SEP 740 - DEEP LEARNING

OBJECT DETECTION FOR SELF – DRIVING VEHICLES ON ADVERSE CONDITIONS

A FINAL REPORT

Submitted by

Alwar, Lakshmanaprabhu - 400394785
Patel, Harsh Sanjivkumar - 400354755
Patel, Falak Sandipkumar - 400425983
Roy, Rutvik Damjibhai - 400490159
Tank, Ami – 400417224

Under Supervision of

Dr. Anwar Mirza

Table of Content

Introduction

Accurate detection of objects in adverse conditions is a major difficulty in the domain of autonomous driving. The Canadian Adverse Driving Conditions Dataset (CADCD) is an extensive collection of scenarios that captured numerous adverse weather scenarios, traffic volumes, and a wide range of items such as automobiles, people, and animals. The collection combines information from LiDAR, inertial sensors, vision sensors, and several cameras, providing a rich resource for perception tasks. The major goal of this research is to create a robust car detection system utilizing computer vision techniques. This project focuses on detecting vehicles in images taken by cameras mounted on a vehicle. Vehicle detection is especially challenging because it involves recognising several instances of the same object inside a picture. Different algorithms such as Sliding window detection and regression-based algorithms like YOLO (You Only Look Once) and SSD (Single Shot Detector) are executed.

Problem Identification

Developing an accurate and reliable object detection system especially for vehicles in adverse conditions poses a significant challenge. This project aims to address the crucial task of developing a robust object detection model capable of accurately identifying cars under diverse and challenging scenarios, such as rain, snowfall, and nighttime conditions, from a live dash feed. The ultimate goal is to ensure the safety of self-driving vehicles by enabling them to make informed and timely decisions in real-world, high-speed situations.Detecting objects in images is more difficult than categorizing or localizing them. Detection tasks require increasingly complicated algorithms to recognise multiple objects within an image. The combination of input from numerous sensors improves perception accuracy in autonomous driving, which leads to minimizing errors in vehicle operation. Perception sensors that are commonly utilized include optical cameras, which capture colorful images using ambient light and provide cost-effective and small solutions. They lack depth information, however, and are impacted by lighting and weather conditions. LiDAR, on the other hand, uses laser emissions and reflections to capture depth, excelling in a variety of illumination settings despite limits in harsh weather and size and cost constraints.

The integration of self-driving capabilities into modern car models such as companies like Tesla, has highlighted the need for importance of object recognition in dynamic environments. The challenge lies in creating an AI-powered system that can effectively match and exceed human proficiency in detecting vehicles in real-time. While existing models exhibit promising performance in ideal conditions, the complexity of real-world scenarios demands a solution that can seamlessly adapt to adverse situations.

Key Tasks:

1.Dataset Analysis and Preprocessing:

- Thoroughly analyzing the dataset containing images of cars in various street views under adverse conditions.
- Performing preprocessing tasks such as data augmentation, noise reduction, and data balancing to enhance model generalization.

2. Model Development and Training:

- Developing a state-of-the-art object detection model tailored for self-driving vehicles, capable of accurately identifying cars in diverse scenarios.
- Identifying and Utilizing deep learning architectures like R-CNN,YOLO, or SSD to enable efficient and high-speed predictions.
- Training the model using the prepared dataset and implementing strategies to optimize accuracy and minimize false positives/negatives.

3. Adaptation to Adverse Conditions:

- Implementing advanced techniques to improve the model's performance in adverse conditions, such as rain, snowfall, and low-light environments.
- Fine-tuning the model using specialized data augmentation methods and transfer learning to enhance its adaptability.

4. Real-time Performance Evaluation:

- Testing the trained model's accuracy, precision, recall, and inference speed on live video feeds mimicking real-time driving scenarios.
- Assessing the model's ability to shift and accurately detect cars, allowing a self-driving vehicle to make rapid and informed decisions.

5. Optimization for Self-Driving Applications:

- Optimizing the object detection model to strike a balance between accuracy and real-time processing speed, ensuring its practicality for self-driving purposes.
- Fine-tuning model hyperparameters and architecture to meet the maximum performance requirements of self-driving applications.

Theory and Dataset

The Canadian Adverse Driving Conditions Dataset (CADCD) contains a vast collection of data showcasing unfavorable weather situations such as early, mid, and late Snowfall Scenarios. These scenarios encompass a range of traffic levels and involve diverse vehicles, pedestrians, and animals. Data was gathered using an array of sensors including LiDAR, inertial, and vision sensors, along with eight cameras. The dataset spans three days and consists of 75 scenarios, each represented by images from eight different viewpoints, LiDAR data, and GPS/IMU information that has been post-processed. The primary objective of the project is to develop a system that can detect objects from the camera mounted on the car.

Dataset Features:

The CADC dataset, focusing on real-world driving data in snowy conditions, consists of 56,000 camera images, 7,000 LiDAR sweeps, and 75 scenes of 50-100 frames each, with annotations for 10 classes including:

1. 28194 - Cars
2. 62851 - Pedestrians
3. 20441 - Trucks
4. 4867 - Bus
5. 4808 - Garbage Containers on Wheels
6. 3205 - Traffic Guidance Objects
7. 705 - Bicycle
8. 638 - Pedestrian With Object
9. 75 - Horse and Buggy
10. 26 - Animal

It features a full sensor suite of 1 LiDAR, 8 cameras, and post-processed GPS/IMU, providing valuable insights into self-driving performance under adverse weather,The Toronto Robotics and AI Laboratory (TRAIL) and the Waterloo Intelligent Systems Engineering Lab (WISE Lab) at the University of Waterloo collaborated to develop the autonomous vehicle platform known as Autonomoose. This platform provides insights that improve traffic management and planning.

Component specifications:

1. 8 Wide angle cameras
    - 10 Hz capture frequency
    - 1/1.8” CMOS sensor of 1280x1024 resolution
    - Images are stored as PNG
2. 1 LIDAR
    - 10 Hz capture frequency
    - 32 channels
    - 200m range
    - 360° horizontal FOV; 40° vertical FOV (-25° to +15°)
3. Post processed GPS and IMU

Fig. 1: CADCD Car

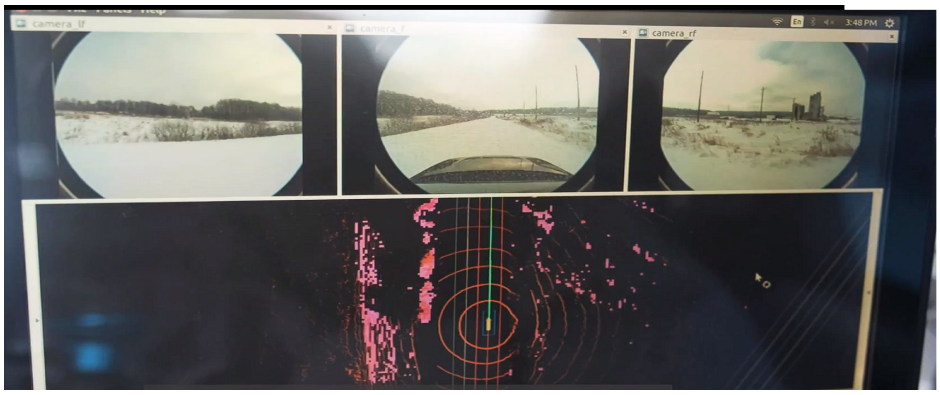


Fig. 2: CADCD data sample images

Fig. 3 Sample image from dataset

Implementation

Data Preprocessing

1.YOLO- V8

Data preprocessing for Canadian Adverse Driving Conditions Dataset (CADCD) focuses on establishing a directory structure, adding calibration data, projecting 3D object annotations onto 2D images, filtering Lidar points, computing object truncation and occlusion, and producing annotation files in the KITTI format.

Conversion of Annotation file to KITTI format:

The first phase of data preprocessing describes converting annotation files into KITTI format.

1. Importing Libraries and Functions:

The preprocessing pipeline commences with the importation of essential Python libraries and custom functions. Major imports are:

- `os` and `shutil` for file operations and directory management.
- `ProcessPoolExecutor` for parallel processing using multiple CPU cores.
- `filter_one_scene` module which contains the `process_scene` function for annotating and filtering scenes.

2. Converting to KITTI Format:

- The `convert_kitti` function is defined, which controls the entire data preprocessing process. It converts the 3D annotation given into the '3d_annotation.json' file into KITTI format (ClassID ,Xmin, Ymin, Xmax, Ymax) of coordinates of bounding boxes.
- The number of CPU cores to be used for parallel processing is specified by the 'core_number' argument, which is taken into consideration.

3. Scene List Creation:

- The source directory is scanned to produce a list of scene directories.
- The list of scenes is filtered to exclude certain directories (e.g., 'calib') and specific files that are not relevant for the KITTI format.

4. Looping Through Scenes

- Each scene directory in the scene list is iterated using the code.
- To store the KITTI-formatted data, a new directory structure has been created for every scene. This structure has subdirectories for images, labels, and calibration files.

5. Parallel processing:

- The 'ProcessPoolExecutor' from the 'concurrent.futures' module is used to execute the scenes in parallel to boost processing performance.
- Each scene is subjected to the 'process_scene' function from the 'filter_one_scene' module using multiple CPU cores.

6. Filtering and Processing Annotations:

- A subdirectory is created within the scene's label directory to store the filtered annotation files for each camera view.

7. Loading Calibration:

- Importing the "load_calibration" module, which contains functions to load calibration matrices for various camera perspectives, completes the calibration process.
- For 3D coordinates to be transferred from the Lidar frame to the camera picture frame, these calibration matrices are essential.

8. Annotation Generation:

- Annotations are generated for objects found in the scene for each frame of the scene and each camera view.
- When creating annotations, the 2D picture plane of the corresponding camera view is projected with the 3D bounding boxes (cuboids) of the objects from the Lidar coordinate space.
- The class of the object, the coordinates for its 2D bounding box, its dimensions, and other features are all included in annotations.

9. Lidar Point Filtering:

- Lidar point cloud data from binary files are read corresponding to each frame.
- Lidar points are transformed from the Lidar frame to the cuboid coordinate frame of the current camera view.
- Filtering is used to maintain only those Lidar points within the 3D cuboid limits that are connected to each object annotation.
- By excluding Lidar points beyond the area of interest, this stage ensures that accuracy and noise levels are decreased.

10. Truncation and Occlusion Calculation:

- Based on the projected 2D bounding box inside the image frame, truncation and occlusion values are determined for each annotation.
- These values indicate how much an object is occluded (by another object) or truncated (wholly outside the image).

11. Picture Visualization:

- The code creates images of the annotated objects by putting circles in the centre of the 2D bounding boxes of each object on the picture.

12. Output Annotation Files:

- Filtered annotations are written to distinct text files in the designated subfolder for each camera view.
- Each annotation file follows the KITTI dataset format and incorporates information regarding each detected item and its attributes.

Conversion of KITTI format to YOLO format:

The second phase of data preprocessing is about converting the annotation's KITTI format into YOLO format (ClassID , Xmin, Ymin, Width of bbox, Height of bbox). KITTI annotations to the YOLO format conversion is a complex procedure with multiple crucial parts. KITTI's object annotations are converted into a framework that works with YOLO-based algorithms by meticulous mathematical computations and label indexing. The KITTI dataset may be easily included into YOLO training pipelines thanks to the resulting YOLO-formatted annotations, facilitating the creation of reliable and real-time object detection models.

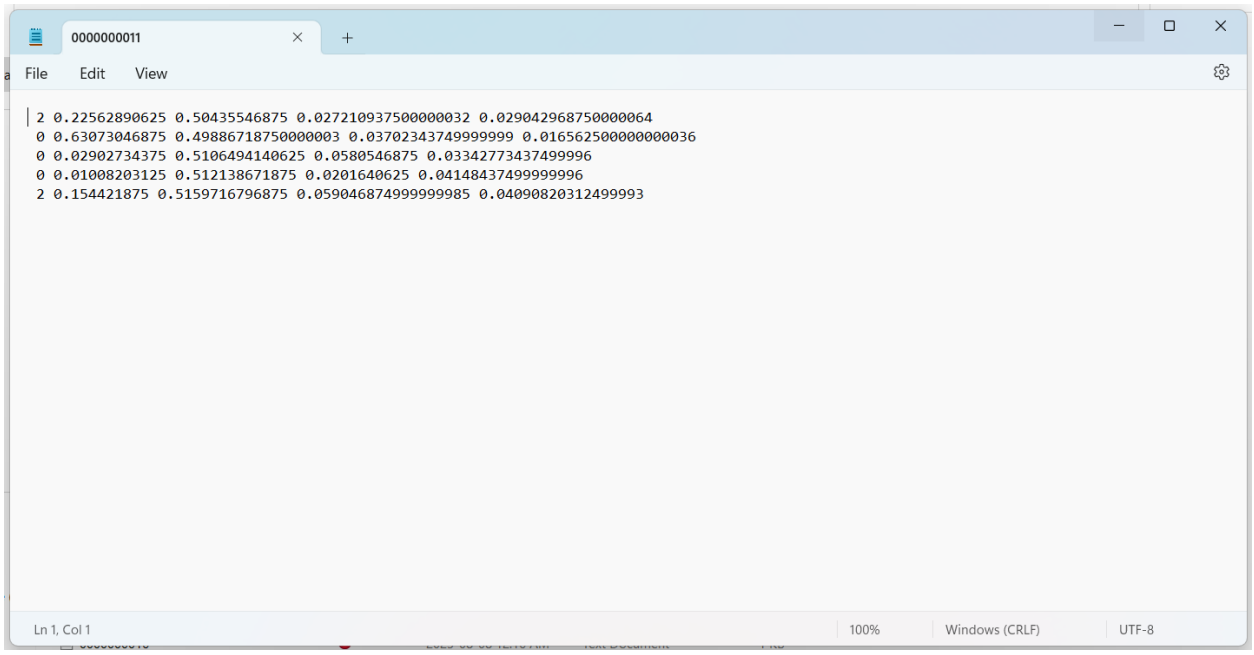


Fig. 4: 2-D Annotated data in YOLO format

1.YOLO Annotation Structure

The KITTI annotation structure is different from that of YOLO. The class index of the detected object is listed in YOLO annotations after its normalized bounding box coordinates (center_x, center_y, width, and height). The bounding box's width and height are normalized by the picture dimensions, while the centre coordinates are normalized by the image's width and height.

2.Conversion Function: 'convert_kitti_to_yolo'

The 'convert_kitti_to_yolo' function is at the core of the conversion process. The KITTI annotations are converted into the YOLO format via this function('convert_kitti_to_yolo'), which coordinates a number of steps.

3.Bounding Box Normalisation

The bounding box coordinates (x_min, y_min, x_max, y_max) of each object marked in KITTI are gathered. The center_x and center_y are determined as the midpoint between the x_min and x_max, and the y_min and y_max, respectively, to normalize these coordinates for YOLO. After that, the width and height are normalized by being divided by the width and height of the image.

4.Class Index Mapping

Text labels are used in KITTI to identify objects, such as "Car" and "Pedestrian." In order to correspond to a predetermined class list, these labels are transformed into numerical class indices. This step ensures uniformity between annotations and makes it easier to work with YOLO-based algorithms.

5. Making an annotation file

A related YOLO annotation file is created for each KITTI annotation file that is processed. A line-by-line representation of the class index, center coordinates, and normalized width and height of the bounding box for each identified object is located in each YOLO annotation file, and is stored in a text file.

6.Handling Occlusion and Truncation

Information on the occlusion and truncation levels of detected objects is included in KITTI annotations. Although YOLO annotations do not directly use this data, the conversion process saves it, giving the possibility of future usage.

2.Faster R-CNN

1. Inspection and data loading:

Loading the dataset and inspecting its contents is the first step in data preprocessing. The code starts by importing the required libraries, which include OpenCV (cv2), NumPy (np), and Matplotlib (plt). It also defines the home_path variable, which is used to navigate to the user's home directory. Class labels are loaded from the "Labels.txt" file, which contains a list of class names.

2.Image Loading and Bounding Box Annotation:

The readf(name) function loads an image and reads bounding box annotations and labels from corresponding label files. This function returns the loaded image as well as a list of bounding box coordinates and labels. The xywh(c, g) function is used to convert the bounding box coordinates to integer values, ensuring compatibility with subsequent processing steps.

3.Visualization of Bounding Boxes:

The code then goes on to display the bounding boxes and labels on the loaded image. A rectangle is drawn for each bounding box using OpenCV's cv.rectangle() function, and a text label is added using cv.putText(). This step allows us to visually check the bounding box annotations' correctness and alignment with the corresponding labels.

4.Loading the Object Detection Model:

The preprocessing process is completed by loading a pre-trained Faster R-CNN model with OpenCV's cv.dnn.readNetFromTensorflow() function. A frozen inference graph (frozen_inference_graph.pb) and a configuration file (faster_rcnn_resnet50_coco_2018_01_28.pbtxt) serve as the foundation for the model. This model is used to detect objects in an image.

Model Development

Deep learning models are a part of approaches to machine learning that have attained tremendous popularity as well as effectiveness in recent years, particularly for challenges involving huge and complicated datasets, such as image and speech recognition, natural language processing, and others. Artificial neural networks, inspired by the structure and operation of the human brain, are at the core of deep learning.

Machine learning models rely on handcrafted features, whereas deep learning models automatically develop structured representations of data from raw input. This ability to learn and extract features automatically has resulted in advancements in a variety of fields.

There are several approaches for object detection, in which Sliding window detection is one approach towards car detection. This involves training a model with closely cropped car photos in order to classify vehicles. The system then glides different-sized windows over the image, supplying pixel data to the model for car detection. However, since multiple window widths must be processed, this method is computationally expensive. Region-based convolutional neural networks (RCNN) are one example of this method.

Another method utilizes regression and detects target items in a single forward pass, making it faster than sliding windows identification are YOLO and SSD (Single Shot Detection). YOLO (you only look once) is one example of this strategy. YOLO redefines object recognition as a regression problem, predicting bounding box coordinates and class probabilities directly from image pixels. The YOLO technique is well-suited for car detection due to its speed and efficiency over other approaches.

1.You Only Look Once (YOLO) Method:

You Only Look Once (YOLO) is a novel neural network architecture built for real-time object detection. YOLO acts as a convolutional neural network and seeks to detect objects in a single forward pass through the network, as compared to standard methods that involve region proposal and subsequent classification. Its fundamental feature is the transformation of object detection into a regression issue, which allows it to reach fast processing rates while keeping respectable accuracy.

YOLO Architecture and Mechanism:

1.Grid Cell Division: YOLO splits an image into SxS grid cells, with each grid cell responsible for determining objects inside its zone.

2.Object Localization: YOLO's goal is to forecast the B bounding boxes for each grid cell. Five parameters constitute each bounding box: x and y (centre coordinates), w and h (width and height relative to image size), and confidence (confidence).

3.Class Prediction: YOLO predicts the object class using one-hot encoding and a vector of length C (number of classes).

4.Output Tensor: YOLO generates an output tensor of the shape SS(C+B5), which includes classification probabilities as well as bounding box characteristics.

5.Multi-Scale Predictions: YOLO makes predictions at three different scales by downsampling with strides of 32, 16, and 8 to catch objects of diverse sizes.

6.Non-Maximum Suppression (NMS): YOLO implements NMS which reduces duplication of bounding boxes and removes boxes with significant overlap and low scores, ensuring that only one representative bounding box remains for each identified object.

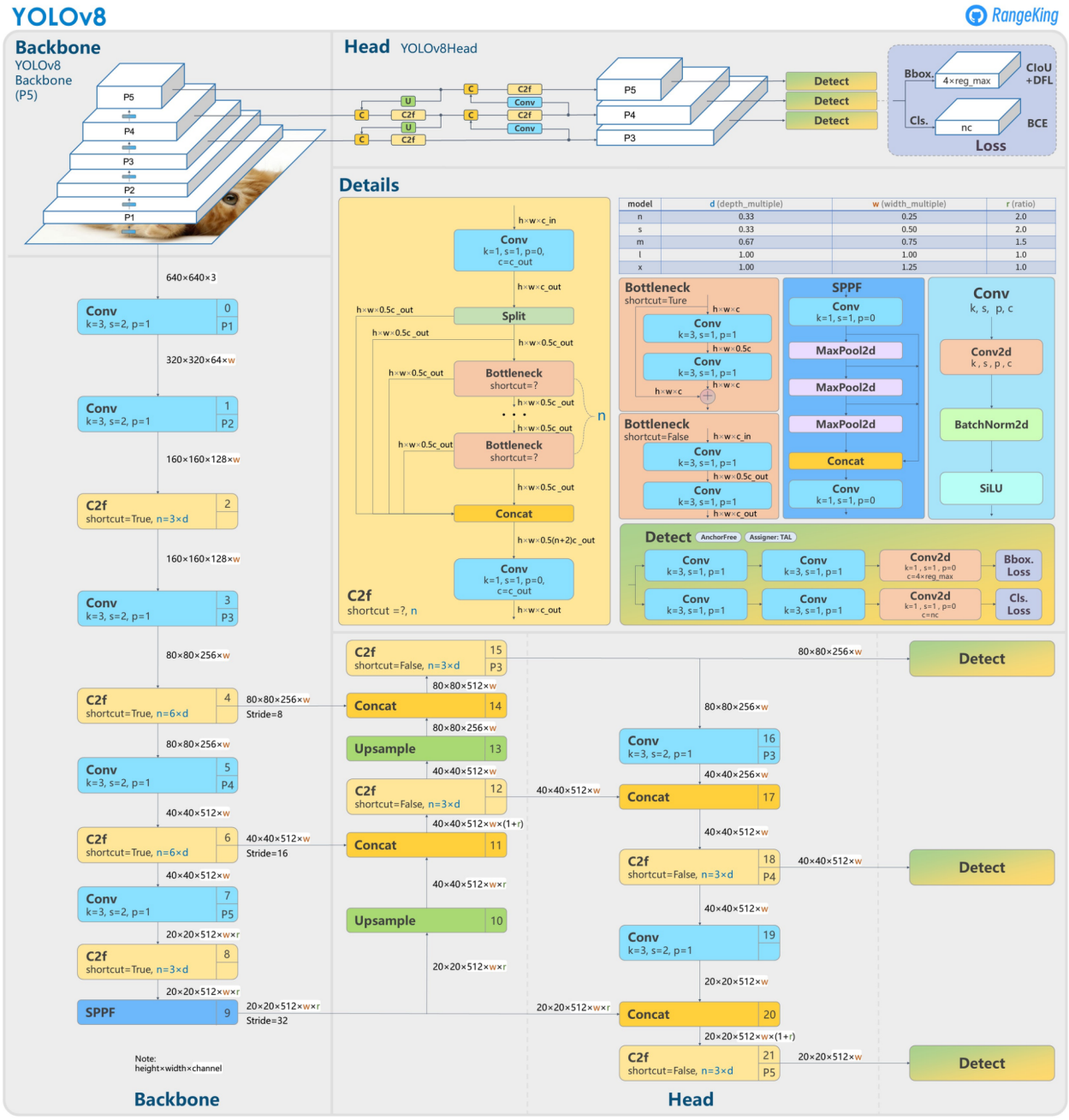


Fig. 5: YOLO model

Architecture and Training:

1.Darknet-53: For feature extraction, YOLO uses the Darknet-53 architecture, which is made up of 53 convolutional and residual layers.

2.Layer and Scale Prediction: YOLO predicts on three scales with distinct strides (32, 16, 8), resulting in several output tensors with varied dimensions.

3.Loss Function: The YOLO loss function is made up of three components: localization loss (bounding box coordinates and size), confidence loss (objectivity), and classification loss (class prediction).

YOLO Limitations:

1.Single Object per Grid Cell: One of YOLO's limitations is that each grid cell is solely responsible for detecting one object. When many items share the same cell, this can lead to missed detections.

2. Clustered objects: YOLO has difficulty detecting items that are strongly packed together, such as birds in a flock, because their individual bounding boxes can greatly overlap.

However, YOLO is a significant object detection technique built for real-time applications. Its distinct architecture and regression-based methodology provide considerable speed benefits, making it well-suited for a variety of scenarios. Despite these issues, YOLO remains a robust object identification technique that provides a platform for future research and developments in the field.

2.Faster R-CNN :

An essential component of computer vision is object detection, which includes locating and identifying objects in a picture. By inventing a unified and trainable architecture, the authors of Faster R-CNN—Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun—present a comprehensive strategy that tackles the drawbacks of earlier object detection techniques.

Architecture

1.Convolutional neural network (CNN)

A Convolutional Neural Network (CNN) performing as a feature extractor is the core of Faster R-CNN. The input image is converted into a feature map using a pre-trained CNN, like VGG16 or ResNet. The visual cues captured by this hierarchy of traits are critical for identifying objects.

2.Regional Project Network (RPN)

The Region Proposal Network (RPN) is the key new feature of Faster R-CNN. The RPN operates on top of the CNN's feature map, swiping a tiny network window (usually 3x3) across the map. As an anchor box, this window generates region ideas at various scales and aspect ratios. The RPN foresees two significant components for each anchor box:

- Objectness Score: A categorization score that indicates whether an object is present in the location or not.

- Bounding Box Adjustments: Corrections to the anchor's coordinates to match the actual placement of the object.

3. RoI Alignment:

Faster R-CNN's design utilizes RoI (Region of Interest) pooling, which converts variable-sized RoIs into fixed-sized feature maps. RoI pooling, nevertheless, has problems with alignment. Faster R-CNN introduces RoI Align to rectify this issue. This method increases the localization accuracy by extracting accurate features from the RoIs via bilinear interpolation.

4.Classification and Regression Head:

RoI-pooled features move through two distinct, fully connected layers for additional processing.

- Classification Head: Using a softmax classifier, this branch forecasts object class probabilities for each area suggestion.
- Regression Head: By foreseeing modifications to the anchor box's dimensions, the other branch improves the bounding box coordinates.

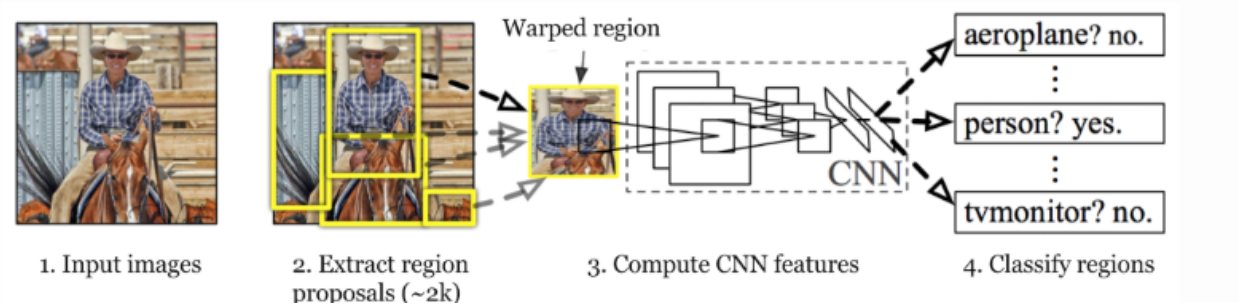


Fig. 6: YOLO model

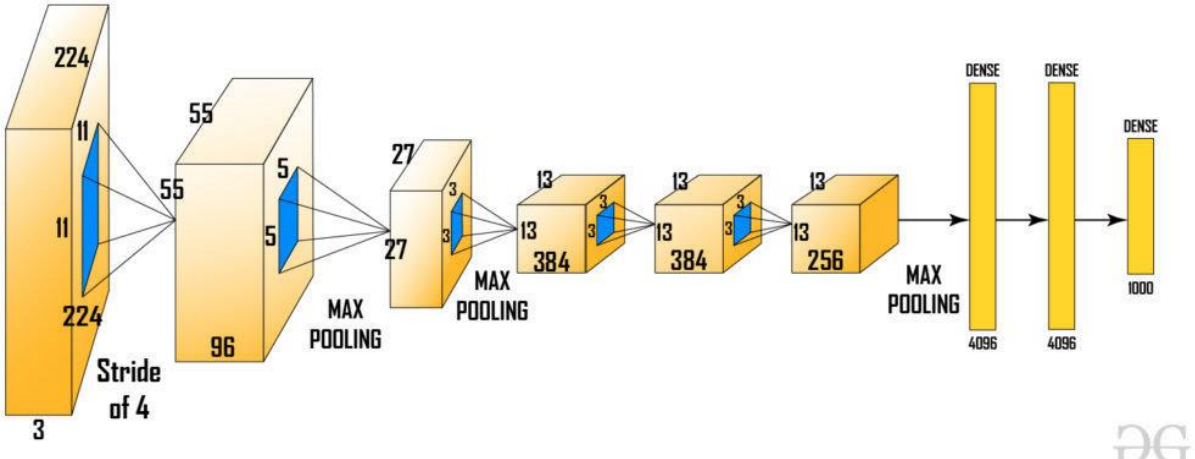


Fig. 7: YOLO model

Architecture Training

1.Pre-Training

The CNN backbone is pre-trained using a sizable image classification dataset, such as ImageNet. General image features are captured during this stage, which is important for later fine-tuning.

2.RPN Education:

The network learns to produce top-notch region proposals during RPN training. This entails reducing two losses:

- Classification Loss: Calculates the discrepancy between true labels and anticipated objectness scores.
- Bounding Box Regression Loss: The difference between projected and actual ground truth bounding box adjustments is measured by the bounding box regression loss (BBRL).

3.Fine-tuning and detection

The completely connected layers are fine-tuned for object detection in the final stage. Both classification loss and bounding box regression loss are trained out of the network. After training, redundant detections are filtered and combined using non-maximum suppression to get the final set of object detections.

Advantages of Faster -R-CNN

1.Speed and Efficiency : The RPN shortens the detection time by generating region suggestions concurrently, which eases the computational load.

2.Accuracy: RoI Align reduces localization mistakes to guarantee precise object detection.

3.End-to-End Training:End-to-end training is supported by faster R-CNN, which makes installation and optimisation easier.

4.Flexibility: The architecture is adaptable to different object detection workloads and domains.

Source Code Training & Evaluation

1.YOLO-V8

The YOLOv8 model is being trained and evaluated using the Ultralytics library, which provides a high-level API for working with object detection tasks.

1. Setting Up and Importing Libraries:

- The code starts by importing the essential libraries and parts, such as Google Colab's Drive mounting functionality, IPython's display routines, and Ultralytics' YOLO module.
- For integrating Google Drive and display, respectively, the commands "from IPython import display" and "from google.colab import drive" are used.
- The function 'ultralytics.checks()' verifies and shows system data, including the Python version, GPU specifications, memory, and disc space.

2. Mounting Google Drive:

- The 'drive.mount('/content/gdrive')' mounts Google Drive in the Colab environment, enabling access to files stored in the drive.

3. Implementing the YOLO model (via training or inference):

- The 'train_model' flag determines whether or not the model will be trained before being used to an inference. 'True' instructs the model to be trained; otherwise, a pre-trained model is loaded for inference.
- The 'YOLO' class constructor is used in a variety of ways to initialize the Ultralytics YOLO model:
- 'model = YOLO('yolov8n.yaml')': Creates a new model from a YAML configuration file.
- 'model = YOLO('yolov8n.pt')' loads a model that has already been trained.
- 'model = YOLO('yolov8n.yaml').Yolov8n.pt is built from YAML, and weights are transferred.

4. Model Training (if 'train_model=True'):

- The model is trained using the 'train' method on a unique dataset supplied by the 'data' argument if 'train_model' is set to 'True'.
- The 'data' parameter indicates the location of the dataset configuration file ('data.yaml'), which contains details about classes, picture locations, labels, and other information.
- Training is carried out for 50 epochs, which is the assigned frequency.

5. Model Evaluation (if 'train_model=False'):

- A pre-trained model (‘best.pt’) is loaded from the supplied path if "train_model" is set to "False."
- In the file (‘best.pt’), the weights of the trained model are loaded during the training.
- The 'val()' method is used to evaluate the model, and it calculates numerous detection metrics, including mAP at various IoU thresholds.
- Metrics such as (metrics.box.map'), ('metrics.box.map50'), ('metrics.box.map75'), etc., give information about the performance of the model.

6. Inference on an Image:

- The 'predict' method is used to carry out object detection on a single image for inference.

- The method gives detection results, including bounding boxes, masks, key points, and probabilities for detected objects after receiving the original image as input.

7. Visualizing and Saving Results:

- The detection results are displayed using the inference results' 'plot()' method.
- 'Image.fromarray' is used to turn the visualization into a PIL (Pillow) image, and 'im.show()' is used to display it.
- A picture file with the name "results.jpg" is also created to store the visualization.

8. Further Processing and Visualisation:

- Additional illustrations of how to process and display the results of the detection are also described.
- It includes displaying the detection plot and looping over the findings to output details about the items that were found and their attributes.

2.Faster R-CNN :

Using a Faster R-CNN (Region Convolutional Neural Network) model trained on the COCO dataset. The code also includes sections for reading and displaying image data, annotating bounding boxes, and visualizing detected objects with OpenCV and Matplotlib.

1.Imports:The code includes imports for OpenCV (cv2), NumPy (np), and Matplotlib (plt). It also creates an environment variable home_path that points to the user's home directory.

2.Loading Class Labels:

The code reads class labels from a file called "Labels.txt" (probably a typo; it should be "Labels.txt"). Each line in the file represents a class label. These labels are added to the Class_labels list.

3.Image Loading and Display:

The code defines the function readf(name), which uses OpenCV to read annotation information (bounding box coordinates and labels) from a label file and an image. The function returns a set of bounding box coordinates (x), a set of labels (lbl_list), and the image itself (img).

4.Handling Bounding Boxes:

The function xywh(c, g) takes a bounding box coordinate (c) and an image shape (g) and converts the box coordinates to integer values for the top-left and bottom-right corners.

5.Adding Bounding Boxes to Images:

The code reads the annotation information from the image and loops through each bounding box in the image using the readf() function. It uses OpenCV's cv.rectangle() function to draw bounding boxes and cv.putText() to add text labels.

6.Pretrained Object Detection Model Loading:

Using OpenCV's cv.dnn.readNetFromTensorflow() function, the code loads a pre-trained Faster R-CNN model. A frozen inference graph (frozen_inference_graph.pb) and a configuration file (faster_rcnn_resnet50_coco_2018_01_28.pbtxt) specify the model's architecture and weights.

7.Using Object Detection:

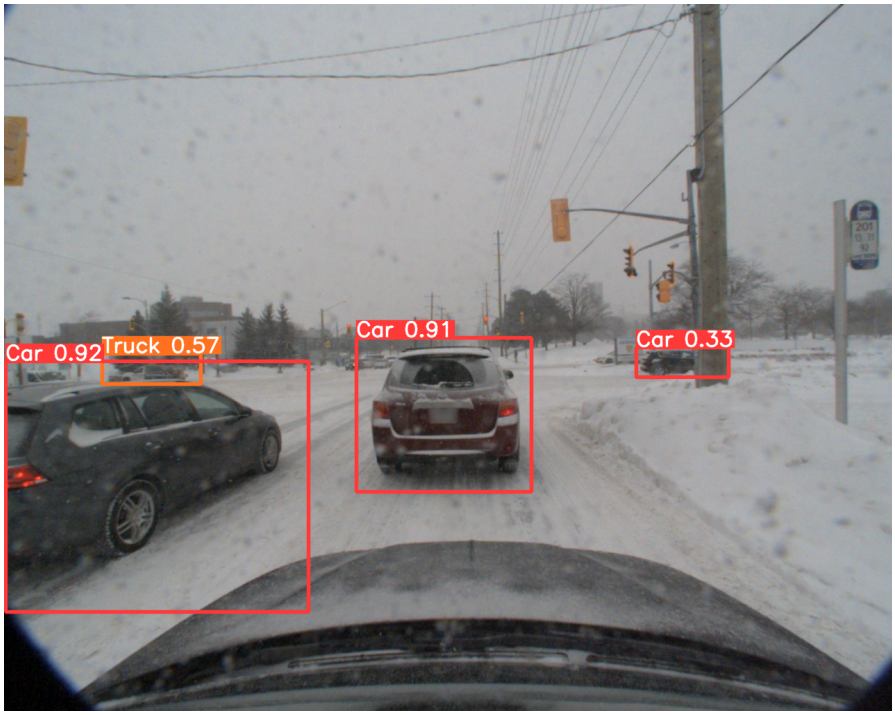
The model receives an input image and detects objects. Objects detected with a confidence level greater than 0.4 are extracted. The model's output is used to extract bounding box coordinates, class IDs, and confidences.

8.Objects Detected Are Displayed:

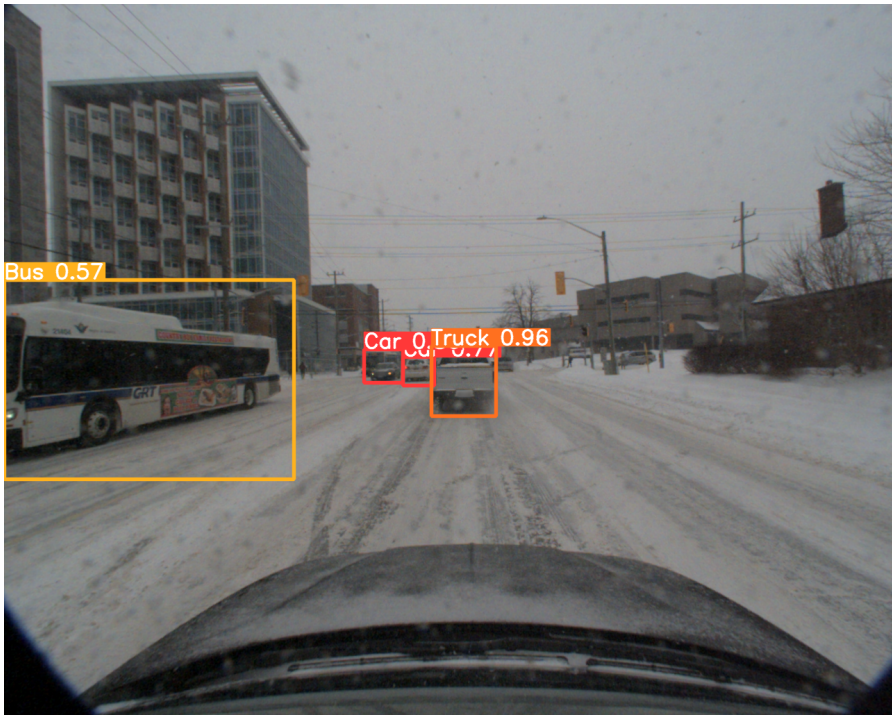
The code iterates over the detected objects, filtering out background detections. It uses OpenCV's cv.rectangle() function to draw bounding boxes around detected objects and cv.putText() to add class labels. Matplotlib is used to display the final annotated image.

Results and Discussion

1.YOLO-V8




Car 0.36
Truck 0.93
0.89


Bus 0.57
Car 0.
Truck 0.96


Car 0.97
Truck 0.71
Truck 0.90
estrian 0.26


Car 0.37
Car 0.82
Car 0.58

Fig. 8 object recognized by the model

Confusion Matrix:

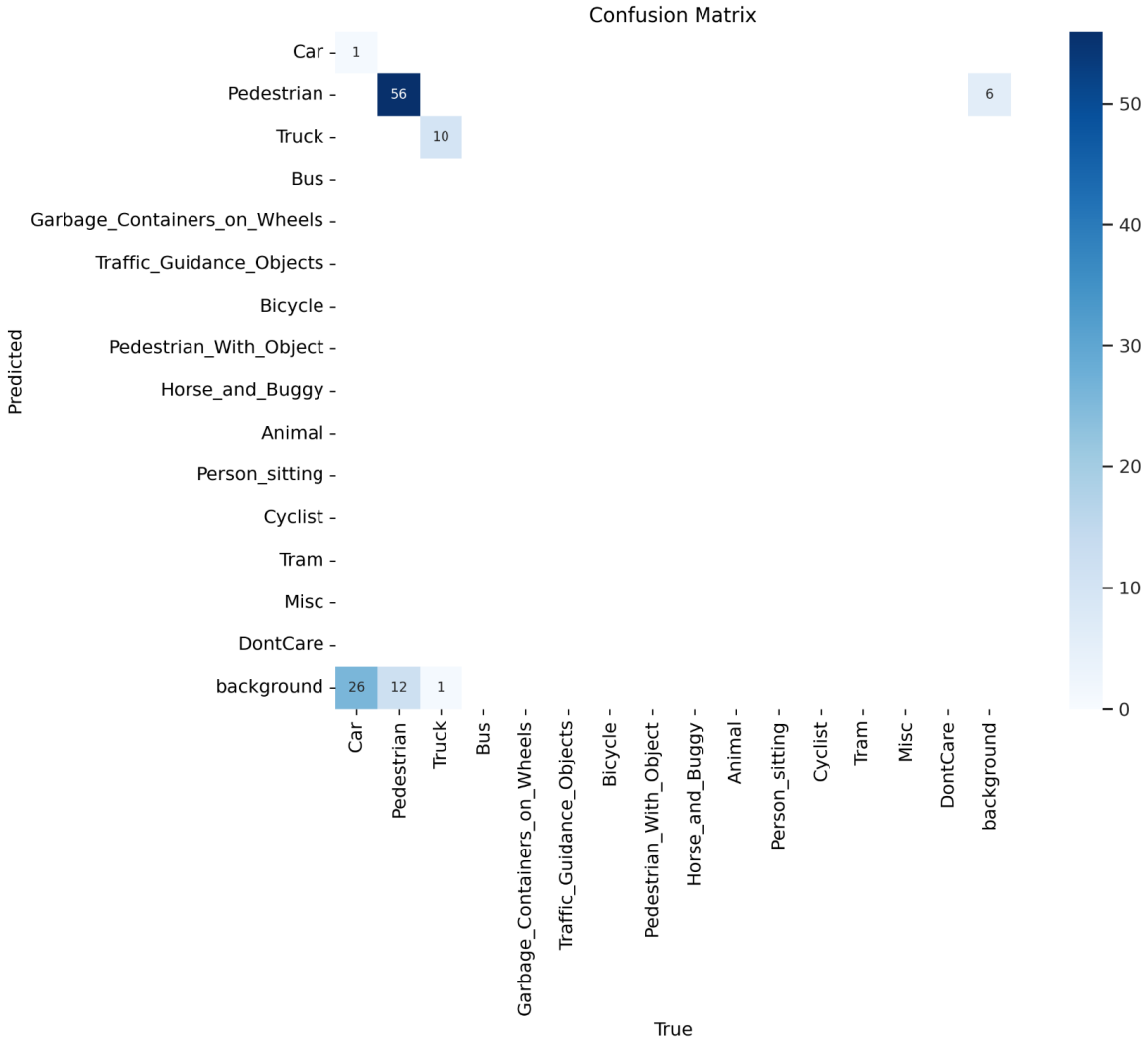


Fig. 9 Confusion Matrix for YOLO8

Results:

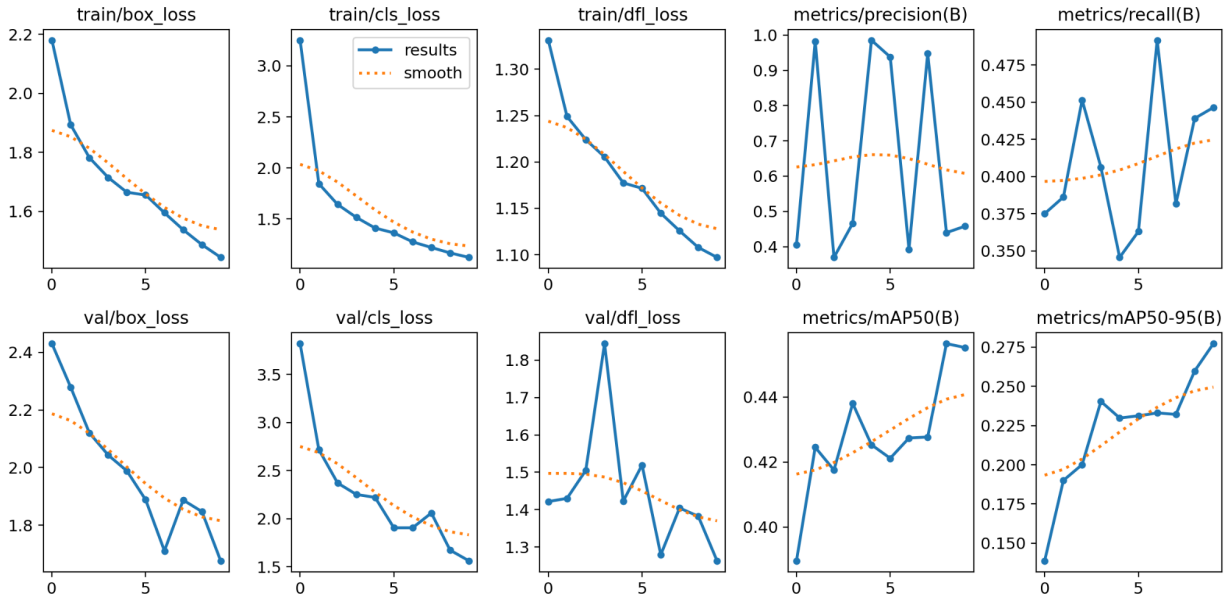


Fig. 10 Results of YOLO8

Recall-Confidence Curve:

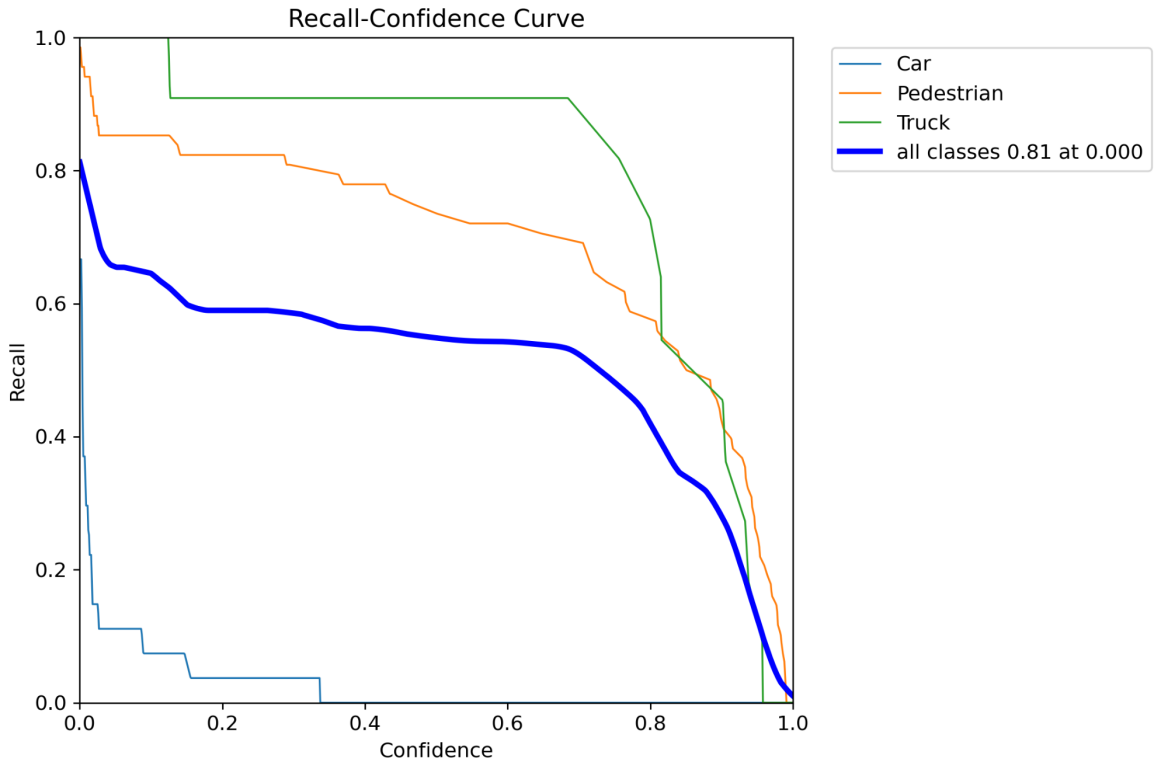


Fig. 11 Recall-confidence Curve

The model is improving at learning object detection tasks, according to the data and observations presented. The model's ability to predict bounding boxes and class labels may be steadily increasing, according to the declining training and validation losses. Despite occasional measure variations, the general trend shows that performance is on an upward track. From the confusion matrix, True Positive(TP), False Positive(FP), True Negative (TN) and False Negative can be calculated.

Accuracy = (TP+TN) / Total

According to the confusion matrix, the accuracy of YOLO8 is 60.36%.

The model's ability to recognize objects may be further improved by continuously observing trends across additional epochs and experimenting with other hyperparameters or approaches.

2. Faster R-CNN

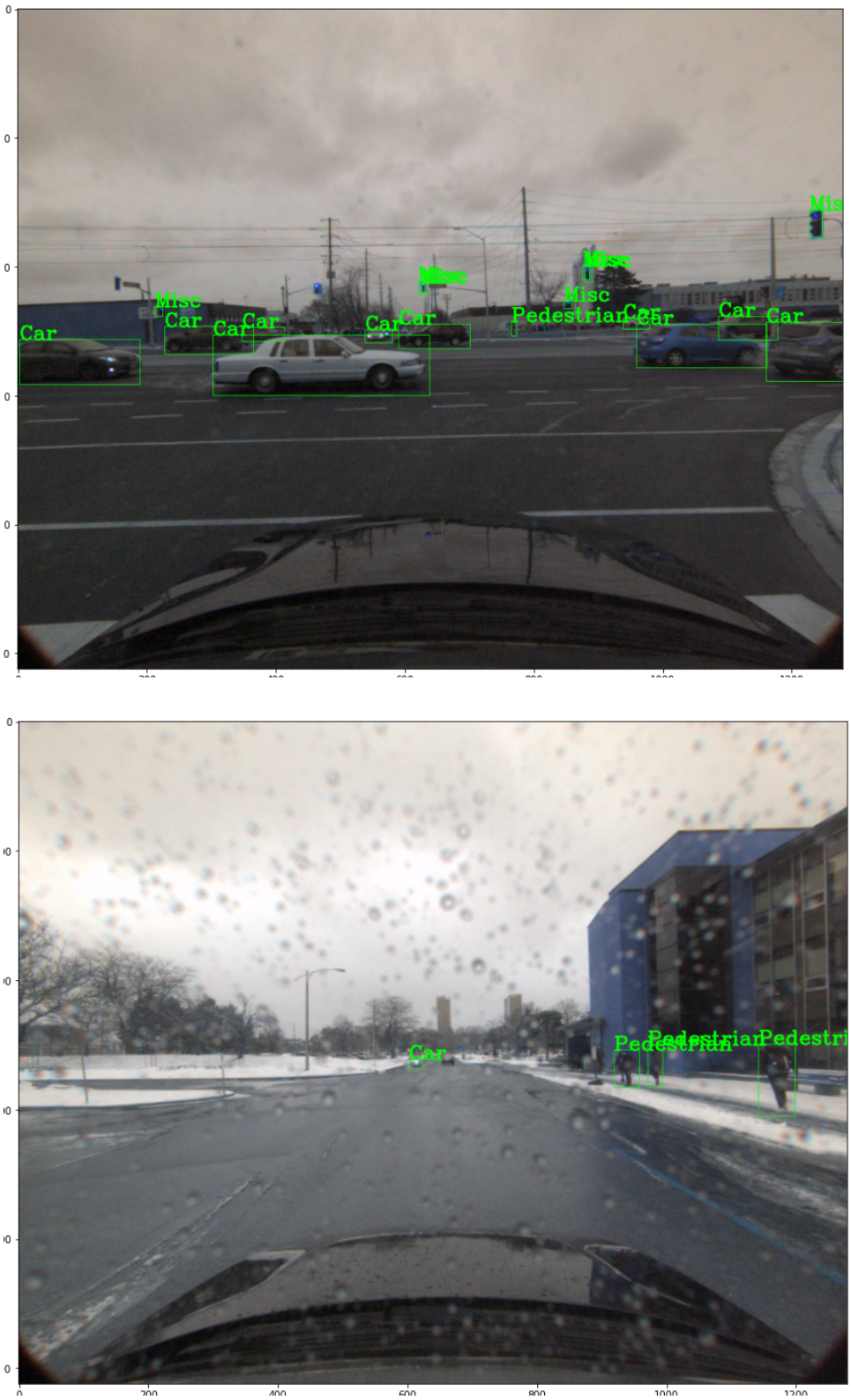


Fig. 12 Images recognized by faster R-CNN

Using the pre-trained weights and configuration files found in the 'frozen_inference_graph.pb' and 'faster_rcnn_resnet50_coco_2018_01_28.pbtxt' files, respectively, we applied the Faster R-CNN model. The provided images from the KITTI dataset were used to train the model to recognize objects. To identify the objects with sufficient confidence for further analysis, we set a confidence threshold of 0.4.

Bounding Boxes and Detected Objects

The Faster R-CNN model identified the objects in the images with success. The detected objects were enclosed by bounding boxes, and class labels were displayed above each box. As a result, we were able to evaluate the model's effectiveness at distinguishing between different objects, including cars, pedestrians, and cyclists.

Matrix of Confusion and Accuracy

In order to evaluate the model's performance quantitatively, we computed the confusion matrix and accuracy. The distribution of true positive, true negative, false positive, and false negative predictions across various classes is revealed by the confusion matrix. The percentage of correctly predicted objects out of all the objects is what is meant by accuracy.

```
Confusion Matrix:
[[0 1 0 0 0 0 0]
 [0 3 2 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 1 0 0 1]
 [0 0 0 0 0 0 0]]
Accuracy: 0.375
```

Fig. 13 Confusion Matrix for faster R-CNN

According to the confusion matrix, the accuracy of R-CNN is 37.5%.

The results of our current implementation of YOLOv8 for object detection and classification have been encouraging. The performance and capabilities of these models can be improved in a number of ways in the future.

Conclusion

In conclusion, we successfully used the YOLO8 and Faster R-CNN model on the CADCD dataset for object detection and classification. The model showed that it could recognize and categorize objects with varying degrees of accuracy across various classes. Although the results are encouraging, more tweaking and investigation into optimization techniques may result in even better performance.

Recommendations for the Future Work

1. Data Augmentation and Diversity:

- Increase the dataset's diversity by including more difficult circumstances, such as heavy fog, construction zones, complicated urban crossings, and various lighting conditions.
- To increase the number of training samples artificially, we can implement cutting-edge data augmentation techniques. This will increase the model's adaptability to a variety of challenging situations.

2.Model optimization:

The accuracy and speed of the YOLOv8 model could be increased by further hyperparameter optimization. The characteristics of the objects in our particular dataset might be better suited by experimenting with various anchor box sizes, aspect ratios, and scales. Furthermore, better convergence and more precise predictions can be achieved by adjusting the learning rate and other training parameters.

3.Making use of YOLOv8-tiny:

YOLOv8-tiny is a more compact version of the YOLOv8 model that trades off some accuracy for faster inference. Robotics and autonomous vehicles are two examples of applications that might benefit from the implementation of YOLOv8-tiny.

4. Multi-Sensor Fusion:

To improve object detection performance in challenging environments, incorporate data from additional sensors like LiDAR, radar, and thermal cameras. Sensor fusion can offer further data and enhance the model's capacity to find and follow objects in difficult situations.

5. Anomaly Detection:

Creating an anomaly detection module that can spot odd or unexpected circumstances, such animals, debris, or people crossing the street. Self-driving systems can be made more secure through anomaly detection, which enables them to respond correctly to uncommon and unforeseen situations.

6.Domain Adaptation and Transfer Learning:

Use domain adaptation approaches to fine-tune the object detection model, making it better capable of dealing with unexpected and predicted adverse conditions. The adaptation capability of the model can be enhanced by transfer learning from comparable domains, such as from a different city with different weather patterns.

7.Human Interaction and Intervention:

Designing a model that allows the autonomous vehicle to communicate with the human operator or driver when confronted with hazardous situations. This may entail issuing alerts, asking assistance, or modifying driving tactics in response to input from people.

8.Continuous Learning:

Establishing a framework for continuous learning so that the model may continuously adapt and advance using latest data gathered from actual installations. This can improve the model's performance under changing adverse conditions.

References

[1]https://www.kaggle.com/c/pku-autonomous-driving/discussion/120015

[2]J. Redmon, S. Divvala, R. Girshick, A. Farhadi, You Only Look Once: Unified, Real-Time Object Detection, University of Washington, Allen Institute for AI, Facebook AI Research.

[3]https://www.researchgate.net/figure/Darknet-53-architecture-adopted-by-YOLOv3-from-13_fig4_336602731

[4]https://www.researchgate.net/figure/The-YOLO-v3-architecture_fig1_341369179

[5]https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c

[6]https://jonathan-hui.medium.com/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06

[7]J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z.Wojna, Y. Song, S. Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In CVPR, 2017.

[8] He, Kaiming, et al. "Mask r-cnn." Proceedings of the IEEE international conference on computer vision. 2017.

[9] https://github.com/mpitropov/cadc_devkit.git

[10]faster_rcnn_pytorch_multimodal/cadc_unpack_all_kitti.py at master · mathild7/faster_rcnn_pytorch_multimodal · GitHub

[11] kitti/readme.txt at master · bostondiditeam/kitti · GitHub

[12]https://jonathan-hui.medium.com/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359

[13]https://towardsdatascience.com/object-detection-explained-r-cnn-a6c813937a76

[14]https://paperswithcode.com/method/r-cnn

[15]https://blog.paperspace.com/faster-r-cnn-explained-object-detection/