Object detection for Self - driving Vehicles on adverse conditions

Ami Tank
Masters of Engineering at Systems and Technology
McMaster University
Hamilton, Canada

Falak Sandipkumar patel
Masters of Engineering at Systems and Technology
McMaster University
Hamilton, Canada

Harsh Sanjivkumar patel
Masters of Engineering at Systems and Technology
McMaster University
Hamilton, Canada

Lakshmanaprabhu Alwar
Masters of Engineering at Systems and Technology
McMaster University
Hamilton, Canada

Rutvik Damjibhai, Roy
Masters of Engineering at Systems and Technology
McMaster University
Hamilton, Canada

***Abstract*—The issue of correctly recognizing objects in tough driving environments is covered in this abstract. The Canadian Adverse Driving Conditions Dataset (CADCD) provides a thorough assortment of varied conditions, including bad weather and traffic, gathered by several sensors and cameras. With an emphasis on automobiles in camera photos, the study's objective is to build a powerful vehicle recognition system utilizing computer vision techniques. Multiple instances of the same item must be identified, a problem that may be solved with algorithms like Sliding Window, YOLO, and SSD.**

Keywords—object detection, self-driving cars, YOLO, RCNN.

I. Introduction (*Heading 1*)

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.

II. 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. A complete collection of data containing unfavorable weather conditions, including numerous snowfall scenarios, is available in the Canadian Adverse Driving Conditions Dataset (CADCD). Different types of traffic, automobiles, people, and animals are all represented in this dataset. It was gathered over three days with 75 situations utilizing a variety of sensors, including LiDAR, inertial, vision, and eight cameras. The main objective of the project is to create a camera-based object identification system for automobiles.

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.

III. 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:

- 28194 - Cars
- 62851 - Pedestrians
- 20441 - Trucks
- 4867 - Bus
- 4808 - Garbage Containers on Wheels
- 3205 - Traffic Guidance Objects
- 705 - Bicycle
- 638 - Pedestrian With Object
- 75 - Horse and Buggy
- 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.

IV. Data Preprocessing

A. 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 initial data preprocessing phase involves converting annotations to KITTI format. Here's a concise overview of the process:

  1. Imports: Necessary Python libraries and functions are imported, including os, shutil, and parallel processing tools.
  2. Conversion: Annotations from '3d_annotation.json' are transformed into KITTI format, specifying bounding box coordinates.
  3. Scene List: Scene directories are gathered, excluding unnecessary folders and files.
  4. Processing Scenes: Each scene is processed separately, creating a structured directory for images, labels, and calibration.
  5. Parallel Processing: Scenes are processed in parallel for efficiency.
  6. Annotations: Filtered annotations are stored in subdirectories within scene labels.
  7. Calibration: Calibration matrices are imported for coordinate transformation.
  8. Annotation Generation: Annotations are created for objects in frames and camera views.
  9. Lidar Filtering: Relevant Lidar points are retained, reducing noise.
  10. Truncation and Occlusion: Values are calculated to indicate truncation and occlusion.
  11. Image Visualization: Annotated object images are generated.
  12. Output: Filtered annotations are saved in KITTI format in designated subfolders.

  In essence, the process involves converting and structuring annotations, generating KITTI-formatted data, and optimizing for object detection in self-driving scenarios.

- Conversion of KITTI format to YOLO format:

The second data preprocessing phase involves converting KITTI annotations to YOLO format (ClassID, Xmin, Ymin, Width of box, Height of box). This conversion is intricate and critical, enabling seamless integration of KITTI dataset into YOLO training pipelines for real-time object detection.

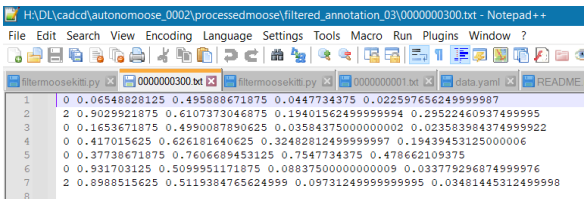

fig.1 YOLO format annotated data

Key steps in YOLO format annotation:

1. YOLO Annotation Structure: YOLO format differs from KITTI. YOLO annotations include normalized bounding box coordinates (center_x, center_y, width, height), preceded by the object's class index. Normalization involves dividing box dimensions by image size.
2. Conversion Function: The core conversion process is handled by the 'convert_kitti_to_yolo' function, orchestrating various steps.
3. Bounding Box Normalization: KITTI's x_min, y_min, x_max, y_max coordinates are normalized. Center coordinates are calculated as midpoints and normalized. Width and height are divided by image dimensions.
4. Class Index Mapping: KITTI's text labels (e.g., "Car," "Pedestrian") are transformed into numerical class indices, ensuring alignment with YOLO's class list.
5. Annotation File Creation: YOLO annotation files are generated for each processed KITTI annotation. These files contain class index, normalized coordinates, and dimensions of bounding boxes, stored in text format.
6. Handling Occlusion and Truncation: KITTI's occlusion and truncation data is preserved during conversion, even though YOLO doesn't directly use it.

This phase enables smooth compatibility between KITTI and YOLO, streamlining the creation of robust and real-time object detection models.

B. 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.

V. Model Development

Deep learning models, powered by artificial neural networks inspired by the human brain, have gained popularity for tasks involving complex datasets like image recognition and natural language processing. They automatically learn data representations, leading to advancements in various fields.

Object detection methods include Sliding Window and YOLO/SSD. Sliding Window involves training on cropped car images and sliding windows for classification, but it's computationally expensive. YOLO (You Only Look Once) and SSD (Single Shot Detection) are faster. YOLO redefines object detection as regression, predicting bounding box coordinates and class probabilities directly from pixels, making it efficient for car detection.

1. YOLO Method:

- YOLO is a neural network for real-time object detection.
- It transforms detection into a regression problem for speed and accuracy.
- YOLO divides images into grid cells, predicts bounding boxes and class probabilities.
- Multi-scale predictions and Non-Maximum Suppression enhance accuracy.

2. Faster R-CNN:

- Faster R-CNN improves object detection with a CNN backbone and Region Proposal Network (RPN).
- RPN generates region proposals concurrently, reducing computational load.
- RoI Align rectifies alignment issues in RoI pooling.
- Fine-tuning and end-to-end training improve detection accuracy and efficiency.

VI. Source Code Training & Evaluation

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.

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.

VII. Results and Discussion

A. YOLO8:

a) 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.

b) Accuracy = (TP+TN) / Total

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

d) 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.

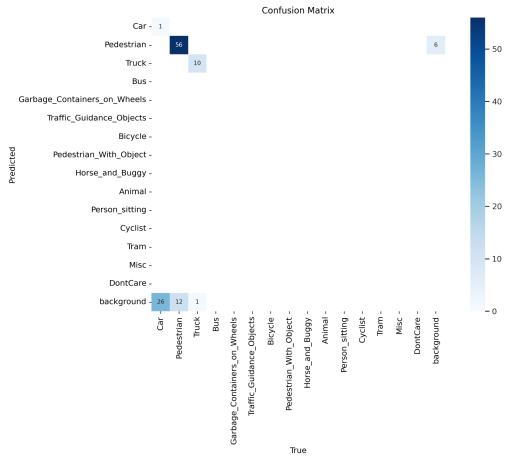

fig. 2. Confusion matrix of YOLO8

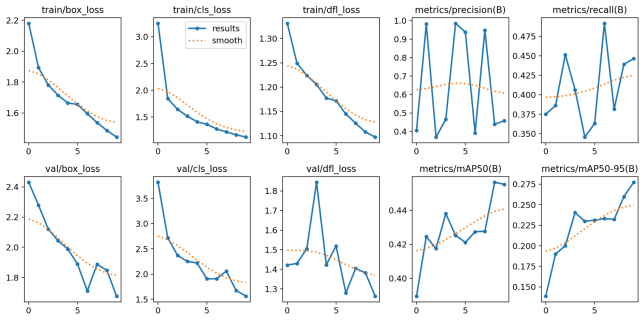

fig. 3. Results of YOLO8

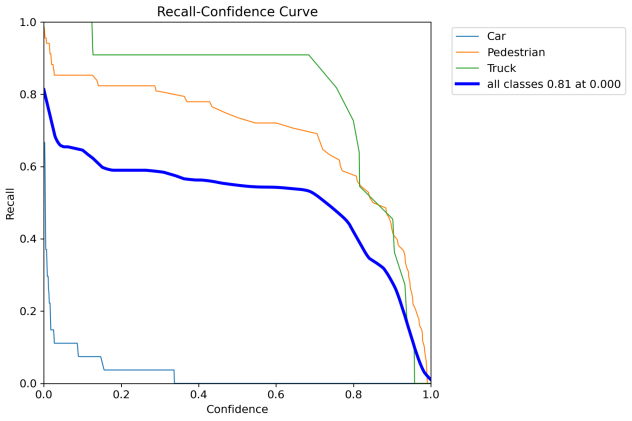

fig. 4. Recall curve of YOLO8

B. 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.

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.

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.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 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.

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] https://towardsdatascience.com/ssd-single-shot-detector-for-object-detection-using-multibox-1818603644ca

[8] https://towardsdatascience.com/review-ssd-single-shot-detector-object-detection-851a94607d11

[9] 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.

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

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

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

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

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

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

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

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