Aerial View Object detection using a custom dataset on YOLOv7

PROJECT SEMESTER: 2022-23 Even Semester (from January 2023)

Internship Student FINAL REPORT

Date of Visit .....

Submitted by

K.Srujan

Roll No. 101906040

(Signature of Faculty Mentor)

(Signature of Industry Mentor)

Name.....

Designation.....

Name.A. NARESH.....

Designation.SCIENTIST-F.....

AKULA NARESH
Scientist-F
Defence Research and Development Laboratory
DRDO, Government of India
Ministry of Defence, Hyderabad-500 058

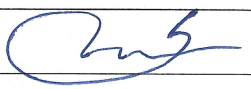
THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT
TIET, PATIALA-147004, PUNJAB
INDIA
May 2023

Declaration

I hereby declare that the working prototype, design principles and model of the project entitled Aerial View object detection using a custom dataset on YOLOv7 is an authentic record of my own work carried out in Defense Research and Development Laboratory (DRDO-DRDL) as part of my Internship Semester in Thapar Institute of Engineering and technology under the guidance of Dr. Akula Naresh during 8th Semester (2023).

Date:

Roll No:	Name	Signature
101906040	K.Srujan	

Counter Signed By:

Faculty Mentor:

Dr.

Designation

ECED

TIET,Patiala

Certificate

Certified that project entitled **"Aerial View Object detection using a custom dataset on YOLOv7"** which is being submitted by Katta Srujan **(University Registration No.** 101906040) to the **Department of Electronics and Communication Engineering, TIET, Patiala, Punjab,** is a record of project work carried out by him/her under guidance and supervision of Dr. Akula Naresh .The matter presented in this project report does not incorporate without acknowledgement any material previously published or written by any other person except where due reference is made in the text.

Katta Srujan
101906040

(Signature of Faculty Mentor)

(Signature of Industry Mentor)

Name.....

Name.....A.NARESH.....

Designation.....

Designation.....SCIENTIST-F.....

AKULA NARESH
Scientist-F
Defence Research and Development Laboratory
DRDO, Government of India
Ministry of Defence, Hyderabad-500 058

Acknowledgment

I would like to thank Dr. Akula Naresh(Scientist-F), who served as my mentor. He has been a crucial source of technical knowledge for my project and has been of great assistance. I would like to also thank the Director of DRDO-DRDL who has given me the opportunity to do an Internship at DRDO-DRDL as part of my project Semester. Additionally, I would like to express my gratitude to the entire faculty and staff of the Electronics & Communication Engineering Department. I appreciate each and every person who made a direct or indirect contribution to this project. Finally, I would like to express my gratitude to my family for their unwavering support and love. I am in awe of their tenacity and commitment because they always wanted the best for me.

Abstract

Aerial view object detection plays a crucial role in various applications such as surveillance, agriculture, urban planning, and disaster management. This project focuses on developing an efficient object detection system using YOLOv7, a state-of-the-art deep learning algorithm. To enhance the performance of the detector, a custom dataset specifically tailored for aerial imagery is created and utilized for training the YOLOv7 model. The dataset includes accurately labeled bounding boxes for different object classes commonly found in aerial images, such as cars, buildings, and vegetation. The YOLOv7 architecture, optimized for GPU acceleration, is employed to leverage the parallel processing capabilities of graphics processing units, enabling faster training and inference times. The developed system demonstrates high accuracy and robustness in detecting objects of interest from aerial images, reducing false detections, and facilitating reliable object recognition in real-world scenarios. The results highlight the effectiveness of the custom dataset and the YOLOv7 algorithm for aerial view object detection, opening up possibilities for improved applications in various domains requiring accurate and efficient analysis of aerial imagery.

List of Figures

List of Abbreviations

YOLO	You only look once
CNN	convolutional neural networks
SSD	**S**ingle Shot Detector
R-CNN	Region-Based Convolutional Neural Network
NMS	non-maximal suppression
GPU	graphics processing unit
CPU	central processing unit
mAP	Mean average Precision

Contents

Page No.

Chapter-4

Simulations and Results

4.1 Output detected

4.2 Challenges faced

4.3 Mitigations

Chapter-5

5.1 References Links

A. Naresh, SC-F)

AKULA NARESH
Scientist-F
Defence Research and Development Laboratory
DRDO, Government of India
Ministry of Defence, Hyderabad-500 058

Chapter-1

Introduction

Aerial object detection is the process of detecting and recognizing objects in aerial images captured by satellites, drones, or other aerial vehicles. This task is important in a wide range of applications, such as agriculture, environmental monitoring, urban planning, disaster response, and military reconnaissance. Aerial object detection typically involves analyzing and interpreting aerial imagery using computer vision techniques such as machine learning algorithms. These algorithms can be trained on aerial photo datasets with labeled objects, where the labels indicate the presence and location of objects of interest. A popular approach to aerial object detection uses convolutional neural networks (CNNs), which can learn complex patterns in images and identify objects with high accuracy. Another common technique is using object detection algorithms such as YOLO or Faster R-CNN, designed to detect and classify objects in images. Detecting objects in aerial photography can be difficult due to several factors, such as the scale, orientation, and changes in lighting conditions of objects in the image. However, advances in computer vision and machine learning techniques have greatly improved the accuracy and speed of detecting objects in aerial images. The detection of sky objects in general is an important task with many practical applications in various fields. This allows us to gain valuable insights from aerial imagery and make informed decisions about our environment and infrastructure.

1.1 Project Overview and Motivation

In this project I need to train a custom dataset for the detection of Cars, Persons from an aerial view with the high accuracy and without any false prediction and add a few parameters to the detection like pointing the center of the object (area covered by the object), locating the center of the object using pixel numbers (X-axis pixels, Y axis pixels) on the Screen, predicting the distance of the object from the lens of the camera, location of the object on the screen, etc using YOLOv7.The Inferencing of the trained model would be done on NVIDIA AGX XAVIER. The main motivation behind this Project is to Learn various object detection algorithms and Implement them to get accurate Real-time detection from an aerial view. The testing of the trained model would be done on various aerial view images captured from the aerial vehicle.

1.2 Novelty of Work

The Novelty of this project lies in the development of a groundbreaking solution: a novel and highly accurate real-time aerial view detection trained model. The distinctive aspect of this endeavor is the creation of a cutting-edge system capable of accurately detecting objects from aerial perspectives in real-time scenarios. An integral part of this project involves gaining a comprehensive understanding of the Nvidia AGX Xavier platform. Leveraging the remarkable capabilities of this hardware platform, we employ it for the inferencing process of our meticulously trained model. By utilizing the power and efficiency of the Nvidia AGX Xavier, we can ensure optimal performance and real-time detection accuracy. Furthermore, this project entails the implementation of additional lines of code and the incorporation of various parameters to extend the functionality and applicability of the trained model. These enhancements empower the model to effectively detect and classify objects within diverse applications, leveraging a custom dataset specifically curated for this purpose. By combining advanced technological knowledge, state-of-the-art hardware utilization, and the fine-tuning of the trained model, this project strives to establish a benchmark in real-time aerial view detection. Through the meticulous integration of these elements, we aim to deliver a highly accurate, versatile, and scalable solution that meets the demanding requirements of various detection applications, ultimately enhancing the effectiveness and efficiency of aerial view analysis.

Chapter-2

Literature Survey

2.1 Literature survey Table

Sr.No	Author Name	Paper Title
1	Zhong-Qiu Zhao	Object detection with Deep Learning
2	Ashwani Kumar	Object Detection System Based on Convolution Neural Networks Using Single Shot Multi-Box Detector
3	Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun	Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks
4	Meian Li, Haojie Zhu, Hao Chen, Lixia Xue and Tian Gao	Research on Object Detection Algorithm Based on Deep Learning
5	Amit Vishvas Divekar, Chandu Anilkumar, Ishika Naik, Ved Kulkarni & V. Pattabiraman	Comparative analysis of deep learning image detection algorithms
6	Sara Beery, Guanhang Wu, Vivek Rathod, Ronny Votel & Jonathan Huang	Context R-CNN: Long-Term Temporal Context for Per-Camera Object Detection
7	Tausif Diwan, G. Anirudh, Jitendra V, Tembhurne	Object detection using YOLO: challenges, architectural successors, datasets, and applications
8	Fangyao Liu, Ying Cai, Bo Ma, Peiyuan Jiang, Daji Ergu	A Review of Yolo Algorithm Developments

2.2 Advantages and Disadvantages of various object detection algorithms which can be used

Common object detection algorithms include YOLO (You Look Only Once), SSD (One Shot Detector), Faster R-CNN (Region-Based Convolutional Neural Network), and RetinaNet. Each of these algorithms has its strengths and weaknesses and is optimized for different uses and applications. Here are the strengths and weaknesses of common object detection algorithms.

Yolo (you only look once)

Strengths:

- Fast and efficient, suitable for real-time applications
- It can detect multiple objects in one pass
- Superior generalization capabilities with an end-to-end educational approach

Disadvantages:

- Less accurate than other methods, especially on small objects
- It can struggle to detect partially occluded or complex objects
- Localization errors may occur when objects are close together

SSD (Single Shot Detector)

Strengths:

- Fast and efficient, suitable for real-time applications
- It can detect multiple objects in one pass
- High precision for small objects

Disadvantages:

- Can have difficulty distinguishing objects at different scales
- Localization accuracy may be lower than other methods
- False positives can occur in chaotic scenes

Faster R-CNN (Region-Based Convolutional Neural Network)

Strengths:

- High accuracy, especially in small objects

- Accurate localization
- It can handle complex scenes with multiple objects

Disadvantages:

- May be slower than other methods
- It requires two steps of training and inference and can be more complex
- It has trouble distinguishing small objects in a cluttered scene

Retinal network

Strengths:

- Good precision in unbalanced data sets
- It can handle objects of different scales well
- Superior generalization capabilities with an end-to-end educational approach

Disadvantages:

- Slower than other methods
- May have difficulty detecting small objects in cluttered scenes
- False positives can occur in difficult scenes

R-CNN mask

Strengths:

- High accuracy in dividing and recognizing objects
- It can handle complex scenes with multiple objects
- Accurate localization

Disadvantages:

- Slower than other methods
- More complex architecture than some other methods
- May have difficulty detecting small objects in cluttered scenes

In general, the choice of object detection algorithm depends on the requirements and constraints of a particular application. YOLO and SSD are suitable for real-time applications, while Faster R-CNN and Mask R-CNN are suitable for applications where accuracy is important. RetinaNet is suitable for unbalanced datasets.

Chapter-3

Project Design and Description

3.1 Description (Work Done)

The first part of getting into object detection is to learn how the object detection algorithms work.

How do object detection algorithms work?

An object recognition algorithm is a type of computer vision algorithm whose purpose is to identify and locate objects in images or videos. There are various methods for object detection, but most modern object detection algorithms use deep learning techniques to identify and classify objects. In general, object detection algorithms work by dividing the input image into a grid of small sub-regions, also called anchor or suggestion boxes. Each of these sub-regions is analyzed by an algorithm to determine if an object is present, and if the object is identified, it is classified.

The most common deep learning method for object recognition is Convolutional Neural Network (CNN). CNNs are trained on large datasets of labeled images to learn to recognize and classify features in images. After training, CNNs can be used to analyze subregions of an image to determine whether they contain objects. Many object detection algorithms use additional techniques such as non-maximal suppression (NMS) to remove duplicate detections, use bounding box regression to adjust the location of detected objects, or use pyramidal networks as a feature to improve detection accuracy at different scales.

So, to learn Object detection algorithms we need to start by learning how a Convolution neural network (CNN) works.

The First Question we need to ask is **What is a Neuron and How does a Neural Network work?**

In neural networks, neurons are the basic computational units that process and transmit information. Neurons receive input signals from other neurons or external sources, perform calculations on the input signals, and then send output signals to other neurons.

The main structure of a neuron consists of three main components:

- **Dendrites**: are the input channels of neurons that receive signals from other neurons or external sources.
- **Cell body**: This is the main processing unit of the neuron where incoming signals are integrated and processed.
- **Axon**: It is the output channel of the neuron that sends the output signal to other neurons.

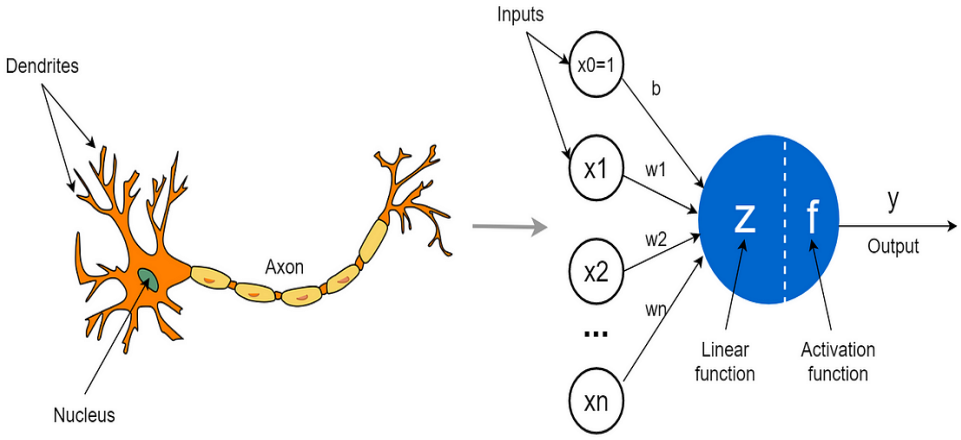


Figure 3.1 Neuron in Neural Network

Computations performed by neurons are usually modeled using mathematical functions that take as input the weighted sum of signals received by dendrites. Each dendrite weight represents the strength of the connection between the neuron and its inputs.

The mathematical functions that neurons use are typically nonlinear activation functions that introduce nonlinearity into their computations and allow neurons to learn complex patterns and relationships in their input data. As neurons perform computations, they send output signals along their axons to other neurons in the network. The strength of the output signal is usually modulated by the synaptic weight, which represents the strength of the connection between the neuron and its output target. By connecting a large number of neurons into a network, neural networks can learn to perform complex tasks such as image classification, speech recognition, and natural language processing. Neurons in the network work together to learn and extract useful features from input data and produce accurate and meaningful output predictions.

Now the Question we need to be asking is **What are the functions Used in Neural Networking?**
A neural network consists of several interconnected layers of neurons that process and transmit information. Each neuron takes the weighted sum of its input signals as input and performs a mathematical function that produces an output signal that is sent to other neurons in the network. Several key features are used in neural networks.

- **Activation function**: The activation function introduces non-linearity in the output of the neuron. Common activation functions include the sigmoid function, the hyperbolic tangent function, and the modified linear unit (ReLU) function. Activation functions help neural networks learn complex patterns and relationships in input data.
- **Loss function**: The loss function measures the difference between the predicted output of the neural network and its actual output. The goal of neural networks is to minimize the loss function during training. Common loss functions are mean squared error (MSE) and binary cross entropy.
- **Optimizer**: The optimizer is used to minimize performance loss during training. Adjust the weights of the network neurons based on the gradient of the weight reduction function. Common optimizers include stochastic gradient descent (SGD), Adam, and Adagrad.
- **Integration functions**: Integration functions are used to sample the output of neural network layers. They help to reduce the spatial dimensions of the data and extract useful features from the input data. Common integration functions include max integration and average integration.
- **Convolution functions**: Convolution functions are used in Convolutional Neural Networks (CNN) to extract features from input data. It applies a set of filters to the input data and produces a set of feature maps that indicate the presence or absence of certain features in the input data.

By combining these capabilities in different ways, neural networks can learn to perform complex tasks such as image classification, speech recognition, and natural language processing.

Now we can start with the main part,

CNN (Convolutional Neural Network) is a class of deep learning algorithms commonly used for image classification, segmentation, and object recognition. A CNN consists of several layers of neurons that learn to recognize patterns and features in images. The first layer in a CNN is usually

a convolution layer that applies a series of filters to the input image to create a series of feature maps. The next layers in the network are usually fully connected layers that learn to classify the features detected by the convolutional layers. Object detection often uses CNNs to generate object proposals or regions of interest (ROIs) in images. After ROIs are identified, another classifier is used to determine whether each ROI contains an object and, if so, classifies the object. A CNN can be trained by combining backpropagation and stochastic gradient to optimize network parameters.

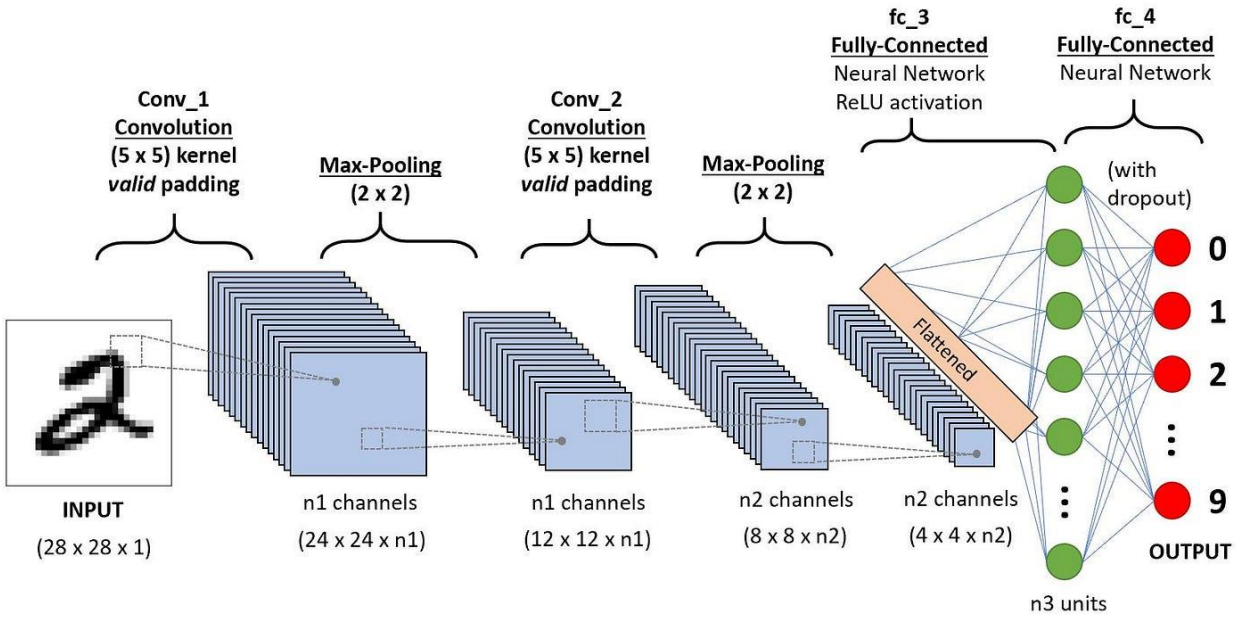


Figure 3.2 Working of CNN (Convolutional Neural Network) algorithms

How do backpropagation and stochastic gradient work in CNN?

So, Backpropagation and stochastic gradient descent are two important techniques used in training convolutional neural networks (CNN). Backpropagation is a technique used to calculate the gradient of the loss function against the weights in a neural network. Gradients are used to update network weights during the training process. Backpropagation works by propagating errors backward from the output layer of the network to the input layer. In the context of CNN, backpropagation involves calculating the gradient of the loss function according to the weights of convolutional filters and fully connected layers in the network. Gradients are used to update filter and layer weights during the training process. This helps the network to learn and extract useful features from the input data.

Stochastic gradient descent (SGD) is an optimization algorithm used to minimize the loss function during training. SGD randomly selects a small set of training examples (usually called the subset), uses these examples to compute the gradient of the loss function with respect to the weights, and updates the network weights based on this gradient. It works by doing. In the context of CNNs, SGD involves randomly selecting a small batch of images and their corresponding labels, propagating the images forward through the network to obtain predictions, and calculating the loss between the predictions and the actual labels. Then the network is used for calculation. The gradient of the loss function is according to the weight. The gradient is then used to update the weights of the filters and layers in the network.

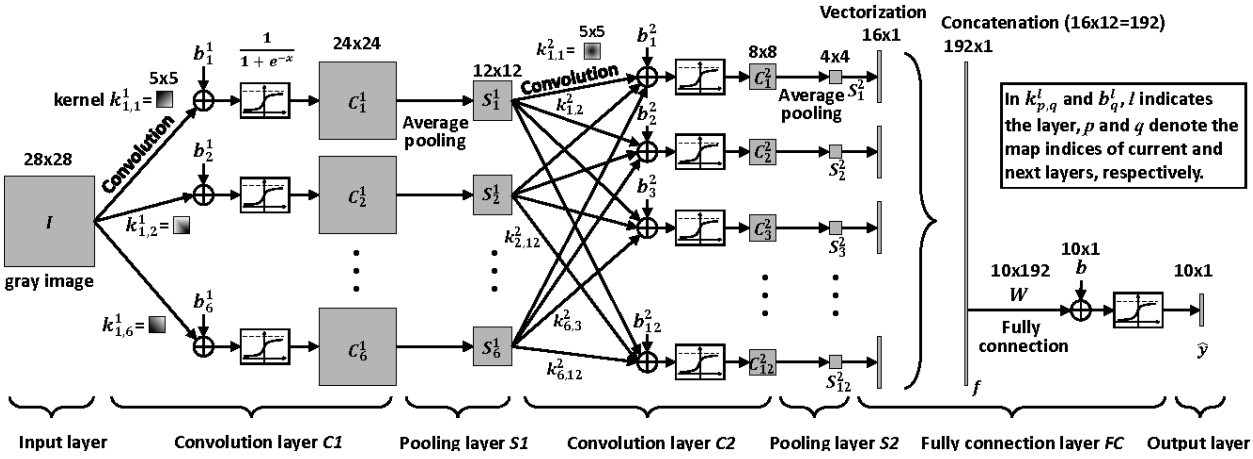


Figure 3.3 Derivation of Backpropagation

Using backpropagation combined with stochastic gradient descent, CNNs can learn to recognize and classify objects in input images with high accuracy. The gradients computed by backpropagation help to update the network weights, while SGD helps to optimize the network performance by minimizing the loss function.

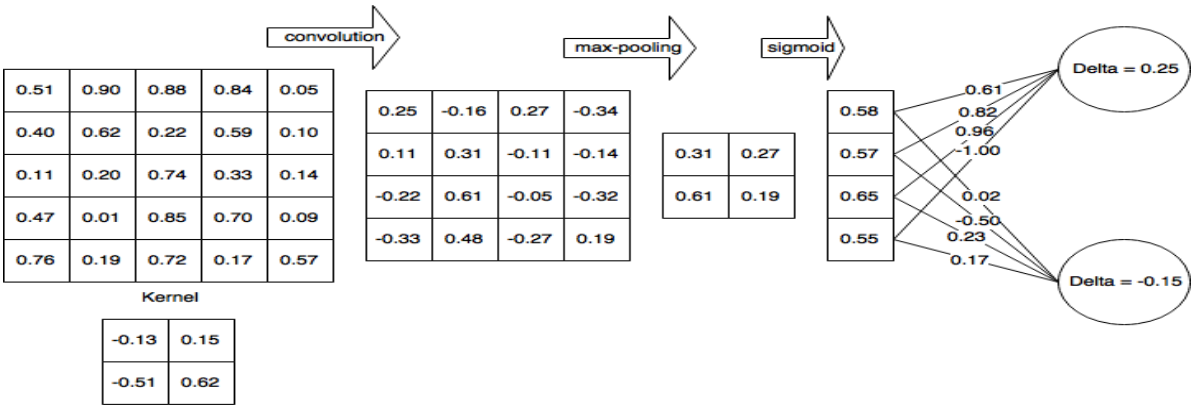


Figure 3.4 Backpropagation Example

NMS (Non-Maximum Suppression) is a post-processing step used in object detection algorithms to reduce the number of duplicate object suggestions or detections. When multiple propositions or detections overlap significantly, it can be difficult to accurately locate and classify each object. NMS overcomes this problem by selecting the proposal or detection with the highest confidence score and discarding duplicate proposals or detection with lower scores.

Figure 3.5 Before and after NMS

The basic steps of NMS are as follows:

1. Sort the proposed bounding boxes according to their objectness score, which is a measure of how likely the bounding box contains an object.

2. Select the bounding box with the highest objectness score and remove all bounding boxes with a high overlap with this bounding box.
3. Repeat step 2 until there are no more bounding boxes with high overlap.
4. Return the remaining bounding boxes as the final detections.

Now we start with YOLO as we found that YOLO is suitable for real-time applications, which is faster and more efficient at the same time when compared to other object detection algorithms.

YOLO (You Only Look Once) is a popular object detection algorithm that uses a neural network to predict bounding boxes and class probabilities directly from full images in real time. The YOLO algorithm works by dividing the input image into a grid of cells and then predicting a fixed number of bounding boxes and class probabilities for each cell. Each bounding box is represented by four parameters: its center coordinates (x, y), its width (w), and its height (h). Class probabilities show the probability of an object belonging to each of the predefined classes. The YOLO network is trained on a large dataset of labeled images using a loss function that penalizes incorrect predictions of both bounding box and class probabilities. During training, the network adjusts its parameters to minimize the loss function using backpropagation and stochastic gradient descent. During prediction, the YOLO algorithm takes an input image and passes it through a neural network to obtain the predicted bounding box and class probabilities. The algorithm then applies non-maximum suppression (NMS) to remove overlapping bounding boxes and select the most confident predictions. One of the main advantages of YOLO is its speed and efficiency. Because it performs object detection directly on the entire image in a single pass, it can achieve real-time performance on many devices, making it useful for applications such as autonomous driving, robotics, and video surveillance. Another advantage of YOLO is its ability to detect multiple objects in a single image, even if they are close to each other or partially occluded. This is because the algorithm can consider multiple bounding boxes per cell, and can adjust class probabilities based on the location and size of the bounding box.

Why did we use YOLO version 7 object detection algorithms?

The YOLO (You Only Look Once) object detection algorithm has evolved through several versions, with each new version introducing improvements and optimizations over the previous ones. YOLOv7 by Wongkinyui is one of the latest versions of the YOLO algorithm and incorporates several new features and optimizations compared to previous versions. Here's a comparison of the different versions of YOLO with YOLOv7:

1. **YOLOv1**: The first version of YOLO introduced the concept of dividing the input image into a grid and predicting bounding boxes and class probabilities for each cell in the grid. It used a single convolutional neural network (CNN) to predict these values and achieved real-time performance on a GPU. However, it had lower accuracy compared to later versions.

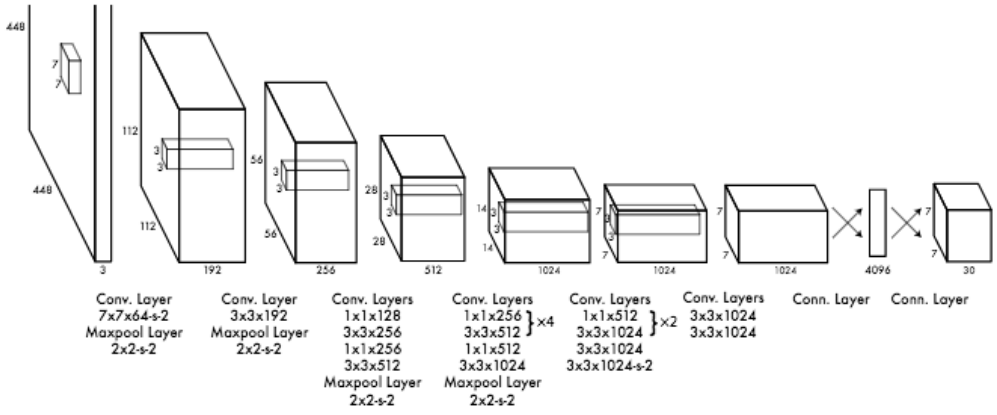


Figure 3.6 YOLO v1 architecture

2. **YOLOv2**: The second version of YOLO introduced several improvements over the original version, including the use of anchor boxes and batch normalization. It also used a more powerful CNN architecture with skip connections and achieved higher accuracy compared to YOLOv1.

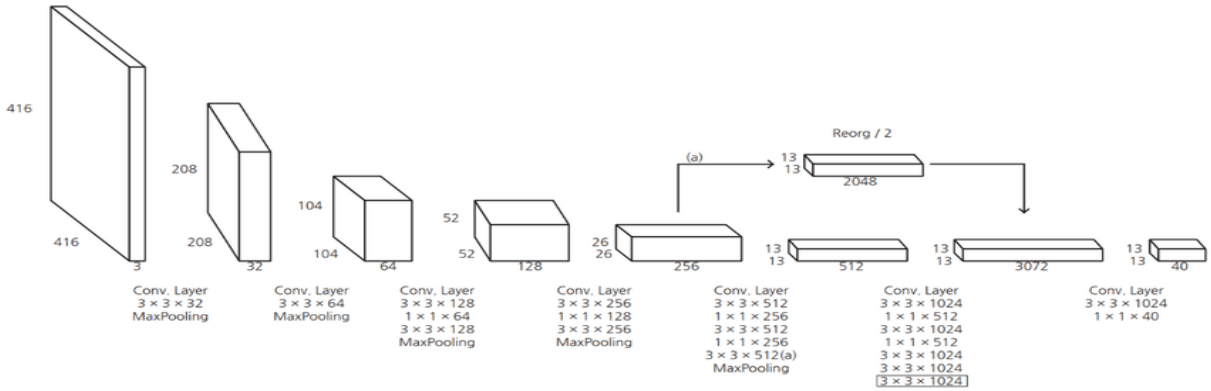


Figure 3.7 YOLO v2 architecture

3. **YOLOv3**: The third version of YOLO introduced several additional improvements over YOLOv2, including the use of a larger network architecture, feature pyramid networks, and multi-scale predictions. It also introduced the concept of dynamic anchor boxes, which adapt to the size and shape of objects in the input image. YOLOv3 achieved state-of-the-art performance on several object detection benchmarks

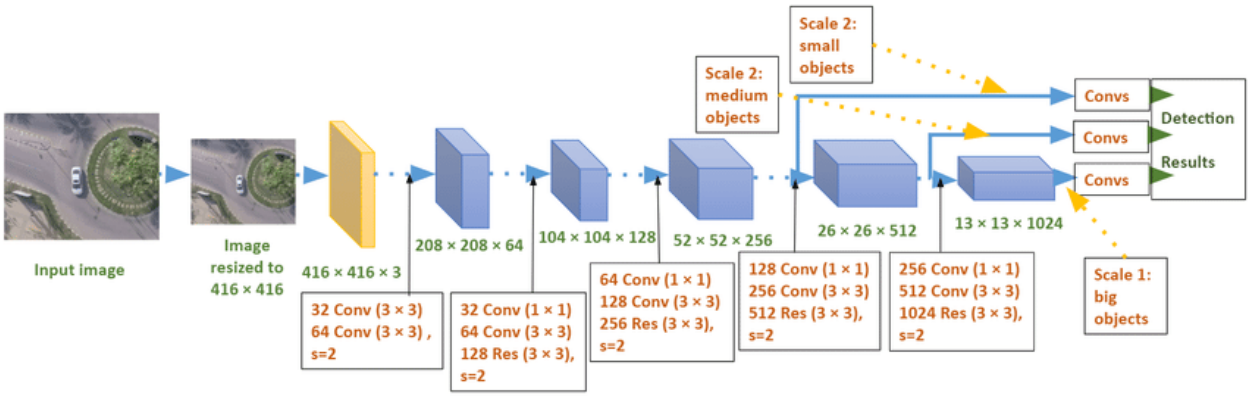


Figure 3.8 YOLO v3 architecture

4. **YOLOv4**: The fourth version of YOLO introduced several new optimizations and features, including the use of a more efficient backbone network, the use of spatial pyramid pooling, and the use of advanced data augmentation techniques. It also introduced the concept of

mosaic data augmentation, which combines multiple images into a single training sample. YOLOv4 achieved even higher accuracy than YOLOv3.

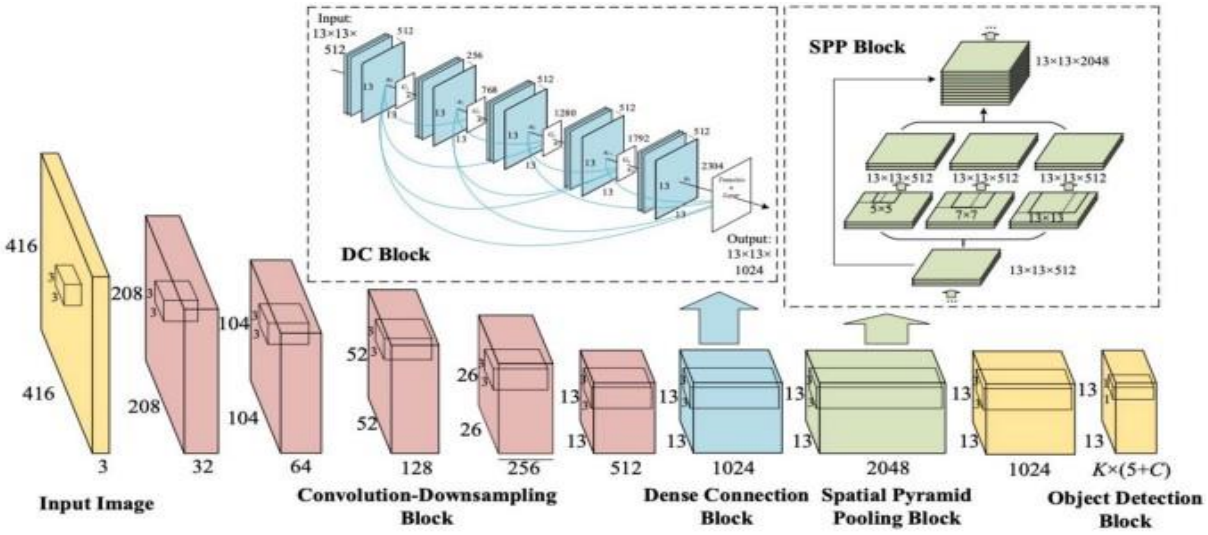


Figure 3.9 YOLO v4 architecture

5. **YOLOv5**: The fifth version of YOLO introduced several new features, including the use of more efficient network architecture, the use of anchor-free object detection, and the use of auto-augmentation to generate new training samples. It also introduced a more efficient training procedure that allows for faster convergence. YOLOv5 achieved state-of-the-art performance on several object detection benchmarks.

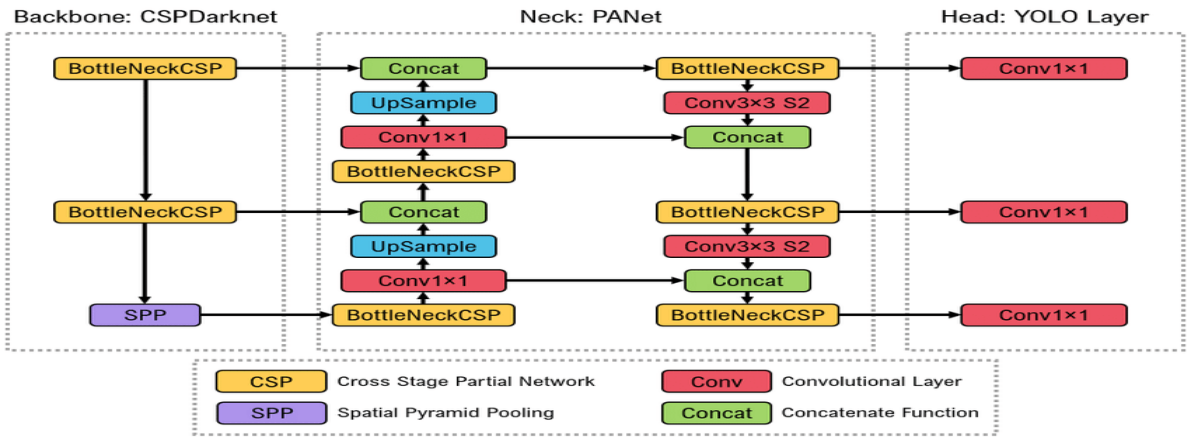


Figure 3.10 YOLO v5 architecture

6. **YOLOv7**: The latest version of YOLO, YOLOv7 by Wongkinyui, builds upon the previous versions of YOLO and introduces several new optimizations and features. It uses a modified version of the Darknet53 backbone network called CSPDarknet53, which is more efficient and accurate than the original Darknet53 network. It also uses a novel approach to the prediction that combines multiple scales and aspect ratios to improve accuracy. YOLOv7 achieves state-of-the-art performance on several object detection benchmarks and is designed to be fast and efficient while still achieving high accuracy.

Each version of YOLO introduced new optimizations and features that improved performance and accuracy. YOLOv7 by Wongkinyui builds upon the success of previous versions and introduces several new optimizations and features that make it one of the best-performing object detection algorithms available.

Models	Video 1		Video 2		Video 3	
	Inference time(ms)	FPS	Inference time(ms)	FPS	Inference time(ms)	FPS
YOLOv4		14.8		14.8		14.5
YOLOv4 Tiny		153		115		96.6
YOLOv7	53	18.75	49.5	20.2	50	20
YOLOv7 Tiny	23.87	41.89	23.08	43.31	22.78	43.88
YOLOv5 Large	28.00	35.71	28.00	35.71	8.81	35.48
Yolov5 Tiny	7.00	142.85	7.90	126.58	8.20	121.95

Figure 3.11 Comparison of Different versions of YOLO's precision

The table in the picture shows a comparison of the performance of YOLOv7, YOLOv5, and other state-of-the-art object detection algorithms on the COCO validation dataset. The table includes

several metrics commonly used to evaluate object detection algorithms, including Average Precision (AP), Frames Per Second (FPS), and model size. Average Precision (AP) is a measure of the accuracy of the model in detecting objects in the images. YOLOv7 achieves an AP of 52.1%, which is higher than the AP of YOLOv5 (50.0%) and other state-of-the-art algorithms, such as EfficientDet and DETR. Frames Per Second (FPS) is a measure of the speed of the model in processing images. YOLOv7 achieves an FPS of 95.9, which is higher than the FPS of YOLOv5 (57.5) and other algorithms, such as EfficientDet and DETR. The model size is a measure of the memory requirements of the model. YOLOv7 has a smaller model size than other state-of-the-art algorithms, such as EfficientDet and DETR, while still achieving comparable performance in terms of AP and FPS. So, we can say that the table shows that YOLOv7 achieves state-of-the-art performance in terms of both accuracy and speed, while also having a smaller model size than other algorithms. This makes YOLOv7 a highly effective and efficient object detection algorithm for a wide range of applications.

3.2 Training of a Model

Here are the general steps for detecting aerial objects using YOLOv7:

- Collect a dataset of aerial photos and their corresponding object labels.
- Images should be of high enough resolution to clearly capture the object of interest. Preprocesses an image by resizing it to a standard size, normalizing pixel values, and applying any necessary transformations (e.g., rotate, flip, crop).
- Divide the dataset into training, validation, and test sets. The training set is used to train the model, the validation set is used to monitor model performance during training, and the test set is used to evaluate model performance after training.
- Use the appropriate loss function and an optimization algorithm to train the YOLOv7 model on the training set.
- The model should be trained over multiple epochs, adjusting the learning rate as needed. Evaluate the performance of the model trained on the validation set and make necessary adjustments to the model's hyperparameters to improve its performance.
- Once the model performs satisfactorily on the validation set, its performance on the test set can be evaluated to get a final measure of its accuracy.
- Use the YOLOv7-trained model to detect objects in new aerial photos. The model can run in real-time on a single image or video stream, depending on the application.
- It should be noted that detecting sky objects can be a difficult task due to factors such as image resolution, lighting conditions, and scale of the object. However, changes to YOLOv7 can help improve its accuracy and speed of detecting objects in aerial images.

3.2.1 Code (adding various parameters to the object detection)

The following codes are written in the detect.py file of Yolov7 :

CODE:

```
# Process detections
      for i, det in enumerate(pred):  # detections per image
         if webcam:  # batch_size >= 1
            p, s, im0, frame = path[i], '%g: ' % i, im0s[i].copy(), dataset.count
         else:
            p, s, im0, frame = path, '', im0s, getattr(dataset, 'frame', 0)

         p = Path(p)  # to Path
         save_path = str(save_dir / p.name)  # img.jpg
         txt_path = str(save_dir / 'labels' / p.stem) + ('' if dataset.mode == 'image' else f'_{frame}')  #
img.txt
         gn = torch.tensor(im0.shape)[[1, 0, 1, 0]]  # normalization gain whwh
         if len(det):
            # Rescale boxes from img_size to im0 size
            det[:, :4] = scale_coords(img.shape[2:], det[:, :4], im0.shape).round()
            for *xyxy,conf, cls in reversed(det):
               c1,c2 = (int(xyxy[0]), int(xyxy[1])), (int(xyxy[2]),int(xyxy[3]))
               center_point = round((c1[0]+c2[0])/2), round((c1[1]+c2[1])/2)
               circle =cv2.circle(im0,center_point,5,(0,255,0),-1)
               text_coord =
cv2.putText(im0,str(center_point),center_point,cv2.FONT_HERSHEY_PLAIN,3,(0,0,255))
               center_width= int(im0.shape[1]/2)
               center_height= int(im0.shape[0]/2)
               center_display = (center_width, center_height)
               cv2.line(im0, (0,center_height),(im0.shape[1],center_height), (255, 0, 0),2)
               cv2.line(im0, (center_width,0),(center_width,im0.shape[0]), (255, 0, 0),2)
               if (center_point[0]) > center_width and center_point[1] < center_height:
```

```
                cv2.putText(im0,'object is on right top side',(im0.shape[1]-200,20),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,0,0),2)
            elif (center_point[0]) < center_width and center_point[1] < center_height:
                cv2.putText(im0,'object is on left top side',(im0.shape[1]-200,20),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,0,0),2)
            elif (center_point[0]) > center_width and center_point[1] > center_height:
                cv2.putText(im0,'object is on bottom right side',(im0.shape[1]-200,20),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,0,0),2)
            else:
                cv2.putText(im0,'object is on bottom left side',(im0.shape[1]-200,20),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,0,0),2)
```

Description of the Code:

This part of the code is responsible for processing the detections made by the model and adding visualizations to the image. The for i, det in enumerate(pred): loop iterates over the detections made by the model for each image. The if webcam: statement checks whether the detections are for a single image or a batch of images. The code then extracts the image path, and its name, and creates a save path and text file path for the image. It also creates a normalization gain for the image. If there are any detections in the image, the code rescales the boxes from the model's output to the original image size using the scale_coords function. It then iterates through each detection and extracts its coordinates, confidence score, and class. The code then calculates the center point of the detection and visualizes it as a small circle using the cv2.circle function. It also puts a text label indicating the center point of the detection on the image using the cv2.putText function. Next, the code calculates the center of the image and draws horizontal and vertical lines passing through it using the cv2.line function. It then uses conditional statements to determine which side of the center point the detection is on and puts a text label indicating the side on the image using the cv2.putText function. Finally, the code saves the modified image to a file named save_path and saves the bounding box information to a text file named txt_path.

Here is a step-by-step explanation of the code:

- The for loop iterates over each detection in pred (i.e., the output of the YOLO model).

- If webcam is True, it gets the image path, label, original image (im0), and frame number for the current detection. Otherwise, it sets s to an empty string and im0 and frame to their respective values in the dataset object.
- It converts the path p to a Path object, and constructs the save path for the image (save_path) and the text file containing labels (txt_path).
- It computes the normalization gain (gn) by taking the shape of the original image im0 and selecting every other element ([1, 0, 1, 0]).
- If there are any detections (len(det) > 0), it rescales the bounding box coordinates from the img size to the im0 size using the scale_coords function and then iterates over each bounding box (for *xyxy, conf, cls in reversed(det):).
- It extracts the (x1, y1, x2, y2) coordinates, confidence score (conf), and predicted class (cls) from the bounding box.
- It computes the center point of the bounding box by taking the average of the (x1, y1) and (x2, y2) coordinates, and then draws a green circle at that point (cv2.circle(im0, Centrepoint, 5, (0, 255, 0), -1)).
- It then draws the center point coordinates as text on the image (cv2.putText(im0, str(CenterPoint), CenterPoint, cv2.FONT_HERSHEY_PLAIN, 3, (0, 0, 255))).
- It computes the center point of the image (center_width, center_height), and then draws horizontal and vertical lines passing through the center (cv2.line(im0, (0, center_height), (im0.shape[1], center_height), (255, 0, 0), 2) and cv2.line(im0, (center_width, 0), (center_width, im0.shape[0]), (255, 0, 0), 2)).
- It then determines the position of the object relative to the center of the image based on the location of the center point of the bounding box. If the center point is on the right and top side of the image, it draws the text 'object is on the right top side' on the image (cv2.putText(im0, 'object is on the right top side', (im0.shape[1]-200, 20), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)). Similarly, if the center point is on the left and top side, it draws the text 'object is on the left top side', and if it's on the bottom right side, it draws 'object is on the bottom right side'. Otherwise, it draws 'object is on the bottom left side'.

3.3 Architecture

The YOLOv7 architecture, developed by Wongkinyui, is an extension of the YOLO (You Only Look One) family of object detection models. It is a Deep Convolutional Neural Network (CNN) architecture designed to be fast and efficient while still achieving high accuracy in object detection. The YOLOv7 architecture can be divided into three main parts: input layer, backbone network, and detection head.

- **Input Layer**: The input layer takes an image as input and resizes it to a fixed size. The resized image is then passed through the backbone network.
- **Backbone Network**: The backbone network is responsible for extracting features from the input image. The YOLOv7 architecture uses the CSPDarknet53 backbone network, which is a modified version of the Darknet53 architecture used in previous YOLO versions. The CSPDarknet53 network is designed to be more efficient and accurate than the original Darknet53 architecture.
- **Detection Head**: The detection head consists of a series of convolutional layers that predict bounding boxes and class probabilities for objects in the input image. The YOLOv7 architecture uses a novel approach to a prediction that involves combining multiple scales and aspect ratios to improve accuracy. The detection head also uses anchor boxes, which are a set of predefined bounding boxes with a fixed size and aspect ratio that are used to improve the accuracy of predictions.

The YOLOv7 architecture also uses several optimization techniques to improve performance, including batch normalization, leaky ReLU activation functions, and spatial pyramid pooling. In spatial pyramid pooling, the input image is divided into multiple regions and a pooling operation is performed on each region to extract finer features.

The YOLOv7 architecture is designed to be fast and efficient while still achieving high accuracy in object detection. Using a modified version of the Darknet53 architecture and a novel approach to inference that combines multiple scales and aspect ratios, YOLOv7 can achieve advanced performance on various object detection benchmarks.

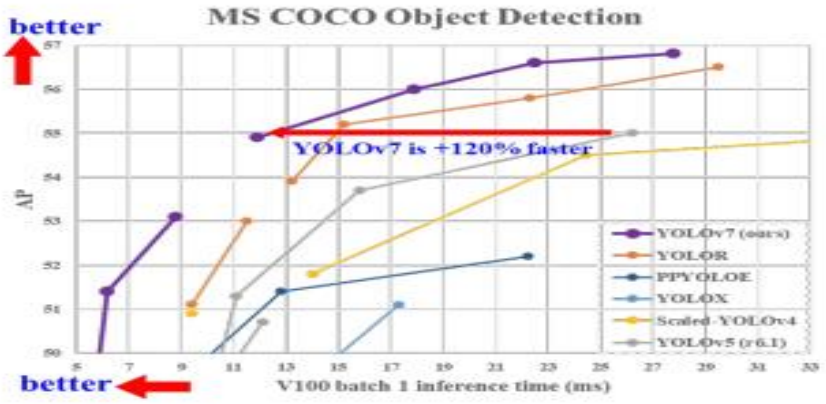


Figure 3.12 MS COCO object detection graph

The MS COCO object detection graph present on the YOLOv7 GitHub page is a visualization of the detection performance of YOLOv7 on the MS COCO dataset. The graph shows the mAP (average precision) of YOLOv7 on the MS COCO dataset for different IoU (intersection over union) thresholds ranging from 0.5 to 0.95. MAP is a commonly used metric in object detection that measures the accuracy of an algorithm in object localization and classification. The graph shows that YOLOv7 achieves a high mAP of over 80% for IoU thresholds between 0.5 and 0.75, indicating that the algorithm can accurately detect and classify objects in the MS COCO dataset. As the IoU threshold increases, the MAP decreases, as it becomes more difficult for the algorithm to accurately match the predicted bounding box with the ground truth box. The graph also shows the performance of YOLOv7 compared to other advanced object detection algorithms, such as EfficientDet and YOLOv5. In general, YOLOv7 achieves similar or better performance than these algorithms, especially for lower IoU thresholds.

The MS COCO object detection graph provides a useful summary of YOLOv7's performance on a challenging and widely used object detection dataset and can help users understand the algorithm's strengths and limitations.

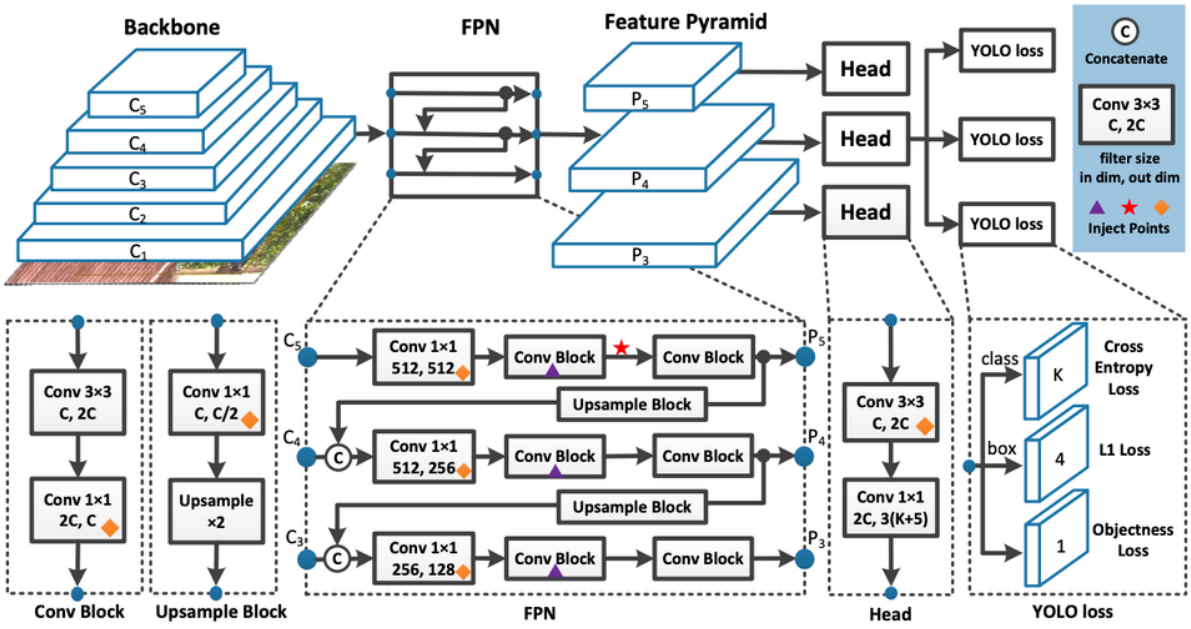


Figure 3.13 Architecture of YOLOv7

3.4 Tools and Technology

3.4.1 NVIDIA AGX XAVIER

Nvidia AGX Xavier is a high-performance, low-power embedded system designed for AI and deep learning applications. It is part of the Nvidia AGX family of systems, which also includes the Nvidia AGX Jetson and Nvidia AGX DRIVE platforms. The Nvidia AGX Xavier is based on a custom system-on-a-chip (SoC) that combines several hardware accelerators, including a multi-core ARM CPU, a Volta GPU with 512 CUDA cores, and an eight-core Nvidia Carmel CPU. Two Nvidia Denver 2 CPUs, and a Deep Learning Accelerator (DLA) for high-performance prediction. The system also includes 16GB of LPDDR4x memory and 32GB of eMMC storage. Nvidia AGX Xavier is designed for use in a wide range of AI and deep learning applications, including robotics, autonomous vehicles, drones, and intelligent video analytics. It provides high efficiency and low power consumption, making it well suited for use in embedded systems where power and space limitations are concerns. The system supports a wide range of AI frameworks, including Tensors, TensorFlow, PyTorch, and Caffe, and includes a software development kit (SDK) that provides tools and libraries for developing and deploying AI applications on the platform. Nvidia AGX Xavier also supports hardware acceleration for various deep learning algorithms, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), and generative adversarial networks (GANs). The Nvidia AGX Xavier is a powerful and versatile embedded system that delivers high-performance AI and deep learning capabilities in a compact and energy-efficient package. It is well suited for a wide range of applications in industries such as robotics, automotive, and video analytics.

YOLOv7 is possible to use on Nvidia AGX Xavier and can be a good option for real-time object detection applications. Nvidia AGX Xavier is a powerful embedded system with a multi-core CPU, Volta GPU, and several hardware accelerators that can be used to speed up deep learning algorithms like YOLOv7. To use YOLOv7 on Nvidia AGX Xavier, you will first need to install the necessary software, including the Nvidia JetPack SDK, which includes CUDA, cuDNN, and other libraries required for deep learning on the Nvidia platform. Once the software is installed, you can download and compile the YOLOv7 code on the Nvidia AGX Xavier. You can use the pre-trained weights for YOLOv7 or train the model on your custom dataset using the provided training script. To run YOLOv7 on Nvidia AGX Xavier, you will typically use the CUDA libraries

and TensorRT optimization framework to optimize the model for deployment on the device. You can then use the optimized model to perform real-time object detection on images or video streams, using YOLOv7 on Nvidia AGX Xavier can be an effective way to perform real-time object detection on embedded systems, taking advantage of the powerful hardware and software capabilities of the Nvidia platform.

Figure 3.14 Nvidia AGX Xavier

Here are some key aspects of the Nvidia AGX Xavier GPU architecture:

1. **GPU:** The Nvidia AGX Xavier features a powerful GPU based on the Nvidia Volta architecture. It incorporates 512 CUDA cores and delivers a peak performance of 1.3 teraflops (floating-point operations per second).

2. **Tensor Cores:** The GPU in the AGX Xavier is equipped with Tensor Cores, which are specialized hardware units designed to accelerate matrix operations commonly used in deep learning algorithms. Tensor Cores can perform mixed-precision calculations, allowing for faster AI computations and improved energy efficiency.

3. **Deep Learning Accelerator (DLA):** The AGX Xavier also includes dedicated Deep Learning Accelerators (DLAs) designed to offload specific AI computations from the GPU. The DLAs are particularly optimized for executing convolutional neural networks (CNNs), which are commonly used in tasks like image recognition and object detection.

4. **CPU:** In addition to the GPU, the AGX Xavier integrates an eight-core ARMv8.2 64-bit CPU complex. This CPU complex includes eight high-performance ARM Cortex-A57 cores, which provide general-purpose computing power for running applications and managing system tasks.

5. **Memory:** The AGX Xavier features 16 GB of high-bandwidth LPDDR4x RAM, providing ample memory capacity for AI workloads. The memory bandwidth is approximately 137 GB/s, allowing for efficient data access and processing.

6. **Storage:** The platform provides 32 GB of eMMC 5.1 onboard storage for the operating system and applications. Additionally, it supports external storage devices through various interfaces like USB, PCIe, and SATA.

7. **Connectivity:** The AGX Xavier is designed to facilitate connectivity with a wide range of devices. It includes multiple USB ports, Gigabit Ethernet, HDMI, DisplayPort, and PCIe Gen3 interfaces. This allows for easy integration with cameras, sensors, displays, and other peripherals.

8. **Power Efficiency:** The AGX Xavier is designed to deliver high performance while keeping power consumption in check. It is built on a 12 nm FinFET manufacturing process, which helps optimize power efficiency, making it suitable for power-constrained applications such as autonomous vehicles and drones.

The Nvidia AGX Xavier GPU architecture combines powerful GPU capabilities with dedicated AI hardware accelerators, efficient CPUs, and ample memory to deliver high-performance AI computing for autonomous machines and AI applications. Its architecture is specifically tailored to meet the demands of AI workloads, making it well-suited for tasks such as deep learning, computer vision, and robotics.

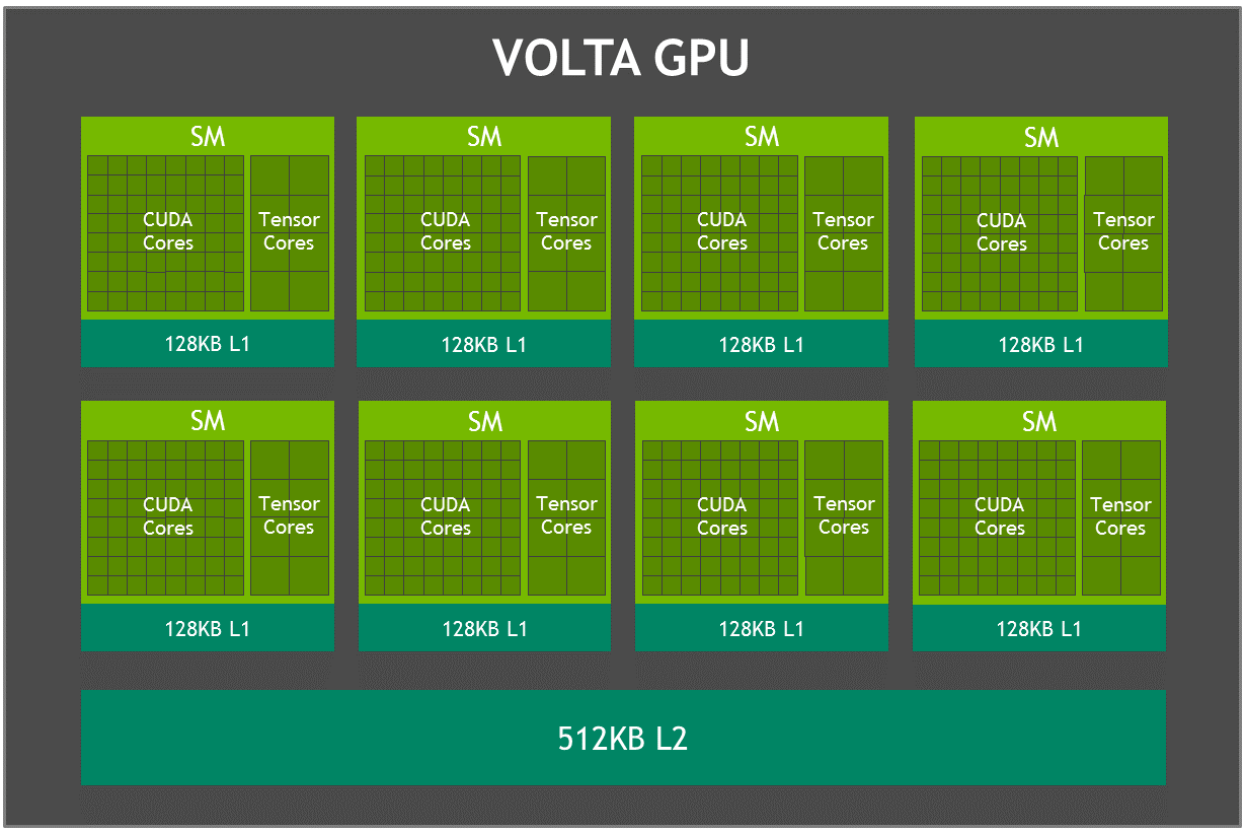


Figure 3.15 GPU architecture

Why to use GPU for training an object detection model rather than a CPU ?

GPUs (Graphics Processing Units) are commonly preferred over CPUs (Central Processing Units) for object detection training using YOLOv7 due to the following reasons:

1. **Parallel Processing:** GPUs excel at parallel processing, meaning they can simultaneously perform multiple computations on a large number of data points. In object detection training, this parallelism is beneficial because it allows for concurrent processing of multiple images or regions within an image, which significantly speeds up the training process. YOLOv7 involves complex mathematical operations on large matrices, and GPUs can efficiently handle these computations in parallel, resulting in faster training times compared to CPUs.

2. **Dedicated AI Acceleration:** Modern GPUs, such as those based on Nvidia's CUDA architecture, include specialized AI acceleration features like Tensor Cores. These

hardware units are designed specifically for accelerating deep learning computations, such as matrix multiplications commonly used in convolutional neural networks (CNNs) like YOLOv7. Tensor Cores can perform mixed-precision calculations and deliver substantial speedups in deep learning workloads, making GPUs highly efficient for object detection training.

3. **Larger Memory Bandwidth:** GPUs typically have much higher memory bandwidth than CPUs. In object detection training, large amounts of data need to be processed and transferred between memory and processing units. The high memory bandwidth of GPUs enables faster data transfer, reducing the time spent on memory-bound operations and improving overall training performance.

4. **Availability of Deep Learning Frameworks:** GPUs are well-supported by popular deep learning frameworks, including TensorFlow, PyTorch, and Caffe, which provide optimized GPU implementations for neural network computations. These frameworks leverage the parallel processing capabilities of GPUs, enabling efficient training and inference of deep learning models, including YOLOv7.

5. **Scalability:** GPUs can be easily scaled by adding multiple GPUs to a system, allowing for distributed training across multiple devices. This enables faster training times by dividing the workload among multiple GPUs and processing data in parallel. Additionally, frameworks like TensorFlow and PyTorch offer built-in support for multi-GPU training, simplifying the process of utilizing multiple GPUs.

6. **Cost-Effectiveness:** While GPUs can be expensive, they often offer a more cost-effective solution for deep learning tasks compared to CPUs. GPUs provide significantly higher performance per dollar when it comes to training deep neural networks. Their parallel processing capabilities allow for faster training times, leading to reduced infrastructure costs in the long run.

While CPUs still play a vital role in object detection training, GPUs are typically favored due to their exceptional parallel processing capabilities, dedicated AI acceleration features, high memory bandwidth, and support from deep learning frameworks. These factors collectively make GPUs

highly efficient and effective for training YOLOv7 and other deep learning-based object detection models.

	GPU / GPGPU	CPU
Memory Latency	• Context switching automatically done in HW (ZERO cycle latency) • GPU can switch to different thread until data returns from memory	• Context switching handled in SW (by OS) • Stalls if there is an L1/L2 cache miss. Creates bubbles during program execution on cache misses
Thread Management	• Thread scheduler and dispatch unit implemented in HW (manages large # threads with minimal overhead) • Solving problems with massively parallel data such as "sliding window" operations like FIRs, convolutions and other signal processing algorithms are more efficient on the GPU.	• Thread overhead and work-item creation done in SW (CPU needs to manage thread pool) • Maximum # threads is limited compared to GPU.
Parallelism	• HW designed to facilitate parallel operations on all shaders (coordination handled in HW).	• CPU/SW and multi-core CPU (inter-core) communication overhead exists • CPU thread overhead is "heavier" than equivalent GPU thread overhead.

Figure 3.16 GPU vs CPU

3.4.2 YOLOv7

YOLOv7 is a modified version of the YOLOv5 algorithm developed by WongKinYiu. The YOLO (You Only Look Once) algorithm is a popular object detection algorithm used in computer vision applications. It is known for its speed and accuracy in detecting objects in live video streams. Changes were made to YOLOv5 in YOLOv7 to further improve its accuracy and speed. These changes include the backbone network architecture, training procedure, and post-processing algorithm changes. These changes are based on the authors' research on object detection and computer vision. It should be noted that YOLOv7 is not the official version of YOLO by the original developer, nor has it undergone rigorous testing and verification like the official version. As such, it may not be as reliable or robust as the official version and should be used with caution.YOLOv7, like its predecessor YOLOv5, aims to be a fast object detection algorithm. It achieves this speed by using a single neural network to perform real-time object detection and classification. This approach contrasts with other algorithms that use two separate networks for object detection and classification, which are slower and less efficient. Additionally, YoloV7 has undergone several changes to further improve its speed and efficiency. These changes include changes to the backbone architecture responsible for feature extraction, as well as improvements to the training process and post-processing algorithms. One of the biggest changes in the YoloV7 architecture is the use of a Modified Spatial Pyramid Pooling (MSPP) module, which reduces the number of parameters in the network while maintaining its accuracy. This module helps speed up the network’s inference time, allowing it to process more frames per second. Additionally, YoloV7 uses a technique called dynamic convolution, which adjusts the network's receptive field (the area of the image each neuron can "see") based on the size and position of objects in the image. This reduces the amount of computation needed for object detection, further increasing the speed and efficiency of the algorithm. The speed of YOLOv7 is achieved through a combination of architecture optimization, efficient training, and post-processing techniques. These optimizations reduce the amount of computation required for object detection, allowing the algorithms to process images in real-time while maintaining high accuracy.

To run YOLOv7, you will need the following requirements:

A computer with a GPU: YOLOv7 requires a graphics processing unit (GPU) to accelerate neural network calculations. Nvidia GPUs are commonly used for deep learning, and YOLOv7 is optimized for the Nvidia CUDA architecture.

CUDA Toolkit: CUDA Toolkit is a software development kit (SDK) provided by Nvidia that allows developers to harness the power of Nvidia GPUs for deep learning. YOLOv7 is optimized for the CUDA toolkit and requires a compatible version to run.

cuDNN: cuDNN is a library of algorithms optimized for deep learning provided by Nvidia. YOLOv7 requires cuDNN for high-performance computations.

Python: YOLOv7 is implemented in Python, so you will need to install Python on your system. It is recommended to use Python 3.6 or higher.

NumPy: NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions. YOLOv7 uses NumPy for array processing.

Matplotlib: Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. YOLOv7 uses Matplotlib for visualization.

SciPy: SciPy is a library for scientific computing in Python. It provides many efficient and user-friendly interfaces for tasks such as numerical integration, optimization, signal processing, linear algebra, and more. YOLOv7 uses SciPy for various mathematical operations.

Pillow: Pillow is a library for the Python programming language that adds support for opening, manipulating, and saving many different image file formats. YOLOv7 uses Pillow for image manipulation.

OpenCV: OpenCV is a popular computer vision library that provides tools for image and video processing. YOLOv7 uses OpenCV to read and process images and videos.

Pre-trained weights: YOLOv7 comes with pre-trained weights for the COCO dataset, but you can also train the model on your custom dataset. If you plan to use pre-trained weights, you will need to download them from the YOLOv7 GitHub repository.

Custom dataset (optional): If you want to train YOLOv7 on a custom dataset, you will need to have labeled images and annotations for object detection.

labelImg: For custom datasets, you will also need labeled images and annotations, labelImg is used for the annotation of objects in the images(For creating a custom dataset).

TensorFlow: YOLOv7 is built on top of the TensorFlow deep learning framework, so you will need to have TensorFlow installed on your system. It is recommended to use TensorFlow 2.4 or higher.

PyTorch is used as a deep-learning framework for building and training neural networks in YOLOv7. YOLOv7 is implemented using PyTorch because it provides an easy-to-use interface and efficient computational graph generation for building and training deep neural networks. In YOLOv7, PyTorch is used to define the architecture of the neural network, including the backbone, neck, and head of the network. The backbone is responsible for extracting features from the input image, the neck combines these features usefully, and the head predicts the final object classes and bounding boxes. PyTorch is also used to train neural networks on custom datasets. The network is trained using a backpropagation algorithm, which updates the network weights to minimize the loss function between the predicted output and the ground truth labels. PyTorch provides efficient computation of gradients required for backpropagation, as well as tools for optimizing network weights using techniques such as stochastic gradient descent. After training, PyTorch is used to save the trained weights of the network, which can be loaded and used to make predictions on new images or video streams. PyTorch plays a crucial role in YOLOv7 by providing an efficient and easy-to-use framework for building, training, and deploying neural networks.

Creating a virtual environment is a recommended step when working with YOLOv7 or any other Python-based project. A virtual environment allows you to create an isolated Python environment with its own packages and dependencies, separate from the global Python environment on your system.

Here are the steps to create a virtual environment for YOLOv7:

1. Install virtualenv using pip:

   Command:

   pip install virtualenv

2. Navigate to the directory where you want to create your virtual environment, then create a new virtual environment with the following command:

   Command:

   virtualenv myenv

   Note: Replace "myenv" with a name of your choice for your virtual environment.

3. Activate the virtual environment:

   Command:

   source myenv/bin/activate

4. Install the required packages for YOLOv7 within the virtual environment:

   Command:

   pip install tensorflow opencv-python numpy matplotlib scipy pillow

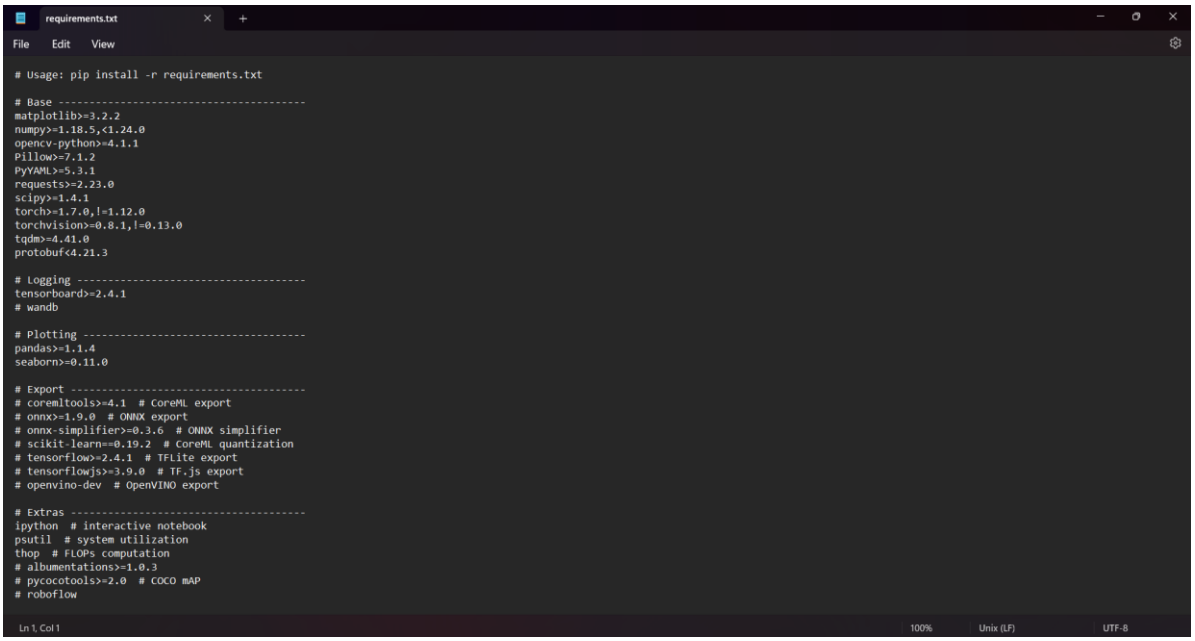


Figure 3.17 Requirements for YOLOv7

Once you have created and activated your virtual environment and installed the required packages, you can use YOLOv7 and its associated scripts and tools within that environment. When you are finished working with YOLOv7, you can deactivate the virtual environment with the command:

deactivate

(OR)

You can install Anaconda to work with YOLOv7 by following these steps:

1. Download the appropriate Anaconda distribution for your system from the official website: https://www.anaconda.com/products/individual
2. Install Anaconda by following the instructions for your operating system.
3. Create a new conda environment for YOLOv7:

Command:

conda create --name yolov7 python=3.8

Note: This will create a new environment called "yolov7" with Python 3.8 installed.

4. Activate the conda environment:

Command:

conda activate yolov7

5. Install the required packages for YOLOv7 within the conda environment:

Command:

conda install tensorflow-gpu=2.6.0 opencv matplotlib pillow

Note: If you don't have an Nvidia GPU or don't want to use it for acceleration, you can install the CPU version of TensorFlow instead:

Command:

conda install tensorflow=2.6.0 opencv matplotlib pillow

6. Clone the YOLOv7 repository from GitHub:

Command:

git clone https://github.com/WongKinYiu/yolov7.git

7. Navigate to the YOLOv7 directory:

Command:

cd yolov7

8. Download the pre-trained weights for YOLOv7:

Command:

wget https://github.com/WongKinYiu/yolov7/releases/download/v1.0/yolov7-tiny.ckpt -P weights/

9. You are now ready to use YOLOv7 within the conda environment. You can run the demo script to test the installation:

Command:

python detect.py --weights weights/yolov7-tiny.ckpt --image data/images/bus.jpg

Note: These instructions assume that you have an Nvidia GPU and want to use it for acceleration. If you don't have an Nvidia GPU, you can still use YOLOv7, but it will run slower without GPU acceleration.

3.4.3 labelImg

To create a custom dataset using labelImg for YOLO v7, you can follow these steps:

1. **Install the required dependencies:**

- Install labelImg: Follow the instructions provided in the labelImg repository (https://github.com/tzutalin/labelImg) to install it on your system.
- Install YOLO v7: Follow the instructions specific to YOLO v7 to install the necessary libraries and dependencies.

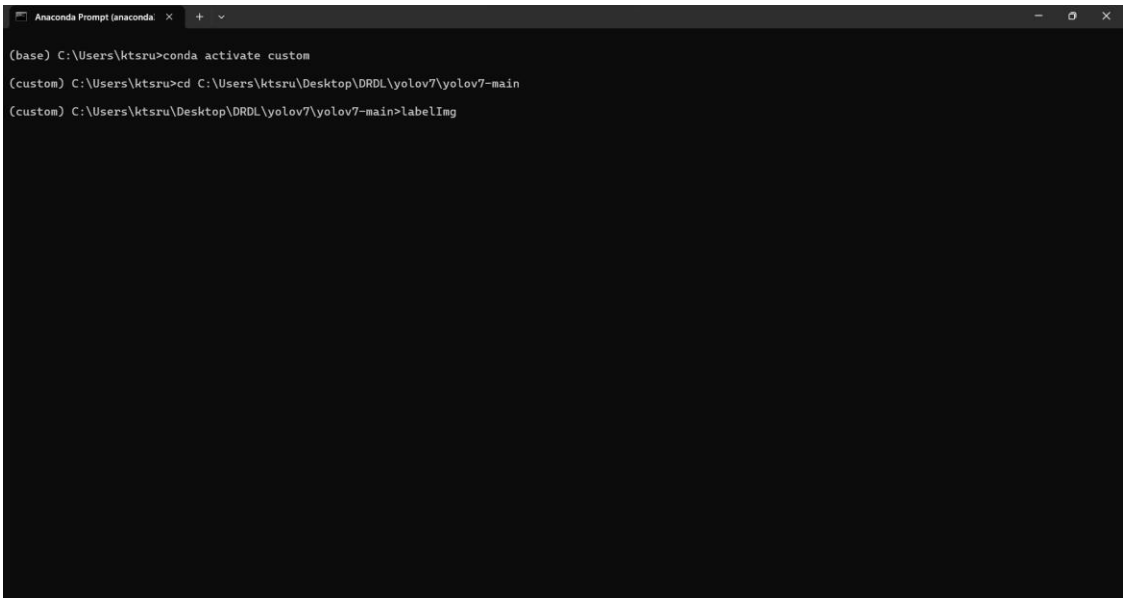


Figure 3.18 Command Prompt

2. **Prepare your dataset:**

- Gather a set of images that you want to annotate for object detection.
- Create a folder structure for your dataset, with separate folders for images and annotations.

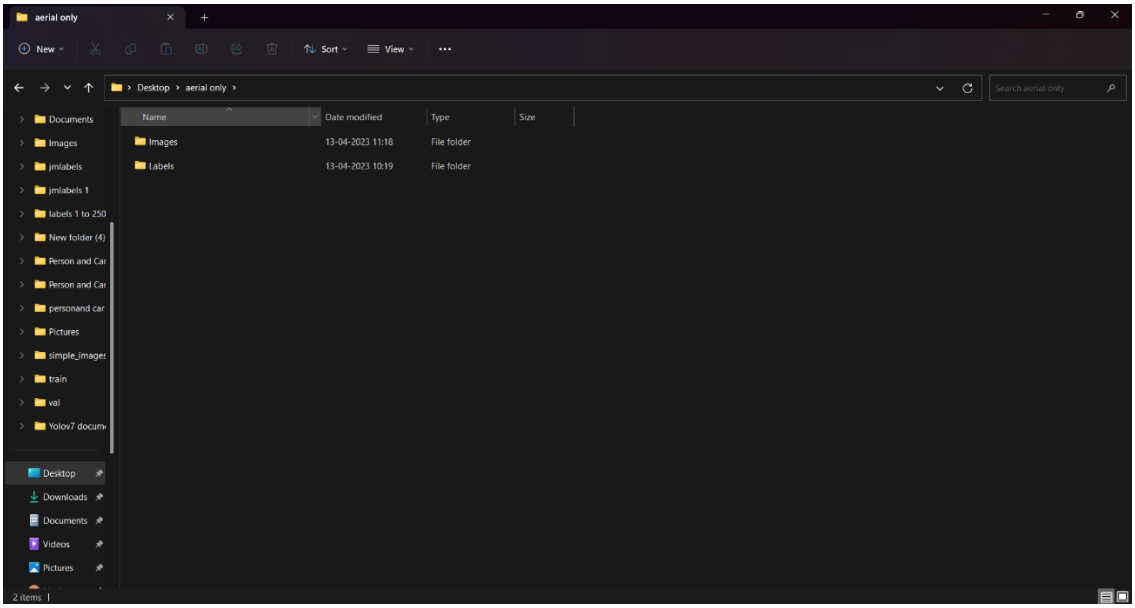


Figure 3.19 Dataset Files

3. **Annotate the images using labelImg:**

- Open the labelImg tool.
- Click on "Open Dir" and select the folder where your images are stored.
- Start annotating objects in the images by drawing bounding boxes around them.
- Select the appropriate label for each annotated object from the dropdown menu.
- Save the annotations in XML format by clicking on "Save" or using the shortcut (Ctrl + s).

4. **Convert annotations to YOLO format:**

- Once you have annotated all the images, you need to convert the annotations from XML to YOLO format.
- You can use a script like xml_to_yolo.py (available in the YOLO repository) to convert the annotations.

- This script will generate text files for each image, where each line contains the object class and the coordinates of the bounding box relative to the image size.

5. Organize your dataset:

- Move the generated annotation files (in YOLO format) to the same folder as their corresponding images.
- Ensure that the image and annotation file names match (e.g., image.jpg and image.txt).

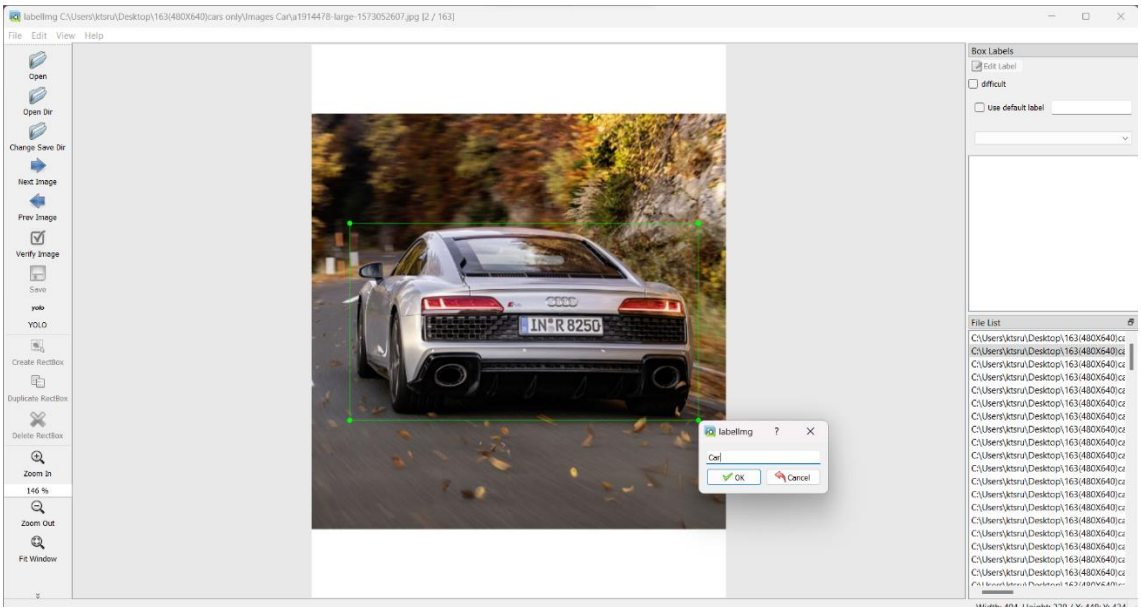


Figure 3.20 labelImg

6. Split your dataset (optional):

- If you want to split your dataset into train and validation sets, you can do so by creating separate folders for each set.
- Move a certain percentage of your images and their corresponding annotation files into the validation set folder.

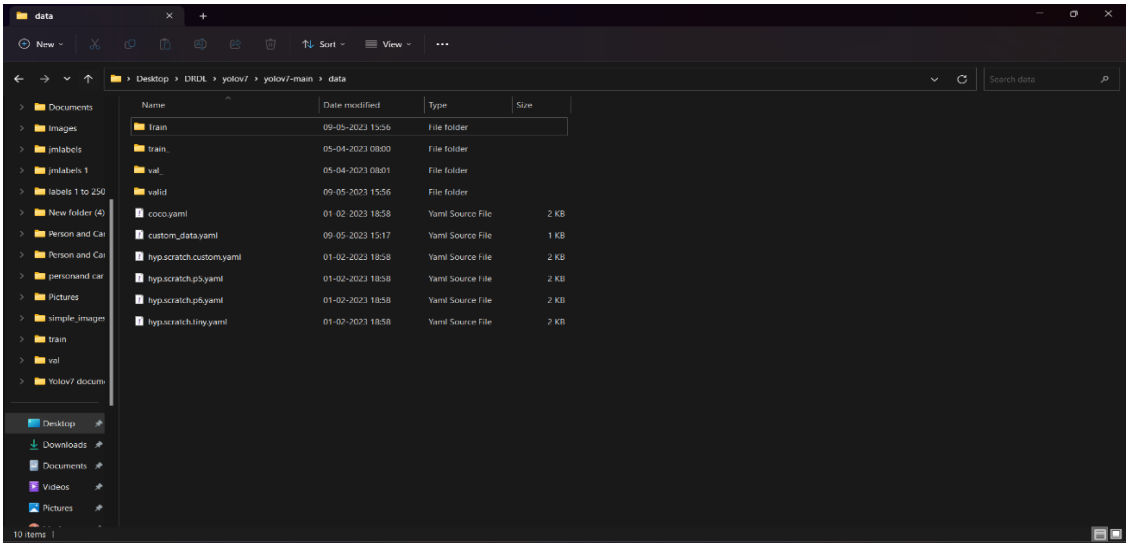


Figure 3.21 Train and Validation Folders

7. Train YOLO v7 on your custom dataset:

- Use the YOLO v7 training script provided in the YOLO repository, specifying your custom dataset and configuration parameters.
- Train the YOLO v7 model using your custom dataset and follow the specific instructions for YOLO v7 training.

By following these steps, you can create a custom dataset using labelImg and train YOLO v7 for object detection.

Chapter-4

Simulations and Results

4.1 Output detected

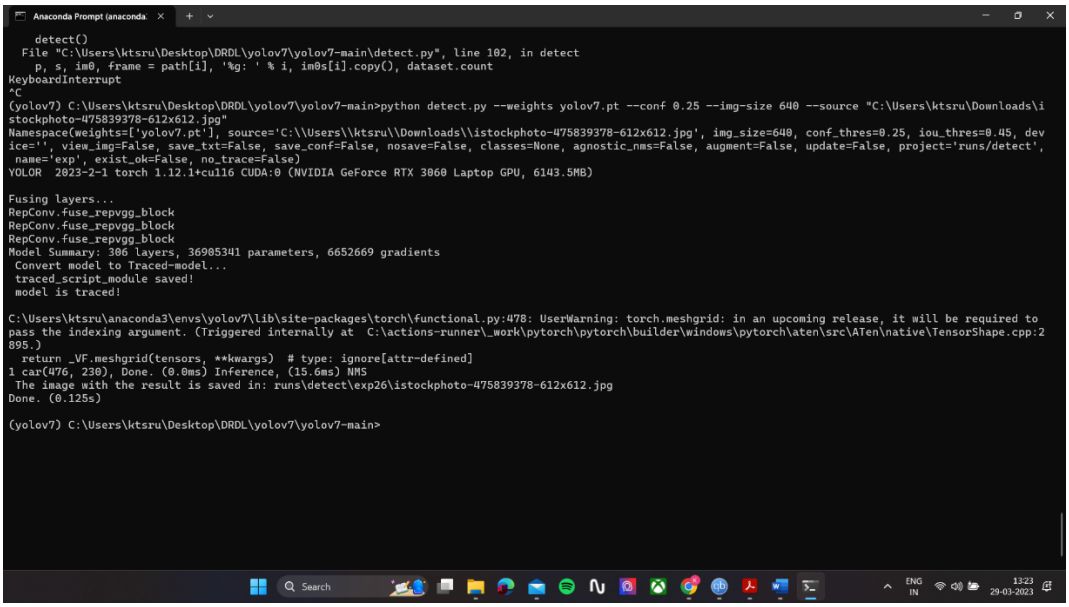


Figure 4.1 Screenshot of the Command Prompt

Figure 4.2 Output of the detection on the trained model of the custom dataset

4.2 Challenges faced

False Predictions: The major problem faced after training the custom dataset is that there are a lot of false predictions. These false predictions are caused due to low-resolution images used in the custom dataset or false labelings of images in the dataset. After training various custom datasets I have observed that the dataset should contain both aerial and ground view images of the object as part of training, if we only give aerial view images of the Car for training then we have seen that the trained model detects every Rectangle object as the car because if the dataset containing only aerial view images of the object then it would not train accurately as there will be very less detail or clarity of the object in the images due to long-range shots(images) of the object. Here are some common causes of false predictions:

- **Insufficient or Imbalanced Training Data:** If the training data does not adequately represent the real-world scenarios or if it is imbalanced in terms of object classes or object instances, the model may struggle to generalize well and produce false predictions.
- **Occlusion or Partially Visible Objects:** When objects are partially occluded or only partially visible in the image, it can be challenging for the model to accurately detect and classify them, leading to false predictions.
- **Ambiguous or Similar Object Classes:** If there are object classes in the dataset that are visually similar or have overlapping features, the model may have difficulty distinguishing between them, resulting in false predictions.\
- **Limited Model Capacity or Complexity:** If the chosen object detection model is not complex enough to capture the intricacies of the dataset, it may struggle to learn and make accurate predictions, leading to false positives or false negatives.
- **Noisy or Incorrect Annotations:** Inaccurate or noisy annotations in the training data can mislead the model during training, causing it to learn incorrect patterns and make false predictions.
- **Adverse Environmental Conditions:** Challenging environmental conditions, such as poor lighting, blurriness, or variations in scale and perspective, can make it difficult for the model to detect objects accurately, leading to false predictions.

- **Model Overfitting or Underfitting:** If the model is overfitting (performing well on the training data but poorly on unseen data) or underfitting (not capturing the patterns in the data), it may produce false predictions.
- **Model Hyperparameters and Training Settings:** Incorrect or suboptimal choices of hyperparameters and training settings, such as learning rate, regularization, or batch size, can impact the model's performance and result in false predictions.
- **Model Architecture and Design Choices:** Different object detection architectures have varying strengths and weaknesses. The chosen architecture may not be well-suited for the specific dataset or task, leading to false predictions.
- **Inference Time Constraints:** In some real-time applications, the model may have limited time for inference, which can impact its accuracy. It may compromise precision to achieve faster processing, resulting in false predictions.

Addressing these factors typically involves analyzing and understanding the specific challenges in your dataset and iteratively refining the model through techniques like data augmentation, model fine-tuning, architecture selection, hyperparameter tuning, and gathering more diverse and representative training data.

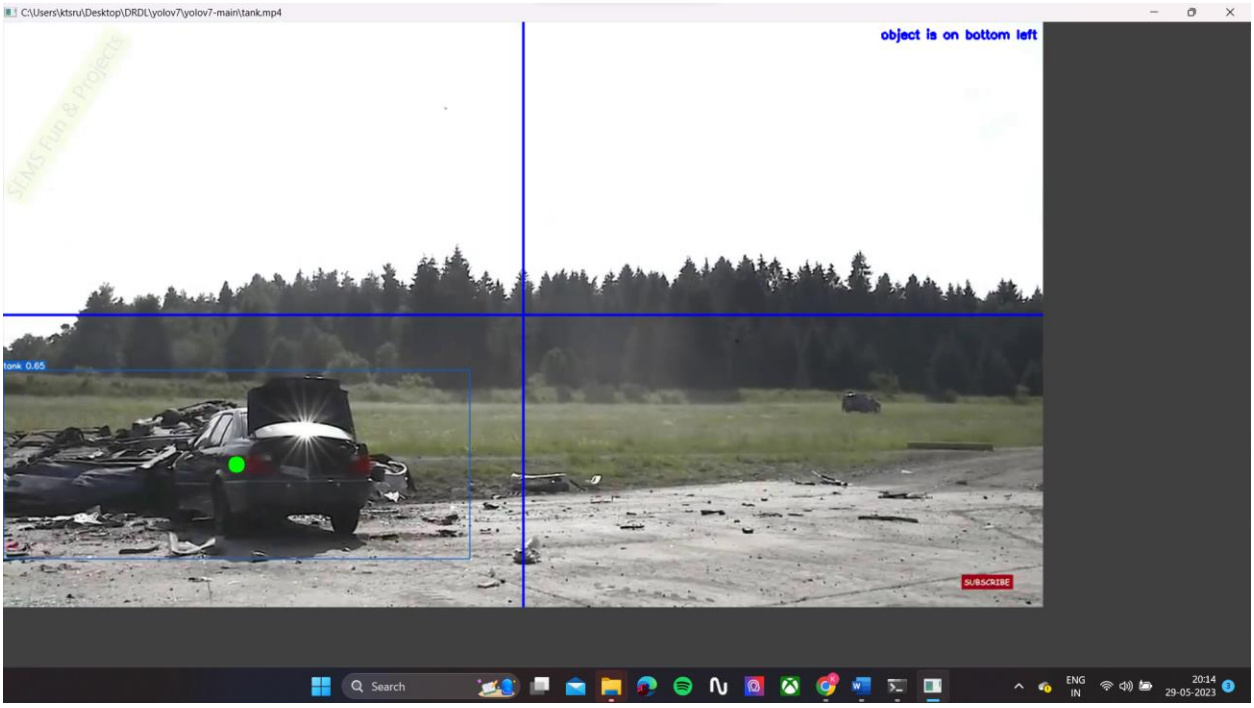


Figure 4.3 Figure of false predictions

4.3 Mitigations

When creating a custom dataset for aerial view object detection using YOLOv7, it's important to follow certain rules of labeling to reduce false detections and improve the accuracy of your model. Here are some guidelines to consider:

- **Accurate bounding boxes:** Ensure that the bounding boxes accurately encapsulate the entire object of interest. The box should tightly fit around the object without including any irrelevant background. In aerial imagery, objects may appear smaller, so precise annotation is crucial.
- **Object occlusion:** If an object is partially occluded by another object or by the environment, label only the visible portion. Avoid labeling occluded or obstructed objects that are not clearly discernible.
- **Object classes:** Define the classes of objects you want to detect and annotate accordingly. Maintain consistency in labeling classes throughout the dataset. For example, if you are detecting cars and buses, label all instances of cars as "car" and buses as "bus" consistently.
- **Object size and resolution:** Consider the size and resolution of the objects in your aerial imagery. If the objects are small, it's important to annotate them accurately. Use appropriate tools and zoom in if necessary to label smaller objects effectively.
- **Labeling difficult cases:** Some objects might be challenging to detect due to their orientation, low contrast, or complex shapes. Make sure to annotate these cases accurately, as they are important for training the model to handle real-world variations.
- **Multiple instances:** If there are multiple instances of the same object class within an image, label each instance with its respective bounding box. Avoid merging multiple objects into a single bounding box.
- **Non-object regions:** In aerial imagery, there can be regions that do not contain any objects of interest. It's crucial to label these regions as "background" or "no object" to help the model differentiate between objects and empty areas.
- **Consistent labeling style:** Follow a consistent labeling style throughout the dataset. Use the same annotation tools and ensure that all annotators understand and adhere to the labeling guidelines to maintain uniformity.

- **Quality control:** Perform regular quality checks on the labeled data to ensure accuracy and consistency. Review a subset of annotations to verify that they align with the labeling rules and meet your requirements.
- **Sufficient data variation:** Aim for a diverse dataset that includes variations in lighting conditions, object sizes, orientations, backgrounds, and occlusions. This will help the model generalize better and reduce false detections in real-world scenarios.

By following these rules of labeling, you can create a high-quality custom dataset for aerial view object detection, reducing false detections and improving the performance of your YOLOv7 model.

Chapter-5

5.1 References Links

https://www.sciencedirect.com/science/article/pii/S1877050922001363

https://link.springer.com/article/10.1007/s11042-022-13644-y

https://journalofbigdata.springeropen.com/articles/10.1186/s40537-021-00434-w

chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://arxiv.org/pdf/1807.05511.pdf

https://arxiv.org/abs/1506.01497

https://www.researchgate.net/publication/353894621_Research_on_Object_Detection_Algorithm_Based_on_Deep_Learning

https://towardsdatascience.com/yolo-object-detection-with-opencv-and-python-21e50ac599e9

https://github.com/heartexlabs/labelImg

https://github.com/WongKinYiu/yolov7

https://viso.ai/deep-learning/yolov7-guide/

https://blog.paperspace.com/yolov7/

https://www.researchgate.net/publication/5847739_Introduction_to_artificial_neural_networks

https://www.academia.edu/7197728/Research_Paper_on_Basic_of_Artificial_Neural_Network

https://arxiv.org/abs/1506.02640

https://arxiv.org/abs/1612.08242

https://arxiv.org/abs/1804.02767

https://arxiv.org/abs/2004.10934

https://arxiv.org/abs/2008.01204