Final Report
December 12 2023
ENPH353

Kai Asaoka, Avery Wong

Table of Contents

Introduction:

For our final project in ENPH 353, we were tasked to develop an algorithm to program a robot in ROS to navigate around a course while reading signs to solve a mystery. We were introduced to a variety of concepts surrounding machine learning throughout the course to tackle this task, from traditional algorithms for image recognition such as SIFT, to more complex methods using CNNs and Deep learning. Using these concepts, we developed an algorithm capable of fully navigating the course while collecting all clues along the way to achieve the maximum points possible.

Design Philosophy:

Before starting to program, we agreed upon key design elements early on to maximize our efficiency. Our core ideas were as follows:

- Use either imitation learning or PID for navigation. We ultimately ended up using PID in the interest of time for all sections of the course, as we found success early on through experimentation. While this ended up being successful, this can largely be attributed to the relative consistency of the course, as well as the clarity in the road (which consisted of a singular shade of gray). Imitation learning, or using machine learning here may have been more beneficial in a more complex environment, but our PID implementation ended up being efficient enough to be more than serviceable for our application.

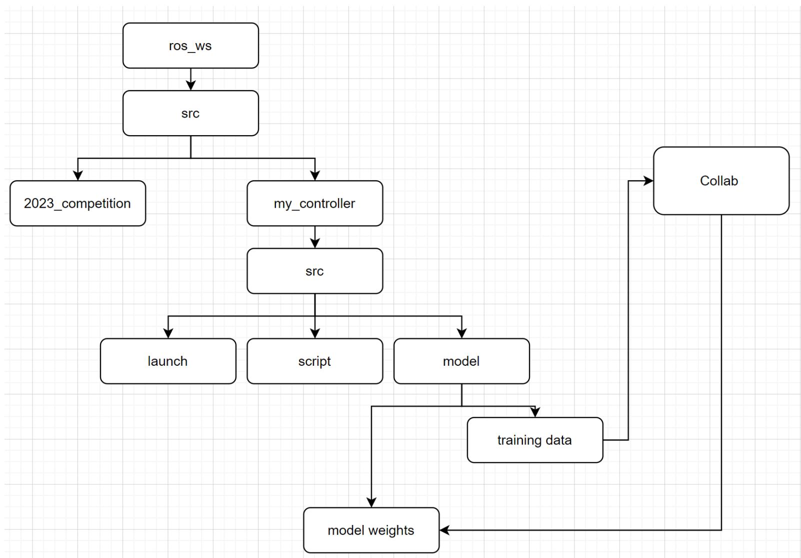
- Use a CNN for letter recognition. We saw this as the most straightforward way of reading clues on the signs around the course, as we had previously trained a neural network to do something similar: reading letters on license plates. We also decided early on to take advantage of contours instead of feeding images into our neural network to improve speed and simplify training. More about this will be elaborated later in the report.

Partition of Work:

- Avery Wong: Designing clue reading architecture (data collection for sign reading, training neural network for sign reading, implementation of clue guessing algorithm), assisting with navigation.

- Kai Asaoka: Navigation (PID on road, grass, mountain, as well as navigation for baby yoda section), pedestrian detection, truck detection, assisting with data collection.

Software Architecture:



Our work primarily took place on our local devices in the ros_ws. Within this workspace, we had the 2023_competition package which was given to us, and the my_controller package, which we created ourselves. The my_controller package contained 3 directors which we worked off of, being the launch folder, scripts folder, and model folder. The launch folder contained the launch files for running the controller scripts, the scripts folder contained the script that was run, and the model folder contained the trained CNN for sign reading, as well as the training data that was used for the model. The training data was collected, processed in collab, and then returned as the model weights which were used by the controller scripts.

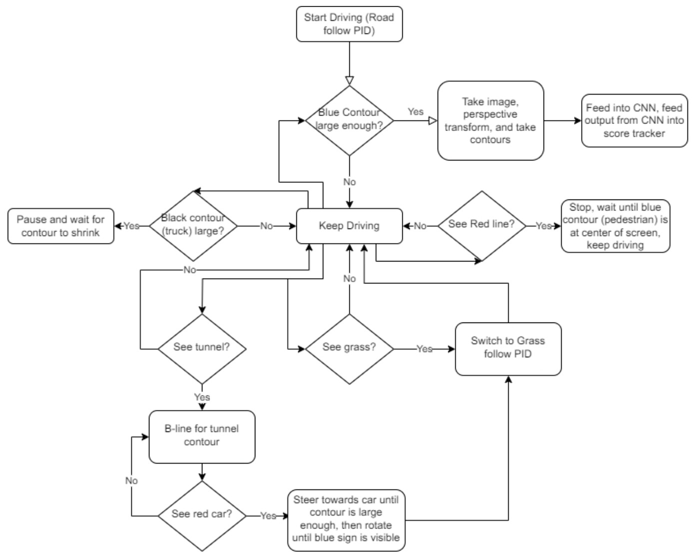
Discussion:

Driving and clue plate recognition modules

Our code was distinctly split into two modules: driving and clue recognition, which was also how we distributed the workload between our two group members. These modules were designed to be completely independent of each other, with the robot being able to navigate completely independently of the clue plate recognition module, and vice versa.

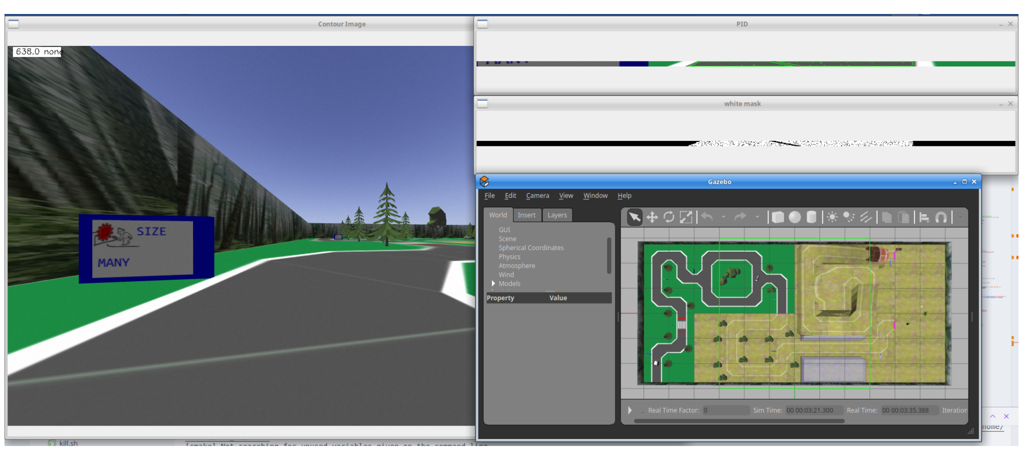
Driving:

Navigation followed an algorithm described by the flowchart below:

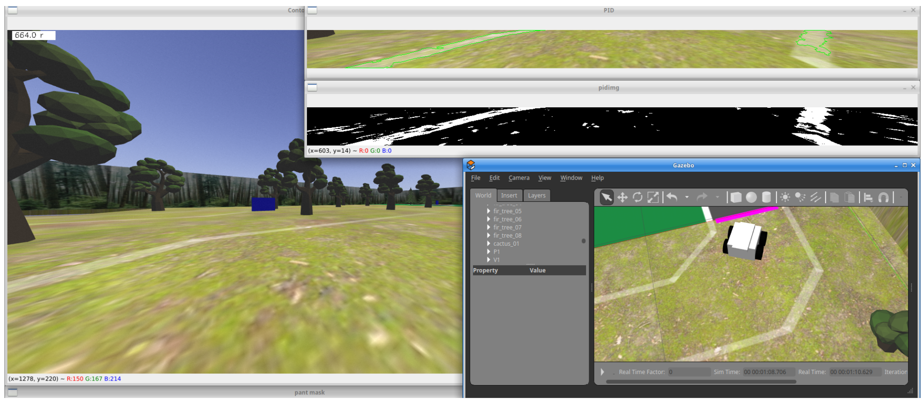


Control:

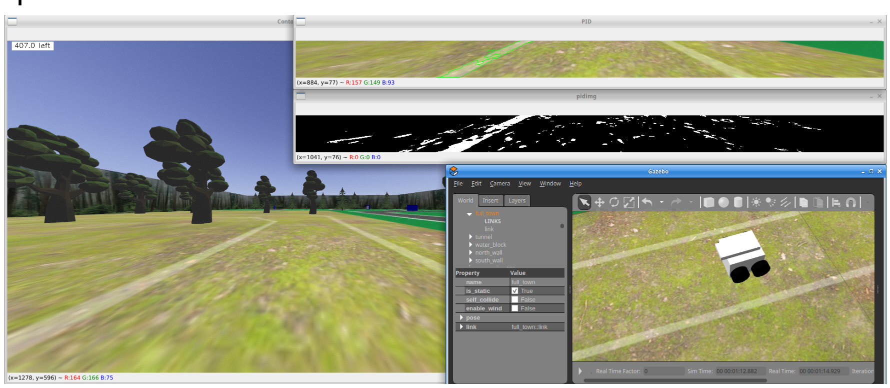
The robot starts driving using its road-following PID algorithm. This algorithm first filters out the gray of the road from a cropped captured image from the robot's frontal camera through binary masking. The image captured from the robot is cropped to a thin ROI (region of interest) to prevent artifacts. This mask is then traced over using contours, and the center of mass of all contours is taken. Depending on how offset the center of mass is from the center of the robot's camera output, a move command to steer is published. The farther the center of mass is from the center of the screen, the more the robot was told to turn.



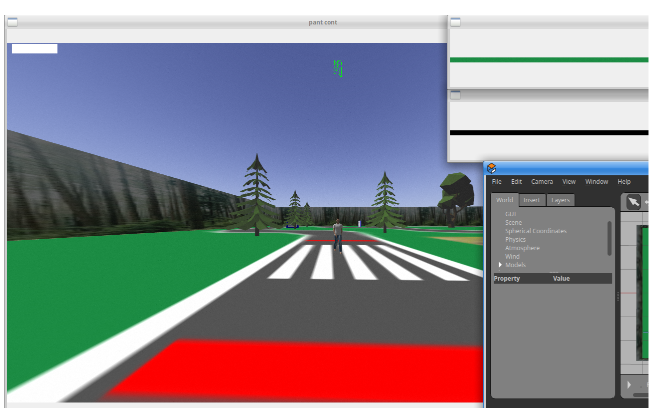
The first iteration of this algorithm filtered for the white borders of the road instead of the gray road itself. This was found to be too unstable under high speed to be usable for the road section, but was adapted to work with the grassy section. As the white lines on the ground are the only things that specify the correct path in the grassy section, it was most logical to use these white lines and navigate based on their position.



Some difficulties arose from filtering for these lines, as some patches of the grass were found to be of a similar HSV value to the white lines on the road. This was partially resolved by removing smaller than a certain size, however artifacts were never able to be fully removed from the contour images in this section. Therefore, precise tuning of both the threshold for removal of small contours and the threshold for the binary mask were necessary, as well as tuning of steering values to allow for correction if the robot failed to detect a stretch of white. The speed of the robot was also decreased to allow for these corrections.

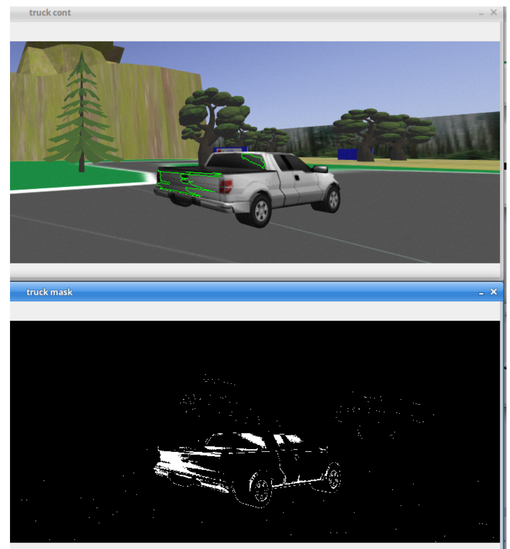


Pedestrian and Truck Detection:

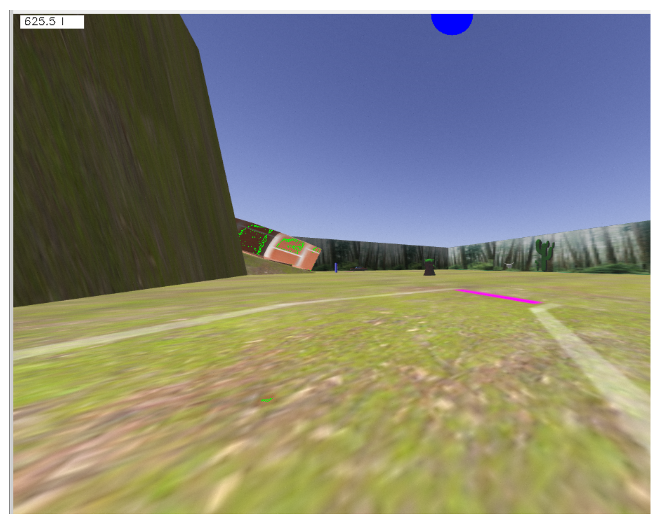


Pedestrian detection was designed to first only activate if a red strip was found to be right in front of the robot through taking a thin ROI at the bottom of the robot’s camera output. If red is detected, the robot stops as it knows it is in front of the crosswalk, which is marked using red strips in the simulation. Once stopped, the algorithm then shifts to scanning for a blue contour using binary masking, which captures the pedestrian’s blue pants. Once the blue contour is at the center of the screen, the robot becomes aware that the pedestrian had freshly crossed the road and is on cooldown, and is therefore safe for the robot to cross. The robot is then told to resume following the road, and the red scanning algorithm is disabled as there are no longer any crosswalks to worry about.

Truck detection works though scanning for black contours, representing the truck’s tires and bumper. This algorithm is only set to be active while road following is active, and functions through scanning for black through contours and a binary mask. Once the black contour becomes of a particular size, the robot is told to pause on the spot until the contour becomes smaller than the threshold. This allows for the robot to stop once the truck is in front of the robot at a certain distance, and only resume when the truck has gotten a certain distance away. This worked relatively consistently, but as the clue signs also contained some black adjustment of contour threshold size was necessary to allow for proper functioning.

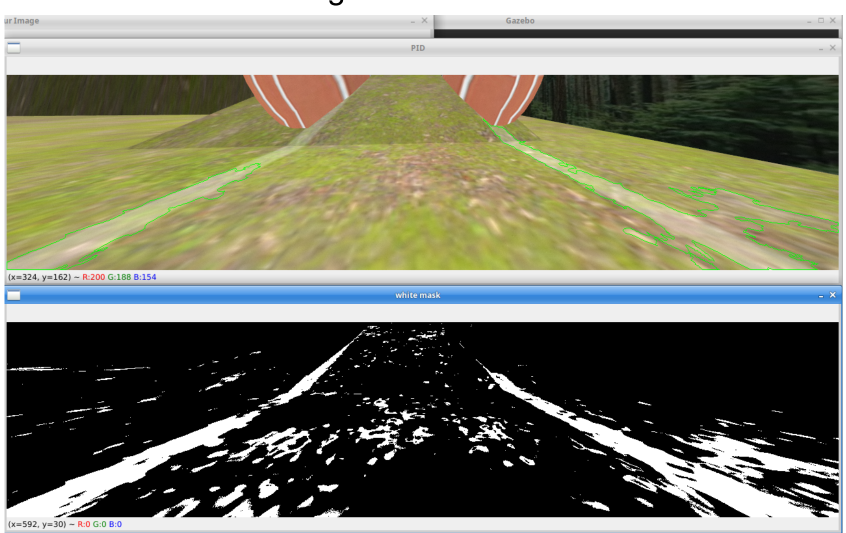


Baby yoda detection was not necessary due to our implementation of tunnel tracking. This algorithm functioned by turning towards the tunnel immediately after a brown-coloured area of a certain size was detected, and sprinting towards the center of mass of the contour. This allowed us to immediately track towards the tunnel at a sharp inner angle in the baby yoda section, which allowed us to ignore baby yoda entirely. To get to the right side of the



tunnel, an offset was introduced for steering so that the center of mass was kept on the left side of the robot at all times. While tracking the tunnel, the algorithm would simultaneously scan for a red contour representing the car placed beside the tunnel. Once the red car was visible to the robot's camera via the red contour becoming a certain size, the car would switch to tracking the red car instead to better align itself with the tunnel. Once the red car became of a certain distance, the robot would stop in place, and turn to the left until a blue contour, representing the sign, became of a certain size. At this moment, as the robot is now aware that it is located directly in front of the tunnel, it would engage a second grass follow mode tuned specifically for the mountain, and begin tracking through the tunnel and up the mountain.

As the robot climbs the mountain, it makes use of a specialized grass following algorithm specifically tuned for the mountain surface. This algorithm differed from the first grass following algorithm as it filtered for the two largest contours on the screen instead of simply filtering out small contours. This was implemented as streaks of white on the cliffside of the mountain would occasionally cause the robot to hug and ride up the cliffside, getting it stuck. Making use of this algorithm, the robot could successfully climb the mountain until it sees the final sign. Once this blue contour became of a certain size, the robot would go straight forward until the final sign was read.



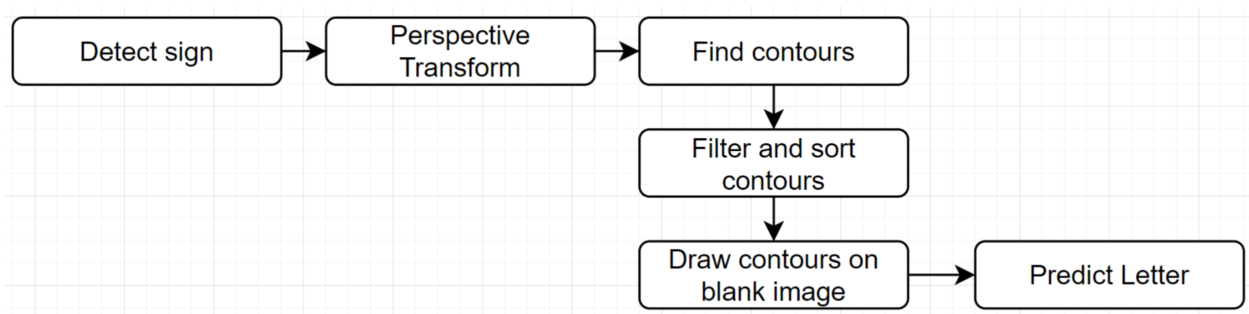
Edge cases:

Edge cases were "handled" by leaving the robot to run off course, and tuning to prevent it from happening further. Notable edge cases were the robot running off course in the roundabout, running off course in the grass section, and running into the mountain face in the mountain section. To prevent these, binary masks, contour thresholds, as well as

ROI were tweaked. It was found that once one upstream issue was fixed, many other edge cases that would have otherwise been propagated down would be automatically fixed as well, to the point where fixing the roundabout issue on the day of the competition ended up suddenly leading to multiple perfect runs in a row. This makes sense however, as small errors early on can easily propagate themselves into the rest of the algorithm, leading to misalignment to the tunnel, running off the mountain, etc.

Clue plate recognition:

Overview:



For clue plate recognition, we started with being able to detect if there was a sign within the robot's field of view, by applying a blue mask, drawing contours on the masked image, and then thresholding for a contour of a specific size. We then applied a perspective transform on the detected sign, which gave a clean, cropped image of just the contents of the sign.

With the cropped image, we found the contours on the sign and processed them by removing overly large and small contours, removing inner contours, and sorting the contours based on their y-position (separating into clue type or crime), and x-position (rearranging letters to read from left to right). We then drew the contours onto a blank image, which was what we used as the input into the CNN.

Data Engine:

From being able to gather letter contours from a sign, we collected data by driving around manually, and publishing a ros message to the controller telling the controller to record the contours of each letter in a list, and pickling the data to a file. Because each sign had two words with multiple letters, it only took a few passes through the entire course to have a dataset of around 1000 letter contours.

Data Processing:

Aside from the data processing of the image and contours outlined in the overview, the only other changes we made to the dataset was shuffling the contents before processing through the CNN to eliminate potential bias. It is notable that the images we sent into the CNN were 100 x 100, and contained just the outlined contour, which we felt kept only the minimal amount of data necessary, which may have led to faster training times.

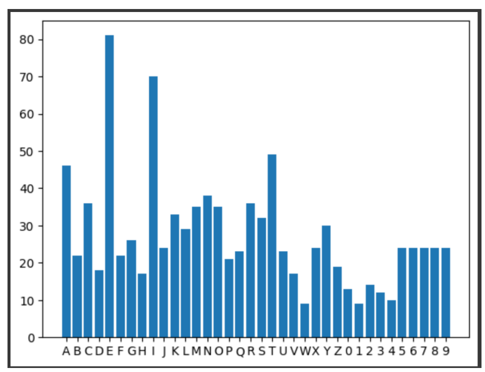
Neural Network Architecture:

Since we based our neural network off the one used in lab 6, the network's architecture ended up being identical. We did not see any problems with using those settings, and the speed was reasonable, so we did not see any reason to change the network's settings. The only thing that was changed was the input shape, as we were using 100 x 100 images.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 98, 98, 32)        320

 max_pooling2d (MaxPooling2  (None, 49, 49, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 47, 47, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 23, 23, 64)        0
 g2D)

 conv2d_2 (Conv2D)           (None, 21, 21, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 10, 10, 128)       0
 g2D)

 conv2d_3 (Conv2D)           (None, 8, 8, 128)         147584

 max_pooling2d_3 (MaxPoolin  (None, 4, 4, 128)         0
 g2D)

 flatten (Flatten)           (None, 2048)              0

 dropout (Dropout)           (None, 2048)              0

 dense (Dense)               (None, 512)               1049088

 dense_1 (Dense)             (None, 36)                18468

=================================================================
```

Training Parameters:

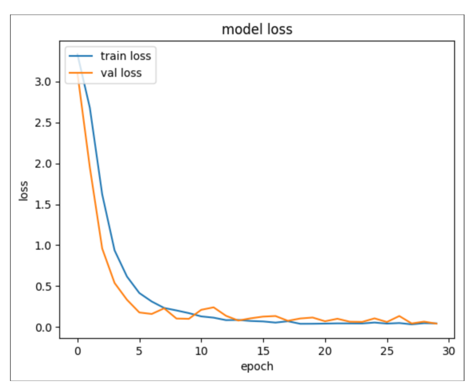
When collecting data for the CNN, we noticed certain letters and numbers were either very uncommon, or not present in the sign generation words. To account for this, we did a couple of collections runs where we changed the sign generation to show strings of letters like "XXZZYYGGBB44". This led to a more even distribution for the training data set. The total data set size was 993 letters, with a validation split of 0.2



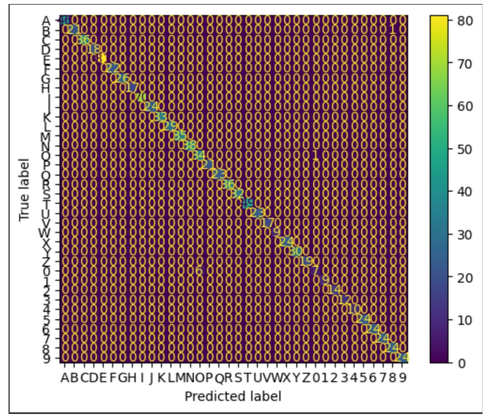

```
X shape: (993, 100, 100, 1)
Y shape: (993, 36)
Total examples: 993
Training examples: 795
Validation examples: 198
```

Training and Validation:

The learning rate was kept at 0.0001, and we trained the model for 30 epochs, but was almost fully trained by the 15th epoch.



Error Analysis:



We created a confusion matrix for our training, and noticed that there were issues with 0's and O's, and once case of 8 and B being mixed up. We decided that these rates were acceptable, and justified the 0's and O's being bad camera placement when we took the photos manually, as the 0's or O's would be distorted based on the perspective transform

Conclusion

Summary

Our robot placed fourth during the competition. Our robot performed exactly how we programmed it to, and managed to complete the full course as well as guess every clue correctly, achieving maximum points. During our testing, we found that our robot occasionally misread signs or deviated from the path, but was deemed reliable enough for competition day.

Given a second chance, we would have liked to explore other means of navigation, such as using neural networks or using imitation learning. We would have also liked to experiment with other neural network settings, playing with ideas such as directly inputting the contours into the network as opposed to images of the contours. We also felt that we could have structured our code better, by using more files for better modularity, creating more functions for repeated code, and documenting code better.

References:

- https://projectlab.engphys.ubc.ca/enph-353/
- https://chat.openai.com/

Appendix:

1. ChatGPT
    a. Ideas for a general outline of sign detection

You
using openCV, what would be the best way to train a CNN to read letters off a sign

ChatGPT
Training a Convolutional Neural Network (CNN) to read letters off a sign using OpenCV involves several steps. Here's a high-level overview of the process:

1. Dataset Preparation:

- Collect a dataset of images containing signs with letters.
- Label each image with the corresponding letters present on the sign.
- Split the dataset into training and testing sets.

2. Image Preprocessing:

- Resize images to a consistent size.
- Normalize pixel values to a range suitable for training (e.g., [0, 1]).
- Consider augmenting the dataset with transformations like rotation, scaling, and flipping to increase variability.

3. Model Selection:

- Choose a CNN architecture suitable for image classification. Common choices include VGG, ResNet, or custom architectures.
- Adjust the input layer to match the size of your resized images and the output layer to match the number of classes (letters).

4. Model Training:

- Compile the model with an appropriate loss function (e.g., categorical crossentropy) and optimizer (e.g., Adam).
- Train the model on your training dataset using the prepared images.
- Monitor the training process for overfitting and adjust hyperparameters accordingly.

5. Evaluation:

- Evaluate the model on the testing set to assess its performance.
- Fine-tune the model or experiment with different architectures if needed.

6. Prediction:

- Use the trained model to make predictions on new images with signs.

7. Integration with OpenCV:

- Use OpenCV to preprocess images and extract regions of interest (ROIs) containing signs.
- Apply the trained model to classify letters within these ROIs.

python   Copy code

```python
# Example prediction using OpenCV
roi = preprocess_and_extract_roi(sign_image)  # Implement this function
predicted_class = model.predict_classes(roi)
predicted_letter = class_to_letter_mapping[predicted_class]
```

Remember to adapt the code to your specific requirements and dataset characteristics. Fine-tuning hyperparameters and experimenting with different architectures can also improve the model's performance. Additionally, you may need a sufficiently large and diverse dataset for effective training.

b. Specific ideas for processing the camera view and narrowing it down to just sign contents.

AV **You**

I am using contours to draw a rectangle over a sign in a gazebo environment. I want to find the corners of the sign in order to perform a perspective transform on the sign

ChatGPT

To perform a perspective transform on a sign in a Gazebo environment using contours, you'll need to follow these general steps:

1. **Image Processing:**
   - Obtain the image of the Gazebo environment.
   - Preprocess the image to enhance features or reduce noise (e.g., convert to grayscale, apply blurring, thresholding, etc.).
2. **Contour Detection:**
   - Use a contour detection algorithm (e.g., OpenCV's `findContours` function) to find contours in the preprocessed image.
3. **Filter Contours:**
   - Filter the contours based on their area, aspect ratio, or any other relevant criteria to identify the contour corresponding to the sign.
4. **Approximate Polygon:**
   - Once you have the contour of the sign, you can approximate the contour to a polygon using the `approxPolyDP` function in OpenCV. This will give you the corners of the sign.
5. **Perspective Transform:**
   - Use the coordinates of the corners obtained in the previous step to perform a perspective transform using the `getPerspectiveTransform` and `warpPerspective` functions in OpenCV.

Here's a simple Python code snippet using OpenCV for these steps:

c. Ideas for data collection

AV **You**

how would i gather images in a gzebo environment?

ChatGPT

In a Gazebo environment, you can generate synthetic images of signs with letters or capture images from simulated cameras attached to a robot. Here's a general outline of how you might gather images in a Gazebo environment:

1. Create a Gazebo World:

1. **Install ROS and Gazebo:**
   - Install the Robot Operating System (ROS) and Gazebo simulator on your system.
2. **Create a Gazebo World:**
   - Design or select a Gazebo world that includes signs with letters. You can use existing Gazebo models for signs or create custom models using tools like Blender.

2. Simulate a Robot:

1. **Select or Create a Robot Model:**
   - Choose a robot model or create a custom one that will navigate the Gazebo environment. Make sure the robot is equipped with a simulated camera.
2. **Launch the Gazebo Simulation:**
   - Launch the Gazebo simulation with your world and robot model. You can use ROS launch files to simplify this process.

3. Capture Images:

1. **Simulated Camera Configuration:**
   - Configure the parameters of the simulated camera attached to your robot. Set the camera resolution, field of view, and other relevant settings.
2. **ROS Camera Node:**
   - Use a ROS camera node to capture images from the simulated camera. ROS provides camera drivers and simulation tools that allow you to interface with Gazebo.
3. **Record Images:**
   - Run a ROS node to continuously capture images from the simulated camera. You can save these images to disk or publish them to a ROS topic for further processing.

d. Ideas for improving masking

AV **You**

what are some ways of improving an image to read letters off a sign using contours in opencv

ChatGPT

Improving an image to facilitate reading letters off a sign using contours in OpenCV can involve several preprocessing steps. Here are some techniques you can consider:

1. **Grayscale Conversion:**

   Convert the image to grayscale to simplify the processing.

   python Copy code
   ```
   gray_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)
   ```

2. **Thresholding:**

   Apply thresholding to segment the image and make the letters more distinguishable.

   python Copy code
   ```
   _, threshold_image = cv2.threshold(gray_image, 127, 255, cv2.THRESH_BI
   ```

3. **Smoothing:**

   Apply Gaussian or bilateral smoothing to reduce noise and make contours more distinct.

   python Copy code
   ```
   blurred_image = cv2.GaussianBlur(threshold_image, (5, 5), 0)
   ```

4. **Dilate and Erode:**

   Use morphological operations like dilation and erosion to enhance or diminish features.

   python Copy code
   ```
   kernel = np.ones((5, 5), np.uint8)
   dilated_image = cv2.dilate(blurred_image, kernel, iterations=1)
   eroded_image = cv2.erode(dilated_image, kernel, iterations=1)
   ```

5. **Contour Detection:**

   Find contours in the preprocessed image.

   python Copy code
   ```
   contours, _ = cv2.findContours(eroded_image, cv2.RETR_EXTERNAL, cv2.CH
   ```

6. **Filter Contours:**

   Filter contours based on area, aspect ratio, or other criteria to focus on potential letters.

   python Copy code
   ```
   filtered_contours = [contour for contour in contours if min_area < cv2
   ```

7. **Draw Contours:**

   Draw the filtered contours on a blank image to visualize and verify the results.

   python Copy code

```python
contour_img = np.zeros_like(original_image)
cv2.drawContours(contour_img, filtered_contours, -1, (0, 255, 0), 2)
```

8. **ROI Extraction:**

   Extract regions of interest (ROIs) based on the contours for further analysis.

   python Copy code

```python
for contour in filtered_contours:
    x, y, w, h = cv2.boundingRect(contour)
    roi = original_image[y:y+h, x:x+w]
    # Process the ROI further if needed
```

9. **Character Recognition (OCR):**

   Use an Optical Character Recognition (OCR) library, such as Tesseract, to recognize text within the ROIs.

   python Copy code

```python
import pytesseract
text = pytesseract.image_to_string(roi, config='--psm 6')
```

10. **Adaptive Thresholding:**

    Apply adaptive thresholding for better handling of varying lighting conditions.

    python Copy code

```python
adaptive_threshold = cv2.adaptiveThreshold(gray_image, 255, cv2.ADAPTI
```