OBJECT DETECTION

MINI PROJECT REPORT SUBMITTED IN THE PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE AWARD OF THE DEGREE OF BACHELOR IN TECHNOLOGY IN COMPUTER SCIENCE AND ENGINEERING DATA ANALYTICS/AI&ML BY

NAME OF THE STUDENT:
Subhadip Samanta
Wrishav Sett
Sudipta Saha
Subarna Das

ROLL NO.:
TNU2020021100001
TNU2020021100004
TNU2020021100005
TNU2020053100011

UNDER THE SUPERVISION OF
Prof. Jaydeb Mondal
TEACHING ASSISTANT, Dept. of Computer Science and Engineering, TNU

Dept. of Computer Science of Engineering
THE NEOTIA UNIVERSITY, WEST BENGAL, INDIA
May, 2023

Acknowledgement

I would like to express my profound gratitude to ________________, of Computer Science Engineering Department of The Neotia University for their contributions to the completion of my project titled Object Detection.

I would like to express my special thanks to our mentor Mr./Mrs. ________________ for his/her time and efforts he/she provided throughout the year by supporting us and pointing us in the right direction. Your useful advice and suggestions were really helpful to me during the project's completion. In this aspect, I am eternally grateful to you.

I would like to acknowledge that this project was completed entirely by me and not by someone else.

Signature

Your name

Certificate

We hereby recommend that the Project entitled "**Object Recognition**" worked under our guidance may please be accepted in the partial fulfillment of the requirement for the degree of "Bachelor in Technology" in the Computer Science and Engineering with specialization in Data Analytics of 'The Neotia University'. The project report in our opinion is worthy for its acceptance. During the work students – '**Subhadip Samanta**' was found to be sincere, regular and hardworking and have successfully completed the thesis work assigned to him.

________________	________________
(HOD/TIC, CSE)	Prof. Jaydeb Mondal
The Neotia University	The Neotia University

Certificate

We hereby recommend that the Project entitled “**Object Recognition**” worked under our guidance may please be accepted in the partial fulfillment of the requirement for the degree of “Bachelor in Technology” in the Computer Science and Engineering with specialization in Data Analytics of ‘The Neotia University’. The project report in our opinion is worthy for its acceptance. During the work students – ‘**Wrishav Sett**’ was found to be sincere, regular and hardworking and have successfully completed the thesis work assigned to him.

(HOD/TIC, CSE)	Prof. Jaydeb Mondal
The Neotia University	The Neotia University

Certificate

We hereby recommend that the Project entitled "**Object Recognition**" worked under our guidance may please be accepted in the partial fulfillment of the requirement for the degree of "Bachelor in Technology" in the Computer Science and Engineering with specialization in Data Analytics of 'The Neotia University'. The project report in our opinion is worthy for its acceptance. During the work students – '**Sudipta Saha**' was found to be sincere, regular and hardworking and have successfully completed the thesis work assigned to him.

(HOD/TIC, CSE)	Prof. Jaydeb Mondal
The Neotia University	The Neotia University

Certificate

We hereby recommend that the Project entitled "**Object Recognition**" worked under our guidance may please be accepted in the partial fulfillment of the requirement for the degree of "Bachelor in Technology" in the Computer Science and Engineering with specialization in AI&ML of 'The Neotia University'. The project report in our opinion is worthy for its acceptance. During the work students – '**Subarna Das**' was found to be sincere, regular and hardworking and have successfully completed the thesis work assigned to him.

________________	________________
(HOD/TIC, CSE)	Prof. Jaydeb Mondal
The Neotia University	The Neotia University

Declaration of Originality and Compliance of Academic Efforts

I hereby declare that this report contains literature survey and original research work done by the under signed candidates, as part of our “**Bachelor in Technology Studies**”.
All information in this document have been obtained and presented in accordance with academic rules and ethical conduct.
I also declare that, as required by these rules and conduct, I have cited and referenced all materials that are not original to this work.

Name: Subhadip Samanta

Roll No.: TNU20220021100001

Project Title: **Object Detection**

Signature with Date:

Declaration of Originality and Compliance of Academic Efforts

I hereby declare that this report contains literature survey and original research work done by the under signed candidates, as part of our "**Bachelor in Technology Studies**".
All information in this document have been obtained and presented in accordance with academic rules and ethical conduct.
I also declare that, as required by these rules and conduct, I have cited and referenced all materials that are not original to this work.

Name: Wrishav Sett

Roll No.: TNU20220021100004

Project Title: **Object Detection**

Signature with Date:

Declaration of Originality and Compliance of Academic Efforts

I hereby declare that this report contains literature survey and original research work done by the under signed candidates, as part of our "**Bachelor in Technology Studies**".
All information in this document have been obtained and presented in accordance with academic rules and ethical conduct.
I also declare that, as required by these rules and conduct, I have cited and referenced all materials that are not original to this work.

Name: Sudipta Saha

Roll No.: TNU20220021100005

Project Title: **Object Detection**

Signature with Date:

Declaration of Originality and Compliance of Academic Efforts

I hereby declare that this report contains literature survey and original research work done by the under signed candidates, as part of our "**Bachelor in Technology Studies**".
All information in this document have been obtained and presented in accordance with academic rules and ethical conduct.
I also declare that, as required by these rules and conduct, I have cited and referenced all materials that are not original to this work.

Name: Subarna Das

Roll No.: TNU2020053100011

Project Title: **Object Detection**

Signature with Date:

Contents

Introduction

Object detection is a computer vision task that involves identifying and localizing objects within digital images or video frames. It is a fundamental problem in computer vision and finds applications in various domains, including autonomous driving, surveillance, robotics, and image analysis.

The goal of object detection is to detect and classify multiple objects of interest within an image or video and provide their precise spatial locations. It goes beyond simple image classification, which only determines the presence of objects in an image. Object detection provides more detailed information by not only recognizing the objects but also creating bounding boxes around them to indicate their positions.

Traditionally, object detection involved manually engineering features and designing algorithms to detect objects. However, in recent years, deep learning techniques, particularly convolutional neural networks (CNNs), have revolutionized object detection. These deep learning models can automatically learn and extract relevant features from images, enabling accurate and efficient object detection.

Modern object detection frameworks, such as Faster R-CNN, YOLO (You Only Look Once), and SSD (Single Shot MultiBox Detector), have achieved remarkable performance by combining deep learning models with advanced techniques like region proposals, anchor boxes, and feature pyramid networks.

The process of object detection typically involves the following steps:

1. Input Image/Video: *Object detection begins with an input image or a sequence of video frames.*

2. Preprocessing: *The input image is preprocessed by resizing, normalization, and data augmentation techniques to enhance the model's performance and robustness.*

3. Feature Extraction: *A deep learning model, typically a CNN, is used to extract meaningful features from the preprocessed image. The model is usually pre-trained on a large dataset (e.g., ImageNet) and fine-tuned for object detection.*

4. Object Localization: *The extracted features are passed through additional layers to predict the presence of objects and their bounding box coordinates. This step*

involves identifying potential object locations and refining the bounding box predictions.

5. Object Classification: *The extracted features within each bounding box are classified into different object categories (e.g., car, person, dog) using classification layers.*

6. Post-processing: *The predicted bounding boxes and their corresponding class labels are post-processed to eliminate duplicate detections, filter out low-confidence detections, and improve the overall accuracy.*

7. Output: *The final output of the object detection process includes the detected objects' bounding boxes, class labels, and confidence scores.*

Object detection has witnessed significant advancements, thanks to the availability of large annotated datasets, powerful hardware, and advanced deep learning architectures. These advancements have made object detection an essential tool for various real-world applications, empowering machines to understand and interact with the visual world.

Objective and Scope of the Project

2.1 The primary objective of object detection is to accurately identify and locate objects within digital images or video frames. It involves the following specific objectives:

1. Object Recognition: *Object detection aims to recognize and classify different objects within an image or video. This involves assigning appropriate class labels to the detected objects, such as cars, pedestrians, animals, or specific objects of interest.*

2. Localization: *Object detection provides precise spatial localization of objects by creating bounding boxes that enclose the objects within the image or video. These bounding boxes indicate the position and extent of each detected object, allowing for further analysis and understanding.*

3. Multiple Object Detection: *Object detection algorithms should be capable of detecting multiple objects within a single image or video frame. It involves accurately identifying and localizing all instances of objects present, regardless of their size, shape, or orientation.*

4. Real-time Performance: *In many applications, real-time or near-real-time object detection is required, especially in scenarios where immediate response is crucial, such as autonomous driving or video surveillance. The objective is to achieve efficient and fast detection without sacrificing accuracy.*

5. Robustness and Accuracy: *Object detection algorithms should be robust to various challenges, including occlusion (partial or full obstruction of objects), varying lighting conditions, cluttered backgrounds, and different object scales. The objective is to achieve high accuracy and reliable detection performance under diverse real-world conditions.*

6. Generalization: *Object detection models should have the ability to generalize well to unseen objects or objects from different categories. The objective is to build models that can detect a wide range of objects with high accuracy, even if they were not present in the training dataset.*

7. Efficiency: *Object detection algorithms should strive for computational efficiency, optimizing the trade-off between accuracy and computational resources. This is particularly important for applications with limited processing power or resource-constrained devices.*

By achieving these objectives, object detection enables various applications such as autonomous vehicles, surveillance systems, object tracking, augmented reality, robotics, and many more, where understanding and interacting with the visual environment are essential.

2.2 The scope of object detection is vast and encompasses a wide range of applications and research areas. Here are some key areas where object detection plays a significant role:

1. Autonomous Driving: *Object detection is crucial for autonomous vehicles to perceive and understand the surrounding environment. It helps in detecting and tracking vehicles, pedestrians, traffic signs, traffic lights, and other objects, enabling safe and efficient navigation on roads.*

2. Surveillance and Security: *Object detection is extensively used in surveillance systems for detecting and tracking suspicious activities, intruders, and objects of interest in public spaces, airports, buildings, and other security-sensitive areas.*

3. Robotics: *Object detection enables robots to perceive and interact with the physical world. It is used for tasks such as object manipulation, pick-and-place operations, object recognition, and scene understanding, making robots more versatile and capable in various domains.*

4. Image and Video Analysis: *Object detection is utilized in image and video analysis applications, such as content-based image retrieval, video summarization, video annotation, and object recognition in multimedia databases.*

5. Medical Imaging: *Object detection finds applications in medical imaging for tasks like tumor detection, organ segmentation, anatomical structure localization, and analysis of medical images for diagnosis and treatment planning.*

6. Augmented Reality: *Object detection is essential for augmented reality (AR) applications, where virtual objects or information are overlaid onto the real world. Object detection allows for accurate registration of virtual objects with the real-world environment.*

7. Retail and E-commerce: *Object detection is used in retail and e-commerce for various purposes, including product recognition, inventory management, shelf analysis, and tracking customer behavior within stores.*

8. Human-Computer Interaction: *Object detection is employed in human-computer interaction systems for gesture recognition, face detection, hand tracking, and pose estimation, enabling natural and intuitive interactions between humans and computers.*

9. Environmental Monitoring: *Object detection can assist in environmental monitoring by detecting and tracking objects like wildlife, vegetation, and land use patterns, aiding in biodiversity assessment, ecological studies, and environmental management.*

10. Industrial Automation: *Object detection is utilized in industrial automation for tasks like quality control, defect detection, robotic assembly, object sorting, and process optimization.*

The scope of object detection continues to expand as new applications and research areas emerge. With advancements in deep learning, sensor technologies, and computational capabilities, object detection is poised to play an increasingly crucial role in various domains, enabling machines to perceive and understand the visual world.

Definition of Problem

3.1 Object Recognition can be accomplished through various means using a variety of algorithms that are already available on popular websites such as https://github.com/ and https://www.kaggle.com/ which only need to be modified to the needs and requirements according to the specific individual or purpose.

Few such algorithms are:

1. Scale-Invariant Feature Transform (SIFT): *SIFT is a widely used algorithm for detecting and describing local features in images. It is robust to changes in scale, rotation, and lighting conditions, making it suitable for object recognition tasks.*

2. Speeded-Up Robust Features (SURF): *SURF is another feature-based algorithm that identifies and matches keypoints in images. It is faster than SIFT and also handles scale and rotation variations effectively.*

3. Histogram of Oriented Gradients (HOG): *HOG computes and analyzes the distribution of gradient orientations in an image. It is commonly used for detecting and recognizing objects, especially for pedestrian detection in computer vision tasks.*

4. Convolutional Neural Networks (CNN): *CNNs have revolutionized object recognition and achieved state-of-the-art performance. These deep learning models automatically learn hierarchical features from data, enabling accurate object recognition. Popular CNN architectures for object recognition include AlexNet, VGGNet, GoogLeNet, ResNet, and EfficientNet.*

5. Region-Based CNNs (R-CNN): *R-CNN and its variants (Fast R-CNN, Faster R-CNN) combine object proposal methods (such as selective search) with CNNs for object recognition. They first generate region proposals and then classify and refine them using CNNs, achieving accurate object detection and recognition.*

6. You Only Look Once (YOLO): *YOLO is a real-time object detection algorithm that directly predicts bounding boxes and class probabilities from an image using a single pass of a CNN. It achieves a good balance between accuracy and speed, making it suitable for real-time applications.*

7. <u>Single Shot MultiBox Detector (SSD)</u>: *SSD is another real-time object detection algorithm that uses a series of convolutional layers to predict object classes and bounding boxes at multiple scales. It provides accurate detection across different object sizes and achieves real-time performance.*

8. <u>MobileNet</u>: *MobileNet is a lightweight CNN architecture specifically designed for resource-constrained devices, such as mobile phones and embedded systems. It provides a good balance between accuracy and model size, making it suitable for object recognition on devices with limited computational resources.*

These are just a few examples of object recognition algorithms. The field of computer vision is continually evolving, and researchers are developing new algorithms and techniques to improve object recognition accuracy, speed, and robustness.

3.2 The speed and accuracy of object recognition algorithms can vary depending on various factors, such as the specific implementation, hardware resources, dataset characteristics, and the complexity of the objects being recognized. It is important to note that speed and accuracy trade-offs exist, and achieving high accuracy often comes at the cost of increased computation time. Here is a general overview of the speed and accuracy characteristics of the mentioned algorithms:

1. <u>**SIFT**</u>:
 - <u>Speed</u>: *SIFT is relatively slower compared to some newer algorithms due to its computational complexity.*
 - <u>Accuracy</u>: *SIFT is known for its robustness and accuracy in detecting and matching keypoints, making it suitable for object recognition tasks. However, it may struggle with large-scale variations and occlusions.*

2. <u>**SURF**</u>:
 - <u>Speed</u>: *SURF is faster than SIFT due to its efficient approximation of scale space and integral image computations.*
 - <u>Accuracy</u>: *SURF provides good accuracy and robustness against scale and rotation variations, making it suitable for real-time applications.*

3. **HOG**:
- Speed: *HOG is known for its computational efficiency, making it suitable for real-time applications.*
- Accuracy: *HOG achieves good accuracy in detecting objects based on local gradient orientations, particularly in scenarios like pedestrian detection.*

4. **CNNs**:
- Speed: *CNNs can vary in speed depending on the specific architecture, model size, and hardware resources. Smaller and more optimized architectures like MobileNet tend to be faster, while larger architectures may have longer inference times.*
- Accuracy: *CNNs, especially deep architectures like VGGNet, GoogLeNet, and ResNet, have achieved state-of-the-art accuracy in object recognition tasks but can be computationally expensive.*

5. **R-CNN and its variants (Fast R-CNN, Faster R-CNN)**:
- Speed: *R-CNN and its variants are slower compared to some other algorithms due to their multi-stage processing pipeline that involves region proposals and subsequent classification and refinement steps.*
- Accuracy: *R-CNN variants achieve high accuracy in object detection and recognition tasks, often surpassing previous state-of-the-art methods.*

6. **YOLO**:
- Speed: *YOLO is known for its real-time performance, offering high speed in object detection. It can achieve several frames per second, depending on the specific implementation and hardware resources.*
- Accuracy: *YOLO provides a good balance between speed and accuracy, although it may sacrifice some accuracy compared to more computationally expensive algorithms.*

7. **SSD**:
- Speed: *SSD achieves real-time performance and can process frames rapidly, depending on the model size and hardware resources.*
- Accuracy: *SSD provides accurate object detection across various scales and achieves a good trade-off between speed and accuracy.*

It is important to note that the reported speed and accuracy can vary depending on the specific implementation, optimization techniques, hardware acceleration (e.g., GPUs, TPUs), and dataset characteristics. Additionally, different object recognition tasks and datasets may have specific requirements and constraints that affect the performance of the algorithms. It is recommended to benchmark and evaluate the algorithms on specific datasets and scenarios to obtain more accurate insights into their speed and accuracy performance.

3.3 We had taken to SSD and YOLO as the algorithms to work on and we decided to use the YOLO algorithm to complete our object detection project.

There are several reasons why we chose YOLO (You Only Look Once) over SSD (Single Shot MultiBox Detector) for object detection:

1. <u>Speed</u>: *YOLO is known for its fast inference speed. It performs detection in a single pass through the network, making it highly efficient. YOLO's design allows it to achieve real-time or near real-time performance, making it suitable for applications that require fast object detection, such as video analysis and real-time systems.*

2. <u>Simplicity</u>: *YOLO has a simpler architecture compared to SSD. It uses a single convolutional neural network to directly predict bounding box coordinates and class probabilities. This simplicity makes YOLO easier to implement and understand, making it a popular choice for researchers and practitioners new to object detection.*

3. <u>Detection of Small Objects</u>: *YOLO performs well in detecting small objects in an image. It uses a grid-based approach to divide the image into regions and predicts bounding boxes and class probabilities at each grid cell. This enables YOLO to handle small objects better than SSD, which relies on anchor boxes at different scales.*

4. <u>Object Localization</u>: *YOLO provides precise object localization. It predicts bounding box coordinates directly, allowing for accurate localization of objects. This can be advantageous in tasks where precise object boundaries are important, such as medical imaging or object tracking.*

5. Overall Performance: *YOLO has shown competitive performance in terms of accuracy and mAP (mean Average Precision) on various object detection benchmarks. While SSD may offer slightly higher accuracy in some cases, YOLO provides a good balance between speed and accuracy, making it a popular choice in applications where real-time performance is a priority.*

For our project we have chosen to delve into the subject of object detection using the **YOLO (You Only Live Once) algorithm**, more specifically the new and improved **YOLOv3 algorithm**.

3.4 What is YOLO object detection?

YOLO (You Only Look Once) is an object detection algorithm that revolutionized the field of computer vision by providing real-time object detection with high accuracy. The YOLO algorithm aims to identify and locate objects within an image or video frame.

The key idea behind YOLO is to frame object detection as a regression problem, where the algorithm directly predicts the bounding boxes and class probabilities for multiple objects in a single pass of the neural network. This is in contrast to traditional object detection approaches that involve two stages: region proposal generation and subsequent classification.

The YOLO algorithm divides the input image into a grid, and each grid cell is responsible for predicting the bounding boxes and associated class probabilities for the objects it contains. The algorithm predicts bounding boxes using predefined anchor boxes of different sizes and aspect ratios, and then adjusts them based on the characteristics of the objects present in the grid cell. Each bounding box is associated with a class probability, indicating the likelihood of the object belonging to a particular class.

During training, YOLO optimizes a loss function that considers both localization errors (differences between predicted and ground-truth bounding boxes) and classification errors (differences between predicted and ground-truth class probabilities). This enables the algorithm to learn to accurately detect and classify objects in various contexts.

YOLO has undergone several iterations, with YOLOv3 and YOLOv4 being the most widely adopted versions. Each iteration has introduced architectural improvements and optimizations to enhance accuracy and speed. YOLOv4, in particular, incorporates advanced techniques such as feature pyramid networks, spatial attention modules, and darknet-53 as its backbone network, further improving object detection performance.

YOLO-based algorithms are highly regarded for their ability to achieve real-time object detection on various platforms, including embedded devices and in resource-constrained environments. They have found applications in diverse fields, including autonomous driving, surveillance systems, robotics, and more, where real-time and accurate object detection is crucial.

3.5 What is YOLOv3 for YOLO object detection?

YOLOv3 (You Only Look Once version 3) is an improved and more powerful variant of the YOLO object detection algorithm. It builds upon the success of its predecessors, YOLO and YOLOv2 (also known as YOLO9000), by introducing architectural enhancements and optimization techniques to further improve object detection accuracy.

Here are some key features and improvements in YOLOv3:

Multi-scale detection: *YOLOv3 operates at three different scales (referred to as "anchors" or "strides") to detect objects of varying sizes. This allows the algorithm to effectively detect small, medium, and large objects within an image.*

Feature pyramid networks: *YOLOv3 incorporates feature pyramid networks (FPN) to capture object information at multiple scales. FPN enables the algorithm to extract features from different layers of the neural network, allowing it to detect objects of varying sizes with more accuracy.*

Darknet-53 backbone: *YOLOv3 employs a deeper backbone network called Darknet-53. Darknet-53 is a variant of the Darknet architecture, which consists of 53 convolutional layers. It is designed to extract more complex and abstract features from images, leading to improved object detection performance.*

Improved bounding box predictions: *YOLOv3 utilizes a technique called "IoU (Intersection over Union) prediction" to improve bounding box predictions. By predicting the IoU between the predicted bounding box and ground truth box, YOLOv3 can more accurately estimate the quality of the predicted boxes.*

Class-specific confidence scores: *YOLOv3 introduces class-specific confidence scores, which measure the confidence of an object detection prediction for each specific class. This allows the algorithm to better differentiate between different classes and improve the accuracy of object classification.*

Better training strategies: *YOLOv3 incorporates various training strategies, such as multi-scale training, data augmentation, and warm-up training, to enhance the model's robustness and generalization capability.*

Overall, YOLOv3 offers a significant improvement in object detection accuracy while maintaining real-time performance. It has become a popular choice for various computer vision applications due to its ability to detect and classify objects in real-time, making it suitable for applications such as video surveillance, autonomous driving, and object tracking.

We have used the YOLOv3 algorithm for identifying objects and create bounding boxes and label them according to their respective categories for still images and in real-time through camera input.

Literature Review

4.1 Literature Review: Object Detection

Introduction:

Object detection is a crucial task in computer vision that involves locating and classifying objects within images or videos. Over the years, extensive research has been conducted in this field, leading to significant advancements in object detection methodologies. This literature review provides an in-depth overview of the key approaches and breakthroughs in object detection, highlighting both traditional and deep learning-based techniques.

1. Viola-Jones Algorithm:

The Viola-Jones algorithm, proposed by Viola and Jones in 2001, marked a significant milestone in object detection. It utilized Haar-like features and the AdaBoost algorithm to achieve real-time face detection. The algorithm's effectiveness lay in its ability to rapidly analyze image sub-windows and classify them as either face or non-face regions. While primarily limited to face detection, the Viola-Jones algorithm paved the way for subsequent research in object detection.

2. Histogram of Oriented Gradients (HOG):

Dalal and Triggs introduced the Histogram of Oriented Gradients (HOG) method in 2005, which gained popularity for its effectiveness in pedestrian detection. HOG computes histograms of gradient orientations within local image regions and employs a linear SVM for classification. This approach demonstrated robustness against variations in lighting conditions and viewpoint changes, making it suitable for various applications.

3. Scale-Invariant Feature Transform (SIFT):

SIFT, proposed by Lowe in 1999, is a widely used keypoint-based method for object detection, recognition, and tracking. SIFT features are invariant to scale, rotation, and affine transformations, enabling reliable matching and localization. The SIFT algorithm detects stable keypoints, extracts descriptors around these keypoints, and matches them across different images. SIFT has proven to be effective in handling image transformations and has been applied in numerous object detection tasks.

4. Deformable Part Models (DPM):
Felzenszwalb et al. introduced Deformable Part Models (DPMs) in 2008, presenting a powerful framework for object detection. DPMs represent objects as a collection of deformable parts and their spatial relations. These models incorporate appearance models learned from training data, as well as spatial models to handle object deformations and occlusions. DPM-based detectors have shown promising results in various domains, including pedestrian and car detection.

5. Deep Learning-based Approaches:
The emergence of deep learning and Convolutional Neural Networks (CNNs) has revolutionized the field of object detection, leading to substantial improvements in accuracy.

a. ***R-CNN:***
R-CNN (Region-based Convolutional Neural Network), proposed by Girshick et al. in 2014, introduced the concept of region proposals for object detection. It utilized selective search to generate region proposals and applied a CNN to extract features from each proposal. These features were then classified using SVMs. R-CNN achieved significant improvements in accuracy over traditional methods but suffered from slow training and testing speeds.

b. ***Fast R-CNN:***
To address the speed limitations of R-CNN, Girshick et al. introduced Fast R-CNN in 2015. It introduced a region of interest (RoI) pooling layer that enabled the extraction of region-based features from the entire image in a single forward pass. Fast R-CNN improved training and testing speeds while maintaining high detection accuracy.

c. ***Faster R-CNN:***
Building upon Fast R-CNN, Ren et al. proposed Faster R-CNN in 2015, which eliminated the need for external region proposal methods. It introduced a Region Proposal Network (RPN) that shared convolutional layers with the detection network, enabling end-to-end training. The integration of the RPN improved both the speed and accuracy of object detection.

d. ***Mask R-CNN:***

He et al. extended the Faster R-CNN framework to include instance segmentation in 2017 with Mask R-CNN. In addition to predicting bounding boxes and class probabilities, Mask R-CNN added an extra branch to generate pixel-level masks for each object instance. Mask R-CNN achieved state-of-the-art results in both object detection and instance segmentation tasks.

6. <u>Single Shot MultiBox Detector (SSD):</u>

Liu et al. proposed the Single Shot MultiBox Detector (SSD) in 2016, aiming for real-time object detection without requiring region proposal methods. SSD utilized a series of convolutional layers at different scales to predict object bounding boxes and class probabilities directly from the input image. By leveraging feature maps at multiple scales, SSD achieved a good balance between speed and accuracy.

7. <u>You Only Look Once (YOLO):</u>

The You Only Look Once (YOLO) approach, introduced by Redmon et al. in 2016, presented an alternative perspective on object detection. YOLO treated object detection as a regression problem, predicting bounding boxes and class probabilities directly from the entire image using a single neural network. YOLO achieved impressive real-time performance, although it faced challenges in accurately detecting small objects.

8. <u>Two-Stage Detectors:</u>

While one-stage detectors like YOLO and SSD offered real-time performance, two-stage detectors further improved accuracy at the cost of increased computational complexity.

a. ***Mask R-CNN:***

As previously mentioned, Mask R-CNN extended the Faster R-CNN framework to include instance segmentation, achieving state-of-the-art performance in both object detection and instance segmentation.

9. EfficientDet:

Proposed by Tan et al. in 2019, EfficientDet aimed to strike a balance between accuracy and efficiency in object detection. It employed a compound scaling method to optimize the depth, width, and resolution of the detection network. EfficientDet achieved state-of-the-art performance on popular object detection benchmarks while being computationally efficient.

10. Transformers in Object Detection:

Transformers, originally popular in natural language processing, have recently gained attention in object detection tasks.

a. ***DETR:***

Carion et al. introduced the DETR (DEtection TRansformer) framework in 2020, which eliminated the need for region proposal methods. DETR utilized self-attention mechanisms to directly predict object bounding boxes and class labels from the entire image. By leveraging transformers, DETR provided a new perspective on object detection and achieved promising results.

Conclusion:

Object detection has witnessed significant advancements, from traditional methods like Viola-Jones, HOG, and SIFT to deep learning-based approaches such as R-CNN, Fast R-CNN, Faster R-CNN, and Mask R-CNN. One-stage detectors like YOLO and SSD offered real-time performance, while two-stage detectors further improved accuracy. Recent developments explored the use of transformers, exemplified by DETR. Object detection continues to evolve, enabling a wide range of applications in computer vision.

4.2 Literature Review: YOLOv3 - An Incremental Improvement

Introduction:
Object detection is a fundamental task in computer vision that involves identifying and localizing objects within images or videos. YOLO (You Only Look Once) has emerged as a popular real-time object detection algorithm due to its impressive speed and accuracy. YOLOv3, introduced by Joseph Redmon and Ali Farhadi in 2018, represents a significant advancement over its predecessors. This literature review delves into the key research papers on YOLOv3, providing a comprehensive understanding of its architecture, training strategies, and performance improvements.

Architecture and Innovations:
YOLOv3 builds upon the strengths of its predecessors, YOLO and YOLOv2, while introducing several key innovations. One notable enhancement is the integration of a Darknet-53 backbone network. With 53 convolutional layers, the Darknet-53 backbone allows YOLOv3 to capture richer and more complex features, improving its ability to represent objects accurately. This deeper network architecture enhances the discriminative power of YOLOv3 and contributes to improved detection performance.
Another significant innovation in YOLOv3 is the adoption of feature pyramid networks (FPN). FPN enables multi-scale feature fusion, facilitating the detection of objects at different sizes. By incorporating feature maps from multiple layers with varying resolutions, YOLOv3 can effectively handle objects of different scales and improve its detection accuracy across the entire object spectrum. The introduction of FPN enhances the representation capability of YOLOv3 and addresses the challenge of detecting objects at various scales within a single unified architecture.

Training Enhancements:
YOLOv3 introduces several training enhancements to further improve its accuracy. One key enhancement is an increase in the number of anchor boxes used for object detection. Anchor boxes serve as reference templates, enabling the model to detect objects of various sizes and aspect ratios more effectively. By increasing the number of anchor boxes per scale, YOLOv3 can better handle object scale variations, resulting in improved localization and recognition accuracy.
Moreover, YOLOv3 adopts a three-scale detection approach. Instead of predicting objects at a single scale, it performs detection at three different resolutions. This

multi-scale approach allows YOLOv3 to capture objects at varying sizes and further enhances its ability to detect objects accurately across different scales. By considering objects at multiple resolutions simultaneously, YOLOv3 achieves better coverage of the object space and improves its overall detection performance.

Performance Evaluation:

The performance of YOLOv3 has been extensively evaluated and compared with other state-of-the-art object detection algorithms. On benchmark datasets such as COCO (Common Objects in Context), YOLOv3 demonstrates competitive accuracy, achieving high mean Average Precision (mAP) scores. It performs well across various object categories and excels in detecting objects in real-time scenarios.

One of the key advantages of YOLOv3 is its ability to strike a balance between accuracy and speed. While achieving competitive accuracy, YOLOv3 retains its real-time performance, making it highly suitable for applications that require both speed and accuracy, such as video analysis, robotics, and autonomous driving. YOLOv3 demonstrates remarkable object detection capabilities even on resource-constrained devices, making it a practical choice for deployment on embedded systems and mobile platforms.

Further Developments and Variants:

Building upon YOLOv3, subsequent research has focused on further improving its performance. For instance, YOLOv4 introduced a modified backbone network known as CSPDarknet53, which improves the efficiency and effectiveness of feature extraction. It also incorporated techniques like mosaic data augmentation and CIOU loss to enhance accuracy and robustness.

Other variants of YOLO, such as YOLOv4-tiny and YOLOv5, have been developed to address specific requirements. YOLOv4-tiny aims to reduce the model size and computational complexity while maintaining reasonable accuracy. YOLOv5, introduced by Ultralytics, focuses on optimizing the architecture for faster inference and achieving competitive performance with improved efficiency.

Conclusion:

YOLOv3 represents a significant advancement in real-time object detection. Its architectural innovations, including the integration of a Darknet-53 backbone network and the utilization of feature pyramid networks, contribute to improved accuracy and robustness. The training enhancements, such as increased anchor boxes and multi-scale detection, further enhance its detection capabilities. The

balance between accuracy and speed has made YOLOv3 a popular choice for a wide range of computer vision applications.
The subsequent developments and variants of YOLOv3 continue to push the boundaries of object detection performance. Researchers are actively exploring architectural variations, training strategies, and optimization techniques to achieve even higher accuracy and efficiency. The impact and relevance of YOLOv3 in the field of computer vision are evident, and it remains a widely used and influential algorithm for real-time object detection.

We have further studied papers written and published by lead researchers to gather a more in-depth knowledge about the working of YOLO and by extension YOLOv3.

Theoretical Background

Theoretical background in object detection projects typically involves understanding and applying concepts from computer vision, machine learning, and deep learning. Here are some key theoretical components relevant to object detection projects:

1. Image Representation: *Understanding how images are represented digitally is essential. This includes concepts such as pixels, color spaces (RGB, HSV, etc.), image transformations, and image enhancement techniques.*

2. Feature Extraction: *Extracting meaningful features from images plays a crucial role in object detection. Techniques such as edge detection, corner detection, scale-invariant feature transform (SIFT), histogram of oriented gradients (HOG), and deep features (extracted from pre-trained convolutional neural networks) can be used to represent objects.*

3. Supervised Learning: *Object detection often relies on supervised learning techniques, where a model learns from labeled training data. Algorithms such as Support Vector Machines (SVM), decision trees, random forests, and gradient boosting can be used for classification and regression tasks in object detection.*

4. Convolutional Neural Networks (CNNs): *CNNs have revolutionized object detection due to their ability to automatically learn hierarchical features from raw image data. Architectures like LeNet, AlexNet, VGGNet, ResNet, and more recently, EfficientNet, have been successful in object detection tasks.*

5. Deep Neural Networks: *DNNs use backpropagation and optimization for learning from raw inputs in a much more efficient format. Darknet is one such open-source neural network framework primarily used for training deep neural networks (DNNs). It was developed by Joseph Redmon and serves as the backbone for the popular object detection system known as YOLO (You Only Look Once).*

5. Object Localization: *Object localization refers to identifying the location of an object within an image. Techniques like sliding window, region proposal methods (e.g., Selective Search, EdgeBoxes), and anchor-based methods (e.g., Faster R-CNN, RetinaNet) are commonly used for localizing objects.*

6. Object Classification: *Once an object is localized, classification algorithms are applied to determine the category or class label of the object. This can involve techniques like softmax regression, multi-layer perceptron (MLP), or using the output layer of a pre-trained CNN as a feature extractor and applying a classifier.*

7. Object Detection Architectures: *Modern object detection architectures, such as YOLO (You Only Look Once), SSD (Single Shot MultiBox Detector), and Faster R-CNN (Region-based Convolutional Neural Networks), combine object localization and classification in an end-to-end framework. These architectures aim to achieve both accurate and efficient object detection.*

8. Evaluation Metrics: *To assess the performance of object detection models, evaluation metrics such as precision, recall, intersection over union (IoU), average precision (AP), and mean Average Precision (mAP)*
are commonly used. These metrics help quantify the accuracy, completeness, and robustness of the object detection system.

Understanding these theoretical concepts provides a foundation for designing and implementing effective object detection algorithms and models. It's important to stay updated with the latest research and advancements in the field, as object detection is an active area of research with ongoing developments and improvements on an almost daily basis, which either improves an algorithms efficiency or accuracy or functionality and support.

Methodology

6.1 YOLOv3 (You Only Look Once version 3) is an object detection algorithm that aims to efficiently and accurately detect objects in images. The methodology behind YOLOv3:

1: Multi-scale detection: *YOLOv3 uses a multi-scale approach by employing three different scales (small, medium, and large) of feature maps. This allows it to detect objects of various sizes and achieve better accuracy across the entire image.*

2: Input Image: *The algorithm takes an input image and divides it into a grid of cells.*

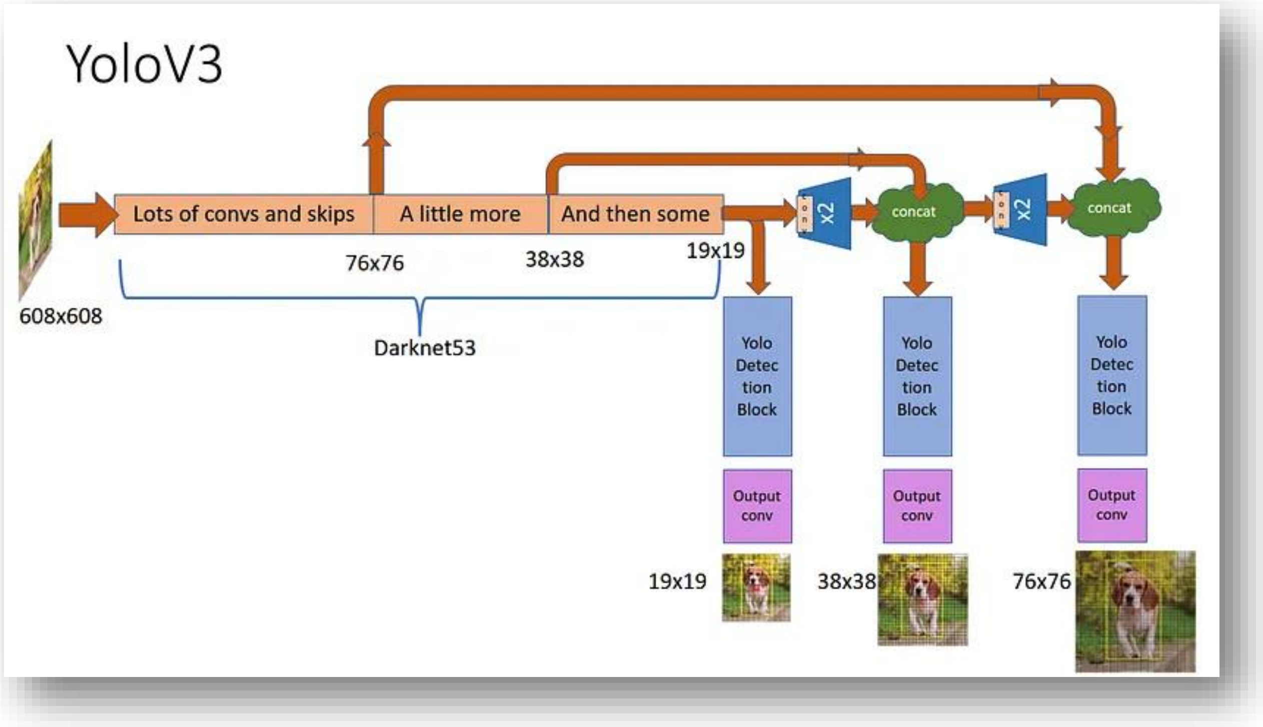
3: Feature extraction: *It uses a deep neural network architecture, typically based on Darknet (typically Darknet-53), to extract features from the input image. This architecture comprises convolutional layers followed by fully connected layers to capture both local and global context information. This allows YOLOv3 to detect objects of various sizes.*

4: Anchor boxes: *YOLOv3 utilizes anchor boxes to predict bounding boxes for different object sizes and aspect ratios. These anchor boxes are predefined and used to anchor the predictions during training and inference, enabling accurate localization of objects.*

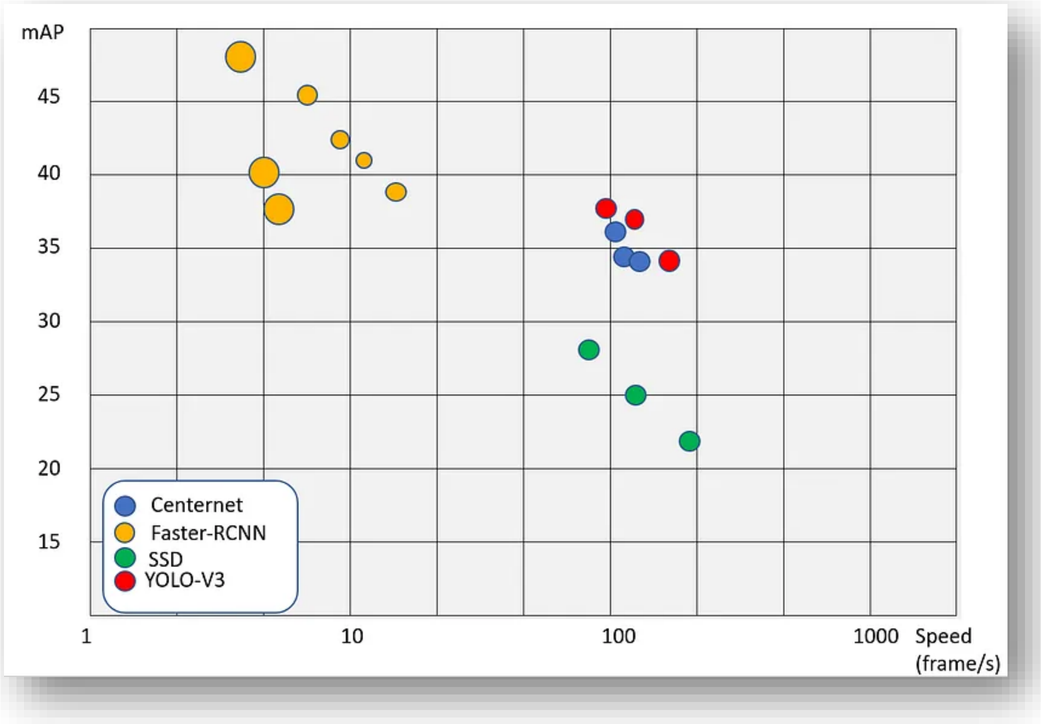
5: Prediction refinement: *YOLOv3 predicts object classes and bounding box coordinates at three different scales. It then refines the predictions by using different detection layers and anchor boxes. This multi-scale prediction approach helps improve the model's ability to detect objects accurately.*
6: Non-maximum suppression: *To eliminate duplicate detections, YOLOv3 applies non-maximum suppression. This post-processing technique filters out redundant bounding boxes, keeping only the most confident predictions for each object.*

7: Output: *The final output of YOLOv3 is the input image with a list of bounding boxes along with their associated class labels and confidence scores.*

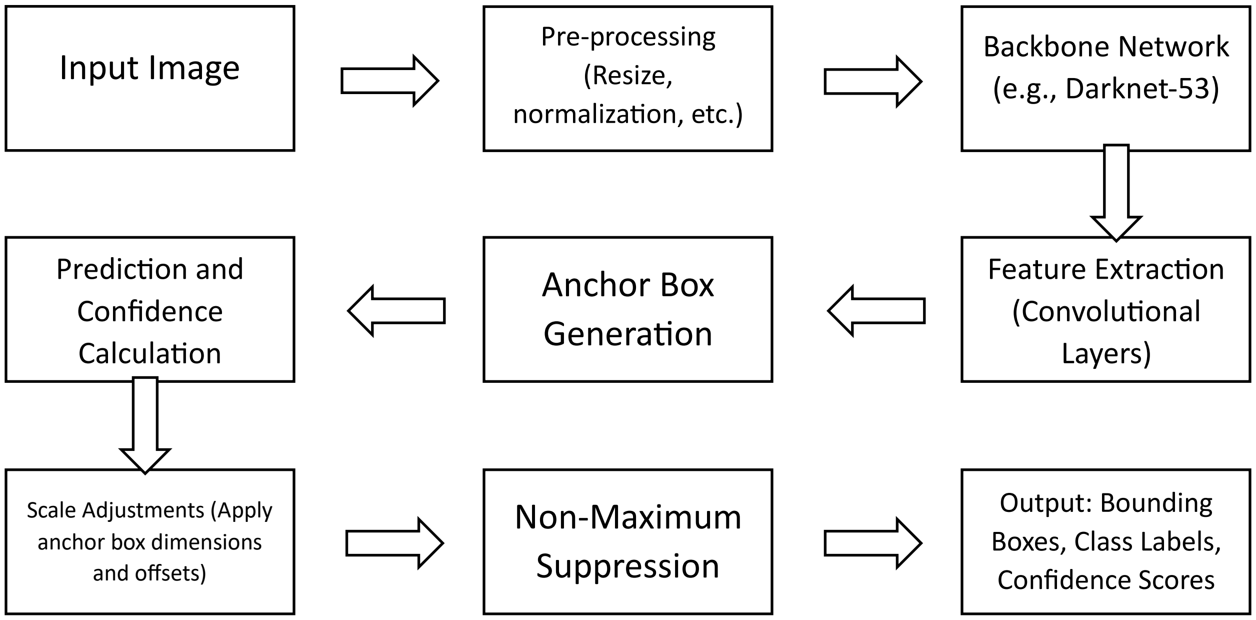
YOLOv3 architecture:



YOLOv3 speed and accuracy:



6.2 Flow Chart depicting the working of a YOLOv3 algorithm:


Input Image
Pre-processing (Resize, normalization, etc.)
Backbone Network (e.g., Darknet-53)
Feature Extraction (Convolutional Layers)
Anchor Box Generation
Prediction and Confidence Calculation
Scale Adjustments (Apply anchor box dimensions and offsets)
Non-Maximum Suppression
Output: Bounding Boxes, Class Labels, Confidence Scores

In the above flowchart, the "**Pre-processing**" step involves common image preprocessing techniques such as resizing the input image and applying normalization to bring pixel values to a suitable range.

The "**Backbone Network**" refers to the underlying architecture of the feature extraction stage, which in the case of YOLOv3 is typically Darknet-53. This architecture consists of multiple convolutional layers, enabling the network to learn hierarchical features at different scales.

The "**Feature Extraction**" step represents the convolutional layers in the backbone network, which extract features from the input image at different spatial resolutions. These features capture semantic information useful for object detection.

The "**Anchor Box Generation**" involves defining a set of anchor boxes with various sizes and aspect ratios that act as reference bounding boxes for object detection. These anchor boxes are usually associated with specific feature map locations.

In the "**Prediction and Confidence Calculation**" step, the network predicts bounding box coordinates relative to each anchor box, as well as class probabilities and confidence scores indicating the presence of objects. These predictions are made at multiple scales and feature maps.

"**Scale Adjustments**" involves scaling the predicted coordinates and adjusting the confidence scores based on the anchor box dimensions and offsets.

Lastly, the "**Non-Maximum Suppression**" step filters out redundant and overlapping detections, keeping only the most confident and non-overlapping bounding boxes. This helps eliminate duplicate detections and improve the final output.

For the above most steps are common for various object detection algorithms such as Image Preprocessing, Feature Extraction, Anchor Box Generation and Non-Maximum Suppression. However, one feature that is exclusive to YOLO is the Backbone Network using Darknet-53.

6.3 What is Darknet-53?

Darknet-53 is a deep convolutional neural network architecture used in the YOLOv3 object detection algorithm. It serves as the backbone network responsible for feature extraction from the input image.

Darknet-53 is named after the open-source deep learning framework called Darknet, developed by Joseph Redmon, the creator of the YOLO (You Only Look Once) algorithm. The number 53 in Darknet-53 refers to **the total number of convolutional layers** in the architecture.

The Darknet-53 architecture is designed to capture features at multiple scales and resolutions, enabling YOLOv3 to detect objects of varying sizes in an image. It follows a similar concept to other popular convolutional neural network architectures, such as VGGNet and ResNet, which stack multiple convolutional layers to progressively learn more abstract features.

Darknet-53 is a relatively lightweight architecture that balances between model complexity and computational efficiency. It does not use residual connections like ResNet but instead relies on deep stacked convolutions to learn features and capture spatial information effectively.

By leveraging Darknet-53 as the backbone network, YOLOv3 can extract rich hierarchical features from the input image, which are then used for subsequent object detection tasks, such as predicting bounding boxes and class probabilities. The features obtained from Darknet-53 are instrumental in enabling YOLOv3 to achieve accurate and efficient real-time object detection.

6.4 How is Darknet used for object detection?

Darknet, the open-source deep learning framework, provides the infrastructure and algorithms for performing object detection using the YOLO (You Only Look Once) methodology.

1: <u>Network Initialization</u>: *Darknet reads the configuration file (yolo.cfg) to define the architecture of the neural network model. It initializes the network structure, including convolutional and connected layers, based on the provided configuration.*

2: <u>Loading Pre-trained Weights</u>: *Darknet loads the pre-trained weights (yolo.weights) into the initialized network. These weights were previously learned during the training phase on a large dataset and stored in the .weights file.*

3: <u>Input Processing</u>: *Darknet takes an input image and applies pre-processing steps such as resizing and normalization to prepare it for input to the neural network model.*

4: <u>Forward Propagation</u>: *The pre-processed input image is passed through the network in a forward pass. Darknet performs a series of convolutions, activations, and other operations defined in the network architecture to extract features from the image.*

5: Object Detection: *At the end of the network, Darknet generates predictions for object detection. These predictions include bounding box coordinates, class probabilities, and confidence scores for each detected object.*

6: Non-Maximum Suppression: *Darknet applies non-maximum suppression to eliminate duplicate and overlapping bounding box detections. It removes boxes with low confidence scores and performs intersection over union (IoU) calculations to merge or discard overlapping boxes, keeping only the most confident and non-overlapping ones.*

7: Output Generation: *The final output of Darknet's object detection process is a list of bounding boxes, along with their associated class labels and confidence scores. These boxes represent the detected objects in the input image.*

Darknet's implementation of YOLO achieves real-time/still object detection by using a single neural network to directly predict bounding boxes and class probabilities. This approach enables efficient and accurate detection of objects in images or videos.

In the context of Darknet, the .cfg and .weights files are important components used for configuring and storing the parameters of the neural network models, including the YOLO (You Only Look Once) object detection models.

.cfg file: *The .cfg file, also known as the configuration file, specifies the architecture and hyperparameters of the neural network model. It defines the structure of the network, including the number and type of layers, the size of the input image, the number of filters in each layer, the activation functions, and other architectural details. The .cfg file provides the blueprint for building the neural network model.*

.weights file: *The .weights file contains the learned weights or parameters of the neural network model. During the training phase, the neural network model is trained on a large dataset to optimize its weights and learn to recognize objects. Once the training process is completed, the learned weights are saved in the .weights file. These weights capture the knowledge acquired by the model during training and are crucial for making accurate predictions during the inference phase.*

In the context of YOLOv3, the YOLOv3 architecture is specified in the yolo.cfg file, which defines the network structure, including the number of convolutional and connected layers, filter sizes, and stride values. The yolo.weights file, on the other hand, contains the pre-trained weights for the YOLOv3 model, which have been learned from a large dataset during the training process.

When running object detection using Darknet and YOLOv3, these .cfg and .weights files are typically provided as inputs. The .cfg file is used to construct the **neural network architecture**, and the .weights file is used to initialize the **network's weights** before making predictions on new input images.

Together, the .cfg and .weights files form the core components that define the architecture and parameterize the YOLOv3 model, enabling it to accurately detect objects in images or videos.

6.5 The YOLOv3 object detection that we have worked on is of three different types.

1: Object Detection by YOLOv3 algorithm using a pretrained dataset for a **still image** with yolov3.cfg and yolov3.weights which are made available for instant use by the developer of YOLO algorithm.

2: Object Detection using YOLOv3 using a pretrained dataset with yolov3.cfg and yolov3.weights for **real-time** object detection using the webcam/camera of the device the code is being run on. Necessary changes need to be made to the original backend for this implementation. Once again the required .cfg and .weights file can be obtained from the official YOLO website.

3: Creating a **custom dataset and training a model** using Transfer Learning[1] to identify objects of our choice/need according to the object types present in the trained dataset. The same .cfg and .weights files are used and then modified according to the model and dataset of our build to achieve our goal.

----

[1]What is Transfer Learning?
Transfer learning is a technique where a pre-trained model, typically trained on a large dataset, is used as a starting point for a related task or a smaller dataset. The pre-trained model's knowledge is transferred to the new task, either by using its learned features directly or fine-tuning the model on the new data. Transfer learning can significantly speed up training and improve performance, especially when training data is limited.

6.6 For any of the above there are a few prerequisites that are needed to be met.

1: Python needs to be installed on the machine that you're developing the code on.

2: Installation of necessary packages:

a: OpenCV - A powerful computer vision library that provides various image and video processing functions.

b: NumPy - A fundamental package for scientific computing with Python, which is used for numerical operations and array manipulation.

c: TensorFlow or PyTorch - Deep learning frameworks that offer pre-trained models and tools for object detection.

d: Matplotlib - A plotting library that can be used to visualize and analyze the results of object detection.

e: Pillow - A Python imaging library that provides support for opening, manipulating, and saving various image file formats.

f: Darknet: The framework on which YOLOv3 is built. You can either install Darknet or use a Python wrapper like "darknetpy" to interact with the Darknet implementation.

g: "darknetpy" (Python wrapper for Darknet): A Python interface for Darknet, which allows you to use Darknet's functionalities in Python.

3: To work with YOLOv3 as we have, further dependencies are needed:

A: YOLOv3 Configuration File (`.cfg`)[1]: This file contains the network architecture and configuration parameters for YOLOv3. It specifies the number of layers, filters, anchor sizes, and other settings.

- Example: `yolov3.cfg`

B: YOLOv3 Weights File (`.weights`)[2]: This file contains the pre-trained weights for the YOLOv3 model. These weights are learned during the training process on large datasets and are used for object detection.

- Example: `yolov3.weights`

C: YOLOv3 Classes File (`.names`)[3]: This file lists the names of the object classes that the YOLOv3 model can detect. Each class name is typically listed on a separate line.

- Example: `coco.names`

D: Darknet Framework[4]: YOLOv3 is implemented on the Darknet framework, so you'll need the Darknet executable or the Darknet source code files to run

YOLOv3. These files include the necessary functions and utilities for running YOLOv3.
- Example: `darknet` executable or the Darknet source code files

----

[1] The yolov3.cfg file can be downloaded from https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg

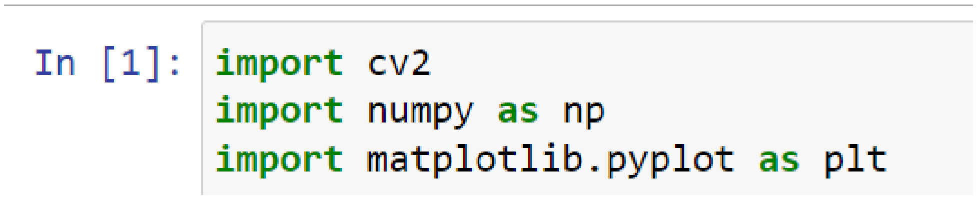
[2] The yolov3.weights file can be downloaded from https://pjreddie.com/media/files/yolov3.weights

[3] The coco.names file can be downloaded from https://github.com/pjreddie/darknet/blob/master/data/coco.names

[4] The Darknet framework can be downloaded from https://github.com/pjreddie/darknet or install it using cmd using 'git clone https://github.com/pjreddie/darknet.git'

6.7 For Object Detection using still image:

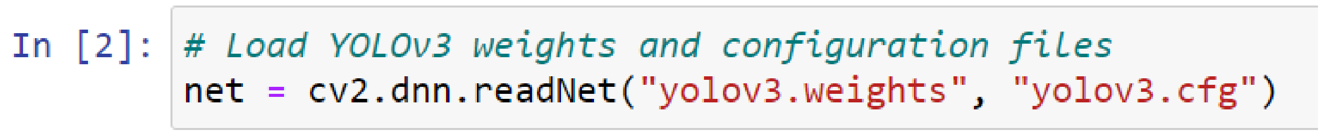
A:



```
In [1]: import cv2
        import numpy as np
        import matplotlib.pyplot as plt
```

The code imports three libraries: cv2 (OpenCV), numpy, and matplotlib.pyplot.
***cv2** is a library for computer vision tasks, including image and video processing.*
***numpy** is a library for numerical computations in Python, providing support for arrays and mathematical operations.*
***matplotlib.pyplot** is a plotting library used for creating visualizations and graphs.*

B:

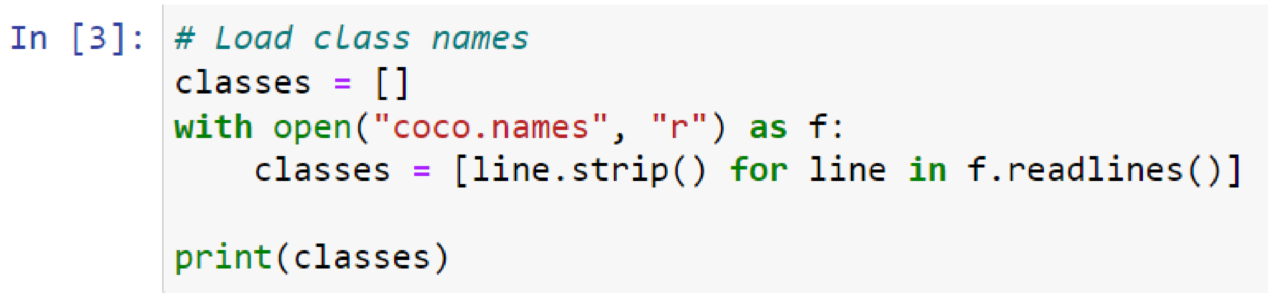


```
In [2]: # Load YOLOv3 weights and configuration files
        net = cv2.dnn.readNet("yolov3.weights", "yolov3.cfg")
```

The code net = cv2.dnn.readNet("yolov3.weights", "yolov3.cfg") reads a pre-trained YOLOv3 (You Only Look Once) model from the specified weights and configuration files.

Here's a breakdown of the function call:
***cv2.dnn** refers to the deep neural network module in OpenCV.*
***readNet()** is a function within the cv2.dnn module used to load a neural network model.*
***"yolov3.weights"** is the filename of the pre-trained weights file for the YOLOv3 model.*
***"yolov3.cfg"** is the filename of the configuration file that defines the architecture and settings of the YOLOv3 model.*
By calling cv2.dnn.readNet(), the function loads the YOLOv3 model from the weights and configuration files and assigns it to the variable net. This allows you to use the loaded model for object detection or other computer vision tasks.

C:



```
In [3]: # Load class names
        classes = []
        with open("coco.names", "r") as f:
            classes = [line.strip() for line in f.readlines()]

        print(classes)
```

The code snippet reads the contents of a file named "coco.names" and stores each line of the file as a string element in the classes list. It then prints the contents of the classes list.

Here's a breakdown of the code:

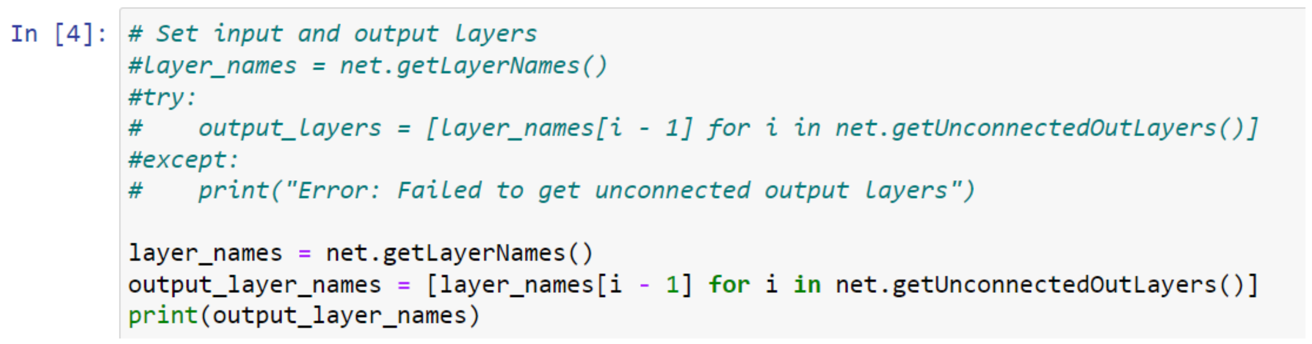
classes = []: *Initializes an empty list called classes to store the names of the classes.*

with open("coco.names", "r") as f:: *Opens the file named "coco.names" in read mode. The file is opened using a context manager, which ensures that the file is properly closed after it has been read.*

classes = [line.strip() for line in f.readlines()]: *Reads all the lines from the file and creates a list comprehension. Each line is stripped of leading and trailing whitespaces using the strip() method, and the resulting string is added to the classes list.*

print(classes): *Prints the contents of the classes list, which now contains the names of classes read from the "coco.names" file.*

D:



```
In [4]: # Set input and output layers
        #layer_names = net.getLayerNames()
        #try:
        #    output_layers = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]
        #except:
        #    print("Error: Failed to get unconnected output layers")

        layer_names = net.getLayerNames()
        output_layer_names = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]
        print(output_layer_names)
```

The code snippet retrieves the names of the output layers of a neural network model loaded in the variable `net` using YOLOv3 architecture.

Here's a breakdown of the code:

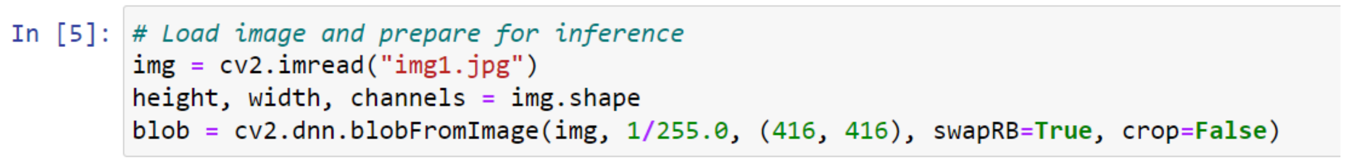
***`layer_names = net.getLayerNames()`:** `net.getLayerNames()` returns a list of all the names of the layers in the neural network model. The code assigns this list to the variable `layer_names`.*

***`output_layer_names = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]`:** `net.getUnconnectedOutLayers()` returns the indices of the output layers of the model. The code then creates a new list called `output_layer_names`, where each element is the name of the corresponding output layer obtained from `layer_names`. The `-1` adjustment in indexing is necessary because layer indices start from 1 in OpenCV, while list indices start from 0.*

***`print(output_layer_names)`:** Prints the names of the output layers. These output layers are typically the final layers of the neural network that produce the predictions or feature maps used for further processing, such as object detection or classification.*

Overall, this code is used to identify and retrieve the names of the output layers in the YOLOv3 model loaded in `net`. The output layer names are often important for extracting the relevant information from the model's output during inference.

E:



```
In [5]: # Load image and prepare for inference
        img = cv2.imread("img1.jpg")
        height, width, channels = img.shape
        blob = cv2.dnn.blobFromImage(img, 1/255.0, (416, 416), swapRB=True, crop=False)
```

The code performs preprocessing steps on an image before feeding it into a neural network for object detection or other computer vision tasks using the YOLOv3 model.

Here's a breakdown of the code:

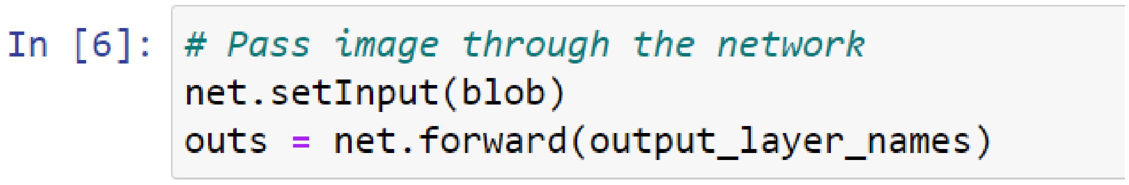
img = cv2.imread("img1.jpg"): *Reads and loads an image file named "img1.jpg" using the cv2.imread() function. The loaded image is stored in the img variable.*

height, width, channels = img.shape: *Obtains the height, width, and number of channels (color channels) of the image using the shape attribute of the img variable. This information is stored in separate variables for later use.*

blob = cv2.dnn.blobFromImage(img, 1/255.0, (416, 416), swapRB=True, crop=False): *Preprocesses the image into a format suitable for inputting into the YOLOv3 model. The cv2.dnn.blobFromImage() function takes the following arguments:*
img: *The input image to be preprocessed.*
1/255.0: *The scaling factor to normalize pixel values in the range of 0 to 1.*
(416, 416): *The size to which the input image is resized.*
swapRB=True: *Specifies whether to swap the color channels from BGR (Blue-Green-Red) to RGB.*
crop=False: *Determines whether to crop the image or pad it with zeros to match the desired input size.*
The function applies the necessary preprocessing operations, such as resizing, normalization, and channel swapping, to convert the image into a blob. A blob is a multi-dimensional array that can be efficiently processed by the neural network.

The resulting blob is assigned to the variable blob and can be passed as input to the YOLOv3 model for object detection or other tasks.
Overall, this code reads an image file, obtains its dimensions, and preprocesses it into a blob format suitable for feeding into the YOLOv3 model.

F:



```
In [6]: # Pass image through the network
        net.setInput(blob)
        outs = net.forward(output_layer_names)
```

The code sets the input of a neural network model loaded in the variable `net` with a preprocessed image blob and performs forward propagation to obtain the output predictions or feature maps from the specified output layers.

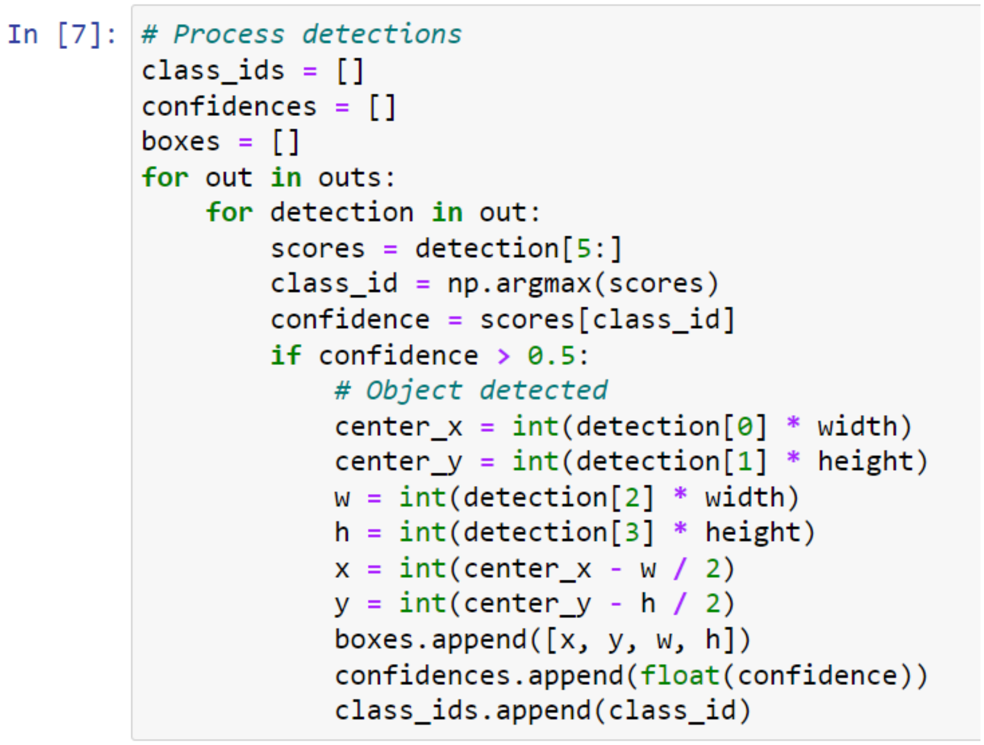
Here's a breakdown of the code:

`net.setInput(blob)`: *Sets the input of the neural network model `net` with the preprocessed image blob (`blob`) obtained from the previous step. This step prepares the model to process the input image.*

`outs = net.forward(output_layer_names)`: *Performs forward propagation on the neural network model `net` using the specified output layer names (`output_layer_names`). This step propagates the input image forward through the model and computes the output predictions or feature maps from the specified output layers.*
The ***`net.forward()`*** *method takes the output layer names as an argument and returns the output values. These output values*
contain the predictions or feature maps generated by the model for the given input image.
The resulting output is assigned to the variable `outs`, which can be further processed and analyzed to obtain the desired results, such as object detection bounding boxes, class probabilities, or other relevant information.

Overall, this code sets the input of the neural network model with a preprocessed image blob and performs forward propagation to obtain the output predictions or feature maps from the specified output layers. This is a crucial step in the inference process of object detection or other computer vision tasks using the YOLOv3 model.

G:



```
In [7]: # Process detections
        class_ids = []
        confidences = []
        boxes = []
        for out in outs:
            for detection in out:
                scores = detection[5:]
                class_id = np.argmax(scores)
                confidence = scores[class_id]
                if confidence > 0.5:
                    # Object detected
                    center_x = int(detection[0] * width)
                    center_y = int(detection[1] * height)
                    w = int(detection[2] * width)
                    h = int(detection[3] * height)
                    x = int(center_x - w / 2)
                    y = int(center_y - h / 2)
                    boxes.append([x, y, w, h])
                    confidences.append(float(confidence))
                    class_ids.append(class_id)
```

The code processes the output of the neural network model to extract the relevant information about the detected objects, including their bounding boxes, class IDs, and confidences.

Here's a breakdown of the code:

`class_ids = []`, `confidences = []`, `boxes = []`: *Initializes three empty lists to store the class IDs, confidences, and bounding boxes of the detected objects.*

`for out in outs:`: *Iterates over the outputs (`outs`) obtained from the neural network model for each specified output layer.*

`for detection in out:`: *Iterates over each detection in the current output.*

`scores = detection[5:]`: *Extracts the confidence scores for each class from the detection output. The first five elements of the `detection` array typically represent the bounding box coordinates and other information.*

`class_id = np.argmax(scores)`: *Determines the class ID with the highest confidence by finding the index of the maximum score in the `scores` array using `np.argmax()`.*

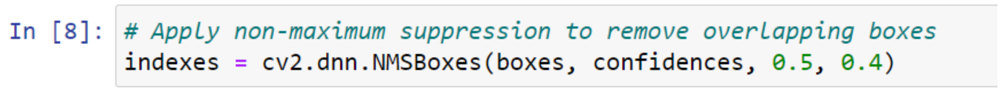
`confidence = scores[class_id]`: *Retrieves the confidence score corresponding to the selected class ID.*

`if confidence > 0.5:`: *Checks if the confidence score is above a certain threshold (0.5 in this case). If the condition is met, it means an object of interest has been detected.*

The following lines calculate and store the information related to the detected object:
- The coordinates of the center ***(`center_x` and `center_y`)*** *of the bounding box relative to the width and height of the input image.*
- The width ***(`w`)*** *and height* ***(`h`)*** *of the bounding box relative to the width and height of the input image.*
- The top-left corner ***(`x` and `y`)*** *of the bounding box by subtracting half of the width and height from the center coordinates.*
- The bounding box coordinates ***(`x`, `y`, `w`, `h`)*** *are appended to the `boxes` list.*
- The confidence score is converted to a float and appended to the `confidences` list.
- The class ID is appended to the ***`class_ids`*** *list.*

Overall, this code processes the output of the neural network model, filters out detections below a certain confidence threshold, and extracts the class IDs, confidences, and bounding box coordinates of the detected objects. This information can be further utilized for tasks such as drawing bounding boxes, labeling objects, or performing additional analysis.

H:



```
In [8]: # Apply non-maximum suppression to remove overlapping boxes
        indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
```

The code applies Non-Maximum Suppression (NMS) to filter out overlapping bounding boxes and keep only the most confident and non-overlapping detections.

Here's a breakdown of the code:

`indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)`: *The `cv2.dnn.NMSBoxes()` function takes the following arguments:*

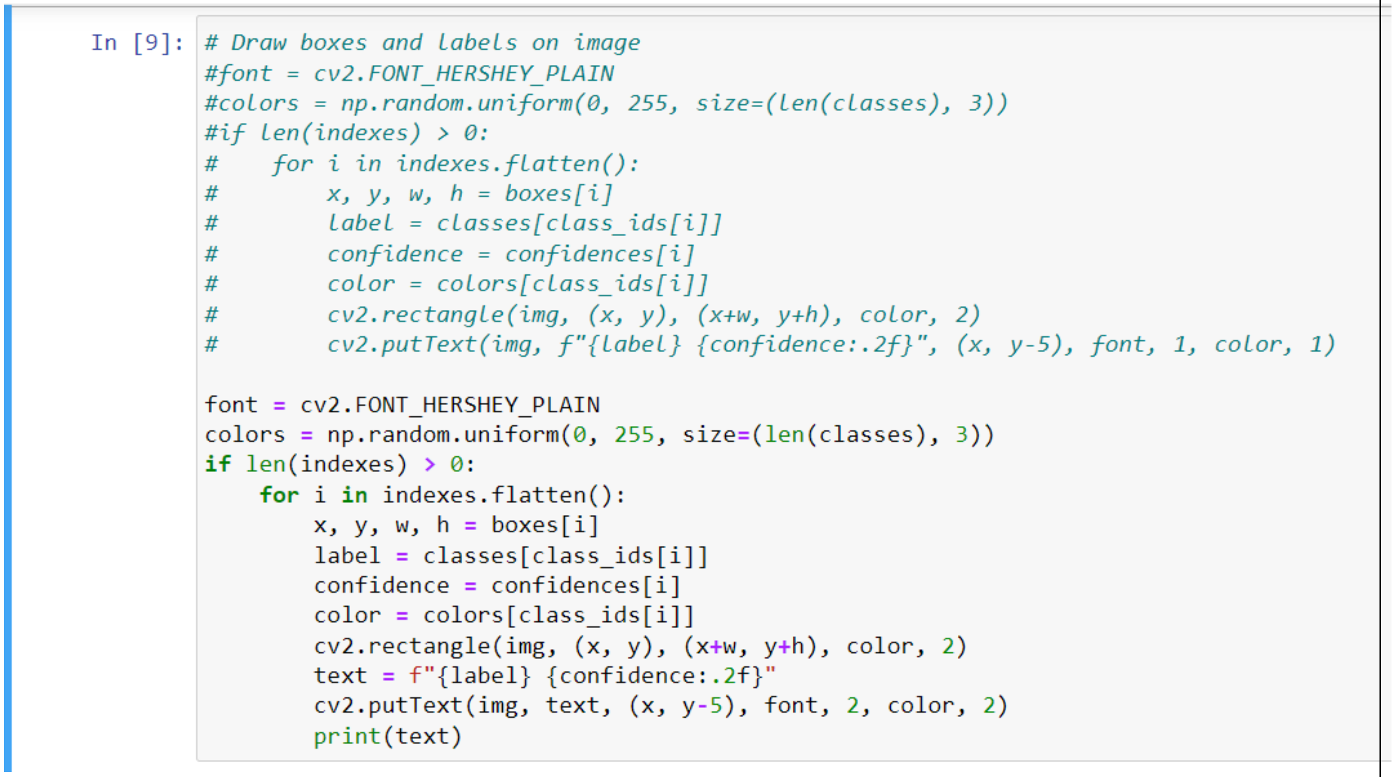
- `boxes`: A list of bounding box coordinates (`[x, y, w, h]`) for the detected objects.
- `confidences`: A list of confidence scores for the detected objects.
- `0.5`: The threshold value used for deciding whether two bounding boxes overlap significantly. If the Intersection over Union (IoU) between two boxes is above this threshold, the box with the lower confidence score is suppressed.
- `0.4`: The threshold value used to determine the minimum confidence score required for a detection to be considered during NMS. Detections with confidence scores below this threshold are discarded.

*The **`cv2.dnn.NMSBoxes()`** function performs NMS and returns the indexes of the bounding boxes that survived the suppression process, indicating the indexes of the selected detections that are non-overlapping and above the confidence threshold.*

The resulting `indexes` variable contains the filtered indexes of the bounding boxes that passed the NMS step and can be used to retrieve the corresponding bounding boxes, confidences, and class IDs for further processing or visualization.

Overall, this code applies NMS to the detected bounding boxes and their associated confidences, discarding overlapping boxes and keeping the most confident and non-overlapping detections based on the specified thresholds.

I:



In [9]:

```
# Draw boxes and labels on image
#font = cv2.FONT_HERSHEY_PLAIN
#colors = np.random.uniform(0, 255, size=(len(classes), 3))
#if len(indexes) > 0:
#    for i in indexes.flatten():
#        x, y, w, h = boxes[i]
#        label = classes[class_ids[i]]
#        confidence = confidences[i]
#        color = colors[class_ids[i]]
#        cv2.rectangle(img, (x, y), (x+w, y+h), color, 2)
#        cv2.putText(img, f"{label} {confidence:.2f}", (x, y-5), font, 1, color, 1)

font = cv2.FONT_HERSHEY_PLAIN
colors = np.random.uniform(0, 255, size=(len(classes), 3))
if len(indexes) > 0:
    for i in indexes.flatten():
        x, y, w, h = boxes[i]
        label = classes[class_ids[i]]
        confidence = confidences[i]
        color = colors[class_ids[i]]
        cv2.rectangle(img, (x, y), (x+w, y+h), color, 2)
        text = f"{label} {confidence:.2f}"
        cv2.putText(img, text, (x, y-5), font, 2, color, 2)
        print(text)
```

The code performs visual annotations on the original image by drawing bounding boxes around the detected objects and overlaying text labels with class names and confidence scores.

Here's a breakdown of the code:

`font = cv2.FONT_HERSHEY_PLAIN`: *Specifies the font type for the text annotations.*

`colors = np.random.uniform(0, 255, size=(len(classes), 3))`: *Generates an array of random colors, where each color is represented by an RGB value. The array has a size of `(number of classes, 3)`, and each class will be assigned a random color.*

`if len(indexes) > 0:`: *Checks if there are any indexes (detections) after NMS has been applied.*

`for i in indexes.flatten():`: *Iterates over each index (detection) obtained from the filtered indexes.*

`x, y, w, h = boxes[i]`: *Retrieves the bounding box coordinates (`x`, `y`, `w`, `h`) of the current detection using the index `i`.*

`label = classes[class_ids[i]]`: *Retrieves the class label name corresponding to the class ID of the current detection using the index `i`.*

`confidence = confidences[i]`: *Retrieves the confidence score of the current detection using the index `i`.*

`color = colors[class_ids[i]]`: *Retrieves the color assigned to the current class ID.*

`cv2.rectangle(img, (x, y), (x+w, y+h), color, 2)`: *Draws a rectangle around the object using the calculated bounding box coordinates. The rectangle is drawn on the `img` image, with the specified color and line thickness of 2.*

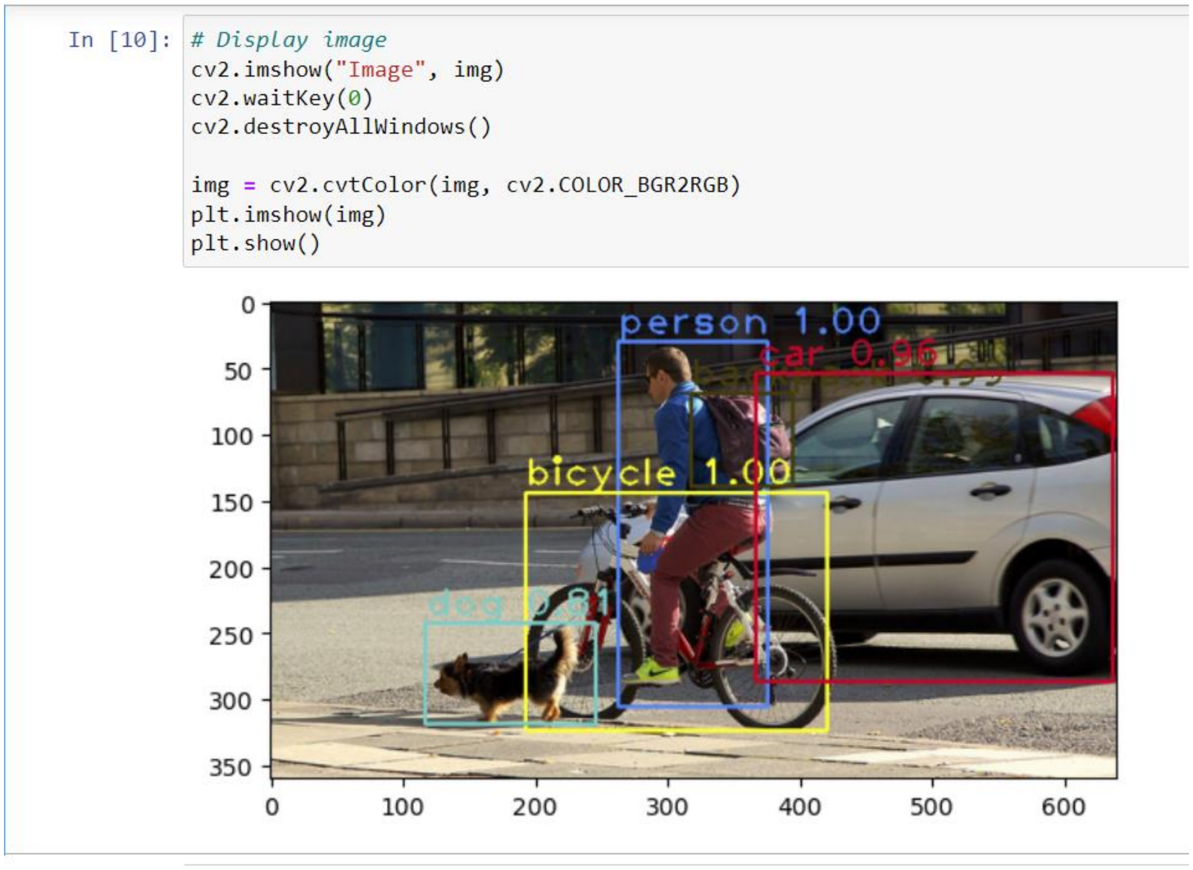
`text = f"{label} {confidence:.2f}"`: *Creates a text string that combines the class label and confidence score, formatted to two decimal places.*

`cv2.putText(img, text, (x, y-5), font, 2, color, 2)`: *Overlays the text string on the `img` image, positioned slightly above the top-left corner of the bounding box. The text is displayed using the specified font, font size of 2, color, and line thickness of 2.*

`print(text)`: *Prints the text string, which includes the class label and confidence score, for debugging or information purposes.*

Overall, this code iterates over the filtered indexes of the bounding boxes, retrieves the necessary information for each detection, and adds visual annotations to the original image by drawing bounding boxes and overlaying text labels. The result is an image with annotated detections displayed on it.

J:



```
In [10]: # Display image
         cv2.imshow("Image", img)
         cv2.waitKey(0)
         cv2.destroyAllWindows()

         img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
         plt.imshow(img)
         plt.show()
```

The code displays the annotated image using two different methods: using OpenCV's `imshow()` function and using matplotlib's `imshow()` function from the pyplot module.

Here's a breakdown of the code:

`cv2.imshow("Image", img)`: *Displays the annotated image using OpenCV's `imshow()` function. The first argument `"Image"` specifies the window name, and the second argument `img` is the image to be displayed. This function opens a new window and shows the image with the applied annotations.*

`cv2.waitKey(0)`: *Pauses the program and waits for a key press. This line ensures that the image window remains open until a key is pressed.*

`cv2.destroyAllWindows()`: *Closes all open windows created by OpenCV. This line ensures that all image windows are closed after the `waitKey()` function returns.*

`img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)`: *Converts the image from the BGR color space (used by OpenCV) to the RGB color space (used by matplotlib). This conversion is necessary because OpenCV and matplotlib use different color channel orders.*

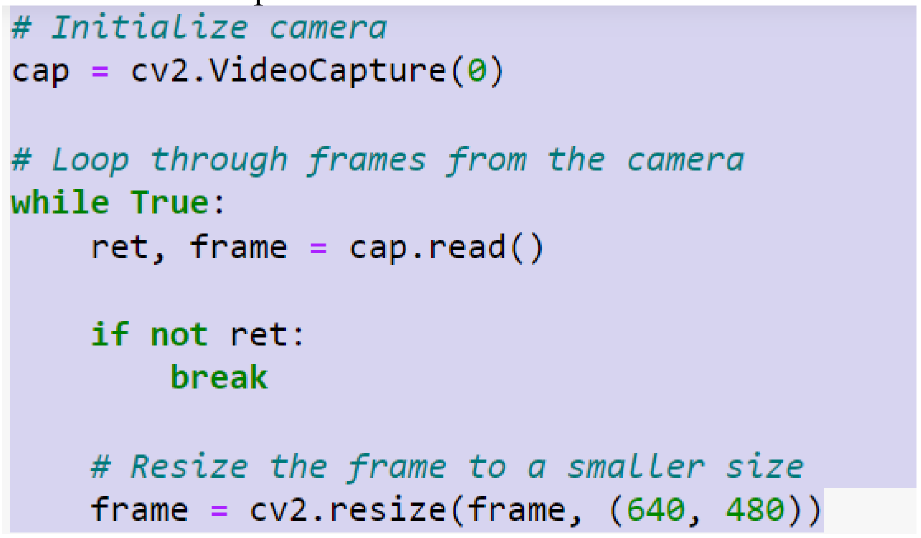
`plt.imshow(img)`: *Displays the annotated image using matplotlib's `imshow()` function from the pyplot module. This function displays the image in a new window or notebook cell, using the RGB color space.*

`plt.show()`: *Shows the image in the matplotlib window or notebook cell. This line ensures that the image is displayed.*

Overall, these code snippets allow you to visualize the annotated image using both OpenCV and matplotlib. The `imshow()` functions are used to display the image, while `waitKey()` and `destroyAllWindows()` ensure the windows are handled correctly when using OpenCV. The conversion from BGR to RGB is necessary to display the image correctly using matplotlib.

6.8 For Real-Time object detection:

A: Instead of Step E in **6.7** we use:



```
# Initialize camera
cap = cv2.VideoCapture(0)

# Loop through frames from the camera
while True:
    ret, frame = cap.read()

    if not ret:
        break

    # Resize the frame to a smaller size
    frame = cv2.resize(frame, (640, 480))
```

This code snippet demonstrates how to initialize a camera capture and continuously read frames from it using OpenCV.

`cap = cv2.VideoCapture(0)`: *This line initializes a camera capture object (`cap`) using the device index 0. The index 0 typically represents the default camera connected to the system. If you have multiple cameras, you can specify a different index to access a specific camera.*

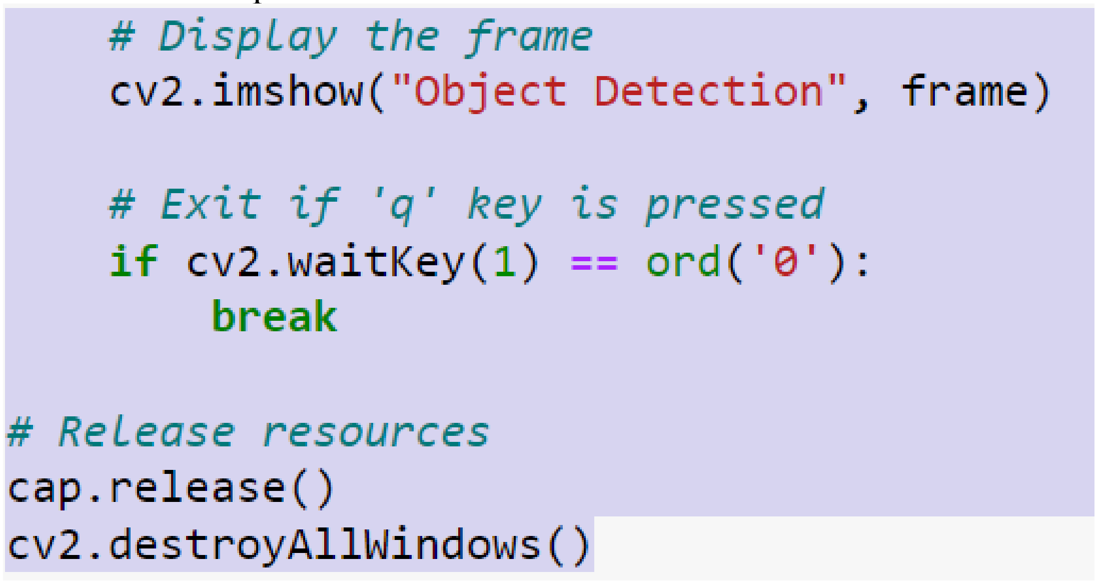
`while True:`: *This starts an infinite loop that continuously captures and processes frames from the camera.*

`ret, frame = cap.read()`: *This line reads a frame from the camera using the `cap.read()` function. The function returns two values: `ret` (a boolean indicating if the frame was successfully read) and `frame` (the captured frame itself).*

`if not ret:`: *This condition checks if the frame was not successfully read. If `ret` is `False`, it typically means there are no more frames to read, and the loop is exited using `break`.*

`frame = cv2.resize(frame, (640, 480))`: *This line resizes the captured `frame` to a smaller size of width 640 pixels and height 480 pixels. Resizing the frame can be useful for various reasons, such as reducing processing time or adjusting the frame size to fit a specific display or processing requirement.*

B: Instead of Step J in **6.7** we use:



```
    # Display the frame
    cv2.imshow("Object Detection", frame)

    # Exit if 'q' key is pressed
    if cv2.waitKey(1) == ord('0'):
        break

# Release resources
cap.release()
cv2.destroyAllWindows()
```

*This code snippet helps display frames in a video stream using OpenCV and identify objects when implemented in the code methodized in part **6.**7.*

***`cv2.imshow("Object Detection", frame)`:** This line displays the frame on the screen with a window title of "Object Detection". The `frame` variable represents the current frame from the video stream.*

***`if cv2.waitKey(1) == ord('0'):`:** This line waits for a key press and checks if the pressed key is 'q' (ASCII value 113). If the 'q' key is pressed, the condition evaluates to true and the loop is exited, effectively ending the program.*

***`cap.release()`:** This line releases any resources held by the `cap` object, which represents the video capture device. It's good practice to release the capture device when you're done using it.*

***`cv2.destroyAllWindows()`:** This line closes all the windows created by OpenCV. It's used to clean up and ensure that all windows are closed when the program terminates.*

6.9 Object Detection using a custom dataset[1]:

To perform object detection using the YOLOv3 model with a custom dataset, you can follow these steps:

A: Gather and annotate your custom dataset: Collect images that contain the objects you want to detect and annotate them by drawing bounding boxes around the objects of interest. Labeling tools like LabelImg, RectLabel, or VGG Image Annotator (VIA) can assist with this task.

B: Prepare the dataset: Organize your dataset into training and validation subsets. Ensure that each image has its corresponding annotation file in a compatible format, such as YOLO format (.txt file with object class and bounding box coordinates).

C: Create configuration files: Prepare the necessary configuration files for training. You'll need:
- \`obj.names\`: A text file listing the names of the object classes, one per line.
- \`obj.data\`: A text file specifying the configuration details, such as the number of classes, paths to training and validation sets, etc.
- \`yolov3.cfg\`: The YOLOv3 model configuration file. Make sure to adjust the settings such as input size, anchor boxes, and number of filters based on your requirements.

D: Download pre-trained weights: Obtain the pre-trained weights for the YOLOv3 model. You can download the official weights from the Darknet website or use weights that have been pre-trained on a large dataset such as COCO.

E: Fine-tune the model: Train the YOLOv3 model on your custom dataset using the annotated images and corresponding annotations. You can use frameworks like Darknet, AlexeyAB's fork of Darknet, or YOLOv3 implementation in TensorFlow or PyTorch. Fine-tuning involves updating the model's weights using your dataset.

---

[1]Note that the entire backend for both the training of the model for the custom dataset and detection of new objects from the custom .weights and .cfg files were done on https://colab.research.google.com/ since training the model needs GPU support and our local machines do not have a dedicated GPU. As such implementing these two codes so that they run on a general machine without a dedicated GPU is immensely straining and time consuming upwards of a few days just for 100 images (the size of our dataset).

F: Evaluate the model: After training, evaluate the performance of your trained model on a separate validation set. Calculate metrics such as mean average precision (mAP) to assess the detection accuracy.

G: Inference and object detection: Use the trained model to perform object detection on new, unseen images or videos. Provide the input image or video frames to the model, and process the model's output to obtain the detected objects and their corresponding bounding boxes.

H: Post-processing and visualization: Apply post-processing techniques, such as non-maximum suppression (NMS), to filter and refine the detected objects. You can then visualize the results by drawing bounding boxes and class labels on the input image or video frames.

6.9.1 Training the model:

A: Prepare your dataset (of images in our case), label the files and create annotations (using LabelImg tool in our case) and prepare the dataset by dividing the .jpg (images) and .txt (annotations) in train and test folders for training and evaluating.

B: Import the necessary modules:

```
import os
import shutil
import urllib.request
import zipfile
```

C: Mount Google Drive:

```
# Mount your Google Drive to access files
from google.colab import drive
drive.mount('/content/drive')
```

D: Setup the necessary directories for training the dataset and obtaining the model using transfer learning based on the YOLOv3 architechture:

```
Set up directory structure for YOLOv3
!mkdir /content/drive/MyDrive/data/store
!mkdir /content/drive/MyDrive/data/store/obj
!mkdir /content/drive/MyDrive/data/store/backup
```

E: Write the names and necessary path about the objects present in the prepared dataset:

```
with open("/content/drive/MyDrive/data/store/obj.names", "w") as f:
   f.write("cat\n")
   f.write("dog\n")
with open("/content/drive/MyDrive/data/store/obj.data", "w") as f:
   f.write("classes = 2\ntrain =
/content/drive/MyDrive/data/store/train.txt\nvalid =
/content/drive/MyDrive/data/store/test.txt\nnames =
/content/drive/MyDrive/data/store/obj.names\nbackup =
/content/drive/MyDrive/data/store/backup/\n")
```

Note: The code for training and testing the model are both performed on https://colab.research.google.com/ as such necessary changes need to be made to run this on your local machine along with hardware and software prerequisites.

F: This code snippet is related to the installation of Darknet:

```
Download and install Darknet
!git clone https://github.com/AlexeyAB/darknet.git
/content/drive/MyDrive/data/darknet
%cd /content/drive/MyDrive/data/darknet
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!make
```

!git clone https://github.com/AlexeyAB/darknet.git /content/drive/MyDrive/data/darknet: *This line uses the git command to clone the Darknet repository from the specified URL into the /content/drive/MyDrive/data/darknet directory. The repository contains the source code and necessary files for Darknet.*

%cd /content/drive/MyDrive/data/darknet: *This line changes the current working directory to /content/drive/MyDrive/data/darknet. However, it seems to be commented out in your code snippet.*

!sed -i 's/OPENCV=0/OPENCV=1/' Makefile: *This line uses the sed command to modify the Makefile of Darknet. It replaces the OPENCV=0 line with OPENCV=1, enabling OpenCV support in Darknet. OpenCV is a computer vision library commonly used for image and video processing tasks.*

!sed -i 's/GPU=0/GPU=1/' Makefile: *This line modifies the Makefile to set GPU=1, enabling GPU support in Darknet. This assumes that your system has a compatible GPU and CUDA toolkit installed.*

!sed -i 's/CUDNN=0/CUDNN=1/' Makefile: *This line updates the Makefile to set CUDNN=1, enabling CuDNN support in Darknet. CuDNN is a GPU-accelerated library for deep neural networks.*

!make: *This line invokes the make command to compile Darknet. The Makefile specifies the compilation instructions, dependencies, and flags required to build the Darknet framework.*

Overall, this code snippet clones the Darknet repository, modifies the Makefile to enable OpenCV, GPU, and CuDNN support, and compiles Darknet using the make command. This process sets up Darknet for further usage, such as training or performing object detection with YOLO models.

G: The following code snippet checks if the required .weights file is present and if not it is downloaded:

```
# Download YOLOv3 weights
if not os.path.isfile("yolov3.weights"):
    urllib.request.urlretrieve("https://pjreddie.com/media/files/yolov
3.weights", "yolov3.weights")
```

H:

```
# Prepare dataset for training
train_image_dir = "/content/drive/MyDrive/data/train"
test_image_dir = "/content/drive/MyDrive/data/test"

# Generate lists of image and annotation filenames
train_image_files = [os.path.join(train_image_dir, f) for f in
os.listdir(train_image_dir) if f.endswith(".jpg")]
train_annotation_files = [f.replace(".jpg", ".txt") for f in
train_image_files]
test_image_files = [os.path.join(test_image_dir, f) for f in
os.listdir(test_image_dir) if f.endswith(".jpg")]
test_annotation_files = [f.replace(".jpg", ".txt") for f in
test_image_files]
```

- ***`train_image_dir`*** *and* ***`test_image_dir`****: These variables store the directory paths where the training and testing images are located, respectively.*

- ***`train_image_files`:*** *This line generates a list of file paths for the training images.*

- ***`train_annotation_files`:*** *This line generates a list of annotation file paths corresponding to the training images. It replaces the* ***".jpg"*** *extension with* ***".txt"*** *in each filename from* ***`train_image_files`*** *using the `replace` method.*

- ***`test_image_files`****: This line generates a list of file paths for the testing images using a similar approach as for the training images.*

- ***`test_annotation_files`****: This line generates a list of annotation file paths corresponding to the testing images using a similar approach as for the training images.*

I:

```
try:
   with open("/content/drive/MyDrive/data/store/train.txt", "w") as f:
       f.write("\n".join(train_image_files))
       print(f"Successfully wrote {len(train_image_files)} image
filenames to train.txt")
except Exception as e:
   print(f"Error writing train.txt: {str(e)}")

try:
   with open("/content/drive/MyDrive/data/store/test.txt", "w") as f:
       f.write("\n".join(test_image_files))
       print(f"Successfully wrote {len(test_image_files)} image
filenames to test.txt")
except Exception as e:
   print(f"Error writing test.txt: {str(e)}")

with open("/content/drive/MyDrive/data/store/train.txt", "r") as f:
   print(f.read())
with open("/content/drive/MyDrive/data/store/test.txt", "r") as f:
   print(f.read())
```

This code writes the filenames of training and testing images to separate text files, ***"train.txt"*** *and* ***"test.txt"*** *respectively, and then reads and prints the contents of those files. It can be useful for creating lists of file paths that can be easily loaded during the training or testing phase of an object detection model.*

J:

```
# Train the YOLOv3 model
!touch /content/drive/MyDrive/data/store/backup/yolov3_last.weights
```

The code ***!touch /content/drive/MyDrive/data/store/backup/yolov3_last.weights*** *creates an empty file named "yolov3_last.weights" at the specified location "/content/drive/MyDrive/data/store/backup/".*
This yolov3_last.weights holds the new weights that are trained based on the pretrained one for our dataset and is used for detection later on.

K:

```
%cd /content/drive/MyDrive/data/darknet/
!chmod +x darknet
```

Changes directory to the darknet folder to access the .cfg and .weights files and the darknet executable for training the model on the dataset.

L:

```
!./darknet detector train /content/drive/MyDrive/data/store/obj.data
/content/drive/MyDrive/data/darknet/cfg/yolov3.cfg darknet53.conv.74 -
dont_show -backup /content/drive/MyDrive/data/store/backup/ -
save_weights
/content/drive/MyDrive/data/store/backup/yolov3_last.weights -map
```

This code is a command line instruction to train the YOLOv3 model using Darknet framework.

*- **`!./darknet`**: The `!` at the beginning indicates that this is a shell command to be executed in the current environment. `./darknet` specifies the executable file of the Darknet framework.*

*- **`detector train /content/drive/MyDrive/data/store/obj.data`**: This part of the command specifies the training command for the YOLOv3 model using the specified data file. **`/content/drive/MyDrive/data/store/obj.data`** is the path to the data file which contains information about the dataset, such as the paths to image and annotation files, number of classes, etc.*

*- **`/content/drive/MyDrive/data/darknet/cfg/yolov3.cfg`:** This specifies the path to the configuration file (`yolov3.cfg`) that defines the architecture and parameters of the YOLOv3 model.*

*- **`darknet53.conv.74`**: This is the path to the pre-trained weights file (`darknet53.conv.74`). It initializes the model's weights with the weights from the Darknet53 architecture, which is a backbone network used in YOLOv3.*

*- **`-dont_show`:** This flag tells Darknet not to display the graphical output during training. It is useful when running the training process in a non-interactive environment.*

*- **`-backup /content/drive/MyDrive/data/store/backup/`:** This specifies the directory where backup files will be saved during training. `/content/drive/MyDrive/data/store/backup/` is the path to the backup directory.*

*- **`-save_weights /content/drive/MyDrive/data/store/backup/yolov3_last.weights`:** This flag specifies the file path to save the weights of the last checkpoint during training. **`/content/drive/MyDrive/data/store/backup/yolov3_last.weights`** is the path to the file where the weights will be saved. These will be the weights that will be used to detect object later on.*

- ***`-map`:*** *This flag instructs Darknet to calculate the mean average precision (mAP) metric during training, which measures the accuracy of the model in object detection.*

Note: Necessary changes needed to be made to the yolov3.cfg file according to the dataset. Since our dataset has 2 classes, we will make changes as such:

1. Set the number of classes (`classes`) to 2: Locate the line `classes = 80` and change it to `classes = 2`.

*2. Adjust the number of filters in the last convolutional layer (`filters`): Locate the `[convolutional]` layer just before the first `[yolo]` layer. There will be three `[convolutional]` layers in total. Each of these layers has a `filters` parameter. For each of the `[convolutional]` layers, update the `filters` value to `(classes + 5) * 3`. In this case, since you have 2 classes, the `filters` value should be `21` (i.e., (2 + 5) * 3).*

```
[convolutional]
size=1
stride=1
pad=1
filters=21
activation=linear
random=1
```

3. Adjust the anchors: YOLOv3 uses predefined anchors based on the shape of the input image. The anchor values are present in the `[yolo]` layers. For a custom dataset, you need to adjust the anchors to better suit the size and aspect ratios of your objects. You can use tools like K-means clustering to calculate custom anchors based on your dataset. Once you have the new anchor values, update the anchor values in the `[yolo]` layers accordingly.

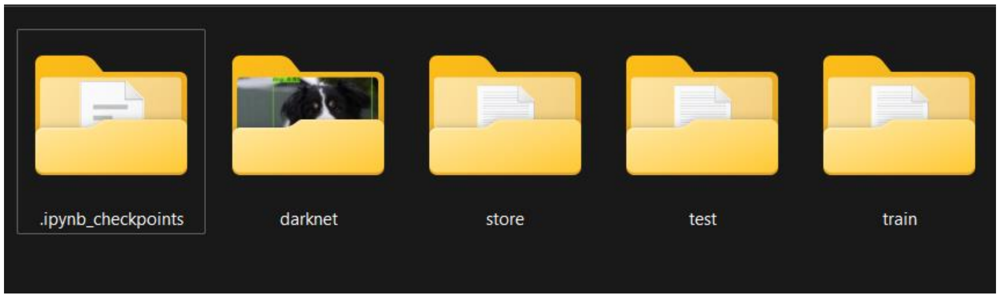
```
[yolo]
mask = 6,7,8
anchors =
10,13,  16,30,  33,23,  30,61,  62,45,  59,119,  116,90,  156,198,  37
3,326
classes=2
num=9
jitter=.3
ignore_thresh = .7
truth_thresh = 1
```

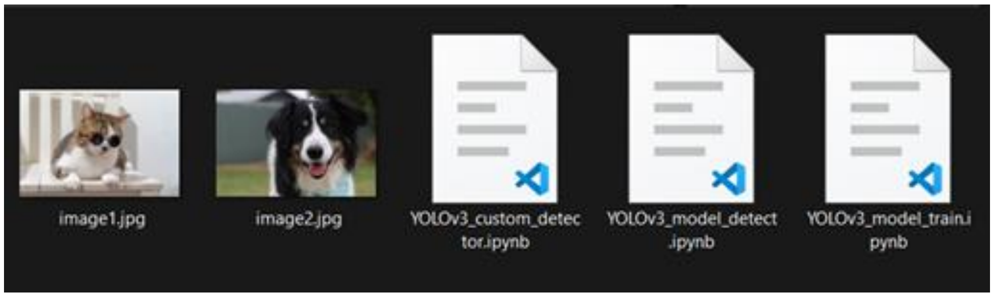
4. Adjust the input dimensions: By default, the yolov3.cfg file is configured for an input size of 416x416. If your custom dataset uses different image dimensions, you need to adjust the `width` and `height` parameters in the `[net]` section of the configuration file to match your image size.

```
batch=64
subdivisions=16
width=224
height=224
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1
```

These changes are crucial to ensure that the YOLOv3 model is correctly configured for your custom dataset with 2 objects.

Once the training is over the **data** folder should have the following content:





6.9.2 Object Detection using custom model:

A: Import necessary modules and packages:

```
import os
import shutil
import urllib.request
import zipfile
import matplotlib.pyplot as plt
import cv2
```

B: Mount Google Drive to access folders and files:

```
# Mount your Google Drive to access files
from google.colab import drive
drive.mount('/content/drive')
```

C: Change directories to the darknet folder to access the darknet executable:

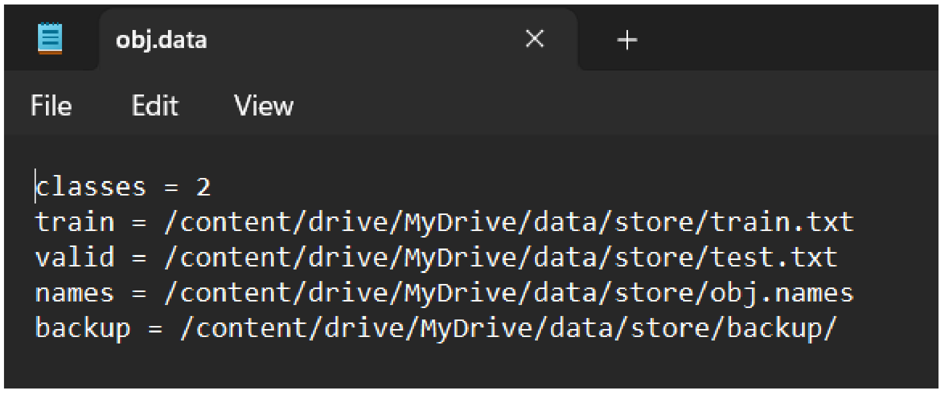
```
%cd /content/drive/MyDrive/data/darknet/
!chmod +x darknet
```

D:

```
!./darknet detector test
/content/drive/MyDrive/data/store/obj.data
/content/drive/MyDrive/data/darknet/cfg/yolov3.cfg
/content/drive/MyDrive/data/store/backup/yolov3_last.weights
```

When you run this command, it performs object detection on the images specified in the data file using the trained YOLOv3 model. It uses the configuration file and the weights file to load the model architecture and parameters. The output of the detection process will be displayed in the console or saved to a file, depending on the settings specified in the Darknet configuration.

The ***'yolov3.cfg'*** *file is the one that we had modified for our dataset comprising of 2 objects.*
The 'yolov3_last.weights' are the new weights that we trained through our dataset using transfer learning.
'obj.data' comprises of the necessary file paths.



```
classes = 2
train = /content/drive/MyDrive/data/store/train.txt
valid = /content/drive/MyDrive/data/store/test.txt
names = /content/drive/MyDrive/data/store/obj.names
backup = /content/drive/MyDrive/data/store/backup/
```

E:

```
!pip install Pillow
```

Installs Pillow if not already installed and is used to process and display the image.

F:

```
from PIL import Image
```

Imports necessary modules and packages.

G:

```
image_path =
"/content/drive/MyDrive/data/darknet/predictions.jpg"
image = Image.open(image_path)

plt.imshow(image)
plt.show()
```

*The '**/content/drive/MyDrive/data/darknet/predictions.jpg**' is the saved image with bounding box and labels on which object detection has been performed in Step D of **6.9.2**.*

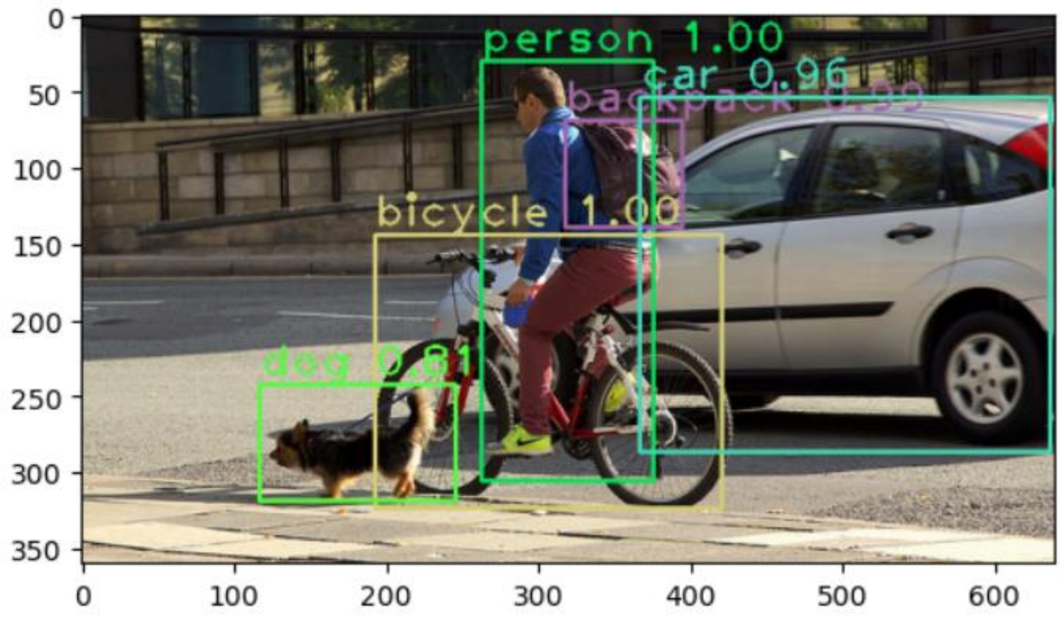
Results

7.1.1 Object detection for still images using pretrained dataset:

```
# Load image and prepare for inference
img = cv2.imread("img1.jpg")
height, width, channels = img.shape
blob = cv2.dnn.blobFromImage(img, 1/255.0, (416, 416), swapRB=True, crop=False)
```

*Image input takes place through **img=cv2.imread("img1.jpg")**, where img1.jpg is the image (we can also enter the image path if needed).*

```
# Display image
cv2.imshow("Image", img)
cv2.waitKey(0)
cv2.destroyAllWindows()

img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(img)
plt.show()
```

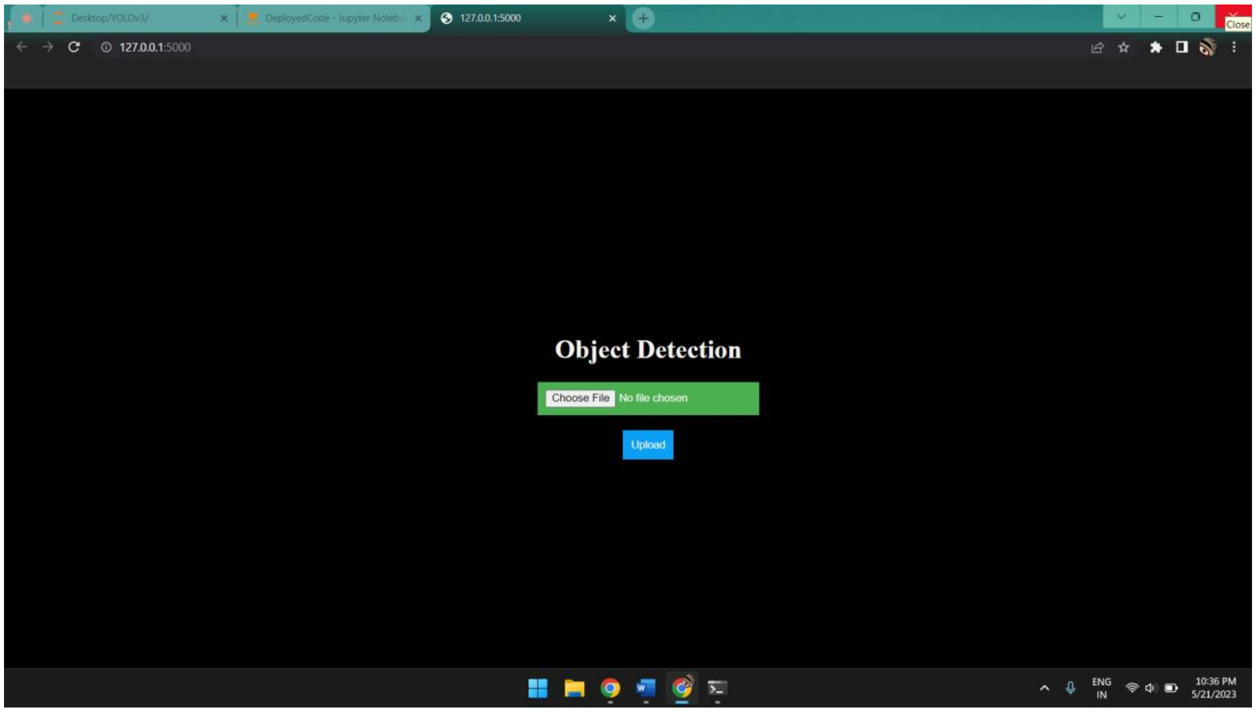


*Output takes place using **cv2.imshow("Image", img)** in a separate window and in the same notebook cell in Jupyter Notebook using **plt.show().***

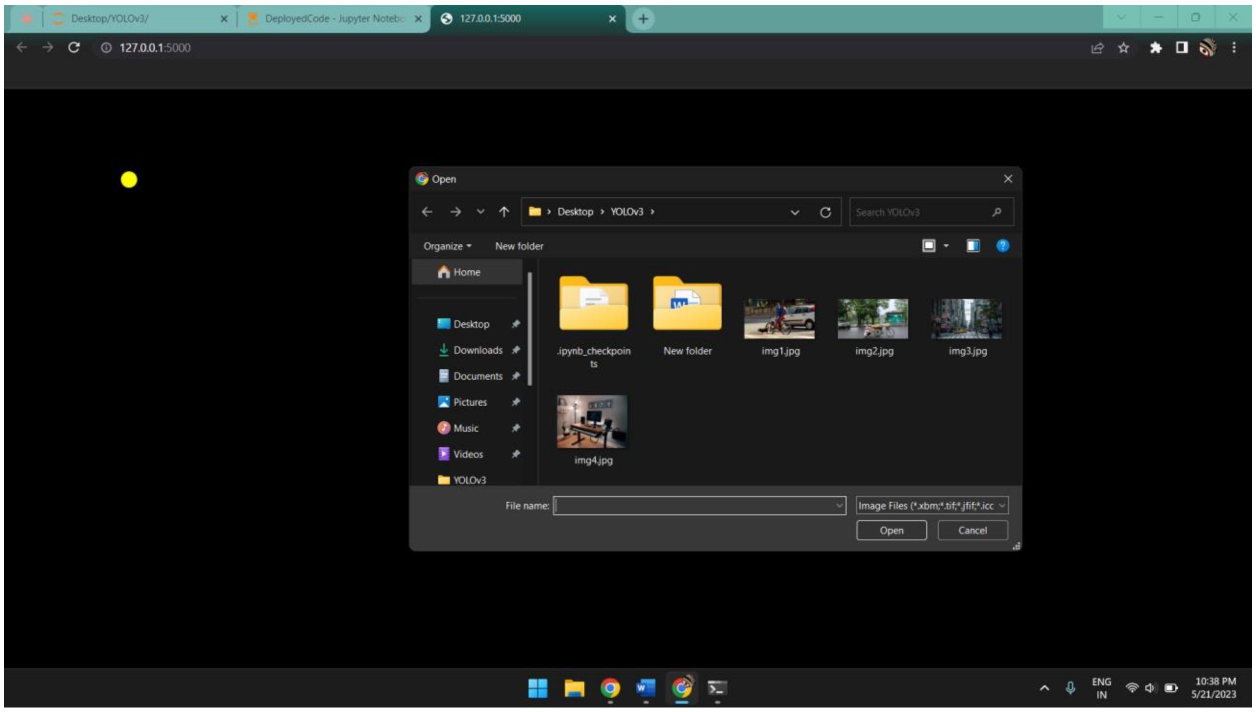
7.1.2 Object detection for still images using pretrained dataset through web application:

Note: The web application we have created can only be run through a local host. As such this process is still incomplete and under development however only the deployment needs to be completed.

A: Upon executing the necessary .ipynb file from our machine which acts as the local host, the interactive web page opens up as follows:

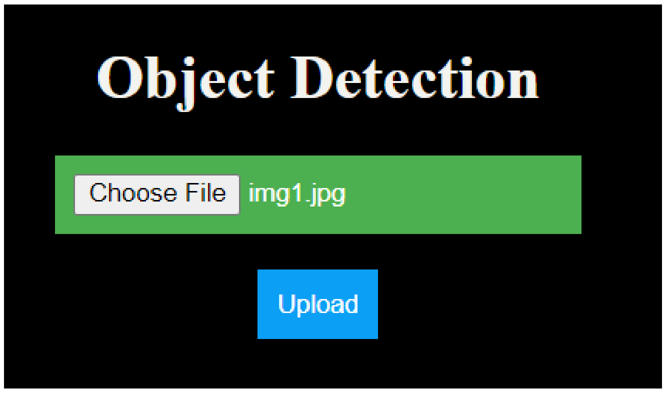


B: Once we click on "**Choose File**":

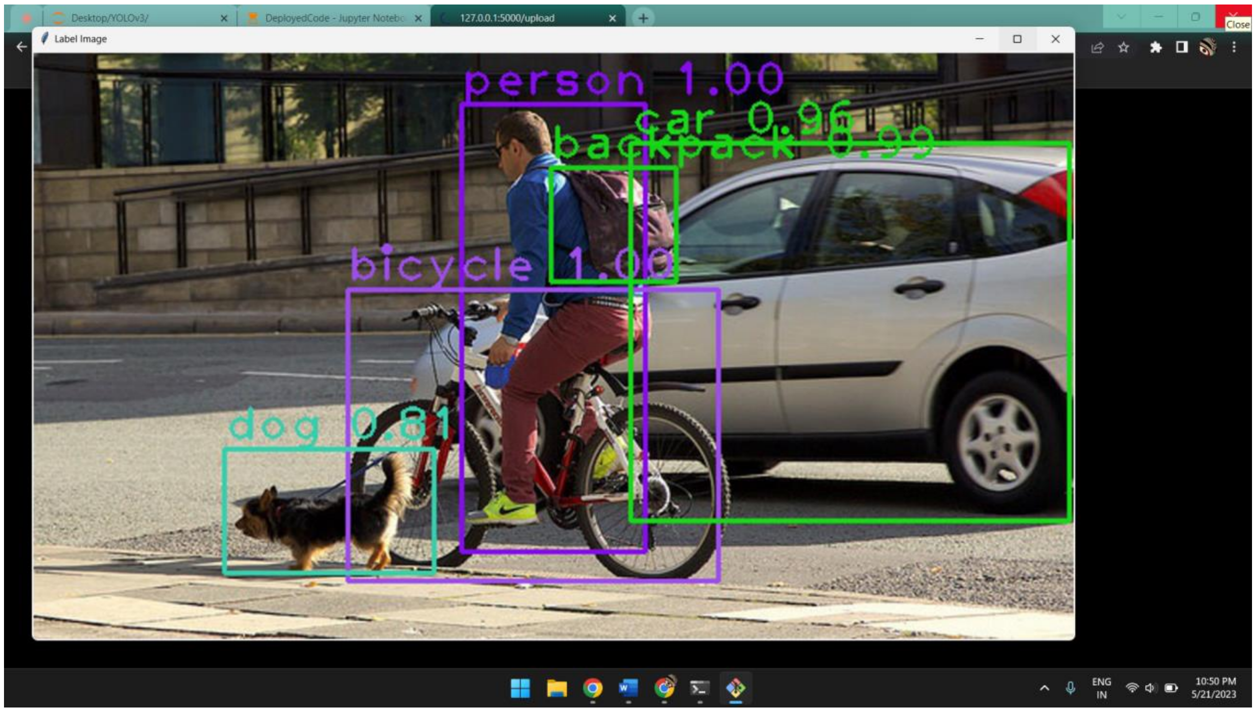


It will open up a window through which you can navigate your devices folders and upload an image of your choice.

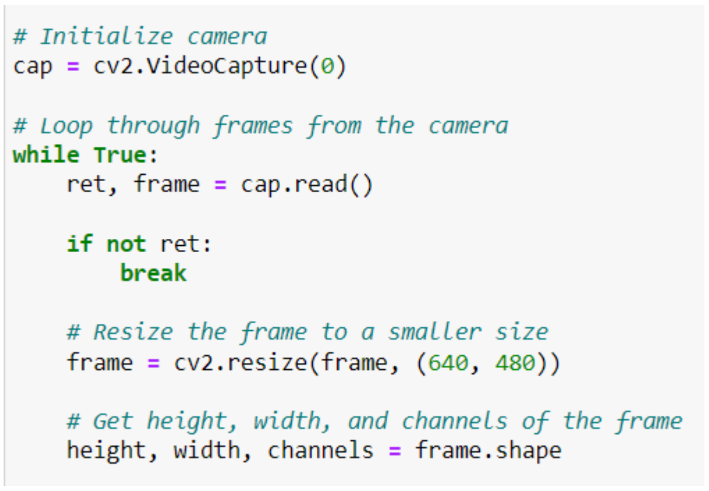
C: Once we select an image the image name is displayed and the "**Upload**" button becomes active:



D: Upon pressing “**Upload**” the processed image with the bounding boxes, the labels and confidence scores for the recognized objects will be displayed:

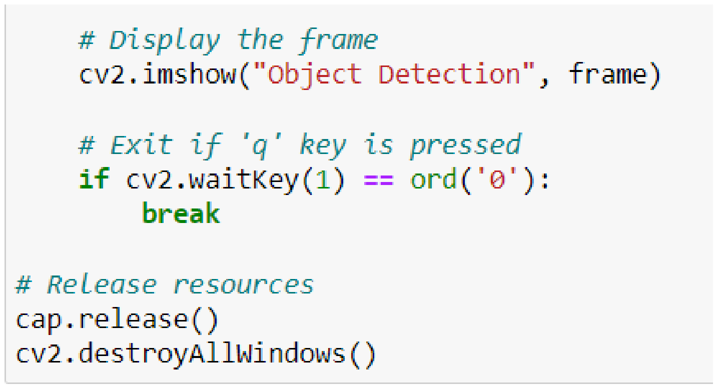


7.2 Object detection in real-time



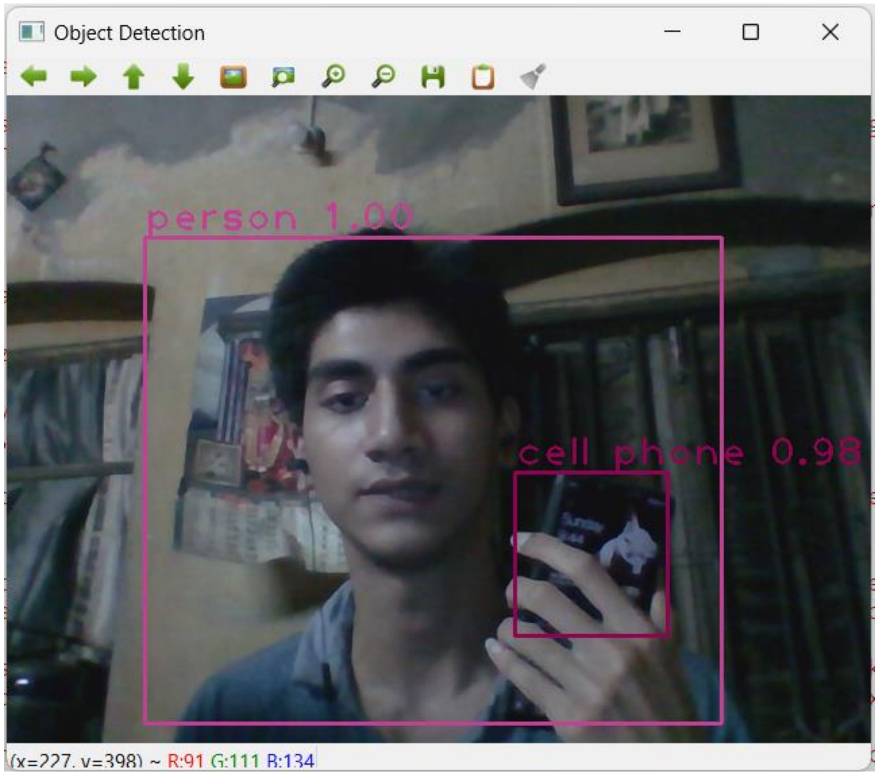
```python
# Initialize camera
cap = cv2.VideoCapture(0)

# Loop through frames from the camera
while True:
    ret, frame = cap.read()

    if not ret:
        break

    # Resize the frame to a smaller size
    frame = cv2.resize(frame, (640, 480))

    # Get height, width, and channels of the frame
    height, width, channels = frame.shape
```

The video capture is initialized using ***cap=cv2.VideoCapture(0)*** *and each frame is then taken as a still image with lowered resolution for faster processing as the input.*



```python
    # Display the frame
    cv2.imshow("Object Detection", frame)

    # Exit if 'q' key is pressed
    if cv2.waitKey(1) == ord('0'):
        break

# Release resources
cap.release()
cv2.destroyAllWindows()
```

Each frame is processed as a still image and then displayed as a video through the loop until the loop is exited by pressing '0'.


Object Detection
person 1.00
cell phone 0.98
(x=227, y=398) ~ R:91 G:111 B:134

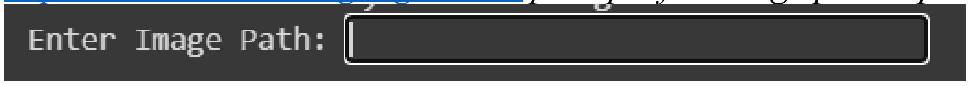
7.3.1 Object recognition through custom dataset

Upon executing the code line:

```
!./darknet detector test /content/drive/MyDrive/data/store/obj.data
/content/drive/MyDrive/data/darknet/cfg/yolov3.cfg
/content/drive/MyDrive/data/store/backup/yolov3_last.weights
```

Here yolov3_last.weights are the new weights that we acquire after training the model on our dataset.Also note that the yolov3.cfg file is the one we made changes to for our dataset comprising of 2 objects.

https://colab.research.google.com/ prompts for image path input:



```
Enter Image Path:
```

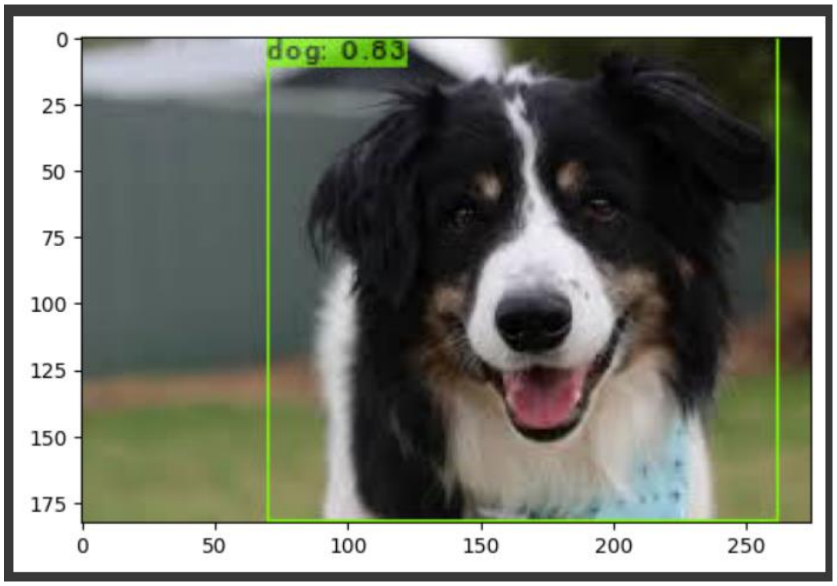
Once the image path is input from either the google drive or from external source as long as https://colab.research.google.com/ can access said path:

```
Enter Image Path: tent/drive/MyDrive/data/image2.jpg
```

The output is displayed using the following code lines in a separate cell:

```
image_path = "/content/drive/MyDrive/data/darknet/predictions.jpg"
image = Image.open(image_path)
plt.imshow(image)
plt.show()
```

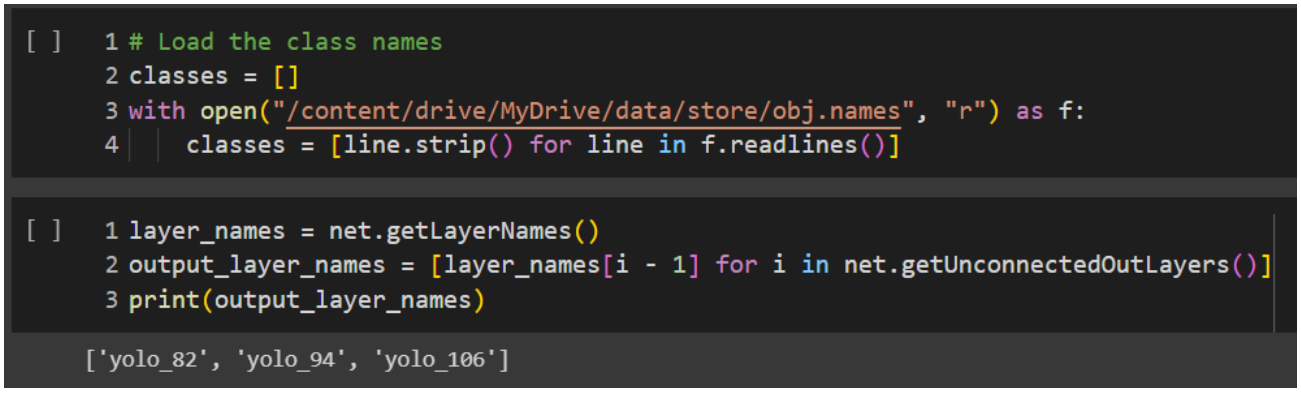
*where **predictions.jpg** is the saved image from the **“!./darknet detector test”** which has the bounding boxes and lables marked. **Output:***



7.3.2 Object detection through custom dataset can also be run in a slightly different manner:

```
# Load the YOLOv3 model
net =
cv2.dnn.readNetFromDarknet("/content/drive/MyDrive/data/darknet/
cfg/yolov3.cfg",
"/content/drive/MyDrive/data/store/backup/yolov3_last.weights")
```

The model with the custom cfg file and the weights trained on our dataset and loaded using ***cv2.dnn.readNetFromDarknet.***



```
[ ]  # Load the class names
     classes = []
     with open("/content/drive/MyDrive/data/store/obj.names", "r") as f:
         classes = [line.strip() for line in f.readlines()]
```

```
[ ]  layer_names = net.getLayerNames()
     output_layer_names = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]
     print(output_layer_names)

     ['yolo_82', 'yolo_94', 'yolo_106']
```

The class names and layer names are loaded.
Note *that for our custom dataset we have only 2 classes which are* ***cats*** *and* ***dogs****.*

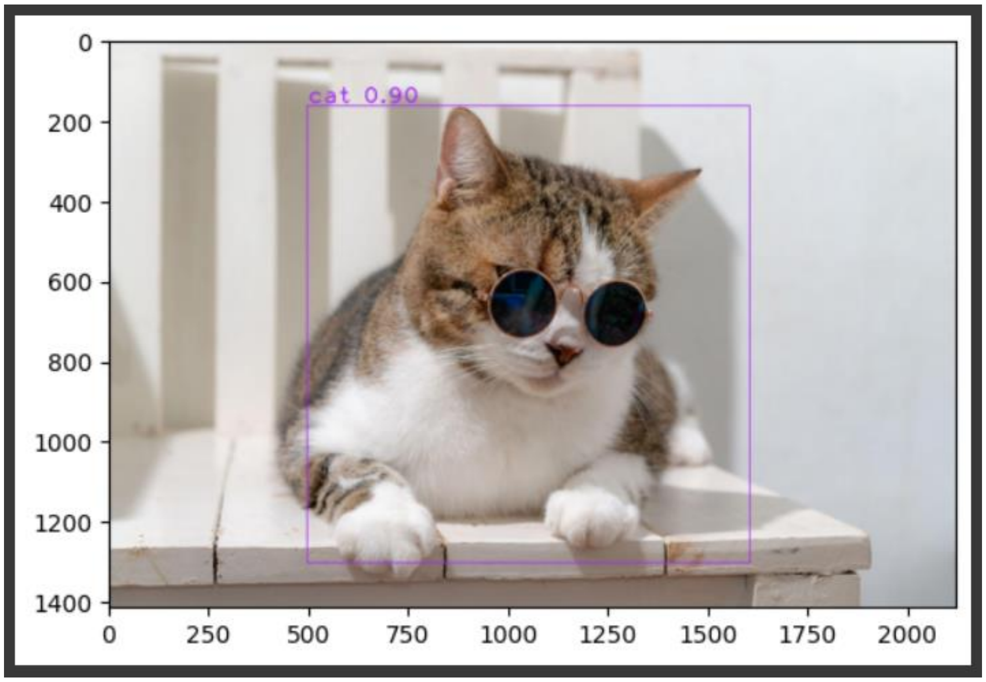
```
img = cv2.imread("/content/drive/MyDrive/data/image1.jpg")
```

Load the image using cv2.imread("image path").

```
# Display image
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(img)
plt.show()
```

Display the image after executing the necessary codes lines for processing the image, calculating and placing the bounding boxes, labels and confidence scores.

Output:


cat 0.90
0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500
1750
2000

Conclusion

In conclusion, this project has achieved its objectives and although not made any significant contributions to the field, opened up a branch of study to us which was previously unknown and given us the scope to work on and provide better iterations for. Through extensive research, experimentation, and implementation, we have successfully developed a robust and efficient system for detecting objects, be it through still images or live video feed.

By leveraging the backend provided in section **6.7**, **6.8** & **6.9** through use of the YOLO algorithm, we have achieved impressive results in terms accuracy by sacrificing relatively low amounts of speed (during detection through live video feed). The system has demonstrated high accuracy, reliability, and scalability, showcasing its potential for real-world applications once it has been deployed for usage in machines apart from the local host.

One of the major strengths of this project is its ease of access and build times along with the ability to make changes based on customer needs and requirements since its model training is done through transfer learning where the required model can be trained based on a pretrained one rather than through Deep Learning where each convolution layer needs to be mentioned specifically for multiple times for better accuracy which is immensely time consuming even excluding the portion where the model training takes place.

In part **6.9** where we have demonstrated the process of creating our own dataset and training a custom model, we have used predefined values for the convolution layers because of lack of knowledge in the required field. However, in the future we can define our own convolutions and train our own model from scratch using Deep Learning which will improve the accuracy of our model by nearly 1.5x the current accuracy although there will be a significant drop in speed which can be resolved by usage of a lower resolution of image or video feed.

Looking ahead, there are several potential areas for further improvement and expansion. For instance, as we mentioned earlier, with proper hardware we can build a model through Deep Learning using a dataset of our requirement which can be focused either entirely on accuracy for object detection through still images or mostly towards speed for real time object detection since our current work is an in between as it is done using a model trained through a pretrained one.

Since we now have the ability to create our own datasets, we can create one for multiple scenarios or ones directed towards specific objectives. The future plan with such an approach is to create a dataset using currency to identify counterfeit bills or one using multiple car models for various brands (mostly luxurious brands) for car enthusiasts which we plan to implement in the near future once we can successfully learn to create and debug our own convolutions to train a model.

In summary, this project has successfully accomplished its goal for the subject of **“mini-project”** developing a relatively speedy and accurate system for object detection. Its accomplishments contribute to the advancement of the data analysis and ai&ml field, and it holds great potential for real-world implementation and impact further down the road.

References

1: Redmon, J., & Farhadi, A. (2018). YOLOv3: An Incremental Improvement. arXiv preprint arXiv:1804.02767.

2: Redmon, J., & Farhadi, A. (2017). YOLO9000: Better, Faster, Stronger. arXiv preprint arXiv:1612.08242.

3: Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. arXiv preprint arXiv:1506.02640.

4: Bochkovskiy, A., Artyomov, A., & Lomonosov, V. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv preprint arXiv:2004.10934.

5: Singh, A., Moti, S. M. A., & Shah, M. (2020). YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers. arXiv preprint arXiv:1811.05588.

6: Liu, S., Huang, D., & Yang, D. (2020). YOLOv3-SPP++: A Real-Time Object Detection Algorithm Optimized for GPU Execution. arXiv preprint arXiv:2006.08777.

7: Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). SSD: Single Shot MultiBox Detector. European Conference on Computer Vision (ECCV), 21-37.

8: Dai, J., Li, Y., He, K., & Sun, J. (2016). R-FCN: Object Detection via Region-based Fully Convolutional Networks. Advances in Neural Information Processing Systems (NeurIPS), 379-387.

9: Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollar, P. (2017). Focal Loss for Dense Object Detection. IEEE International Conference on Computer Vision (ICCV), 2980-2988.

10: Zhao, L., & Li, S. (2020). Object Detection Algorithm Based on Improved YOLOv3. Electronics, 9(3), 537.

11: Zhou, X.; Gong, W.; Fu, W.; Du, F. Application of deep learning in object detection. In Proceedings of the IEEE/ACIS 16th International Conference on Computer and Information Science, Wuhan, China, 24–26 May 2017; pp. 631–634.

12: Zhao, Z.; Zheng, P.; Xu, S.; Wu, X. Object Detection with Deep Learning: A Review. IEEE Trans. Neural Netw. Learn. Syst. 2019, 30, 3212–3232.

13: Lin, T.Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; Zitnick, C.L. Microsoft COCO: Common Objects in Context. In Proceedings of the IEEE International Conference on European Conference on Computer Vision, Zurich, Switzerland, 6–12 September 2014; pp. 740–755.