Object Detection Using YOLO Algorithms and it's Hardware Implementation.

A project report submitted in partial fulfillment of the requirements for the award of the degree of

B.Tech.

in

Electronics and Communication Engineering

By

K. KOUSHIK (620144)

M. VEKSHITH (620206)

K. JAI KRISHNA (620148)

Department of Electronics and Communication Engineering

NATIONAL INSTITUTE OF TECHNOLOGY
ANDHRA PRADESH-534101

APRIL 2024

BONAFIDE CERTIFICATE

This is to certify that the project titled **Object Detection Using YOLO Algorithms and it's Hardware Implementation.** is a bonafide record of the work done by

K. KOUSHIK (620144)

M. VEKSHITH (620206)

K. JAI KRISHNA (620148)

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **ECE** of the **NATIONAL INSTITUTE OF TECHNOLOGY, ANDHRA PRADESH**, during the year 2023-2024.

Dr.Puli Kishore Kumar	**Dr.S.Yuvaraj**
Assistant Professor	Assistant Professor
Project Guide	Head of the Department

ABSTRACT

Our project delves into YOLO Algorithm, a cutting-edge deep learning algorithm renowned for real-time object detection. It provides an in-depth examination of YOLO's architecture, training procedures, and deployment strategies. Comparative analysis with other methods offers insights into its efficacy. Practical implementation aspects, including dataset preparation, fine-tuning, and optimization for real-time performance, are explored. Evaluation metrics such as precision, recall, and mAP are employed for performance assessment. Additionally, advanced techniques like transfer learning and model compression are investigated to enhance efficiency. Real-world deployment challenges and integration strategies are discussed. Overall, this project offers a comprehensive understanding of YOLO's capabilities in object detection, catering to researchers and practitioners in computer vision and deep learning. Implementation code and trained models are provided for further experimentation.

Keywords:YOLO (You Only Look Once), Field-Programmable Gate Array (FPGA), Object Detection, Real-Time Detection, Parallel Processing.

ACKNOWLEDGEMENT

We would like to thank the following people for their support and guidance without whom the completion of this project in fruition would not be possible.

Dr.Puli Kishore Kumar, our project guide, guiding us in the course of this project. We are grateful that sir has provided us with constant support with resources and his invaluable insights. Thank you for being there always whenever there is a need. Thank you for believing in us by providing 24X7 Lab access.

Dr. S. Yuvaraj, the Head of the Department, Department of ECE. For making sure that reviews are conducted smoothly.

Our internal reviewers, **Mr.Kondala Rao Jyothi**, **Dr.Thulyasya Naik B**, **Dr.M Ananda Reddy** for their insight and advice provided during the review sessions and being available for doubt clarification.

We would also like to thank our individual parents and friends for their constant support.

TABLE OF CONTENTS

List of Figures

Chapter 1

Introduction

1.1 General Introduction

Object detection is a crucial area in computer vision, involving the identification and localization of objects within images or videos. It has applications in a wide range of fields, including autonomous vehicles, surveillance, robotics, and medical imaging. The advent of deep learning has significantly advanced object detection capabilities, with algorithms like YOLO (You Only Look Once) achieving real-time performance and high accuracy. YOLOv2, the second iteration of this algorithm, introduces improvements over its predecessor, focusing on speed and versatility while maintaining accuracy.

In recent years, deep learning-based object detection has gained significant traction across a multitude of applications, from autonomous vehicles to security surveillance. Among the prominent architectures, You Only Look Once (YOLO) has emerged as a popular choice due to its real-time processing capability and high accuracy. Specifically, YOLOv2 is known for its balance between speed and performance, making it a favorable model for a wide range of object detection tasks.

This project explores the impact of various key parameters on the performance of YOLOv2, with a focus on optimization algorithms, number of epochs, mini-batch sizes, and initial learning rates. The study aims to understand how these parameters influence the model's accuracy, training time, and convergence behavior. By systematically analyzing the effects of each parameter, this project seeks to provide insights and practical recommendations for practitioners working with YOLOv2 in diverse application settings.

1.2 Motivation

The motivation for this project stems from the growing demand for efficient and accurate object detection in real-time applications. As industries increasingly adopt automation and AI-driven technologies, the ability to quickly and accurately detect objects becomes critical. YOLOv2, with its single-shot detection approach and anchor-based bounding box prediction, is a strong candidate for such tasks, offering a balance between speed and accuracy. Despite its potential, YOLOv2's performance can be significantly influenced by key training parameters, such as optimization algorithms, number of epochs, mini-batch sizes, and learning rates. Understanding the impact of these parameters is crucial to optimizing YOLOv2 for specific use cases, ensuring the algorithm performs effectively in various environments and conditions.

1.3 Problem Statement

While YOLOv2 offers real-time object detection capabilities, its performance can vary based on several training and hyperparameter settings. The lack of comprehensive analysis on the impact of these parameters makes it challenging for practitioners to optimize YOLOv2 for their specific applications. This project's problem statement is: How do different optimization algorithms, number of epochs, mini-batch sizes, and initial learning rates affect the performance of YOLOv2 in object detection tasks?

Chapter 2

Review Of Literature

You Only Look Once: Unified, Real-Time Object Detection By Joseph Redmon , Ali Farhadi

In this paper[1] , A method streamlined the detection process by directly predicting bounding boxes and class probabilities from full images, eliminating the need for intermediate steps and post-processing. This not only simplified the detection pipeline but also reduced the risk of error propagation. Additionally, their approach offered potential efficiency gains by consolidating the detection process into a single network, leading to faster inference times and reduced computational complexity.

A NOVEL METHOD FOR MULTIPLE OBJECT DETECTION ON ROAD USING IMPROVED YOLOV2 MODEL, 2022

In this paper[2], The authors adeptly address several challenges prevalent in existing vehicle detection methodologies, including the lack of vehicle-type recognition, low detection accuracy, and sluggish processing speeds. By leveraging YOLOv2 architecture, renowned for its efficiency and effectiveness in object detection tasks, the proposed model exhibits notable improvements in these critical areas. The model was trained using VOC and COCO datasets and evaluated using KITTI training images.

YOLOv2 based Real Time Object Detection. Published by Sakshi Gupta , Dr. T. Uma Devi

In this paper[3], A new approcah to real-time object detection and classification in computer vision using You Only Look Once version 2 (YOLOv2). By leveraging YOLOv2 and the COCO-2017 dataset, the authors achieve faster computation times, processing

up to 40 frames per second. Their method addresses the challenges of slow processing and errors encountered with small datasets, offering a practical solution for swift and accurate object detection.

Multi-object recognition method based on improved yolov2 model, 2021 Published by Xun Li*, Binbin Shi, Tingting Nie, Kaibin Zhang, Wenjie Wang

In this paper[4], a method of vehicle multi-object identification and classification based on the YOLOv2 algorithm is proposed, which is used to solve the classical multi-object classification problems of low detection rate, poor robustness and unsatisfactory effect on real road environment. It analyzed vehicle objective and training results. The network structure of YOLOv2-voc is improved according to the actual road conditions based on the YOLOv2 algorithm, and the classification training model was obtained by the ImageNet data which is came from many tweaks.

An Improved YOLOv2 for Vehicle Detection by Jun Sang , Zhongyuan Wu , Pei Guo , Haibo Hu , Hong Xiang , Qian Zhang and Bin Cai

This paper introduces a new vehicle detection model called YOLOv2-Vehicle to address issues in existing systems such as low accuracy and slow speed. They utilized the k-means++ clustering algorithm to select anchor boxes and implemented normalization techniques for precise bounding box dimension calculations. Additionally, they adopted a multi-layer feature fusion strategy and streamlined convolution layers to enhance feature extraction. Overall, their model improves accuracy, speed, and adaptability in vehicle detection tasks.

Chapter 3

FPGA Implementation of Basic Image Processing techniques

3.1 Realtime Image Processing on FPGA

Our project ”Real-time Image Processing on FPGA using Xilinx System Generator” involved comprehensive exploration and implementation of image processing techniques on FPGA platforms. Both hardware and software components were developed to execute various image processing operations including contrast stretching, image thresholding figure 3.4, and image negation figure 3.3. These techniques were implemented and compared between the hardware and software domains to analyze performance and efficiency differences. We have shown point-to-point Model in figure 3.1. Hardware Implementation in figure 3.2.

To optimize storage space without compromising processing quality, We introduced the concept of image compression before processing. Discrete Cosine Transform (DCT) compression was selected as the method of choice, requiring matrix multiplication processes.

Leveraging the Xilinx System Generator facilitated hardware design and simulation, enabling seamless integration of image processing algorithms into FPGA architectures. Rigorous testing and analysis were conducted to validate the functionality and accuracy of the implemented techniques, ensuring reliability and robustness in real-time applications.

The project highlighted the versatility of FPGA platforms in handling complex image processing tasks with real-time constraints. By exploring compression techniques alongside traditional image processing operations, We demonstrated innovative approaches to optimizing storage space while maintaining processing efficiency.

Overall, the project not only showcased Our team's technical proficiency in FPGA-based image processing but also underscored the potential for FPGA technologies to drive advancements in fields such as computer vision, embedded systems, and real-time image processing applications.

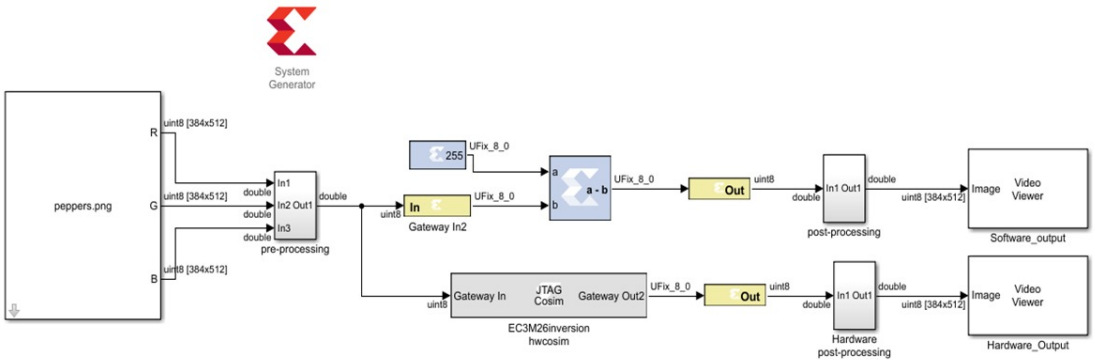

Figure 3.1: Architecture of Image Processing

3.2 Results

3.2.1 Image Processing

Figure 3.2: Hardware Implementation

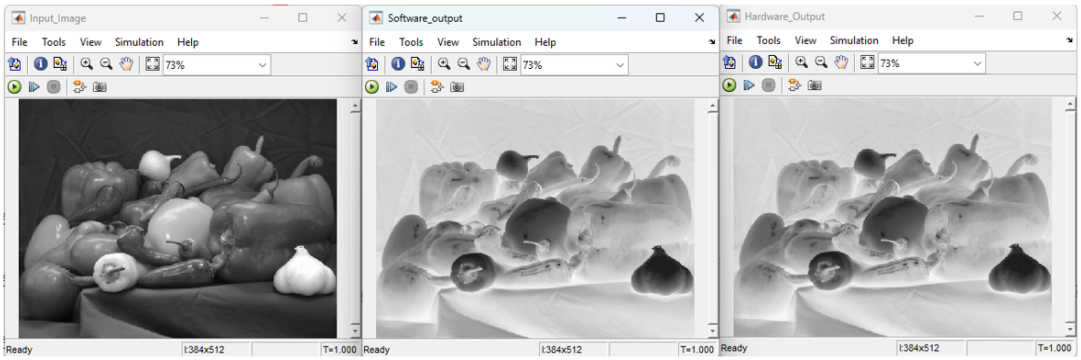

Figure 3.3: Image Negative

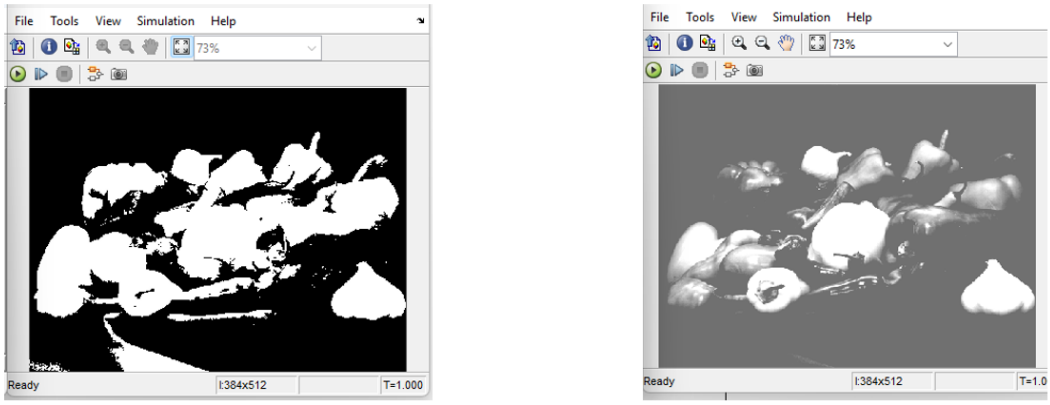

Figure 3.4: Image Threshold and Image Constrast Stretching

3.2.2 DCT Compression

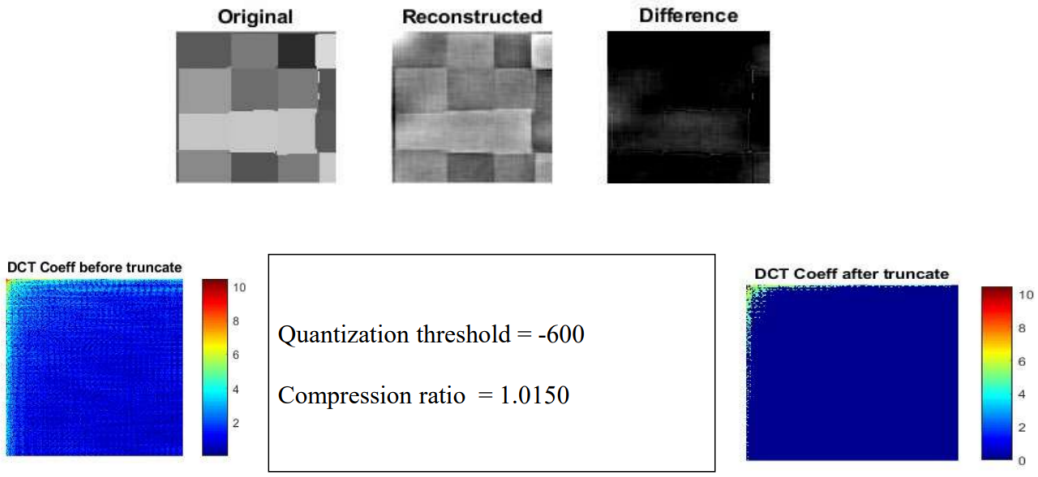

Figure 3.5: DCT Compression

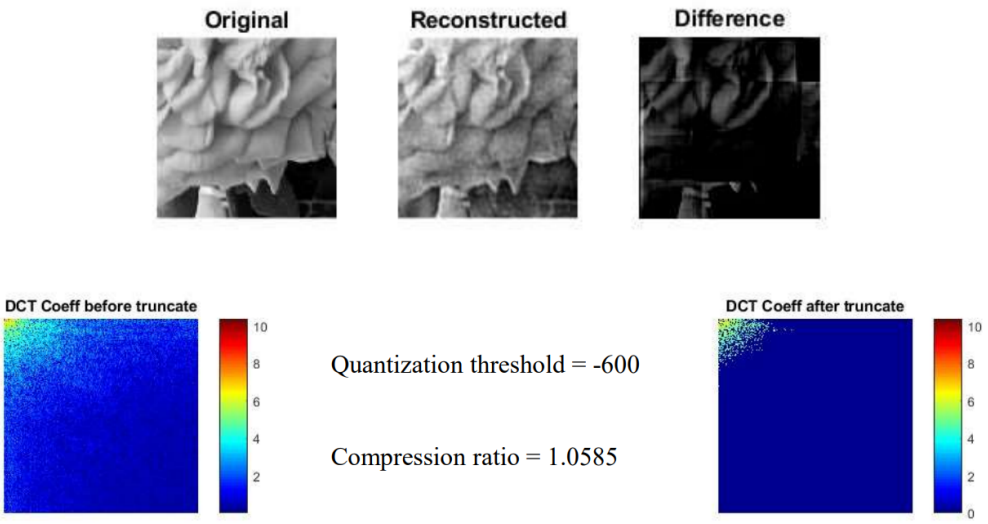

Figure 3.6: DCT Compression

Chapter 4

ANALYSIS OF YOLOv2

4.1 YOLOv2 ALGORITHM

YOLOv2 (You Only Look Once, Version 2) is an object detection algorithm that builds on the original YOLO with significant improvements in accuracy and robustness while maintaining real-time detection speed. The architecture and algorithm are designed to detect multiple objects in an image with a single forward pass through a neural network. YOLOv2 treats object detection as a single regression problem, allowing it to detect objects quickly and accurately in one pass. The algorithm divides the input image into a grid and assigns responsibility for detecting objects to each grid cell. The key concept is that the entire detection process, from locating objects to classifying them, is handled by a single convolutional neural network.

4.2 YOLOv2 ARCHITECTURE

4.2.1 ReNet-50

ResNet-50, a pivotal convolutional neural network architecture, offers unparalleled performance in image classification tasks. Developed by Kaiming He et al., its deep structure employs skip connections, alleviating the vanishing gradient problem and enabling training of exceptionally deep networks. With 50 layers, ResNet-50's residual blocks facilitate direct information flow, leading to superior results on benchmark datasets like ImageNet. Widely recognized for its efficiency and effectiveness, ResNet-50 stands as a cornerstone model in deep learning, extensively utilized across various applications,

from object detection to feature extraction.

4.2.2 Batch Normalization

YOLOv2 incorporates batch normalization throughout the network. This technique normalizes activations during training, reducing internal covariate shift and enabling faster convergence and more stable training.

4.2.3 Anchor Boxes

A significant change in YOLOv2 is the introduction of anchor boxes. Instead of predicting bounding boxes directly, the network predicts offsets relative to predefined anchor boxes. This allows the network to handle objects of various shapes and sizes more effectively.

4.2.4 Multi-Scale Detection

YOLOv2 supports multi-scale detection, allowing it to adapt to objects of different sizes. During training, the network is exposed to images of varying scales, enhancing its ability to generalize across different object sizes.

4.2.5 Bounding Box Prediction

Each grid cell in YOLOv2 is responsible for predicting a certain number of bounding boxes (typically 5). Each bounding box prediction includes: **Coordinates**: The center (x, y), width, and height relative to the grid cell. **Objectness Score**: The confidence that a bounding box contains an object. **Class Probabilities:** The probabilities for each class, indicating the likelihood of an object belonging to a specific class.

4.2.6 Loss Function

YOLOv2 uses a multi-part loss function that considers the following: Bounding Box Coordinates: The error between predicted and ground truth coordinates. **Objectness Score:** The confidence in object detection. **Class Probabilities:** The accuracy of object classification.

4.2.7 Non-Maximum Suppression

After the network's output, YOLOv2 applies non-maximum suppression (NMS) to remove redundant bounding boxes, ensuring that only the most confident predictions are kept.

4.2.8 Output Layer

The output of YOLOv2 consists of bounding boxes, objectness scores, and class probabilities. The output is further processed to identify and classify detected objects.

4.3 Optimization Techniques

The Exponentially Weighted Moving Average (EWMA) formula is given by:

$$v_t = \beta v_{t-1} + (1 - \beta)\theta_t \tag{4.1}$$

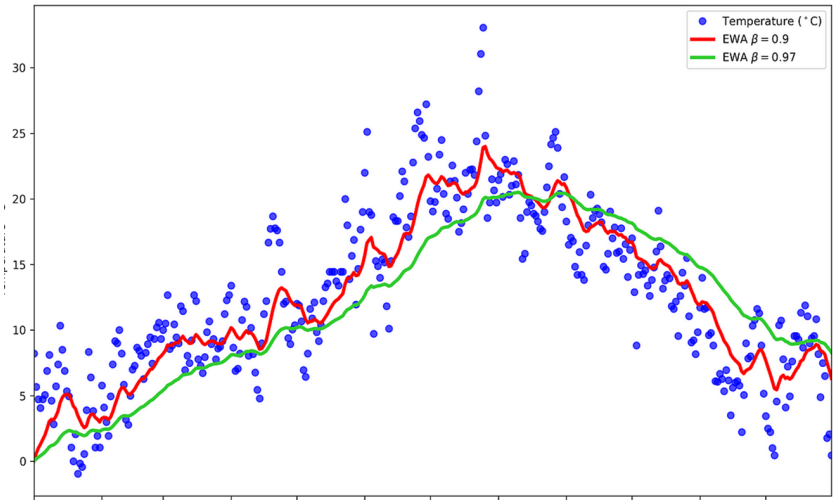

Figure 4.1: EWMA

Here, v_t represents the average at time stamp "t", and v_{t-1} represents the previous average. The parameter β is a hyperparameter that controls the smoothness of the curve. The value of β ranges from 0 to 1: A higher value of β (closer to 1) leads to a smoother curve. A lower value of β (closer to 0) gives more weight to recent data, causing the average to fluctuate more. θ_t represents the new data point at time stamp "t". The initialization condition is $v_0 = 0$.

The first few iterations of the EWMA are:

$$v_1 = \beta v_0 + (1 - \beta)\theta_1 \tag{4.2}$$

$$v_2 = \beta v_1 + (1 - \beta)\theta_2 \tag{4.3}$$

$$v_3 = \beta v_2 + (1 - \beta)\theta_3 \tag{4.4}$$

The general form for EWMA at time "t" can be written as:

$$v_t = (1 - \beta) \sum_{i=0}^{t-1} \beta^i \theta_{t-i} \tag{4.5}$$

This expression shows that older data points are given less weight as β is raised to higher powers. Setting $\beta = 0.9$ is often considered for better results, providing a balance between smoothing and responsiveness.

4.4 Introduction to Gradient Descent

Gradient descent is an optimization algorithm that's used when training a machine learning model. It's based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum. A gradient simply measures the change in all weights with regard to the change in error. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. In mathematical terms, a gradient is a partial derivative with respect to its inputs.

Initialize Weights: Start with an initial guess for the weights W.

Compute Gradient: Calculate the gradient of the function J(W) with respect to the weights W.

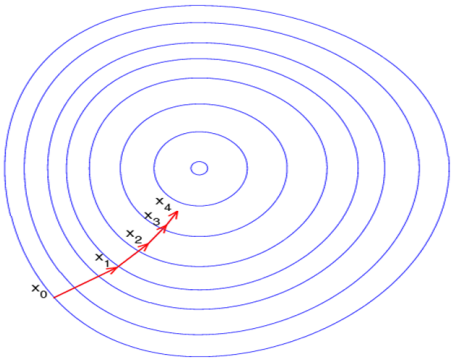

Figure 4.2: Gradient Descent

Update Weights: Adjust the weights in the opposite direction of the gradient to minimize J(W):

$$W = W - \alpha \cdot \nabla J(W) \tag{4.6}$$

where α is the learning rate, a small positive scalar.

Repeat: Iterate steps 2 and 3 until convergence criteria are met.

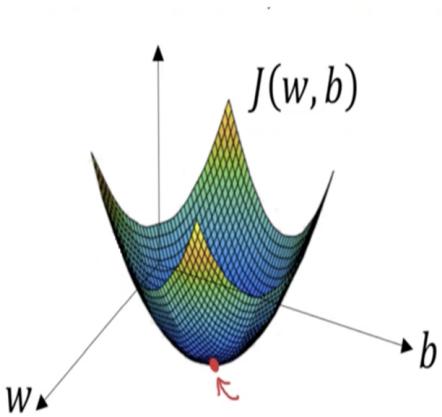

Figure 4.3: Gradient Descent

4.5 Types of Gradient Descent

There are three popular types of gradient descent that mainly differ in the amount of data they use.

1. Batch Gradient Descent

2. Stochastic Gradient Descent

3. Mini-Batch Gradient Descent

4.5.1 Batch Gradient Descent

Calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated.

This whole process is like a cycle and it's called a training epoch.

batch gradient descent are computational efficient.Sometimes result in a state of convergence that isn't the best the model can achieve. It also requires the entire training dataset to be in memory and available to the algorithm.

$$\theta = \theta - \eta \cdot \nabla J(\theta) \tag{4.7}$$

4.5.2 Stochastic Gradient Descent

SGD does this for each training example within the dataset, updates the parameters for each training example one by one.

$$\theta = \theta - \eta \cdot \nabla J(\theta; x^i; y^i) \tag{4.8}$$

This can make SGD faster than batch gradient descent. Frequent updates allow us to have a pretty detailed rate of improvement. Due to frequent updates are more computationally expensive than the batch gradient descent approach. The frequency of those updates can result in noisy gradients, which may cause the error rate to jump around instead of slowly decreasing.

4.5.3 Mini-Batch Gradient Descent

Combination of the concepts of SGD and batch gradient descent.It splits the training dataset into small batches and performs an update for each of those batches.This creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. Common mini-batch sizes range between 50 and 256.

$$\theta = \theta - \eta \cdot \nabla J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{4.9}$$

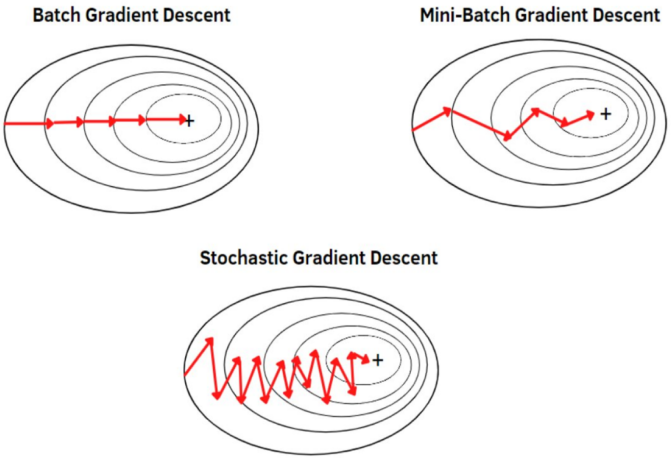

Figure 4.4: Different types Gradient Descent

4.6 Momentum Optimizer

Momentum optimization is an extension of gradient descent that helps accelerate convergence and navigate through saddle points more effectively by adding momentum to the parameter updates.Equation for Parameter Update (Momentum):

$$\nu := \beta\nu + (1 - \beta)\nabla J(W) \tag{4.10}$$

$$W := W - \alpha\nu \tag{4.11}$$

v is the momentum term.

β is the momentum coefficient.

$\nabla J(W)$ is the gradient of the loss function with respect to the parameters W. α is the learning rate.

4.7 RMSprop Optimizer

RMSprop (Root Mean Square Propagation) is an adaptive learning rate optimization algorithm that adjusts the learning rates for each parameter based on the magnitudes o

Let v represent the moving average of squared gradients.

$$v := \beta v + (1 - \beta) \cdot (\nabla J(W))^2 \tag{4.12}$$

$$W := W - \frac{\alpha}{\sqrt{v} + \epsilon} \cdot \nabla J(W) \tag{4.13}$$

The parameter β is the decay rate for the moving average.

$\nabla J(W)$ denotes the gradient of the loss function with respect to the parameters W.

The learning rate is given by α.

A small constant $\epsilon = 10^{-8}$ is used to prevent division by zero.f recent gradients.

4.8 ADAM optimizer

Adaptive Moment Estimation is an optimization algorithm commonly used for training deep learning models. It combines the advantages of two other popular optimization algorithms, namely, RMSprop and Momentum. ADAM is computationally efficient and requires minimal memory. It automatically adapts the learning rate for each parameter, reducing the need for manual tuning.

Compute gradients:

$$g = \nabla J(W) \tag{4.14}$$

$$m := \beta_1 \cdot m + (1 - \beta_1) \cdot g \tag{4.15}$$

$$v := \beta_2 \cdot v + (1 - \beta_2) \cdot g^2 \tag{4.16}$$

$$W := W - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} \tag{4.17}$$

Chapter 5

Object Detection Using YOLOv2

5.1 Introduction

In this project we are making use of two SubNets. ResNet-50 is used for Features extraction from the input images, whereas later SubNet that is YOLO v2 is used for Object Detection. Here we have tried to detect vehicles along with confidence score. Analysed training by changing Optimizers and tuning various parameters.

5.1.1 Load Dataset

In this we uses a small vehicle dataset that contains 295 images. Folder size 3.49MB. Each image contains one or two labeled instances of a vehicle. The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes.

5.1.2 Organizing Data

Split the dataset into training, validation, and test sets. Select 60 percentage of the data for training, 10 percentage for validation, and the rest for testing the trained detector. Use imageDatastore and boxLabelDatastore to create datastores for loading the image

	imageFilename	vehicle
1	'vehicleImages/image_00001.jpg'	[220,136,35,28]
2	'vehicleImages/image_00002.jpg'	[175,126,61,45]
3	'vehicleImages/image_00003.jpg'	[108,120,45,33]
4	'vehicleImages/image_00004.jpg'	[124,112,38,36]

Figure 5.1: Dataset

Figure 5.2: Random Images 1

and label data during training and evaluation. Namely imdsTrain and bldsTrain for Training similarly we can do for validation and testing Combine image and box label datastores.

5.1.3 Preprocessing of Data Before Training

ResNet-50 input Layer size is [224 224 3] meaning that it will take a input image of size 224 by 224 and 3 channels(RGB). Now, our Raw data is of different sizes and may consist of different number of channels therefore, we need to change it input Layer size.

5.1.4 Create a YOLO v2 Object Detection Network

A YOLO v2 object detection network is composed of two subnetworks. A feature extraction network followed by a detection network. In this we are using ResNet-50(Pretrained CNN) for feature extraction. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific for YOLO v2. Using the yolov2Layers (CV tool box) function to create a YOLO v2 object detection network automatically given a pretrained ResNet-50 feature extraction network. yolov2Layers requires you to specify several inputs that parameterize a YOLO v2 network

5.1.5 Network Model

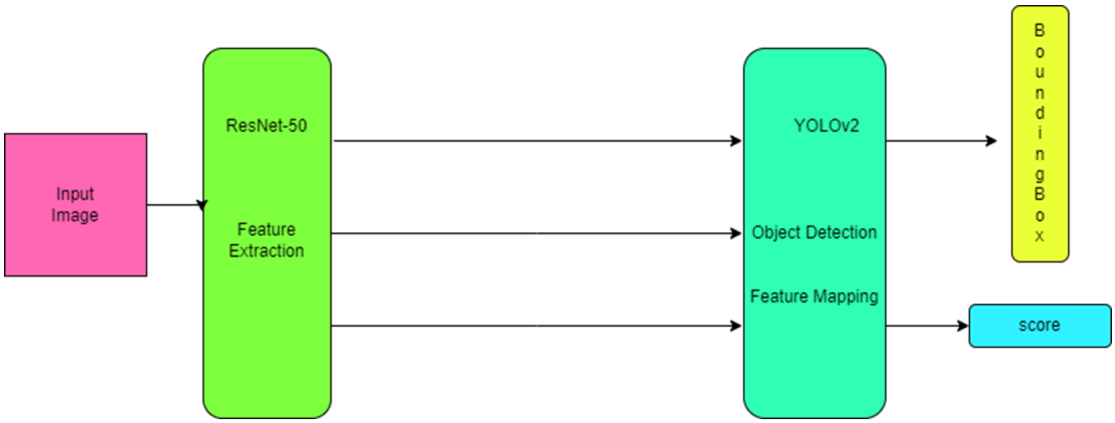

Figure 5.3: Network Model

5.1.6 ResNet-50

ResNet-50 consists of multiple layers organized in a deep convolutional neural network architecture. In true sense it consist of total 177 layers in which we will try to concatenate object detection subNet which is YOLOv2.

Input Layer: This is the first layer of the network, which receives the input image data. In ResNet-50, the input size is typically 224x224 pixels with three color channels (RGB).

Convolutional Layers: ResNet-50 contains several convolutional layers, which perform feature extraction by applying convolutional filters to the input image. These layers are responsible for detecting edges, textures, and other low-level features.

Pooling Layers: ResNet-50 includes pooling layers, typically max pooling, which downsample the feature maps to reduce spatial dimensions while preserving important features.

Fully Connected Layers: At the end of the network, ResNet-50 typically includes one or more fully connected layers followed by a softmax activation func-

tion. These layers perform classification by mapping the extracted features to the output classes.

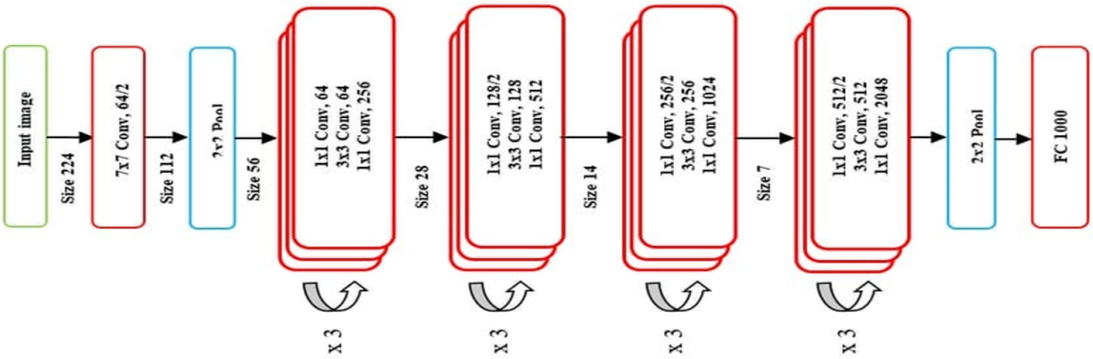

Figure 5.4: ResNet 50 Network

Now we select "activation 40 relu", 141^{th} Layer as the feature extraction layer. Specifically, we will replace the layers after "activation 40 relu" with the detection subnetwork. Creating YOLO V2 sub-Network this only of 9 layers consisting Convolution2DLayer, BatchNormalizationLayer, ReLULayer, YOLv2TransformLayer and YOLOv2OutputLayer. Making overall Network of 150 Layers.

Figure 5.5: YOLOv2 SubNet

5.1.7 Training the Network

Now our initial Network is ready that is without any weights. We need to train using our training dataset. In this we tried use solver such as ADAM, SGDM, RMSprop and tried to analyze the limitations of those solvers. Also, we have analyzed effect of other parameters such as Mini-Batch size, max no of Epochs, Learning rate on the training.

Chapter 6

Results and Analysis

6.1 Changing Solver

6.1.1 SGDM

In this we have used SGDM by keeping other parameters as mentioned. Mini-Batch Size = 16, LearnRate = $1e^{-3}$, Max-Epoch = 20.

Figure 6.1: Bounding Boxes and Scores for SGDM

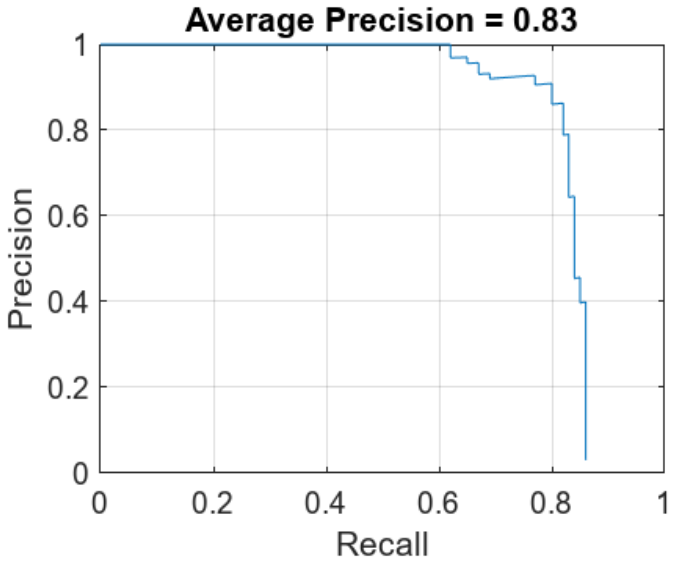

Figure 6.2: Average Precision for SGDM

We have achieved average precision of 0.83 as shown in fig 6.2 and Training time per Epoch 11 seconds.

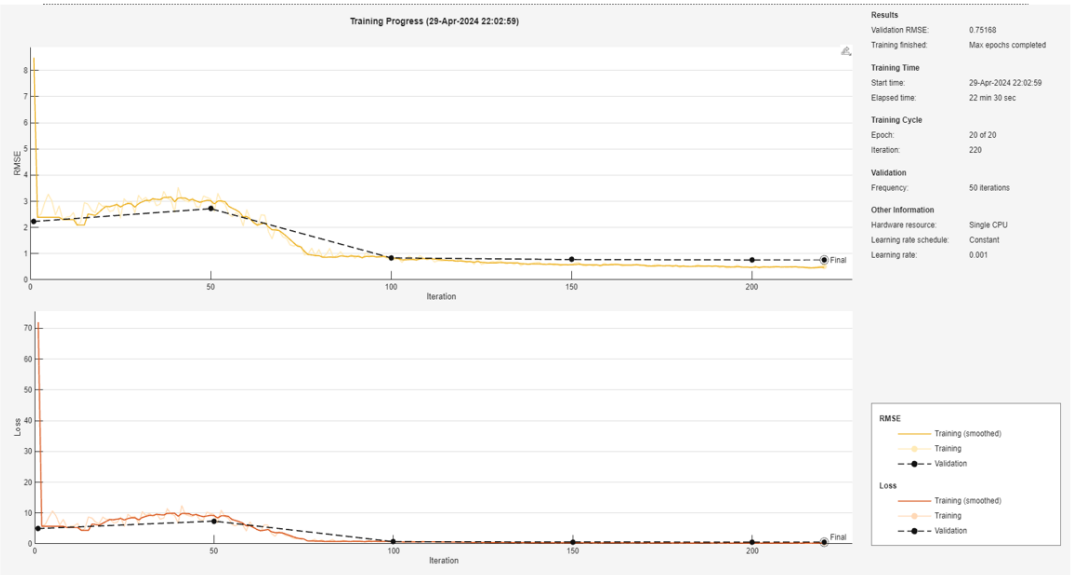

Figure 6.3: RMSE and Loss for SGDM

6.1.2 RMSprop

In this we have used RMSprop by keeping other parameters as mentioned. Mini-Batch Size = 16, LearnRate = $1e^{-3}$, Max-Epoch = 20.

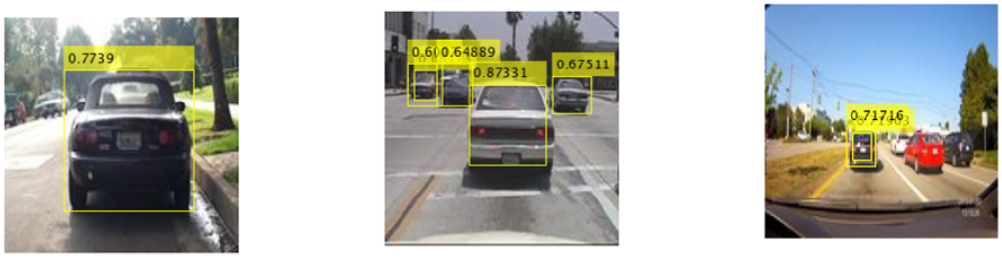

Figure 6.4: Bounding Boxes and Scores for RMSprop

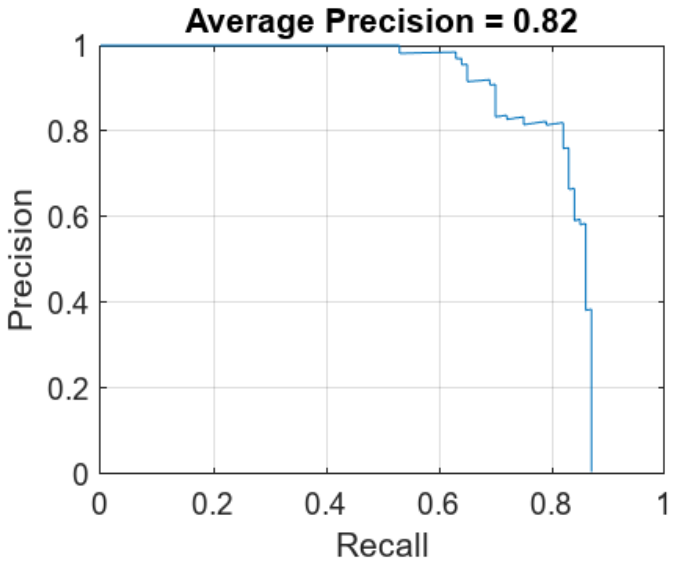

Figure 6.5: Average Precision for RMSprop

We have achieved average precision of 0.82 as shown in fig 6.5 and Training time per Epoch 11 seconds.

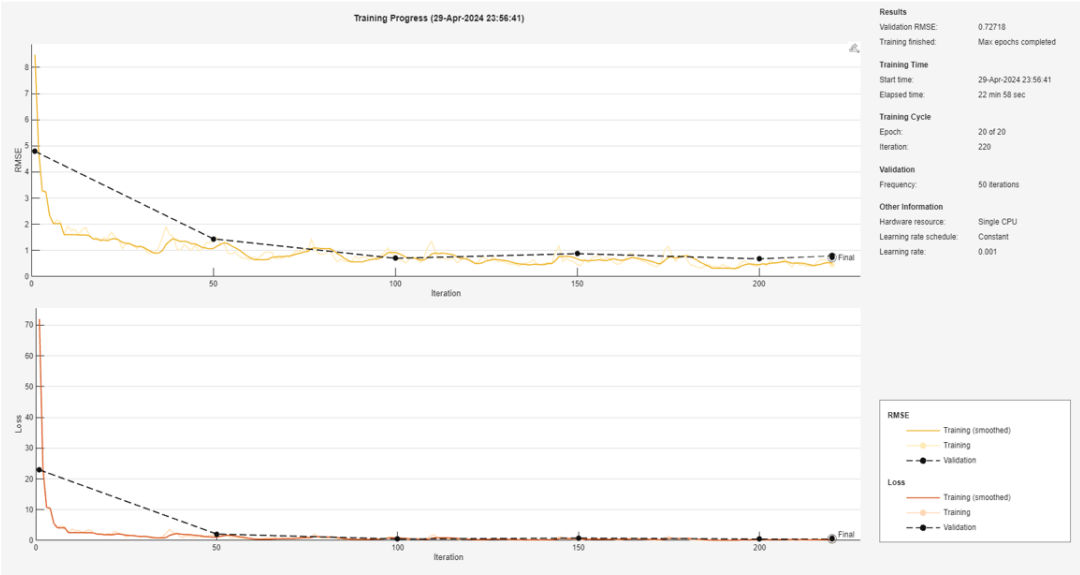

Figure 6.6: RMSE and Loss for RMSprop

6.1.3 ADAM

In this we have used ADAM by keeping other parameters as mentioned. Mini-Batch Size = 16, LearnRate = $1e^{-3}$, Max-Epoch = 20.

Figure 6.7: Bounding Boxes and Scores for ADAM

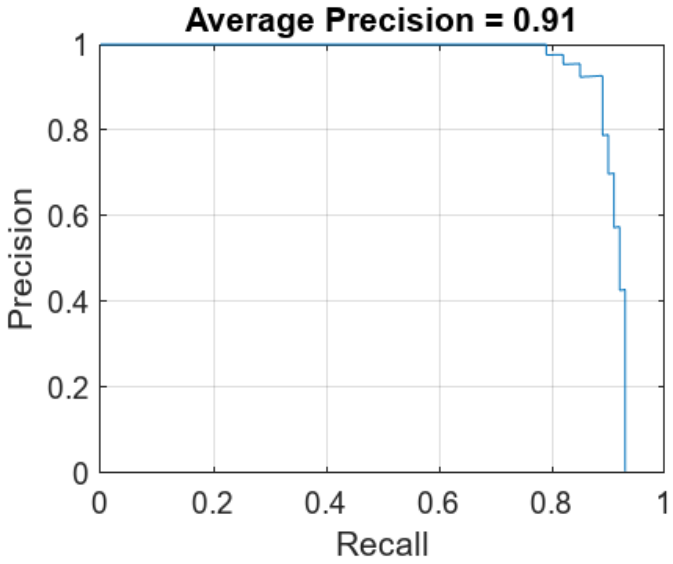

Figure 6.8: Average Precision for ADAM

We have achieved average precision of 0.91 as shown in fig 6.11 and Training time per Epoch 11 seconds.

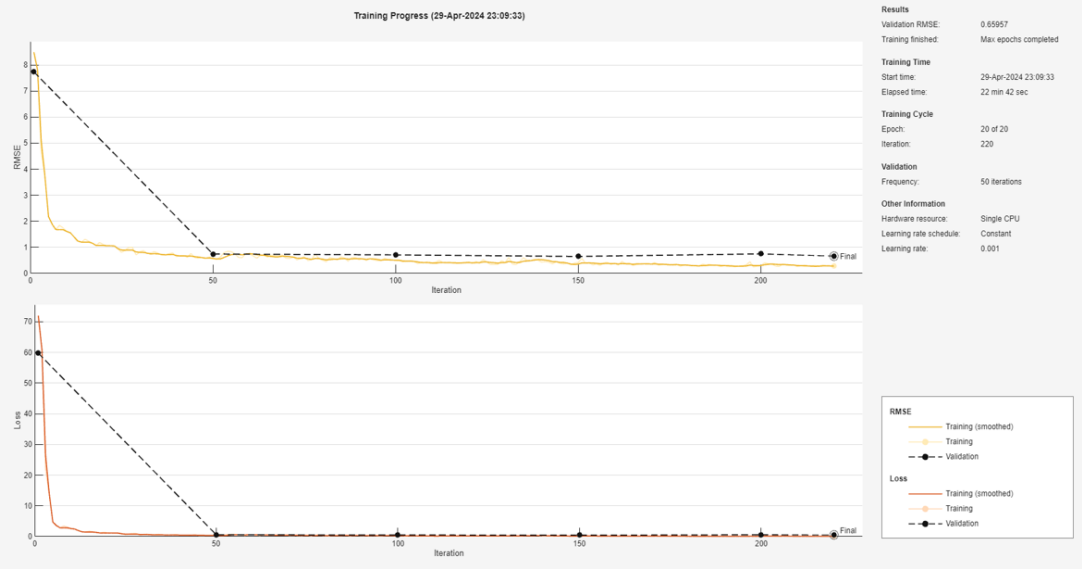

Figure 6.9: RMSE and Loss for ADAM

6.1.4 Optimization Algorithm Analysis

Optimizer	SGDM	RMSProp	ADAM
Training Time per Epoch	11 sec	11 sec	11 sec
Average Precision	0.83	0.82	0.91

Table 6.1: Comparison

As shown in table 6.1 ADAM, SGDM, and RMSprop were compared in terms of average precision and training time per epoch for YOLOv2 object detection. ADAM achieved the highest precision at 0.91, compared to SGDM's 0.83 and RMSprop's 0.82, indicating that ADAM is more accurate for object detection tasks. However, all three algorithms had the same training time per epoch, 11 seconds, suggesting that while ADAM offers better accuracy, it does not incur additional training time, making it the optimal choice among the three.

6.2 Changing the Mini-Batch Size

6.2.1 Mini-Batch size = 10

In this we have used ADAM by keeping other parameters as mentioned. LearnRate = $1e^{-3}$, Max-Epoch = 20.

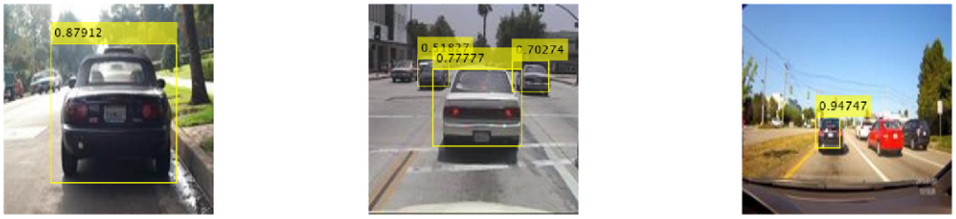

Figure 6.10: Bounding Boxes and Score for Mini-Batch size = 10

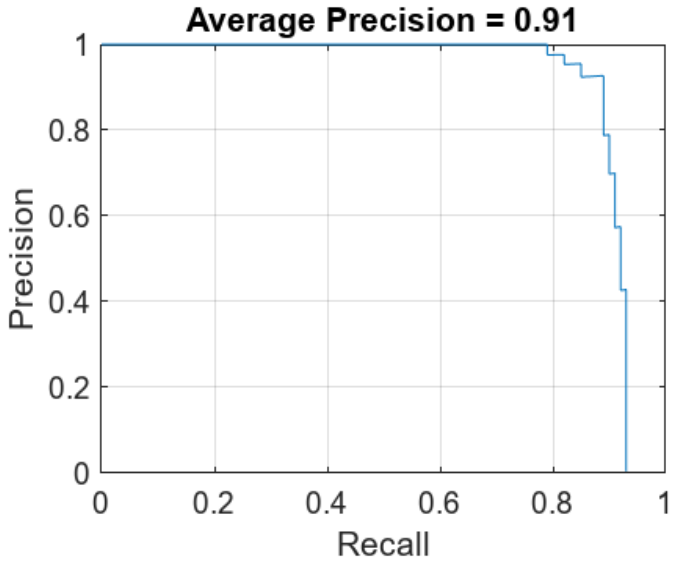

Figure 6.11: Average Precision for Mini-Batch size = 10

We have achieved average precision of 0.91 as shown in fig 6.11 and Training time per Epoch 10 seconds.

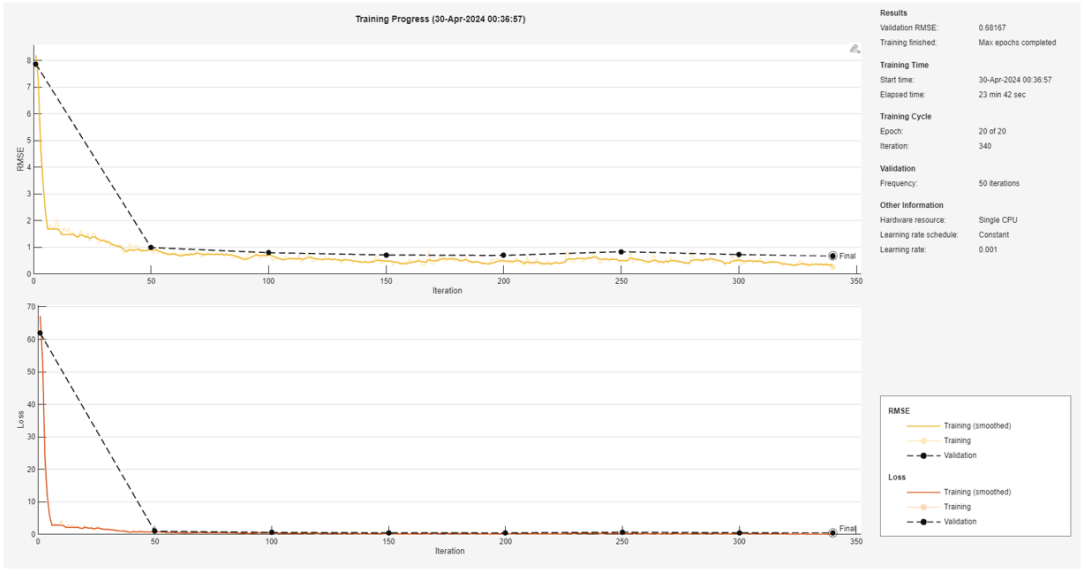

Figure 6.12: RMSE and Loss for Mini-Batch size = 10

6.2.2 Mini-Batch size = 20

In this we have used ADAM by keeping other parameters as mentioned. LearnRate = $1e^{-3}$, Max-Epoch = 20.

Figure 6.13: Bounding Boxes and Score for Mini-Batch size = 20

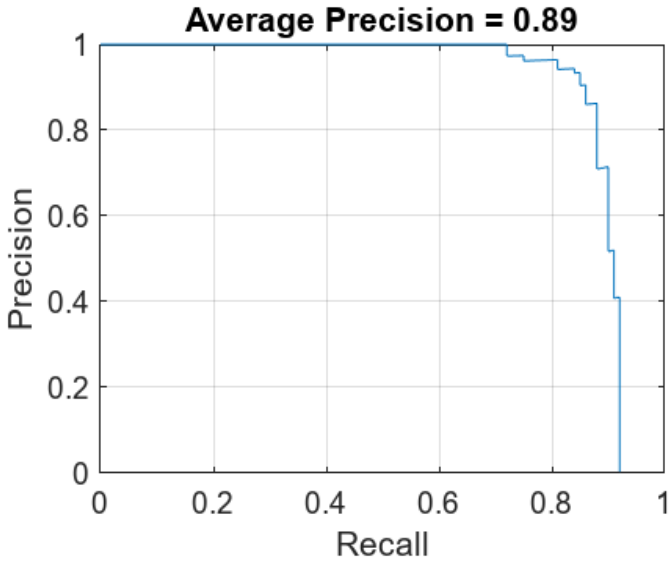

Figure 6.14: Average Precision for Mini-Batch size = 20

We have achieved average precision of 0.89 as shown in fig 6.26 and Training time per Epoch 14 seconds.

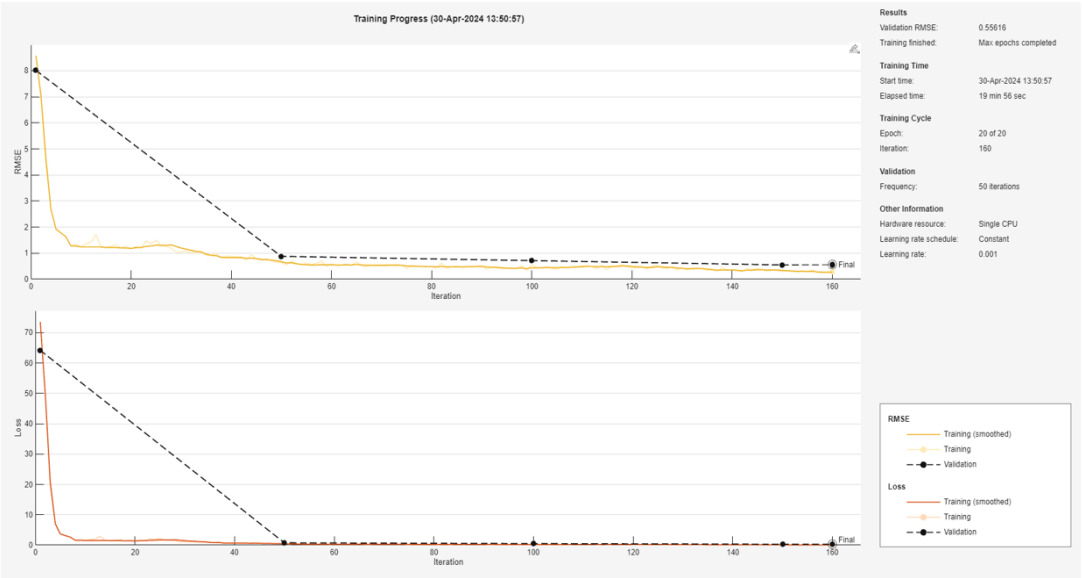

Figure 6.15: RMSE and Loss for Mini-Batch size = 20

6.2.3 Effect of Mini-Batch Size Analysis

Mini-Batch Size	10	16	20
Training Time per Epoch	10 sec	11 sec	14 sec
Average Precision	0.99	0.91	0.89

Table 6.2: Comparison

An analysis of ADAM's performance for YOLOv2 object detection with varying mini-batch sizes reveals that a smaller mini-batch size can lead to faster training without compromising accuracy as shown in table 6.2. A mini-batch size of 10 provides the highest average precision at 0.91 with the fastest training time per epoch (10 seconds). Increasing the mini-batch size to 16 maintains the same precision but increases training time to 11 seconds. A mini-batch size of 20 results in a slight drop in average precision to 0.89 and a longer training time of 14 seconds, suggesting that smaller mini-batches offer a better balance of accuracy and speed.

6.3 Changing Max Number of Epochs

6.3.1 Max Epoch = 10

In this we have used ADAM by keeping other parameters as mentioned. Mini-Batch size = 20, LearnRate = $1e^{-3}$.

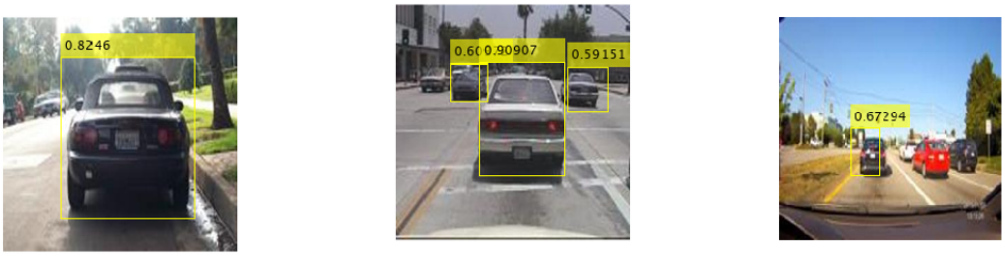

Figure 6.16: Bounding Boxes and Score for Max Epoch = 10

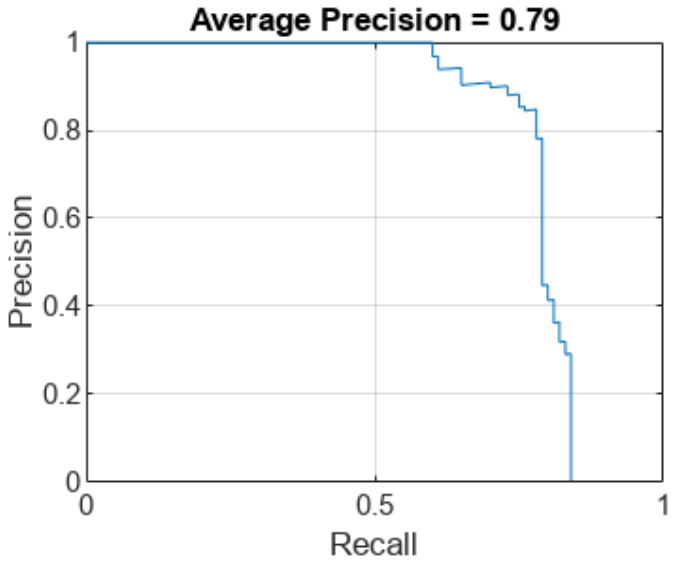

Figure 6.17: Average Precision for Max Epoch = 10

We have achieved average precision of 0.79 as shown in fig 6.17 and Training time per Epoch 13 seconds.

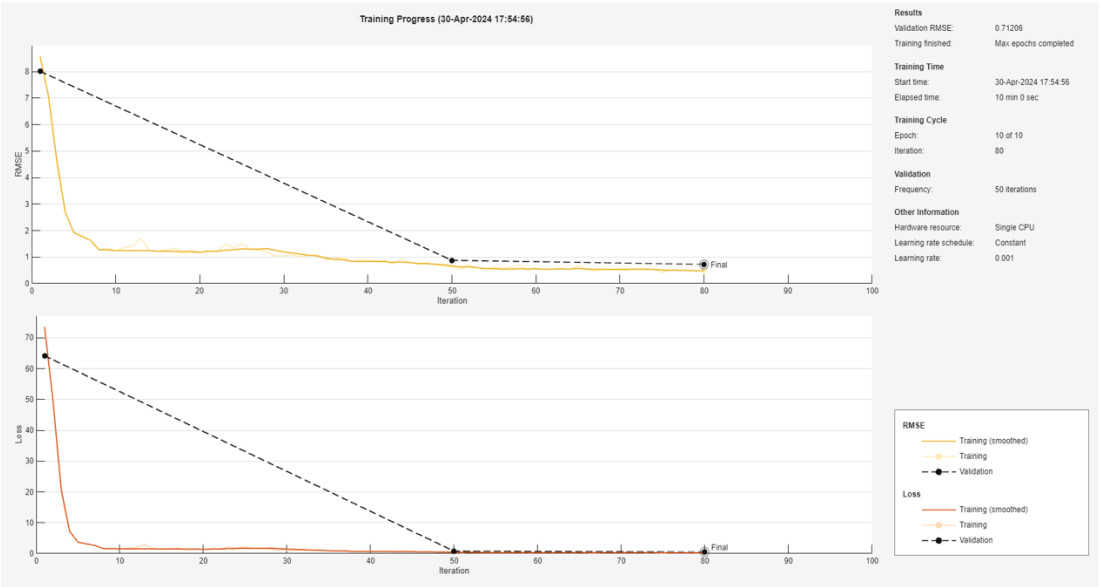

Figure 6.18: RMSE and Loss for Max Epoch = 10

6.3.2 Max Epoch = 30

In this we have used ADAM by keeping other parameters as mentioned. Mini-Batch size = 20, LearnRate = $1e^{-3}$.

Figure 6.19: Bounding Boxes and Score for Max Epoch = 30

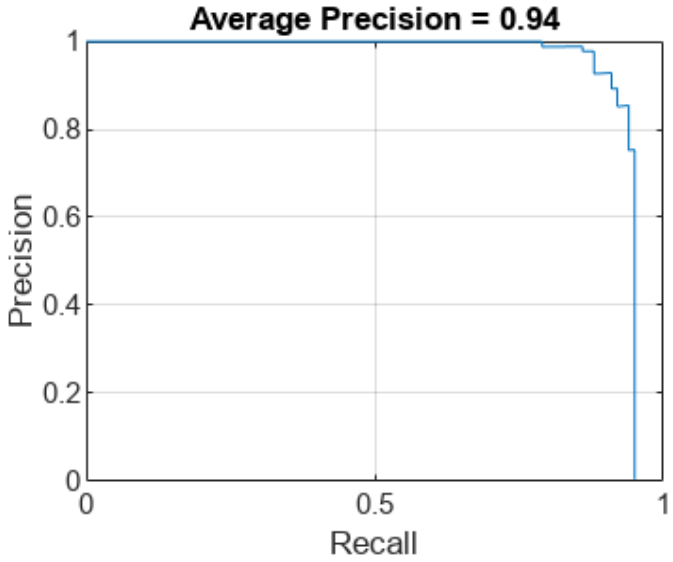

Figure 6.20: Average Precision for Max Epoch = 30

We have achieved average precision of 0.94 as shown in fig 6.20 and Training time per Epoch 14 seconds.

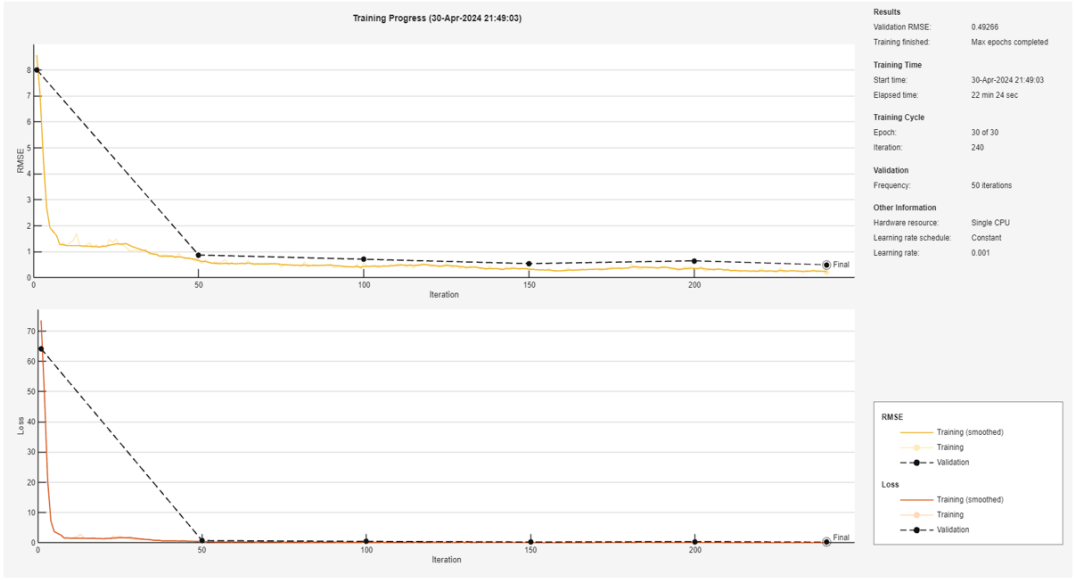

Figure 6.21: RMSE and Loss for Max Epoch = 30

6.3.3 Effect of Max no of Epoch Analysis

Max-Epoch	**10**	**20**	**30**
Training Time per Epoch	13 sec	14 sec	14 sec
Average Precision	0.79	0.89	0.94

Table 6.3: Comparison

When using ADAM as the optimization algorithm for YOLOv2, the number of epochs has a significant impact on both average precision and training time per epoch as shown in table 6.3. With 10 epochs, the average precision is 0.79 with a training time of 13 seconds per epoch, suggesting undertraining. Increasing the epochs to 30 boosts precision to 0.94, indicating higher accuracy, with a slightly longer training time of 14 seconds. At 20 epochs, precision stabilizes at 0.89, also with 14 seconds of training time, indicating that a higher epoch count generally yields better accuracy, albeit with slightly increased training time.

6.4 Changing the Learning Rate

6.4.1 LearningRate = $1e^{-2}$

In this we have used ADAM by keeping other parameters as mentioned. Mini-Batch size = 20, max-Epoch=20.

Figure 6.22: Bounding Boxes and Score for LearningRate = $1e^{-2}$

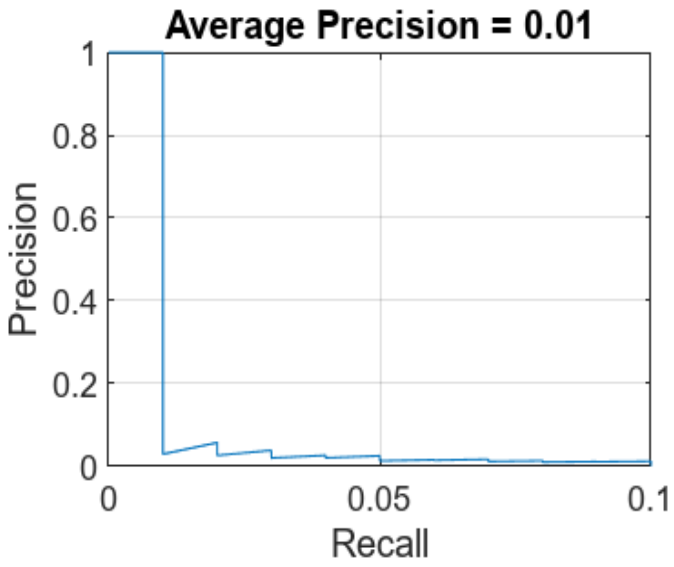

Figure 6.23: Average Precision for LearningRate = $1e^{-2}$

We have achieved average precision of 0.01 as shown in fig 6.23 and Training time per Epoch 13 seconds.

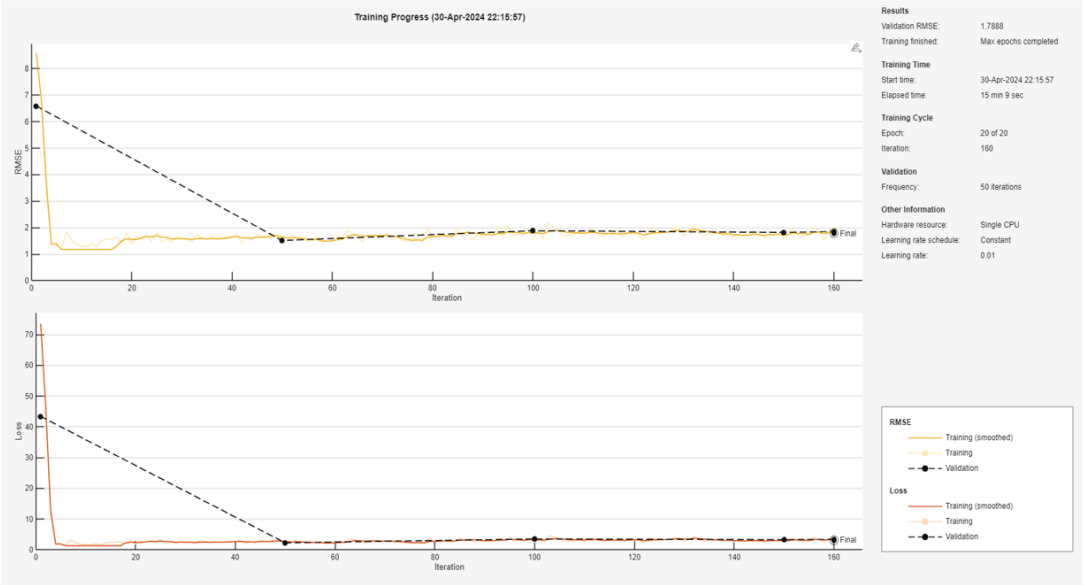

Figure 6.24: RMSE and Loss for LearningRate = $1e^{-2}$

6.4.2 LearningRate = $1e^{-4}$

In this we have used ADAM by keeping other parameters as mentioned. Mini-Batch size = 20, max-Epoch=20.

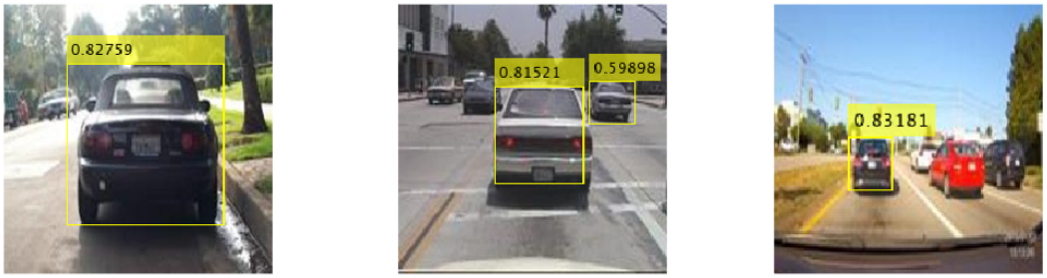

Figure 6.25: Bounding Boxes and Score for LearningRate = $1e^{-4}$

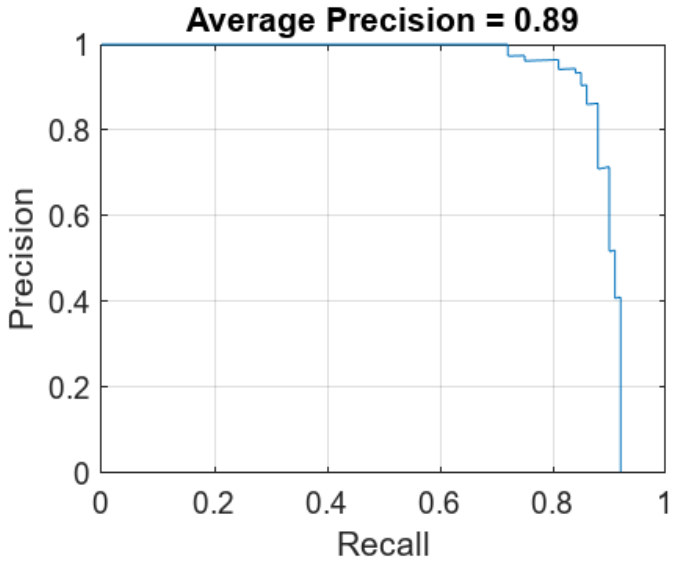

Figure 6.26: Average Precision for LearningRate = $1e^{-4}$

We have achieved average precision of 0.89 as shown in fig 6.26 and Training time per Epoch 14 seconds.

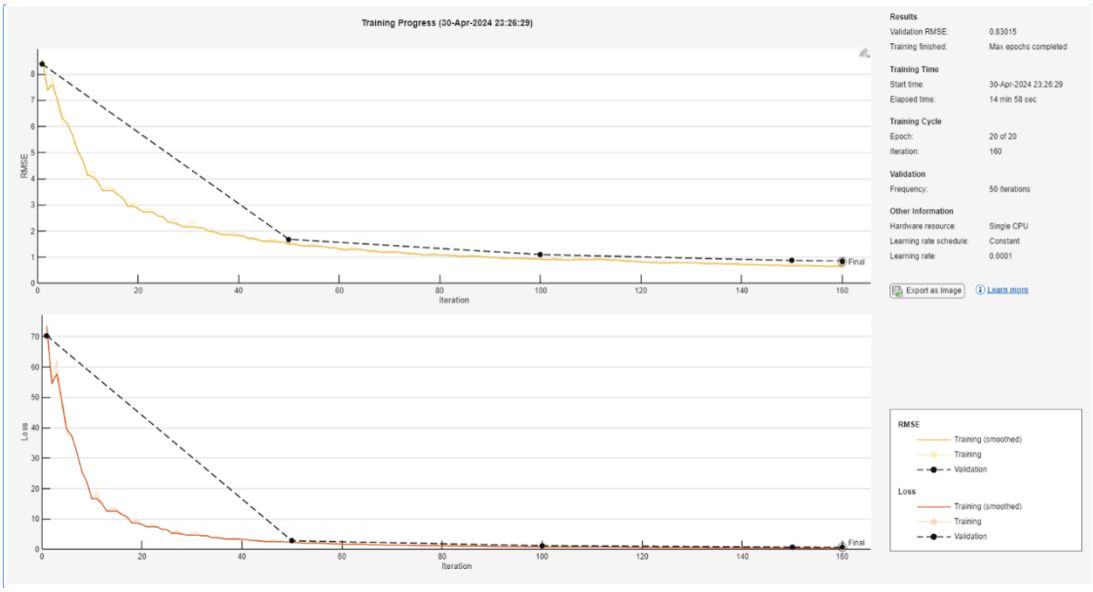

Figure 6.27: RMSE and Loss for LearningRate = $1e^{-4}$

6.4.3 Effect of LearningRate Analysis

LearningRate	$1e^{-2}$	$1e^{-3}$	$1e^{-4}$
Training Time per Epoch	13 sec	11 sec	14 sec
Average Precision	0.01	0.91	0.89

Table 6.4: Comparison

As shown in table 6.4 the impact of learning rate on ADAM's performance for YOLOv2 object detection is significant. With a learning rate of 10^{-3}, ADAM achieved the best results, with an average precision of 0.91 and a training time of 11 seconds per epoch. In contrast, a higher learning rate of 10^{-2} yielded poor precision (0.01), while a lower learning rate of 10^{-4} slightly reduced accuracy to 0.89 and increased training time to 14 seconds, indicating that optimal performance is achieved with a moderate learning rate.

Chapter 7

Hardware Implementation of YOLOv5

7.1 Introduction

As Analysis of YOLv2 has been done in previous chapter, in this chapter we have tried to implement YOLOv5 algorithm on to FPGA board specifically ZCU104 evalutionary board.

7.2 YOLOv5 ALGORITHM

YOLOv5, the fifth iteration of the YOLO (You Only Look Once) object detection framework, stands out for its flexibility, scalability, and user-friendliness. Developed by Ultralytics, it's one of the most popular versions, appreciated for its straightforward implementation and support across multiple hardware platforms, including CPUs, GPUs, and specialized hardware such as Tensor Processing Units (TPUs).

7.2.1 YOLOv5 Architecture

Convolutional Backbone

YOLOv5 employs a modular backbone inspired by YOLOv4 but optimized for efficiency. It consists of convolutional layers with residual connections, allowing for deeper networks while minimizing the risk of vanishing gradients. The backbone is designed to extract features from the input image to be used in object detection.

Neck with Path Aggregation

YOLOv5 uses a neck architecture similar to YOLOv4's Path Aggregation Network (PAN), which helps aggregate feature maps from different stages of the backbone. This design improves the flow of information and enhances detection of objects at various scales.

Grid-based Detection with Anchor Boxes

Like other YOLO versions, YOLOv5 divides the input image into a grid, with each cell predicting bounding boxes, objectness scores, and class probabilities. Anchor boxes are used to predict bounding box offsets, allowing for detection of objects with different shapes and sizes.

Bounding Box Prediction

Each grid cell predicts a set of bounding boxes with coordinates (x, y, width, height), objectness scores, and class probabilities. YOLOv5 has flexible configurations, allowing users to adjust the number of anchor boxes and the grid size based on their application requirements.

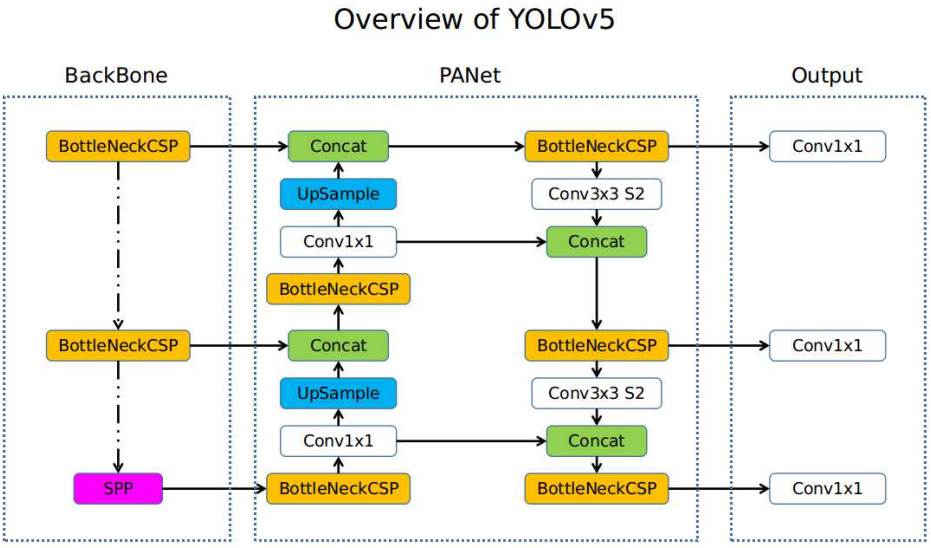

Figure 7.1: Overview of YOLOv5

Output and Post-Processing

YOLOv5 outputs a set of bounding boxes, objectness scores, and class probabilities. Non-maximum suppression (NMS) is applied to remove redundant bounding boxes and refine detection results. The final output is a list of detected objects with bounding boxes and class probabilities.

7.3 YOLOv5 HARDWARE SETUP

BOOTING AND SETUP PROCESS

1. Initially we have to Boot the FPGA board with PYNQ image.
2. PYNQ is a framework and toolset that extends Linux-based operating systems, combining their functionalities with FPGA programming through Python integration.
3. Download latest PYNQ image file from PYNQ site for ZCU104 board.
4. Flash the PYNQ image to an SD card using Etcher (Balena Etcher).
5. Set the Boot Dip Switches (SW6) to the following positions:(This sets the board to boot from the Micro-SD card).

   Dip switch 1 (Mode 0): On (down)

   Dip switch 2 (Mode 1): Off (up)

   Dip switch 3 (Mode 2): Off (up)

   Dip switch 4 (Mode 3): Off (up)
6. Connect the 12V power cable. Note that the connector is keyed and can only be connected in one way.
7. Insert the Micro SD card loaded with the appropriate PYNQ image into the MicroSD card slot underneath the board.
8. (Optional) Connect the USB cable to your PC/Laptop, and to the USB JTAG UART MicroUSB port on the board.

9. Connect the Ethernet port on your board to a router/switch. Slide the power switch to the ON position to turn on the board.

10. A Red LED and some additional yellow board LEDs will come on to confirm that the board has power.

11. After a few seconds, the red LED will change to Yellow. This indicates that the bitstream has been downloaded and the system is booting.

IP ADDRESS OF THE BOARD

1. Connect Board and your PC through USB and find the COM port number to which it is connected.

2. Open the Device Manager, expand the Ports menu.

3. Find the COM port for the USB Serial Port. e.g. COM5.

4. Open PuTTY, Once PuTTY is open, enter the following settings.

5. Select serial, Enter the COM port number.

6. Enter the serial terminal settings (below), Click open.

7. Full Terminal Settings:

   • 115200 baud

   • 8 data bit

   • 1 stop bit

   • No parity

   • No Flow control

8. Hit Enter in the terminal window to make sure you can see the command prompt.

9. You can check the IP address of the board using ifconfig command.

Figure 7.2: FPGA Hardware Setup

7.4 RESULTS

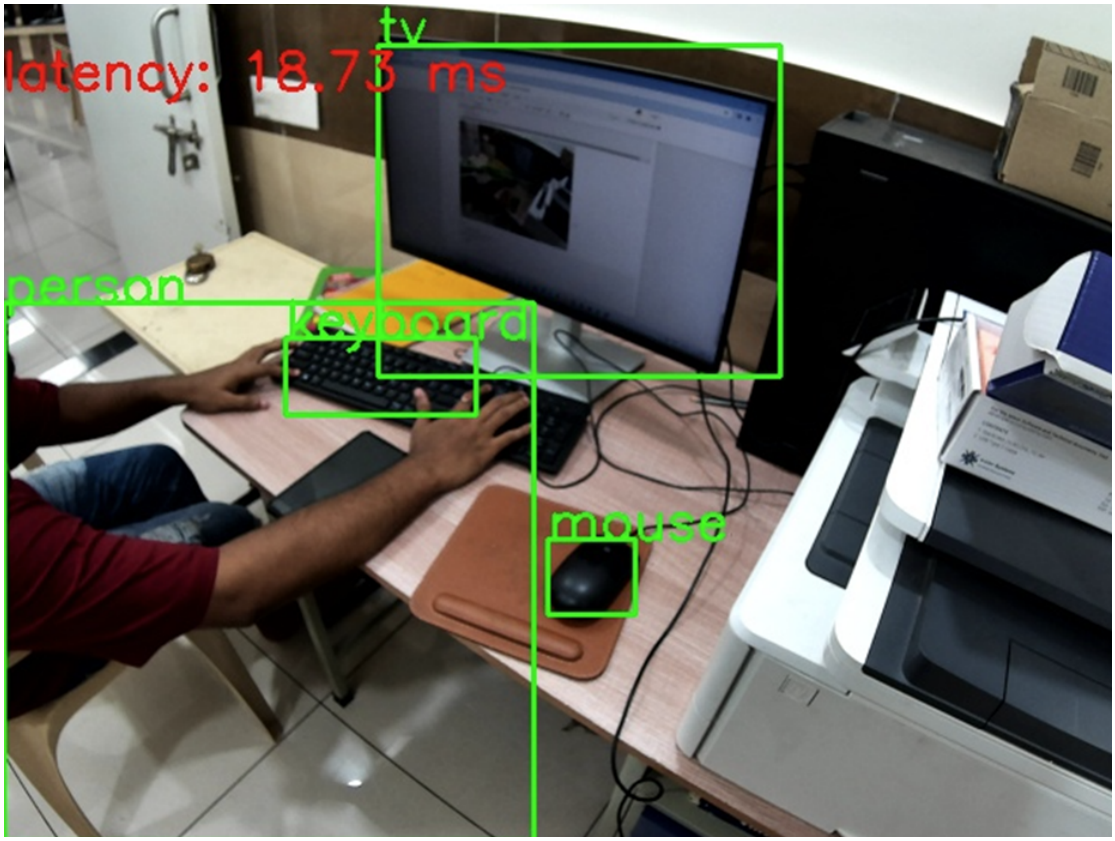

Figure 7.3: Demo of YOLOv5

Chapter 8

Conclusion and Future Scope

8.1 Conclusion

Overall, the choice of optimization algorithm, number of epochs, mini-batch size, and initial learning rate plays a critical role in the performance of our model. Experimentation and careful tuning are necessary to identify the optimal hyperparameters for achieving the best results on a given dataset and task.

8.2 Future Scope

YOLOv2 maintains relevance in real-time object detection tasks, particularly in resource-constrained environments like edge devices and embedded systems. Its efficiency makes it suitable for applications such as surveillance, autonomous driving, and robotics. Additionally, it can be fine-tuned through transfer learning for specific domains, combined with other techniques for hybrid approaches, and customized for specialized use cases. While newer versions exist, YOLOv2 continues to offer a balance of speed and accuracy for various applications and research areas.

Appendices

Appendix A

Code Attachments

```
doTraining = false;
if ~doTraining && ~exist("yolov2ResNet50VehicleExample_19b.mat","file")
    disp("Downloading pretrained detector (98 MB)...");
    pretrainedURL = "https://www.mathworks.com/supportfiles/vision/data/yolov2ResNet50VehicleExample_19b.mat";
    websave("yolov2ResNet50VehicleExample_19b.mat",pretrainedURL);
end

unzip vehicleDatasetImages.zip
data = load("vehicleDatasetGroundTruth.mat");
vehicleDataset = data.vehicleDataset;

# Display first few rows of the data set.
vehicleDataset(1:4,:)

# Add the fullpath to the local vehicle data folder
vehicleDataset.imageFilename = fullfile(pwd,vehicleDataset.imageFilename);

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

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

validationIdx = idx+1 : idx + 1 + floor(0.1 * length(shuffledIndices) );
validationDataTbl = vehicleDataset(shuffledIndices(validationIdx),:);

testIdx = validationIdx(end)+1 : length(shuffledIndices);
testDataTbl = vehicleDataset(shuffledIndices(testIdx),:);

imdsTrain = imageDatastore(trainingDataTbl(:,"imageFilename"));
bldsTrain = boxLabelDatastore(trainingDataTbl(:,"vehicle"));

imdsValidation = imageDatastore(validationDataTbl(:,"imageFilename"));
bldsValidation = boxLabelDatastore(validationDataTbl(:,"vehicle"));

imdsTest = imageDatastore(testDataTbl(:,"imageFilename"));
bldsTest = boxLabelDatastore(testDataTbl(:,"vehicle"));



trainingData = combine(imdsTrain,bldsTrain);
validationData = combine(imdsValidation,bldsValidation);
testData = combine(imdsTest,bldsTest);

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

inputSize = [224 224 3];

#Define the number of object classes to detect.
numClasses = width(vehicleDataset)-1;

trainingDataForEstimation = transform(trainingData,@(data)preprocessData(data,inputSize));
numAnchors = 7;
[anchorBoxes, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation, numAnchors)
featureExtractionNetwork = resnet50;
```

```
featureLayer = "activation_40_relu";
lgraph = yolov2Layers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featureLayer);
augmentedTrainingData = transform(trainingData,@augmentData);

#Visualize the augmented images.
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},"rectangle",data{2});
    reset(augmentedTrainingData);
end
figure
preprocessedTrainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize));
preprocessedValidationData = transform(validationData,@(data)preprocessData(data,inputSize));
montage(augmentedData,"BorderSize",10)

#Read the preprocessed training data.
data = read(preprocessedTrainingData);

# Display the image and bounding boxes.
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,"rectangle",bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)



options = trainingOptions("sgdm", ...
        "MiniBatchSize",16, ....
        "InitialLearnRate",1e-3, ...
        "MaxEpochs",20, ...
        "CheckpointPath",tempdir, ...
        "ValidationData",preprocessedValidationData);
if doTraining
    % Train the YOLO v2 detector.
    [detector,info] = trainYOLOv2ObjectDetector(preprocessedTrainingData,lgraph,options);
else
    % Load pretrained detector for the example.
    pretrained = load("yolov2ResNet50VehicleExample_19b.mat");
    detector = pretrained.detector;
end

I = imread("highway.png");
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);
# Display the results.
I = insertObjectAnnotation(I,"rectangle",bboxes,scores);
figure
imshow(I)
preprocessedTestData = transform(testData,@(data)preprocessData(data,inputSize));
detectionThreshold = 0.01;
detectionResults = detect(detector, preprocessedTestData, Threshold=detectionThreshold);
metrics = evaluateObjectDetection(detectionResults,preprocessedTestData);
classID = 1;
precision = metrics.ClassMetrics.Precision{classID};
recall = metrics.ClassMetrics.Recall{classID};

figure
plot(recall,precision)
xlabel("Recall")
ylabel("Precision")
grid on
title(sprintf("Average Precision = %.2f",metrics.ClassMetrics.mAP(classID)))
```

```
function B = augmentData(A)
% Apply random horizontal flipping, and random X/Y scaling. Boxes that get
% scaled outside the bounds are clipped if the overlap is above 0.25. Also,
% jitter image color.

B = cell(size(A));

I = A{1};
sz = size(I);
if numel(sz)==3 && sz(3) == 3
    I = jitterColorHSV(I,...
        "Contrast",0.2,...
        "Hue",0,...
        "Saturation",0.1,...
        "Brightness",0.2);
end

% Randomly flip and scale image.
tform = randomAffine2d("XReflection",true,"Scale",[1 1.1]);
rout = affineOutputView(sz,tform,"BoundsStyle","CenterOutput");
B{1} = imwarp(I,tform,"OutputView",rout);

% Sanitize boxes, if needed. This helper function is attached as a
% supporting file. Open the example in MATLAB to access this function.
A{2} = helperSanitizeBoxes(A{2});



% Apply same transform to boxes.
[B{2},indices] = bboxwarp(A{2},tform,rout,"OverlapThreshold",0.25);
B{3} = A{3}(indices);

% Return original data only when all boxes are removed by warping.
if isempty(indices)
    B = A;
end
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to the targetSize.
sz = size(data{1},[1 2]);
scale = targetSize(1:2)./sz;
data{1} = imresize(data{1},targetSize(1:2));

% Sanitize boxes, if needed. This helper function is attached as a
% supporting file. Open the example in MATLAB to access this function.
data{2} = helperSanitizeBoxes(data{2});

% Resize boxes to new image size.
data{2} = bboxresize(data{2},scale);
end
```