOLOv2 is introduced, [85]. The backbone of YOLOv2 is called the darknet-19 that

consists of 19 convolutional layers. The improvement over YOLOv1 can be summarized into batch normalization, high resolution classifier, anchor boxes, dimension clusters, direct location prediction, fine-grained features, and multi-scale training.

In YOLOv2, by adding batch normalization to every convolutional layer, the mAP is improved by more than 2%. Batch normalization also eliminates the need of other forms of regularization. Training the classification network on high resolution input (448×448) increased the mAP by 4%. YOLOv2 adopt an anchor box-styled approach to capture specific object classes. An anchor box is a bounding box but has a predefined size based on the training data. Because objects in the same class have similar aspect ratios and scales, instead of arbitrarily predicting the size of each bounding box, YOLOv2 uses anchor boxes as the guideline. Given all the ground truth, a modified k-means clustering method is used to find the optimal number and size of the anchor box using the distance metric $d(box, centroid) = 1 - IoU(box, centroid)$. This k-means method groups similar-sized ground truth into a cluster and finds the number of anchors that lead to the highest IoU. The anchor box-styled approach in YOLOv2 enables the network to predict the coordinate of 5 bounding boxes at each cell defined as $(t_x, t_y, t_w, t_h, t_o)$ which are the same as in YOLOv1. The location of the bounding box is then adjusted by the offset to the anchor using:

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e_w^t \\
b_h &= p_h e_h^t
\end{aligned}
\tag{2.2.23}
$$

where $\sigma(t_x)$ and $\sigma(t_y)$ are the distance relative to the grid cell, (c_x, c_y) is the offset to

the top left corner of the image, and p_w and p_h are the width and height of the anchor box. This can be seen in Figure 2.17 where the dotted box represents the anchor box. Using k-means clustering together with direct location prediction, the mAP increases by 5%.

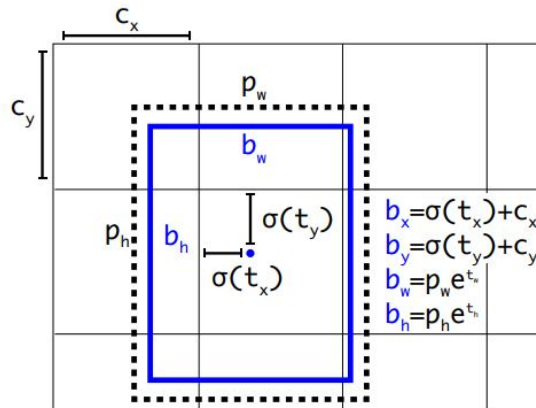

Figure 2.17: The prediction of a bounding box in YOLOv2, [85].

Because YOLOv2 removes the fully connected layers at the end, so it is not limited to produce only 1 classification per grid cell. A passthrough layer is used to concatenates earlier high resolution features with the downsampled lower resolution feature. This allows the network to learn fine-grained features and increases the mAP by 1%. Finally, To enable multi-scale training, YOLOv2 randomly resizes the input for every 10 batches. This forces the network to learn on various input scales.

In YOLOv3, the darknet-19 is replaced by the 53-layered convolutional neural network namely, the darknet-53, [86]. The darknet-53 consists of mainly 3×3 and 1×1 filters and features skip connection like ResNet. Redmon showed that darknet-53 has an overall better mAP than ResNet, and it is $1.5 - 2$ times faster than ResNet.

YOLOv3 changes the way how the loss function is calculated; If an anchor box has the highest IoU with the ground truth, the objectness score is 1. Meanwhile, ignoring

the anchors that have greater than 0.5 IoU but do not best overlap the ground truth. Additionally, multi-label classification is allowed in YOLOv3, where an object can have overlapping labels, for example, woman and person. This is achieved by using independent logistic classifiers instead of softmax. Finally, YOLOv3 uses the concept of Feature Pyramid Network (FPN) as shown in Figure 2.18 where 3 different-scaled bounding boxes at each location is predicted, [87]. These feature maps are then merged together and thus the network can learn more meaningful information.

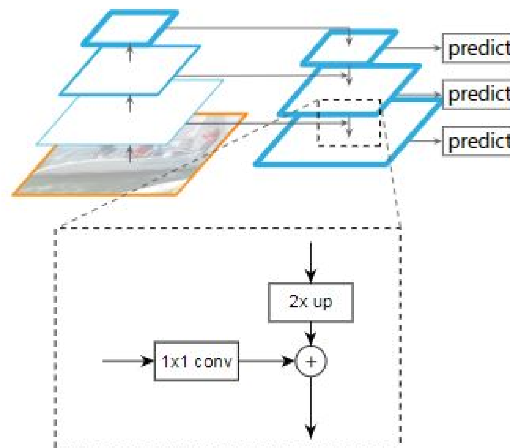

Figure 2.18: In FPN, the top-down feature maps (indicated by a downward-pointing arrow) are scaled up by a factor of 2 and merged with the bottom-up feature maps (indicated by a upward-pointing arrow) by addition, [87].

The CSPDarknet-53 (see Figure 2.19) is the backbone of YOLOv4 and it is the Darknet-53 combined with the CSPNet, [90].

	Type	Filters	Size	Output
	Convolutional	32	3x3	256x256
	Convolutional	64	3x3/2	128x128
	CrossStagePartial			
1x	Convolutional	32	1x1	
	Convolutional	64	3x3	
	Residual			128x128
	Convolutional	128	3x3/2	64x64
	CrossStagePartial			
2x	Convolutional	64	1x1	
	Convolutional	64	3x3	
	Residual			64x64
	Convolutional	256	3x3/2	32x32
	CrossStagePartial			
8x	Convolutional	128	1x1	
	Convolutional	128	3x3	
	Residual			32x32
	Convolutional	512	3x3/2	16x16
	CrossStagePartial			
8x	Convolutional	256	1x1	
	Convolutional	256	3x3	
	Residual			16x16
	Convolutional	1024	3x3/2	8x8
	CrossStagePartial			
4x	Convolutional	512	1x1	
	Convolutional	512	3x3	
	Residual			8x8

Figure 2.19: The CSPDarknet-53, [88].

CSP stands for cross stage partial connections where the input is divided into two parts, one part is skip connected to the transition layer and the other part is fed into the convolutional layers, [88]. This new design significantly reduces computational complexity. In addition, the Spatial Pyramid Pooling (SPP) blocks are added to the CSPDarknet-53. SPP spatially divides the feature map into different scales and then applies max pooling, [89]. The SPP block significantly increases the receptive field and causes almost no speed reduction. It is believed that a bigger receptive field and

more parameters are beneficial for the training, [90].

In addition to the YOLOv3's FPN approach, YOLOv4 also uses a Path Aggregation Network (PAN) to concatenate feature maps in a bottom-up fashion at different detector levels. This can be seen in Figure 2.20. In FPN, feature maps are concatenated in a top-down fashion (see Figure 2.20(a)). Propagating spatial information through the entire network is lengthy, and it is possible to generate duplicate predictions. Whereas in PAN (see Figure 2.20(b)), feature maps are concatenated in a bottom-up path by using clean lateral connections to record spatial information and therefore shortening this path. An element-wise max operation is applied at the end to prevent duplicate predictions, [91]. The overall architecture of YOLOv4 is in Figure 2.21. Note that in Figure 2.21, the convolutional layers, dense connection layers, SPP layers, and fully-connection layers are depicted in red, blue, green, and yellow, respectively.

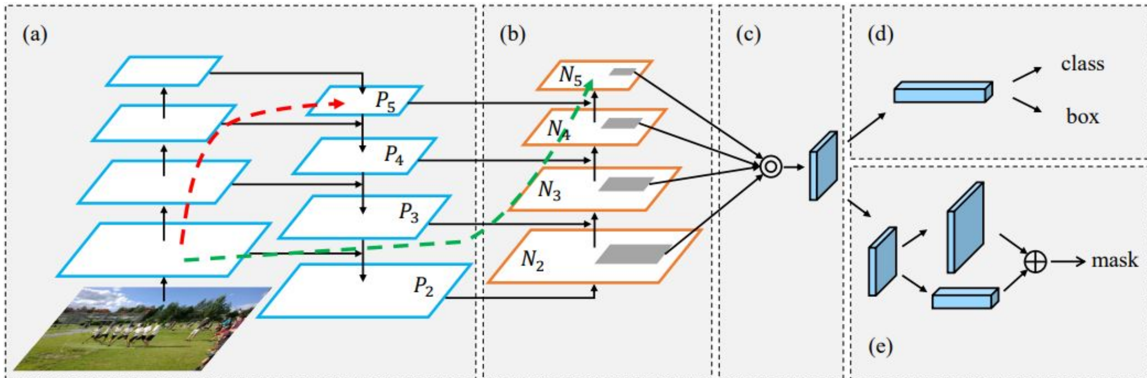

Figure 2.20: (a) In FPN, the top-down feature maps are labeled as P5-P2, [87]. (b) In PAN, the bottom-up path is labeled as N2-N5, [91].

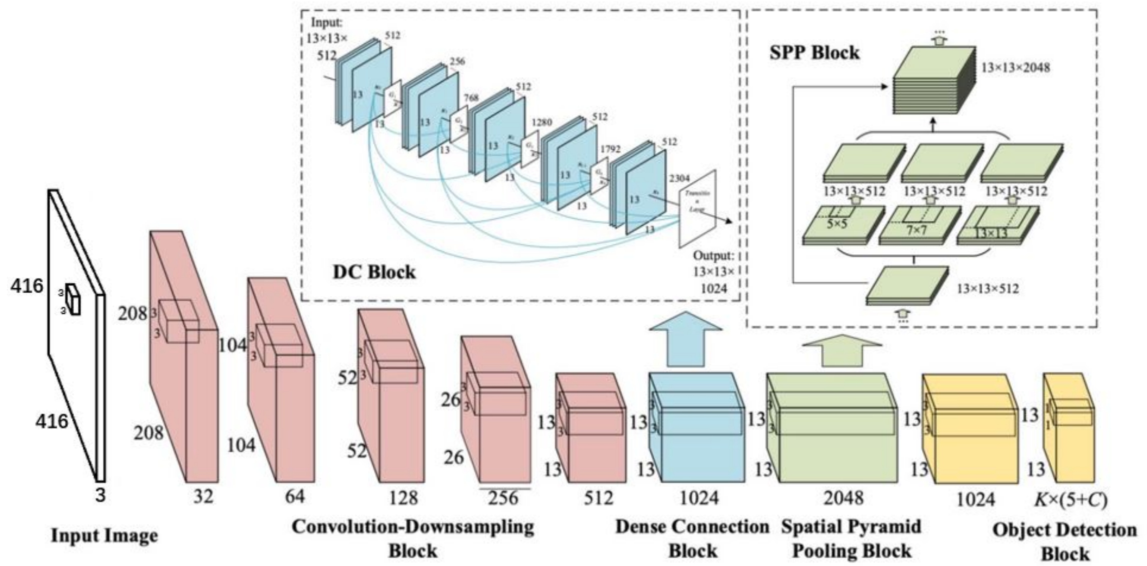

Figure 2.21: The YOLOv4's architecture, [90].

Chapter 3

Camera Calibration

Camera calibration, also known as camera resectioning is the act of estimating the parameters of a camera lens and the image sensor. Calibrating regular cameras is well-studied. However, using these traditional methods to calibrate an infrared camera is usually challenging, especially during winter season. A camera perfroms a mapping from the three-dimensional world to a two-dimensional image. When taking a picture, four coordinate systems (reference frames) are involved, the world coordinates, the camera coordinates, the image coordinates, and the pixel coordinates. The location of an object is defined as (X_w, Y_w, Z_w) in the three-dimensional world coordinates, the camera has three-dimensional coordinates defined as (X_c, Y_c, Z_c), an image has two-dimensional coordinates defined as (x, y), and finally, the two-dimensional pixel coordinates is defined as (u, v). To reconstruct the three-dimensional world from the pixels, the camera's parameters need to be determined. These parameters are obtained through camera calibration.

3.1 The Camera Parameters

The camera's parameters consist of intrinsic, extrinsic, and distortion coefficients, [92]. The intrinsic parameters describe the geometric properties of the camera, and it is used to transform between camera coordinates and image coordinates. This projection is modeled by the ideal pinhole camera as illustrated in Figure 3.1. The extrinsic parameters describe how to transform a point from world coordinates to camera coordinates.

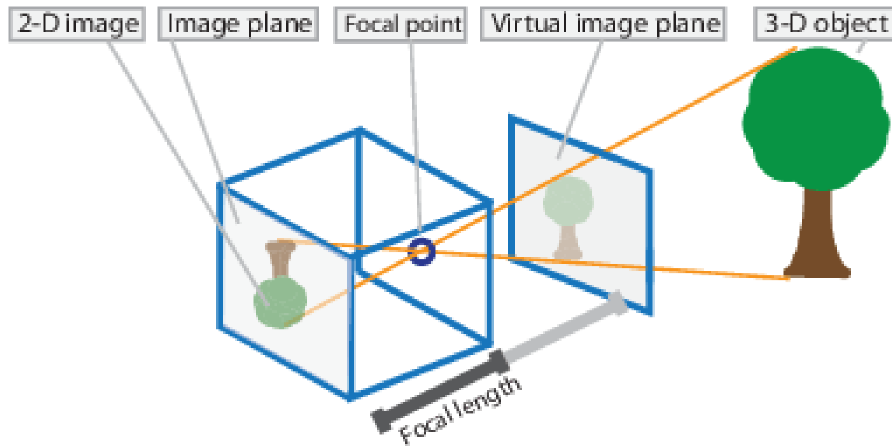

Figure 3.1: The ideal pinhole camera, [93].

The intrinsic parameters are defined through a matrix K:

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where f_x and f_y are the focal length measured in pixels, c_x and c_y are the principal point offsets, and s is the skew coefficient. When the image axes are not perfectly

perpendicular, the skew coefficient s can be calculated using:

$$s = f_x \tan \alpha \tag{3.1.1}$$

where

$$f_x = \frac{F}{p_x} \tag{3.1.2}$$

F represents the focal length in world units usually in millimeters, and p_x is the size of the pixel in world units. The measurement of α, p_x can be seen in Figure 3.2.

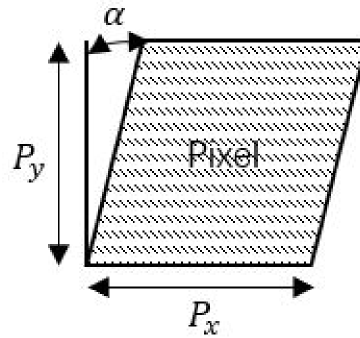

Figure 3.2: The pixel skew.

The extrinsic parameters consist of a 3×3 rotation matrix R and a three-dimensional translation vector t defined as:

$$R = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} \\ r_{2,1} & r_{2,2} & r_{2,3} \\ r_{3,1} & r_{3,2} & r_{3,3} \end{bmatrix} t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

The translation vector tells the relative positive between the origins of the two coordinates, and the rotation matrix represents the directions of the world axes in camera

coordinates. Using the entrinsic parameters, the relationship between the coordinates of a point P in the world (P_w) and camera (P_c) is characterized as:

$$P_c = R(P_w - t) \tag{3.1.3}$$

Combining t and R, the result is called the extrinsic matrix defined as $[R \mid t]$. The homogeneous transformation can be writen as:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = K[R \mid t] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{3.1.4}$$

The distortion coefficients account for both radial and tangential distortion. Radial distortion happens near the edges of a lens, and a distorted point is denoted as $(x_{distorted}, y_{distorted})$. To correct radial distortion, a set of radial distortion coefficients of the lens is needed, denoted as k. Transferring from $(x_{distorted}, y_{distorted})$ to the undistorted (x, y), the following equations can be used:

$$x_{distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{3.1.5}$$

$$y_{distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{3.1.6}$$

where k_1, k_2, and k_3 are three of the radial distortion coefficients, and $r^2 = x^2 + y^2$. Tangential distortion happens when the lens is not perfectly aligned with the image plane. Transferring from $(x_{distorted}, y_{distorted})$ to the undistorted (x, y), use:

$$x_{distorted} = x + (2p_1xy + p_2(r^2 + 2x^2)) \tag{3.1.7}$$

$$y_{distorted} = y + \left[p_1(r^2 + 2y^2) + 2p_2xy)\right] \tag{3.1.8}$$

where p_1 and p_2 are the tangential distortion coefficients of the lens, and $r^2 = x^2 + y^2$.

3.2 The Implementation

The traditional camera calibration methods can be categorized into photogrammetric calibration and self-calibration, [94]. The photogrammetric calibration method is performed by observing a calibration object whose dimension is known in the world coordinates. The calibration object or pattern can be three-dimensional, two-dimensional, or even one-dimensional. The checkerboard pattern is widely used because the sharp gradient between the black and white cells is easy to localize. By taking a series of images of the checkerboard from different viewpoints, a set of three-dimensional points in world coordinates can be obtained. Then these points can be projected to the two-dimensional pixel coordinates by using the method proposed in [92] which multiplies the three-dimensional world points by the extrinsic matrix. This projection can be automated by using software applications such as OpenCV and MATLAB. The self-calibration methods observe the geometric rigidity of the static world, so it can be performed by just moving the camera around. Although these methods do not require any calibration object, they cannot always yield accurate results.

In this experiment, a calibration method that requires the least amount of apparatus and has the simplest setup was wanted. Therefore, based on the instrumentation available on hand, it was decided to use the checkerboard pattern approach along

with the camera calibrator toolbox provided by MATLAB.

3.3 A Creative Calibration Method for Infrared Cameras

When a checkerboard is left in room temperature, the temperature of the black cells is higher than the white cells due to emissivity and absorption. However, this temperature gradient is usually less than 1 °C and is not sufficient for the infrared camera to detect the corners. In most cases, heating the checkerboard with direct sunlight is good enough to create a perceivable temperature gradient for the infrared camera because black absorbs the most heat. However, during winter, cold temperatures will actually cool down the checkerboard. Prakash et al. proposed a method that uses a heat lamp to heat the checkerboard, but this method requires careful timing, [95]. If heated for too long, all cells will have similar temperatures. Conversely, the temperature gradient is not perceivable as illustrated in Figure 3.3

Figure 3.3: (Left) Untreated. (Middle) Heated for too short. (Right) Heated for too long

There are methods where a checkerboard made of two materials is used, but since it requires a lot of manufacturing expertise and presumably still does not work

in cold temperatures, they are not adopted in this experiment. After evaluating the feasibility of all these methods, a new checkerboard has been designed that is aimed to calibrate infrared cameras in cold temperatures. Rather than heating a single-layered checkerboard, the new 10×7 checkerboard is designed with detachable black cells of 34.5mm in side length. To make this possible, a sheet of plain white cardboard is used as the background, and overlapped by a sheet of black cells as shown in Figure 3.4. Cardboard is used as it is less susceptible to crinkles and bends and therefore creates less unwanted distortion. Ideally, the checkerboard should be printed on a completely rigid body.

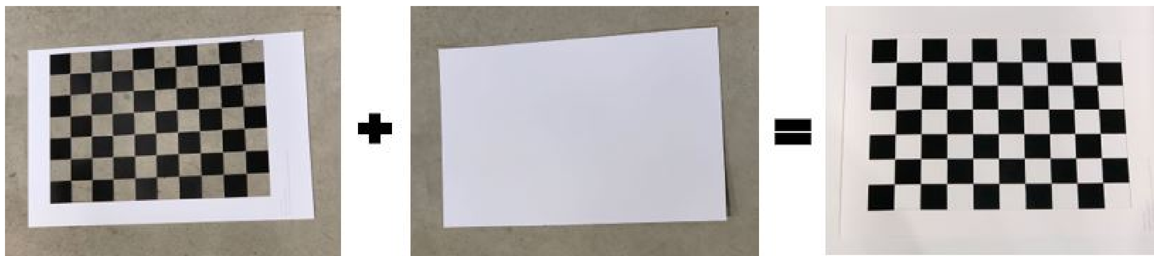

Figure 3.4: The checkerboard used in this experiment.

The sheet of black cells is detached and heated alone using a heat lamp, and the white cardboard is placed outside for cooling. The white layer can be left untreated (∼10 °C), but for a better result, it is preferred to cool the white layer. During this experiment, the temperature outside is around 0 °C. The black cells are heated to around 36 °C. The two layers are then quickly overlapped and placed in front of the camera. By using this method, the 36 °C in temperature gradient is significant enough as shown in Figure 3.5.

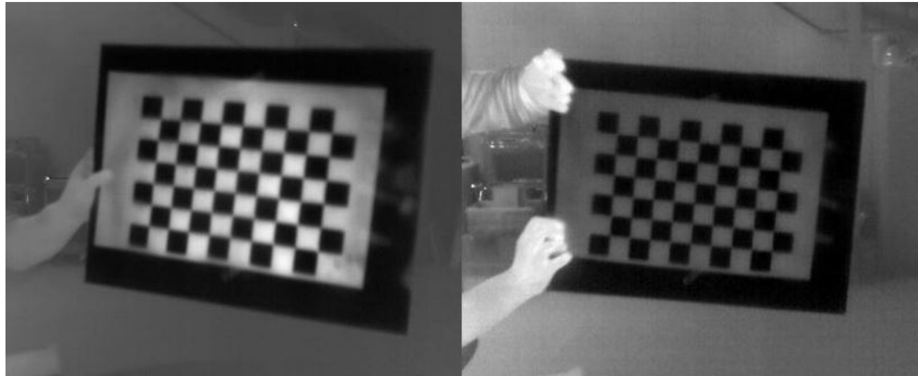

Figure 3.5: (Left) When the white layer is left outside for cooling. (Right) When the white layer is left in room temperature (∼10 °C).

Becasue the FLIR A65 camera used in this experiment is mounted on top of a vehicle and cannot be moved, the checkerboard must be posed at different angles in front of the camera. As the intrinsic parameters are estimated using this camera-centric orientation, it is assumed that the camera is placed at the origin in the world coordinates. A total of 185 images are captured during the calibration session and the orientation of the images can be seen in Figure 3.6. The image axes at each location are along the edges of the checkerboard. The origin of the image coordinates is located at the bottom-right corner of the top-left black cell as indicated by the yellow square in Figure 3.7. A total of 54 corners are detected in each image and compared with the reprojected points from the world coordinates.

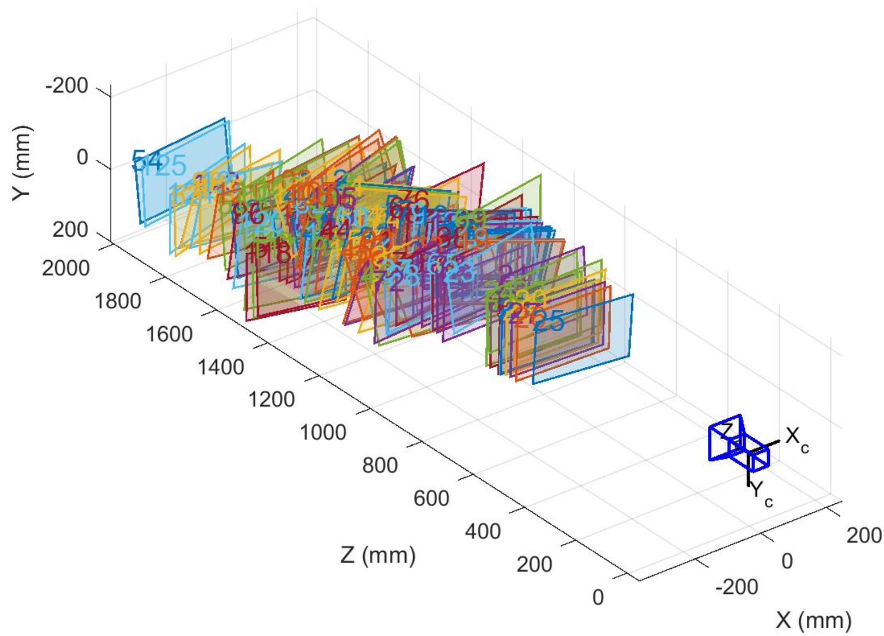

Figure 3.6: The orientation of the images during calibration.

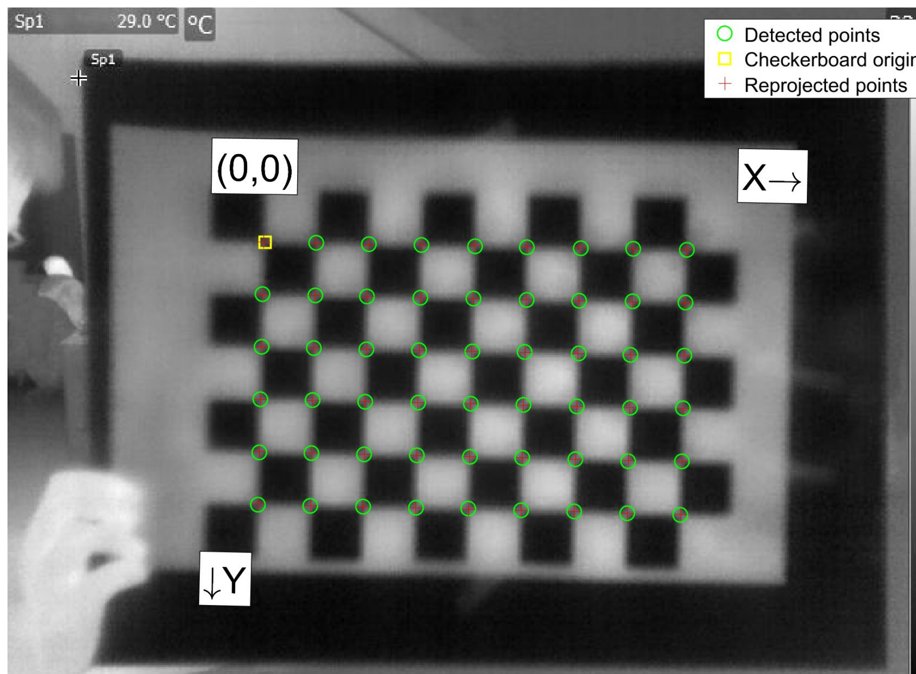

Figure 3.7: The detected corners on the checkerboard.

3.4 The Calibration Result

The result of this calibration session lead to 156 images being accepted, 29 rejected (failed to detect corners), and an initial mean reprojection error of 0.28 pixels. After removing some outliers, an overall mean reprojection error of 0.16 pixels is obtained. The estimated intrinsic parameters are:

$$K = \begin{bmatrix} 778.1765 & 0 & 324.6721 \\ 0 & 776.1906 & 245.6962 \\ 0 & 0 & 1 \end{bmatrix}$$

The extrinsic parameters can be measured either manually or automatically by using MATLAB. When manually measuring the extrinsic parameters, the position and orientation of the camera in the world coordinates need to be known. This measurement has 6 degree-of-freedom, the camera's x, y, and z position and its row, yaw, and pitch. Because physically measuring these parameters can be inaccurate due to human factors and the uncertainties in the measuring device, so it is decided to use MATLAB to automate the measurement. The origin of the world coordinates is defined to be at the front-most point of the vehicle and the directions of the axes are shown in Figure 3.8.

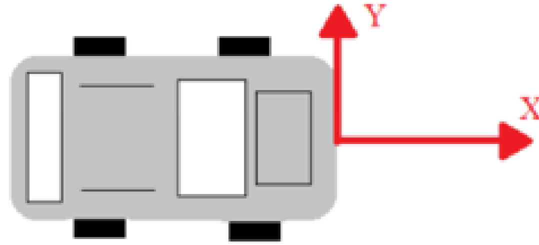

Figure 3.8: The world coordinates.

Using the intrinsic matrix and a checkerboard placed at the origin of world coordinates, the extrinsic parameters can be determined by using MATLAB's camera calibrator or the built-in extrinsics function. Combining the translational vector and the rotation matrix, the extrinsic matrix is:

$$[R \mid t] = \left[\begin{array}{ccc|c} 0 & -1 & 0 & 0.169 \\ -0.1659 & 0 & -1.0741 & 1.2931 \\ 1.0741 & 0 & -0.1659 & 2.2212 \end{array}\right] \quad (3.4.1)$$

Because during the calibration, no pixel skew or tangential distortion is found, and the FLIR A65 camera only exhibits minimal radial distortion. Therefore, it is decided that no correction is needed for skew and distortion.

Chapter 4

A You Only Look Once (YOLO) Based Approach

When developing a real-time object detection and image classification application, a widely adopted approach is to employ the CNN architectures. These CNNs usually have a fixed structure and offer the "plug-and-play" ability that allows users to easily train the model. When training a CNN model such as the YOLO, the traditional vision pipeline can be automated by the network. Because these real-time detectors pursue faster speed, they generally do not use complex methods to preprocess the frames while streaming. However, because images are noisy, adding additional preprocessing can eliminate the disturbance in pixels and potentially improve the detection results. Therefore, instead of modifying the YOLO's architecture, this chapter focuses on studying how additional preprocessing influences the YOLO's real-time performance and investigating the impact on the overall accuracy when altering the network size. The full training process of a YOLOv4 network based on a custom infrared driving dataset will be explained. Moreover, some advanced data manipulation

and training techniques will also be explained.

4.1 Image Preprocessing

In YOLO, the network or input size determines the size of the input image at the first layer of the network. Generally, YOLO takes images of any size and its internal resizing mechanism alters the images to match the network size. Some commonly used network sizes are 320×320, 416×416, and 608×608. When the network size gets bigger, the accuracy will be improved correspondingly. Conversely, when the network size gets smaller, the network will process at a faster speed. Although the original YOLO paper did not mention any specific image preprocessing other than resizing that has been used in the training process, it is assumed that the filtering methods can also be applied to the YOLO to further improve accuracy.

Because YOLO is designed to be a "real-time" detector, so the processing time of each filtering technique in Chapter 2.2.3 is tested on a sample image to validate if additional filtering will affect the YOLO's real-time performance. Note that the term "real-time" can be defined in many ways, but in this thesis, real-time refers to 30 FPS. A timer is implemented in python by using the "time" library and started when the grayscaling takes place and stopped when the filtering is done. All unnecessary processes in the background are killed during the experiment to prevent interference. Each filtering technique has been executed 10 times, and the average processing time is recorded in Table 4.1.

Network size (image size)	Average filtering	Gaussian filtering	Bilateral filtering	Median filtering	Non-Local Means
320×320	0.000496s	0.000496s	0.001488s	0.000992s	0.156239s
416×416	0.000495s	0.000516s	0.002976s	0.001487s	0.259407s
608×608	0.000495s	0.000496s	0.002480s	0.001983s	0.182528s

Table 4.1: The average processing time of each filtering method at different network sizes.

From Table 4.1, because most filtering techniques can be done in less than 1ms, it is concluded that these filtering methods can also be applied to YOLO and will not affect its real-time performance at 30FPS. However, some complex filtering techniques such as the Non-Local Mean denoising take longer to process, and can potentially affect the YOLO's real-time performance at 30FPS.

4.2 Data Augmentation

Data augmentation is a strategy that increases data diversity. Some commonly used data augmentation practices can be put into two categories: traditional transformation, [96] and Generative Adversarial Networks (GAN), [98]. Methods under the traditional transformation category mainly involve affine image transformation and color modification including rotation, reflection, scaling (zooming in or out), shearing, histogram equalization, enhancing contrast or brightness, white-balancing, sharpering, and blurring, [96]. These methods have been proved to be fast and reliable and effective in increasing the number of training samples and balancing the size of the

dataset. The way how data augmentation is done in GANs is that two adversarial networks are used where one generates "fake" images to trick the other network to misclassify. Although GANs can produce satisfactory results, adversarial networks are usually difficult to train. In YOLOv4, three data augmentation methods, Cutmix [99], Mosaic, and Self-Adversarial Training (SAT) have been used, [90]. Each of the method will be explained in greater detail below.

4.2.1 CutMix Data Augmentation

Given two images, x_A and x_B and their labels, y_A and y_B, CutMix is a augmentation strategy that removes ("cut") a patch from x_A and filled ("mix") it with a patch from x_B as shown in Figure 4.1. The ground truth labels or the bounding boxes, y_A and y_B are also mixed together proportionally to the number of pixels in the combined image.

Figure 4.1: The CutMix, [99].

When using CutMix in a CNN, the network's localization ability can be further improved because the network is forced to identify an object from its partial view. The CutMix operation can be mathematically defined as:

$$\tilde{x} = M \odot x_A + (1 - M) \odot x_B \tag{4.2.1}$$

$$\tilde{y} = \lambda y_A + (1 - \lambda) y_B \tag{4.2.2}$$

where $(\tilde{x}, \tilde{y})$ is the resulting sample from the CutMix operation, $M \in \{0,1\}^{W \times H}$ is a binary mask that indicates the location of the patch, $\odot$ is the Hadamard or Schur product [100] that represents the element-wise multiplication, and λ is the combination ratio. In the original CutMix paper by Yun et al., the λ was sampled from a uniform distribution, [99]. The location of the patch can be represented by the bounding box coordinates $B = (r_x, r_y, r_w, r_h)$, and the box coordinates are uniformly sampled:

$$r_x \sim Unif(0, W), \quad r_w = W\sqrt{1 - \lambda} \tag{4.2.3}$$

$$r_y \sim Unif(0, H), \quad r_h = H\sqrt{1 - \lambda} \tag{4.2.4}$$

where $\frac{r_w r_h}{WH} = 1 - \lambda$. The values within the binary mask M are all 0, and the values outside M are all 1. Therefore, the operation $M \odot x_A$ removes the patch from x_A as the results are all zero, then $(1 - M) \odot x_B$ preserves and extracts the patch from x_B. Finally, $\tilde{x}$ is formed by adding them together. During training, a new $(\tilde{x}, \tilde{y})$ is generated by combining every two randomly selected images within the mini-batch. According to the original paper, CutMix has a negligible computational overhead, so it can be efficiently used in training any network architecture, [99].

4.2.2 Mosaic Data Augmentation

YOLOv4 introduces a new data augmentation technique called the Mosaic, [90]. In Mosaic, a 2×2 grid is created, and 4 randomly selected training images and their labels are resized and then put into each location of the grid. Finally, a window is placed at a random location around the grid center to crop out the image. The content within the window is the resulting augmented image. This process can be seen in Figure 4.2.

Figure 4.2: The Mosaic data augmentation.

For example in Figure 4.2, when training a 416×416 sized network, 4 randomly selected images from the training set are resized to 416×416 and combined to form a 2×2 grid. The combined image has the size of 832×832. Then, a 416×416 window (the red box in 4.2) is placed at a random location around the center of the 2×2

grid. Finally, the combined image within the red box is the Mosaic-augmented result and will be extracted and used as a new training sample. A new augmented image will be generated by combining every 4 images within the mini-batch. Similar to the CutMix, the Mosaic augmentation also force the network to identify partial objects and thus enhance the network's localization capability.

4.2.3 Self-Adversarial Training

Self-Adversarial training is another data augmentation technique that has been used in YOLOv4, and the idea of this technique is to use an adversarial CNN to alter the image. Briefly speaking, adversarial networks generate adversarial images by adding a small perturbation to the input pixels, and this perturbation is generally derived from the gradient of the cost function, [97][98]. Although the perturbation can sometimes be imperceptible by human eyes, the detector model might produce a completely different result. Adversarial examples are usually used to improve the stability, reliability, and accuracy of a detector model.

The SAT in YOLOv4 involves two forward-backward stages, [90]. In the first forward pass, the network is trained on the image as usual. However, the backpropagation alters the image instead of updating the weights. In this way, the network performs an adversarial attack on itself and creating false information that says no object of interest is in the image. In the second forward pass, the network is trained to detect objects from the altered images, and the backpropagation updates the weights in the normal way.

4.3 Feature Extraction

In the realm of machine learning, feature extraction is generally done by convolving the image with a manually designed kernel. For example, when extracting edges, the Sobel, Roberts, Prewitt, and Laplacian kernel can be used. Similarly, the feature extraction in YOLO is also done by convolution, but just a series of convolutions using convolutional layers. Each convolutional layer has many filters of a predefined size, and each filter produces a feature map that stacks on top of each other. The feature maps are propagated in the forward direction to the next convolutional layer that repeats the same process. Eventually, after backpropagation, the trained weights are an abstract representation of the features.

In general, pre-trained CNNs are commonly used as the basic building block for a deep learning based detector. Some common features are saved in these networks and can be directly used to make a prediction. Alternatively, base upon the pre-trained model, the features from a custom dataset can be learned by using transfer learning, [101]. In YOLOv4, the feature extractor is a CNN called the CSPDarknet53, and it is commonly pre-trained on the ImageNet. The CSPDarknet-53 perfroms consectutive 3×3 and 1×1 convolutions on the images to generate feature maps.

4.4 Transfer Learning

When training a deep neural network to do tasks like image classification and object detection, transfer learning is commonly used, [102]. Transfer learning usually refers to the technique that reuses a previously trained model as the starting point of training a new model. In image classification, transfer learning works by first training

a base network on some large and challenging datasets such as the ImageNet, MS COCO, or PASCAL VOC. Then the first n layers of this base network are copied to the new network, [103]. The base network is called the pre-trained model and usually remains unchanged during the rest of the training. However, if the new network requires fine-tuning, then the errors can be backpropagated through the entire network including the base network.

Because the features at the beginning of the deep neural network are usually "general", and the features close to the end of the network are "specific" to the dataset and the corresponding task, the dataset used in the new task can refine these "general" features from the base network and makes the output features specific to the new task. Using the "general" features provided by the base network, the training time can be significantly reduced compared to training the entire network from scratch. Moreover, because the new network refines the "general" features rather than learning "general" features, the overall performance can also be improved.

4.5 Dataset Preparation and Labelling

The object classes in this thesis are designed to cover people, vehicles, bicycles, and animals, and therefore the experiment vehicle has been driven in two scenarios, urban and highway. The urban area is located next to a university campus, and this scenario is mainly focusing on capturing people, bicycles, and animals (dogs for now). Because a vehicle can look very different in an infrared camera from different angles as shown in Figure 4.3, the highway scenario is catered to capture only vehicles to ensure all angles of the vehicle and all types of vehicle are covered in the dataset.

Figure 4.3: (Left) The side view of a vehicle. (Middle) The rear view of a vehicle. (Right) The front view of a vehicle.

There are around 9,000 images captured and after removing some images with no object in them, the resulting dataset has 8,192 infrared frames. In addition to this custom dataset, FLIR provides a dataset - "FREE FLIR Thermal Dataset for Algorithm Trainin" that offers 10,228 more labeled infrared images, [104]. This dataset contains 28,151 labeled persons, 46,692 labeled vehicles, 4,457 labeled bicycles, and 240 labeled dogs. These images cover all the object classes of interest in this project. Based on the available feedbacks, using this dataset alone can achieve relatively good performance. However, there are only 240 dogs present in this dataset. After carefully examining the dogs, most of them are small and blended in the background. Because these dog objects need to be further split into a training, testing, and a validation set, it is believed that only 240 objects are not enough. YOLO recommends having more than 2,000 objects for each class. Moreover, small objects usually cannot provide representative features. Therefore, it is decided to combine the self-labeled dataset with the FLIR dataset to maximize data variety.

A labeler (the author of this thesis) labeled every object in the custom dataset using a bounding box that encloses the object as tight as possible (with a minimal amount of background included). When labeling people and dogs, the labeler was

instructed to enclose the object as long as its main body (chest) is visible. If only one limb (i.e. an arm, a leg, or the tail) is visible then it is not labeled. When pedestrians appear in crowds, each individual is labeled if it can be separated from the crowd. Otherwise, the individuals are labeled as a whole. Cyclists are labeled as two objects, the bicycle and the person who's riding it. Because when someone is riding the bicycle, the handle is usually covered up and thus it is decided to enclose only the wheels. However, when bicycles appear alone, the entire bicycle is labeled. Personal accessories such as a backpack are not included in the bounding box. Overall, if the object is partially visible and is not distinguishable for the labeler, it is not labeled. There are a total of 7,579 persons, 5,546 vehicles, 1,031 bicycles, and 1,692 dogs labeled in this session. Figure 4.4 shows an example of a labeled image.

Figure 4.4: (Top) The original image frame. (Bottom) The labeled image frame.

When labeling a dataset to train the YOLO network, a unique labeling format is required by YOLO. In most applications, although the methods used to draw the

bounding boxes may vary, but the locations of the bounding boxes are usually represented by the pixel coordinates of the corners. For example, in a typical OpenCV application, the bounding boxes are produced by OpenCV's "rectangle" function. To draw a bounding box using the "rectangle" function, the acutal coordinates of the top-left and the bottom-right corner are required, [105]. Whereas in YOLO, each bounding box is defined by its normalized central coordinates, width, and height. Because the values are normalized, so they are floating point numbers that fall between 0 and 1, [106]. A typical bounding box in YOLO has the following format

$$B \ = < class > < x > < y > < width > < height >$$

where "class" is a non-negative integer number that defines the object class, and

$$\begin{aligned} x &= absolute_x \ / \ image_width \\ y &= absoulte_y \ / \ image_height \\ width &= absolute_width \ / \ image_width \\ height &= absolute_height \ / \ image_height \end{aligned}$$

where $absolute_x$, $absolute_y$, $absolute_width$, and $absolute_height$ are a bounding box's actual central coordinates, width, and height, respectively.

4.6 Training a YOLOv4 Network

Given the prepared dataset, a 608×608 network is first trained using the open-source framework "darknet" following the same strategy as described in the original paper, so none of the filtering techniques other than resizing is used. A pre-trained model on ImageNet is used as the starting point. Transfer learning is used later in the process to further train the network on the custom dataset. 70% of the data in the custom dataset are used for training, 15% are used for validation, and the rest 15% are used for testing. The training involves 15,000 iterations, and 64 images are loaded for each iteration. These 64 images are then split into 16 mini-batches, so there are 4 images in each mini-batch. The training is done on a Nvidia GTX 1080Ti GPU, and it requires 127.294 billion floating point operations per second (BFLOPS). In addition to the 608×608 network, a 320×320 and a 416×416 network are also trained using the same 70%-15%-15% data split. The training results at 0.5 IoU are in Table 4.4 to Table 4.2. Note that 0.5 is a popular IoU threshold that has been vastly adopted by other mainstream datasets, so $IoU = 0.5$ will also be used throughout the rest of this thesis.

320 × 320 full YOLOv4					
Class	TP	FP	FN	Precision	Recall
People	4271	1774	1508	70.7%	73.9%
Bicycle	1646	783	739	67.8%	69.0%
Vehicle	4660	1866	772	71.4%	85.8%
Animal	581	291	245	66.7%	70.3%
35.262 BFLOPS					

Table 4.2: The training results of a 320 × 320 full YOLOv4 network.

416 × 416 full YOLOv4					
Class	TP	FP	FN	Precision	Recall
People	4517	1964	1262	69.7%	78.2%
Bicycle	1742	841	698	67.4%	71.3%
Vehicle	4801	2155	631	69.0%	88.4%
Animal	599	323	217	65.0%	73.4%
59.592 BFLOPS					

Table 4.3: The training results of a 416 × 416 full YOLOv4 network.

608 × 608 full YOLOv4					
Class	TP	FP	FN	Precision	Recall
People	4788	1913	991	71.5%	82.9%
Bicycle	939	421	228	69.0%	80.5%
Vehicle	4896	1892	536	72.1%	90.1%
Animal	635	314	170	66.9%	78.9%
127.294 BFLOPS					

Table 4.4: The training results of a 608 × 608 full YOLOv4 network.

Comparing the results in Table 4.4 to Table 4.2, the 608×608 network has the highest accuracy and recall, but it also requires the most computing power. Whereas the 320×320 network requires the least computing power, but the accuracy and recall are not the lowest. An interesting observation is that although the 416×416 network is bigger than the 320×320 network, it actually produces worse results. According to the results, when the system does not have a bottleneck in computing power, increasing the network size can, but not always lead to an increase in precision. The root cause of the performance reduction in the 416×416 network will not be investigated at the current stage. Although using the full 608×608 YOLO while achieving around 70% of mean average precision is not perfect, but it is rather acceptable. However, whether keep increasing the network size will still increase the precision remains skeptical, and will not be investigated in this thesis. If more data for the bicycle and animal class becomes available, it is believed that these networks can achieve even higher mean average precision.

4.7 Training a Tiny-YOLOv4 Network

Sometimes, due to hardware limitations, using the full YOLOv4 network for real-time detection might not possible. To solve this, a compressed variation of the YOLOv4, namely the tiny YOLOv4 is also trained. Instead of using the CSPDarknet-53, the backbone of the tiny YOLOv4 only has the first 29 convolutional layers. Using the same setup, 16 mini-batches, 4 images per mini-batch, 15,000 iterations, 70%-15%-15% data split, a pre-trained model (on ImageNet), and transfer learning, the training results on a Nvidia GTX 1080Ti GPU at 0.5 IoU is in Table 4.5 to Table 4.7.

320 × 320 tiny YOLOv4					
Class	TP	FP	FN	Precision	Recall
People	2630	1816	3149	59.2%	45.5%
Bicycle	1013	802	1543	55.8%	39.6%
Vehicle	3906	1873	1526	67.6%	71.9%
Animal	494	409	527	54.7%	48.4%
4.020 BFLOPS					

Table 4.5: The training results of a 320 × 320 tiny-YOLOv4 network.

416 × 416 tiny YOLOv4					
Class	TP	FP	FN	Precision	Recall
People	3288	1731	2496	65.5%	56.8%
Bicycle	1267	764	1223	62.4%	50.9%
Vehicle	4262	1713	1170	71.3%	78.5%
Animal	539	374	404	59.0%	57.2%
6.793 BFLOPS					

Table 4.6: The training results of a 416 × 416 tiny-YOLOv4 network.

608 × 608 tiny YOLOv4					
Class	TP	FP	FN	Precision	Recall
People	3804	2152	1975	63.9%	65.8%
Bicycle	1466	931	968	61.2%	60.2%
Vehicle	4433	1847	999	70.6%	81.6%
Animal	623	441	345	58.5%	64.4%
14.512 BFLOPS					

Table 4.7: The training results of a 608 × 608 tiny-YOLOv4 network.

Because the tiny YOLOv4 only uses the first 29 convolution layers, there is a significant drop in accuracy and recall. Although the tiny YOLO only requires a fraction of the computing power compare to the full YOLO, only getting around 65% of mean average precision with the largest 608×608 tiny network is not ideal. The performance drop is more drastic for the classes with fewer training samples such as the bicycle and animal class, especially in a smaller-sized network. Comparing the tiny YOLOv4 at 608×608 with the corresponding full YOLOv4, the best performing vehicle class exhibits a 2% drop in precision, but other classes have around an 8% reduction in precision. This can be explained by the number of training samples used by each class. Because the vehicle class has the most training samples, it is the most stable class. Another interesting observation is that although the 608×608 network is bigger than the 416×416 network, the precision is actually lower. Again, the cause of this phenomenon will not be investigated in this thesis. Overall, The poor performance in the bicycle and animal class can be explained by the lack of training samples. If more training samples can be added, getting a 70-80% of overall precision should not be difficult.

4.8 Running YOLOv4 and Tiny-YOLOv4

Deciding which network to be used in the final model cannot be determined solely based on precision and recall. The confidence score is another important criterion. Running the YOLOv4 and Tiny-YOLOv4 networks on some test images, Figure 4.5 is used to visualize the detection results because it has the most objects in it. Although the confidence scores are not given in Figure 4.5, they will be listed and explained below.

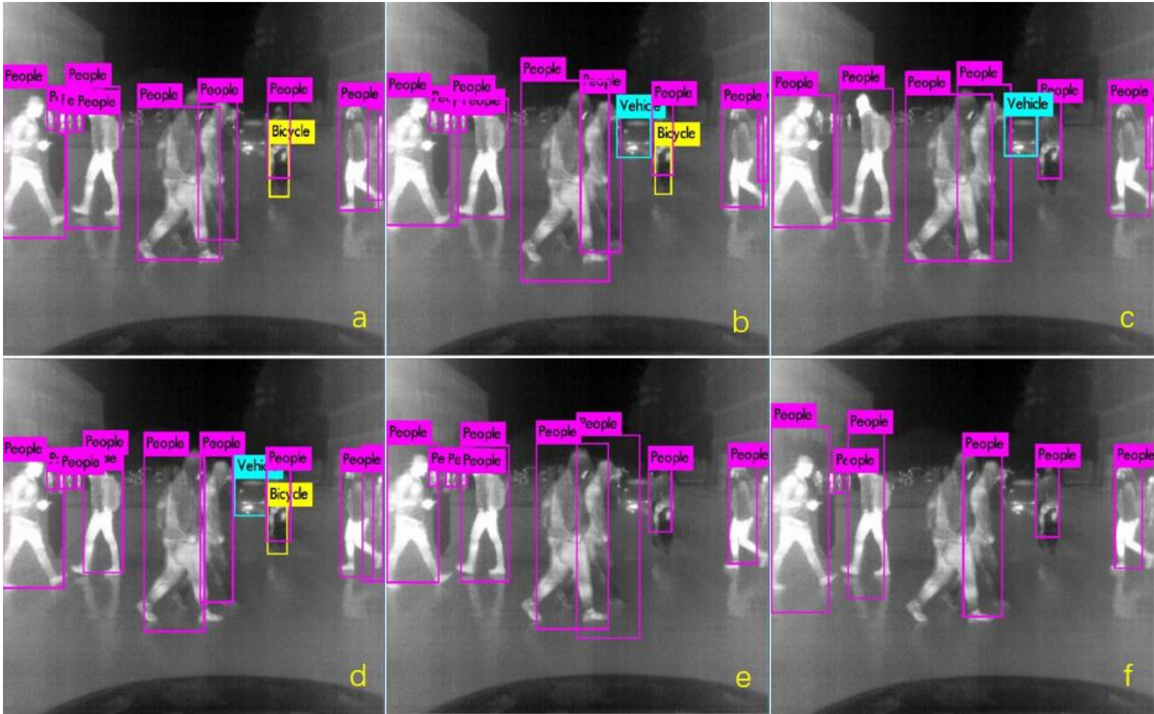

Figure 4.5: (a) Running the full YOLOv4 at 320 × 320. (b) Running the full YOLOv4 at 416 × 416. (c) Running the full YOLOv4 at 608 × 608. (d) Running the Tiny-YOLOv4 at 320 × 320. (e) Running the Tiny-YOLOv4 at 416 × 416. (f) Running the Tiny-YOLOv4 at 608 × 608.

Comparing the results across the full YOLOv4 networks in Figure 4.5, although the full 608 × 608 YOLOv4 network has the highest mean average precision, the full 416 × 416 YOLOv4 network detects the most objects. However, the confidence score (not shown in Figure 4.5) of each detected pedestrian fluctuates between 85%-100% in the full 416 × 416 YOLOv4 network, but maintains at 97%-100% in the full 608 × 608 YOLOv4 network. Similarly, the full 608 × 608 YOLOv4 network is around 94% confident of each detected vehicle, but the full 416 × 416 YOLOv4 network is only around 84% confident. Note that the confidence score in this section refers to how confident the network is about each detected object in Figure 4.5. Because no

animal is presenting in Figure 4.5, the confidence score for the dog class will not be discussed. Based on precision and confidence score, the 608×608 full YOLOv4 network is selected and will be used in future comparisons because it has the best overall performance.

Comparing the results across the tiny-YOLOv4 networks, although the tiny 320×320 YOLOv4 network has the lowest mean average precision, it detects the most objects and is 28%-96% confident of the detected pedestrians, 30% confident of the bicycles, and 36% confident of the vehicles. Whereas the tiny 608×608 YOLOv4 network is 41%-95% confident of the pedestrians, 45% confident of the bicycles, and 25% confident of the vehicles. Despite that the tiny 608×608 YOLOv4 network misses some of the objects in the test image, it still gets a relatively higher confidence score for the detected objects. Although the 608×608 tiny YOLOv4 network has a slightly lower mean average precision than the 416×416 tiny network, it has the highest confidence score. Therefore, the 608×608 tiny YOLOv4 network will be used in the final comparison.

In addition, the average processing time of each network is recorded in Table 4.8. Note that the resulting processing time and FPS in Table 4.8 are generated from a Nvidia GTX 1080Ti GPU. When running the network on a CPU, it will be relatively slower and at a lower FPS. Overall, the full 608×608 YOLOv4 network will be used as the final model as it produces the highest mean average precision along with the highest confidence score. However, if computing power really becomes a hindrance , the tiny 608×608 YOLOv4 network might be preferable as it has the highest confidence.

320×320 Full	16.237ms	61.59FPS
416×416 Full	20.992ms	47.64FPS
608×608 Full	28.781ms	34.75FPS
320×320 Tiny	2.473ms	404.37FPS
416×416 Tiny	4.007ms	249.56FPS
608×608 Tiny	5.294ms	188.89FPS

Table 4.8: The average processing time (also converted to FPS) of each YOLOv4 network.

Chapter 5

A Support Vector Machine (SVM) Based Approach

Because the most commonly used CNN models have gone through many iterations of development, these developed models are usually easier to train on a custom dataset while achieving satisfying results. However, a "fixed" structure usually lacks the flexibility in method selection and parameter tuning. Therefore, in this chapter, a traditional SVM is built to compare with the YOLO and see if more flexibility leads to improved performance.

When building a classic machine learning classifier, a common approach is to follow the traditional vision pipeline. In the vision pipeline, images are first preprocessed to remove any unwanted noise. Then feature extraction methods are applied to capture the common characteristics within training samples. Finally, a classifier is trained for decision-making. In this chapter, following the same flow, some commonly used image preprocessing methods are first studied. Then, some popular edge extraction methods are reviewed. Next, an SVM classifier is built for decision-making, and

finally, the entire pipeline is applied to a selected dataset and evaluates the impact of different preprocessing and feature extraction methods on the classification result.

5.1 Color Space Conversion

Red, green, and blue (RGB) is a three-channeled color scheme that has been widely used in digital images. Given a RGB image, each pixel is represented by a combination of the three color values. For some image processing tasks, color information is important, so it is necessary to preserve the color information by using RGB format. However, in autonomous driving related computer vision applications, especially in object detection, the presence of an object can usually be determined without mentioning the color. Therefore, images are usually converted to grayscale in these applications for simplicity. There are exceptions where some edges are difficult to detect in a grayscale image, but because thermal images look mainly black-and-white, reducing to a single color channel is enough to represent all possible colors and distinguish most edges in an infrared image. Hence, images in this experiment were converted to grayscale.

Another benefit of using grayscaled images is that it requires fewer computational resources. Because two color channels are omitted, the processing time can be reduced by at least a factor of 3. This is extremely helpful when dealing with a large dataset. Figure 5.1 shows the result of converting both a regular and an infrared image to grayscale.

Figure 5.1: (a) A regular image. (b) A grayscaled regular image. (c) An infrared image. (d) A grayscaled infrared image.

5.2 Noise Reduction

As stated in Chapter 2, the selection of noise reduction methods for regular images is well-studied. However, there is not much literature referring to infrared images. Infrared images look completely different from regular images, so directly applying conventional noise reduction methods may not yield an ideal result. Therefore, in this section, some conventional noise reduction methods are evaluated on infrared images, and these methods are outlined as below.

- Average filtering

- Gaussian filtering
- Bilateral filtering
- Median filtering
- Non-local means filtering

5.2.1 Average Filtering

Average or mean filtering is a linear smoothing technique that reduces the intensity variation between neighbouring pixels. Average filtering works by replacing a pixel value with the averaged value in the neighborhood, and it is equivalent to convolving the image space with a kernel. For example, when using a 3×3 kernel that is defined as:

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

the central value under the kernel is replaced by the average in its surrounding 3×3 neighborhood. This process is continued until the last pixel in the image is reached. Average filtering is simple. However, when there exist outliers with extreme values, the consistency and the overall smoothing effect will be significantly affected. Additionally, because the pixel values are averaged, certain information such as the edge is lost and therefore producing blurry edges. Overall, average filtering is not an ideal choice when sharp edges in the output image are required.

5.2.2 Gaussian Filtering

Gaussian filtering is another linear smoothing technique, and it modifies the adjacent pixel intensities around a central pixel by a weighted average based on a Gaussian function, [44]. The Gaussian function is mathematically defined as:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{5.2.1}$$

where σ is the standard deviation of the Gaussian distribution (with 0 mean), and x and y are the pixel coordinates. The σ value controls the extent of the blurring effect, a big σ corresponds to a more blurry effect, and conversely, a small σ correspond to a less blurry effect. Due to the shape of the Gaussian function, the central pixel is always weighted more than those on the periphery. Therefore, the kernel coefficient becomes smaller when it gets further away from the kernel's center. This can be seen in Figure 5.2, when σ becomes larger, the peak is wider and thus generates a greater overall blurring effect.

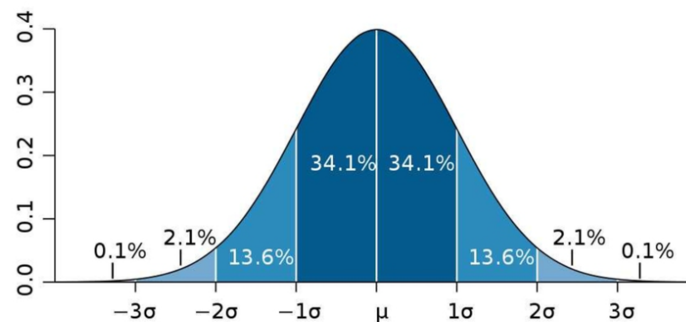

Figure 5.2: The Gaussian distribution.

Below is an example of a 5×5 Gaussian filter when σ equals 1.

$$K_{Gaussian} = \frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

It is worth mentioning that Gaussian filter is separable. This means that the aforementioned 2D Gaussian function can be expressed as the product of two 1D Gaussian function as:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}} = \left(\frac{1}{\sqrt{2\pi\sigma}}e^{-\frac{x^2}{2\sigma^2}}\right)\left(\frac{1}{\sqrt{2\pi\sigma}}e^{-\frac{y^2}{2\sigma^2}}\right) \tag{5.2.2}$$

By separating the Gaussian function, a 2D convolution can be reduced to two 1D convolutions. When using a $m \times m$ Gaussian kernel to perfrom 2D convolution on a $n \times n$ image, the complexity is $O(n^2m^2)$. Whereas, the complexity of 1D convolution is $O(n^2m)$ which is significantly faster. Gaussian filtering is highly effective in removing Gaussian noise, but it does not well preserve the edge.

5.2.3 Bilateral Filtering

Similar to Gaussian filtering, bilateral filtering can also be modeled by a weighted average, but it is a non-linear method. The bilateral filter is defined as:

$$BF[I]_p = \frac{1}{W_p}\sum_{q \in S} G_{\sigma_s}(\|p - q\|)G_{\sigma_r}(I_p - I_q)I_q \tag{5.2.3}$$

where I_p and I_q represent the intensity of pixel p and q, and W_p is a normalizer that

can be calculated using:

$$W_p = \sum_{q \in S} G_{\sigma_s}(\|p - q\|) G_{\sigma_r}(I_p - I_q) \tag{5.2.4}$$

The space Gaussian, G_{σ_s}, is identical to the normal Gaussian filtering, and σ_s controls the extent of blurring within the neighborhood. The range Gaussian, G_{σ_r}, makes sure that only similar pixels will be accounted for during smoothing, and σ_r controls the intensity similarity between pixels. The term “spatial” refers to pixel location, and the term “pixel intensity” describes the quantities related to pixel values, [45]. Modeling pixel intensity by a Gaussian function is an improvement over the normal Gaussian filtering, and it preserves the edges.

5.2.4 Median Filtering

Median filtering is another nonlinear noise reduction method that is particularly effective against impulsive or “salt and pepper” noise [107] while preserving the edges, [46]. This method simply works by first placing a $m \times m$ window centered at a pixel. Then, the intensity of all pixels under the window is sorted in ascending order. Next, the median intensity is calculated. Finally, the centering pixel's intensity is replaced by the median. This process is repeated until all pixel intensities have been replaced. The rationale of its edge preserving property is that the intensity on both sides of the edge are drastically different. When the median is calculated, the intensity of the edge is unchanged.

5.2.5 Non-Local Means Filtering

In addition to the aforementioned filtering methods, the non-local means filtering method proposed by Buades et al. can also be applied, [47]. In [47], the color of a pixel is replaced by the average color of all simialr pixels. Previously, noise reduction methods only consider pixels in a close neighborhood. However, given a pixel, the most similar pixels can also be outside of the close neighborhood. The non-local means filter used in this method is defined as:

$$NLu(p) = \frac{1}{C(p)} \int f(d(B(p), B(q))u(q)dq \tag{5.2.5}$$

where $d(B(p), B(q))$ is the Euclidean distance between the image patches centered at p and q, f is a decreasing function, and $C(p)$ is the normalizer. It is obvious that this method is not truly "non-local" as pixels are still compared locally, but just in a larger neighbourhood. When computing the Euclidean distance, all pixels in the image patch B are equally important, so not only limited to the pixel p, all surrounding pixels in $B(p)$ can also be filtered. This method has been originally applied to color images denoted as $u = (u_1, u_2, u_3)$, and the result can be written as:

$$\hat{u_i}(p) = \frac{1}{C(p)} \sum_{q \in B(p,r)} u_i(q)w(p, q) \tag{5.2.6}$$

$$C(p) = \sum_{q \in B(p,r)} w(p, q) \tag{5.2.7}$$

where $i = 1, 2, 3$ and $B(p, r)$ is the squared neighbourhood centered at pixel p with size $(2r + 1) \times (2r + 1)$. The weight $w(p, q)$ depends on the Euclidean distance

$d^2 = d^2(B(p,f), B(q,f))$ of the two color patches centered at p and q. $w(p,q)$ is defined as:

$$w(p,q) = e^{-\frac{max(d^2 - 2\sigma^2, 0)}{h^2}} \tag{5.2.8}$$

where σ is the standard deviation of the noise, and h is a filtering parameter set depending on the value of σ. When applying this method to grayscaled images, two color channels can be omitted, and the result is then writen as:

$$\hat{u}(p) = \frac{1}{C(p)} \sum_{q \in B(p,r)} u(q)w(p,q) \tag{5.2.9}$$

5.3 Filtering Application Results

Applying the aforementioned filtering methods on an image, the results are in Figure 5.3. In Figure 5.3, the filtered images are on the first row of each method, and the images on the second row show the contour generated by OpenCV. In theory, if the images are well-filtered, the contour should be smooth as it is less affected by noise.

By inspecting the original image on the first row, it is clearly noisy because the corresponding contour is spiky. Moving on to the average filtered image, fewer speckle noises are observed, but the edges become blurry. Although the contour in average filtering is smoother than the original, the remaining noise still makes it spiky. Visually looking at the Gaussian filtered image on the first row, it can be concluded that the image is less blurry than the result from average filtering, and the edges look sharper. However, the contour is barely changed meaning that there is little to no improvement in the filtering result. Comparing the image in bilateral filtering

with the one in Gaussian filtering, the edges produced by bilateral filtering are even more blurry, but bilateral filtering significantly smoothes the contour. The resulting image from median filtering looks less blurry, and sharper edges can be observed, but the contour does not look as smooth as the result from bilateral filtering. Finally, the non-local means filtering seems to generate the sharpest edges and the smoothest contour among all the filtering methods. Overall, bilateral, Median, and non-local means filtering seem to be the most effective options and will be considered in the final implementation.

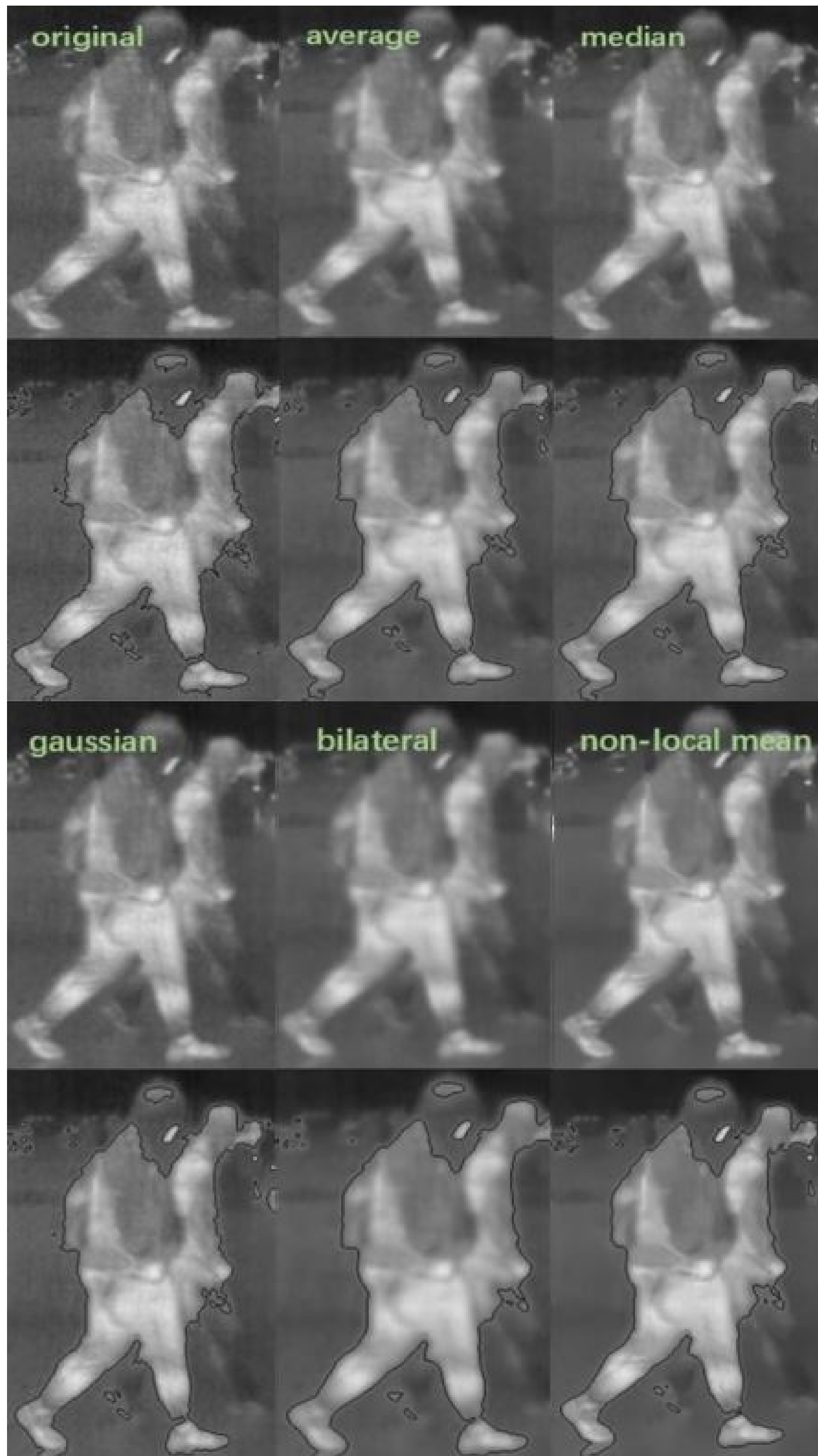

Figure 5.3: The effect of each denoising method.

5.4 Feature Extraction

In infrared images, live objects appear in white, and the background is in black or gray depending on the temperature. Due to the color contrast between the object and its background, edges can be easily localized. Because edge features are representative in infrared images, so they can be used to describe the common characteristics of a class of objects. Therefore, in this thesis, it is decided to use contour-based edge extraction methods to calculate features. Based on the order of differentiation, the following commonly used edge extractors are reviewed and evaluated:

- Sobel edge detector
- Canny edge detector
- Roberts edge detector
- Prewitt edge detector
- Laplacian edge detector
- Laplacian of Gaussian (LOG) edge detector
- Histogram of Oriented Gradients (HOG)

5.4.1 Sobel Edge Detector

Sobel edge detector is one of the most commonly used edge detectors, and it works by calculating the magnitude of the gradient and its direction at each pixel location, [57]. Given a 3×3 image patch centering at $[i, j]$:

$$\begin{matrix} a_0 & a_1 & a_2 \\ a_7 & [i,j] & a_3 \\ a_6 & a_5 & a_4 \end{matrix}$$

the gradient in the horizontal (G_x) and vertical (G_y) direction at $[i, j]$ can be mathematically computed as:

$$G_x = (a_2 + ca_3 + a_4) - (a_0 + ca_7 + a_6) \tag{5.4.1}$$

$$G_y = (a_0 + ca_1 + a_2) - (a_6 + ca_5 + a_4) \tag{5.4.2}$$

In Sobel detector, $c = 2$. The direction is then defined as:

$$\theta = tan^{-}1(\frac{G_y}{G_x}) \tag{5.4.3}$$

Combining the gradient in each direction, the magnitude can be calculated using:

$$M = \sqrt{G_x^2 + G_y^2} \tag{5.4.4}$$

G_x and G_y can also be implemented using 3×3 convolution kernels:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

These kernels are applied in each direction until they fully traversed the entire image. It is worth mentioning that the Sobel detector emphasizes the pixels that are closer to the center of the kernel. The result of applying the Sobel edge detector on an image

is shown in Figure 5.4.

Figure 5.4: Applying the Sobel edge detector.

5.4.2 Canny Edge Detector

The Canny edge detector is a popular edge detection process, [56]. The reason it is called a process is that a pipeline usually needs to be followed in order to extract edges. Unlike the Sobel operator that simply performs convolution on the image to extract edges, the Canny edge detector performs the following operations:

- Grayscale conversion
- Gaussian blur
- Gradient calculation

- Non-Maximum suppression
- Hysteresis thresholding

Because the grayscale conversion and Gaussian blur are identical as it is described in section 5.1 and 5.2.2, so these two steps are omitted here. The gradient in the Canny edge detector is determined by using the Sobel operator as explained in section 5.4, so it will not be explained again here.

Non-Maxima Suppression

Edges can be usually found when there is a rapid change in pixel intensity. This is equivalent to finding the local maxima in the magnitude of gradient. When the magnitude is large, the edges cannot be located as the result is a broad region. To identify the edge, the broad region must be thinned so that only the magnitudes at the local maxima remain. This edge-thinning process is called non-maxima suppression (NMS), [108]. Note that when the word “magnitude” is used in this section, it means “magnitude of gradient”. The non-maxima suppression finds the maximum value on the edge while suppressing other values in the same gradient direction. In this method, the gradient directions $\theta[i, j]$ are divided into four symmetric partitions labeled from 0 to 3 as shown in Figure 5.5.

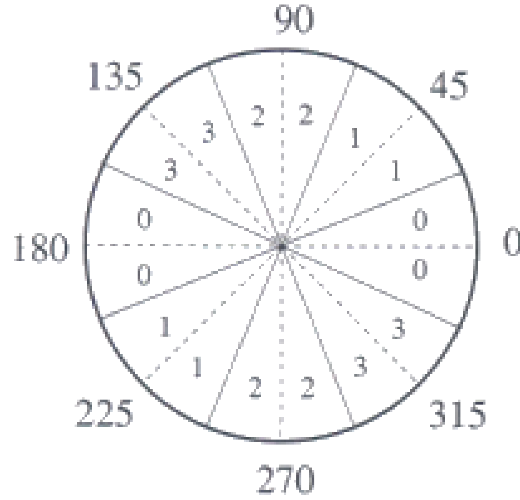

Figure 5.5: The partitions of the reduced gradient direction, [108].

When a 3×3 neighbourhood is centered at a pixel $[i, j]$. The magnitude, $M[i, j]$, is then compared with its neighbours that are in the same graident direction, $\theta[i, j]$. If $M[i, j]$ is not the greatest, it is then set to zero. By repeating the process, the edge will become only one pixel wide. This process can be writen as:

$$N[i, j] = nms(M[i, j], \zeta[i, j]) \tag{5.4.5}$$

where $N[i, j]$ is the suppressed magnitude, and $\zeta[i, j]$ is the partition label between 0 and 3 in Figure 5.5.

Hysteresis Thresholding

After the non-maxima suppression, the hysteresis thresholding is performed on $N(i, j)$ to further reduce the number of false edges. In this process, a lower threshold and an upper threshold denoted as τ_1 and τ_2 are used, with $\tau_2 \approx 2\tau_1$. This technique is also known as "double thresholding", [109]. The thresholded images are denoted as

T_1 and T_2. Because T_2 is produced by using the higher threshold, fewer false edges will be observed. However, sometimes there will be gaps in the contour. In T_2, when it reaches the end of a contour, the algorithm will compare T_2 with T_1. If the contour ends in both T_1 and T_2, the whole process is completed. Otherwise, T_1 will be used to fill in the discontinuity in T_2. This process is continued until all the gaps in T_2 have been bridged. It is worth mentioning that the Canny edge detector performs edge linking as a by-product of thresholding, [56]. Figure 5.6 shows the result of using the Canny edge detector without hysteresis thresholding, and discontinuities in the edge can be clearly visualized. Figure 5.7 demonstrates the effect of edge linking. Notice that there are still some small gays after hysteresis thresholding, but this can be improved by fine-tuning the threshold values.

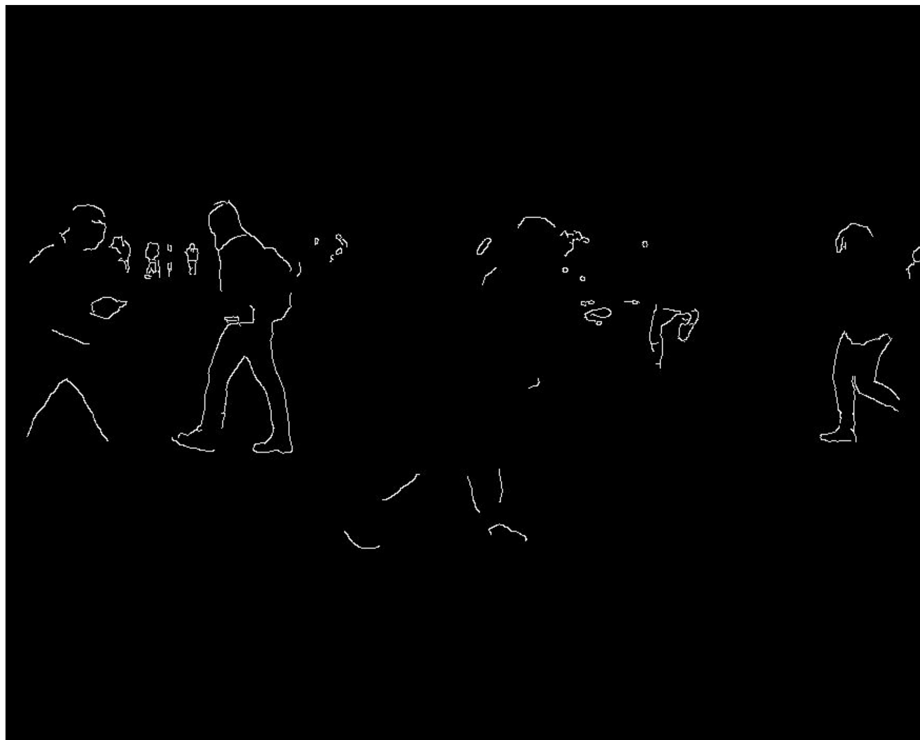

Figure 5.6: Using the Canny edge detector without hysteresis thresholding.

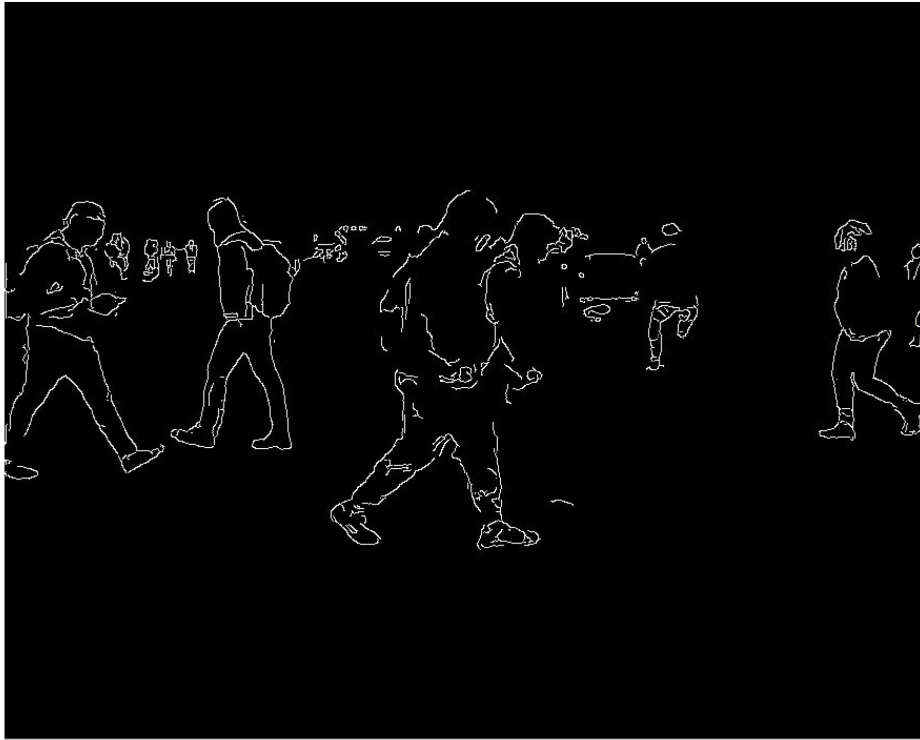

Figure 5.7: Properly applying the Canny edge detector with hysteresis thresholding.

5.4.3 Roberts Edge Detector

Given an image I, the Roberts edge detector [110] is a first-order gradient based method that approximates the gradient magnitude M at pixel (i, j) using:

$$M(i, j) = |I(i, j) - I(i+1, j+1)| + |I(i+1, j) - I(i, j+1)| \tag{5.4.6}$$

Let G_x and G_y be the gradient in the horizontal and vertical direction, the following kernels implement G_x and G_y:

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Equation 5.4.6 is the mathematical formulation when using the L1 norm. The Roberts operator can also be implemented using the L2 or the L-infinity norm. The L1 norm gives stronger edges in the horizontal and vertical direction, the L-infinity norm produces stronger diagonal edges, and the L2 norm accounts for both. However, the L2 norm is computationally expensive, so the L1 norm is commonly used. The Roberts operator generally gives no information about gradient direction, and the result of using the Roberts operator on an image is shown in Figure 5.8.

Figure 5.8: The result of applying the Roberts operator.

5.4.4 Prewitt Edge Detector

Given the equations used by the Sobel edge detector, when $c = 1$, it becomes the Prewitt edge detector, [58]. Therefore, the convolution kernels are:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Unlike the Sobel edge detector, the Prewitt operator does not discriminates the pixels away from the center. The result of applying the Prewitt operator is shown in Figure 5.9.

Figure 5.9: The result of applying the Prewitt operator.

5.4.5 Laplacian Edge Detector

Unlike all aforementioned edge detectors that approximate the first-order derivative, the Laplacian edge detector approximates the second-order derivative, [111]. When

using the first-order derivative, edges appear at local maxima, but when approximating the second-order derivative, they appear at the location where the function crosses zero as shown in Figure 5.10.

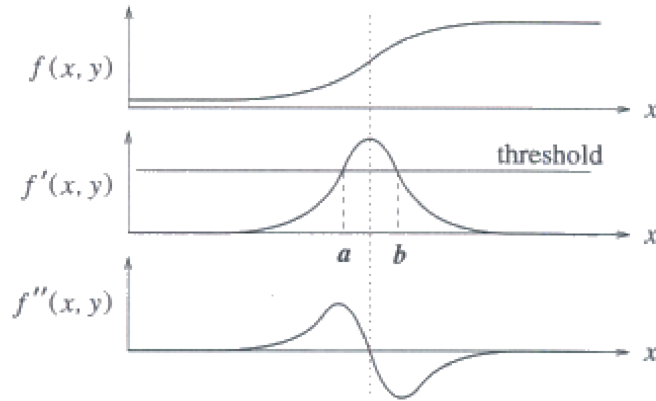

Figure 5.10: The second-order derivative crosses zero at the edge.

The Laplacian function $f(x, y)$ is defined as:

$$\nabla_f^2 = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{5.4.7}$$

Given a pixel at (i, j), the second-order derivative in the x and y directions are approximated using:

$$\frac{\partial^2 f}{\partial x^2} = f(i, j+1) - 2f(i, j) + f(i, j-1) \tag{5.4.8}$$

$$\frac{\partial^2 f}{\partial y^2} = f(i+1, j) - 2f(i, j) + f(i-1, j) \tag{5.4.9}$$

and this is equivalent to using the following convolution kernel:

$$\nabla^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{5.4.10}$$

When the central pixel needs to be emphasized, the Laplacian kernel can be changed to:

$$\nabla^2 = \begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix} \tag{5.4.11}$$

Variations that are rotation-invariant or emphasize on other parts of the image such as the boarder also exist. This section will not provide an exhuastive list of all these variations. On the left side of Figure 5.11 is the result of using the kernel in 5.4.10, and on the right side of Figure 5.11 is the result of using the kernel in 5.4.11.

Figure 5.11: The results of using the Laplacian operator.

5.4.6 Laplacian of Gaussian Edge Detector

When approximating the second-order derivative, the result is extremely sensitive to noise. Therefore, applying the Laplacian operator on a noisy image yields undesirable output. To solve this problem, the LOG detector is proposed that combines Gaussian smoothing with the Laplacian edge detector, [112]. The main idea is to first filter the image using Gaussian filtering as described in section 5.2.2, and then use the Laplacian operator described in section 5.4.5 to detect edges. Mathematically, the LOG operator $h(x, y)$ is defined using convolution as:

$$h(x,y) = \nabla^2[(g(x,y) * f(x,y)] = [\nabla^2 g(x,y)] * f(x,y) \tag{5.4.12}$$

where

$$\nabla^2 g(x,y) = (\frac{x^2+y^2-2\sigma^2}{\sigma^4})e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{5.4.13}$$

The LOG detector can also be implemented using the following 5×5 LOG kernel:

$$h(x,y) = \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

The result of using the LOG detector is shown in Figure 5.12.

Figure 5.12: The result of using the LOG operator.

5.4.7 Histogram of Oriented Gradients Feature Descriptor

The HOG feature descriptor is another gradient-based feature extraction method, [41]. Unlike all aforementioned detectors that identify edges using either the first or the second-order derivative, the HOG uses the distribution of the gradient directions as feature. The distribution is called the "Histogram", and the gradient direction is called the "Oriented Gradients". As proposed in [41], the HOG consists of the following five steps:

- Preprocessing
- Calculate the gradient

- Calculate the histogram of gradients
- Block normalization
- Calculate the HOG feature vector

Preprocessing

In the original paper, [41], the HOG uses 64×128 image patches for human detection. But in fact, a human may appear in any size or scale in an image. For example, when a person is close to the camera, it may appear in 128×256. Similarly, when it is distant from the camera, it may be in 32×64. Therefore, instead of using the HOG solely on 64×128 images, using images that have the aspect ratio of $1:2$ for human detection is more appropriate as the images can be later resized. Additionally, the HOG can also be used to detect other types of objects such as vehicles or animals. Figure 5.13 illustrates the resizing process that is described in the original paper, where a region is first cropped from the entire image and then resized to the aspect ratio of $1:2$.

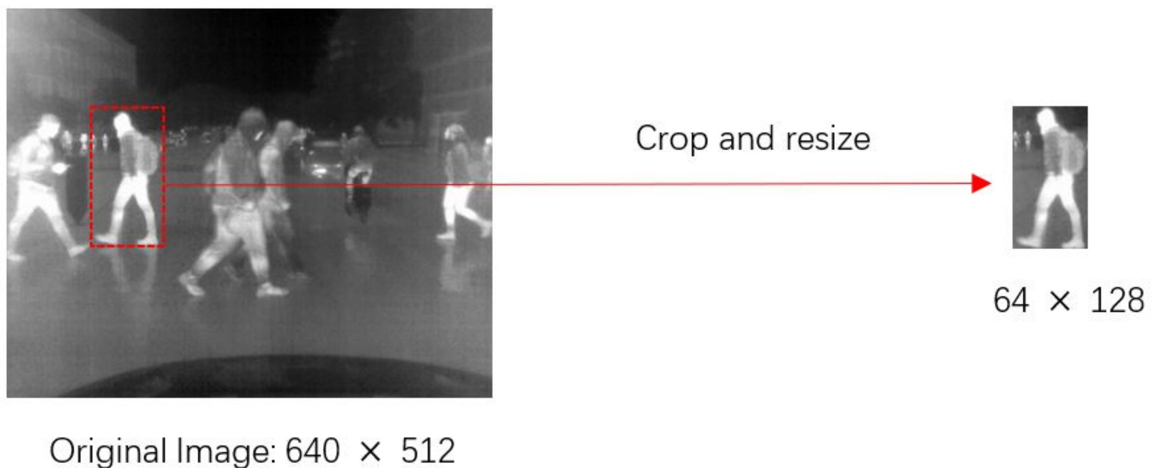

Figure 5.13: Resizing the image patch to 1:2 aspect ratio.

The original paper also mentioned using the gamma or color normalization during pre-processing, but only a modest effect on performance is observed. Therefore, gamma or color normalization is not used in this thesis.

Gradient Calculation

The gradient in the x and y directions can be calculated using the following kernel

$$G_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

Alternatively, the gradient can also be calculated using the Sobel operator as described in section 5.4.

Calculating the Histogram of Gradients

In this step, image patches are divided into blocks of 8×8 cells. For example, in a 64×128 image, there are 8 blocks of 8×8 cells on each row and 16 blocks of 8×8 cells on each column as shown in Figure 5.14.

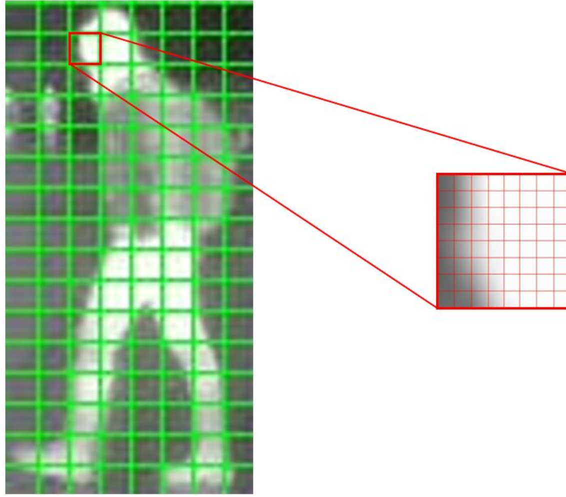

Figure 5.14: The resized image is divided into 8×8 cells. Given a 64×128 image, there are 8 blocks of 8×8 grids on each row and 16 blocks of 8×8 grids on each column.

By using this formulation, each block is represented by $8 \times 8 \times 2$ values. The value 2 indicates that both magnitude and direction of the gradient are calculated for each individual cell. Then, a 9-bin histogram that evenly separates a 180° span is created to store the magnitudes as shown in Figure 5.15.

0°	20°	40°	60°	80°	100°	120°	140°	160°

Figure 5.15: The 9-bin histogram.

By dividing the directions into 9 bins, the first and last bin represent the gradient direction of 0° and 160°, respectively. When a gradient direction falls perfectly in one of the bins, the magnitude is directly added to that bin. Otherwise, when gradient directions fall between two bins, the amount of magnitudes is proportionally added to each bin based on the gradient directions. For example, given a gradient direction of 10°, because it is in between the first and the second bin, so the magnitude is equally divided and added to each of the two bins. Similarly, when a gradient direction of

20° is found, the magnitude is added directly to the second bin.

Block normalization

Gradient magnitudes are the representations of the change in pixel intensities. When the ambient lighting is altered, the calculated gradient will be different. Therefore, gradients are sensitive to the overall lighting, but ideally, gradients should be lighting-invariant to ensure consistency. To achieve this, the histogram is normalized. As suggested in the original paper, 16×16 block normalization using L2-norm is used, [40]. The block normalization slides a 16×16 window across the image, and the L2 norm is calculated at each location. This process continues until the histograms at all locations are normalized.

Calculating the HOG Feature Vector

After the 16×16 block normalization, the result at each location is a 36×1 vector, as each 8×8 cell produces a 9×1 histogram, and concatenating four of these histograms gives a 36×1 vector. The 16×16 window is moved 7 times horizontally and 15 times vertically. Therefore, the final HOG feature vector has a dimension of $7 \times 15 \times 36$. The result of applying the HOG on the same testing image is shown in Figure 5.16.

Figure 5.16: The result of applying the HOG descriptor.

5.5 Method Selection

Given the filtering and feature extraction methods that are described in the previous sections, the performance of these methods on infrared images must be studied and evaluated. However, evaluating every combination is unnecessary as some methods are based upon the others. For example, the canny edge detector uses Gaussian filtering with Sobel operator, the LOG operator combines Laplacian operator will Gaussian filtering, and the Prewitt operator is just setting a different constant value in the Sobel operator. Therefore, only a few combinations will be evaluated in the SVM training, and these combinations are:

- Canny Edge detector (Gaussian filtering and Sobel operator)
- LOG (Gaussian filtering and Laplacian operator)
- Gaussian filtering and HOG
- Median filtering and HOG
- Non-Local Mean filtering and HOG

5.6 Dataset Preparation and Labelling

The dataset that is used to train the SVM is the same as the dataset in Chapter 4.5 except that the labeling format is different. YOLO represents the location of a bounding box by its normalized central coordinates, but OpenCV localizes a bounding box by using the coordinates of the top-left and bottom-right corner. Previously, the dimensions of the input image, the normalized central coordinates, and the dimensions of each bounding box have been defined in YOLO, the coordinates of the top-left and bottom-right corners in OpenCV can be easily calculated by using Equation 5.6.1 to 5.6.4.

$$w = normalized_w \times 640 \tag{5.6.1}$$

$$h = normalized_h \times 512 \tag{5.6.2}$$

$$x = normalized_x \times 640 \pm 0.5 \times w \tag{5.6.3}$$

$$y = normalized_y \times 512 \pm 0.5 \times h \tag{5.6.4}$$

The normalized terms refer to the YOLO coordinates, and 640 and 512 are the width and height of the input image, respectively. The Python script in Appendix. A.1 is written to automate the format conversion. Given the OpenCV coordinates, the dataset can be prepared either by directly extracting the bounding boxes from the full images before training or directly using the full images and later extract the boxes during run-time. Both approaches have been tested but collecting the bounding boxes in the first place makes the dataset visually appealing. One can easily visualize all labeled objects in the dataset (see Figure 5.17).

Figure 5.17: Visualizing the dataset after all bounding boxes have been collected (ignore the file name).

5.7 Decision Making Using Windowing

Because an image contains too much irrelevant information such as the background or the objects that are not of interest, instead of extracting features from the entire

image, a smaller image patch is commonly used to calculate features. For example, given the image in Figure 5.18, to extract features from one of the pedestrians, the patch around that pedestrian is cropped to make sure the background has minimal impact on the pedestrian's feature. Similarly, the SVM generally produces higher accuracy when working with smaller image patches. This process of finding the smaller image patches is commonly called "windowing". Sliding a window across the entire image is similar to the kernel-sliding operations in convolution.

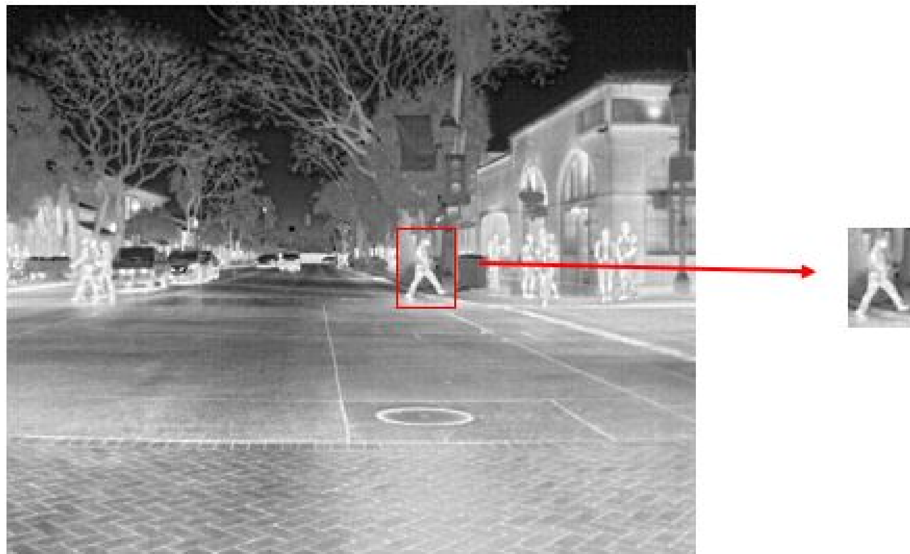

Figure 5.18: Cropping a patch from the original image.

At each window location, features are calculated and a decision is made. In this thesis, when applying the SVM to full images, a 64×128 HOG window is used to slide across the image like it is suggested in the original HOG paper, [41]. The window will then move by the stride amount to the next location. This process is repeated until the window fully traversed the entire image and all decisions are made. It is worth mentioning that there is always a speed-to-accuracy tradeoff, a bigger stride corresponds to a faster processing time but usually produces a relatively low accuracy.

Conversely, it takes longer to process but often yields a higher accuracy. Figure 5.19 is a simple demostration of the windowing process. Note that the window grid is partially outside the image on the edge, this is because the dimensions of the window are not perfectly selected. Adjusting the dimensions of the window or padding the image with zeros will solve this.

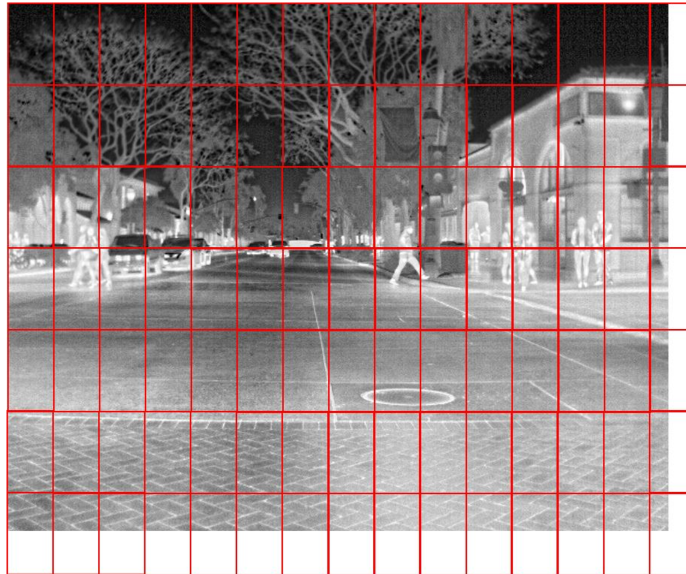

Figure 5.19: The windowing process.

5.7.1 Image Pyramid

After windowing is applied to 640×512 images, images are divided into many 64×128 regions. However, the objects that appear in the window can still be small such as 32×64 or even smaller as shown in Figure 5.20. Note that on the left of Figure 5.20, the object almost occupies the entire 64×128 window, and on the right, the object occupies less than half of the window.

Figure 5.20: When objects appear at different scales in the window..

Without scaling, although the images look similar, the features are in fact completely different from each other, mathematically. Therefore, because the object largely occupies the 64×128 patch, so the SVM will probably correctly classify the left one as "dog" and completely misses the right one. To solve this issue, a process called "Image pyramid" is used. The image is resized to different scales and placed in each layer of the image pyramid as shown in Figure 5.21.

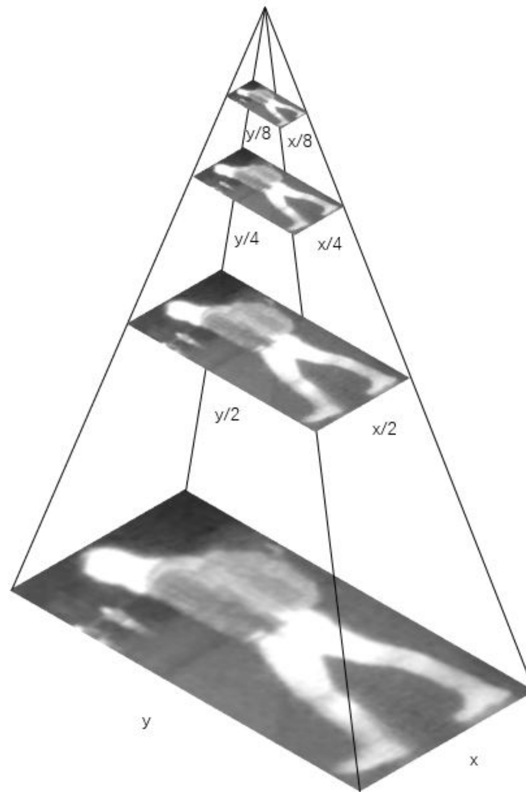

Figure 5.21: The image pyramid.

When applying the image pyramid, the window size remains unchanged, but a window scans the image at four different scales in Figure 5.21. With the help of the image pyramid, a fixed-sized window should have no problem in detecting objects that are bigger or smaller than the window.

5.8 Training a Support Vector Machine

After selecting the filtering and feature extraction methods, three attempts are made during the SVM training. Each attempt involves the following steps:

- Creating a positive and a negative training set

- Preparing negative images for testing
- Training the SVM with hard example mining

In the original HOG paper, features are calculated from 64×128 images. Because other feature extraction methods do not have specified size requirements, the SVMs in this thesis are all trained on the features that are calculated from 64×128 images, all in one scale, to ensure the consistency in the shape and size of the feature, and also for simplicity.

5.8.1 The First Attempt - Training a Unified Linear Binary SVM to Classify Them All

The first attempt is aimed to convert the multi-class classification problem into a single binary classification problem. By using this formulation, when the model detects an object that belongs to one of the four classes, the output is "yes". Otherwise, the output is "no". Having this in mind, the people's feature, vehicle's feature, bicycle's feature, and animal's feature are combined together denoted as x_i with the label "object" denoted as y_i. Then, the SVM is trained on (x_i, y_i) pairs to make binary classification.

Creating a Positive and a Negative Training Set

The positive set contains only the objects that belong to the four classes. For example, images in this set are resized to 64×128 for consistency and contain only people, vehicles, bicycles, and animals. The features from this set are denoted as x_p with the label "+1". An example of the images in the positive set can be seen in Figure 5.22.

Figure 5.22: An image in the positive set.

On the contrary of the positive set, the negative set does not contain any objects that belong to the four classes. Images in the negative set are also 64×128 and contain the background such as a tree, a house, or other objects such as a plane or a boat. The features from this set are denoted as x_n with the label "-1". An example of the images in the negative set can be seen in Figure 5.23.

Figure 5.23: An image in the negative set.

Preparing Negative Images for Testing

The "negative images" and "the images in the negative training set" are two different concepts. Images in the negative training set are "small". For example, they are 64×128. Whereas the negative images used for testing are full-sized 640×512. In this case, 640×512 is the resolution of the FLIR A65 camera. Because both terms share the common keyword "negative", so they do not contain any objects that belong to the four classes. The trained SVM is tested on the negative images, and producing hard examples by "hard example mining". Adding hard examples is equivalent to expanding the size of the negative training set, and can further improve the training result. How to extract hard examples will be explained in the following section. An example of the negative images can be seen in Figure 5.24.

Figure 5.24: An example of the negative images.

Training the SVM with Hard Example Mining

Given the feature and label pairs denoted as $(x_p, +1)$ and $(x_n, -1)$, one linear binary SVM is trained. By using the linear kernel, an assumption made is that all data

points are linearly separable. To validate this assumption, a common approach is to use clustering methods. If data points are linearly separable, the clustering methods will group data points into individual clusters separated by a linear boundary. The K-means clustering method is used in this thesis due to its popularity. Figure 5.25 is a graphical representation of the clustering results, showed in two ways.

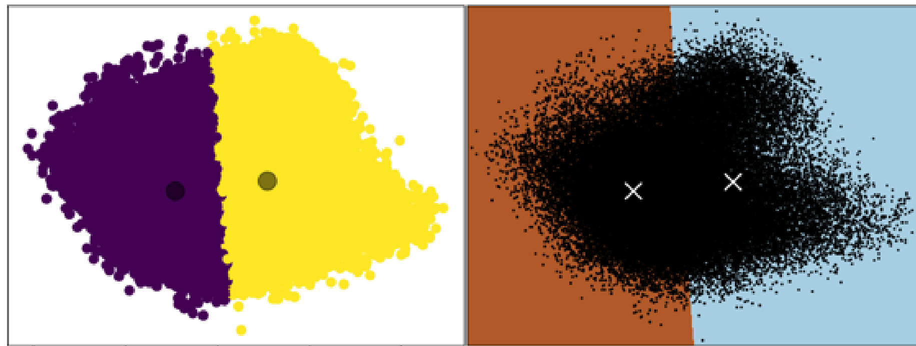

Figure 5.25: The K-means clustering results.

From Figure 5.25, it can be concluded that the training data is in fact linearly separable, and the assumption is valid. On the left, the purple cluster represents the positive class, the yellow class represents the negative class, and the gray dots are the centroid or the data mean. Similarly, on the right-hand side, the boundary between the positive and negative classes can be visualized by the line between the brown and the blue region. Note that because the feature extraction methods produce high-dimensional data, visualize the data in a 2-d plot is impossible. Therefore, Principal Component Analysis (PCA) is used to compress the dimension. PCA reduces the dimensionality by projecting the higher-dimensional data onto the principal components that best approximates the data in the least-square sense, [113].

Starting by grayscaling the input images and using the selected filtering methods for noise suppression, the Sobel, Laplacian, and HOG features are then extracted and

normalized to values between 0 and 1. The hinge loss is used and penalized by L2 regularization. The regularization constant is set to 1 for now.

The extracted features and the corresponding labels are encoded in dense vectors and served as the input to the SVM. Next, the trained model is tested on the negative images to find "hard examples". Because the negative images do not contain any objects that belong to the four classes, if the SVM classifies anything as an "object", it is then a false classification. These false classifications (the images within the bounding box) are called "hard examples" denoted as $(x_{fn}, -1)$ and added to the negative training set. Because the binary SVM can only classify two classes, the hard examples have the same label as the objects in the negative class. In this case, "-1", meaning that it is not an object of interest.

The normalized features are then calculated for the hard examples and appended to the feature vector. Meanwhile, the labels for the hard examples are appended to the label vector. Then, these two vectors are fed back to the SVM to train the model again using $(x_p, +1)$, $(x_n, -1)$, and the newly added $(x_{fn}, -1)$. Hard example mining generates more negative training samples, and one can repeat this process multiple times until the final model achieves a relatively satisfying result. The overall training procedure can be seen in Figure 5.26 via the flowchart, and the Python code for the hard example mining is in Appendix. A.2.

Additionally, when preparing training samples, the dataset can be either balanced or unbalanced. A balanced dataset has an equal amount of positive and training samples. Whereas, the number of positive and negative training samples is not equal in an unbalanced dataset. Both approaches are evaluated on one of the SVMs. Here, for example, the Gaussian and HOG SVM is used, and the results are in Table 5.1.

Because the precision, recall, and the F-1 score between the two approaches only have little to no difference, it is concluded that either approach can be used. Therefore, for simplicity and also for future use, a balanced dataset is used. In total, the positive and negative set each has around 87,269 samples.

Dataset type	Precision	Recall	F1-score
Balanced	93.5%	93.8%	93.6%
Unbalanced	95.2%	95.3%	93.3%

Table 5.1: Training on a balanced and an unbalanced dataset.

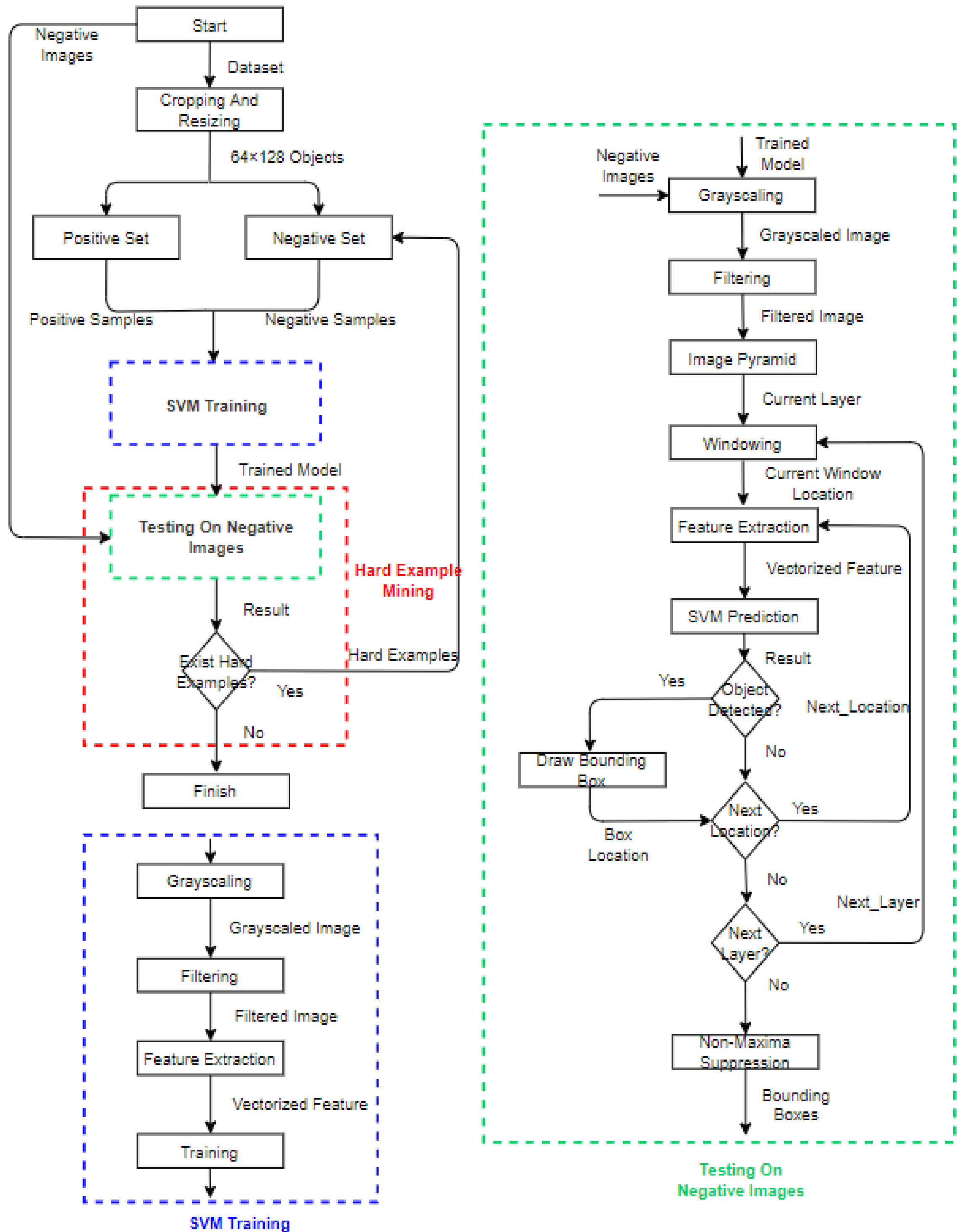

Figure 5.26: The overall SVM training procedure.

Testing the SVM

As previously explained, images in the dataset are all resized to 64×128. 70% of the images are used for training and the rest 30% are used for testing. Note that this $7:3$ data split is commonly used in other literature and experiments, so it is also used in the rest of this thesis. When testing the SVM, the same processing procedure in training carries over, where images in the testing set are grayscaled and filtered, and features are then extracted by using the selected methods. The features are also encoded in vectors and fed to the SVM for prediction. The prediction is simply done by using the "predict" function in the "SVM" package provided by OpenCV. The testing results are in Table 5.2.

Method	Precision	Recall
Canny edge detector	66.2%	58.3%
LOG	61.2%	75.8%
Gaussian filtering and HOG	95.2%	95.3%
Median filtering and HOG	96.5%	95.9%
Bilateral filtering and HOG	94.9%	94.6%
Non-Local Means filtering and HOG	87.9%	86.7%

Table 5.2: The testing results of the unified linear binary SVM.

Based on the results, it is concluded that using only one binary SVM to detect whether an object is presenting in the image seems to be a viable approach. Because the positive training samples cover all classes of objects, the classification problem is simplified. Although this approach shows promising results, usually no image classifier

is trained in this way. Reformulating the multi-class classification problem into a single binary classification problem definitely works, but the classification results are too general which makes it rather meaningless. The detectors in autonomous driving applications usually require more detailed information instead of a simple binary "yes" or "no". Because this attempt produces results that are too general to be used in an actual autonomous driving application, it will only serve as a starting point and will not be considered in the final implementation.

5.8.2 The Second Attempt - Training a Linear Binary SVM for Each Class

The second attempt is an improvement over the previous unified SVM on refining feature selection and reformulates a multi-class classification problem into many sub-binary classification problems by using an OAA-styled approach. Previously when training the unified SVM, because all features are combined, the classification problem becomes over-simplified. Any object of interest triggers a detection, which makes the precision incredibly high. However, the results are binary and provide no information on the class name. Therefore, instead of training only one binary SVM to classify it all, four binary SVMs that correspond to the four classes are trained in this attempt.

This attempt is slightly different from the common OAA approach in the way that, in OAA, the SVM can predict multiple class labels. But, the SVMs used in this attempt are still binary. For example, the SVM used for classifying animals will return the output "yes" when an animal is detected. Otherwise, the result is "no". By using this approach, each SVM is only responsible for classifying one class of objects.

Training each SVM still follows the same procedure except the dataset is split

into four sub-datasets that contain only people, vehicles, bicycles, and animals, respectively. By splitting the dataset, the positive training samples for each SVM become only the objects of interest of that class. The negative training samples combine the previously used common negative samples and all positive samples from the other three classes. For example, the positive training samples of the "people" SVM are just images of people, but the negative training samples include vehicles, bicycles, animals, and the previously used common negative samples. By doing this, each dataset is no longer balanced since the number of negative samples well-exceeded the number of positive samples, but the data points are still linearly separable.

As usual, the selected filtering methods are first applied, and the features are extracted from the positive and negative samples and fed into each SVM. Four SVMs are trained simultaneously. All the hyperparameters remain unchanged. Note that because each SVM still performs binary classification, the hard examples produced by each SVM still have "-1" as the label, which means they do not belong to that class.

Testing the SVM

Table 5.3 and 5.4 show the results of this attempt. Because there is a huge difference in the number of training samples between classes, each SVM exhibits drastically different results. For example, the vehicle class has the most training samples, so the "vehicle" SVM is the overall most accurate one. Because only hundreds of samples are available for the animal class, the "animal" SVM is the overall worst performing one.

Looking at the last four rows in both Table 5.3 and 5.4, the filtering methods can be

compared at the same level. Based on precision, it can be concluded that Gaussian, median, and bilateral filtering have comparable results, and the non-local means filtering is around 2% below the previous methods. Similarly, looking at the first three rows, the feature extraction methods can be compared at the same level because they all use Gaussian filtering. Overall, the HOG significantly outperforms other feature extraction methods by 10-15% in terms of precision. This conclusion can also be validated from the recall perspective. Because better-performing methods produce fewer misclassifications or false positives, the recall is higher. Poorly performing methods generate more false negatives and therefore the recall is lower.

This attempt reformulates a multi-class classification problem into four binary classification problems and has proved that assigning one SVM to each class is a better approach. Moreover, because bilateral filtering with HOG features achieves the highest overall performance, this combination will be considered in the final model.

Method (precision)	people	vehicle	bicycle	animal
Canny edge detector	64.9%	57.9%	54.7%	52.2%
LOG	67.1%	69.3%	63.4%	57.5%
Gaussian filtering and HOG	82.3%	80.6%	77.9%	74.3%
Median filtering and HOG	81.5%	81.7%	79.4%	72.9%
Bilateral filtering and HOG	82.7%	80.5%	78.4%	74.5%
Non-Local Means filtering and HOG	81.3%	78.1%	75.2%	73.8%

Table 5.3: The precision of each linear binary SVM in the second attempt.

Method (recall)	people	vehicle	bicycle	animal
Canny edge detector	60.9%	63.1%	62.7%	60.3%
LOG	77.9%	61.9%	59.5%	57.8%
Gaussian filtering and HOG	84.6%	79.2%	77.9%	74.3%
Median filtering and HOG	82.6%	81.4%	79.4%	78.5%
Bilateral filtering and HOG	83.6%	80.6%	82.4%	79.7%
Non-Local Means filtering and HOG	81.4%	78.6%	77.2%	76.4%

Table 5.4: The recall of each linear binary SVM in the second attempt.

5.8.3 The Third Attempt - Adjusting the Window Size During Training

In the previous two attempts, all training images are resized to 64×128. This is desirable for the people class as the aspect ratios in this class are usually $1:2$. In other words, the height of the bounding boxes around people is usually twice as big as the width. However, by inspecting the dataset used in this experiment, most bounding boxes around vehicles, bicycles, and animals have an aspect ratio of $2:1$. Therefore, similar to the second attempt, four datasets are prepared, but the images in the vehicle, bicycle, and animal set are resized to 128×64 rather than 64×128 to represent the $2:1$ ratio. The positive and negative training samples are identical to those in the second attempt except the aspect ratio has been adjusted correspondingly. The rest of this attempt including the training and testing procedures remains unchanged from the second attempt.

Testing the SVM

According to the results in Table 5.5 and Table 5.6, because a 128×64 window fits vehicles, bicycles, and animals better, it is more likely to detect these objects during windowing, and therefore exhibits a 1-2% improvement in precision. The recall is also slightly improved. Additionally, when using a 64×128 window, as the amount of space surrounding these objects is usually bigger compared to when using a 128×64 window, the background has more impact on the detection results which yields slightly lower performance. Therefore, it is concluded that adjusting the aspect ratio of the training samples based on their natural shape during training is helpful, but it only offers little improvement. Because the improvement is so little, this attempt will not be considered in the final implementation.

Method (precision)	people	vehicle	bicycle	animal
Canny edge detector	64.7%	59.6%	56.2%	51.9%
LOG	67.3%	72.7%	64.6%	57.6%
Gaussian filtering and HOG	82.5%	81.5%	79.3%	75.1%
Median filtering and HOG	81.9%	83.2%	79.9%	72.7%
Bilateral filtering and HOG	83.1%	82.2%	80.1%	74.9%
Non-Local Means filtering and HOG	80.9%	79.9%	77.0%	73.9%

Table 5.5: The precision of each linear binary SVM in the third attempt.

Method (recall)	people	vehicle	bicycle	animal
Canny edge detector	60.6%	64.4%	64.1%	59.8%
LOG	78.2%	63.7%	61.3%	58.0%
Gaussian filtering and HOG	82.5%	80.6%	79.8%	75.4%
Median filtering and HOG	84.1%	83.4%	82.1%	78.2%
Bilateral filtering and HOG	83.7%	81.7%	83.2%	79.8%
Non-Local Means filtering and HOG	80.5%	79.6%	78.7%	76.4%

Table 5.6: The recall of each linear binary SVM in the third attempt.

5.9 Running the SVM on Full Images and Comparing with the YOLOv4

Based on the results gathered from the three different attempts, it is finally decided to use bilateral filtering paired with HOG features to train the SVM because this method combination has the highest overall performance from previous experiments. The "bilateral and HOG" SVM has been chosen to compare with the YOLOv4 networks.

Because the training and testing images in the dataset are originally resized to 64×128, testing the SVM and evaluating its performance on these "small" images can be directly done without using windowing and image pyramid. However, due to the YOLOv4 networks can directly perform detection on 64×128 images, in order to compare the SVM with the YOLOv4 at the same level, in order words, to run the SVM on full-sized 64×128 images, windowing and image pyramid is required to scan the full image. In this thesis, the window size is decided to be 64×128, and the stride

is set to 64. Therefore, the image in the first layer of the pyramid is divided into a 10×4 grid, and HOG features can be extracted at each location without any major modification to the code.

The scaling factor used by the image pyramid is set to 1.5 on a trial-and-error basis. By testing the image pyramid using different scaling factors, 1.5 produces the best overall results. The dimensions of the 640×512 images are divided by 1.5 each time until the 64×128 window no longer fits inside the image. Because the images captured by the FLIR A65 camera in this experiment are all 640×512, by using this setting, the 64×128 window scans each image 5 times. In other words, the image pyramid has 5 layers.

When sliding the window in each layer of the pyramid, the part of the image within the window is grayscaled first and filtered by bilateral filtering. HOG features are then extracted and fed to the SVMs for prediction simply by calling the "predict" function. This process is the same as previously explained. Because there are four SVMs used in the final model, each SVM makes the corresponding binary classification simultaneously.

If an object is detected at the current window location, a bounding box is produced at that location by using OpenCV's "rectangle" function. When an object is not perfectly placed in a bounding box, in other words, if multiple bounding boxes enclose the same object, the non-maxima suppression is applied. This is done by setting the overlapping threshold to 0.3, meaning that when two bounding boxes overlapped each other by greater than or equal to 30%, the current bounding box is deleted. A list is used to keep track of the indices of all bounding boxes. This threshold is also set based on trial-and-error. The detection results before and after the non-maxima

suppression are in Figure 5.27 and Figure 5.28, and the flowchart that depicts the entire pipeline when running the SVM on full-sized 640×512 images is in Figure 5.29.

Figure 5.27: Before the Non-Max suppression.

Figure 5.28: After the Non-Max suppression.

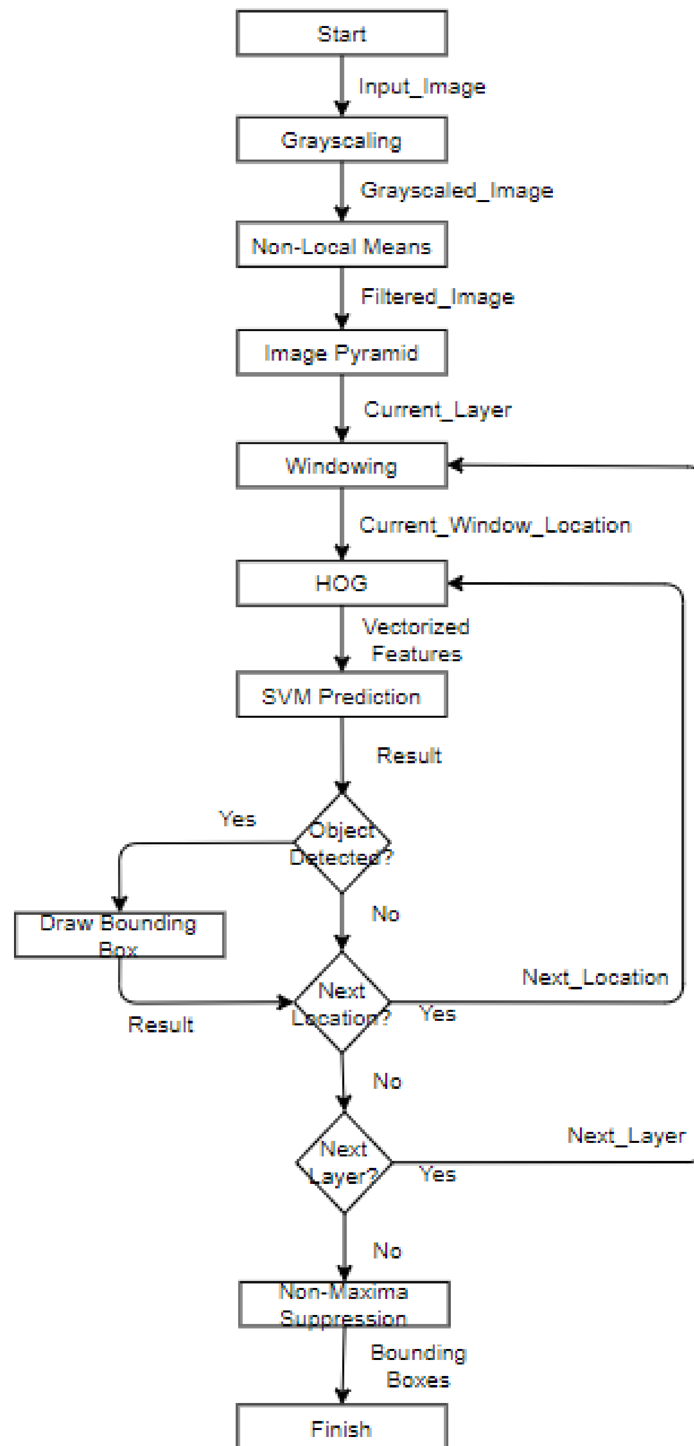

Figure 5.29: Running the SVM on full-sized 640×512 images.

Comparing the SVM model with the YOLOv4 and tiny YOLOv4 networks, the results are in Table 5.7. Note that this comparison is made on 15% of the 640×512 testing images that are originally used in the YOLO training.

Method AP @ 0.5 IoU	people	vehicle	bicycle	animal	mAP	Time(ms)
SVM	79.7%	76.1%	73.9%	70.2%	75.0%	345.358
Full YOLOv4 @ 608	68.9%	69.3%	66.0%	62.8%	66.8%	28.781
Tiny YOLOv4 @ 608	60.8%	68.4%	57.7%	55.3%	61.0%	5.294

Table 5.7: Comparing the SVM with the YOLOv4 and tiny YOLOv4 network.

From Table 5.7, the "bilateral and HOG" SVM achieves the highest 75.0% overall mean average precision, the full 608×608 YOLOv4 network placed second with 66.8% mAP, and the 608×608 tiny YOLOv4 network placed last with only 61.0% mAP. On average, each image takes the YOLOv4, tiny YOLOv4, and the SVM 28.781ms, 5.294ms, and 336.288ms to process, respectively. Converting to FPS, the tiny YOLOv4 is the fastest and can process up to 189 FPS whereas the SVM is the slowest and can only process at around 3 FPS. The longer processing time in the SVM can be explained by the use of filtering, windowing, and image pyramid.

In the current setup, the SVM applies bilateral filtering and scans the image 5 times by using the image pyramid, whereas the full YOLOv4 and tiny YOLOv4 do not employ any filtering and only scan the image once. Because the tiny YOLOv4 network utilizes only a part of the convolutional layers, its processing time is further reduced. However, as features are relatively coarse in the earlier convolutional layers, the tiny YOLOv4 network performs poorly on detecting small objects such as dogs.

This can be seen from the reduction in the mean average precision between the full YOLOv4 and tiny YOLOv4 networks. Conversely, as the features become finer further down the convolutional layers, the full YOLOv4 network has no issues detecting small objects.

Transfer learning is another key factor that can potentially improve performance, but this is not the case in this thesis. The pre-trained model used by the YOLOv4 is trained on the ImageNet, but because the images in the ImageNet are all regular images rather than infrared images, the transferred features do not help much. This can also be explained from a different perspective, as the pre-trained model is used to detect objects in regular images, it generally does not perform well in infrared applications. However, it is still believed that the pre-trained regular features can in fact help to train an infrared network, but how much it can affect the overall performance will not be investigated in this thesis. If a network that has been previously trained for infrared applications can be used, it is believed that the overall performance of the YOLOv4 networks can be further improved.

Moreover, the YOLOv4 networks use data augmentation which provides them with more training samples. If data augmentation is used in the SVM, the overall performance of the SVM might be even higher as more training samples are always beneficial. Using data augmentation can potentially close the precision gap between classes.

Finally, the SVM is implemented in Python and does not utilize parallel computing. If the windowing and image pyramid process are parallelized, it is believed that the processing time of the SVM can be further reduced. Using fewer layers in the image pyramid and a bigger stride value can also speed up the SVM's detection

time. However, because there is always a speed-to-accuracy tradeoff, a faster speed will eventually lead to a reduction in precision. Therefore, improving detection speed will not be considered at the current stage. Overall, the training process of the SVM needs to be improved, and the final model requires more fine-tuning. Besides, a more carefully labeled dataset is also preferable.

Chapter 6

Conclusion and Future Work

This thesis has proved that using machine learning and deep learning-based algorithms on an infrared camera to do onboard vehicle-based computer vision tasks shows promising results. However, in order to validate whether the additional information supplied by infrared cameras can further improve the overall performance of the ADAS, and if infrared cameras should become an essential part of the ADAS for improved safety, more work is required. A conclusion that summarizes everything that has been learned throughout the writing of this thesis following a brief explanation of some potential areas for future research will be discussed in this chapter.

6.1 Conclusion

Autonomous driving is no doubt one of the most rapidly growing and eye-catching fields in today's technology sector. With the increasing popularity of electric vehicles and autopilot systems, sensory technologies have been thoroughly studied in recent

years. The existing perception systems or the advanced driver-assist systems developed by many automakers are capable of providing enough information around the vehicle that can facilitate driving autonomously in various conditions, but even with a wide variety of sensor combinations, not every system works perfectly, 100% of the time.

Typical perception systems consist of regular daylight cameras, radars, LIDARs, GPS, and ultrasonic sensors. Regular cameras are efficient at working under daylight but usually fail to detect objects at night. Radar waves are good at penetrating substances such as fog and dust, but radars usually struggle to detect small objects. LiDAR sensors are powerful tools to provide reliable distance measurements under most circumstances, but due to light refraction, LiDAR sensors can generate faulty data on rainy days. GPSs have been widely used for positioning, navigation, and timing, but most of their functionality relies on satellites. Finally, ultrasonic sensors are only good at working in close proximity. There are not many perception systems that utilize thermal imagings to facilitate autonomous driving, but using the additional information provided by infrared cameras has been proved helpful.

This thesis has explored the background of autonomous driving, the current trend of autonomous driving, sensors and their capabilities, image processing techniques, camera calibration methods, and machine learning and deep learning-based detection and classification algorithms. The goal of this thesis is to develop a thermal imaging vehicle perception framework using both machine learning and deep learning approaches and compare the performances, and ultimately, validate using infrared cameras to aid the existing ADAS is a viable approach for improved safety.

The camera used in this thesis is a FLIR A65 thermal imaging camera with a

640 × 512 resolution. When calibrating the FLIR A65 camera, some popular camera calibration methods have been examined. After studying the limitations of each of these methods, based on these methods, an improved method has been proposed that is aimed to solve the difficulty of calibrating infrared cameras in cold temperatures. After calibrating the infrared camera using the improved method, a dataset that contains people, vehicles, bicycles, and animals is prepared and labeled. The calibration results, namely the extrinsic and intrinsic matrix can be used for distance measurements in the future.

The discussion of object detection and classification leads to an investigation of traditional machine learning and modern deep learning algorithms. For the deep learning approach, it is decided to use one of the fastest state-of-the-art convolutional neural network detectors namely, the You Only Look Once (YOLOv4). Considering the full YOLOv4 network may not be able to perform real-time detection on some platforms, a faster version of the YOLOv4, the tiny YOLOv4 has also been trained. Because altering the size and depth of a convolutional neural network can significantly affect its performance, so the performance of the YOLOv4 network has been evaluated at 320×320, 416×416, and 608×608 using 53 and 29 convolutional layers, respectively.

Running on an Nvidia GTX 1080Ti GPU, ranking the network from the smallest to the biggest, the full YOLOv4 achieves around 69.1%, 67.8%, and 70.0% mean average precision while running at 61.59 FPS, 47.64 FPS, and 34.75 FPS, respectively. The tiny YOLOv4 gets 59.3%, 64.6%, and 63.6% mean average precision at 404.37 FPS, 249.56 FPS, and 188.89 FPS, respectively. Usually, more than 30 FPS can be considered real-time, so all YOLOv4 networks are qualified to perform real-time

detection. Moreover, the 608×608 networks have the highest confidence score. Although the 608×608 tiny YOLOv4 network does not have the highest precision, it has the highest confidence score. As detection speed is not a concern, the network with the highest mean average precision and confidence score from both the full and tiny categories, namely the 608×608 full YOLOv4 and 608×608 tiny YOLOv4 have been selected and compared. Both the full and tiny YOLOv4 networks are trained using transfer learning on a pre-trained network, but how much transfer learning affects the overall accuracy is not studied in this thesis.

When using the machine learning approach, it is decided to train a support vector machine. Because machine learning classifiers offer more flexibility in method selection and parameters tunning, the goal here is to see how a flexible model compares to the fix-structured convolutional neural network. In order to effectively train a machine learning model, one usually needs to follow the vision pipeline where images are first preprocessed, then extract features, next train the classifier, and finally, use the classifier to make decisions.

Following the vision pipeline, three attempts have been made. In the first attempt, a single binary SVM is built to classify everything. However, the SVM model in this attempt over-simplifies the problem which makes the results too general and rather meaningless. The results are simply binary “yes” or “no”, and no class label is given. To produce class label predictions, four binary SVMs are built in the second attempt that corresponds to the four classes. The second attempt employs a One-Against-All styled approach and reformulates a multi-class classification problem into many sub-binary problems which demonstrate promising results. The third attempt alters the aspect ratio of some training samples, but only minimal improvements are observed.

Therefore, based on these results, it is decided to use the second attempt with four individual linear binary SVMs in the final model.

In each of the attempts, training samples are cropped from the original dataset and resized to a unified size to ensure consistency in the shape and size of the features. Upon investigating the image preprocessing methods and feature extraction techniques, six popular method combinations are used to prepare the features including the Canny edge detector, Laplacian of Gaussian, Gaussian filtering and HOG, Median filtering and HOG, bilateral filtering and HOG, and non-local means filtering and HOG. By using these combinations, the Sobel, Laplacian, and HOG features can be compared at the same level. Similarly, the Gaussian, median, bilateral, and non-local means filtering can be compared at the same level.

During the SVM training, positive and negative features are calculated using the aforementioned method combinations. A process called hard example mining is used to find false classifications. These false classifications are called the hard examples and added back to the negative training set to train the SVM a second time. In the end, the “bilateral filtering and HOG” SVM achieves the highest 79.0% mean average precision, and therefore it is selected as the final model and used to compare with the YOLOv4 networks. Running the “bilateral filtering and HOG” SVM on full-sized 640×512 images with windowing and image pyramid achieves around 75.0% mean average precision, but it is only able to run at 3 FPS.

Overall, the SVM is more accurate, but the YOLOv4 networks are faster. Because the YOLOv4 pursues a faster speed, it only looks at the image once and does not employ any filtering methods. However, experiments have shown that additional filtering methods only have minimal impact on the processing time. Therefore, adding

filtering methods to the YOLOv4 network might improve the overall accuracy. The SVM on the other hand uses an image pyramid and scans the same image multiple times at different scales which makes it operating at a significantly slower speed, but with higher accuracy. If the windowing and image pyramid process in the SVM can be parallelized, its processing time can be further reduced.

There is definitely a lot of space to improve in both approaches, but because there will always be a speed-to-accuracy tradeoff, a faster speed will ultimately lead to a reduction in accuracy. Because the SVM models in this thesis are accuracy-oriented, improving speed is not considered at the current stage. However, If accuracy can no longer be improved, then improving speed is the next thing to be considered.

6.2 Future Work

This section suggests several ways that can potentially improve the performance of the SVM and YOLOv4 networks based on the observations made while completing this thesis. Anything that was considered previously but got discarded in the final implementation will also be discussed.

6.2.1 Expanding the Current Dataset and Refining Labeling Quality

The SVM and YOLOv4 algorithms require carefully labeled images for training. Although combining the self-prepared dataset and the FLIR's official driving dataset prominently maximizes data variety, the bicycle and animal class only cover a limited amount of training samples. The training images should theoretically cover the object

posing at different angles, and thus a new dataset should be introduced to resolve the severe lack of training samples. Investigating and leveraging other publicly available infrared datasets can also fulfill this need. The new dataset should provide thousands of more images under the bicycle and animal class including multiple angles of the bicycle and the front and side views of the animal while it's standing, sitting, crouching, and lying down.

Moreover, because infrared cameras excel at detecting heat, justifying whether infrared cameras are capable of detecting objects at night or under inclement conditions, using images that are captured only in sunny conditions is not enough. Therefore, night-time capturing and foggy, dusty, and rainy conditions should also be considered and added to the dataset.

Labeling quality is critically important when building classifiers. An inappropriately labeled dataset directly leads to poor performance. A common approach that ensures the labeling quality is to have multiple (at least three) labelers label the same image. The results can be later compared and checked by picking the box with the highest IoU with respect to the ground truth. When there are more than ten objects or the objects are tightly overlapped with each other in the frames, even with multiple labelers in place, the quantity of coarsely drawn boxes can increase substantially, and therefore the overall labeling quality still can not be guaranteed. To solve this, the labeling can be outsourced to labeling service providers such as Amazon's Mechanical Turk and Scale.ai. The labeling service offered by Amazon still uses human labelers which inevitably generates errors, whereas Scale.ai provides a labeling service that utilizes a machine learning-based method to check its labeling quality.

Having a professionally labeled dataset in place, the labeling quality will not

become a major concern when the trained model exhibits unexpected behaviors.

6.2.2 Improving the YOLOv4 and Tiny YOLOv4

The major issues found in the YOLOv4 and tiny YOLOv4 training are the number of training samples and labeling quality. Combining the FLIR driving dataset and a custom dataset, and using a pre-trained model paired with transfer learning, the YOLOv4 and tiny YOLOv4 networks work just fine when detecting and classifying people and vehicles. However, the final dataset is still rather small for detecting the rest of the classes.

The original authors suggested a minimum of 2,000 training samples per class, so at least 3,000 samples (2,000 (70%) for training and 1,000 (30%) for validation and testing) should be prepared for each class. But apparently, there are not enough samples for the dog and bicycle class. Although 2,000 samples are the minimum requirement, based on the findings collected throughout the experiment, it is believed that a good-performing network requires far more than 2,000 training samples.

Besides the number of training samples, labeling quality is another major concern. The official FLIR driving dataset occupies two-third of the dataset, and the rest one-third is the custom dataset. Assuming all the images in the FLIR's dataset are professionally labeled, the discrepancy in labeling quality between the two datasets can significantly affect the performance. One potential solution to this is to first train the network on the FLIR driving dataset and then use transfer learning again to train on the custom model. Otherwise, relabeling the entire dataset from scratch will also work but is time-consuming.

Another area that can potentially improve YOLOv4's accuracy is preprocessing.

Previously, experiments on how filtering methods affected the overall processing time are conducted. Applying fast filtering methods such as Gaussian filtering may also improve the overall accuracy while maintaining YOLO's real-time property. Therefore, filtering methods will be tested in the future.

Last but not least, how transfer learning affects the overall accuracy is not evaluated in this thesis. Because the pre-trained network is trained on the ImageNet that mainly uses regular images, how regular features help to train a network that is designed to work on infrared images cannot be determined at the current stage. Therefore, in the future, the YOLOv4 networks will also be trained from scratch and compared to the ones that use transfer learning.

6.2.3 Improving the SVM

Several improvements can be made in the SVM training starting with normalization. Pixel values are originally normalized to floating-point values between 0 and 1. This is done by dividing each value by the largest pixel value (255), in other words, pixel values are "hard" normalized. However, a "soft" normalization might be more appropriate where the mean value is subtracted from each value and then divided by the standard deviation. Similarly, feature vectors can also be soft-normalized before feeding to the SVM.

Features are currently extracted in a sequential manner, so the program waits for the current windowing process to finish until it moves to the next position and will not move to the next layer of the pyramid until it finishes processing the current layer. Ideally, The feature extraction pipeline in the windowing and image pyramid should be parallelized. By using multiple threads to calculate the features from a 5-layered

pyramid simultaneously, the processing time can be reduced to at least one-fifth of the original value. Furthermore, if the windowing locations are also computed in parallel, the processing time can be further reduced.

The final SVM model in this thesis employes an OAA-styled approach, so four SVMs are used to match the number of classes. However, a true OAA multi-class SVM only uses one SVM to classify multiple class labels simultaneously. Therefore, in the future, four binary SVMs will be reduced to a single multi-class OAA SVM.

In terms of parameter tuning, because the data are linearly separable in this thesis, so a linear kernel is used for simplicity. However, a non-linear kernel such as the polynomial and Gaussian RBF kernel should also be considered when data becomes linearly inseparable. Additionally, setting different regularization parameters may also improve the training results. Because the regularization parameter alters the size of the margin, so it controls the amount of misclassification the model produces. The regularization parameter is set to 1 in this thesis, but in the future, an optimal value will be used to improve the result.

Another major drawback of the trained SVM is that it did not use data augmentation. Because there are only a limited amount of training samples available for the bicycle and animal class, data augmentation can be extremely useful and substantially improve the accuracy, specifically for the bicycle and animal class.

To improve the robustness of the SVM, multiple SVM classifiers can be trained and apply ensemble methods such as bagging and boosting to combine the results. Bagging averages the predictions, and boosting combines several weaker models to produce a stronger model.

In terms of labeling quality, because training samples are cropped and resized

from the original images, the problem with the poor labeling quality in the YOLOv4 carries over to the SVM training. Improving the labeling quality by hiring multiple labelers or outsourcing can alleviate this problem.

Finally, the SVM can potentially run at a faster speed by using a larger window, bigger stride, and fewer layers in the pyramid. When the precision can no longer be improved, the balance between speed and accuracy needs to be considered.

6.3 Closing Remark

In this thesis, using thermal imaging cameras to do onboard vehicle perception tasks such as object detection and classification shows promising results. Regardless of the cost, infrared cameras can provide invaluable information to improve the robustness of the ADAS on autonomous vehicles, especially when fusing infrared detections with detections made by regular daylight camera, radar, LIDAR, ultrasonic sensor, and GPS sub-systems. Concluding from this thesis, infrared cameras are useful in autonomous driving, but can never be used as a standalone sensor. However, its ability to complement other sensors and improve the overall performance of the existing sensory systems is undeniable. Therefore, infrared cameras should definitely be used more often in sensor fusion applications and considered becoming an essential part of the modern ADAS for improved safety.

Appendix A

Python Code

```
14  '''
15  As YOLO uses a special format to represent a bounding box's location, it is impossible to directly
16  use it in openCV. Therefore, this piece of code converts the YOLO labelling format to normal openCV
17  format and saves each object to a folder that stores only that type of objects.
18  '''
19  for entries in os.listdir(path):
20      '''
21      YOLO assigns each image with a .txt file that contains the corresponding labelling information.
22      An example of the labelling information is: 3 0.476953 0.543750 0.077344 0.170833.
23      3 represents the class label, and the rest represent the x,y,w,h.
24      '''
25      #Searching for all the labelling information
26      if entries.endswith(".txt"):
27          #Save the file name for later use
28          name = entries.split(".")[0]
29          jpg_name = name +".jpg"
30
31          #if multiple objects present in the same image, get all the labelling information
32          f = open(entries, "r")
33          lines = f.readlines()
34          count = len(lines)
35
36          #Open the original image to extract patch within each bounding box
37          img = cv2.imread(os.path.join(path, jpg_name))
38
39          #Converting the YOLO format to OpenCV format
40          for i in range(count):
41                  w = int(float(lines[i].split(" ")[3])*640)
42                  h = int(float(lines[i].split(" ")[4])*512)
43                  x = int(float(lines[i].split(" ")[1])*640 - 0.5*w)
44                  y = int(float(lines[i].split(" ")[2])*512 - 0.5*h)
45
46                  '''
47                  #Saving each object to the corresponding folder.
48                  0, 1, 2, 3 represent the people, bicycle, vehicle, and dog class, respectively.
49                  '''
50                  #Save images of people to the people folder
51                  if float(lines[i].split(" ")[0]) == 0:
52                              #Extracting the image patch within the bounding box
53                              new_img = img[y:y+h,x:x+w]
54                              #Some file naming convention
55                              cv2.imwrite(os.path.join(people_path,str(counter)+".jpg"), new_img)
56                              counter +=1
57
58                  #Save images of bicycle to the bicycle folder
59                  elif float(lines[i].split(" ")[0]) == 1:
60                                  new_img = img[y:y+h,x:x+w]
61                                  cv2.imwrite(os.path.join(bicycle_path, str(counter)+".jpg"), new_img)
62                                  counter +=1
63
64                  #Save images of vehicles to the vehicle folder
65                  elif float(lines[i].split(" ")[0]) == 2:
66                                  new_img = img[y:y+h,x:x+w]
67                                  #cv2.imshow("2", new_img)
68                                  cv2.imwrite(os.path.join(car_path, str(counter)+".jpg"), new_img)
69                                  counter +=1
70
71                  #Save images of dog to the dog folder
72                  elif float(lines[i].split(" ")[0]) == 3:
73                                  new_img = img[y:y+h,x:x+w]
74                                  #cv2.imshow("3", new_img)
75                                  cv2.imwrite(os.path.join(dog_path, str(counter)+".jpg"), new_img)
76                                  counter +=1
77          f.close()
```

Figure A.1: The code snippet for the YOLO-OpenCV labelling format conversion.

```
141     '''
142     This is the hard example mining process where hard examples are extracted from negative images
143     '''
144     #Retrieving the trained SVM model on HOG features
145     hog.setSVMDetector(get_svm_detector(svm))
146
147     #Traversing through all the negative images
148     for i in range(len(full_negative_list)):
149         '''
150         Detecting hard examples based on HOG features
151
152         @params:
153         winStride: the step size that represents the window's movement in the x and y direction.
154         padding: the amount of padding on the edge of the image.
155         scale: the scaling factor used by the image pyramid.
156
157         @return:
158         rects: the bounding boxes' x and y coordinates and height and weight.
159         weights: distance to the hyperplane, the bigger the better.
160         '''
161         rects, weights = hog.detectMultiScale(full_neg_list[i], winStride=(2, 2), padding=(4,4), scale=1.5)
162         for (x, y, w, h) in rects:
163             #Extracting the image within the bounding box
164             hard_examples = full_negative_list[i][y:y + h, x:x + w]
165             #Resizing hard example images and append to a list that stores all the hard examples
166             hard_example_list.append(cv2.resize(hard_examples, (64, 128)))
167
168     #Extracting the HOG features from all the hard examples
169     for i in range(len(hard_list)):
170         #Computing HOG features
171         hard_feature = hog.compute(hard_example_list[i])
172         #Standardizing the feature by removing the mean and scaling to unit variance
173         #Note that standardizing the feature vector is helpful for the training
174         scaler = StandardScaler()
175         standardized_feature = scaler.fit_transform(hard_feature)
176         #Append the hard features to the training feature set
177         features.append(standardized_feature)
178         #Append the hard labels to the training label set
179         labels.append(-1)
180
181     '''
182     Train the SVM again with the additional hard examples
183     '''
184     svm.train(np.array(features), cv2.ml.ROW_SAMPLE, np.array(labels))
```

Figure A.2: The code snippet for the hard example mining process.

```
import numpy as np
import os
import skimage
import cv2
import glob
import pickle
import matplotlib.pyplot as plt
from skimage import data
from skimage import io
from skimage.color import rgb2gray
from skimage.transform import resize
from skimage.restoration import denoise_bilateral, denoise_nl_means
from skimage.filters import gaussian, sobel, roberts, prewitt, laplace, median
from skimage.feature import canny, hog, blob_log
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from scipy.ndimage import convolve1d
from scipy.ndimage import uniform_filter, gaussian_laplace
from sklearn import svm
from sklearn import metrics

class Image:
    def __init__(self, img, feature):
        self.img = img
        self.feature = feature

    #Create a copy of the object for manipulation
    def copy(self):
        return Image(self.img, self.feature)

    #Image manipulation using skimage.
    #Resize the image, size enclosed in brackets.
    def resize(self, size):
        resized = self.copy()
        resized.img = resize(self.img, size, anti_aliasing=True)
        return resized

    #Grayscale the image
    #Note: rbg2gray normalize pixel values
    def gray_scale(self):
        grayscaled = self.copy()
        grayscaled.img = rgb2gray(self.img)
        return grayscaled

    #Average filtering
    def average_filtering(self):
        avg = self.copy()
        kernel = np.ones(3, self.img.dtype) / 3
        avg.img = convolve1d(convolve1d(self.img, kernel, axis=0), kernel, axis=1)
        return avg

    '''
    #Average filtering by scipy. 3 by 3 average kernel.
    def average_filtering(self):
        avg = self.copy()
        avg.img = uniform_filter(self.img, 3)
        return avg
    '''

    #Gaussian filtering
    def gaussian_filtering(self):
        gauss = self.copy()
        gauss.img = gaussian(self.img,sigma=1,multichannel=False)
        return gauss

    #Median filtering
    def median_filtering(self):
        med = self.copy()
        med.img = median(self.img)
        return med
```

Figure A.3: The code snippet for image preprocessing.

```
    #Bilateral filtering
    def bilateral_filtering(self):
        bilat = self.copy()
        bilat.img = denoise_bilateral(self.img)
        return bilat

    #Non-local means denoising, fast version
    def nonlocalmeans(self):
        nonlocalmean = self.copy()
        nonlocalmean.img = denoise_nl_means(self.img,fast_mode=True,patch_size=5,patch_distance=6,preserve_range=True)
        return nonlocalmean

    #Display image for visulization
    def display_image(self):
        #slice for test use only
        #io.imshow(self.img[185:225,190:235])
        io.imshow(self.img)
        io.show()

    #When implementing functions from scratch, pad image for convolution to maintain spatial size
    def zero_padding(self):
        padded = self.copy()
        padded.img = np.insert(self.img, 0, 0, axis = 0)
        padded.img = np.insert(padded.img, padded.img.shape[0], 0, axis = 0)
        padded.img = np.insert(padded.img, 0, 0, axis = 1)
        padded.img = np.insert(padded.img, padded.img.shape[1], 0, axis = 1)
        return padded

    #Show image size
    def get_size(self):
        print(self.img.shape)

    #Sobel edge detector
    def sobel_edge(self):
        #ravel() converts ndarray to 1d
        #boundary mode set to constant, equivalent to pad the image with 0 at boundary.
        self.feature = sobel(self.img, mode='constant',cval=0.0).ravel()

    #Canny edge detector
    def canny_edge(self, sigma=1.0):
        self.feature = canny(self.img).ravel()

    #Roberts edge detector
    def robert_edge(self):
        self.feature = roberts(self.img).ravel()

    #Prewitt edge detector:
    def prewitt_edge(self):
        self.feature = prewitt(self.img, mode='constant', cval=0.0).ravel()

    #Laplacian edge detector
    def laplacian_edge(self):
        self.feature = laplace(self.img, ksize=5).ravel()

    #LOG edge
    def log_edge(self):
        self.feature = gaussian_laplace(self.img, sigma=1, mode='constant', cval=0.0).ravel()

    #HOG
    def hog_edge(self):
        self.feature = hog(self.img, orientations=9, pixels_per_cell=(8,8), cells_per_block=(2,2), feature_vector=True)

    #Display feature and its dimension
    def display_feature(self):
        print(self.feature, self.feature.shape)

class support_vector_machine:
    def __init__(self, clf):
        self.clf = svm.LinearSVC()
```

Figure A.4: The code snippet for feature extraction.

```
143    #Standardize the feature before feeding to the SVM
144    X_scaler = StandardScaler().fit(features)
145    scaled_X = X_scaler.transform(features)
146
147    #Use 70% data for training and 30% for testing
148    rand_state = np.random.randint(0, 100)
149    X_train, X_test, y_train, y_test = train_test_split(scaled_X, labels, test_size=0.3, random_state=rand_state)
150
151    #Training
152    clf = svm.LinearSVC()
153    clf.fit(X_train, y_train)
154
155    #Predicting
156    y_pred = clf.predict(X_test)
157
158    #Calculate confusion matrix
159    print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
160    print("Precision:", metrics.precision_score(y_test, y_pred))
161    print("Recall:", metrics.recall_score(y_test, y_pred))
162
163    #Save the model for later use
164    obj = {"clf":clf, "X_scaler":X_scaler}
165    pickle.dump(obj, open("model.pickle", "wb"))
```

Figure A.5: The code snippet for training the SVM.

Bibliography

[1] World Health Organization. Road traffic injuries. [online]. https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries, 02 2020. Accessed: 2021-01-12.

[2] Government of Canada. Canadian motor vehicle traffic collision statistics: 2018. [online]. https://www.tc.gc.ca/eng/motorvehiclesafety/canadian-motor-vehicle-traffic-collision-statistics-2018.html, 12 2020. Accessed: 2021-01-12.

[3] Walter Brenner and Andreas Herrmann. *An Overview of Technology, Benefits and Impact of Automated and Autonomous Driving on the Automotive Industry*, pages 427–442. 01 2018.

[4] Neville Stanton and Paul Salmon. Human error taxonomies applied to driving: A generic driver error taxonomy and its implications for intelligent transport systems. *Safety Science*, 47:227–237, 02 2009.

[5] Santokh Singh. Critical reasons for crashes investigated in the national motor vehicle crash causation survey. 02 2015.

[6] Morgan Stanley Research Global. Autonomous Cars: Self-Driving the New Auto Industry Paradigm, 11 2013.

[7] Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. 06 2018.

[8] Keshav Bimbraw. Autonomous cars: Past, present and future - a review of the developments in the last century, the present scenario and the expected future of autonomous vehicle technology. *ICINCO 2015 - 12th International Conference on Informatics in Control, Automation and Robotics, Proceedings*, 1:191–198, 01 2015.

[9] Janosch Delcker. The man who invented the self-driving car (in 1986). [online]. `https://www.politico.eu/article/delf-driving-car-born-1986-ernst-dickmanns-mercedes/`, 07 2018. Accessed: 2021-01-12.

[10] Reinhold Behringer, S. Sundareswaran, Billy Gregory, R. Elsley, B. Addison, Wayne Guthmiller, R. Daily, and D. Bevly. The darpa grand challenge - development of an autonomous vehicle. pages 226 – 231, 07 2004.

[11] Massimo Bertozzi, Luca Bombini, Alberto Broggi, Michele Buzzoni, Elena Cardarelli, Stefano Cattani, Pietro Cerri, Alessandro Coati, Stefano Debattisti, Andrea Falzoni, Rean Fedriga, Mirko Felisa, Luca Gatti, A. Giacomazzo, Paolo Grisleri, Maria Laghi, Luca Mazzei, Paolo Medici, Matteo Panciroli, and Pietro Versari. Viac: An out of ordinary experiment. pages 175 – 180, 07 2011.

[12] The Tesla Team. Introducing software version 10.0. [online]. `https://www.`

tesla.com/en_CA/blog/introducing-software-version-10-0, 09 2019. Accessed: 2021-01-12.

[13] Clifford Atiyeh. Delphi engineers finish first autonomous cross-country road trip in an audi sq5. [online]. https://www.caranddriver.com/news/a15356366/delphi-engineers-finish-first-autonomous-cross-country-road-trip-in-an-audi-sq5/, 04 2015. Accessed: 2021-01-12.

[14] SAE International. Autodrive challenge. [online]. https://www.sae.org/attend/student-events/autodrive-challenge. Accessed: 2021-04-30.

[15] Keenan Burnett, Jingxing Qian, Xintong Du, Linqiao Liu, David Yoon, Tianchang Shen, Susan Sun, Sepehr Samavi, Michael Sorocky, Mollie Bianchi, Kaicheng Zhang, Arkady Arkhangorodsky, Quinlan Sykora, Shichen Lu, Yizhou Huang, Angela Schoellig, and Timothy Barfoot. Zeus: A system description of the two-time winner of the collegiate sae autodrive competition. *Journal of Field Robotics*, 38, 05 2020.

[16] Neal E.Boudette. Autopilot cited in death of chinese tesla driver. [online]. https://www.nytimes.com/2016/09/15/business/fatal-tesla-crash-in-china-involved-autopilot-government-tv-says.html, 09 2016. Accessed: 2021-01-12.

[17] Puneet Kohli and Anjali Chadha. *Enabling Pedestrian Safety Using Computer Vision Techniques: A Case Study of the 2018 Uber Inc. Self-driving Car Crash*, pages 261–279. 01 2020.

[18] National Transportation Safety Board. Collision between a sport utility vehicle

operating with partial driving automation and a crash attenuator mountain view, california march 23, 2018 hwy18fh011, 02 2020.

[19] Nidhi Kalra and Susan Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 12 2016.

[20] Waymo Team. Waymo reaches 5 million self-driven miles. [online]. `https://medium.com/waymo/waymo-reaches-5-million-self-driven-miles-61fba590fafe`, 02 2018. Accessed: 2021-01-12.

[21] Uwe Voelzke. Three sensor types drive autonomous vehicles. [online]. `https://evnoiagroup.com/three-sensor-types-drive-autonomous-vehicles/`. Accessed: 2021-01-12.

[22] Tesla. Advanced sensor coverage. [online]. `https://www.tesla.com/en_CA/autopilot`. Accessed: 2021-01-12.

[23] Klaus Mangold, Joseph Shaw, and Michael Vollmer. The physics of near-infrared photography. *European Journal of Physics*, 34:51–, 11 2013.

[24] Australian Government Bureau of Meteorology. How radar works. [online]. `http://www.bom.gov.au/australia/radar/about/index.shtml`. Accessed: 2021-04-30.

[25] Velodyne Lidar. Lidar 101 what is lidar?. [online]. `https://velodynelidar.com/what-is-lidar/`. Accessed: 2021-04-30.

[26] Toyota. Toyota research institute introduces next-generation automated driving research vehicle at consumer electronics show. [online].

https://www.toyota.ca/toyota/en/about/news/toyota-research-institute-introduces-nextgeneration-automated-driving-research-vehicle-at-consumer-electronics-show, 04 2018. Accessed: 2021-01-12.

[27] Alessio Carullo and Marco Parvis. An ultrasonic sensor for distance measurement in automotive applications. *Sensors Journal, IEEE*, 1:143 – 147, 09 2001.

[28] Thuy Mai. Global positioning system history. [online]. https://www.nasa.gov/directorates/heo/scan/communications/policy/GPS_History.html, 08 2017. Accessed: 2021-03-11.

[29] Heather Somerville, Paul Lienert, and Alexandria Sage. Uber's use of fewer safety sensors prompts questions after arizona crash. [online]. https://www.reuters.com/article/us-uber-selfdriving-sensors-insight/ubers-use-of-fewer-safety-sensors-prompts-questions-after-arizona-crash-idUSKBN1H337Q, 03 2018. Accessed: 2021-03-11.

[30] Apollo Auto. Inside apollo 6.0: A road towards fully driverless technology. [online]. https://medium.com/apollo-auto/inside-apollo-6-0-a-road-towards-fully-driverless-technology-522b2b4295cc, 10 2020. Accessed: 2021-04-30.

[31] MIT Technology Review. How coronavirus is accelerating a future with autonomous vehicles. [online]. https://www.technologyreview.com/2020/05/18/1001760/how-coronavirus-is-accelerating-autonomous-vehicles/, 05 2020. Accessed: 2021-04-30.

[32] Sarah Dai. Chinese internet giant baidu offers free trial robotaxi rides through

search and map apps in changsha. [online]. https://iot-automotive.news/baidu-apollo-robotaxi/. Accessed: 2021-03-11.

[33] Joanne Zwinkels. *Light, Electromagnetic Spectrum*, pages 1–8. 01 2015.

[34] Antoni Rogalski. History of infrared detectors. *Opto-Electronics Review*, 20, 09 2012.

[35] Kevin Dobbin and Richard Simon. Optimally splitting cases for training and testing high dimensional classifiers. *BMC medical genomics*, 4:31, 04 2011.

[36] Mohammad Hossin and Sulaiman M.N. A review on evaluation metrics for data classification evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5:01–11, 03 2015.

[37] Taiwo Ayodele. *Introduction to Machine Learning*. 02 2010.

[38] E. Forgy. Cluster analysis of multivariate data : efficiency versus interpretability of classifications. *Biometrics*, 21:768–769, 1965.

[39] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Zitnick. Microsoft coco: Common objects in context. volume 8693, 04 2014.

[40] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: a large-scale hierarchical image database. pages 248–255, 06 2009.

[41] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2005)*, 2, 06 2005.

[42] Exposure Therapy. Image noise in digital cameras. [online]. https://exposuretherapy.ca/photography-guide/image-noise-in-digital-cameras/. Accessed: 2021-03-11.

[43] Lewis Griffin. Mean, median and mode filtering of images. *Proceedings of the Royal Society*, 456:2995–3004, 12 2000.

[44] Estevao Gedraite and M. Hadad. Investigation on the effect of a gaussian blur in image filtering and segmentation. pages 393–396, 01 2011.

[45] Sylvain Paris, Pierre Kornprobst, Jack Tumblin, and Frédo Durand. A gentle introduction to bilateral filtering and its applications. *ACM SIGGRAPH 2007 Papers - International Conference on Computer Graphics and Interactive Techniques*, page 1, 08 2008.

[46] Sebastian Villar, Sebastian Torcida, and Gerardo Acosta. Median filtering: A new insight. *Journal of Mathematical Imaging and Vision*, 58:1–17, 05 2017.

[47] Bartomeu Coll and Jean-Michel Morel. Non-local means denoising. *Image Processing On Line*, 1, 09 2011.

[48] Graham Finlayson, Bernt Schiele, James Crowley, and Cambridge Ma. Comprehensive colour image normalization. *Proc ECCV*, 1, 12 1999.

[49] Saravanan Chandran. Color image to grayscale image conversion. pages 196 – 199, 04 2010.

[50] Agnieszka Mikołajczyk and Michał Grochowski. Data augmentation for improving deep learning in image classification problem. pages 117–122, 05 2018.

[51] Dongping Tian. A review on image feature extraction and representation techniques. *International Journal of Multimedia and Ubiquitous Engineering*, 8:385–395, 01 2013.

[52] Anil K. Jain and Aditya Vailaya. Image retrieval using color and shape. *Pattern Recognition*, 29(8):1233–1244, 1996.

[53] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Qian Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the qbic system. *Computer*, 28(9):23–32, 1995.

[54] G. Pass and R. Zabih. Histogram refinement for content-based image retrieval. In *Proceedings Third IEEE Workshop on Applications of Computer Vision. WACV'96*, pages 96–102, 1996.

[55] B. S. Manjunath and W. Y. Ma. Texture features for browsing and retrieval of image data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(8):837–842, 1996.

[56] William McIlhagga. The canny edge detector revisited. *International Journal of Computer Vision*, 91, 02 2011.

[57] Wenshuo Gao, Xiaoguang Zhang, Lei Yang, and Huizhong Liu. An improved sobel edge detection. volume 5, pages 67 – 71, 08 2010.

[58] J. M. S. prewitt. Object enhancement and extraction. *Picture processing and Psychopictorics, B. Lipkin and A. Rosenfeld, Eds. New York: Academic*, pages 75–149, 1970.

[59] Ashis Pradhan. Support vector machine-a survey. *IJETAE*, 2, 09 2012.

[60] Debasish Basak, Srimanta Pal, and Dipak Patranabis. Support vector regression. *Neural Information Processing – Letters and Reviews*, 11, 11 2007.

[61] Claudio Gentile and Manfred Warmuth. Linear hinge loss and average margin. pages 225–231, 01 1998.

[62] Sebastian Ruder. An overview of gradient descent optimization algorithms. 09 2016.

[63] Mariette Awad and Rahul Khanna. *Support Vector Machines for Classification*, pages 39–66. 01 2015.

[64] Fereshteh Falah Chamasemani and Yashwant Singh. Multi-class support vector machine (svm) classifiers – an application in hypothyroid detection and classification. pages 351–356, 09 2011.

[65] Gualtiero Piccinini. The first computational theory of mind and brain: A close look at mcculloch and pitts's "logical calculus of ideas immanent in nervous activity". *Synthese*, 141, 08 2004.

[66] Christoph von der Malsburg. Frank rosenblatt: Principles of neurodynamics: Perceptrons and the theory of brain mechanisms. *Brain Theory*, pages 245–248, 01 1986.

[67] Marvin Minsky and Seymour A.Papert. *Perceptrons: An Introduction to Computational Geometry*. 01 1969.

[68] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 03 2020.

[69] Xavier Glorot, Antoine Bordes, and Y. Bengio. Deep sparse rectifier neural networks. volume 15, 01 2010.

[70] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *IEEE International Conference on Computer Vision (ICCV 2015)*, 1502, 02 2015.

[71] Kalyan Das, Jiming Jiang, and J.N.K. Rao. Mean squared error of empirical predictor. *Annals of Statistics*, 32, 07 2004.

[72] Tianfeng Chai and R. Draxler. Root mean square error (rmse) or mean absolute error (mae)? *Geosci. Model Dev.*, 7, 01 2014.

[73] Shuang Song, Kamalika Chaudhuri, and Anand Sarwate. Stochastic gradient descent with differentially private updates. pages 245–248, 12 2013.

[74] Sagar Sharma. Activation functions in neural networks. [online]. `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`, 06 17. Accessed: 2021-03-11.

[75] Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278 – 2324, 12 1998.

[76] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[77] Matthew Zeiler and Rob Fergus. Visualizing and understanding convolutional neural networks. volume 8689, 11 2013.

[78] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. 09 2014.

[79] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

[80] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 7, 12 2015.

[81] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. pages 779–788, 06 2016.

[82] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Region-based convolutional networks for accurate object detection and segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38:1–1, 12 2015.

[83] Ross Girshick. Fast r-cnn. 04 2015.

[84] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39, 06 2015.

[85] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. 12 2016.

[86] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. 04 2018.

[87] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. 12 2016.

[88] Chien-Yao Wang, Hong-yuan Liao, Yuen-Hua Wu, Ping-Yang Chen, Jun-Wei Hsieh, and I-Hau Yeh. Cspnet: A new backbone that can enhance learning capability of cnn. pages 1571–1580, 06 2020.

[89] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37, 06 2014.

[90] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.

[91] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation. 03 2018.

[92] Zhengyou Zhang. A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22:1330 – 1334, 12 2000.

[93] MathWorks. What is camera calibration? [online]. `https://www.mathworks.com/help/vision/ug/camera-calibration.html`. Accessed: 2021-03-15.

[94] Clive Fraser. Digital camera self-calibration. *ISPRS Journal of Photogrammetry and Remote Sensing*, 52:149–159, 08 1997.

[95] Surya Prakash, Pei Lee, Terry Caelli, and Tim Raupach. Robust thermal camera calibration and 3d mapping of object surface temperatures - art. no. 62050j. volume 6205, pages 62050J–62050J, 04 2006.

[96] Agnieszka Mikołajczyk and Michał Grochowski. Data augmentation for improving deep learning in image classification problem. pages 117–122, 05 2018.

[97] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv 1412.6572*, 12 2014.

[98] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Y. Bengio. Generative adversarial networks. *Advances in Neural Information Processing Systems*, 3, 06 2014.

[99] Sangdoo Yun, Dongyoon Han, Sanghyuk Chun, Seong Joon Oh, Youngjoon Yoo, and Junsuk Choe. Cutmix: Regularization strategy to train strong classifiers with localizable features. pages 6022–6031, 10 2019.

[100] E. Million. The hadamard product elizabeth million april 12 , 2007 1 introduction and basic results. 2007.

[101] L. Torrey and J. Shavlik. Transfer learning. *Handbook of Research on Machine Learning Applications*, 01 2009.

[102] Karl Weiss, Taghi Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big Data*, 3, 05 2016.

[103] Jason Yosinski, Jeff Clune, Y. Bengio, and Hod Lipson. How transferable are features in deep neural networks? pages 3320–3328, 01 2014.

[104] FLIR. Free flir thermal dataset for algorithm training [online]. `https://www.flir.ca/oem/adas/adas-dataset-form/`. Accessed: 2021-03-15.

[105] OpenCV. Drawing functions in opencv [online]. `https://docs.opencv.org/master/dc/da5/tutorial_py_drawing_functions.html`. Accessed: 2021-03-15.

[106] Sabina Pokhrel. Image data labelling and annotation — everything you need to know [online]. https://towardsdatascience.com/image-data-labelling-and-annotation-everything-you-need-to-know-86ede6c684b1. Accessed: 2021-03-15.

[107] Jamil Al Azzeh, Bilal Zahran, and Ziad Alqadi. Salt and pepper noise: Effects and removal. *JOIV : International Journal on Informatics Visualization*, 2, 07 2018.

[108] Rasmus Rothe, Matthieu Guillaumin, and Luc Van Gool. Non-maximum suppression for object detection by passing messages between windows. volume 9003, 04 2015.

[109] Rafael Medina-Carnicer, A. Carmona-Poyato, Rafael Muñoz-Salinas, and Francisco Madrid-Cuevas. Determining hysteresis thresholds for edge detection by combining the advantages and disadvantages of thresholding methods. *Image Processing, IEEE Transactions on*, 19:165 – 173, 02 2010.

[110] Saket Bhardwaj and Ajay Mittal. A survey on various edge detector techniques. *Procedia Technology*, 4:220–226, 12 2012.

[111] Xin Wang. Laplacian operator-based edge detectors. *IEEE transactions on pattern analysis and machine intelligence*, 29:886–90, 06 2007.

[112] Hui Kong, Hatice Cinar Akakin, and Sanjay Sarma. A generalized laplacian of gaussian filter for blob detection and its applications. *IEEE Transactions on Cybernetics*, 43:1719–1733, 01 2013.

[113] Rasmus Bro and Age Smilde. Principal component analysis. *Analytical methods*, 6:2812, 05 2014.