Object Detection using YOLOv3 and OpenCV

Project Report

Machine Learning (UGCSA212)

Bachelor of Computer Application

PROJECT GUIDE:

ANSHUL GUPTA

SUBMITTED BY:

MD SHABBIR (23CSA2BC214)
MOHAMMAD DANISH (23CSA2BC323)
KUNAL KUMAR (23CSA2BC230)
MUKESH KUMAR (23CSA2BC280)

MAR,2025

VIVEKANANDA GLOBAL UNIVERSITY

ACKNOWLEDGEMENT

I have taken this opportunity to express my gratitude and humble regards to the Vivekananda Global University to provide an opportunity to present a project on the "Object Detection using YOLOv3 and OpenCV" which is a Machine Learning Project.

*I would also be thankful to my project guide **ANSHUL GUPTA** to help me in the completion of my project and the documentation. I have taken efforts in this project but the success of this project would not be possible without their support and encouragement.*

I would like to thanks our principal sir "Dr. Surendra Yadav" to help us in providing all the necessary books and other stuffs as and when required .I show my gratitude to the authors whose books has been proved as the guide in the completion of my project I am also thankful to my classmates and friends who have encouraged me in the course of completion of the project.

Thanks

MD SHABBIR (23CSA2BC214)
MOHAMMAD DANISH (23CSA2BC323)
KUNAL KUMAR (23CSA2BC230)
MUKESH KUMAR (23CSA2BC280)

Place: Jaipur

Date: 31/03/25

DECLARATION

We hereby declare that this Project Report titled “Object Detection using YOLOv3 and OpenCV” submitted by us and approved by our project guide, to the Vivekananda Global University, Jaipur is a bonafide work undertaken by us and it is not submitted to any other University or Institution for the award of any degree diploma / certificate or published any time before.

Project Group

Student Name:	**MD SHABBIR (23CSA2BC214)** **MOHAMMAD DANISH (23CSA2BC323)** **KUNAL KUMAR (23CSA2BC230)** **MUKESH KUMAR (23CSA2BC280)**
Project Guide:	**ANSHUL GUPTA**

Table of Contents

1. Introduction

Object detection is one of the most significant and widely researched domains in computer vision. It involves identifying and locating instances of objects from predefined classes (such as people, vehicles, or traffic signs) within digital images or videos. This capability has found applications across various industries such as surveillance, healthcare, robotics, and especially in autonomous vehicles, where detecting pedestrians, other vehicles, and traffic lights is crucial for navigation and safety.

In this project, we have utilized the YOLOv3 (You Only Look Once version 3) deep learning model in combination with OpenCV (Open Source Computer Vision Library) to detect three types of objects: **cars**, **pedestrians**, and **traffic lights**. YOLOv3 stands out due to its high speed and accuracy in real-time applications. Unlike traditional approaches that use region proposals followed by classification, YOLO performs object detection in a single forward pass, making it exceptionally fast and efficient.

By integrating this model with OpenCV, we were able to create a functional detection pipeline that accepts static images as input and outputs annotated versions highlighting the detected objects. This report elaborates on the objectives, tools used, implementation strategy, results, limitations, and the scope for future enhancement.

---

2. Objectives

The core objectives of this project are as follows:

1. Implement and utilize a pre-trained YOLOv3 deep learning model for object detection tasks.
2. Detect and accurately classify instances of cars, pedestrians, and traffic lights from input images.
3. Visualize the detections using clearly marked bounding boxes and labels.

4. Analyze the model's performance in terms of detection accuracy, speed, and its limitations.
5. Explore the potential of using the system in real-time applications and propose improvements.

---

3. Methodology

3.1 Tools and Technologies

- **OpenCV**: A powerful and widely-used library in the computer vision community, utilized for image processing and visual output.
- **YOLOv3**: A fast, real-time object detection system that can detect multiple objects in a single evaluation.
- **COCO Dataset**: The model is pre-trained on the COCO (Common Objects in Context) dataset, which includes 80 object classes. For our project, we specifically target class IDs:
    - Pedestrian (class ID: 0)
    - Car (class ID: 2)
    - Traffic light (class ID: 9)

3.2 Workflow

1. **Model Loading**
   Load the pre-trained weights (`yolov3.weights`), configuration file (`yolov3.cfg`), and class labels (`coco.names`) required for YOLOv3.
2. **Image Preprocessing**
    - The input image is resized to 416x416 pixels and converted into a 4D blob using `cv2.dnn.blobFromImage()`.
    - This step normalizes the image and makes it suitable for processing through the deep neural network.
3. **Detection**

- The blob is passed through the network using `net.forward()` to obtain object predictions.
- The model outputs several bounding boxes along with class scores for each object. Detections are filtered by a confidence threshold of 50%.

4. **Post-Processing**
   - **Non-Max Suppression (NMS)** is applied to eliminate redundant overlapping boxes, retaining only the best predictions.
   - Final bounding boxes and labels are drawn onto the image using `cv2.rectangle()` and `cv2.putText()`.

---

4. Implementation

4.1 Project Structure

```
Project_Folder/
├── models/       # Contains YOLOv3 model files (weights, config, coco.names)
├── images/               # Directory of test images
├── object_detection.py   # Main Python script for detection
└── requirements.txt      # File listing Python dependencies
```

4.2 Key Code Snippets

Model Loading:

```
net = cv2.dnn.readNet("models/yolov3.weights", "models/yolov3.cfg")
```

Drawing Bounding Boxes:

```
cv2.rectangle(img, (x, y), (x + w, y + h), color, 2)
cv2.putText(img, label, (x, y + 30), font, 2, color, 2)
```

The complete source code is included in the appendix section of this report.

---

4. Implementation

4.1 Project Structure

```
Project_Folder/
├── models/               # YOLOv3 model files (weights, config, class names)
├── images/               # Test images
├── object_detection.py   # Main Python script for object detection
└── requirements.txt      # List of required Python packages (e.g., opencv-
python, numpy)
```

4.2 Code Walkthrough and Explanation

The core detection logic is implemented in `object_detection.py`. Below is a step-by-step breakdown of how the code works and why each part is important.

---

1. Importing Required Libraries

```
import cv2
import numpy as np
```

- **cv2 (OpenCV)** is used for reading images, displaying results, and integrating the YOLOv3 model.
- **numpy** is used for matrix operations and numerical calculations required in post-processing (e.g., bounding box coordinates).

---

2. Loading the YOLOv3 Model

```
❖  def load_model():
❖      weights_path = r"D:\CODES\Python\models\yolov3.weights"
❖      config_path = r"D:\CODES\Python\models\yolov3.cfg"
❖      names_path = r"D:\CODES\Python\models\coco.names"
❖
❖      print(f"Checking paths:\nWeights: {weights_path}\nConfig:
   {config_path}\nNames: {names_path}")
❖
❖      if not os.path.exists(weights_path):
```

```
❖          raise FileNotFoundError(f"Model weights file not found at
   {weights_path}")
❖      if not os.path.exists(config_path):
❖          raise FileNotFoundError(f"Model config file not found at
   {config_path}")
❖      if not os.path.exists(names_path):
❖          raise FileNotFoundError(f"Class names file not found at
   {names_path}")
❖
❖      net = cv2.dnn.readNet(weights_path, config_path)
❖      classes = []
❖      with open(names_path, "r") as f:
❖          classes = [line.strip() for line in f.readlines()]
❖      layer_names = net.getLayerNames()
❖      output_layers = [layer_names[i - 1] for i in
   net.getUnconnectedOutLayers().flatten()]
❖      return net, classes, output_layers
```

❖ **What It Does**:

- Loads the model's **weights**, **architecture configuration**, and **class labels** (from COCO dataset).
- Retrieves only the **output layers** that produce final predictions.

❖ **Why It's Needed**:

YOLOv3 is a deep neural network. To use it, we must load the trained model and identify which layers are responsible for object detection output.

---

3. Reading and Preprocessing the Image

```
 img = cv2.imread(img_path)
    if img is None:
        print("Error: Image not found!")
        return
    height, width, channels = img.shape

    blob = cv2.dnn.blobFromImage(img, 0.00392, (416, 416), (0, 0, 0), True,
crop=False)
    net.setInput(blob)
```

❖ **What It Does**:

- Reads the image from the file system.
- Converts it into a blob (binary large object), which resizes and normalizes the image for YOLO's input requirements.

❖ **Why It's Needed**:

YOLOv3 accepts input in a fixed format: 416x416 pixel RGB image, normalized between 0 and 1. This ensures consistent detection across varying image sizes.

---

4. Running Forward Pass and Getting Predictions

```
outs = net.forward(output_layers)
```

❖ **What It Does**:

- Performs a forward pass through the YOLO network.
- Returns raw detection predictions from the network, which include bounding boxes, confidence scores, and class scores.

❖ **Why It's Needed**:

This is where actual detection happens — the model analyzes the image and tries to predict where and what objects are present.

---

5. Filtering Relevant Detections

```
for detection in out:
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]
            if confidence > 0.5 and class_id in [0, 2, 9]:
```

❖ **What It Does**:

- Extracts confidence and class ID from each detection.

- Filters out low-confidence detections and keeps only cars, pedestrians, and traffic lights.

❖ **Why It's Needed**:

This ensures we only process **relevant** and **accurate** detections. YOLO predicts 80 classes; we only care about 3 of them.

---

6. Calculating Bounding Boxes

```
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)
```

❖ **What It Does**:

- Converts relative coordinates (ranging from 0 to 1) to actual pixel values.
- Calculates top-left corner of each bounding box.

❖ **Why It's Needed**:

YOLO outputs box centers — but to draw them, we need top-left and bottom-right coordinates.

---

7. Applying Non-Maximum Suppression (NMS)

```
indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
```

❖ **What It Does**:

- Suppresses overlapping boxes and keeps the one with the highest confidence.

❖ **Why It's Needed**:

Prevents duplicate boxes being drawn for the same object. Essential in crowded scenes.

8. Drawing Results on the Image

```
for i in indexes.flatten():
            x, y, w, h = boxes[i]
            label = f"{classes[class_ids[i]]} {confidences[i]:.2f}"
            color = colors[class_ids[i]]
            cv2.rectangle(img, (x, y), (x + w, y + h), color, 2)
```

❖ **What It Does**:

- Draws a rectangle around the detected object.
- Displays the object class and confidence percentage above the box.

❖ **Why It's Needed**:

Visualization helps understand model output and verify detection accuracy.

---

9. Displaying the Final Image

```
    cv2.imshow("Object Detection", img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

❖ **What It Does**:

- Opens a window and displays the final image.
- Waits for a key press before closing.

❖ **Why It's Needed**:

Provides user feedback — without this, we wouldn't see the detection results.

4.3 Summary of the Code Pipeline

The current codebase includes safeguards like file existence checks and debug prints to streamline testing and avoid runtime failures.

1. **Model Initialization** → Load pre-trained YOLO model and class labels.
2. **Image Input** → Read the test image.
3. **Preprocessing** → Resize and normalize image to YOLO format.
4. **Prediction** → Run the model and get outputs.
5. **Post-Processing** → Filter relevant detections, apply NMS, and draw results.
6. **Visualization** → Display annotated output image.

This modular design ensures easy extension to real-time detection using video or webcam feeds.

---

5. Results

Input vs. Output Comparison:

Input Image

Observations:

- The model demonstrates high precision in detecting **cars** and **pedestrians**, especially when the images are of high resolution and taken under good lighting.
- **Traffic lights** were sometimes missed, particularly when small or partially obstructed.
- The bounding boxes and class labels were visually clear and accurate in most test cases.

---

6. Challenges

Despite the impressive performance, several challenges were encountered during development and testing:

- **Speed vs. Accuracy Trade-off**: While YOLOv3 is faster than traditional models, running it on machines without GPU support leads to delays.
- **Overlapping Objects**: Dense traffic scenarios or crowded pedestrian zones sometimes result in missed detections due to overlapping bounding boxes.
- **Lighting Conditions and Image Quality**: Night-time images or low-resolution inputs reduced detection accuracy.
- **Model Generalization**: Since YOLOv3 is trained on a general dataset (COCO), it may not perform as well on region-specific or specialized traffic signs and vehicles.

---

7. Future Enhancements

1. **Real-Time Video Detection**: Implementing video stream processing using `cv2.VideoCapture()` to enable detection in live camera feeds.
2. **Custom Training**: Fine-tuning YOLOv3 on custom datasets (e.g., night-time or foggy weather conditions) to improve performance in specific environments.
3. **Advanced Metrics**: Adding performance metrics like **Precision**, **Recall**, and **F1-Score** to better evaluate the model.

4. **Hardware Acceleration**: Deploying on GPU-enabled systems or edge devices (e.g., NVIDIA Jetson Nano) for real-time, low-latency detection.

---

8. Conclusion

This project successfully demonstrates the practical implementation of object detection using OpenCV and YOLOv3. Through a well-defined pipeline, we were able to detect and label cars, pedestrians, and traffic lights with satisfactory accuracy. The approach is efficient, modular, and can be adapted for real-time applications with minor enhancements. However, challenges like processing speed on CPUs and issues with low-quality images suggest room for improvement.

Going forward, integrating this solution with live video processing and custom datasets can elevate it for deployment in smart traffic systems and autonomous driving frameworks. Overall, this project has been a valuable experience in applying deep learning techniques to real-world problems using Python and OpenCV.

---

9. References

1. Redmon, J., & Farhadi, A. (2018). *YOLOv3: An Incremental Improvement*. arXiv:1804.02767
2. OpenCV Documentation: https://docs.opencv.org/
3. COCO Dataset: https://cocodataset.org/

10. Appendix

Source Code

```
import cv2
import numpy as np
import os

def load_model():
    weights_path = r"D:\CODES\Python\models\yolov3.weights"
    config_path = r"D:\CODES\Python\models\yolov3.cfg"
    names_path = r"D:\CODES\Python\models\coco.names"

    print(f"Checking paths:\nWeights: {weights_path}\nConfig:
{config_path}\nNames: {names_path}")

    if not os.path.exists(weights_path):
        raise FileNotFoundError(f"Model weights file not found at
{weights_path}")
    if not os.path.exists(config_path):
        raise FileNotFoundError(f"Model config file not found at {config_path}")
    if not os.path.exists(names_path):
        raise FileNotFoundError(f"Class names file not found at {names_path}")

    net = cv2.dnn.readNet(weights_path, config_path)
    classes = []
    with open(names_path, "r") as f:
        classes = [line.strip() for line in f.readlines()]
    layer_names = net.getLayerNames()
    output_layers = [layer_names[i - 1] for i in
net.getUnconnectedOutLayers().flatten()]
    return net, classes, output_layers

def detect_objects(img_path):
    net, classes, output_layers = load_model()
    img = cv2.imread(img_path)
    if img is None:
        print("Error: Image not found!")
        return
    height, width, channels = img.shape

    blob = cv2.dnn.blobFromImage(img, 0.00392, (416, 416), (0, 0, 0), True,
crop=False)
    net.setInput(blob)
    outs = net.forward(output_layers)
```

```
    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 and class_id in [0, 2, 9]:
                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)

    indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)

    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 = f"{classes[class_ids[i]]} {confidences[i]:.2f}"
            color = colors[class_ids[i]]
            cv2.rectangle(img, (x, y), (x + w, y + h), color, 2)
            cv2.putText(img, label, (x, y + 30), font, 2, color, 2)

    cv2.imshow("Object Detection", img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

if __name__ == "__main__":
    try:
        detect_objects(r"D:\CODES\Python\test1.jpg")
    except FileNotFoundError as e:
        print(e)
```

OUTPUT: