CSC420: Introduction to Image Understanding

Project 2: Autonomous driving explorations

Project Report

Students:
Laura Madrid Lucas
Noritomi-Hartwig

Submitted to:
Professor Sanja Fidler &
Teaching Assistants

April 20 , 2024

Please refer to our repository: https://github.com/Laura05010/autonomous_driving

Introduction

Autonomous driving is currently a focal point of research, receiving significant attention from both academia and industry. This project offers an opportunity to delve into key challenges within this field by working with stereo image pairs and camera parameters, alongside a dataset of annotated images containing 2D bounding boxes and viewpoint annotations for cars, as well as annotations for road segments.

Repository

Please note that this public repository has been created for you to track our complete progress, including the data used for training the semantic segmentation, object detection, and viewpoint detection models. You can access the repository at:
https://github.com/Laura05010/autonomous_driving

Distribution of Group Responsibilities

Our group believes in the importance of everyone contributing to all aspects of the project to maximize learning opportunities. However, we also recognize the need for task managers who can oversee delegation and organization to ensure smooth and timely completion of project milestones. Initially, we both aimed to fully participate in every part of the project, and we did so until we encountered challenges with computing depth and training the road classifier. At that point, Lucas took over the road classifier task while Laura proceeded with implementing tasks independent of road segmentation. Throughout this period, both members continued to consult each other on model parameters and features, despite working on different tasks. The following table illustrates the distribution of tasks:

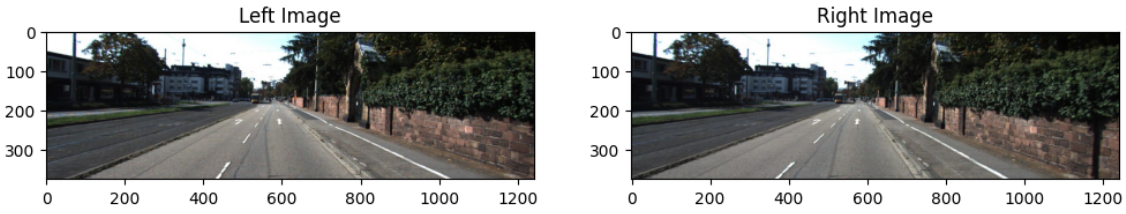
Task	**Manager**
Compute Disparity	Laura & Lucas
Compute Depth and 3D location	Lucas
Train Road classifier	Lucas
Train Car Detection Model	Laura
General Model Architecture Decisions	Laura & Lucas
General Model Hyperparameter Tuning	Laura & Lucas
Train viewpoint classifier	Laura
Final Report	Laura & Lucas

Part 1: Road semantic segmentation

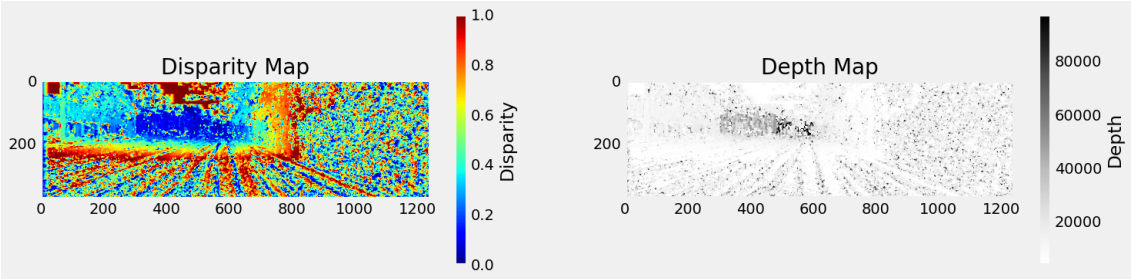
Computing disparity, depth, and 3D location

To implement the disparity computation, we referred to our assignment 4 solution of outlining an implementation. We also referred to a tutorial detailing an implementation in MATLAB McCormick2014 which we followed to implement in Python. The implementation in MATLAB used sum of absolute differences (SAD) which was switched our for sum of squared differences (SSD) in our implementation.

The following is a selected pair of corresponding left and right images from the dataset: first admissable sample):

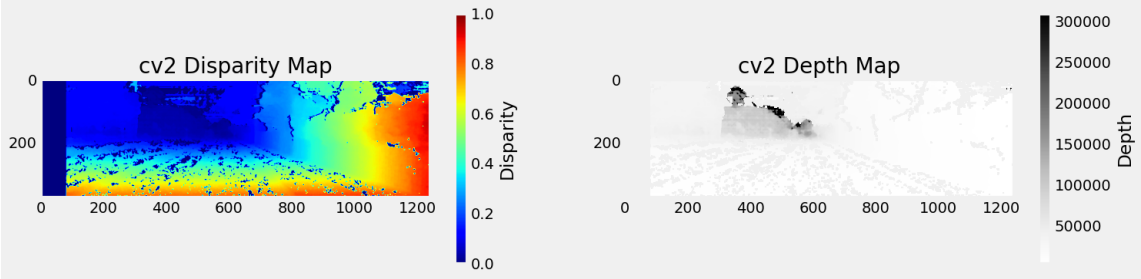


We attained the following disparity and depth maps (example for first admissable sample):



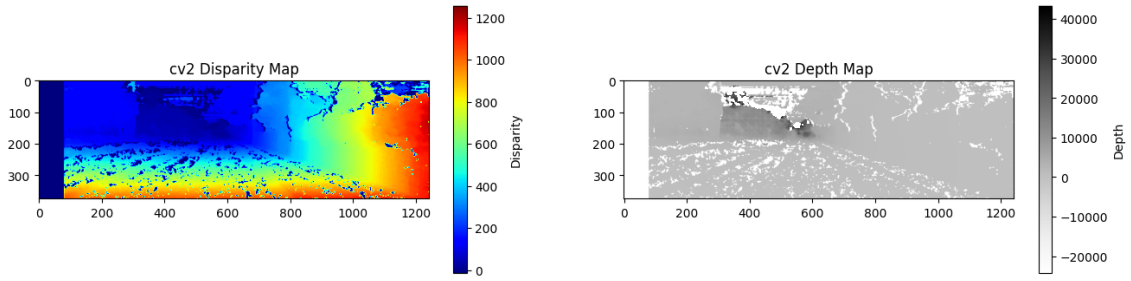
Some samples ("um_000000") were not admissible, as their corresponding ground truths were incorrect (mentioned in discussion forum), and were thus omitted from the dataset.

To compute the depth of the pixels, the focal length of the camera was inferred to be 7.215377 millimetres from the text files in the "calib" folder of the dataset, and the baseline of 54 centimetres was inferred from a YouTube video covering the details of the camera used to capture the data [1]. We compare our in-house disparity and depth maps with those computed using OpenCV's builtin functions:



We see a much clearer and smoother progression of disparity and depth as objects in the scene appear further from the camera. A major benefit from using OpenCV's implementation is the ability to leverage the parameters of their function such as `uniquenessRatio`, `speckleWindowSize`, `speckleRange`, etc. However, we still note that there are some issues with the depth map as, with some disparity values being near 0, corresponding depth values may diverge causing undefined behaviour.

We elected to normalize the disparity map as otherwise the depth map would appear less discernible:



Additionally, without normalization, the disparity and depth maps would sometimes render negative values as seen in the example above, which we know to be improper values of these types.

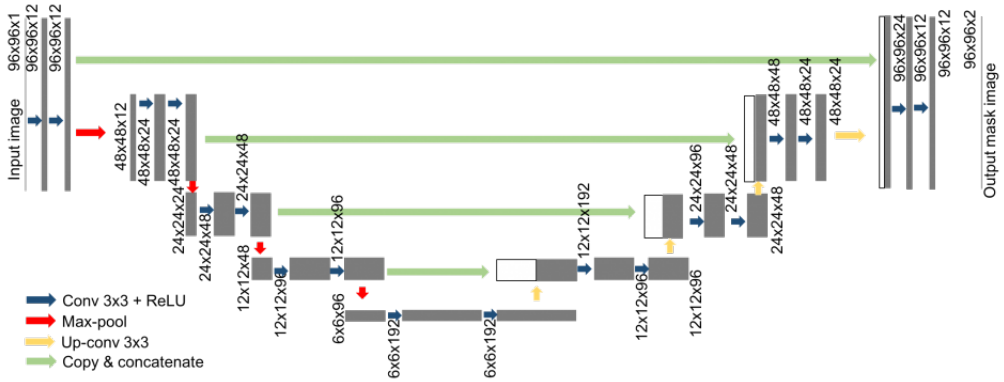
For computing the 3D location of the pixels in the image, we first create an xy-coordinate matrix, then multiply by and concatenate with the depth map. An issue from the depth map persists is the diverging/infinity values that arise from dividing by zero or near-zero disparities. Since the 3D locations depend on the depth, it too will have infinite values, which do not cooperate with plane generation as will be discussed in the 3D plane fitting subsection.

```
1  print(depth_map_cv2.shape)
2  print(compute_3D_location(depth_map_cv2)[0])
✓ 0.0s
(375, 1242)
[[            -inf             -inf              inf]
 [            -inf             -inf              inf]
 [            -inf             -inf              inf]
 ...
 [ -950829.38627727  1663042.37645455  1125500.42049545]
 [-2614780.8122625   4577656.16025     3095126.1563625 ]
 [-2614780.8122625   4581945.78525     3095126.1563625 ]]
```

Training the road classifier

The model used for the road classifier was U-Net. This architecture is used very commonly for generating masks of images, and many other types of similar-shape generation tasks. The particular architecture used was outlined in an article discussing a dataset competition, and an implementation that used the U-Net model, [2].

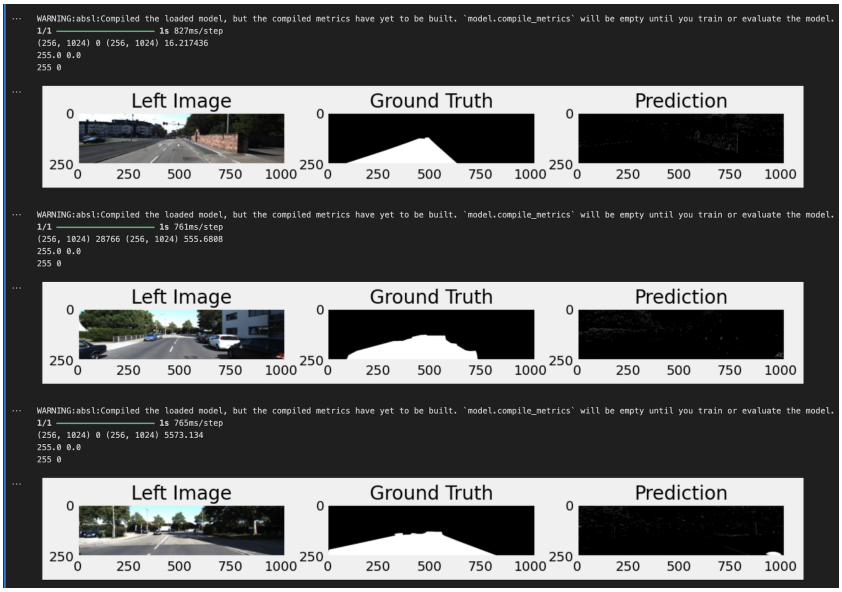
Below is an example of the model architecture, however, the exact values such as the input shape and the number of filters differ in our implementation to accommodate our particular dataset.



U-Net Architecutre - U-Net keras segmenting images

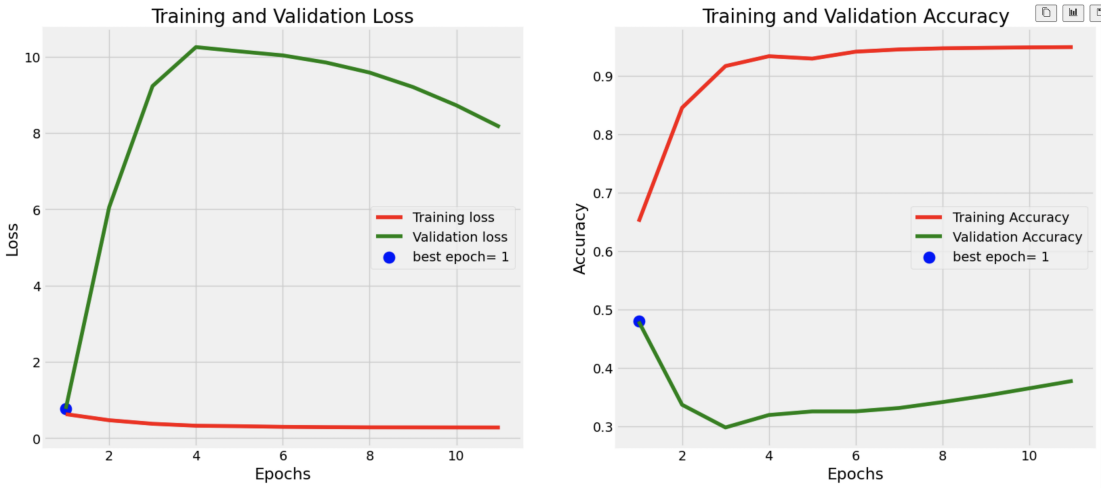
To allow the model the ability to capture proper details of our image, we change the input size to (256, 1024) instead of (96, 96), where the original image shape was around (375, 1242) plus or minus some change. The number of filters was also changed from 12 to 16. For simplicity, we are choosing to use powers of 2. During training, we invoked batch normalization, thus we did not require normalizin the image data beforehand.

This model was changed many times during development as there were many issues that arose. At first, we chose to begin with 64 filters, and passing in all 2D feature (RGB values) as well as all 3D features (disparity, depth, etc.). This resulted in near-zero predictions for masks as shown below:



The next attempt that showed a significant change (after a multitude of attempts in between) was to inlcude all the 2D features (RGB) but to only include the normalized disparity as a 3D feature for the model. This was rationalized by the fact that, while the disparity can be nicely normalized to a range [0; 1], the depth, and thus by consequence the 3D locations, will have diverging values for disparities near 0. This resulted in the following predictions:

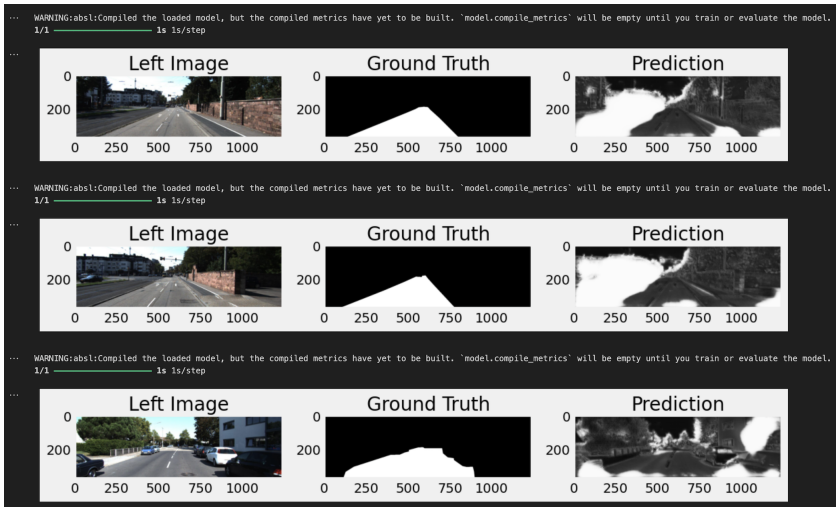
We see that the prediction ended up being an quasi-opposite version of the grayscale input image. It was unclear why this was happening, however, we especially noted that the prediction seemed to pay little to no attention to the ground truth mask. In the next attempt that showed significant improvement, similarly, after many other failed attempts in-between, we decided to only input 2D features, still RGB, and no 3D features. This choice was a shot in the dark, as we were expecting that as we removed input channels, the ability of the model to accurately predict the mask would be hindered. The following is the training curve relaying the change the model's accuracy and loss during training:



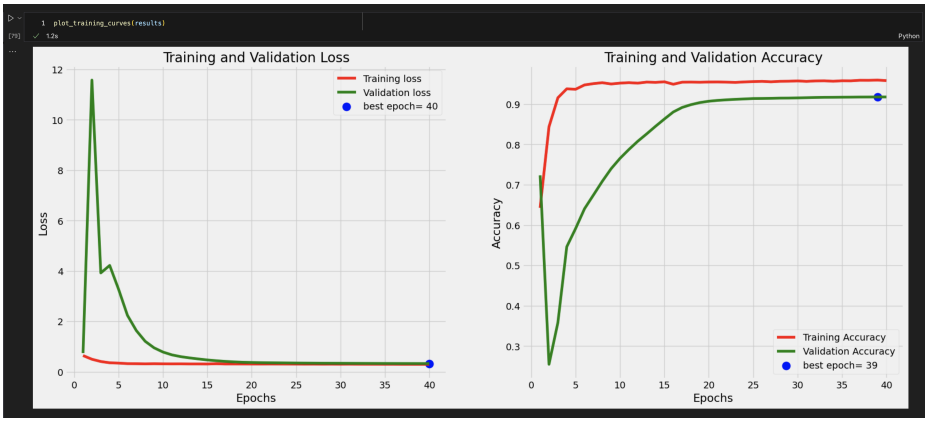
Below are the predictions attained by the attempt:

In these predictions, we see that the mask falls over all objects in the image except for the sky, which would only suggest that the car cannot fly. Obviously, the model appeared to be masking all but the most bright areas of the input image, still remaining indifferent for the road masked by the ground truth.

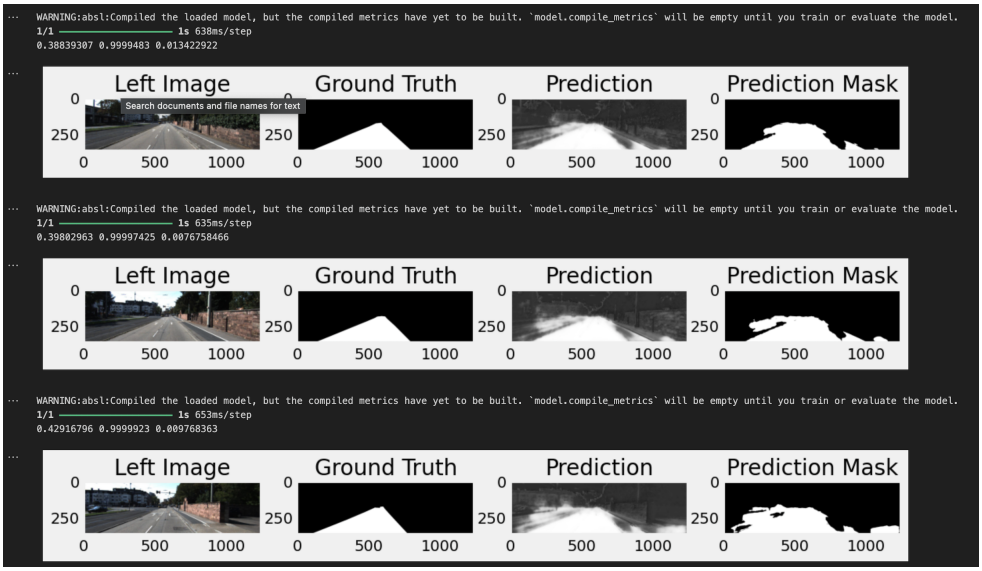
A major breakthrough, however, was that the prediction was appearing to be some sort of mask, as would be expected. The only drawback was that it was not masking what was intended. Above, we mentioned that the model uses batch normalization. However, this was not used until the fourth model generation, which showed another substantial improvement. Again, we emphasize that this next model was only attained after numerous other failed attempts in between the previously discussed, and this next model. Below are the predictions:



Here we can see that the model is now masking specific segments of the image. However, the mode was still missing the key objective of masking the road ahead of the vehicle. After a lot of sifting through our implementation, we found that, while the input images were being normalized systematically during each batch, which thus required no manual normalization on our end, the ground truths themselves were never normalized to the range [0; 1], which greatly affected the model's ability to "find" and mask the road. Below are the training curves for the same model above, while also normalizing the ground truths:



Below are the predictions of the final model:



We see that the model finally masks the road relatively correctly, showing that normalization of the ground truths were required. To the far left, we implement a threshold of 0.5 on the prediction so that “Prediction Mask" can be best compared to the ground truth.

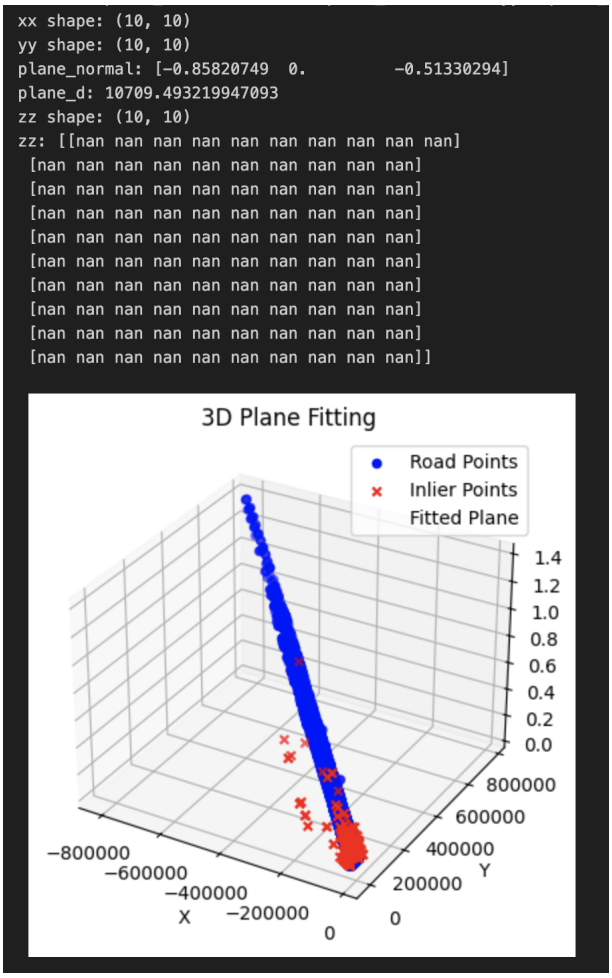
For future work, we encourage further investigating the U-Net model to determine how best to incorporate the 3D features computed in the previous tasks, as well as figuring out how to address the issue of diverging values of depth and 3D pixel location.

3D Plane fitting and 3D Point Cloud

In order to fit a plane in 3D to the road pixels using the pixel depths, we utilized the 3D pixel locations from the previous section. To perform the fitting, we implement RANSAC. We use RANSAC as the project handout pdf hints towards it by suggesting that the “algorithm [be] robust to *outliers*", as well as RANSAC being a highly useful tool towards plotting subplanes in spaces (regression) [3].

On each iteration, we sample three points and compute two vectors connecting the three points. Ideally, these two vectors would lie on two different lines, whose cross product would yield a norm vector for a plane. We then compute the distance between the first sampled point (the origin) and and the plane and count the inliers within the distance threshold of 0.1. We optimize the plane over the iterations by the number of inliers and attempt to plot.

Unfortunately, as warned in the previous section, the issue of the infininty values in the depth, and thus the 3D locations persists, and we were unable to plot the plane. We were, however, able to plot the road points as shown below:



We see that the values of the plane are `nan`, indicating an undefined behaviour.

Unfortunately, given the time constraint, the current exam season, as well as the heavy trial-and-error nature of the other main tasks of this project, we were not able to plot a point cloud of the image, nor show the estimated ground plane apart from preliminary planning of how the implementation would be done.

We thank you for your understanding.

Part 2: Object detection & Viewpoint classifier

Understanding the dataset

To gain a deeper understanding of the problem, we watched the tutorial video Vehicle and Pedestrian Detection Using YOLOv8 and Kitti dataset by Code With Aarohi [4]. In the video, Aarohi instructed viewers to download the 7480 left color images of object dataset (12 GB) and the training labels of the object dataset (5 MB) from the Kitti dataset by Geiger et al. [4]. The labels followed a specific format, as explained by Haq [5]:

- <**object_type**>: Specifies the type of annotated object, such as 'Car', 'Van', 'Truck', 'Pedestrian', 'Person_sitting', 'Cyclist', 'Tram', 'Misc', or 'DontCare'. 'DontCare' denotes objects that are present but ignored for evaluation.
- <**truncation**>: Indicates the fraction of the object that is visible, with values in the range [0.0, 1.0]. 0.0 means that the object is fully visible, while 1.0 means that the object is completely outside the image frame.
- <**occlusion**>: Represents an integer indicating the degree of occlusion of the object. 0 denotes full visibility while higher values indicate increasing levels of occlusion.
- <**alpha**>: Denotes the observation angle of the object in radians relative to the camera, the angle between the object's heading direction and the positive x-axis of the camera.
- <**left**>, <**top**>, <**right**>, <**bottom**>: Corresponds to the 2D bounding box coordinates of the object in the image, representing the pixel locations of the top-left and bottom-right corners of the bounding box.
- <**height**>, <**width**>, <**length**>: Indicate the 3D dimensions of the object (height, width, and length) in meters.
- <**x**>, <**y**>, <**z**>: Specify the 3D location of the object's centroid in the camera coordinate system (in meters).
- <**rotation_y**>: Represents the rotation of the object in radians around the y-axis in camera coordinates.

Each line of the labels file adheres to the format described above, as shown in the image below:

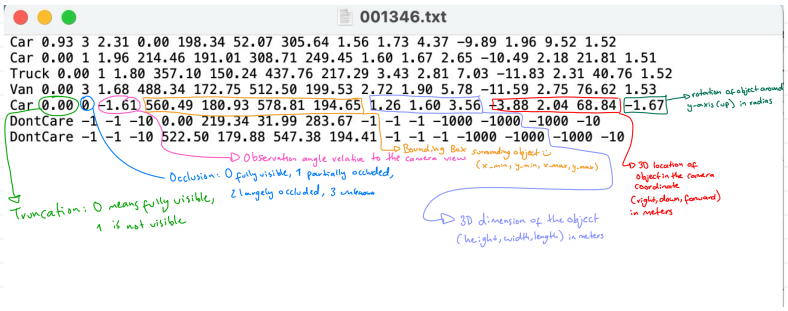


Figure 1: Each line has the format:
<object_type> <truncation> <occlusion> <alpha> <left>, <top>, <right>, <bottom> <height>, <width>, <length> <x>, <y>, <z> <rotation_y>

Due to time constraints, we made sure to utilize the dataset that Aarohi recommended downloading because the .txt files were generally easier to understand compared to the provided MATLAB files for labels. Additionally, we opted to only use the images from the `train_angle\train\image` directory along with their corresponding .txt files. This resulted in a total of 195 images, divided as follows:

- 148 images for training
- 30 images for validation
- 17 images for testing

We also made sure to create new .txt files that would meet the requirements of the YOLOv8 model. Each line in these files contains the class ID ('Car': 0, 'Pedestrian': 1, 'Van': 2, 'Cyclist': 3, 'Truck': 4, 'Misc': 5, 'Tram': 6, 'Person_sitting': 7, 'DontCare': 8). We excluded 'Misc' and 'DontCare' since we did not want the model to detect these classes. Additionally, each line includes the bounding box coordinates of the object, as shown in the following image:

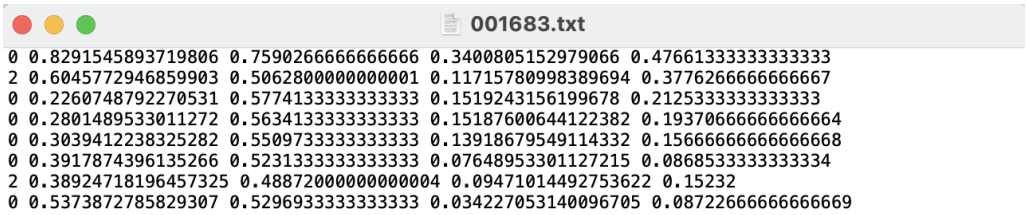


Figure 2: Each line has the YOLOv8 format:
<class_id> <left>, <top>, <right>, <bottom>

We chose to use YOLOv8 over a pretrained model because the data format for the pretrained models was unclear, and the provided code for preprocessing the data was outdated, dating back 11 years. After spending half a day attempting to make it work, we realized that continuing with this approach would hinder progress compared to the preprocessing plan We already had in place.

Detect cars in the image ... and a few more objects

Despite the limited time and the small number of training images, We decided to set up the model infrastructure to detect vehicles (Car, Van, Truck, Tram) and pedestrians (Pedestrian, Person_sitting, Cyclist). This decision was made so that the infrastructure would be prepared for further training on thousands of images rather than just hundreds. As shown in the image below, most of the images contained instances of cars compared to the other classes:

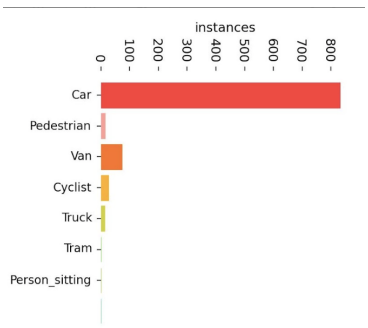


Figure 3: There are roughly over 800 instances of cars compared to less than 100 instances for the remaining classes

We trained both models on Google Colab utilizing the GPU.
For the initial training of the YOLOv8 model, we conducted 10 epochs with a batch size of 8, considering our limited time and data availability. This training session lasted approximately 25 minutes. The model is saved at: `MODEL_1_CAR_DETECTION/train/weights/best.pt` The model performed well in making predictions for cars overall, but it struggled to detect the other classes as frequently or would misinterpret some objects. The following images display the validation set compared to the predictions:

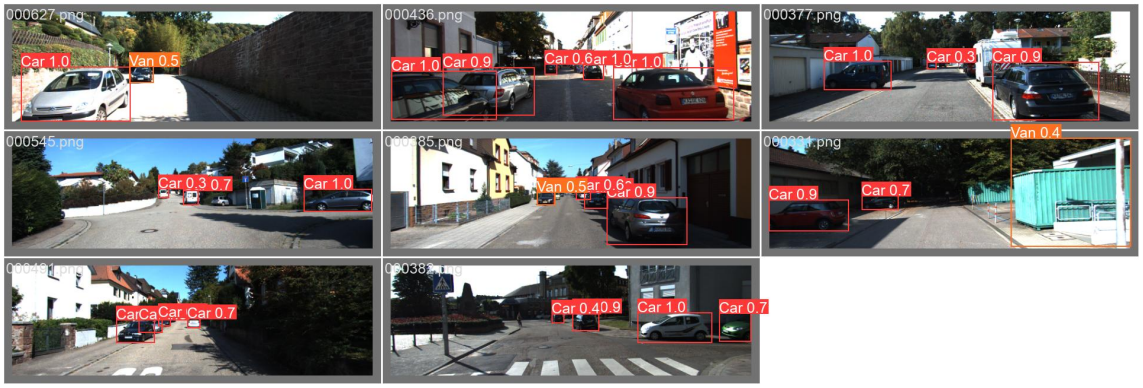
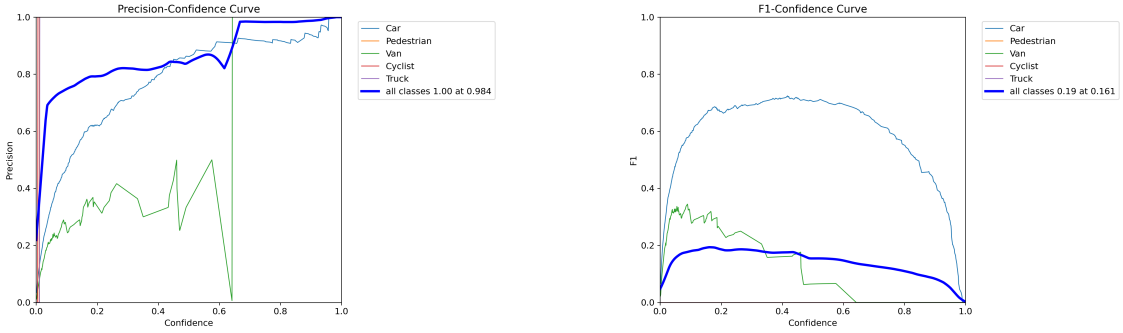


Figure 4: In these predictions, a container can be easily mistaken for a van, and some classes (e.g., cyclists) are missing.

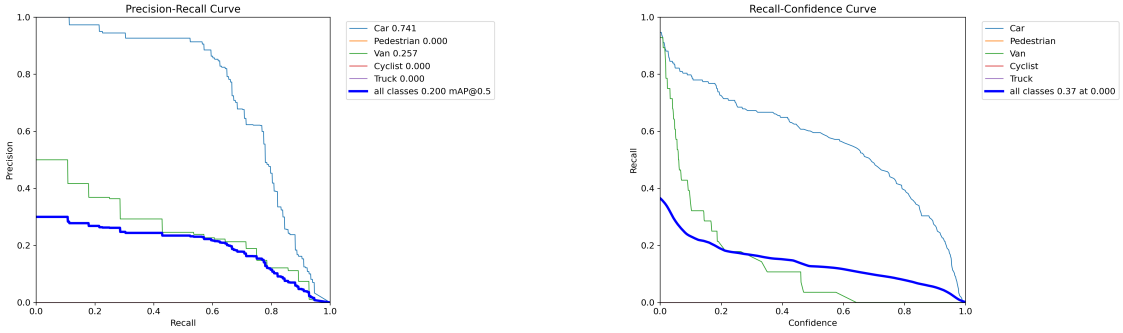
Figure 5: The ground truth includes pedestrians and cyclists, which the model struggles to detect accurately.

The following are the validation diagrams for the model, which indicate that the model is generally very confident about cars but less confident about the other classes:



(a) (b)

Figure 6: (a) Shows that the model is not as confident about pedestrians and cyclists, which correlates with the previously mentioned prediction. (b) Demonstrates how the confidence in the van class is low, explaining why the model mistakenly predicted a container as a van.



(a) (b)

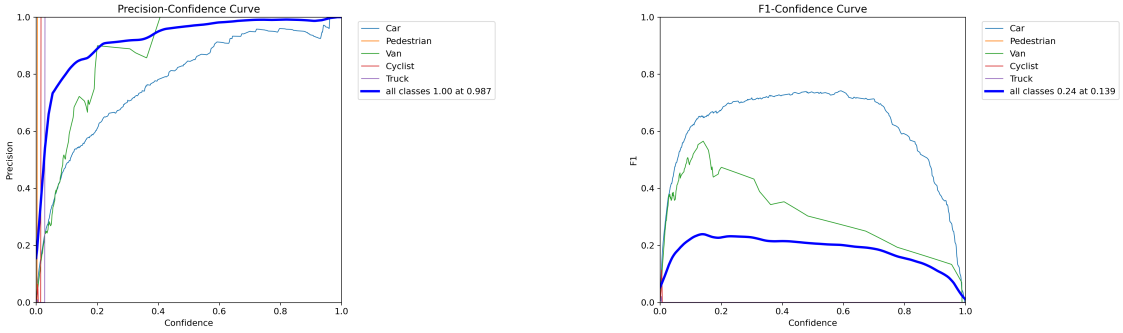
Figure 7: These curves show that the model is very confident with the car class compared to the other classes.

For the second training session, we increased the number of epochs to 20 and maintained the batch size at 8, resulting in a training time of around an hour. The model is saved at: `MODEL_2_CAR_DETECTION/train/weights/best.pt` In this iteration of the model, the model was more confident about it's car prediction and although it still struggled to detect the other classes as frequently, it would no longer misinterpret object. The following images display the validation set compared to the predictions:

Figure 8: In these predictions, the container is no longer mistaken for a van, and the model seems to only predict the classes it is confident about.

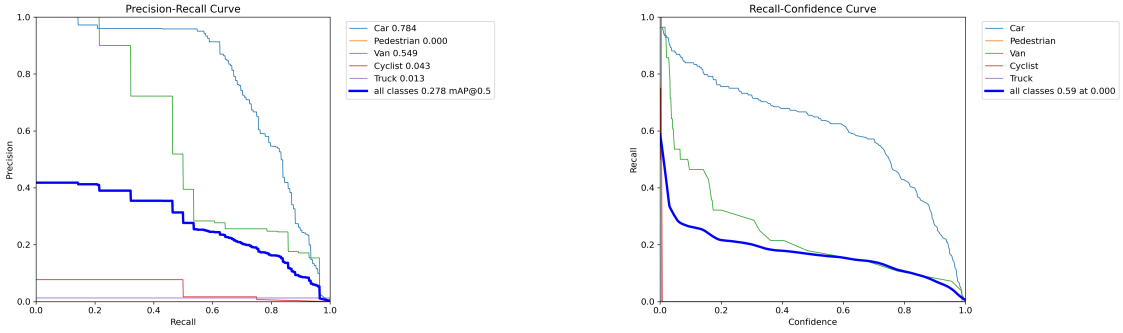
Figure 9: Although the ground truth includes pedestrians and cyclists, which the model still struggles to detect accurately, the predictions for cars seem to match the model well.

The following are the validation diagrams for the model, which indicate that the model is generally very confident about cars but less confident about the other classes:



(a) (b)

Figure 10: (a) Shows that the model is still not as confident about pedestrians and cyclists, but it is more concerned about correctly predicting cars, which aligns with the earlier observations. (b) Demonstrates higher confidence levels in the van class and the car class, explaining why the model no longer mistakenly predicts the container as a van.



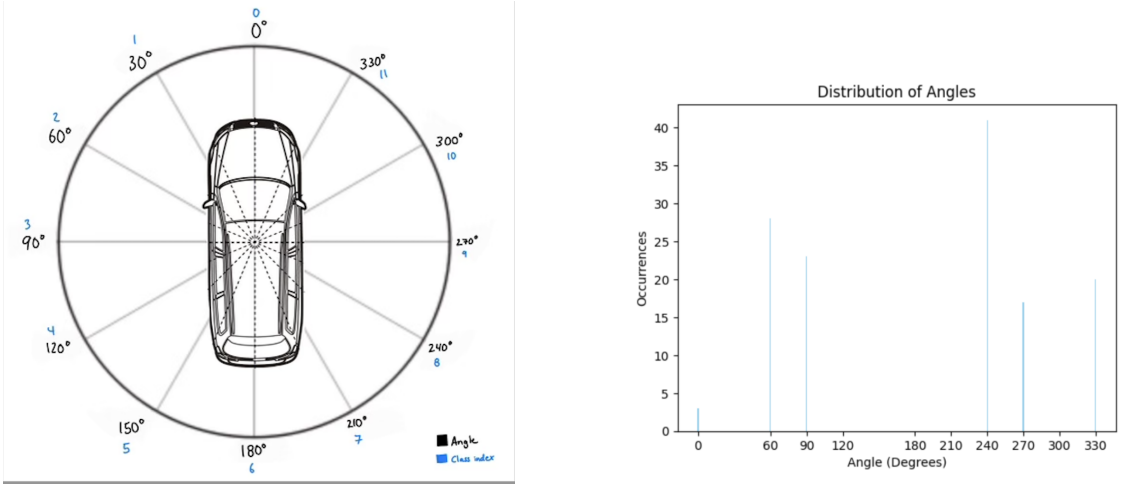
(a) (b)

Figure 11: These curves show that the model is overall more confident, but it still shows higher confidence with the car class compared to the other classes.

Training the viewpoint classifier for each vehicle

As outlined in the dataset section, the last value of each line in the .txt labels represents the rotation of the object in radians around the y-axis in camera coordinates. To train the viewpoint

model, we utilized the original .txt labels as they contained crucial information. Specifically, we used the <rotation_y> value from these labels as it was necessary for the model's prediction. Additionally, We created patches of 100 by 100 pixels for each object. This approach was adopted to aid the model in distinguishing whether the vehicle was facing forwards or backwards, which in turn influences the view angle. Moreover, we incorporated the width and height of each vehicle, as the orientation of the parallelogram could provide valuable information to the model.
Before training the model, an analysis of the angle distribution was conducted, depicted in the following figures:

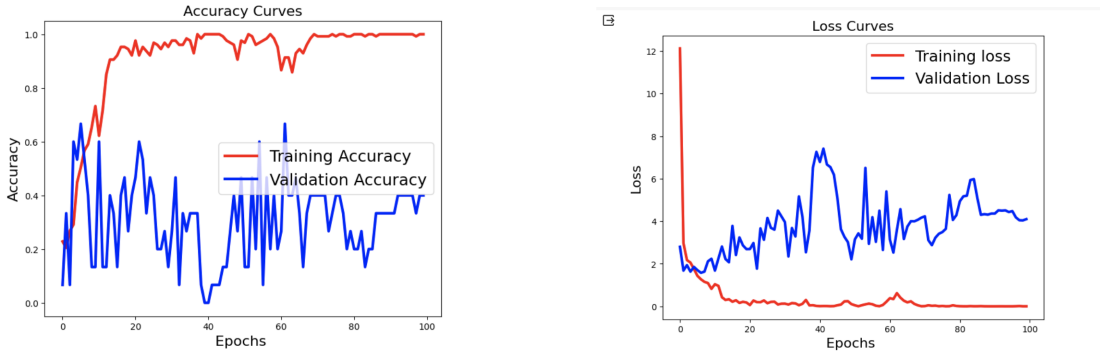


(a) (b)

Figure 12: Note that the car is more likely to turn right when facing forwards on the right lane (angle ranges: 270 to 360), to turn left when facing forwards on the right lane (angle ranges: 0 to 90), or turn left when facing forwards on the left lane (angle ranges: 240 to 270)

It is noticeable that there are some angles with zero occurrences, which is reasonable as vehicles at such sharp angles are unlikely to be on the road, as they indicate impending collisions or erratic driving behavior.

The first model architecture can be found in `object_detection_dataset/MODEL_1_VIEWPOINT'`, and it was trained with a batch size of 4 for 100 epochs. Training only took a few minutes. The following images display the accuracy and loss curves:



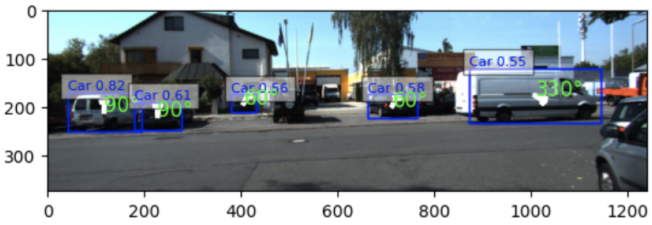
(a) (b)

Figure 13: The curves correspoding to the first model.

Overall, the prediction closely aligns with the angle grid, as shown in the figure below:

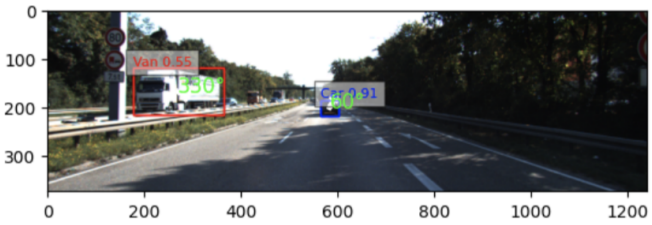
```
[115] # Check predictions!
      # Path to the image file
      img_file = '/content/drive/MyDrive/CSC420_FINAL/object_detection_dataset/test/images/001773.png'

      # Visualize cars with predicted viewpoints
      visualize_vehicles_with_viewpoint(car_detection_model, viewpoint_model, img_file)
```

```
image 1/1 /content/drive/MyDrive/CSC420_FINAL/object_detection_dataset/test/images/001773.png: 224x640
Speed: 3.0ms preprocess, 207.6ms inference, 1.6ms postprocess per image at shape (1, 3, 224, 640)
1/1 [==============================] - 0s 263ms/step
1/1 [==============================] - 0s 56ms/step
1/1 [==============================] - 0s 56ms/step
1/1 [==============================] - 0s 64ms/step
1/1 [==============================] - 0s 63ms/step
```




```
[116] img_file = '/content/drive/MyDrive/CSC420_FINAL/object_detection_dataset/test/other_cases/000051.png'
      # Visualize cars with predicted viewpoints
      visualize_vehicles_with_viewpoint(car_detection_model, viewpoint_model, img_file)
```

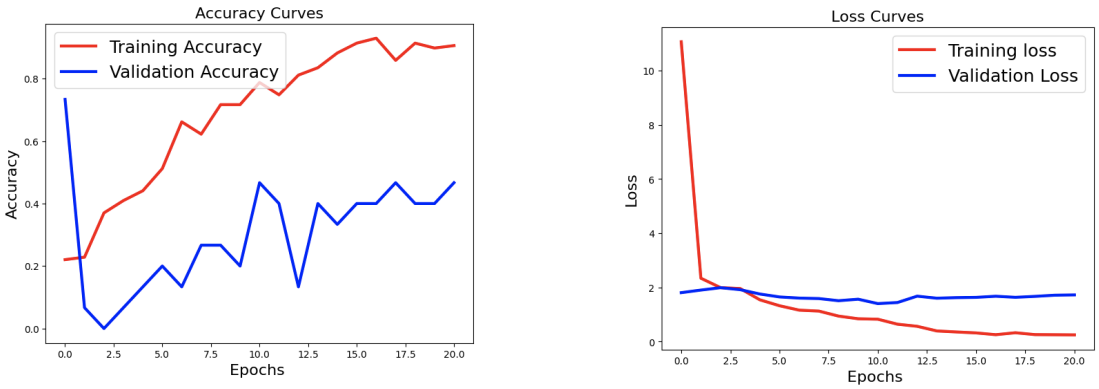
```
image 1/1 /content/drive/MyDrive/CSC420_FINAL/object_detection_dataset/test/other_cases/000051.png: 224
Speed: 1.7ms preprocess, 75.9ms inference, 0.9ms postprocess per image at shape (1, 3, 224, 640)
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 38ms/step
```



Since the initial model trained swiftly, We attempted to enhance it by increasing the number of epochs and reducing the batch size. Additionally, We implemented early stopping to halt training if the model ceased improving for 10 consecutive instances. Although these adjustments prolonged training, the results were mostly inaccurate, particularly regarding cars facing forwards on the right lane, which were predicted to have roughly 240 degrees instead of 0. However, this prediction aligns with our angle distribution. Throughout these attempts, We observed that the model often halted training around the 15th epoch, prompting me to experiment with a deeper model architecture. Although this model still produced inaccurate predictions, it showed slight improvement over the first model, notably in accurately predicting the van's orientation in the first image. This model was trained using a batch size of 32 for 100 epochs.

Figure 14: In the first image, the van's viewpoint is correctly predicted.

The following images show the accuracy and loss curves of the model:



(a) (b)

Figure 15: Although both curves are much smoother the predictions seem to be more on par with the distribution

Training these models taught us that having more data and time for training leads to better results. From various iterations of training the viewpoint models, it became evident that more "accurate" models attempted to align closer with the distribution of angles in the training images. However, there were still cases not adequately addressed, as demonstrated by the predictions on test images. This suggests that with a more comprehensive angle distribution and more images, we could achieve more accurate viewpoint predictions. Another indication of this was the model's tendency to early stop at lower epochs, indicating a need for more information to increase confidence in its predictions.

Part 3: Computing 3D bounding box for each detected vehicle

Unfortunately, due to time constraints, we faced challenges in implementing tasks 4 and 5 robustly. Specifically, we encountered issues with infinite depths in our OpenCV implementation, which persisted across these tasks and directly affected their outcomes. However, given the ground plane, estimated depth, and the location of the car's bounding box, we could compute the 3D bounding box around each detected car. This could be achieved by determining the location of the points that form the bounding box of the cars (from the detection model) on the xy-plane. Then, we could utilize the depth information to account for the z-coordinate. Each point in the car's bounding box would become lines, thus forming a 3D bounding box for the car.

Bibliography

[1] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[2] T. Sterbak, "U-net for segmenting seismic images with keras," 2018.

[3] E. Lewinson, "Dealing with outliers using three robust linear regression models," 2022. Technical Blog.

[4] C. W. Aarohi, "Vehicle and pedestrian detection using yolov8 and kitti dataset," 2024. Youtube Tutorial.

[5] abdul haq, "Explain label file of kitti dataset," 2023. Medium Article.

[6] C. McCormick, "Stereo vision tutorial - part i," 2014. Matlab Tutorial.