Introduction

This project utilizes a fine-tuned YOLOv8 object detection model for detecting vehicles in autonomous driving systems. Leveraging transfer learning from a pre-trained YOLOv8 backbone, the model was adapted to the KITTI 2D Object Detection dataset. It achieves a mean Average Precision (mAP@0.5) of 0.938, demonstrating high accuracy in detecting vehicles and related classes, making it suitable for real-world autonomous driving applications.

Table of contents

Repository

Metrics --> Includes the visual representations

Model --> The final training model

Scripts --> Some codes used for multiple reasons

Testing Videos --> Real Time testing videos

Metrics.csv --> Primary metrics resource

Documentation

Dataset

The KITTI 2D Object Detection dataset was used for this project. As a benchmark in autonomous driving research, it provides a robust and well-annotated foundation, making it an ideal choice for fine-tuning and evaluating object detection models.

Class Categories:

The model was trained to detect the following eight object classes, along with their corresponding instance counts in the dataset:

Car – 22,949 instances

Pedestrian – 3,649 instances

Cyclist – 1,313 instances

Truck – 898 instances

Van – 2,383 instances

Person_sitting – 182 instances

Tram – 427 instances

Misc (Miscellaneous) – 802 instances

These instance counts highlight a significant class imbalance within the dataset, which—as discussed later in the results section—has a notable impact on the model's performance across different object categories.

How is the data split?

The dataset contains a total of 7,481 images along with their corresponding labels, but no separate validation set is provided. Additionally, 7,518 testing images in PNG format are available, but these do not include labels. To obtain evaluation metrics for model predictions on the test set, developers must upload their results to the KITTI server, which provides metrics only to researchers with "novel ideas." Consequently, the dataset was split into training and validation sets using a stratified sampling method.

How the train data was split to Val and train?

The training data was split into train and Val folders using the Stratified method.

So, in order to get 80/20 split:

Train ≈ 5984 images

Val ≈ 1497 images

Total = 7481 images

The **Stratified** sampling method works by splitting the data into two sets on the image level. so for example, the car object is the dominate class in 5000 (example) images of 7481 images so we specify 80% for the training and 20% of the validation same for each class we decide what is the dominate class in each image for the 7481 images then we split them 80/20.

The initial implemented Stratified sampling –object level Stratified - was to count the number of each object across all the 7481 image so for example we get 10000 car objects in total and 5000 pedestrians in total. If splitted 80/20 into each separate obj category it will cause data leakages.

Splitting the data manually could have caused one object to be overrepresented in the Val data for example, which would have lowered the model's accuracy in return.

The Stratified split code is available in the GitHub repository with comments for clarification.

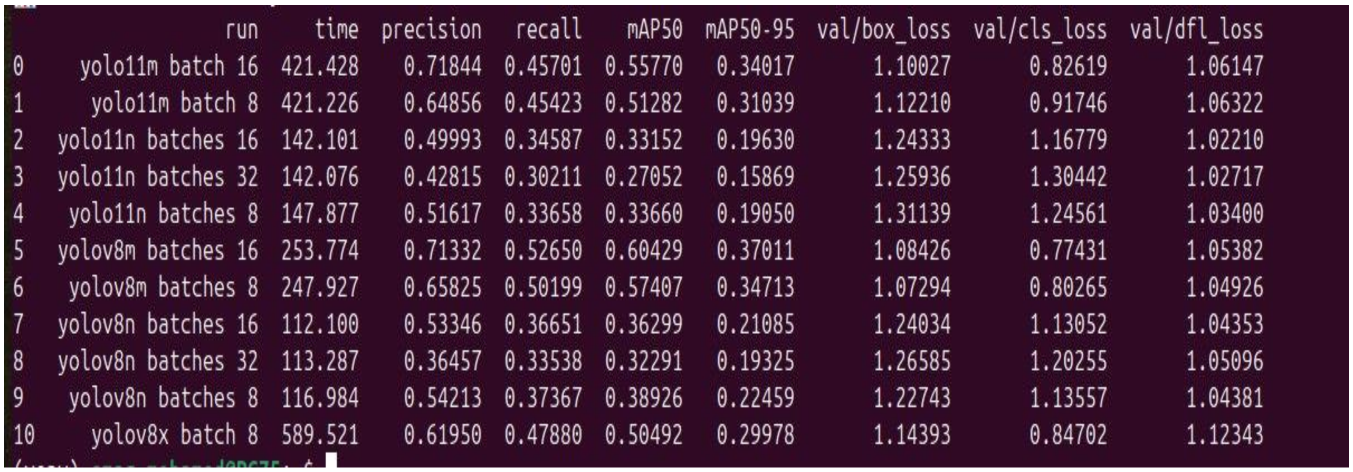
Transfer learning model

YOLO

YOLO is a pretrained object detection model; most yolo models are trained on the COCO dataset. YOLO has different versions, the most famous are 5, 8 and 11. Each version has smaller different models YOLOv8n YOLOv8s YOLOv8m YOLOv8l YOLOv8x. They differ in performance and resources needed.

YOLO model used

The model used in this project is YOLOv8m as it provides the best performance in consideration of the time / resources needed.



```
               run     time  precision    recall     mAP50  mAP50-95  val/box_loss  val/cls_loss  val/dfl_loss
0    yolo11m batch 16  421.428    0.71844  0.45701  0.55770   0.34017       1.10027       0.82619       1.06147
1     yolo11m batch 8  421.226    0.64856  0.45423  0.51282   0.31039       1.12210       0.91746       1.06322
2  yolo11n batches 16  142.101    0.49993  0.34587  0.33152   0.19630       1.24333       1.16779       1.02210
3  yolo11n batches 32  142.076    0.42815  0.30211  0.27052   0.15869       1.25936       1.30442       1.02717
4   yolo11n batches 8  147.877    0.51617  0.33658  0.33660   0.19050       1.31139       1.24561       1.03400
5  yolov8m batches 16  253.774    0.71332  0.52650  0.60429   0.37011       1.08426       0.77431       1.05382
6   yolov8m batches 8  247.927    0.65825  0.50199  0.57407   0.34713       1.07294       0.80265       1.04926
7  yolov8n batches 16  112.100    0.53346  0.36651  0.36299   0.21085       1.24034       1.13052       1.04353
8  yolov8n batches 32  113.287    0.36457  0.33538  0.32291   0.19325       1.26585       1.20255       1.05096
9   yolov8n batches 8  116.984    0.54213  0.37367  0.38926   0.22459       1.22743       1.13557       1.04381
10    yolov8x batch 8  589.521    0.61950  0.47880  0.50492   0.29978       1.14393       0.84702       1.12343
```

This is the output in the terminal when running the **compare results.py** code on the csv files generated during training. It was essential in determining what model to use and in parameter tunning phase.

Training

The train command used:

yolo detect train

model= yolov8m.pt

data="/home/omar-mohamed/ML_Project/kitti training/kitti.yaml"

epochs=100

imgsz=640

fraction=1

amp=False

workers=2

batch=16

device=0

project=/home/omar-mohamed/runs/detect

name=continue_to_epoch100

resume=True

Explanation of Hyperparameters used:

- **model** → Specifies the pre-trained or custom model to use.
- **epochs** → Number of iterations over the full training dataset.
- **imgsz** → Image size; increasing it usually improves performance but consumes more resources.
- **fraction** → Fraction of the dataset used during training for sanity checks or initial evaluations. For example, `0.5` runs on 50% of the data; `1` uses the full dataset.
- **amp** → Automatic Mixed Precision. Intended to increase speed without reducing model performance. Set to `False` because training crashed when enabled, likely due to GPU compatibility.
- **workers** → Number of CPU threads used to load data from storage to GPU. Increasing this can improve data throughput.
- **batch** → Number of images processed simultaneously by the model.
- **device** → Specifies which GPU to use for training.
- **resume** → Optional flag to continue training from an earlier checkpoint meaning it can be used for training the same model on separate periods. This is extremely useful when the model is trained locally.

The model was trained using the advantage of the “resume=True” hyperparameter as it allowed for training the model on a 100 epoch without constantly using the machine for several hours

The process is simple as YOLO automatically saves the weights and “simple metrics in a csv file” after each epoch so if the training was interrupted unexpectedly or by the user using the ctrl + c command the training can still be “resumed” from the last completed epoch using the “above command”

Checking for Overfitting and Underfitting

Overfitting and underfitting are major challenges in training AI models and must be monitored carefully. Excessive training epochs can increase the risk of overfitting, where the model performs well on the training data but poorly on unseen data. YOLO mitigates this by automatically saving two sets of weights after training: **LAST** and **BEST**. The **LAST** weights correspond to the final state of the model at the end of training, regardless of performance. The **BEST** weights represent the model that achieved the optimal performance during training, as determined automatically by YOLO based on evaluation metrics such as mean Average Precision (mAP). While the **BEST** model may coincide with the **LAST** model, this is not always the case.

Results

There are several metrics used to evaluate the performance of a model. The gold metric is mAP@0.5. It is the primary metric for object detection. The final model achieved a 0.938 mAP@0.5.

Per-Class Average Precision (AP@0.5):

- Car: 0.980
- Truck: 0.988
- Tram: 0.972
- Van: 0.978
- Misc: 0.960
- Cyclist: 0.929
- Pedestrian: 0.875
- Person_sitting: 0.825

The model performs exceptionally well on large, distinct vehicle classes like **Car, Truck, and Tram**. The slightly lower performance on **Pedestrian** and **Person sitting** is common and can be attributed to their smaller size, greater occlusion, and more varied poses in the dataset.

The mAP@0.5 is not the only metric as evaluation of the Confidence Threshold is important.

Precision-Confidence

The curve shows that precision remains very high (>0.90) across most confidence levels, reaching a maximum of 1.00. This indicates that when the model is highly confident, it is almost always correct.

Recall-Confidence

Recall starts high at lower confidence thresholds, with the "all classes" recall peaking at 0.96 when the confidence threshold is near zero. This is expected, as a lower threshold allows the model to detect more objects, including those it is less certain about.

F1-Confidence

The F1-score, which balances Precision and Recall, peaks at 0.91 for all classes at a confidence threshold of 0.451. This suggests that 0.45 is an optimal default confidence threshold for this model to balance false positives and false negatives.

Confusion Matrix

The normalized confusion matrix provides a detailed breakdown of the model's classification errors.

High Diagonal Values:

The matrix shows strong diagonal dominance, with values such as **0.97 for Car** and **0.99 for Truck**, confirming high per-class classification accuracy.

Primary Misclassifications:

The most significant confusion occurs between **Pedestrians** and the **background**, which is a typical challenge.

There is minor confusion between structurally similar classes, such as **Car/Van** and **Cyclist/Pedestrian**.

The **Person_sitting** class has the lowest recall (as seen in the PR curve), which is often due to its similarity to the Pedestrian class and its relatively low occurrence in training data.

The metrics.csv file serves as the primary source of performance metrics generated during training. Note that the **time** column may appear inconsistent due to the use of the *resume* feature, as explained earlier.

In addition to the CSV logs, visual representations of key evaluation metrics—such as mAP, F1-score, and the confusion matrix—are also produced to help monitor the model's performance throughout training.

Conclusion

The fine-tuned object detection model has achieved state-of-the-art performance on the KITTI dataset. With a final mAP@0.5 of **93.8%**, it is a highly accurate and reliable model for detecting the eight primary classes of interest. The analysis confirms that it performs best on vehicle classes and reasonably well on vulnerable road users, with a recommended operational confidence threshold of **0.45** to maximize the F1-score.

Recomindations

The KITTI dataset is great, but it is still quite small, especially regarding other classes like pedestrians. Moreover, it obviously lacks night, different weather and diverse road type images. A better model would be trained on multiple datasets or one larger dataset in order to achieve that. A better model would be able to detect pedestrians with less mistakes.

Scripts

The scripts used were uploaded on the GitHub repository below is names of the files and what they do

Test ROCm and pytorch.py

Test that pytorch and ROCm are working installed

stratified split.py

Used to do the stratified sampling as explained above.

compare results.py

Before starting the training, multiple tests were conducted in order to determine which YOLO model works best for this project. The code runs by the following command in the terminal

```
python3 /home/omar-mohamed/ML_Project/"comparing kitti.py" /home/omar-mohamed/runs/detect/kitti_train_test
```

The code searches for all csv files in the folder and compares them

add missing labels.py

YOLO will not create a label for any test image that it couldn’t find any objects in. Therefore, this code was used to fill in the empty files in the results folder.

External links

KITTI Dataset: **https://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=2d**

YOLO Documintation: **https://docs.ultralytics.com/**

PyTorch ROCm: **https://pytorch.org/get-started/locally/**