COMPARATIVE STUDY OF CONVOLUTIONAL NEURAL NETWORK OBJECT DETECTION ALGORITHMS FOR VEHICLE DETECTION IN IMAGE PROCESSING

by

Saieshan Reddy
(21557490)

Submitted in fulfilment of the requirements for the degree of:

MASTER OF ENGINEERING
in the
DEPARTMENT OF ELECTRONIC AND COMPUTER ENGINEERING,

FACULTY OF ENGINEERING AND THE BUILT ENVIRONMENT

at the

DURBAN UNIVERSITY OF TECHNOLOGY

July 2024

Supervisor: Dr N Pillay
Co-Supervisor: Dr N Singh

PREFACE

I, Saieshan Reddy, a Masters student at the Durban University of Technology hereby present this thesis titled “Comparative Study of Convolutional Neural Network Object Detection Algorithms for Vehicle Detection in Image Processing” to fulfil the requirements for the degree of Master of Engineering in Electronic & Computer Engineering.

In an era dominated by vast amounts of visual data, the ability to extract meaningful information from images has become paramount. A fundamental task in computer vision which plays a pivotal role in various applications ranging from autonomous driving to medical imaging is object detection. Convolutional Neural Networks (CNNs), with their remarkable ability to learn hierarchical features, have revolutionized the field of object detection. This thesis presents a comparative study of three prominent CNN-based object detection algorithms: Faster R-CNN, YOLO v3 (You Only Look Once), and SSD (Single Shot MultiBox Detector) with specific application to vehicle detection.

With anticipation and excitement, I extend my heartfelt gratitude for your interest in this work. May this thesis serve as both a source of enlightenment and inspiration, igniting curiosity and fostering a deeper understanding within the dynamic field of computer vision.

Saieshan Reddy

Durban University of Technology

July 2024

DECLARATION 1: SUPERVISOR

According to the contents of this thesis, as the candidates' Supervisor, I agree to the submission of the thesis.

Dr N Pillay

(Main supervisor)

Date: 5/03/2025

Dr N Singh

(Co-supervisor)

Date: 5 March 2025

DECLARATION 2: PLAGIARISM

I, **Saieshan Reddy (21557490)**, the undersigned, declare that:

(i) The research reported in this thesis, except where otherwise indicated, is my original work.

(ii) This thesis has not been submitted for degree or examination at any other university.

(iii) This thesis does not contain other persons' data, images, graphs, or other information unless specifically acknowledged as being sourced from other persons.

(iv) This thesis does not contain other persons' writing unless specifically acknowledged as being sourced from other researchers. Where other written sources have been quoted, then:

a. their words have been re-written, but the general information attributed to them has been referenced.

b. where their exact words have been used, their writing has been placed inside quotation marks, and referenced.

(v) Where I have reproduced a publication of which I am an author, co-author or editor, I have indicated in detail which part of the publication was actually written by myself alone and have fully referenced such publications.

(vi) This thesis does not contain text, graphics, or tables copied and pasted from the Internet unless specifically acknowledged and the source is detailed in the dissertation and the Reference sections.

Saieshan Reddy

July 2024

DECLARATION 3: PUBLICATIONS

I, **Saieshan Reddy (21557490)** declare that the following publications came of this thesis.

1. S. Reddy, N. Pillay and N. Singh, “Comparative Study of Convolutional Neural Network Object Detection Algorithms for Image Processing,” 2023 International Conference on Electrical, Computer and Energy Technologies (ICECET), Cape Town, South Africa, 2023, pp. 1-5, doi: 10.1109/ICECET58911.2023.10389186.
2. S. Reddy, N. Pillay, and N. Singh, “Comparative Evaluation of Convolutional Neural Network Object Detection Algorithms for Vehicle Detection,” Journal of Imaging, vol. 10, no. 7, p. 162, Jul. 2024, doi: 10.3390/jimaging10070162.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Dr N Pillay, and co-supervisor, Dr N Singh, for their unwavering support, guidance, and mentorship throughout this research endeavour. Their expertise, encouragement, and invaluable feedback have been instrumental in shaping this thesis.

My heartfelt appreciation extends to my family and friends for their endless love, encouragement, and understanding during this journey. Their unwavering belief in my abilities has been a constant source of strength and motivation.

I extend my sincere thanks to all the researchers, developers, and contributors whose work laid the foundation for this thesis. Their groundbreaking discoveries and open-source contributions have been indispensable in advancing the field of computer vision.

Lastly, I would like to express my profound gratitude to the readers of this thesis. Your interest, engagement, and support are deeply appreciated, and it is my hope that this work contributes meaningfully to the academic discourse and inspires future research endeavours.

ABSTRACT

The introduction of Convolutional Neural Networks (CNNs) has revolutionised the field of computer vision, particularly in the domain of object detection. This thesis presents a comprehensive comparative study of CNN-based object detection algorithms for vehicle detection, aiming to explain their intricate architecture, differentiation in methodology, and characteristics in performance. Beginning with an exploration of the theoretical underpinnings of CNNs and object detection, a solid foundation is established upon which the comparative analysis is built. Practical implementation and experimentation play a pivotal role in this study. The three different object detector algorithms being evaluated in this study within the MATLAB® development environment are Single Shot MultiBox Detector (SSD), Faster Region-Based Convolutional Network (R-CNN), and You Only Look Once (YOLO v3). Through research and experimentation, the strengths and limitations of each algorithm are provided. The findings of this comparative study not only contribute to the academic understanding of CNN-based object detection but also offer practical guidance to practitioners and researchers in selecting appropriate algorithms for specific domain applications such as vehicle image processing. Furthermore, this thesis serves as a roadmap for future research endeavours, highlighting areas for further exploration and improvement within the realm of object detection using convolutional neural networks.

Keywords: Convolutional Neural Networks, Object Detection, Faster R-CNN, YOLO v3, SSD, MATLAB

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ANN	Artificial Neural Network
AP	Average Precision
APReLU	Adaptive Parametric Rectified Linear Unit
BCE	Binary Cross-Entropy
CE	Cross-Entropy
CNN	Convolutional Neural Network
ELU	Exponential Linear Unit
FP	False Positive
FN	False Negative
FPN	Feature Pyramid Network
GB	Gigabyte
GHz	Gigahertz
GPS	Global Positioning System
GPU	Graphical Processing Unit
ICECET	International Conference on Electrical, Computer and Energy Technologies
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IoU	Intersection Over Union
JPEG	Joint Photographic Experts Group
KCF	Kernelized Correlation Filters
mAP	Mean Average Precision
MSE	Mean Squared Error
PC	Personal Computer
PR	Precision-Recall
PReLU	Parametric Rectified Linear Unit
RAM	Random Access Memory
R-CNN	Region-Based Convolutional Neural Network
ReLU	Rectified Linear Unit
RMSprop	Root Mean Squared Propagation
RNN	Recurrent Neural Network

RoI	Region-of-Interest
RPN	Region Proposal Network
SBC	Single-Board Computer
SGD	Stochastic Gradient Descent
SGDM	Stochastic Gradient Descent with Momentum
SPP	Spatial Pyramid Pooling
SSD	Single Shot MultiBox Detector
TP	True Positive
YOLO	You Only Look Once

CHAPTER 1 Introduction

1.1 Introduction

Convolutional Neural Networks (CNNs) form part of the ever-growing field of computer vision and has revolutionised the manner in which machines interpret and perceive visual information [1]. Object detection is a fundamental principle that lies at the heart of this revolution. It has profound implications over numerous domains, namely autonomous vehicles, healthcare, robotics and surveillance [2]. Emerging as the catalyst for unprecedented advancements in the field of computer vision, CNN-based object detection algorithms offer unparalleled scalability, accuracy and efficiency [3].

The journey towards effective object detection with CNNs begins with their unique architecture, inspired by the intricate connectivity patterns found in the visual cortex of biological organisms [4]. By leveraging layers of convolutional filters, pooling operations, and non-linear activation functions, CNNs excel at automatically extracting hierarchical representations of features from raw image data which is illustrated in Fig. 1.1. This innate ability to learn discriminative features directly from pixel values has propelled CNNs to the forefront of image processing tasks namely object detection, object tracking [5], image segmentation [6], learning and indexing [7].

Object detection, as a task, encompasses two primary objectives: accurately identifying and localizing the spatial extent of objects found within images [6]. Traditional approaches to object detection typically depended on handcrafted feature extraction methods and elaborate post-processing steps. The advent of CNNs ushered in revolutionary

developments, enabling end-to-end feature representation learning and spatial localisation directly from the data [8].

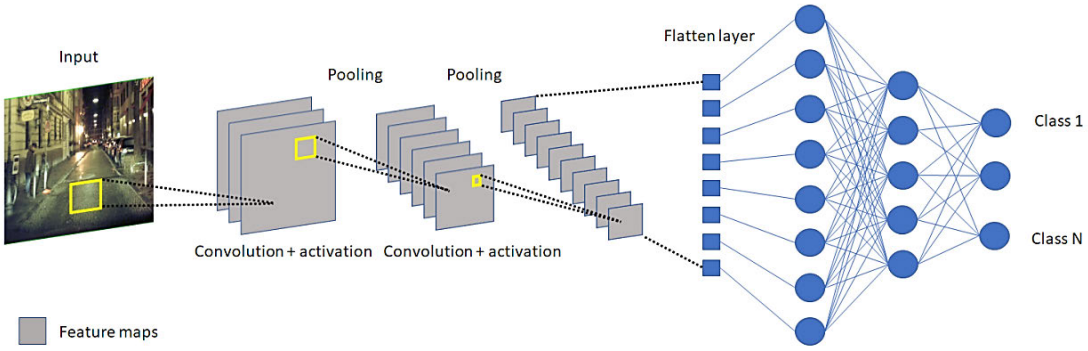

Fig. 1.1. Simplified structure of CNNs [7]

CNN-driven object detection algorithms usually follow a multi-step pipeline, wherein proposals which are known as candidate object regions are first created and subsequently improved on through a process of bounding box regression and classification. This multi-step pipeline is often augmented with techniques such as anchor box mechanisms, non-maximum suppression, and region proposal networks (RPNs) in order to improve efficiency and accuracy.

Key milestones in the evolution of CNN-based object detection algorithms include popular algorithms such as Faster Region-Based Convolutional Network (R-CNN) [9], You Only Look Once (YOLO) [10], and Single Shot MultiBox Detector (SSD) [11]. Faster R-CNN introduced the groundbreaking concept of integrating region proposal generation into the network architecture itself, leading to significant improvements in both speed and accuracy. YOLO, with its innovative single-shot detection approach, redefined the boundaries of real-time object detection, making it suitable for applications requiring rapid

inference speeds. SSD, on the other hand, prioritized efficiency by predicting object bounding boxes directly from feature maps at multiple scales, thus achieving a balance between speed and accuracy.

Beyond the realm of research and development, CNN-based object detection algorithms have found widespread adoption in practical applications across various industries. From enabling autonomous vehicles to navigate complex environments to empowering medical imaging systems to detect abnormalities with high precision, the impact of CNN-based object detection is profound and far-reaching. Implementing CNN-based object detection on single-board computers (SBCs) has garnered significant attention due to the increasing demand for embedded vision systems in various applications such as autonomous vehicles, robotics, drones, smart surveillance, and Internet of Things (IoT) devices [12]. Several studies have explored the challenges and solutions associated with deploying CNN-based object detection algorithms on resource-constrained SBCs, aiming to achieve real-time inference while balancing algorithm complexity, accuracy, and efficiency.

In this work, we aim to evaluate the performance of three CNN object detection algorithms in MATLAB® used for vehicle image processing. By assessing the effectiveness and efficiency of these models, we can gain insights into their strengths and limitations. Furthermore, a comparative analysis will be provided to understand how each of these CNN algorithms perform relative to each other in terms of vehicle detection capabilities. This analysis will shed light on which algorithm offers the best balance of speed, accuracy, and computational efficiency, helping advance the deployment of CNNs in practical vehicle detection applications. As we delve deeper into the intricacies of CNN-based object detection algorithms, we embark on a journey of exploration and discovery, uncovering the underlying principles, architectural innovations, optimization techniques,

and real-world applications that continue to shape the future of computer vision. By understanding the nuances of these algorithms, researchers can harness the full potential of CNNs to tackle complex visual recognition tasks for vehicle detection and pave the way for a future where machines perceive and understand the visual world with unprecedented accuracy and efficiency.

1.2 Research Problem Statement

There has been a significant and rapid transformation in object detection within the field of computer vision [1]. The identification and classification of objects have long been key areas of focus in computer vision research due to their usefulness across various domains such as video surveillance, artificial intelligence vision, and autonomous driving [2]. Prakas and Nagalakshmi [12] demonstrated that object detection can be efficiently carried out on SBCs through the utilization of kernelized correlation filters (KCF), marking a shift in machine learning practices. This approach incorporates human interaction by enabling data processing with limited resources while maintaining the effectiveness and efficiency of computer vision. Consequently, this makes object detection technology more accessible and cost-effective for a broader range of users and applications.

This research addresses the gap in evaluating the performance and effectiveness of CNN-based object detection algorithms for vehicle image processing on personal computers (PCs). The problem lies in determining whether CNN-based algorithms can provide a fast and economical solution by leveraging pre-trained models that require less computational power during inference. Therefore, this research aims to evaluate the performance of three CNN object detection algorithms in MATLAB® and provide a comparative analysis of their strengths and weaknesses in terms of speed, accuracy, and

efficiency. Solving this problem will help demonstrate that CNN-based object detection can be a practical, accessible, and cost-effective solution for vehicle image processing.

1.3 Research Question

This research aimed to compare the performance of CNN object detection algorithms on SBCs/PCs. The following research questions could be developed:

- Can CNN object detection algorithms be employed on SBCs/PCs?
- Which CNN object detection algorithm has the best combination of speed vs accuracy?

1.4 Research Aims and Objectives

The aim of this paper is to conduct a comparative study using MATLAB® to evaluate CNN object detection algorithms for vehicle detection on a portable PC that has limited graphical processing unit (GPU) computing.

To achieve this aim, the following objectives were set:

- To comprehensively review relevant publications on CNN object detection algorithms.
- To develop three CNN object detection algorithms in MATLAB®.
- To evaluate the performance of three CNN object detection algorithms for vehicle image processing.
- To provide a comparative analysis of the three CNN object detection algorithms evaluated in MATLAB® for vehicle image processing.

1.5 Research Limitations

This research is limited in scope as follows:

- Only three CNN object detection algorithms were studied in MATLAB®: Faster R-CNN, YOLO v3 and SSD. These were the only three deep learning one-stage and two-stage object detector functions provided by MATLAB® in their Computer Vision Toolbox™ and Deep Learning Toolbox™ at the inception of this research. All three of these algorithms have the following feature support in MATLAB®: multiple classes, deep learning, code generation and GPU.
- The dataset class used in this research was limited to only vehicles.
- A single formulated dataset of 652 images of vehicles from the rear split between two different resolutions of 896 x 592 pixels and 360 x 240 pixels was used for training and evaluation. The formulated dataset is described in Section 3.3.

1.6 Research Contributions

This research makes the following contributions:

- The publication of two research outputs, a conference paper and a journal article, employing and evaluating CNN object detection algorithms for vehicle image processing on a PC.
- The investigation of commonly used object detection algorithms and comparing their performance using a single dataset.
- The latter also enables future work for improvements on the algorithms.

1.7 Structure of the Thesis

Chapter 1 is an introduction to the background of the research and gives motivation for this research through the problem statement, objectives, and contributions.

Chapter 2 provides a comprehensive literature review of relevant publications based on CNN and object detection algorithms.

Chapter 3 presents detailed steps in developing the CNN object detection algorithms and choice of evaluation metrics.

Chapter 4 gives comprehensive results of the developed algorithms and presents a detailed analysis of the results obtained.

Chapter 5 concludes this thesis by highlighting the challenges, limitations, and recommendations for future research.

CHAPTER 2 Literature Review

2.1 Introduction

This chapter provides a comprehensive review of relevant literature which influenced the work undertaken in this research. The literature review was accomplished mainly by reviewing publications, conference papers, and journal articles found in common online databases such as the IEEE Xplore digital library and Google Scholar.

The broad topics addressed during the literature review include CNN and Object Detection Algorithms. The section on CNN aims to provide a foundational understanding of CNNs by analysing its history, fundamental principles, key components, and applications. This section then delves into the common layers of CNN: convolutional layer, activation layer, pooling layer, and the fully connected layer. An in-depth exploration of these layers is given by examining the underlying principles, architectural variations, optimization techniques, and applications in computer vision tasks. The section on object detection algorithms focuses on the three algorithms selected for this research: Faster R-CNN, YOLO v3 and SSD. A comprehensive review of these algorithms is provided by examining their architecture, training methodology, performance evaluation and significant contributions towards the field of object detection.

2.2 Convolutional Neural Networks

CNNs represent a pivotal advancement in the field of artificial intelligence, particularly within the domain of computer vision. CNNs have emerged as a powerful tool

for extracting meaningful features from visual data, enabling tasks such as object detection, image classification, segmentation, and more. A common example of the CNN architectural framework is illustrated in Fig. 2.1.

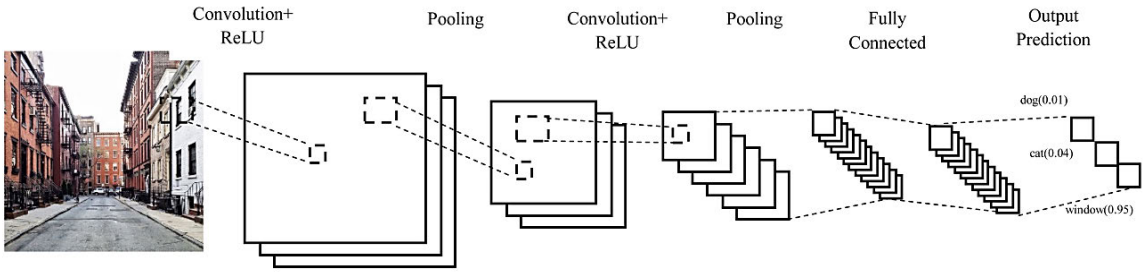

Fig. 2.1. Architecture of CNN framework [13]

The origin of CNNs dates back to the 1950s and 1960s [14], with the pioneering work of neuroscientists and computer scientists on understanding visual processing in the brain and developing early artificial neural networks (ANNs). However, it wasn't until the late 1980s and early 1990s that the concept of convolutional networks began to take shape, with the introduction of the Neocognitron model of Fukushima [15] and the later LeNet-5 architecture of LeCun et al [16]. These early algorithms laid the groundwork for modern CNNs, demonstrating the efficacy of hierarchical feature learning and weight sharing in visual pattern recognition tasks.

At the core of CNNs lie several fundamental principles that differentiate them from traditional fully connected neural networks [4]. Convolutional layers, the building blocks of CNNs, employ learnable filters or kernels to extract spatial features from input data through convolution operations. These filters capture local patterns and spatial relationships, enabling the network to learn hierarchical representations of the input. Pooling layers reduce spatial dimensions while preserving important features, improving

the algorithm's translation invariance and computational efficiency. Additionally, activation functions enables the network to learn complex mappings between input and output spaces by the introduction of non-linearity.

CNNs comprise several key components that work together to enable effective feature extraction and representation learning. These include convolutional layers for feature extraction, pooling layers for spatial down sampling, activation functions for introducing non-linearity, fully connected layers for classification or regression tasks, and various optimization techniques for training the network parameters. Recent advancements have also introduced components such as skip connections, attention mechanisms, and normalization layers to further enhance algorithm performance and interpretability [17].

The versatility and effectiveness of CNNs have led to their widespread adoption across a multitude of applications. In addition to their prominent role in image classification tasks, CNNs are utilized in object detection, where they can accurately localize and classify objects within images. Semantic segmentation tasks [18] leverage CNNs to assign class labels to each pixel, enabling fine-grained understanding of scene contents. CNNs also find applications in medical imaging, autonomous vehicles, natural language processing, and beyond, showcasing their adaptability to diverse domains.

2.2.1 Convolutional Layer

The convolutional layer is the fundamental building block of CNNs, playing a pivotal role in feature extraction and hierarchical representation learning from input data.

The convolutional layer operates by applying learnable filters, also known as kernels or feature detectors, to the input data through convolution operations and is illustrated in Fig. 2.2. These filters slide over the input image, extracting local patterns and

spatial features while preserving spatial relationships. By sharing parameters across spatial locations, convolutional layers facilitate weight sharing, which reduces algorithm complexity and enhances generalization [19]. Additionally, padding and strides control the spatial dimensions of the output feature maps, enabling flexible receptive fields and spatial resolutions [20].

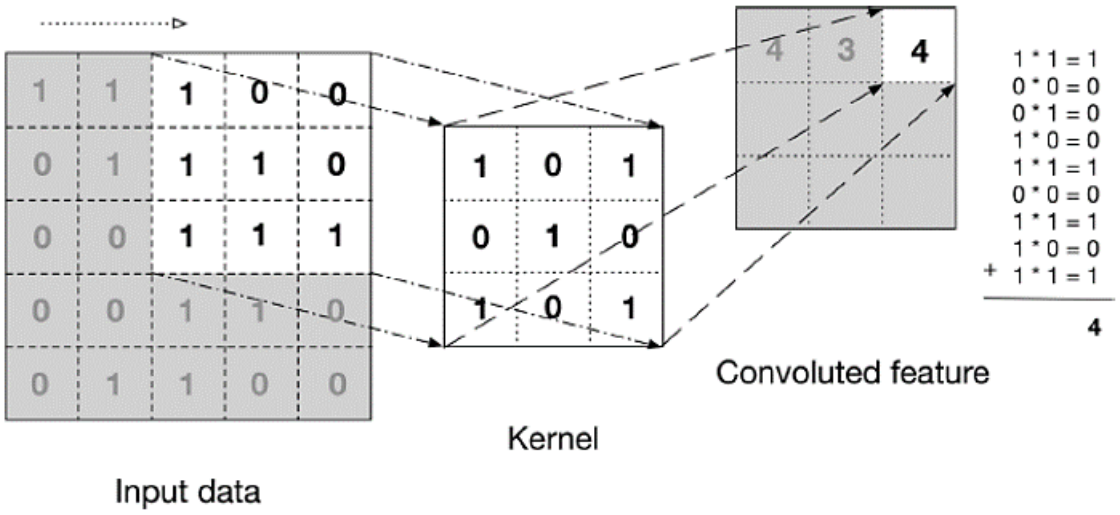

Fig. 2.2. The convolutional operation [4]

Several architectural variations of the convolutional layer have been proposed to enhance its capabilities and efficiency. One such variation is dilated convolutions [21], which introduces gaps between filter elements to increase the receptive field without sacrificing resolution. Depthwise separable convolutions [22] break down the standard convolution into separate depthwise and pointwise convolutions, reducing computational complexity and memory footprint. Transposed convolutions [23], also known as deconvolutions or upsampling layers, enable the generation of higher-resolution feature maps through learnable upsampling kernels.

Optimizing the convolutional layer's parameters is crucial for effective feature learning and algorithm training. Stochastic gradient descent (SGD) [24] with momentum and adaptive learning rate methods like the Adam Optimization Algorithm [25] and RMSprop (Root Mean Squared Propagation) [26] are commonly used optimization algorithms. Dropout and weight decay are regularization techniques which help prevent overfitting and improve generalization performance. Batch normalization stabilizes the activations within each mini-batch, accelerating convergence and improving gradient flow. Additionally, initialization schemes like Xavier Initialization and He Initialization ensure stable training dynamics and prevent vanishing or exploding gradients [27].

The convolutional layer serves as the cornerstone of CNNs, enabling a wide range of applications in computer vision. In image classification tasks, convolutional layers extract hierarchical features such as edges, textures, and object parts, facilitating discriminative feature representation. Object detection frameworks like Faster R-CNN and YOLO utilize convolutional layers for region proposal generation and feature extraction, enabling accurate localization and classification of objects within images. Semantic segmentation networks employ convolutional layers to produce dense pixel-wise predictions, facilitating scene understanding and image parsing [18].

2.2.2 Activation Layer

The activation layer plays a critical role in CNNs by introducing non-linearity, enabling the network to learn complex mappings between input and output spaces [28].

The activation layer introduces non-linearities into CNNs, allowing them to model complex relationships in data. Without activation functions, CNNs would be restricted to linear transformations, severely limiting their representational capacity and learning

capabilities. By applying non-linear activation functions element-wise to the output of convolutional layers, the activation layer introduces rich non-linearities that enable CNNs to approximate arbitrary functions and capture intricate patterns within data [29].

Several activation functions have been proposed for use in CNNs, each offering unique properties and characteristics, and are illustrated in Fig. 2.3. The Rectified Linear Unit (ReLU) [30] is one of the most commonly used activation functions, offering simplicity, computational efficiency, and improved gradient flow during backpropagation. Variants of ReLU, such as Leaky ReLU [31], Parametric ReLU (PReLU) [32], and Exponential Linear Unit (ELU) [33], enhances algorithm performance and stability and addresses limitations like dead neurons and vanishing gradients. Additionally, activation functions like Sigmoid [34] and Hyperbolic Tangent (Tanh) [35] are suitable for binary classification tasks or scenarios requiring bounded outputs.

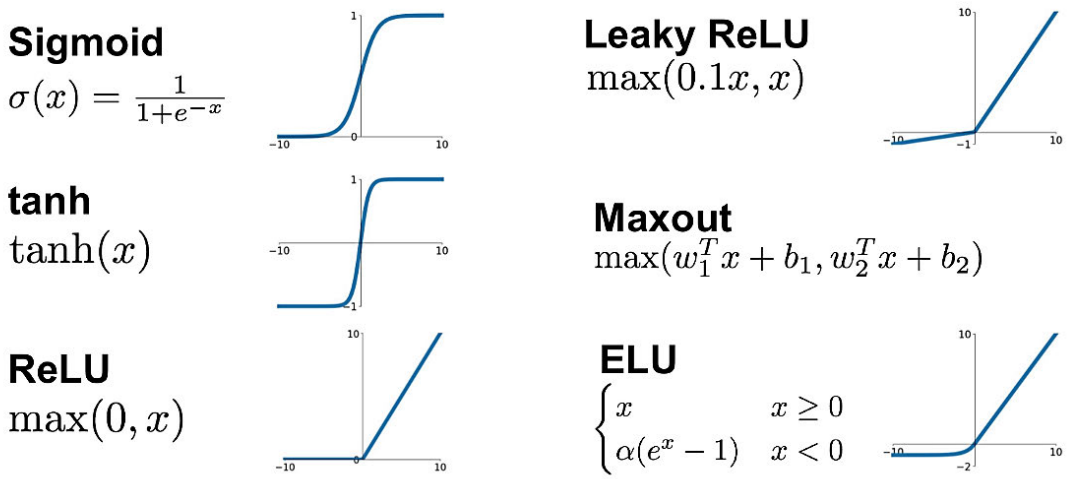

Fig. 2.3. Activation functions [30 – 35]

Beyond traditional activation functions, recent research has explored novel variations and extensions to enhance the capabilities of the activation layer. Swish [36], a

self-gated activation function, has demonstrated superior performance over ReLU and its variants in certain scenarios, leveraging dynamic non-linearities and smooth gradients. ELUs with squared non-linearity introduce self-normalizing properties, enabling stable training dynamics and improved generalization performance [37]. Adaptive activation functions, such as Adaptive Parametric Rectified Linear Unit (APReLU) [38], dynamically adjusts activation slopes based on input statistics, enhancing flexibility and adaptability.

Optimizing the activation layer is essential for effective algorithm training and performance. Initialization techniques, such as He Initialization for ReLU-based activations and Xavier Initialization for Sigmoid and Tanh activations, ensure stable training dynamics and prevent saturation or vanishing gradients. Regularization methods such as dropout and weight decay improves generalization and helps prevent overfitting. Furthermore, optimization algorithms like Adam and RMSprop adaptively adjust learning rates based on parameter gradients, accelerating convergence, and improving optimization efficiency [26].

The activation layer is a critical component of CNNs and deep learning architectures, enabling effective feature learning and representation. In image classification tasks, activation layers introduce non-linearities that enable CNNs to capture complex image patterns and learn discriminative features. In natural language processing, recurrent neural networks utilize activation functions to model sequential data and capture semantic relationships. Additionally, activation layers are integral to generative algorithms, reinforcement learning agents, and other deep learning paradigms, demonstrating their broad applicability across diverse domains.

2.2.3 Pooling Layer

The pooling layer is a fundamental component of CNNs, serving to downsample feature maps and reduce spatial dimensions while preserving important features.

The pooling layer operates by partitioning the input feature map into non-overlapping regions and applying a pooling function to each region to obtain a summary statistic [39]. This summary statistic, typically the maximum value (max pooling) or average value (average pooling), serves as a downsampled representation of the original input. Pooling layers introduce translation invariance, enabling CNNs to focus on the most important features while reducing computational complexity and memory requirements [40].

Two primary pooling operations are widely used in CNNs: max pooling [41] and average pooling [42]. Max pooling retains the maximum value within each pooling region, effectively preserving the most dominant features. This operation is particularly effective in capturing sharp transitions and invariant to small spatial translations. Average pooling, on the other hand, computes the average value within each pooling region, providing a smoother downsampled representation and promoting spatial smoothing. Both max pooling and average pooling are effective at reducing feature map dimensions while retaining important information. An example of max pooling and average pooling is illustrated in Fig. 2.4.

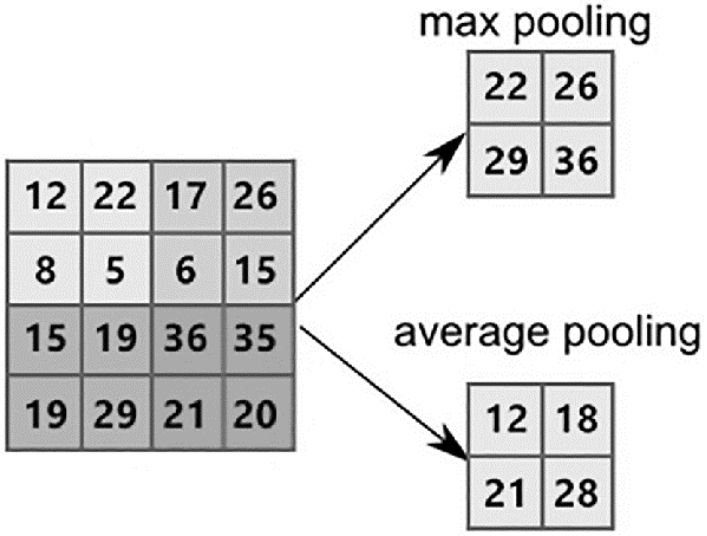

Fig. 2.4. Example of max and average pooling [4]

Several variations and extensions of the pooling layer have been proposed to enhance its capabilities and flexibility. Fractional max-pooling enables adaptive downsampling by randomly selecting fractional indices within pooling regions, promoting robustness to spatial translations and distortions [41]. Global average pooling computes the spatial average of each feature map, reducing spatial dimensions to a single value per feature, thereby reducing the algorithm parameters and enhancing interpretability. Additionally, learnable pooling mechanisms, such as adaptive pooling and dynamic pooling, enable the network to adaptively learn pooling regions based on input data characteristics, improving feature representation and discriminative power [40].

Optimizing the pooling layer is essential for effective algorithm training and performance [43]. Pooling operations typically do not involve trainable parameters, but optimization strategies such as padding and strides influence the spatial dimensions of the output feature maps. Proper selection of pooling region size, padding, and strides can significantly impact algorithm performance, spatial resolution, and computational

efficiency. In addition, the dropout and batch normalization regularization techniques help prevent overfitting and improve generalization.

The pooling layer is a critical component of CNNs, enabling effective feature reduction and spatial summarization in various computer vision tasks [39]. In image classification, pooling layers downsample feature maps while preserving important visual cues, facilitating hierarchical feature learning and discriminative representation. Object detection frameworks like R-CNN and SSD utilize pooling layers for region proposal generation and feature extraction, enabling accurate object localization and classification. Semantic segmentation networks employ pooling layers to reduce spatial dimensions while maintaining contextual information, facilitating dense pixel-wise predictions and scene understanding [18].

2.2.4 Fully Connected Layer

The fully connected layer is a vital component of CNNs, responsible for integrating high-level features extracted by convolutional layers and making predictions for classification or regression tasks [44].

The fully connected layer serves as the decision-making component of CNNs, transforming the high-dimensional feature representations extracted by preceding convolutional layers into class probabilities or regression outputs [45]. Depicted in Fig. 2.5, each neuron in the fully connected layer is connected to every neuron in the previous layer, forming dense connections that allow the network to capture complex relationships in the data. By applying non-linear activation functions to the weighted sum of inputs, the fully connected layer introduces non-linearities essential for modelling intricate patterns and decision boundaries.

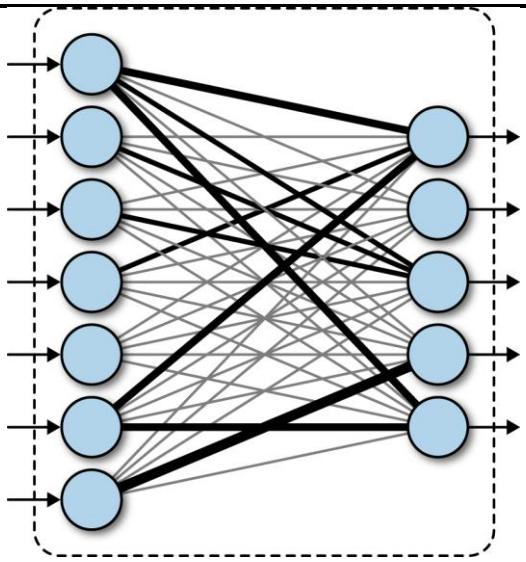

Fig. 2.5. A fully connected layer in a deep network [45]

The architecture of the fully connected layer is characterized by the number of neurons or units, which determines the dimensionality of the output space. The choice of the number of neurons in the fully connected layer is often guided by the complexity of the task and the desired algorithm capacity [46]. In classification tasks, the number of neurons in the fully connected layer corresponds to the number of output classes, whilst in regression tasks, a single neuron may suffice to produce continuous predictions. Dropout and weight decay regularization techniques are commonly applied to prevent overfitting and improve generalization performance [47].

Optimizing the fully connected layer is crucial for efficient algorithm training and performance. SGD with momentum and adaptive learning rate methods like Adam and RMSprop are commonly used optimization algorithms for updating the weights of fully connected layers [26]. Initialization schemes such as Xavier and He [27] initialization ensure stable training dynamics and prevent saturation or vanishing gradients. Batch normalization stabilizes activations within each mini-batch, accelerating convergence and improving gradient flow.

The fully connected layer finds applications across diverse domains, including object detection, image classification, reinforcement learning, and natural language processing [44]. In image classification tasks, the fully connected layer produces class probabilities based on extracted features, enabling accurate classification of objects within images. Object detection frameworks like Faster R-CNN utilize fully connected layers for bounding box regression and region-based classification, enabling precise object localization and recognition. In natural language processing, fully connected layers are employed in recurrent neural networks (RNNs) and transformer architectures for sequence classification and language modelling, facilitating semantic understanding and context-based predictions.

2.3 Object Detection Algorithms

Recently, the field of object detection has observed significant advancements, particularly due to the emergence of deep learning techniques [1]. These advancements have led to the categorization of popular object detection methods into two main groups, namely one-stage detectors and two-stage detectors. The architecture of these two categories is illustrated in Fig. 2.6.

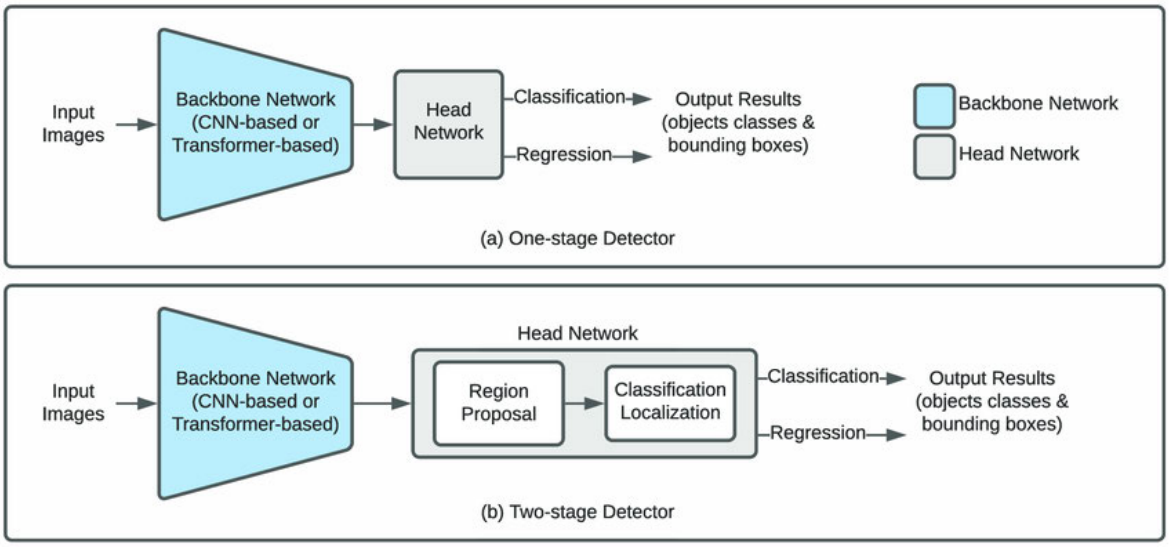

Fig. 2.6. One-stage vs. two-stage detectors [48]

One category encompasses algorithms like R-CNN, Fast R-CNN, and Faster R-CNN, which rely on a two-stage process based on region proposal. Initially, these methods utilize heuristic techniques such as Selective Search [49] or CNNs to generate region proposals. Subsequently, they conduct classification and regression tasks on these proposed regions.

The other category consists of one-stage algorithms such as YOLO and SSD. These algorithms directly predict the categories and positions of objects using a single CNN network. Unlike the two-stage approach, they omit the intermediate phase of region extraction and instead execute target classification, feature extraction and position regression directly inside the convolutional network [48].

2.3.1 Faster R-CNN

Object detection is a fundamental task in computer vision, enabling machines to identify and localize objects within images or video frames. Faster R-CNN has emerged as

a significant milestone in the field, offering a unified framework that combines region proposal generation and object detection within a single deep learning architecture.

Faster R-CNN, proposed by Shaoqing Ren et al. in 2015 [9], introduces a two-stage object detection framework that integrates a Region Proposal Network (RPN) [50] accompanied by a Fast R-CNN [51] detector. A basic structure of the Faster R-CNN object detector algorithm is illustrated in Fig. 2.7. The architecture consists of the following key components:

- **Region Proposal Network (RPN):** The RPN is a fully convolutional network that operates on a feature map generated by the CNN backbone network, namely VGG [52] or ResNet [53]. Bounding boxes (region proposals) are created for possible objects by moving an anchor over the feature map and predicts the bounding box offsets and objectness scores for every anchor.
- **Fast R-CNN Detector:** Once the region proposals are generated by the RPN, it undergoes object classification and bounding box refinement by being passed through the Fast R-CNN detector. The detector extracts features from each region proposal using RoI (Region of Interest) [54] pooling and inserts it into fully connected layers for bounding box regression and object classification.

Faster R-CNN is trained end-to-end using a multi-task loss function by combining classification loss, bounding box regression loss, and anchor classification loss. The network is optimized using SGD or other advanced optimization algorithms [26]. During training, the RPN and Fast R-CNN share convolutional layers, allowing them to be jointly fine-tuned for improved performance [19].

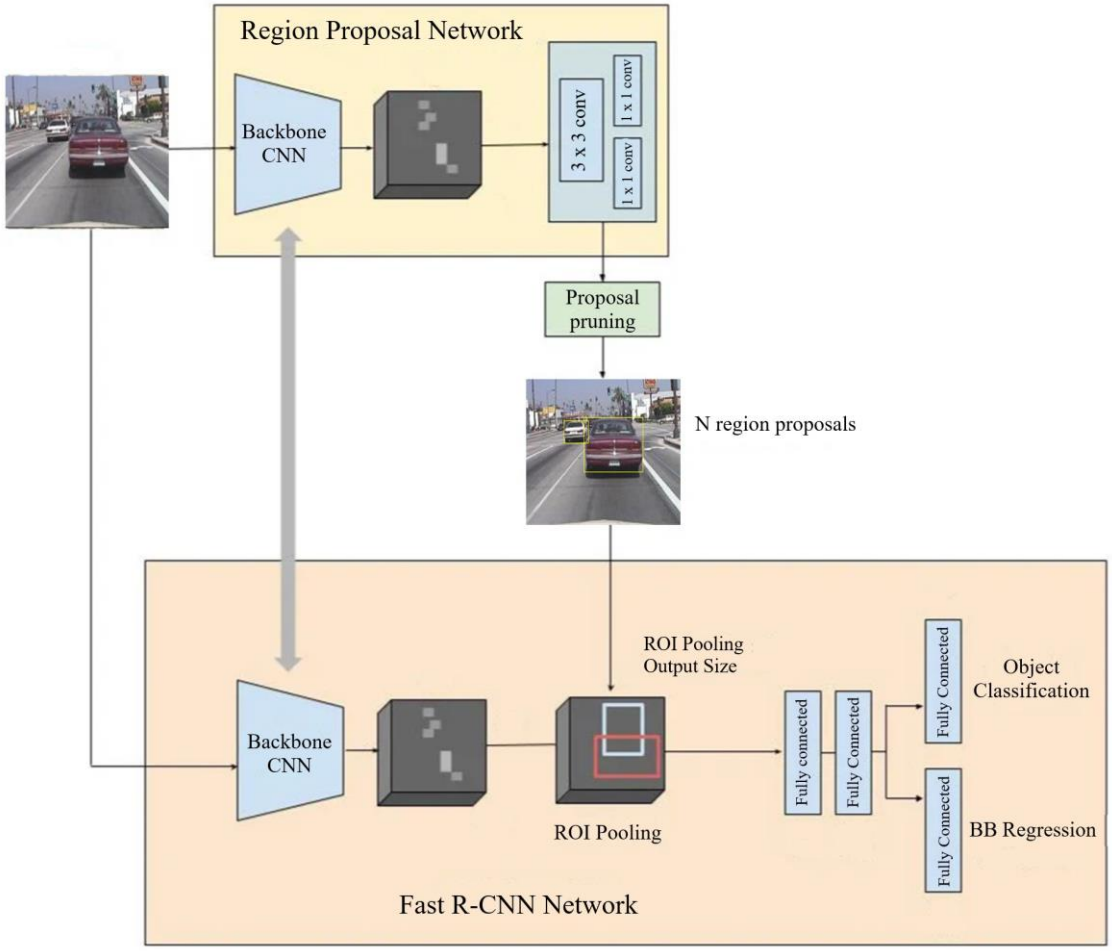

Fig. 2.7. Structure of Faster R-CNN for Vehicle Detection [9]

Faster R-CNN has demonstrated impressive performance on benchmark datasets such as PASCAL VOC [55] and COCO [56], achieving high accuracy in object detection tasks while maintaining fast inference speeds. Evaluation metrics include mean average precision (mAP) and processing time per image [57]. Compared to previous approaches, Faster R-CNN significantly reduces computation time by eliminating the requirement for external region proposal methods and enables end-to-end learning.

From its introduction, Faster R-CNN spurred numerous expansions and improvements into the research of object detection [58 – 59]. Some significant improvements include:

- **Feature Pyramid Networks (FPN):** The integration of FPN architectures and Faster R-CNN enhanced its capability to detect an object at varied scales. This allowed for an improvement in performance on both larger and smaller objects.
- **Attention Mechanisms:** The integration of attention mechanisms with Faster R-CNN lead to improved robustness and localization accuracy by allowing it to pay close attention to applicable regions within the image being passed into the network.
- **Real-Time Implementation:** There are on-going endeavours being made by researchers for the optimization of Faster R-CNN object detection application. These efforts will enable efficient performance on devices with limited resources for real-time application.

2.3.2 YOLO v3

YOLO, recognised for its high level of accuracy and real-time performance, is a pioneering object detection algorithm. YOLO v3, the third iteration of the YOLO series, represents a significant improvement over its predecessors in terms of both speed and detection accuracy.

YOLO v3, introduced by Redmon and Farhadi in 2018 [10], builds upon the previous versions of YOLO by the introduction of numerous key architectural advancements. The fundamental framework of a YOLO v3 object detector algorithm is illustrated in Fig. 2.8. The architecture of YOLO v3 can be summarized as follows:

- **Darknet-53 Backbone:** YOLO v3 utilizes a 53-layer deep CNN known as Darknet-53 [60] which is pre-trained using the ImageNet dataset. It can perform object detection at varied scales by serving as a feature extractor in the network.

- **Feature Pyramid Network (FPN):** YOLO v3 incorporates a feature pyramid network to improve object detection at varied scales. This enables the network to detect small objects more effectively whilst sustaining a high degree of accuracy for larger objects.
- **Multiple Detection Scales:** YOLO v3 predicts bounding boxes at three different scales, allowing it to detect objects of varying sizes. To improve the performance of object detection, every scale corresponds to a particular set of anchor boxes that undergoes optimisation during training.

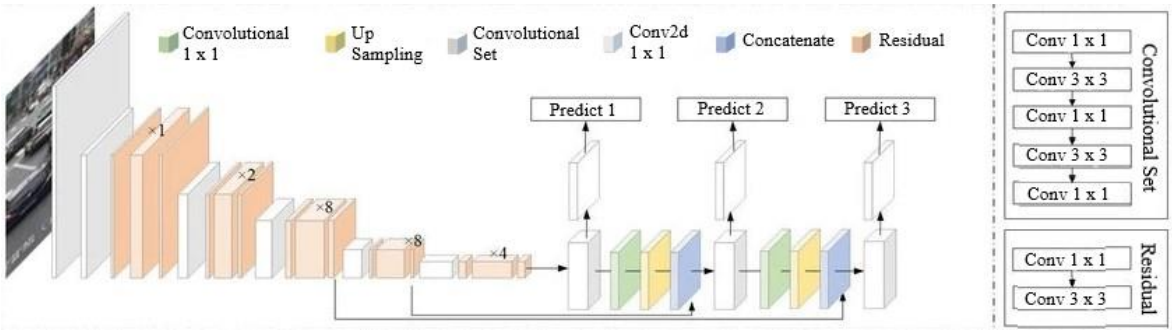

Fig. 2.8. Structure of YOLO v3 [10]

YOLO v3 is trained end-to-end using a multi-task loss function that combines classification loss, objectness loss, and bounding box regression loss. The network is optimized using SGD or other advanced optimization techniques [26]. During training, YOLO v3 learns to predict object probabilities, bounding box coordinates, and class labels directly from raw pixel values.

YOLO v3 has demonstrated impressive performance on benchmark datasets such as COCO and PASCAL VOC. It achieves high accuracy in object detection tasks while maintaining real-time inference speeds. Evaluation metrics include mean average precision

(mAP) and processing time per image [57]. Compared to previous versions of YOLO, YOLO v3 achieves significant improvements in both accuracy and speed.

From its introduction, YOLO v3 has inspired multiple expansions and improvements to the field of object detection [61 – 63]. A few significant developments consist of:

- **YOLOv3-tiny:** Designed as a lightweight version of YOLO v3 for resource-constrained devices with real-time applications.
- **YOLOv3-spp:** Incorporated spatial pyramid pooling to enhance detection accuracy and feature representation.
- **YOLOv3-tiny-prn:** Incorporated a partial residual network to YOLOv3-tiny to improve the performance of object detection.

2.3.3 SSD

SSD is a prominent object detection algorithm renowned for its efficiency and accuracy. It stands out as a unified framework capable of detecting objects in real-time while maintaining high detection accuracy across various scales.

SSD, proposed in 2016 by Wei Liu et al. [11], is characterized by its single-shot detection approach. This concurrently predicts class probabilities and object bounding boxes over several feature maps. The basic structure of the SSD object detector algorithm is illustrated in Fig. 2.9. The architecture of SSD can be summarized as follows:

- **Base Convolutional Network:** SSD utilizes a base convolutional network (e.g., ResNet, VGG) which extracts feature maps from input images [64]. These feature maps are subsequently processed by further convolutional layers to predict class scores and bounding boxes at different spatial resolutions.

- **Multi-scale Feature Maps:** SSD can perform object detection on objects varying in size by incorporating feature maps at varied scales. This is achieved through the application of convolutional layers using different kernel strides and sizes to the input feature maps, allowing it to detect objects at various degrees of granularity.
- **Default Boxes (Anchor Boxes):** SSD predicts bounding boxes using a set of default or anchor boxes, defined at various scales and aspect ratios. The default anchor boxes produce reference templates for the prediction of object locations and sizes.

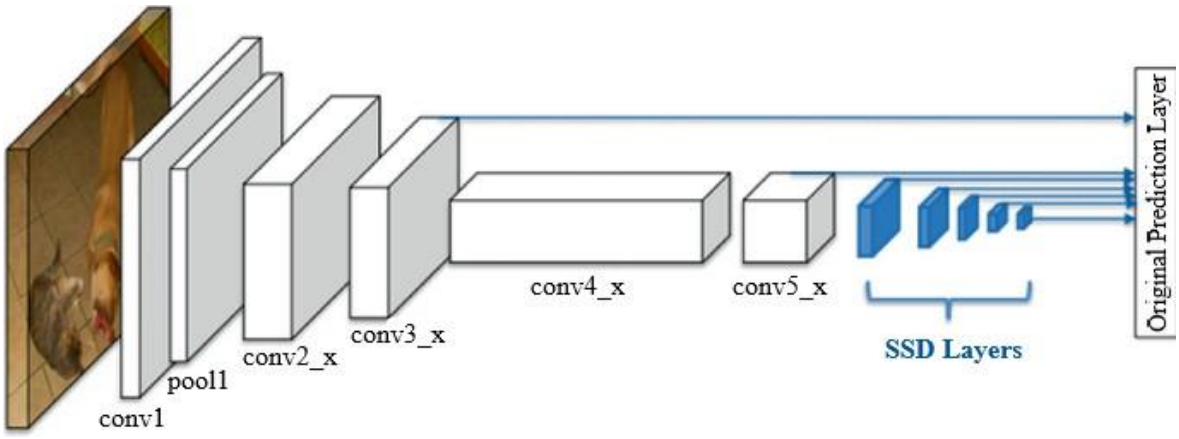

Fig. 2.9. Structure of SSD [11]

SSD is trained end-to-end using a combination of localization loss such as Smooth L1 loss and classification loss (cross-entropy loss). The network is optimized using SGD or other advanced optimization techniques. During training, SSD learns to predict object bounding boxes and class probabilities directly from input images, without the need for region proposal networks or multiple stages of inference [26].

SSD has demonstrated impressive performance on benchmark datasets such as COCO and PASCAL VOC. It achieves high accuracy in object detection tasks across a wide range of object categories while maintaining real-time inference speeds. Evaluation metrics include mean average precision (mAP) and processing time per image [57]. Compared to other object detection methods, SSD offers a suitable trade-off between speed and accuracy.

From its introduction, SSD inspired multiple expansions and improvements in the field of object detection [65 – 67]. A few significant developments consist of:

- **Improved Feature Extraction:** FPN and residual connections are advanced feature extraction techniques that have been explored by researchers in efforts to improve detection performance and increase the representational power of SSD.
- **Efficient Backbones:** To enable efficient inference without compromising detection accuracy on resource-constrained devices, lightweight backbone architectures have been developed for SSD.
- **Domain Adaptation:** To facilitate real-world deployment across various datasets and environments, multiple domain adaptation techniques have been proposed for the improvement of SSD generalization capabilities.

2.4 Summary

This chapter comprehensively reviewed relevant literature on CNNs and object detection algorithms which serves as a foundation for the work in this thesis. There was a need to evaluate the performance of CNN-based object detection algorithms specifically in vehicle image processing tasks, an area of growing importance for applications like autonomous driving. Despite the success of existing algorithms, questions remain about

their comparative performance when tailored to specific domains, such as vehicle detection. While many object detection algorithms are optimized for either high accuracy or real-time performance, a thorough comparative analysis of algorithms like Faster R-CNN, YOLO v3, and SSD in MATLAB® have not been extensively documented, particularly when applied to vehicle images.

The convolutional layer, activation layer, pooling layer and fully connected layer are critical components of CNNs and deep learning architectures. By leveraging principles of downsampling and pooling operations, weight sharing, spatial convolution, architectural variations, activation functions, and optimization strategies, researchers can design and train CNNs with improved performance, stability, and generalization capabilities across various computer vision tasks. By integrating high-level features extracted by convolutional layers and applying non-linear transformations, the fully connected layer enables CNNs to model complex relationships and make accurate predictions across various domains.

Faster R-CNN offers a unified framework that achieves high accuracy and efficiency. Its two-stage architecture, comprising a Region Proposal Network and Fast R-CNN detector, has become a cornerstone in modern object detection systems.

YOLO v3 offers a powerful combination of accuracy and speed. Its architectural enhancements, training methodology, and performance improvements have established it as a benchmark for real-time object detection systems.

SSD offers a powerful combination of efficiency and accuracy. Its single-shot detection paradigm, coupled with multi-scale feature maps and default boxes, enables robust detection of objects across various sizes and aspect ratios.

CHAPTER 3 Research Methodology

3.1 Introduction

Thomas Schwandt [68] defines research methodology as a guideline of how research should be undertaken. It involves the analysis of assumptions, theories, and procedures in a particular approach to a topic. A comparative study performs an analysis and comparison between two or more objects or theories/topics [69]. A quantitative approach to research involves the collection and analysis of numerical data to test hypotheses and answer research questions [70]. This method employs statistical techniques to analyse data which allows researchers to draw conclusions and make generalizations about the results. Therefore, a quantitative approach was implemented in this research study with priority given to the object detection algorithms that was selected.

This chapter details the steps taken to meet the research objectives and the systematic approach used in this study is through experimentation. The mathematical methodology undertaken in this research are reflected in both the conference paper presented at the International Conference on Electrical, Computer and Energy Technologies (ICECET) which was published in IEEE Xplore after peer review [71] as well as the peer reviewed article published in the MDPI Journal of Imaging for the Computer Vision and Image Processing topic (2nd edition) [72]. Publication of these papers after peer review cements the validity of the approach taken in this research.

3.2 Research Instruments

The results for the research were obtained using MATLAB® R2022b to develop and execute the object detection algorithms. The experiments were carried out on an HP ZBook with a 11th Generation Intel® Core™ i7-11850H @ 2.50 GHz. The laptop ran on a Windows 10 operating system with 32 GB of RAM. The parallel computing platform and application programming interface used by MATLAB® was "CUDADevice" which allowed it to use the onboard NVIDIA T1200 Laptop GPU card. This allowed 3.31 GB of available dedicated memory at a clock rate of 1.56 GHz. Table 3.1 details the GPU device information that was used in MATLAB® to evaluate the object detection algorithms.

Table 3.1 MATLAB® GPU Device Information

CUDADevice Properties	
Name	NVIDIA T1200 Laptop GPU
Compute Capability	7.5
Driver Version	11.7
Toolkit Version	11.2
Max Threads Per Block	1024
Max Thread Block Size	[1024 1024 64]
Total Memory	4.29 GB
Available Memory	3.31 GB
Multiprocessor Count	16
Clock Rate	11560000 KHz

3.3 Dataset

The dataset created for the purposes of this research was consolidated using two datasets:

- **Caltech Cars 1999:** Formulated by Weber and Perona [73], this dataset contains 126 images of the rear end of vehicles with approximate scale normalisation and a size of 896 x 592 pixels.
- **Caltech Cars 2001:** Formulated by Philip, Updikeand and Perona [74], this dataset contains 526 images of the rear end of vehicles with no scale normalisation and a size of 360 x 240 pixels. Repeat images are included in the dataset.

For this research, the created dataset included a total of 652 images. The dataset was separated into a training dataset and test dataset using a ratio of 3:2. This resulted in 391 images used for training and 261 images used for testing. The format of the images in the dataset was Joint Photographic Experts Group (JPEG). Examples of the dataset images utilised in this research is illustrated in Fig. 3.1. Zhu et al [75] conducted an experiment regarding massive training datasets in the object detection community and concluded that given the size of existing datasets that detectors are pretrained on, none of the algorithms would benefit greatly from large datasets during evaluation. This research aimed to evaluate existing object detector algorithms and hence a small dataset was useful. This is evident in literature that there is no required minimum dataset size for evaluating detector algorithms, however, it was observed that the training dataset was on average 60% - 70 % greater than the test dataset [76 – 78].

Fig. 3.1. Dataset sample

The "Image Labeler" application in MATLAB® was used to generate the ground truth data for the dataset. This consisted of bounding boxes encompassing vehicles and labelled images. RoI labels was created by bounding boxes in the form [***x y w h***], whereby:

- ***x*** : Bounding box left corner x-coordinate
- ***y*** : Bounding box top y-coordinate
- ***w*** : Bounding box width (x-axis length)
- ***h*** : Bounding box height (y-axis length)

Bounding box label samples generated in MATLAB® for the dataset images is illustrated in Fig. 3.2.

Fig. 3.2. Dataset sample with generated bounding box labels

Every image comprises of either one or two labelled instances of a vehicle. Unzipping of the vehicle images and loading of the dataset ground truth data is the first step in MATLAB®. A two-column table is used to store the vehicle data, where column one contains the file path of the vehicle image and column two contains the bounding boxes of the vehicle. As mentioned in Section 3.3 (p. 31), the dataset needed to be split into training and test sets. 60% of the data was used for training and the remaining 40% was used for testing the trained object detector. MATLAB® functions *boxLabelDatastore* and *imageDatastore* was used to create the training and test datastores for loading label and image data. The image and box label datastores were combined using the *combine* function and resulted in the consolidated dataset used for training and testing. Fig. 3.3 illustrates the major steps in creating the datastore.

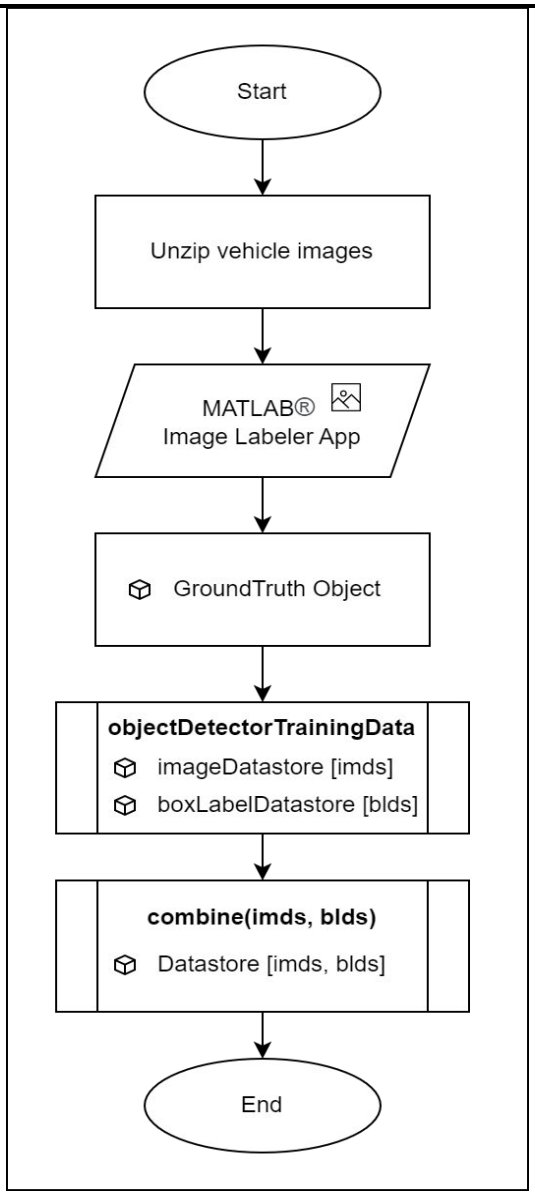

Fig. 3.3. Major steps in creating the datastore

3.4 Data Pre-Processing and Augmentation

Before processing, all images were resized. This allowed for the computational cost to be reduced during the training phase of the object detection algorithms. Random images were also transformed using data augmentation, which allowed for further improvement to the detector's accuracy. Data augmentation allowed for increased diversity in the data used

for training without requiring more labelled training samples. In MATLAB®, the *transform* function was used to augment images by performing scaling, random flipping of the image and its box labels horizontally, and colour jitter which allows variations in contrast, hue, brightness and saturation of the image. The *transform* function used the functions below:

- ***randomAffine2d***: Allows for the creation of a randomized 2-D affine transformation. By parsing the *XReflection* as an input argument into the function, it performs random horizontal reflection.
- ***imwarp***: Applies geometric transformation to the image.
- ***bboxwarp***: Applies geometric transformation to the bounding boxes.

The test dataset is meant to represent original/raw data and was left untouched to enable an objective evaluation of the detector. Therefore, only the training dataset went through data augmentation and not the test dataset. A sample of data augmentation being performed on one of the images from the training dataset is illustrated in Fig. 3.4.

Fig. 3.4. Data augmentation on one image

Algorithm 1, shown in Fig. 3.5, outlines the steps used in MATLAB® for augmenting and pre-processing images.

Algorithm 1: Augmentation and Pre-processing

Input: Training image datastore

Output: Transformed datastore

1. Create an *ImageDatastore* for training images
2. Read the images into the workspace to create image files
3. **for** each image in the datastore **do**
4. Read each image
5. Randomly flip images and bounding boxes horizontally using *randomAffine2d*, *imwarp* and *bboxwarp*
6. Resize images and bounding boxes using *imresize* and *bboxresize*
7. **end for**

Fig. 3.5. Algorithm 1: Implementation of Data Augmentation and Pre-processing in MATLAB®

3.5 Experimental Models

This section details the techniques and methods used to train and evaluate the algorithms for object detection on a PC. Although the different object detection algorithms utilise varying methods, the algorithm structure follows a similar process. Fig. 3.6 depicts the major steps when applying the various object detection algorithms.

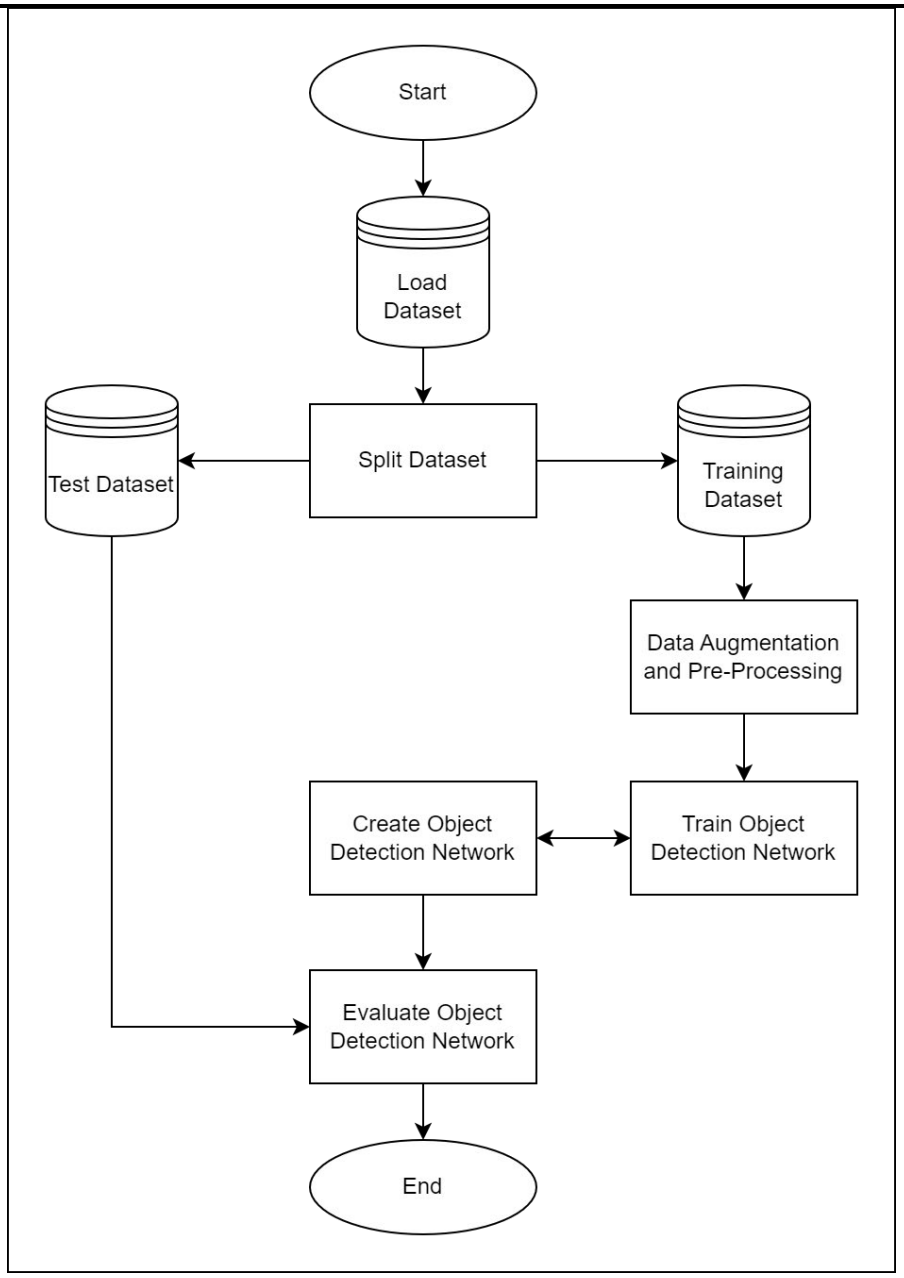

Fig. 3.6. Major steps in implementing object detection algorithms

3.5.1 Faster R-CNN

The Faster R-CNN algorithm improves on the Fast R-CNN algorithm which in turn was derived from the foundational R-CNN algorithm. The design of the Faster R-CNN algorithm in MATLAB® is illustrated in Fig. 3.7. The foundational R-CNN algorithm

begins with a pretrained network and is indicated by the blue circles in Fig. 3.7. The addition of a box regression layer in Fast R-CNN allows the network to learn a set of box offsets, therefore improving object detection. The insertion of an ROI pooling layer allows CNN features to be pooled for each region proposal within the network. These layers are indicated by the red circles in Fig. 3.7. Faster R-CNN adds the RPN to produce the region proposals instead of retrieving the proposals from an external algorithm and is indicated by the green circles in Fig 3.7.

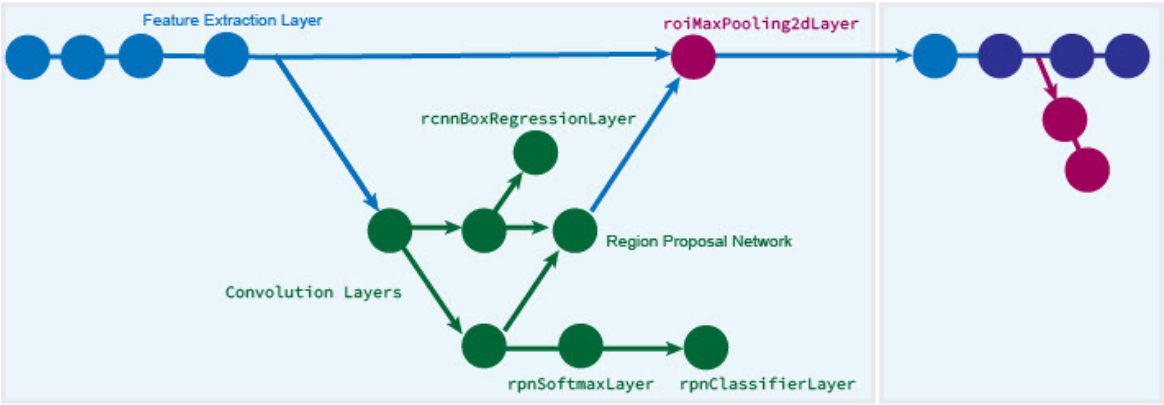

Fig. 3.7. Design of Faster R-CNN [79]

The feature extraction network is usually a pretrained CNN and ResNet-50 [80] was selected for feature extraction in this study. MATLAB® allows you to automatically create a Faster R-CNN network using the *fasterRCNNLayers* function. *fasterRCNNLayers* required the following input parameters to create the Faster R-CNN network:

- Network input size
- Anchor boxes
- Feature extraction network

When choosing the network input size, a few things needed to be taken into consideration. This included the minimum size required to run the network, the size of the

training images, and the computational cost needed to process the images at the selected size. For this research, a minimum network input size of [224 224 3] is required to run the network and was selected to be the network input size used in order to decrease the computational cost of running the network. It is noted that the dataset used for training was greater than the selected network input size of 224 x 224 and these images were resized as part of the pre-processing step discussed in Section 3.4. Thereafter, the *estimateAnchorBoxes* function was used to estimate the anchor boxes based on the size of the images in the training dataset.

The feature extraction layer was set to 'activation_40_relu' as this layer outputs feature maps that are downsampled by a factor of 16. This provides a good trade-off between spatial resolution and the strength of the extracted features [81]. The cost of spatial resolution increases when features are extracted further down the network, even though the network can encode stronger image features.

During training of the Faster R-CNN network, the t*rainFasterRCNNObjectDetector* function was used to specify the network training options. These parameters are explained in detail in Section 3.6.

Algorithm 2, shown in Fig. 3.8, details the steps undertaken in MATLAB® for the manual creation of the object detector using Faster R-CNN.

Algorithm 2: Faster R-CNN

1. Load a pretrained network
2. Remove the last 3 classification layers
3. Define and add new object classification layers
4. Define the number of outputs of the fully connected layer
5. Create and add the box regression layers
6. Connect the regression layers to the *avg_pool* layer
7. Disconnect the layers attached to the feature extraction layer
8. Add a ROI max pooling layer
9. Connect feature extraction layer to ROI max pooling layer
10. Create the region proposal layer
11. Define the number of anchor boxes and feature maps
12. Connect to RPN to feature extraction layer
13. Add RPN classification and regression layers
14. Connect the layers to the RPN network

Fig. 3.8. Algorithm 2: Implementation of Faster R-CNN in MATLAB®

3.5.2 YOLO v3

The YOLO v3 algorithm initiates deep learning by using a feature extraction network as its base network layer, which comprises of either an un-trained or pre-trained CNN. It is worth noting that only a pre-trained network can be used to perform transfer learning. Detection subnetworks are created by adding ReLU, batch normalisation, and convolution layers. The detection network source is created by output layers which connect as inputs to the detection subnetworks. This design of YOLO v3 in MATLAB® is illustrated in Fig. 3.9.

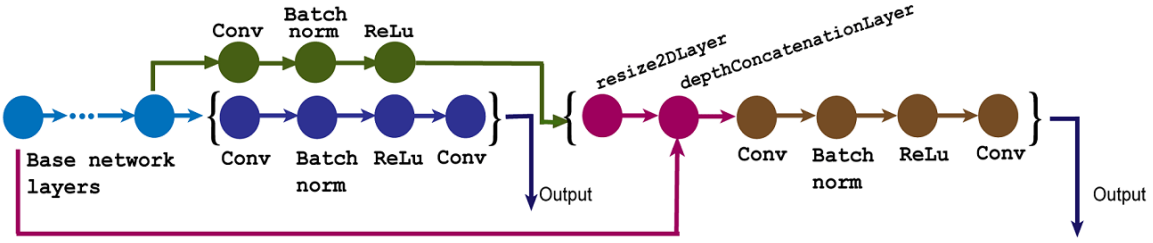

Fig. 3.9. Design of YOLO v3 [79]

For the purposes of this research, the YOLO v3 algorithm is based off SqueezeNet [82]. In SqueezeNet the feature extraction network is used in conjunction with two detection heads at the end. The YOLO v3 network can be created automatically in MATLAB® with the use of the *yolov3ObjectDetector* function. *yolov3ObjectDetector* required the following input parameters to create the YOLO v3 network:

- Network input size
- Anchor boxes
- Base network

As with Faster R-CNN, a few things needed to be taken into consideration when choosing the network input size, this includes the minimum size required to run the network, the size of the training images, and the computational cost needed to process the images at the selected size. For this research, a minimum network input size of [227 227 3] is required to run the network and was selected to be the network input size used to decrease the computational cost of running the network. It is noted that the dataset used for training were larger than the selected network input size of 227 x 227 and these images were resized as part of the pre-processing step discussed in Section 3.4. To attain a good trade-off between the mean IoU and number of anchors, the total number of anchor boxes was set to 6. Thereafter, the *estimateAnchorBoxes* function was used to estimate the anchor boxes based on the size of the images in the training dataset. The anchor boxes were then

specified to be used in both the detection heads. A cell array of value $[\boldsymbol{M} \times \mathbf{1}]$ illustrates *anchorBoxes* in MATLAB®, whereby the amount of detection heads is denoted by M. Each detection head consists of a $[\boldsymbol{N} \times \mathbf{1}]$ matrix of anchors, whereby the amount of anchors to be used is denoted by N. The detection network source was set to 'fire5-concat' and 'fire9-concat' layers [83].

During training of the YOLO v3 network, the *minibatchqueue* and *sgdmupdate* functions were used to specify the network training options. These parameters are explained in detail in Section 3.6.

Algorithm 3, shown in Fig. 3.10, details the steps undertaken in MATLAB® for the manual creation of the object detector using YOLO v3.

Algorithm 3: YOLO v3

1. Load a pretrained network
2. Inspect the base network architecture using *analyzeNetwork*
3. Train the YOLO v3 network by specifying the anchor boxes and the classes to use
4. Select two feature extraction layers in the base network to serve as the source for detection subnetwork
5. Add detection heads to the feature extraction layers of the base network to create the YOLO v3 detector
6. Specify the model name, anchor boxes, and classes
7. Use *analyzeNetwork* to inspect the YOLO v3 deep learning network

Fig. 3.10. Algorithm 3: Implementation of YOLO v3 in MATLAB®

3.5.3 SSD

The SSD algorithm initiates deep learning by using a feature extraction network as its base network layer, which comprises of either an un-trained or pre-trained CNN. It is worth noting that only a pre-trained network can be used to perform transfer learning. Any layer within the feature extraction network can be utilised for the selection of prediction layers. Prediction layer outputs are fed into regression and classification branches. Regression branches are merged together from each prediction layer. This merger is then connected to a RCNN box regression layer from which bounding box loss can be calculated. Classification branches are merged together from each prediction layer. This merger is then subsequently connected to the SoftMax and Binary Cross-Entropy (BCE) layers from which classification loss can be calculated. This design of SSD in MATLAB® is illustrated in Fig. 3.11.

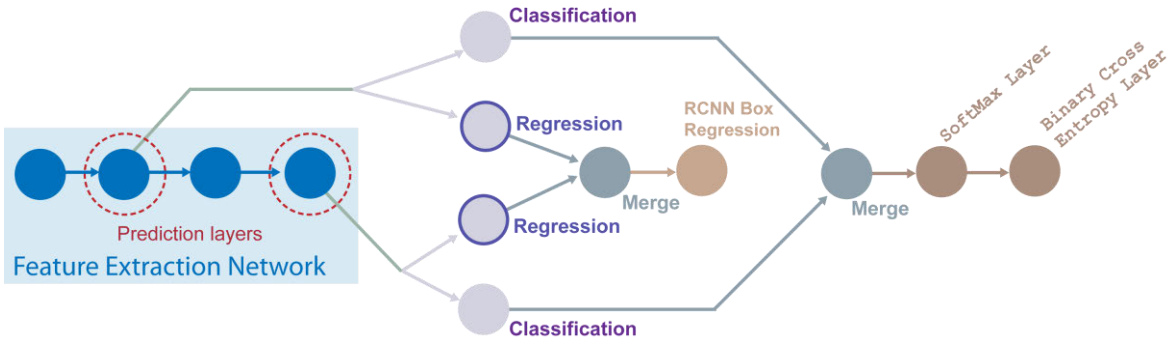

Fig. 3.11. Design of SSD [79]

The MATLAB® function, *ssdObjectDetector*, allows for the automatic creation of the SSD network. *ssdObjectDetector* requires several input parameters to create the SSD network, namely the input size, anchor boxes, class names, detection network sources, and the feature extraction network serving as the base network. The feature extraction network is usually a pretrained CNN. ResNet-50 was selected for feature extraction in this study.

As with the previous two detectors, multiple considerations needed to be taken into account when choosing the network input size, including the minimum size required to run the network, the size of the training images, and the computational cost needed to process the images at the selected size. For this research, a minimum network input size of [300 300 3] is required to run the network and was selected to be the network input size used to decrease the computational cost of running the network. It is noted that the dataset used for training was greater than the selected network input size of 300 x 300 and these images were resized as part of the pre-processing step discussed in Section 3.4.

The feature extraction layer was set to 'activation_40_relu' as this layer outputs feature maps that are downsampled by a factor of 16. This provides a good trade-off between spatial resolution and the strength of the extracted features [81]. The cost of spatial resolution increases when features are extracted further down the network, even though the network can encode stronger image features. Thereafter, convolutional layers corresponding to the ReLU layers were added to make the backbone network more robust. The *estimateAnchorBoxes* function was then used to estimate the anchor boxes based on the size of the images in the training dataset.

During training of the SSD network, the *trainSSDObjectDetector* function was used to specify the network training options. These parameters are further explained in Section 3.6.

Algorithm 4, shown in Fig. 3.12, details the steps undertaken in MATLAB® for the manual creation of the object detector using SSD.

Algorithm 4: SSD

1. Load a pretrained network
2. Display the base network architecture using *analyzeNetwork*
3. Specify the class names and anchor boxes to use for training
4. Specify the names of the feature extraction layers in the base network to use as the detection heads
5. Use the specified base network and detection heads to create the SSD object detector.

Fig. 3.12. Algorithm 4: Implementation of SSD in MATLAB®

3.6 Tuning and Training Parameters

Tuning and training parameters for object detection algorithms involve adjusting various settings and configurations to optimise performance and enhance accuracy. This process typically includes fine-tuning hyperparameters such as batch sizes, learning rates and optimisation algorithms to improve convergence and prevent overfitting. Iterative experimentation and validation on training and test datasets are essential for identifying optimal parameter configurations and ensuring robust performance of object detection algorithms.

3.6.1 Minibatch Size

Minibatch size refers to the number of data samples (instances) used in one iteration of training to calculate the gradients and parameter updates. Instead of using the entire dataset at once (batch gradient descent), minibatch training divides the dataset into smaller subsets or minibatches. The choice of minibatch size depends on multiple factors such as the size of the dataset, computational resources, and the architecture of the neural

network. Typically, larger minibatch sizes may lead to faster convergence but requires greater memory and computational power. Conversely, smaller minibatch sizes may provide more noise in the gradient estimates but can help the algorithm generalize better. The algorithms were tested with minibatch sizes of 2, 4 and 8. Testing with these minibatch sizes ensured that the algorithms could be trained effectively on a PC with limited GPU and memory capacity, which was a key focus of this research.

3.6.2 Number of Epochs and Number of Iterations

An epoch is defined as one complete pass through the entire dataset during the training phase. The number of epochs is the hyperparameter that defines the number of times the learning algorithm will run through the complete training dataset. An iteration, also known as a training step, refers to one update of the algorithm's parameters during the training process. Each iteration typically involves processing one minibatch of data through the algorithm and updating the algorithm’s parameters based on the computed gradients. Multiple epochs and one iteration were employed during training. The algorithms were tested with a maximum of 10, 20 and 80 epochs. Testing at shorter training periods reduces the computational load, which is important for PCs with limited processing power. This aligned with the research focus on assessing the efficiency of these algorithms in resource-constrained environments.

3.6.3 Learning Rate

The learning rate is a hyperparameter that controls the step size or rate at which the algorithm’s parameters are updated during the training process. It determines how much the algorithm parameters are adjusted in response to the estimated gradient of the loss

function. For comparison purposes in this study, the standard learning rate of 0.001 was used.

3.6.4 Activation Function

An activation function is a mathematical operation applied to the output of a neuron in a neural network. This enables it to learn complex patterns and relationships in the data by introducing non-linearity into the network.

The sigmoid function defined by equation (3.1) is a smooth, S-shaped curve that squashes the input values into the range [0, 1]. It is often used in binary classification problems where the output needs to be interpreted as a probability.

$$\boldsymbol{S(x) = \frac{1}{1+e^{-x}}} \tag{3.1}$$

Where:

- $\boldsymbol{S(x)}$: Sigmoid function
- $\boldsymbol{e}$: Euler's number

ReLU defined by equation (3.2) is a piecewise linear function that returns the input if it is positive and zero otherwise. It is the most commonly used activation function in deep learning due to its simplicity and effectiveness in training deep neural networks.

$$\boldsymbol{f(x) = max(0, x)} \tag{3.2}$$

Softmax defined by equation (3.3) is used in the output layer of classification algorithms to transform the raw output scores into probability distributions over multiple classes. It ensures that the sum of the probabilities for all classes is equal to one.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad (3.3)$$

Where:

- σ : Softmax
- $\vec{z}$: Input vector
- K : Total number of elements in the vector
- e^{z_i} : Standard exponential function for the input vector
- e^{z_j} : Standard exponential function for the output vector

3.6.5 Loss Function

A loss function is a measure of the algorithm's performance or error during the training phase. It quantifies how well the predicted output of the algorithm matches the true target values in the training data. The aim during the training process is to minimise the value of the loss function, which indicates that the algorithm's predictions are as close as possible to the true targets. In order to help improve detection accuracy the loss function used for training is separated into binary cross-entropy for object classification and mean squared error (MSE) for bounding box regression. The loss functions: localization, confidence, classification and total loss is calculated by usings equations (3.4) - (3.7) respectively.

$$L_{loc} = \lambda_i \sum \left[(b_x - t_x)^2 + (b_y - t_y)^2 + \left(\sqrt{b_w} - \sqrt{t_w}\right)^2 + \left(\sqrt{b_h} - \sqrt{t_h}\right)^2 \right] \quad (3.4)$$

$$L_{conf} = \sum \left[c\, BCE(c_p, c_t) + (1 - c) BCE(c_{p,c_t}) \right] \quad (3.5)$$

$$L_{cls} = \lambda_c \sum [c_i\, CE(p(i), p(i)_{true})] \quad (3.6)$$

$$L_{total} = L_{loc} + L_{conf} + L_{cls} \quad (3.7)$$

Where:

- λ_c : Hyperparameter that controls classification loss
- λ_i : Hyperparameter that controls localization loss

- c_p : Predicted confidence score
- c_t : Ground truth label
- $p(i)$: Predicted class probability for class i
- $p(i)_{true}$: Ground truth class label
- BCE : Binary cross-entropy loss function
- CE : Categorical cross-entropy loss function

3.7 Evaluation Methods

mAP is one of the most widely used metrics for evaluating object detection performance and offers a comprehensive measure of algorithm performance, enabling fair comparisons between the different algorithms [57]. The calculation of mAP involves multiple steps, including the computation of recall and precision for every class, the creation of a precision-recall (PR) curve, as well as the calculation of the average precision (AP) throughout every class.

The algorithm's ability to recognize objects of significance is known as precision. This is calculated by using the True Positive (TP) and False Positive (FP). Precision is calculated by using equation (3.8).

$$P = \frac{TP}{(TP+FP)} \tag{3.8}$$

Recall is a fraction of relevant instances that are successfully retrieved and is calculated by using TP and False Negative (FN). Recall is calculated by using equation (3.9).

$$R = \frac{TP}{(TP+FN)} \tag{3.9}$$

AP is the metric that is calculated by using the area under the PR curve. The mAP is calculated by using equation (3.10).

$$\boldsymbol{mAP} = \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{i}=\mathbf{0}}^{\boldsymbol{N}} \boldsymbol{AP_i} \tag{3.10}$$

Where:

- $\boldsymbol{N}$: Total number of classes being evaluated
- $\boldsymbol{AP_i}$: AP in the $\boldsymbol{i}$th class

The MATLAB® function used to evaluate the precision metric for the object detection algorithms is a built-in function called *evaluateDetectionPrecision* which is based on equations (3.8) - (3.10) and returns the precision, recall and mAP values.

3.8 Summary

This chapter outlined the steps taken to achieve the objectives of developing and assessing performances of the three CNN object detection algorithms in MATLAB®. The first phase of the chapter covered the acquisition of the dataset. This was followed by data pre-processing and augmentation, algorithmic implementation of the object detection algorithms, hyperparameters for training, and finally the evaluation method employed. All of these processes were crucial for achieving the principal contribution of this research work. Furthermore, this allows the reader to replicate the methods discussed. The MATLAB® source code used in this research was adapted from MathWorks® [84] and is provided for reference in Appendix A – C.

CHAPTER 4 Results and Discussion

4.1 Introduction

This chapter presents the experimental results and their corresponding analysis to accomplish the objectives of this study, which is aimed at objectively measuring and comparing the performance of the three object detection algorithms. The results obtained from each of the three object detection algorithms are presented and discussed. Thereafter, section 4.5 compares the performance and overall results of the three algorithms used for object detection.

4.2 Faster R-CNN

Table 4.1 shows the hyperparameter settings used during the evaluation of the Faster R-CNN algorithm that provided optimal results. The algorithm was evaluated using 10 epochs and a batch size of 2. A batch size of 2 was chosen to accommodate memory constraints on the PC. The decision to cap training at 10 epochs was also influenced by practical considerations, particularly related to memory constraints. Larger batch sizes and epochs required more memory, slowed down the training process and eventually caused the MATLAB® application to shut down due to low GPU memory. A learning rate of 0.001 was used and the learning rate schedule was set to “piecewise” which allowed the network to update the learning rate periodically by multiplying it according to the Stochastic Gradient Descent with Momentum (SGDM) optimizer drop factor. SGDM enhances traditional stochastic gradient descent by incorporating momentum, which

accelerates convergence in the relevant direction and helps to smooth out fluctuations in the training process, leading to faster and more stable training.

Table 4.1 Faster R-CNN: Hyperparameter Settings

Hyperparameter	Value
Learning Rate	0.001
Learning Rate Schedule	Piecewise
Optimizer	SGDM
Batch Size	2
Epochs	10

Table 4.2 shows the average detection time of the Faster R-CNN algorithm. Average detection time varied across different epochs, ranging from 4.9 seconds to 5.3 seconds. These fluctuations indicated potential variations in the computational workload and efficiency of the Faster R-CNN algorithm at different stages of training due to the dynamic adjustment of the RPN as the algorithm learns over time. Despite fluctuations, the average detection time remained relatively consistent throughout the epochs, with small variations around the mean value of 5.1 seconds. This consistency indicated stable performance of the Faster R-CNN algorithm in terms of computational efficiency over the course of training.

Fig. 4.1 shows the detection time across the ten epochs for the Faster R-CNN algorithm. Significant spikes in the detection time were noted during the initial iteration of every epoch, ranging from 10 to 14 seconds. These spikes represented significant deviations from the average detection time observed in subsequent iterations within the same epoch. The spikes in the detection time for the initial iteration were due to the

initialisation processes which are the initialisation of the network weights, loading of data and compiling computational graphs. These tasks typically require extra time and computational resources as compared to future iterations.

Table 4.2 Faster R-CNN: Average Detection Time

Epoch	1	2	3	4	5	6	7	8	9	10
Time (s)	5.2	5	5.25	4.95	5.1	5	5.2	4.9	5.15	5.3

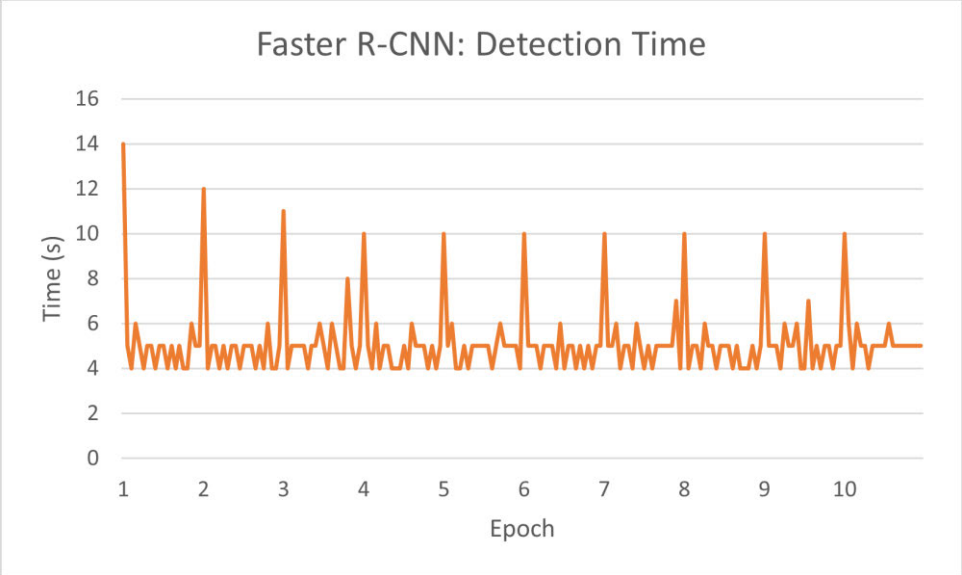

Fig. 4.1. Faster R-CNN: Detection Time

Table 4.3 shows the average detection loss of the Faster R-CNN algorithm. Average loss decreased over subsequent epochs, beginning at 1.453 in the first epoch and ending at 0.105 in the last epoch. The decreasing trend indicated that the Faster R-CNN algorithm was learning effectively from the training data by adapting its parameters to reduce the loss function, leading to improved performance. The most notable reduction in loss happened between epochs one and two, where the loss dropped from 1.453 to 0.766. This significant reduction indicated that the algorithm learned quickly from the initial training data and

made substantial improvements in its predictions. Subsequent epochs continued to show reductions in loss, however at a slower rate, indicating that the algorithm continued to refine its predictions and converge towards a better solution.

Fig. 4.2 shows the loss across the ten epochs for the Faster R-CNN algorithm. There were erratic spikes in loss for the first few epochs, with values ranging from 3.551 to 0.218. These spikes indicated high instability or variability within the training process during the initial iterations. Such spikes can be attributed to factors like random initialization of network weights, insufficient training data, or high learning rates causing large updates to the algorithm parameters. Erratic spikes in loss during the initial epochs are frequent in deep learning training processes and are often associated with the algorithm's exploration of the solution space and adjustment of network parameters. The erratic spikes in loss observed during the initial epochs settled down as the training progressed. The loss became more stable and decreased steadily towards the later epochs. This stabilization of loss indicated that the training process became more stable, and the algorithm converged towards an optimal solution as it approached the end of training.

Table 4.3 Faster R-CNN: Average Loss

Epoch	1	2	3	4	5	6	7	8	9	10
Loss	1.453	0.766	0.667	0.426	0.347	0.218	0.295	0.259	0.138	0.105

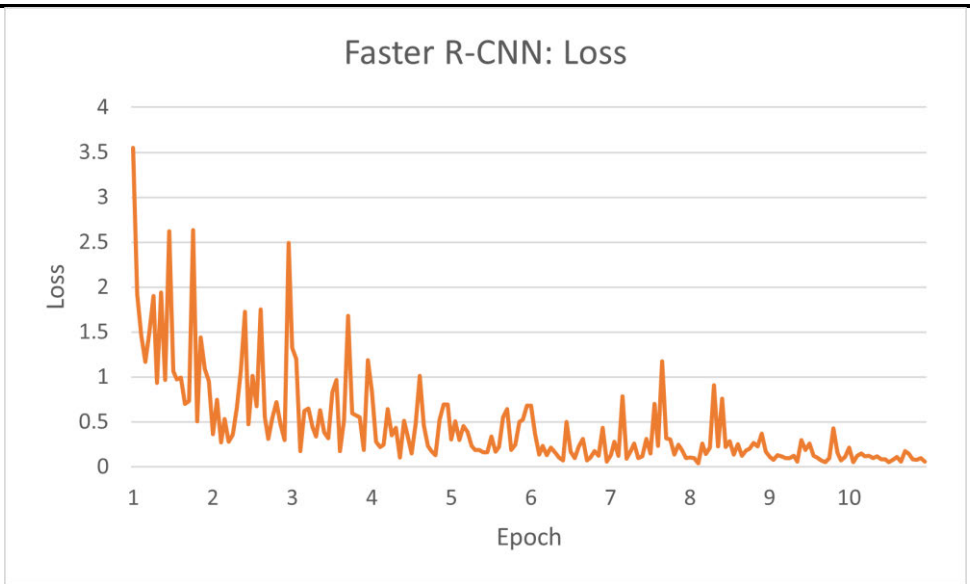

Fig. 4.2. Faster R-CNN: Loss

Table 4.4 shows the average detection accuracy of the Faster R-CNN algorithm. The average accuracy steadily increased across epochs, beginning at 98.60% in the first epoch and reached 99.88% in the tenth epoch. The consistent increase in accuracy indicated that the Faster R-CNN algorithm was learning effectively from the training data and improving its ability to correctly classify objects over time and was also validated by the steady decrease in loss depicted in Fig. 4.2. These high accuracy levels indicated that the Faster R-CNN algorithm was achieving a high level of precision in object detection tasks.

Fig. 4.3 shows the accuracy across the ten epochs for the Faster R-CNN algorithm. After an initial increase in accuracy in the first epoch, the rate of improvement in accuracy slowed down, and the accuracy values plateaued around higher levels. This plateauing indicated that the algorithm had reached a satisfactory level of performance, and further improvements in accuracy would be marginal or require more extensive training or algorithm modifications. The consistent increase in accuracy demonstrated the effectiveness of the Faster R-CNN algorithm in learning from training data and improving its object detection capabilities over epochs.

Table 4.4 Faster R-CNN: Average Accuracy

Epoch	1	2	3	4	5	6	7	8	9	10
Accuracy	98.60	99.12	99.60	99.77	99.67	99.81	99.78	99.79	99.79	99.88

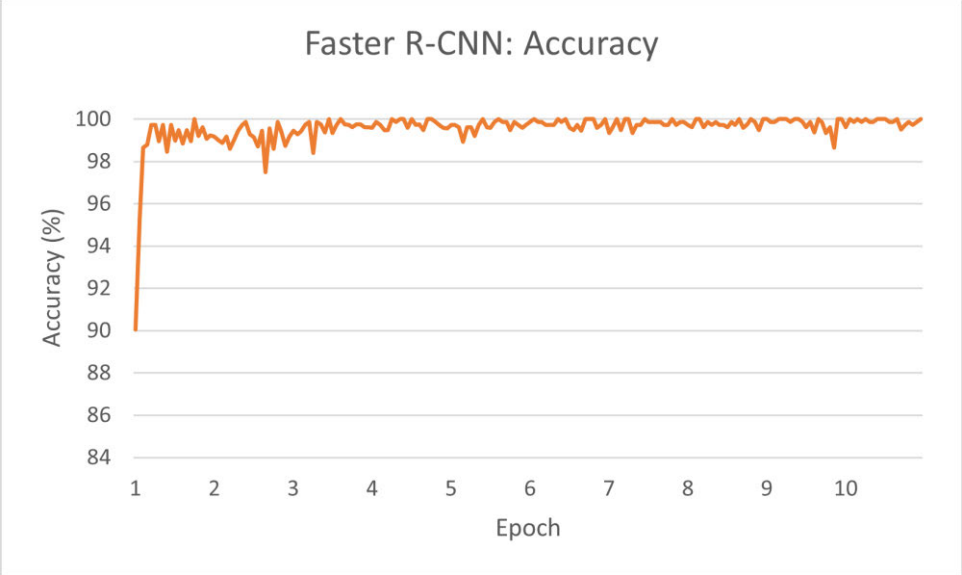

Fig. 4.3. Faster R-CNN: Accuracy

Fig. 4.4 illustrates the precision vs recall curve of the Faster R-CNN algorithm during testing. When recall ranged between 0 and 0.59, the precision remained steady at 1. This indicated that when the recall was low, the algorithm exhibited very high precision, meaning that the majority of the detections made by the algorithm were correct. As the recall increased beyond 0.6, there was a slight drop in precision, averaging around 0.9 between a recall of 0.6 to 0.7. This indicated that as the algorithm tried to capture more true positives (increasing recall), it also included some false positives, leading to a slight decrease in precision. Between a recall of 0.7 to 0.8, the precision decreased further, averaging around 0.8. This indicates that as the algorithm aims to obtain even more true positives, it starts to include more false positives, resulting in a noticeable decrease in

precision. The PR curve ends with a precision value of 0.75 and a recall value of 0.81. This indicated that as the algorithm attempted to obtain more true positives, the precision continued to decrease, indicating that the algorithm's ability to differentiate true positives from false positives diminished as recall increased. The overall performance of the Faster R-CNN detector was evaluated using the mAP metric, which is calculated based on the area under the PR curve. A mAP value of 0.76 indicated that, on average, the algorithm achieved a PR balance across different object categories, with a precision of 76%.

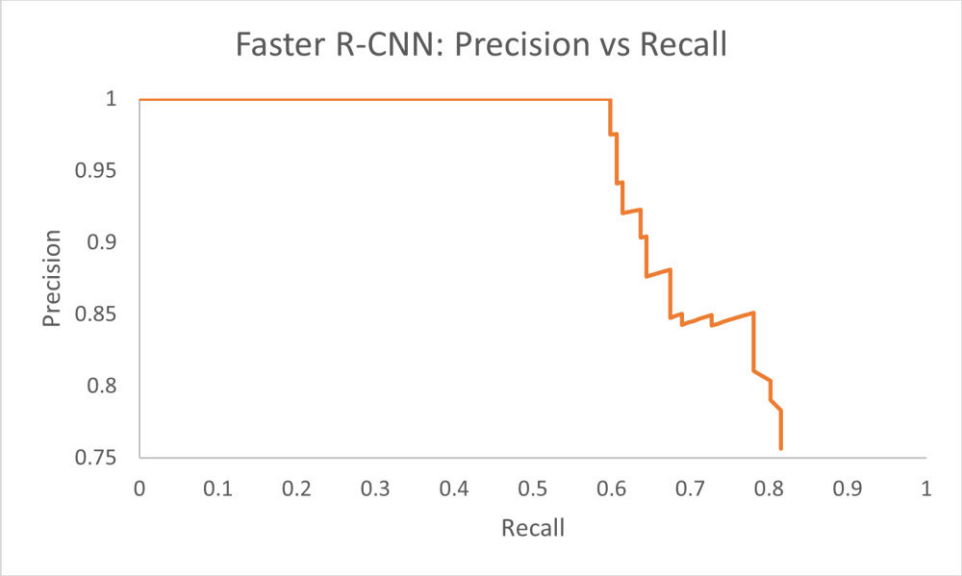

Fig. 4.4. Faster R-CNN: Precision vs Recall Curve

Fig. 4.5 shows the mAP values obtained from a selection of test images in the dataset when employing the Faster R-CNN object detection algorithm.

Fig. 4.5. Output results (mAP values) on a selection of test images for Faster R-CNN

4.3 YOLO v3

Table 4.5 shows the hyperparameter settings used during the evaluation of the YOLO v3 algorithm that provided optimal results. The algorithm was evaluated using 10 epochs and a batch size of 8. Initial experiments with batch sizes of 2 and 4 did not yield desirable results. A batch size of 8 was eventually chosen as it balanced the need for efficient memory usage with effective detection rates, leading to a more stable and consistent training performance. Training was also limited to 10 epochs due to practical constraints related to memory availability. Attempts to extend training beyond 10 epochs resulted in low GPU memory errors. As with Faster R-CNN, a learning rate of 0.001 was used and the learning rate schedule was set to “piecewise”. This allowed the YOLO v3 network to update its learning rate periodically by multiplying it according to the SGDM optimizer drop factor, leading to faster and more stable training.

Table 4.5 YOLO v3: Hyperparameter Settings

Hyperparameter	Value
Learning Rate	0.001
Learning Rate Schedule	Piecewise
Optimizer	SGDM
Batch Size	8
Epochs	10

Table 4.6 displays the average detection time of the YOLO v3 algorithm. Average detection time persisted consistently across all epochs, with values ranging from 1.10 to 1.30 seconds. This consistency indicated that the YOLO v3 algorithm maintained a stable computational efficiency throughout the training process. There was minimal variability in detection time across epochs, with small fluctuations within a narrow range of values. This indicated that the YOLO v3 algorithm's performance in terms of computational efficiency was robust and did not significantly fluctuate over time. The consistent and relatively low average detection time for YOLO v3 indicates that the algorithm is capable of efficiently processing and detecting objects in images in real-time or near real-time scenarios.

Fig. 4.6 illustrates the detection time across the ten epochs for the YOLO v3 algorithm. Significant spikes in the detection time were noted during the initial iteration of every epoch, ranging from 17 to 20 seconds. These spikes represented significant deviations from the average detection time observed in subsequent iterations within the same epoch. The spikes in the detection time for the initial iteration were due to the initialisation processes which are the initialisation of the network weights, loading of data

and compiling computational graphs. These tasks typically require extra time and computational resources as compared to future iterations.

Table 4.6 YOLO v3: Average Detection Time

Epoch	1	2	3	4	5	6	7	8	9	10
Time (s)	1.30	1.10	1.10	1.15	1.15	1.15	1.15	1.15	1.15	1.15

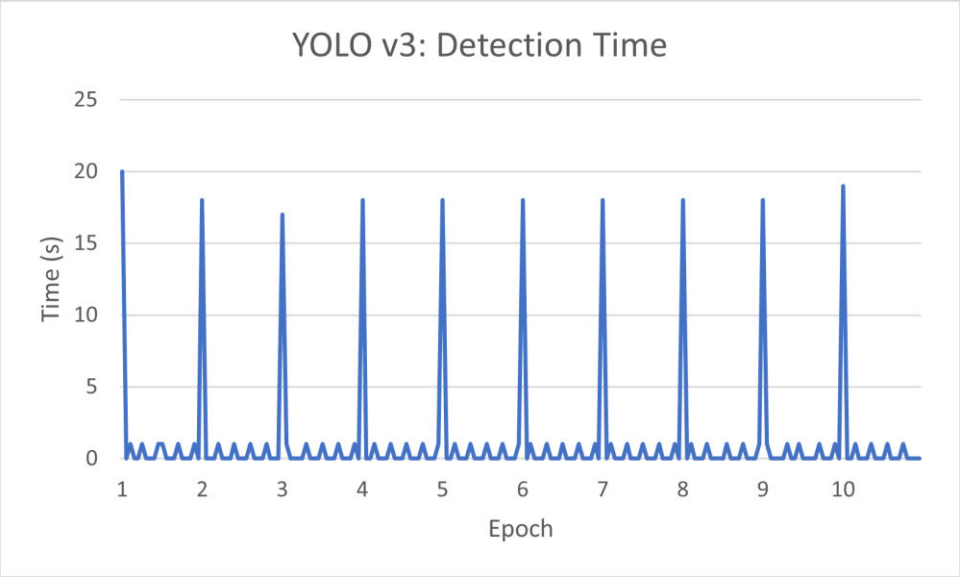

Fig. 4.6. YOLO v3: Detection Time

Table 4.7 displays the average detection loss of the YOLO v3 algorithm. The average loss started relatively high at 3.202 in the first epoch, indicating that the algorithm initially struggled to make accurate predictions and fit the training data well. As the training advanced, the average loss gradually declined over subsequent epochs. The loss dropped significantly from 3.202 in the first epoch to 0.080 in the tenth epoch. This decreasing trend indicated that the YOLO v3 algorithm was effectively learning from the training data and improving its ability to make accurate predictions over time. The most significant reduction in loss occurred between the first and fourth epochs, where the loss decreased

from 3.202 to 0.788. This sharp reduction indicated that the algorithm quickly learned from the initial training data and made substantial improvements in its predictions.

Fig. 4.7 illustrates the loss across the ten epochs for the YOLO v3 algorithm. The first three epochs exhibited erratic spikes in loss, with values ranging from 1.161 to 6.524. These spikes indicated high instability or variability whilst in the process of training throughout the early iterations. Loss reached a maximum of 3.202 in the first epoch, increased to 6.524 in the second epoch, and then decreased to a maximum of 3.741 in the third epoch. The loss also reached a minimum of 1.5 in the first epoch, 1.786 in the second epoch and 1.161 in the third epoch. Such erratic behaviour indicated that the algorithm encountered difficulties in fitting the training data during the early stages of training. After the initial erratic spikes, the loss began to decrease significantly from the fourth epoch. From the seventh epoch onwards, the loss values stabilized around lower levels, ranging from 0.028 to 0.567. This stabilization indicated that the algorithm reached a more optimal state and further improvements in loss became marginal.

Table 4.7 YOLO v3: Average Loss

Epoch	1	2	3	4	5	6	7	8	9	10
Loss	3.202	3.667	2.004	0.788	0.791	0.759	0.260	0.171	0.112	0.080

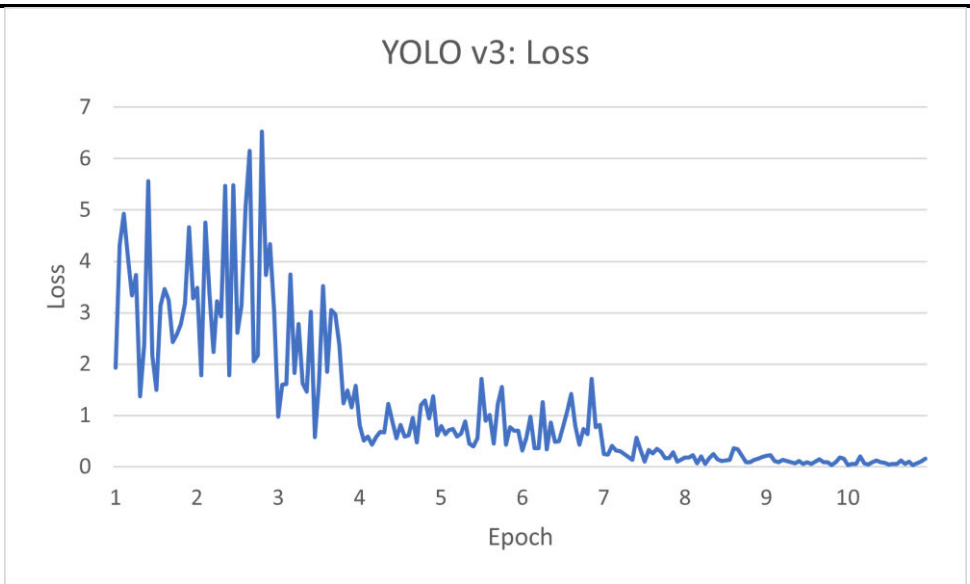

Fig. 4.7. YOLO v3: Loss

Table 4.8 displays the average detection accuracy of the YOLO v3 algorithm. The average accuracy increased consistently across epochs, beginning at 96.58% in the first epoch and achieving 99.91% in the tenth epoch. The increasing trend indicated that the YOLO v3 algorithm was learning effectively from the training data and improving its ability to classify objects accurately over time. The most notable improvement in accuracy occurred between the first and second epochs, where the accuracy increased from 96.58% to 99.44%. This significant improvement indicated that the algorithm quickly learned from the initial training data and made substantial improvements in its predictions. Subsequent epochs continued to show improvements in accuracy, albeit at a slower rate, indicating that the algorithm continued to refine its predictions and converge towards a better solution.

Fig. 4.8 illustrates the accuracy across the ten epochs for the YOLO v3 algorithm. The increasing trend in accuracy indicated that the YOLO v3 algorithm was effectively learning to detect objects in the training data and improving its performance over time. High accuracy values indicated that the algorithm was making accurate predictions on the training data and had a high level of confidence in its classifications. The significant

improvement in accuracy during the early epochs, starting off with an accuracy of 69.24%, indicated that the algorithm quickly learned from the initial training data, but as training progressed, further improvements became marginal.

Table 4.8 YOLO v3: Average Accuracy

Epoch	1	2	3	4	5	6	7	8	9	10
Accuracy	96.58	99.44	99.73	99.83	99.79	99.87	99.85	99.86	99.87	99.91

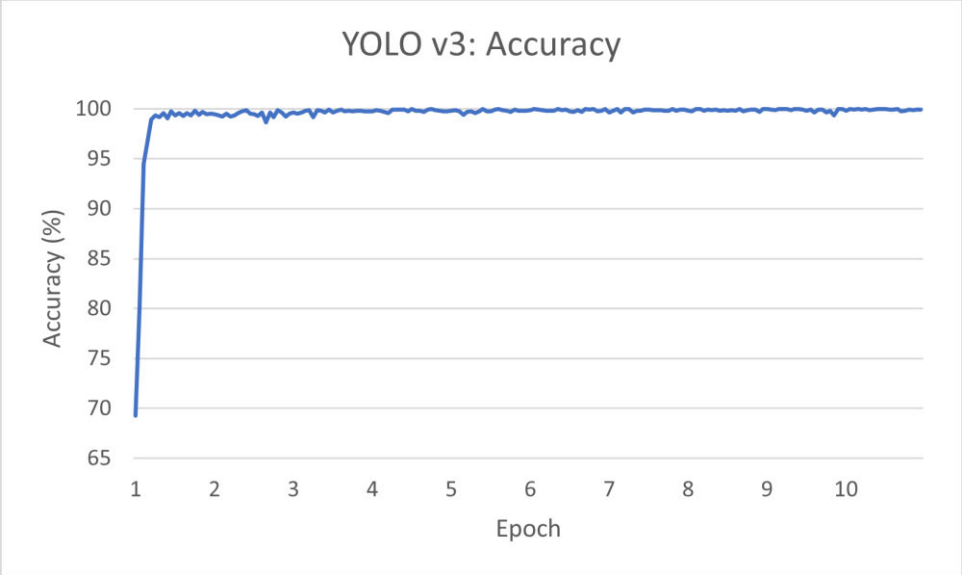

Fig. 4.8. YOLO v3: Accuracy

Fig. 4.9 illustrates the precision vs recall curve of the YOLO v3 algorithm during testing. When recall ranged between 0 and 0.67, the precision remained steady at 1. This indicated that the algorithm achieved perfect precision for detecting objects with relatively low recall. A precision value of 1 indicated that all detections made by the algorithm at those recall levels were correct, without any false positives. As the recall surpassed 0.67, the precision started to decline. This decline indicated that as the algorithm attempted to capture more objects (higher recall), it also introduced some false positives, leading to a

lower precision. The average precision dropped to 0.97 between a recall of 0.67 and 0.8, indicating a slight decrease in precision while capturing more objects. The precision further decreased to an average of 0.88 between a recall of 0.8 and 0.85, indicating a more significant decrease in precision as more objects were detected. The overall performance of the YOLO v3 detector was evaluated using the mAP metric, which is calculated based on the area under the PR curve. A mAP value of 0.81 indicated that, on average, the algorithm achieved a PR balance across different object categories, with a precision of 81%.

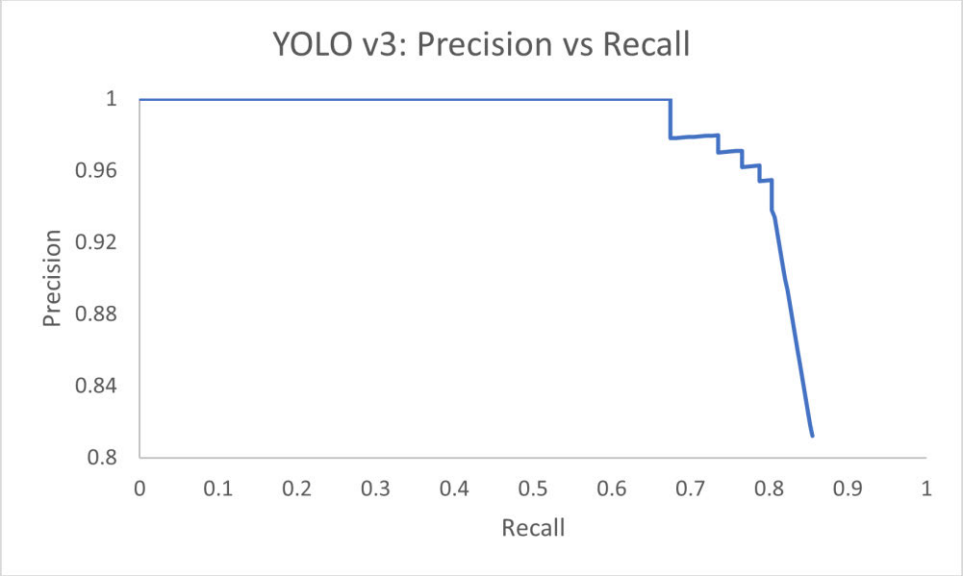

Fig. 4.9. YOLO v3: Precision vs Recall Curve

Fig. 4.10 shows the mAP values obtained from a selection of test images in the dataset when employing the YOLO v3 object detection algorithm.

Fig. 4.10. Output results (mAP values) on a selection of test images for YOLO v3

4.4 SSD

Table 4.9 shows the hyperparameter settings used during the evaluation of the SSD algorithm that provided optimal results. The algorithm was evaluated using 10 epochs and a batch size of 4. Initial experiments with a batch size of 2 did not yield desirable results. It was also noted that batch sizes higher than 4 and epochs exceeding 10 resulted in low GPU memory errors. Therefore, a batch size of 4 was eventually chosen as it balanced the need for efficient memory usage with effective detection rates, leading to a more stable and consistent training performance. As with Faster R-CNN and YOLO v3, a learning rate of 0.001 was used and the learning rate schedule was set to “piecewise”. This allowed the SSD network to update its learning rate periodically by multiplying it according to the SGDM optimizer drop factor, leading to faster and more stable training.

Table 4.9 SSD: Hyperparameter Settings

Hyperparameter	**Value**
Learning Rate	0.001
Learning Rate Schedule	Piecewise
Optimizer	SGDM
Batch Size	4
Epochs	10

Table 4.10 displays the average detection time of the SSD algorithm. Average detection time stayed consistent and low across all epochs, ranging from 0.45 to 0.80 seconds. This consistency indicated that the SSD algorithm maintained a stable and efficient computational performance throughout the training process. There was minimal variability in detection time across epochs, with small fluctuations within a narrow range of values. This indicated that the SSD algorithm's performance in terms of computational efficiency was robust and did not significantly fluctuate over time. The consistent and low average detection time for SSD indicates that the algorithm is capable of efficiently processing and detecting objects in images in real-time or near real-time scenarios.

Fig. 4.11 illustrates the detection time across the ten epochs for the SSD algorithm. Significant spikes in the detection time were noted during the initial iteration of every epoch, ranging from 3 to 4 seconds, with the first iteration in the first epoch being substantially higher at 9 seconds. These spikes represented significant deviations from the average detection time observed in subsequent iterations within the same epoch. The spikes in detection time for the first iteration can be attributed to initialization processes or other factors that required extra time and computational resources as compared to future

iterations. Despite the spikes in detection time for the first iteration, the SSD algorithm demonstrated consistent and efficient computational performance throughout the training process, as evidenced by its stable average detection time across multiple epochs.

Table 4.10 SSD: Average Detection Time

Epoch	1	2	3	4	5	6	7	8	9	10
Time (s)	0.80	0.50	0.45	0.50	0.50	0.45	0.45	0.45	0.45	0.45

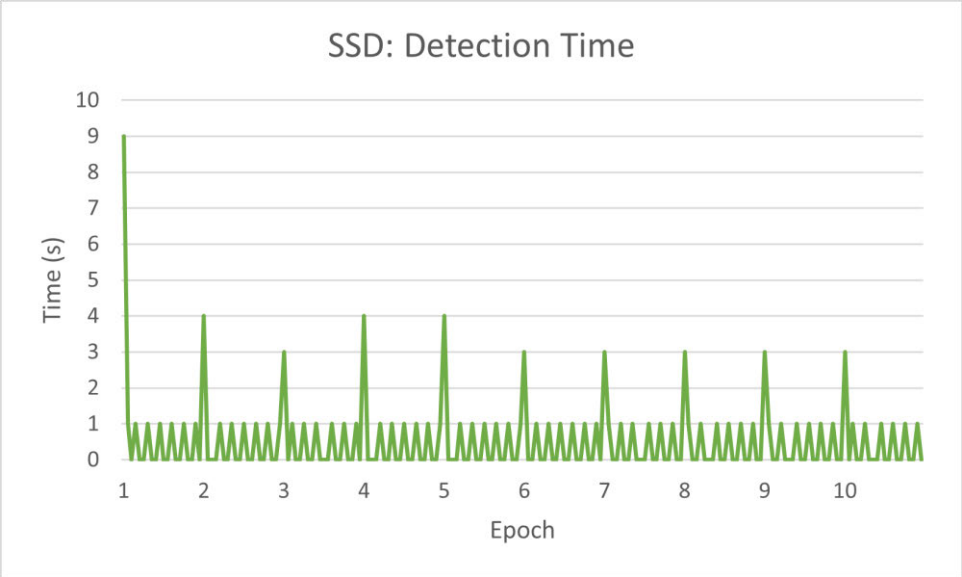

Fig. 4.11. SSD: Detection Time

Table 4.11 displays the average detection loss of the SSD algorithm. The average loss started relatively high at 11.992 in the first epoch, indicating that the algorithm initially struggled to make accurate predictions and fit the training data well. As the training continued, the average loss gradually declined over subsequent epochs. The loss dropped significantly from 11.992 during the initial epoch to 0.807 in the last two epochs. The decreasing trend indicated that the SSD algorithm was learning effectively from the training data and improving its ability to make accurate predictions over time.

Fig. 4.12 illustrates the loss across the ten epochs for the SSD algorithm. The first epoch demonstrated a considerable drop in loss from a high value of 11.992 to 9.347, followed by a further reduction to 3.514 in the second epoch. This drastic decrease signified rapid learning or adaptation of the SSD algorithm to the training data during the initial training phase. Such a substantial drop indicated that the algorithm underwent significant optimizations or adjustments during the initial training iterations, resulting in enhanced performance and reduced loss. The stabilization of loss at lower levels indicated that the algorithm had converged towards a satisfactory solution, and additional training would not yield significant improvements in performance without further optimization or changes to the algorithm architecture.

Table 4.11 SSD: Average Loss

Epoch	1	2	3	4	5	6	7	8	9	10
Loss	11.992	4.049	2.270	1.597	1.362	1.187	1.246	0.934	0.807	0.807

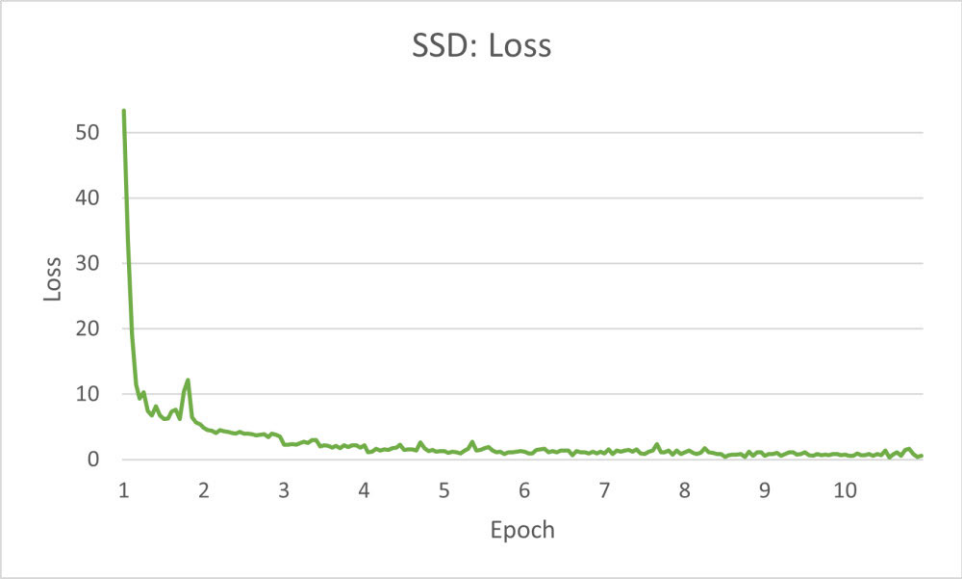

Fig. 4.12. SSD: Loss

Table 4.12 displays the average detection accuracy of the SSD algorithm. The average accuracy increased consistently across epochs, beginning at 94.57% in the first epoch and obtaining 99.95% in the tenth epoch. The increasing trend indicated that the SSD algorithm was learning effectively from the training data and improving its ability to classify objects accurately over time. The most notable improvement in accuracy occurred between the first and second epochs, where the accuracy increased from 94.57% to 99.75%. This significant improvement indicated that the algorithm quickly learned from the initial training data and made substantial improvements in its predictions.

Fig. 4.13 illustrates the accuracy across the ten epochs for the SSD algorithm. The significant improvement in accuracy during the early epochs, starting off with an accuracy of 48.44%, indicated that the algorithm quickly learned from the initial training data, but as training progressed, further improvements became marginal. The increasing trend in accuracy indicated that the SSD algorithm was effectively learning to detect objects in the training data and improved its performance over time. High accuracy values indicated that the algorithm was making accurate predictions on the training data and had a high level of confidence in its classifications.

Table 4.12 SSD: Average Accuracy

Epoch	1	2	3	4	5	6	7	8	9	10
Accuracy	94.57	99.75	99.85	99.88	99.90	99.92	99.91	99.93	99.94	99.95

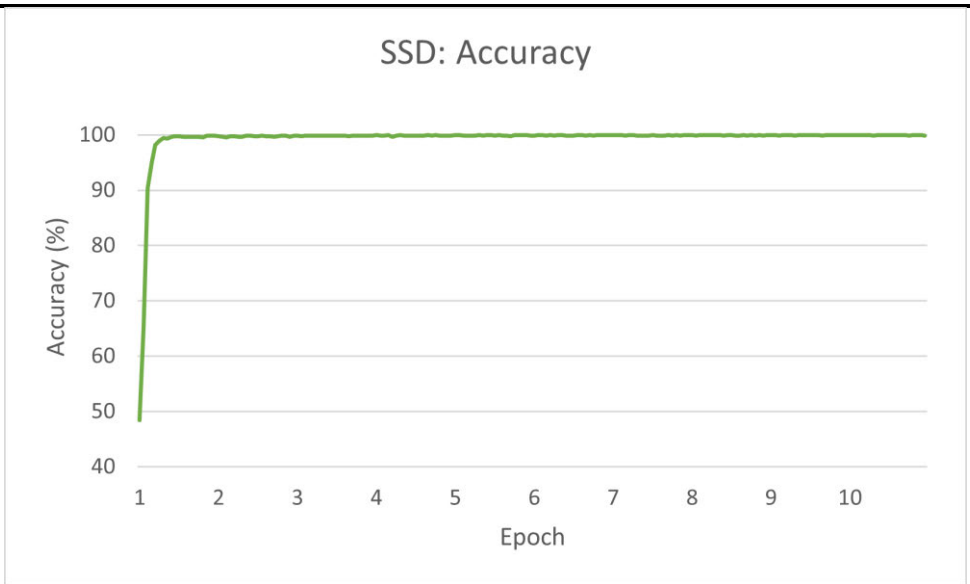

Fig. 4.13. SSD: Accuracy

Fig. 4.14 illustrates the precision vs recall curve of the SSD algorithm during testing. When recall ranged between 0 and 0.29, the precision remained steady at 1. This indicated that the algorithm achieved perfect precision for detecting objects with relatively low recall. A precision value of 1 indicated that all detections made by the algorithm at these recall levels were correct, without any false positives. At a recall of 0.3, the precision dropped slightly to 0.98. This indicated that as the algorithm attempted to capture more objects (higher recall), it introduced some false positives, leading to a slight decrease in precision. However, after this initial drop, the precision gradually increased to 0.99 between a recall of 0.3 and 0.75. This indicated that the algorithm could detect more objects whilst maintaining high precision. The PR curve ended at a recall of 0.89 and the precision dropped to 0.87. This indicated that as the algorithm attempted to capture a larger number of objects, it introduced more false positives, leading to a decline in precision. The drop in precision at higher recall values indicates a trade-off between recall and precision, where the algorithm may sacrifice some precision to achieve higher recall. The overall performance of the SSD detector was evaluated using the mAP metric, which is calculated

based on the area under the PR curve. A mAP value of 0.92 indicates that, on average, the algorithm achieved a PR balance across different object categories, with a precision of 92%.

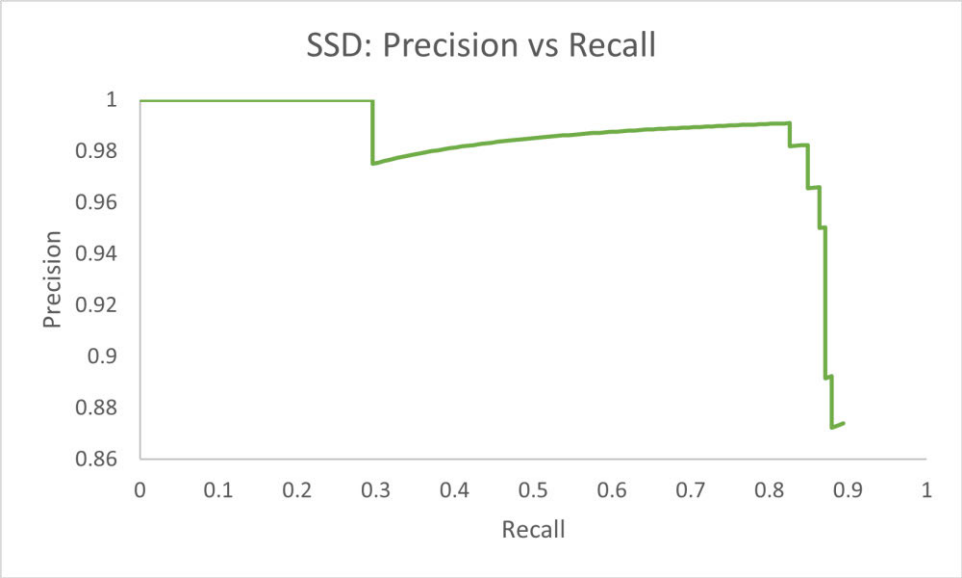

Fig. 4.14. SSD: Precision vs Recall Curve

Fig. 4.15 shows the mAP values obtained from a selection of test images in the dataset when employing the SSD object detection algorithm.

Fig. 4.15. Output results (mAP values) on a selection of test images for SSD

4.5 Overview

Fig. 4.16 illustrates the precision vs recall curve for the three object detection algorithms: Faster R-CNN, YOLO v3, and SSD. Faster R-CNN showed a high level of precision when at a low value of recall due to its two-staged technique, which results in proposals being created first and then classified. Precision decreased as recall increased due to the algorithm’s difficulty with sustaining accuracy at a higher recall. YOLO v3 exhibited a similar trend with a higher precision when at a low value of recall, followed by a steady decrease with increasing recall values. In comparison to two-stage methods such as Faster-RCNN, YOLO's single stage architecture allows for the faster processing of images, however this can lead to a decline in precision. SSD exhibited a high level of precision when at a low value of recall but had a moderate decrease with increasing recall values. The design of SSD intends to find a good trade-off between speed and accuracy by predicting bounding boxes and object categories directly from each feature map at multiple scales.

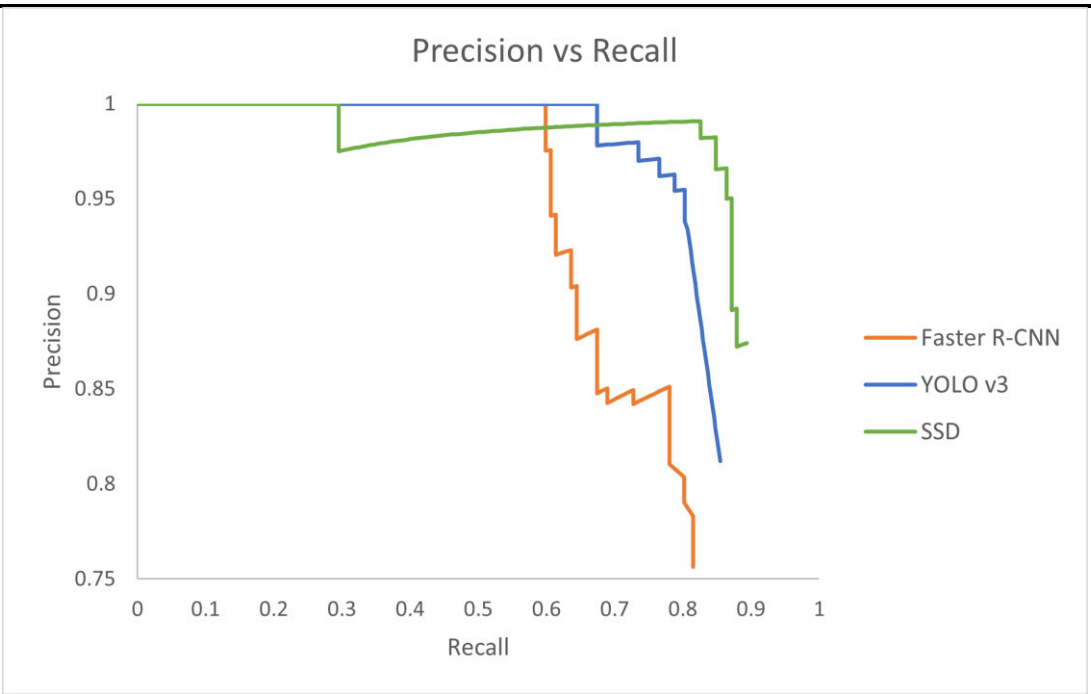

Fig. 4.16. Precision vs Recall Curve

Fig. 4.17 depicts the mAP for the three object detection algorithms. SSD achieved the highest mAP of 0.92, indicating its superior performance in accurately detecting objects across various categories in the dataset. This could be attributed to its efficient single-stage architecture. Faster R-CNN and YOLO v3 both demonstrated competitive performance, with Faster R-CNN having a slightly lower mAP compared to YOLO v3.

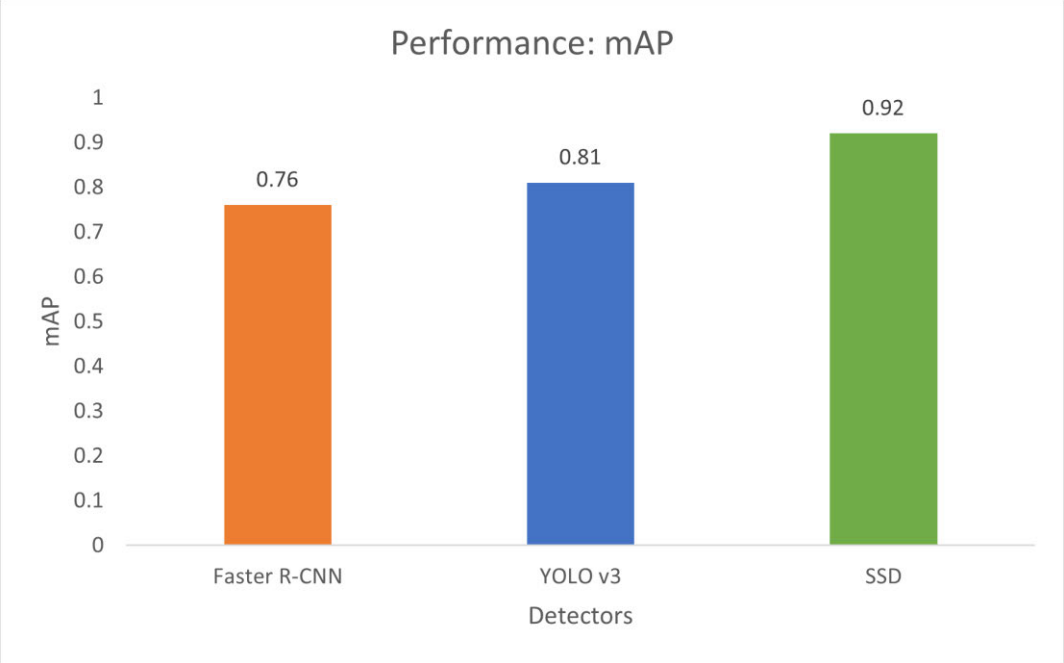

Fig. 4.17. Performance: mAP

Table 4.13 displays the overall detector results for the three object detection algorithms. SSD exhibited the lowest average detection time, making it suitable for real-time applications where speed is crucial. YOLO v3 provided a decent detection time, achieving a good trade-off between accuracy and speed. Faster R-CNN had a higher detection time attributed to its two-staged approach, whereby region proposals are first created and then classified, reducing its suitability for real-time applications, however it has the potential for greater accuracy. The analysis showed high initial loss for all three algorithms, which gradually decreased over epochs as the algorithms continued to learn and adjust their parameters. SSD demonstrated stabilization in loss after an early decrease, whilst both YOLO v3 and Faster R-CNN exhibited greater fluctuations. Each algorithm exhibited high average accuracy, but Faster R-CNN achieved a marginally higher accuracy compared to YOLO v3 and SSD. This indicated that all algorithms performed well in correctly identifying the vehicles in this dataset. Exhibiting the greatest stability in performance throughout all epochs was SSD, with consistent loss, accuracy, and detection

time. YOLO v3 and Faster R-CNN exhibited a strong performance in terms of accuracy but had slight variations in its loss as well as its detection time.

Table 4.13 Overall Detector Results

Algorithm	Average Detection Time	Average Loss	Accuracy	mAP
Faster R-CNN	5.1 s	0.467	99.58 %	0.76
YOLO v3	1.16 s	1.183	99.47 %	0.81
SSD	0.5 s	2.625	99.36 %	0.92

4.6 Summary

The choice of batch size and number of epochs used when training the three object detector algorithms significantly impacted the algorithm's performance. It was essential to strike a balance between training efficiency and algorithm stability based on the limitations of the dataset and available computational resources. Larger batch sizes required more memory and was not feasible with the resources available in this study.

In the conducted experiments for vehicle detection, SSD showed the most superior performance from all three algorithms. It achieved the shortest average detection time of 0.5 seconds, the highest mAP value of 0.92, and an accuracy of 99.36%. YOLO v3, balancing speed and accuracy, performed competitively with an average detection time of 1.16 seconds, a fairly high mAP value of 0.8, and an accuracy of 99.47%. Meanwhile, Faster R-CNN achieved the highest accuracy of 99.58% yet had the longest average detection time of 5.1 seconds and a mAP value of 0.76.

In summary, the results presented illustrates that each algorithm has distinct weaknesses as well as strengths, but the optimal solution is dependent on the specific demands of the application, such as the need for real-time processing, accuracy requirements, and computational constraints.

CHAPTER 5 Conclusion

5.1 Thesis Conclusion

In this research, three CNN object detection algorithms were experimentally evaluated for their performance in vehicle detection when used on a portable PC with limited GPU computing. This research aimed to investigate whether these algorithms can be employed on PCs and which of these algorithms has the best combination of speed vs accuracy.

Chapter 1 provided an introduction to CNNs and object detection, explained the scope and limitations of the research and highlighted the aims and objectives of this research. The research objectives formulated were:

- To comprehensively review relevant publications on CNN object detection algorithms.
- To develop CNN object detection algorithms in MATLAB®.
- To evaluate the performance of CNN object detection algorithms for vehicle detection.

Chapter 2 addressed the first objective by reviewing relevant literature on CNNs, the chosen object detection algorithms for this research (Faster R-CNN, YOLO v3, and SSD), and the evaluation methods used for comparing object detection algorithms.

Chapter 3 addressed the second objective by documenting the methodology followed to develop the chosen object detection algorithms in MATLAB®. This included the hyperparameters used for training as well as the pre-processing and data augmentation

phase. The dataset used to train and evaluate the performance of the object detection algorithms was presented.

Chapter 4 addressed the third objective by presenting and discussing the experimental results which included the average detection time, average loss, average accuracy, and mAP for each of the object detection algorithms.

Within the parameters of the experiments conducted in this research, SSD demonstrated the best performance between all algorithms, with the lowest average detection time of 0.5 seconds, highest mAP value of 0.92, and a competitive accuracy of 99.36%. Its efficient architecture allows for fast and accurate detection of objects, making it preferable for real-time applications in vehicle detection. By providing a good balance between accuracy and speed, YOLO v3 exhibits competitive performance as a result of a decent detection time of 1.16 seconds and a fairly high mAP value of 0.81. The one-stage approach of YOLO v3 enables efficient processing of images, making it suitable for various applications in vehicle detection. While achieving an accuracy of 99.58% and a mAP of 0.76, Faster R-CNN's prolonged average detection time of 5.1 seconds might limit its applicability to real-time applications in vehicle detection. However, Faster R-CNN's two-stage architecture allows for accurate classification and object detection, making it suitable for tasks where accuracy is paramount.

In summary, each object detection algorithm has its own weaknesses and strengths, but the choice of algorithm will ultimately depend on specific application requirements such as accuracy, speed, and computational resources. This may only be applicable to vehicle recognition, but SSD will excel in real-time applications with its fast detection time and high mAP, while YOLO v3 will offer a good trade-off between speed and accuracy. Faster R-CNN may be more accurate but may not be suitable for real-time applications due to its higher computational cost.

5.2 Research Challenges and Limitations

Some of the challenges and concerns highlighted in this section may not be directly linked to the objectives of this study, but their significance in the field of CNN based object detection should be noted.

1. **Data Limitations:** CNNs typically need greater quantities of labelled data. However, for this study, a smaller dataset was sufficient and useful when evaluating the various object detection methods. Practically, a far larger number of labelled images would be required for the training of a robust network for object detection. The creation of ground truth images requires manual segmentation, a process that can be both time-intensive and subjective.
2. **Overfitting:** CNN algorithms are prone to overfitting, where they memorize training data patterns instead of learning generalizable features. Regularization techniques such as dropout and weight decay are commonly used to mitigate overfitting, but finding the right balance remains a challenge.
3. **Class Imbalance:** Object detection datasets often exhibit class imbalance, where certain object classes are much more prevalent than others. This can result in biased algorithms that perform poorly on minority classes. In this research the dataset class was limited to “vehicles”.
4. **Real-world Variability:** Object detection algorithms trained on controlled datasets may not generalize well to real-world scenarios with variability in lighting conditions, clutter, and background complexity. Robustness to such environmental factors is essential for real-world deployment.

5.3 Recommendations for Future Work

Future research directions may include:

- Explore novel architectural designs to improve speed and accuracy without sacrificing one for the other.
- Develop lightweight versions of these algorithms with reduced parameters and computational requirements for deployment on resource-constrained devices.
- Address issues related to generalization in challenging scenarios, such as varying lighting conditions and cluttered backgrounds.
- Research methods for domain adaptation to improve algorithm performance on target domains with limited labelled data, particularly for real-world applications where domain shifts are common.
- Establish standardised evaluation protocols and benchmarks to compare object detection algorithms across various datasets and tasks.
- Additional experimentation to verify if similar results would be achieved on different datasets and algorithms developed in different programming languages outside of MATLAB®.

REFERENCES

[1] Z. Zou, Z. Shi, Y. Guo and J. Ye, "Object Detection in 20 Years: A Survey", arXiv e-prints, vol. 190505055, 2019. Available: https://arxiv.org/pdf/1905.05055.pdf.

[2] N. Yadav and U. Binay, "Comparative Study of Object Detection Algorithms", International Research Journal of Engineering and Technology, vol. 04, no. 11, pp. 586-591, 2017.

[3] J. Shanahan and L. Dai, "Introduction to Computer Vision and Real Time Deep Learning-based Object Detection", Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020. Available: https://doi.org/10.1145/3394486.3406713

[4] A. Gibson and J. Patterson, Deep Learning: A Practitioner's Approach. Sebastopol: O'Reilly, 2017.

[5] K. Kang, H. Li, J. Yan, X. Zeng, B. Yang, T. Xiao, C. Zhang, Z. Wang, R. Wang, X. Wang et al., "T-CNN: Tubelets With Convolutional Neural Networks for Object Detection From Videos", IEEE Transactions on Circuits and Systems for Video Technology, vol. 28, no. 10, pp. 2896-2907, 2018. Available: https://doi.org/10.1109/tcsvt.2017.2736553

[6] B. Hariharan, P. Arbel´aez, R. Girshick, and J. Malik, "Simultaneous Detection and Segmentation", European Conference on Computer Vision. Springer, pp. 297–312, 2014. Available: https://arxiv.org/abs/1407.1808

[7] Y. LeCun, Y. Bengio and G. Hinton, "Deep learning", Nature, vol. 521, no. 7553, pp. 436-444, 2015. Available: https://doi.org/10.1038/nature14539

[8] A. Karne, R. Karne, V. Kumar, and A. Arunkumar, “Convolutional Neural Networks for Object Detection And Recognition,” Journal of Artificial Intelligence, Machine Learning and Neural Network, vol. 3, no. 2, pp. 1–13, Feb. 2023. doi:10.55529/jaimlnn.32.1.13

[9] S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 6, pp. 1137-1149, 2017. Available: https://doi.org/10.1109/tpami.2016.2577031.

[10] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection”, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 779–788, 2016. Available: https://arxiv.org/abs/1506.02640

[11] W. Liu et al., "SSD: Single Shot MultiBox Detector", Computer Vision – ECCV 2016. Lecture Notes in Computer Science, vol. 9905, 2016. Available: https://doi.org/10.1007/978-3-319-46448-0_2

[12] P. Prakas and T. Nagalakshmi, "Fast and Economical Object Tracking using Raspberry Pi 3.0", International Journal of Engineering and Advanced Technology, vol. 8, no. 6, pp. 336-338, 2019. Available: https://doi.org/10.35940/ijeat.f1070.0886s19

[13] J. S. Ren and Y. Wang, “Overview of Object Detection Algorithms Using Convolutional Neural Networks,” Journal of Computer and Communications, vol. 10, pp. 115–132, Jan. 2022. Available: https://doi.org/10.4236/jcc.2022.101006

[14] J. Schmidhuber, “Deep learning in neural networks: An overview,” Neural Networks, vol. 61, pp. 85–117, Jan. 2015. doi:10.1016/j.neunet.2014.09.003

[15] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” Biological Cybernetics, vol. 36, no. 4, pp. 193–202, Apr. 1980. doi:10.1007/bf00344251

[16] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998. doi:10.1109/5.726791

[17] B. Singh and L. S. Davis, “An analysis of scale invariance in object detection - snip,” 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 3578–3587, Jun. 2018. doi:10.1109/cvpr.2018.00377

[18] Y. Guo, Y. Liu, T. Georgiou, and M. S. Lew, “A review of semantic segmentation using Deep Neural Networks,” International Journal of Multimedia Information Retrieval, vol. 7, no. 2, pp. 87–93, Nov. 2017. doi:10.1007/s13735-017-0141-z

[19] M. D. Zeiler and R. Fergus, “Visualizing and understanding Convolutional Networks,” Computer Vision – ECCV 2014, pp. 818–833, 2014. doi:10.1007/978-3-319-10590-1_53

[20] N. Ketkar and J. Moolayil, “Convolutional Neural Networks,” Deep Learning with Python, pp. 197–242, 2021. doi:10.1007/978-1-4842-5364-9_6

[21] F. Yu and V. Koltun, “Multi-Scale Context Aggregation by Dilated Convolutions,” International Conference on Learning Representations 2016, 2016. Available: https://doi.org/10.48550/arXiv.1511.07122

[22] L. Kaiser, A. Gomez, and F. Chollet, Depthwise Separable Convolutions for Neural Machine Translation, Jun. 2017. Available: https://doi.org/10.48550/arXiv.1706.03059

[23] C. Ye *et al.*, “Network Deconvolution,” *International Conference on Learning Representations 2020*, 2020. Available: https://doi.org/10.48550/arXiv.1905.11926

[24] Y. Cao and Q. Gu, “Generalization Bounds of Stochastic Gradient Descent for Wide and Deep Neural Networks,” Advances in Neural Information Processing Systems, vol. 32, 2019.

[25] Z. Zhang, "Improved Adam Optimizer for Deep Neural Networks," 2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS), Banff, AB, Canada, 2018, pp. 1-2, doi: 10.1109/IWQoS.2018.8624183.

[26] F. Zou, L. Shen, Z. Jie, W. Zhang, and W. Liu, “A sufficient condition for convergences of Adam and rmsprop,” 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Jun. 2019. doi:10.1109/cvpr.2019.01138

[27] L. Datta, A Survey on Activation Functions and their relation with Xavier and He Normal Initialization, 2020. Available: https://doi.org/10.48550/arXiv.2004.06632

[28] W. Hao, W. Yizhou, L. Yaqin and S. Zhili, "The Role of Activation Function in CNN," 2020 2nd International Conference on Information Technology and Computer Application (ITCA), Guangzhou, China, 2020, pp. 429-432, doi: 10.1109/ITCA52113.2020.00096.

[29] S. Sharma, S. Sharma, and A. Athaiya, “ACTIVATION FUNCTIONS IN NEURAL NETWORKS,” International Journal of Engineering Applied Sciences and Technology, vol. 4, no. 12, pp. 310–316, Apr. 2020.

[30] I. Daubechies, R. DeVore, S. Foucart, B. Hanin, and G. Petrova, “Nonlinear Approximation and (Deep) ReLU Networks,” Constructive Approximation, vol. 55, no. 1, pp. 127–172, Apr. 2021. doi:10.1007/s00365-021-09548-z

[31] X. Qi, Y. Wei, X. Mei, R. Chellali, and S. Yang, “Comparative analysis of the linear regions in relu and leakyrelu networks,” Communications in Computer and Information Science, pp. 528–539, Nov. 2023. doi:10.1007/978-981-99-8132-8_40

[32] J. Crnjanski, M. Krstić, A. Totović, N. Pleros, and D. Gvozdić, “Adaptive sigmoid-like and PReLU activation functions for all-optical Perceptron,” Optics Letters, vol. 46, no. 9, pp. 2003–2006, Apr. 2021. doi:10.1364/ol.422930

[33] Z. Qiumei, T. Dan and W. Fenghua, "Improved Convolutional Neural Network Based on Fast Exponentially Linear Unit Activation Function," in IEEE Access, vol. 7, pp. 151359-151367, 2019, doi: 10.1109/ACCESS.2019.2948112.

[34] H. Pratiwi et al., “Sigmoid activation function in selecting the best model of Artificial Neural Networks,” Journal of Physics: Conference Series, vol. 1471, no. 1, p. 012010, Feb. 2020. doi:10.1088/1742-6596/1471/1/012010

[35] M. A. Sartin and A. C. R. da Silva, "Approximation of hyperbolic tangent activation function using hybrid methods," 2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), Darmstadt, Germany, 2013, pp. 1-6, doi: 10.1109/ReCoSoC.2013.6581545.

[36] P. Ramachandran, B. Zoph, and Q. V. Le, Searching for Activation Functions, 2017. Available: https://doi.org/10.48550/arXiv.1710.05941

[37] G. S. Chadha and A. Schwung, Learning the Non-linearity in Convolutional Neural Networks, 2019. Available: https://doi.org/10.48550/arXiv.1905.12337

[38] M. Zhao, S. Zhong, X. Fu, B. Tang, S. Dong and M. Pecht, "Deep Residual Networks With Adaptively Parametric Rectifier Linear Units for Fault Diagnosis," in IEEE Transactions on Industrial Electronics, vol. 68, no. 3, pp. 2587-2597, March 2021, doi: 10.1109/TIE.2020.2972458.

[39] M. Sun, Z. Song, X. Jiang, J. Pan, and Y. Pang, “Learning pooling for Convolutional Neural Network,” Neurocomputing, vol. 224, pp. 96–104, Feb. 2017. doi:10.1016/j.neucom.2016.10.049

[40] K. He, X. Zhang, S. Ren, and J. Sun, “Spatial pyramid pooling in deep convolutional networks for visual recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 9, pp. 1904–1916, Sep. 2015. doi:10.1109/tpami.2015.2389824

[41] B. Graham, Fractional Max-Pooling, 2014. Available: https://doi.org/10.48550/arXiv.1412.6071

[42] S. Zhai et al., “S3pool: Pooling with Stochastic Spatial sampling,” 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4003–4011, Jul. 2017. doi:10.1109/cvpr.2017.426

[43] P. K K and S. S, “Optimization of deep convolutional neural network with the Integrated Batch Normalization and Global Pooling,” International Journal of Communication Networks and Information Security (IJCNIS), vol. 15, no. 1, pp. 59–69, May 2023. doi:10.17762/ijcnis.v15i1.5617

[44] S. H. S. Basha, S. R. Dubey, V. Pulabaigari, and S. Mukherjee, “Impact of fully connected layers on performance of convolutional neural networks for Image Classification,” Neurocomputing, vol. 378, pp. 112–119, Feb. 2020. doi:10.1016/j.neucom.2019.10.008

[45] B. Ramsundar and B. R. Zadeh, Tensorflow for Deep Learning from Linear Regression to Reinforcement Learning. Beijing: O’Reilly, 2018.

[46] T. N. Sainath, O. Vinyals, A. Senior and H. Sak, "Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks," 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), South Brisbane, QLD, Australia, 2015, pp. 4580-4584, doi: 10.1109/ICASSP.2015.7178838.

[47] J. Kukačka, V. Golkov, and D. Cremers, Regularization for Deep Learning: A Taxonomy, 2017. Available: https://doi.org/10.48550/arXiv.1710.10686

[48] J. Kang, S. Tariq, H. Oh, and S. S. Woo, “A survey of deep learning-based object detection methods and datasets for overhead imagery,” IEEE Access, vol. 10, pp. 20118–20134, 2022. doi:10.1109/access.2022.3149052

[49] K. E. A. van de Sande, J. R. R. Uijlings, T. Gevers and A. W. M. Smeulders, "Segmentation as selective search for object recognition," 2011 International Conference on Computer Vision, Barcelona, Spain, 2011, pp. 1879-1886, doi: 10.1109/ICCV.2011.6126456.

[50] J. Wang, K. Chen, S. Yang, C. C. Loy, and D. Lin, “Region proposal by guided anchoring,” 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Jun. 2019. doi:10.1109/cvpr.2019.00308

[51] R. Girshick, "Fast R-CNN", 2015 IEEE International Conference on Computer Vision (ICCV), 2015. Available: https://doi.org/10.1109/iccv.2015.169

[52] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, “Going deeper in spiking neural networks: VGG and residual architectures,” Frontiers in Neuroscience, vol. 13, Mar. 2019. doi:10.3389/fnins.2019.00095

[53] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Jun. 2016. doi:10.1109/cvpr.2016.90

[54] A. Nieto-Castanon, S. S. Ghosh, J. A. Tourville, and F. H. Guenther, “Region of interest based analysis of functional imaging data,” NeuroImage, vol. 19, no. 4, pp. 1303–1316, Aug. 2003. doi:10.1016/s1053-8119(03)00188-5

[55] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, “The Pascal Visual Object Classes (VOC) challenge,” International Journal of Computer Vision, vol. 88, no. 2, pp. 303–338, Sep. 2009. doi:10.1007/s11263-009-0275-4

[56] T.-Y. Lin et al., “Microsoft Coco: Common Objects in Context,” Computer Vision – ECCV 2014, pp. 740–755, 2014. doi:10.1007/978-3-319-10602-1_48

[57] J. Kaur and W. Singh, “Tools, techniques, datasets and application areas for object detection in an image: A Review,” Multimedia Tools and Applications, vol. 81, no. 27, pp. 38297–38351, Apr. 2022. doi:10.1007/s11042-022-13153-y

[58] T.-Y. Lin et al., “Feature Pyramid Networks for Object Detection,” 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Jul. 2017. doi:10.1109/cvpr.2017.106

[59] Z. Niu, G. Zhong, and H. Yu, “A review on the attention mechanism of Deep Learning,” Neurocomputing, vol. 452, pp. 48–62, Sep. 2021. doi:10.1016/j.neucom.2021.03.091

[60] L. Yang, G. Chen, and W. Ci, “Multiclass objects detection algorithm using Darknet-53 and DenseNet for intelligent vehicles,” EURASIP Journal on Advances in Signal Processing, vol. 85, Aug. 2023. doi:10.1186/s13634-023-01045-8

[61] H. Gong, H. Li, K. Xu and Y. Zhang, "Object Detection Based on Improved YOLOv3-tiny," 2019 Chinese Automation Congress (CAC), Hangzhou, China, 2019, pp. 3240-3245, doi: 10.1109/CAC48633.2019.8996750.

[62] S. Dong, Y. Ma, and C. Li, “Implementation of detection system of grassland degradation indicator grass species based on Yolov3-SPP algorithm,” Journal of Physics: Conference Series, vol. 1738, no. 1, p. 012051, Jan. 2021. doi:10.1088/1742-6596/1738/1/012051

[63] M. S. Asyraf, I. S. Isa, M. I. Marzuki, S. N. Sulaiman, and C. C. Hung, “CNN-based Yolov3 comparison for underwater object detection,” Journal of Electrical and Electronic Systems Research (JEESR), vol. 18, pp. 30–37, Apr. 2021. doi:10.24191/jeesr.v18i1.005

[64] M. F. Haque, H. -Y. Lim and D. -S. Kang, "Object Detection Based on VGG with ResNet Network," 2019 International Conference on Electronics, Information, and Communication (ICEIC), Auckland, New Zealand, 2019, pp. 1-3, doi: 10.23919/ELINFOCOM.2019.8706476.

[65] T. Jaiswal, M. Pandey and P. Tripathi, "Real Time Multiple-Object Detection Based On Enhanced SSD," 2022 Second International Conference on Power, Control and Computing Technologies (ICPC2T), Raipur, India, 2022, pp. 1-5, doi: 10.1109/ICPC2T53885.2022.9776899.

[66] S. Jia et al., “Object detection based on the improved single shot Multibox Detector,” Journal of Physics: Conference Series, vol. 1187, p. 042041, Apr. 2019. doi:10.1088/1742-6596/1187/4/042041

[67] S. Kanimozhi, G. Gayathri and T. Mala, "Multiple Real-time object identification using Single shot Multi-Box detection," 2019 International Conference on Computational Intelligence in Data Science (ICCIDS), Chennai, India, 2019, pp. 1-5, doi: 10.1109/ICCIDS.2019.8862041.

[68] T. Schwandt, The SAGE Dictionary of Qualitative Inquiry, 4th ed. University of Illinois, Urbana-Champaign, USA: SAGE Publications, Inc, 2015

[69] S. Bukhari, "What is Comparative Study", SSRN Electronic Journal, 2011. Available: https://dx.doi.org/10.2139/ssrn.1962328

[70] J. W. Creswell and J. D. Creswell, Research Design: Qualitative, Quantitative, and Mixed Methods Approaches, 5th ed. Los Angeles: SAGE, 2018.

[71] S. Reddy, N. Pillay and N. Singh, "Comparative Study of Convolutional Neural Network Object Detection Algorithms for Image Processing," 2023 International Conference on Electrical, Computer and Energy Technologies (ICECET), Cape Town, South Africa, 2023, pp. 1-5, doi: 10.1109/ICECET58911.2023.10389186.

[72] S. Reddy, N. Pillay, and N. Singh, “Comparative Evaluation of Convolutional Neural Network Object Detection Algorithms for Vehicle Detection,” Journal of Imaging, vol. 10, no. 7, p. 162, Jul. 2024, doi: 10.3390/jimaging10070162.

[73] M. Weber and P. Perona, “Caltech Cars 1999”. CaltechDATA, Apr. 06, 2022. doi: 10.22002/D1.20084

[74] B. Philip, P. Updikeand P. Perona, “Caltech Cars 2001”. CaltechDATA, Apr. 06, 2022. doi: 10.22002/D1.20085

[75] X. Zhu, C. Vondrick, D. Ramanan, and C. Fowlkes, “Do we need more training data or better models for object detection?,” Procedings of the British Machine Vision Conference 2012, 2012. doi:10.5244/c.26.80

[76] R. Islam, K. F. Sharif, and S. Biswas, “Automatic vehicle number plate recognition using structured elements,” 2015 IEEE Conference on Systems, Process and Control (ICSPC), pp. 44–48, Dec. 2015. doi:10.1109/spc.2015.7473557

[77] H. Huang and L.-Y. Hou, “Speed limit sign detection based on gaussian color model and template matching,” 2017 International Conference on Vision, Image and Signal Processing (ICVISP), Sep. 2017. doi:10.1109/icvisp.2017.30

[78] N.-A.- Alam, M. Ahsan, Md. A. Based, and J. Haider, “Intelligent system for vehicles number plate detection and recognition using convolutional neural networks,” Technologies, vol. 9, no. 1, p. 9, Jan. 2021. doi:10.3390/technologies9010009

[79] “Recognition, object detection, and semantic segmentation,” Recognition, Object Detection, and Semantic Segmentation - MATLAB & Simulink, https://www.mathworks.com/help/vision/recognition-object-detection-and-semantic-segmentation.html (Accessed 2023).

[80] B. Koonce, “ResNet 50,” Convolutional Neural Networks with Swift for Tensorflow, pp. 63–72, 2021. doi:10.1007/978-1-4842-6168-2_6

[81] S. Mascarenhas and M. Agarwal, "A comparison between VGG16, VGG19 and ResNet50 architecture frameworks for Image Classification," 2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON), Bengaluru, India, 2021, pp. 96-99, doi: 10.1109/CENTCON52345.2021.9687944.

[82] S. I. Safie, N. S. A. Kamal, E. M. M. Yusof, M. Z. -W. M. Tohid and N. H. Jaafar, "Comparison of SqueezeNet and DarkNet-53 based YOLO-V3 Performance for Beehive Intelligent Monitoring System," 2023 IEEE 13th Symposium on Computer Applications & Industrial Electronics (ISCAIE), Penang, Malaysia, 2023, pp. 62-65, doi: 10.1109/ISCAIE57739.2023.10165285.

[83] J. Amin et al., “Convolutional Bi-LSTM based human gait recognition using video sequences,” Computers, Materials & Continua, vol. 68, no. 2, pp. 2693–2709, 2021. doi:10.32604/cmc.2021.016871

[84] "Getting Started with Object Detection Using Deep Learning," MathWorks, https://www.mathworks.com/help/vision/ug/getting-started-with-object-detection-using-deep-learning.html (Accessed 2023).

APPENDIX A: Faster R-CNN MATLAB® Source Code

```
% Code adapted from MathWorks® example:
% https://www.mathworks.com/help/vision/ref/trainfasterrcnnobjectdetector.html#bvkk009-7

% -------------------------------------------------------------------
% Load Dataset
% -------------------------------------------------------------------

unzip datasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;

rng(0)
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * height(vehicleDataset));

trainingIndex = 1:idx;
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIndex),:);

testIndex = trainingIndex(end)+1 : length(shuffledIndices);
testDataTbl = vehicleDataset(shuffledIndices(testIndex),:);

trainImageDatastore = imageDatastore(trainingDataTbl{:,'imageFilename'});
trainBoxLabelDatastore = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

testImageDatastore = imageDatastore(testDataTbl{:,'imageFilename'});
testBoxLabelDatastore = boxLabelDatastore(testDataTbl(:,'vehicle'));

trainingData = combine(trainImageDatastore,trainBoxLabelDatastore);
testData = combine(testImageDatastore,testBoxLabelDatastore);

data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)

% -------------------------------------------------------------------
% Create Faster R-CNN Detection Network
% -------------------------------------------------------------------

inputSize = [224 224 3];

preprocessedTrainingData = transform(trainingData,
@(data)preprocessData(data,inputSize));
numAnchors = 3;
anchorBoxes = estimateAnchorBoxes(preprocessedTrainingData,numAnchors)
```

```
featureExtractionNetwork = resnet50;

featureLayer = 'activation_40_relu';

numClasses = width(vehicleDataset)-1;

lgraph =
fasterRCNNLayers(inputSize,numClasses,anchorBoxes,featureExtractionNetwor
k,featureLayer);

% ------------------------------------------------------------------
% Data Augmentation
% ------------------------------------------------------------------

augmentedTrainingData = transform(trainingData,@augmentData);

augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},rectangle = data{2});
    reset(augmentedTrainingData);
end

figure
montage(augmentedData,BorderSize = 10)

% ------------------------------------------------------------------
% Preprocess Training Data
% ------------------------------------------------------------------

preprocessedTrainingData =
transform(augmentedTrainingData,@(data)preprocessData(data,inputSize));

data = read(preprocessedTrainingData);

I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)

% ------------------------------------------------------------------
% Train Faster R-CNN
% ------------------------------------------------------------------

options = trainingOptions('sgdm',...
    MaxEpochs = 10,...
    MiniBatchSize = 2,...
    InitialLearnRate = 1e-3,...
    VerboseFrequency = 1, ...
    CheckpointPath = tempdir,...
    Shuffle = every-epoch', ...
    Plots = training-progress');

[detector, info] =
trainFasterRCNNObjectDetector(preprocessedTrainingData,lgraph,options,'Ne
gativeOverlapRange',[0 0.3],'PositiveOverlapRange',[0.6 1]);
```

```
counter = 0;
for j = 1:261
    I = imread(testDataTbl.imageFilename{j});
    I = imresize(I,inputSize(1:2));
    [bboxes,scores] = detect(detector,I);
    n = testDataTbl.imageFilename{j};
    y = 0;

    startPat = wildcardPattern + "/";
    endPat = ".jpg";
    newStr = extractBetween(n,startPat,endPat);
    iName = "faster_rcnn_" + newStr;

    try
       I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
    catch
       fprintf('Error: %d | %s | %s\n',j,scores,n);
       counter=counter+1;
       y=1;
    end
end
fprintf('Total Errors: %d \n',counter);

% ------------------------------------------------------------------
% Evaluate Detector Using Test Set
% ------------------------------------------------------------------

testData = transform(testData,@(data)preprocessData(data,inputSize));

detectionResults = detect(detector,testData,'MinibatchSize',4);

[ap, recall, precision] =
evaluateDetectionPrecision(detectionResults,testData);

figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f', ap))
```

APPENDIX B: YOLO v3 MATLAB® Source Code

```
% Code adapted from MathWorks® example:
% https://www.mathworks.com/help/vision/ref/trainyolov3objectdetector.html#mw_00a733fc-e3fe-4cc3-9b13-74233dbead5a

% ---------------------------------------------------------------------
% Load Dataset
% ---------------------------------------------------------------------

unzip datasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;

rng(0)
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * height(vehicleDataset));

trainingIndex = 1:idx;
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIndex),:);

testIndex = trainingIndex(end)+1 : length(shuffledIndices);
testDataTbl = vehicleDataset(shuffledIndices(testIndex),:);

trainImageDatastore = imageDatastore(trainingDataTbl{:,'imageFilename'});
trainBoxLabelDatastore = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

testImageDatastore = imageDatastore(testDataTbl{:,'imageFilename'});
testBoxLabelDatastore = boxLabelDatastore(testDataTbl(:,'vehicle'));

trainingData = combine(trainImageDatastore,trainBoxLabelDatastore);
testData = combine(testImageDatastore,testBoxLabelDatastore);

data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)

% ---------------------------------------------------------------------
% Create YOLO v3 Object Detector
% ---------------------------------------------------------------------

networkInputSize = [227 227 3];

rng(0)
trainingDataForEstimation = transform(trainingData,
@(data)preprocessData(data, networkInputSize));
numAnchors = 6;
[anchors, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation,
numAnchors)
```

```
area = anchors(:, 1).*anchors(:, 2);
[~, idx] = sort(area, 'descend');
anchors = anchors(idx, :);
anchorBoxes = {anchors(1:3,:)
    anchors(4:6,:)
    };

baseNetwork = squeezenet;
classNames = trainingDataTbl.Properties.VariableNames(2:end);

yolov3Detector = yolov3ObjectDetector(baseNetwork, classNames,
anchorBoxes, 'DetectionNetworkSource', {'fire9-concat', 'fire5-concat'},
InputSize = networkInputSize);

% ------------------------------------------------------------------
% Data Augmentation
% ------------------------------------------------------------------

augmentedTrainingData = transform(trainingData,@augmentData);

augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},rectangle = data{2});
    reset(augmentedTrainingData);
end

figure
montage(augmentedData,BorderSize = 10)

% ------------------------------------------------------------------
% Preprocess Training Data
% ------------------------------------------------------------------

preprocessedTrainingData = transform(augmentedTrainingData,
@(data)preprocess(yolov3Detector, data));

data = read(preprocessedTrainingData);

I = data{1,1};
bbox = data{1,2};
annotatedImage = insertShape(I, 'Rectangle', bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)

reset(preprocessedTrainingData);

% ------------------------------------------------------------------
% Train Model
% ------------------------------------------------------------------

options = trainingOptions('sgdm', ...
        MiniBatchSize = 8, ....
        InitialLearnRate = 1e-3, ...
        MaxEpochs = 80, ...
        VerboseFrequency = 1, ...
        CheckpointPath = tempdir, ...
        Shuffle = 'every-epoch', ...
```

```
        Plots='training-progress');

for epoch = 1:numEpochs

      [gradients, state, lossInfo] = dlfeval(@modelGradients,
yolov3Detector, XTrain, YTrain, penaltyThreshold);
      gradients = dlupdate(@(g,w) g + l2Regularization*w, gradients,
yolov3Detector.Learnables);
      currentLR = piecewiseLearningRateWithWarmup(iteration, epoch,
learningRate, warmupPeriod, numEpochs);

      [yolov3Detector.Learnables, velocity] =
sgdmupdate(yolov3Detector.Learnables, gradients, velocity, currentLR);
end

counter = 0;
for j = 1:261
    I = imread(imdsTest.Files{j});
    [bboxes,scores,labels] = detect(yolov3Detector,I);
    n = imdsTest.Files{j};
    y = 0;

    startPat = wildcardPattern + "vehicleImages\";
    endPat = ".jpg";
    newStr = extractBetween(n,startPat,endPat);
    iName = "yolo_v3_" + newStr;

    try
       I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
    catch
       fprintf('Error: %d | %s | %s\n',j,scores,n);
       counter=counter+1;
       y=1;
    end
end
fprintf('Total Errors: %d \n',counter);

% ------------------------------------------------------------------
% Evaluate Model
% ------------------------------------------------------------------

results = detect(yolov3Detector,testData,'MiniBatchSize',8);

% Evaluate the object detector using Average Precision metric.
[ap,recall,precision] = evaluateDetectionPrecision(results,testData);

% Plot precision-recall curve.
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f', ap))
```

APPENDIX C: SSD MATLAB® Source Code

```
% Code adapted from MathWorks® example:
% https://www.mathworks.com/help/vision/ref/trainssdobjectdetector.html#mw_aadddb45-dcc1-4968-8d1d-d9ec600dc23e

% ---------------------------------------------------------------------
% Load Dataset
% ---------------------------------------------------------------------

unzip datasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;

rng(0)
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * height(vehicleDataset));

trainingIndex = 1:idx;
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIndex),:);

testIndex = trainingIndex(end)+1 : length(shuffledIndices);
testDataTbl = vehicleDataset(shuffledIndices(testIndex),:);

trainImageDatastore = imageDatastore(trainingDataTbl{:,'imageFilename'});
trainBoxLabelDatastore = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

testImageDatastore = imageDatastore(testDataTbl{:,'imageFilename'});
testBoxLabelDatastore = boxLabelDatastore(testDataTbl(:,'vehicle'));

trainingData = combine(trainImageDatastore,trainBoxLabelDatastore);
testData = combine(testImageDatastore,testBoxLabelDatastore);

data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)

% ---------------------------------------------------------------------
% Create SSD Object Detection Network
% ---------------------------------------------------------------------

net = resnet50();
lgraph = layerGraph(net);

inputSize = [300 300 3];

classNames = {'vehicle'};

idx = find(ismember({lgraph.Layers.Name},'activation_40_relu'));
```

```
removedLayers = {lgraph.Layers(idx+1:end).Name};
ssdLayerGraph = removeLayers(lgraph,removedLayers);

weightsInitializerValue = 'glorot';
biasInitializerValue = 'zeros';

extraLayers = [];

filterSize = 1;
numFilters = 256;
numChannels = 1024;
conv6_1 = convolution2dLayer(filterSize, numFilters, NumChannels =
numChannels, ...
    Name = 'conv6_1', ...
    WeightsInitializer = weightsInitializerValue, ...
    BiasInitializer = biasInitializerValue);
relu6_1 = reluLayer(Name = 'relu6_1');
extraLayers = [extraLayers; conv6_1; relu6_1];

filterSize = 3;
numFilters = 512;
numChannels = 256;
conv6_2 = convolution2dLayer(filterSize, numFilters, NumChannels =
numChannels, ...
    Padding = iSamePadding(filterSize), ...
    Stride = [2, 2], ...
    Name = 'conv6_2', ...
    WeightsInitializer = weightsInitializerValue, ...
    BiasInitializer = biasInitializerValue);
relu6_2 = reluLayer(Name = 'relu6_2');
extraLayers = [extraLayers; conv6_2; relu6_2];

filterSize = 1;
numFilters = 128;
numChannels = 512;
conv7_1 = convolution2dLayer(filterSize, numFilters, NumChannels =
numChannels, ...
    Name = 'conv7_1', ...
    WeightsInitializer = weightsInitializerValue, ...
    BiasInitializer = biasInitializerValue);
relu7_1 = reluLayer(Name = 'relu7_1');
extraLayers = [extraLayers; conv7_1; relu7_1];

filterSize = 3;
numFilters = 256;
numChannels = 128;
conv7_2 = convolution2dLayer(filterSize, numFilters, NumChannels =
numChannels, ...
    Padding = iSamePadding(filterSize), ...
    Stride = [2, 2], ...
    Name = 'conv7_2', ...
    WeightsInitializer = weightsInitializerValue, ...
    BiasInitializer = biasInitializerValue);
relu7_2 = reluLayer(Name = 'relu7_2');
extraLayers = [extraLayers; conv7_2; relu7_2];

filterSize = 1;
numFilters = 128;
numChannels = 256;
```

```
conv8_1 = convolution2dLayer(filterSize, numFilters, NumChannels =
numChannels, ...
    Name = 'conv8_1', ...
    WeightsInitializer = weightsInitializerValue, ...
    BiasInitializer = biasInitializerValue);
relu8_1 = reluLayer(Name = 'relu8_1');
extraLayers = [extraLayers; conv8_1; relu8_1];

filterSize = 3;
numFilters = 256;
numChannels = 128;
conv8_2 = convolution2dLayer(filterSize, numFilters, NumChannels =
numChannels, ...
    Name = 'conv8_2', ...
    WeightsInitializer = weightsInitializerValue, ...
    BiasInitializer = biasInitializerValue);
relu8_2 = reluLayer(Name ='relu8_2');
extraLayers = [extraLayers; conv8_2; relu8_2];

filterSize = 1;
numFilters = 128;
numChannels = 256;
conv9_1 = convolution2dLayer(filterSize, numFilters, NumChannels =
numChannels, ...
    Padding = iSamePadding(filterSize), ...
    Name = 'conv9_1', ...
    WeightsInitializer = weightsInitializerValue, ...
    BiasInitializer = biasInitializerValue);
relu9_1 = reluLayer('Name', 'relu9_1');
extraLayers = [extraLayers; conv9_1; relu9_1];

if ~isempty(extraLayers)
    lastLayerName = ssdLayerGraph.Layers(end).Name;
    ssdLayerGraph = addLayers(ssdLayerGraph, extraLayers);
    ssdLayerGraph = connectLayers(ssdLayerGraph, lastLayerName,
extraLayers(1).Name);
end

detNetworkSource = ["activation_22_relu", "activation_40_relu",
"relu6_2", "relu7_2", "relu8_2"];

anchorBoxes = {[60,30;30,60;60,21;42,30];...
               [111,60;60,111;111,35;64,60;111,42;78,60];...
               [162,111;111,162;162,64;94,111;162,78;115,111];...
               [213,162;162,213;213,94;123,162;213,115;151,162];...
               [264,213;213,264;264,151;187,213]};

detector =
ssdObjectDetector(ssdLayerGraph,classNames,anchorBoxes,DetectionNetworkSo
urce=detNetworkSource,InputSize=inputSize,ModelName='ssdVehicle');

% ------------------------------------------------------------------
% Data Augmentation
% ------------------------------------------------------------------

augmentedTrainingData = transform(trainingData,@augmentData);

augmentedData = cell(4,1);
for k = 1:4
```

```
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},rectangle = data{2});
    reset(augmentedTrainingData);
end

figure
montage(augmentedData,BorderSize = 10)

% ---------------------------------------------------------------------
% Preprocess Training Data
% ---------------------------------------------------------------------

preprocessedTrainingData =
transform(augmentedTrainingData,@(data)preprocessData(data,inputSize));

data = read(preprocessedTrainingData);

I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)

% ---------------------------------------------------------------------
% Train SSD Object Detector
% ---------------------------------------------------------------------

options = trainingOptions('sgdm', ...
        MiniBatchSize = 4, ....
        InitialLearnRate = 1e-3, ...
        MaxEpochs = 20, ...
        VerboseFrequency = 1, ...
        CheckpointPath = tempdir, ...
        Shuffle = 'every-epoch', ...
        Plots='training-progress');

[detector, info] =
trainSSDObjectDetector(preprocessedTrainingData,detector,options);

counter = 0;
for j = 1:261
    I = imread(imdsTest.Files{j});
    I = imresize(I,inputSize(1:2));
    [bboxes,scores] = detect(detector,I);
    n = imdsTest.Files{j};
    y = 0;

    startPat = wildcardPattern + "vehicleImages\";
    endPat = ".jpg";
    newStr = extractBetween(n,startPat,endPat);
    iName = "ssd_" + newStr;

    try
       I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
    catch
       fprintf('Error: %d | %s | %s\n',j,scores,n);
       counter=counter+1;
       y=1;
```

```
    end
end
fprintf('Total Errors: %d \n',counter);

% ---------------------------------------------------------------------
% Evaluate Detector Using Test Set
% ---------------------------------------------------------------------

preprocessedTestData =
transform(testData,@(data)preprocessData(data,inputSize));

detectionResults = detect(detector, preprocessedTestData, MiniBatchSize =
32);

[ap,recall,precision] = evaluateDetectionPrecision(detectionResults,
preprocessedTestData);

figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f',ap))
```