Computer Vision Project UNITS

alberto.zurini

January 2025

1 Project description

The ultimate goal of the project is to detect traffic lights in a video recorded from a moving car, estimate their distance and color (red, yellow, green or black). The project is designed to be modular, with each component addressing a specific aspect of the task. This approach allows for easy adaptability and potential expansion, such as the integration of additional sign detection or other dynamic objects. The project is divided into the following key stages:

1. **Calibrate the camera and estimate the distance using the checkerboard pattern:** The first step involves calibrating the camera to ensure accurate measurements. This process uses a series of images of a checkerboard pattern taken at varying distances. These images are utilized to estimate the camera's intrinsic parameters, such as focal length and optical center. Once the calibration is complete, the images are undistorted to compensate for lens distortions, ensuring greater precision in subsequent steps. The same checkerboard pattern is also used to estimate the distance by analyzing the known geometric properties of the pattern.

2. **Use YOLOv11 for traffic light detection:** The next step involves using YOLOv11 [6], a state-of-the-art object detection model, to identify and locate traffic lights within the video frames. YOLOv11 provides accurate bounding boxes around detected traffic lights, which are essential for further analysis.

3. **Classify traffic light color using MobileNetV2 with fine-tuning:** Once traffic lights are detected, their color must be classified (red, yellow, green or black). MobileNetV2 [7], a lightweight convolutional neural network (CNN), is fine-tuned using a pre-trained model to efficiently classify the traffic light color, allowing for real-time processing with minimal computational overhead.

4. **Estimate the traffic light distance using video stream:** The final step focuses on estimating the distance to the detected traffic light. This involves measuring the size of the traffic light in the image, and using known geometric relationships to compute the distance. The results are

based on the initial calibration performed in step one, ensuring consistency and accuracy throughout the process.

The modular nature of this project allows for future scalability, with potential applications extending to the detection of other dynamic objects, such as road signs and pedestrians, which may require real-time detection and tracking.

2 Camera calibration and checkerboard distance estimation

Before estimating the distance to a traffic light, it was essential to first evaluate the accuracy of the distance measurement process. For this purpose, a checkerboard was utilized. The dimensions of the boxes on the checkerboard can be determined with sub-pixel precision, making it one of the most accurate methods for measuring object dimensions in photographs. Furthermore, the checkerboard possesses a fixed geometric structure that does not vary, thereby enhancing the reliability of the results.

The hypotheses to be tested are as follows:

1. Perspective is a linear phenomenon, which implies a linear relationship between an object's distance and its apparent size.

2. What is the accuracy of the distance estimation? To answer this, the estimated distances will be compared against measurements taken with a ruler, which provides the actual distance between the camera and the checkerboard.

The distances are going to be estimated using two methods, the first one is simpler to implement, but might be less accurate:

1. Knowing a side length of the checkerboard and the distance we can estimate the distance of different shoots taken at the checkerboard using the linear properties of perspective projection.

2. Using the extrinsic parametrization of the camera (which is gotten from the camera calibration procedure) we can use the translation vectors to get an estimate of the distance of the camera to the checkerboard.

2.1 Measuring distance using the known length of a side of the checkerboard

The camera was first calibrated, and all the images were subsequently undistorted. This step is a mandatory requirement, especially for images captured closer to the checkerboard. In the Figure 1, it can be observed that, in the undistorted image (on the right), the lines of the checkerboard appear straight and properly aligned.

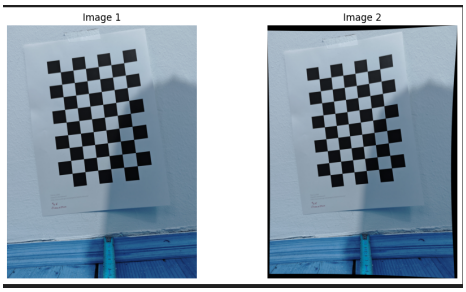

Figure 1: Distorted vs undistorted images

2.1.1 How the checkerboard distances were measured

The phone was placed in a holder next to a ruler 2. The initial distance was set at 300 mm, and then, in steps of 100 mm, the phone was moved up to a maximum distance of 2000 mm from the checkerboard.

In total, 18 pictures of the checkerboard were captured at known distances.

Figure 2: Measurement of the checkerboard distance

2.1.2 Perspective is a linear phenomena

After the measurements were taken, they were plotted on a chart to confirm the presence of a linear relationship. The results indicate a fairly linear trend, suggesting that the captured images correspond closely to the ground truth distances observed on the ruler. This confirms the accuracy of the data, enabling the progression to the next steps of the experiment, as can be seen in Figure 3.

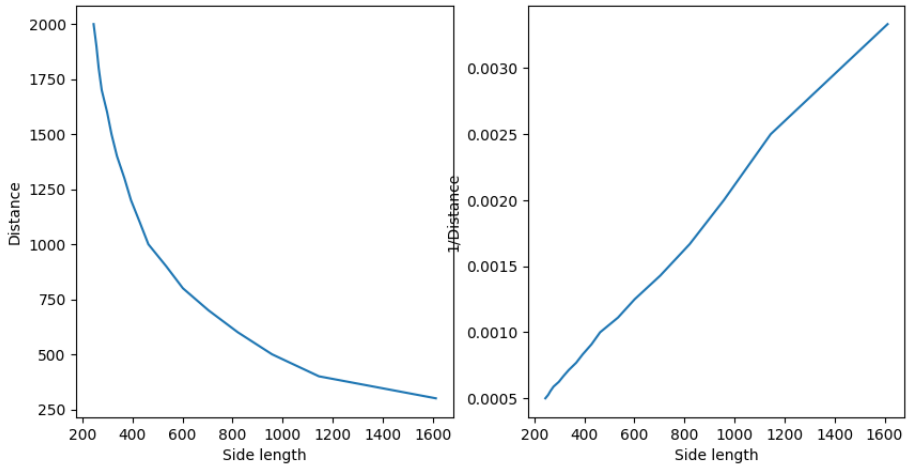

Figure 3: Comparison of the side length of the checkerboard with the distance seen in the ruler

2.1.3 Distance estimation

The distance can be estimated by measuring the length of one side of the checkerboard, specifically the height of the checkerboard, which is the longest side. This value is placed in the denominator of an expression [2].

The formula used is:

$$distance_{mm} = \frac{FACTOR}{sideLength_{px}}.$$

To calculate $FACTOR$, the phone can either be placed at a known distance and the value retrieved, or a linear regression model can be trained. I opted for the second method, as it allows for the averaging out of individual measurement errors.

The gotten results are visible in Figure 4.

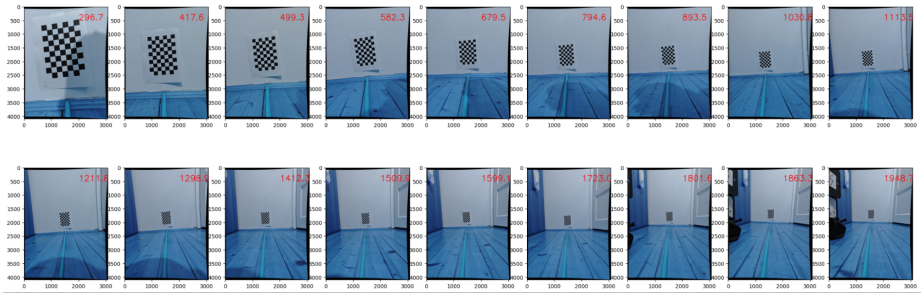

Figure 4: Distances estimation using only the side length of the checkerboard

The results are quite accurate and fall at most in a 50 mm confidence (assuming 0 error in the known distance of each image!), as can be noticed in Figure 5.

Figure 5: Error vs measurement number

2.2 Another method to compute the distance

Since the geometric representation of the checkerboard is known, its distance can be measured using the same formulas applied for calculating the extrinsic parameters of the camera calibration matrices. The camera-world model that has been used is the one depicted in Figure 6. Specifically, OpenCV's API allows for the retrieval of translation and rotation vectors using the SolvePnP function [4].

This function requires the following inputs:

1. The vector containing the known locations of the corners (in a normalized coordinate system)
2. The coordinates of the corners (sub-pixel accuracy ones will be used)
3. `mtx`, which contains the intrinsic parameters of the camera (focal lengths, optical center, skew coefficient)
4. `dist`, which contains the distortion coefficients

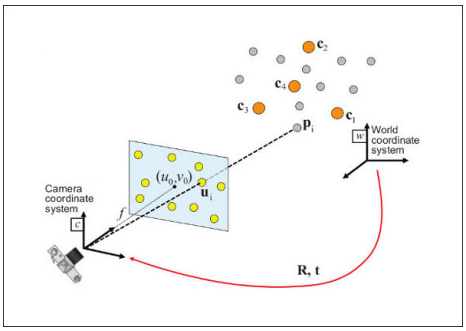

Figure 6: Camera-world model

The results obtained from this function can be used to estimate the distance to the checkerboard. Specifically, the norm of the translation vector can be calculated, which should provide an indication of how far the checkerboard is from the camera.

The results were plotted and compared with the estimations obtained using the other method, as shown in Figure 7. The comparison is further illustrated in the chart in Figure 8, which plots the first distance method against the second, as well as the ground truth against the second method.

It can be observed that there is a stronger linear relationship in the first chart (Figure 8). This is likely because the measurements taken with the ruler are not 100% accurate, while the linear regression model trained using the perspective projection relation (also called "first method", during the course of this report) aligns more closely with the method used to compute the translation vectors (which is called "second method" in this report).

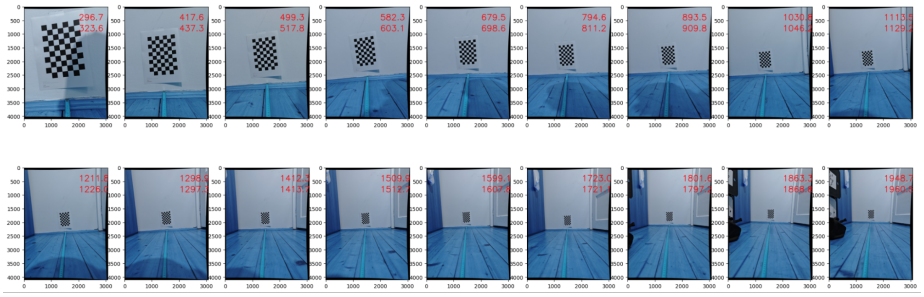

Figure 7: Distance estimation comparison using the two methods (top: first method; bottom: second method)

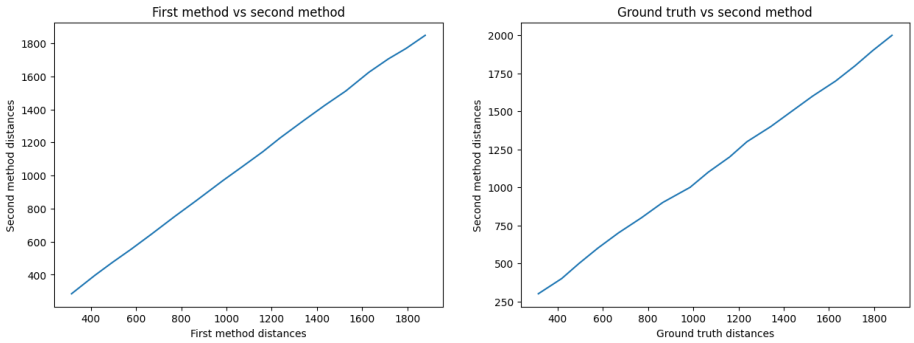

Figure 8: First method vs second method and ground truth vs second method

Figure 9 shows the errors of the two distance estimation methods for each measurement. The second method is slightly more accurate than the first one. However, considering the added complexity, the first method appears to be a reliable enough way to estimate the distance to objects. It is interesting to note that the error increases as the distance to the checkerboard increases. This is reasonable because, at greater distances, fewer pixels are available for the calculations. As a result, a small absolute error in the pixel count (e.g., due to slight camera shaking) leads to a higher relative error.

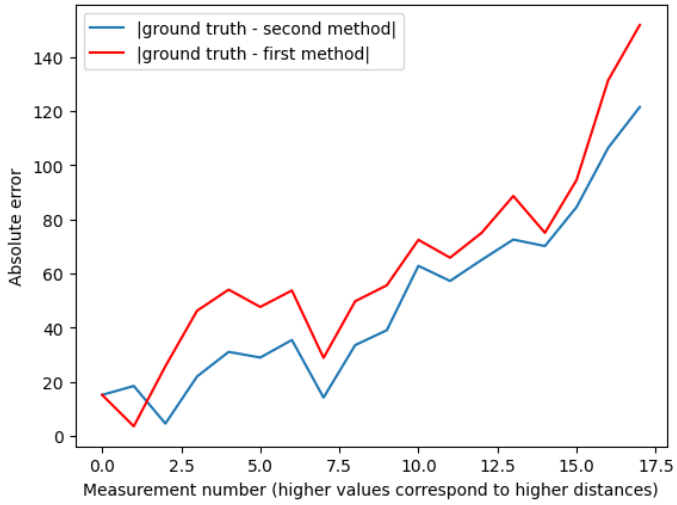

Figure 9: Side by side error comparison of the two methods vs ground truth

3 YOLOv11 usage to find the traffic lights in the image

According to the official homepage of YOLOv11, getting started with the model requires just a few lines of code. This simplicity is one of its key strengths. To begin, you only need to import a single Python module at a high level and provide the model's name. Additionally, some custom libraries must be installed, but these are straightforward to set up and do not require extensive configuration.

Once the setup is complete, the model can be initialized, and the detections array will contain all the necessary information for subsequent steps. Specifically, the detections array includes the bounding boxes, class labels, and accuracy scores for each detected object in the image. This allows for easy extraction of relevant data and makes it simple to proceed with further processing, such as filtering or visualizing the results.

4 Traffic light color classification

A MobileNetV2 model is going to be fine-tuned using PyTorch [3]. The goal of using MobileNetV2 is to leverage its lightweight architecture, which makes it suitable for real-time applications where computational efficiency is a priority. While the initial application of this model will be focused on detecting traffic lights, the long-term goal is to expand its use to detect other types of signs that rely on lights for identification. These signs can be more complex and dynamic, presenting additional challenges for detection.

Traffic lights themselves can appear in various orientations, both horizontal and vertical, and may feature different designs or configurations. Therefore, the aim is to create a model that is as generalizable as possible, capable of recognizing traffic lights in diverse environments and orientations. Rather than relying on traditional hardcoded computer vision techniques, which might struggle to generalize across variations in lighting, camera angle, and environmental conditions, this approach will use a machine learning model that can adapt to such variations automatically through training.

MobileNetV2 was specifically chosen for this purpose because of its efficiency in handling such tasks, offering a balance between performance and speed. Fine-tuning the model on a dataset of traffic lights allows the model to first learn the key features relevant to traffic light detection, such as shape, color, and light patterns. Once fine-tuned, the model can be further extended to handle more complex tasks, such as detecting other traffic-related signs, which are equally dynamic and require the model to be capable of recognizing different types of signals under varying conditions.

The process is straightforward:

1. Freeze the layers

```python
# Set the output layer to only predict 3 different classes (red, orange, green, black)
NUM_CLASSES=4
model.classifier[1] = torch.nn.Linear(in_features=model.classifier[1].in_features, out_features=NUM_CLASSES)

# Freeze the layers
for param in model.features.parameters():
    param.requires_grad = False

print(model.classifier)
```

2. Prepare the data loader

```python
# Define transformations for the images
transform = transforms.Compose([
    transforms.Resize((64, 64)),  # Resize images to 224x224
    transforms.ToTensor(),          # Convert images to PyTorch tensors
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize to [-1, 1]
])
```

```python
dataset_path = "/workspace/dataset/train"
# Load dataset
dataset = datasets.ImageFolder(root=dataset_path, transform=transform)
# Create a DataLoader for batching and shuffling
dataloader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=4)
class_labels = dataset.classes
print("Class Labels:", class_labels)
```

3. Train the model

```python
    val_loss /= len(val_loader)
    accuracy = 100 * correct / total
    print(f"Validation Loss: {val_loss:.4f}, Accuracy: {accuracy:.2f}%")
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
Epoch [1/10], Loss: 0.5074
 10%|▉         | 1/10 [00:01<00:13,  1.52s/it]
Validation Loss: 0.5241, Accuracy: 82.25%
Epoch [2/10], Loss: 0.1904
 20%|█▊        | 2/10 [00:02<00:10,  1.27s/it]
Validation Loss: 0.4291, Accuracy: 85.12%
Epoch [3/10], Loss: 0.1396
 30%|██▋       | 3/10 [00:03<00:08,  1.20s/it]
Validation Loss: 0.3433, Accuracy: 89.30%
Epoch [4/10], Loss: 0.1258
 40%|███▌      | 4/10 [00:04<00:06,  1.16s/it]
Validation Loss: 0.2409, Accuracy: 92.17%
Epoch [5/10], Loss: 0.0982
```

4. Export the model

```python
!mkdir /workspace/model
```

```python
torch.save(model.state_dict(), "/workspace/model/traffic_light_model.pth")
```

The dataset used is available on Kaggle [1]. This dataset has been chosen because it is quite challenging: the images are screenshots from the Carla sim-

ulator [5]. The aim was to assess how well the model can adapt to and classify real-world traffic lights from such low-quality data.

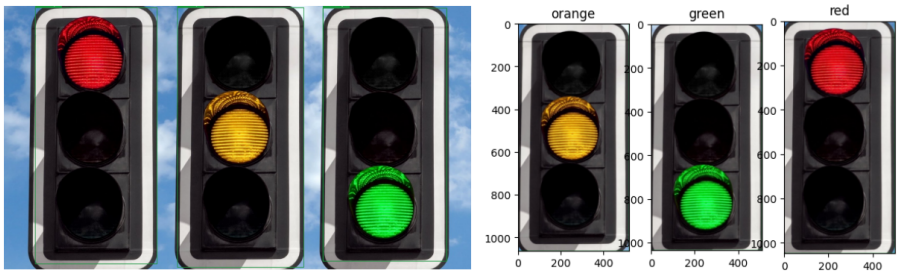

Figure 10: On the left the traffic lights are being detected using YOLOv11. On the right traffic lights are getting classified using the MobilenetV2 fine-tuning

5 Combining everything into a pipeline

The workflow for the project consists of several sequential steps to achieve the desired result. Each step builds upon the previous one to eventually detect, classify, and estimate the distance to traffic lights (and potentially other objects). Below is the breakdown of the steps involved:

1. **Image Acquisition:** First, an image is captured from the camera or video feed.
2. **Object Detection (YOLO):** The image is passed through an object detection model (YOLOv11) to identify potential traffic light locations within the image. This model detects bounding boxes around the objects of interest, specifically traffic lights in this case.
3. **Segmentation:** After object detection, a segmentation step is performed to isolate the traffic light or other relevant objects from the rest of the image, ensuring that only the relevant portions of the image are considered in the next steps.
4. **Object Classification (MobileNetV2 Fine-tuning):** The segmented image is then passed through a fine-tuned MobileNetV2 model, which classifies the color of the traffic light based on the features it has learned.
5. **Distance Estimation (Using Side Length):** Once the object is classified, the distance to the object (in this case, just the traffic light) is estimated. This is done by measuring the side length of the detected object in pixels and applying the previously determined calibration factor.

Unfortunately, the performance is not optimal, with the system processing only 15 frames per second on my laptop equipped with an RTX 3060. This issue

is not due to the model's computational complexity, but rather the fact that the individual steps—such as object detection, segmentation, and classification—are not yet optimized to run efficiently on my machine in real-time.

6 Result

To estimate the distance to the traffic light, I drove my car in my hometown. A frame was extracted from the video feed, with the distance to the traffic light known beforehand. This known distance was then used as a reference to compute the traffic light's distance based on the height (side length) of the detected traffic light in the image.

Here is the result: `https://youtu.be/03zHheRwvPo`.

Code: `https://github.com/albertoZurini/UNITS-ComputerVision`

7 Conclusion

In conclusion, while the classification of traffic lights did not yield optimal results, this can largely be attributed to the dataset used, which was not sufficiently generalized for real-world scenarios. Although the dataset performed well with high-quality images downloaded from the internet, it failed to generalize effectively when applied to real-world conditions. Despite this limitation, the rest of the project progressed smoothly.

Further investigations are needed to improve the calculation of perspective and better estimate the side length of traffic lights. Additionally, more tests should be conducted with different traffic light types and sizes. Exploring alternative methods, such as using the circles of the traffic lights (which may have more standardized dimensions), or comparing ratios of various objects within the traffic light, could lead to more accurate results.

This experiment does not aim to demonstrate that computer vision alone is a reliable method for depth estimation. In fact, using a single camera, as done in this study, is not an ideal approach for such tasks. At least a pair of cameras or, even better, a LiDAR system, would be required to achieve more accurate depth measurements.

Nonetheless, the results obtained from the checkerboard distance estimation show that when the geometry of the scene is well-known, distance can be reliably calculated. This suggests that with further improvements and the use of more appropriate data and techniques, depth estimation using computer vision could become a more viable solution in the future.

References

[1] Carla traffic light images. `https://www.kaggle.com/datasets/sachsene/carla-traffic-lights-images`.

[2] How to calculate distance between camera and object using an image. https://stackoverflow.com/questions/14038002/opencv-how-to-calculate-distance-between-camera-and-object-using-image.

[3] Mobilenet v2. https://pytorch.org/vision/main/models/mobilenetv2.html.

[4] Terzakis, g., and m. lourakis. "opencv: Perspective-n-point (pnp) pose computation." opencv documentation. https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html.

[5] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator, 2017.

[6] Rahima Khanam and Muhammad Hussain. Yolov11: An overview of the key architectural enhancements, 2024.

[7] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. 2018.