CS/ME/ECE/AE/BME 7785

Lab 6 and Final

Part 1 (LAB 6): April 9^{th}, 2025 at 11:59 pm

Part 2 (FINAL): April 21^{st}, 2025 at 4 pm

1 Overview

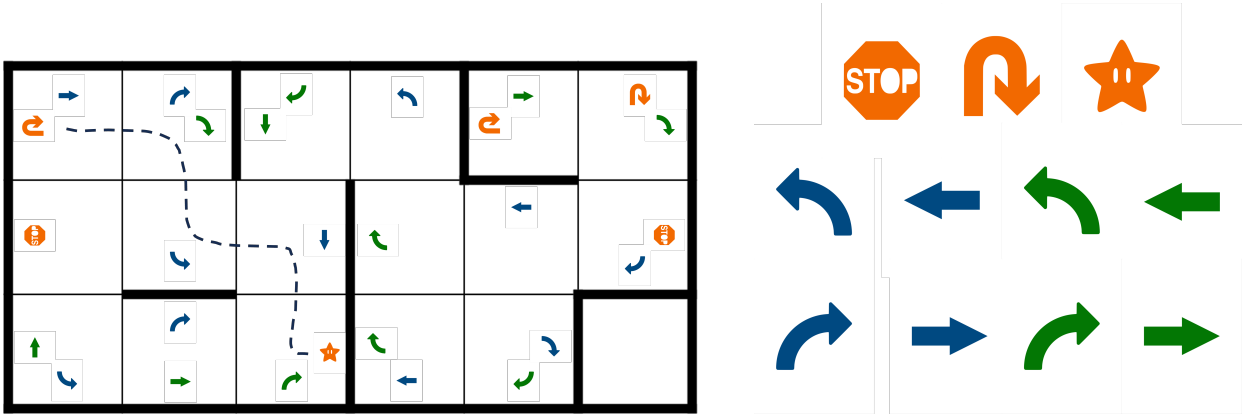
The objective of the final project is to integrate multi-modal sensing and navigation into a single reasoning system. Our goal will be for the robot to complete a maze by following signs. As always you can solve this problem however you would like. Note: Since this is your final project, you may use any ROS configuration for computation that you would like. Everything can be run on the robot or you can have specific nodes processing data and sending commands from your computer. However, if you run nodes on your computer (e.g. passing images to do images processing on your machine) you must accept the risk or have a backup strategy if the GT network is slow that day.

The robot will be placed in a world such as the one pictured below. You may map the environment a priori and you will be able to localize your robot before starting. Your objective is for the robot to find the goal (designated by the red target sign in this example) in the shortest path possible dictated by the signs in the maze. The location of the goal nor your starting position will not be known ahead of time, but signs throughout the world will always direct you from any initial starting position to the goal position.

In the example world presented in Figure 1, if the robot were to start in the top left grid cell, it would be able to follow the signs on the wall in order to reach the goal represented by the star.

Eleven different signs will be present in the world, organized into four categories: wrong way (stop and turn around signs) indicating the robot should turn around, goal, turn 90 degrees to the left (four left arrow signs), and turn 90 degrees to the right (four right arrow signs). The signs are colored and taped to the walls of the robot space.

The project is split into two parts, a **Vision Checkpoint (Part 1)**, and the **Final Demo (Part 2)**. As always, we strongly encourage you to use all available resources to complete this assignment. This includes looking at the sample code provided with the robot, borrowing pieces of code from online tutorials, and talking to classmates. You may discuss solutions and problem solve with others in the class, but this remains a team assignment and each team must submit their own solution. Multiple teams should not jointly write the same program and each submit a copy, this will not be considered a valid submission.



(a) (b)

Figure 1

2 Part 1: Vision Checkpoint (Due April 9^{th})

The first step of the project is to create an algorithm to classify the signs, without the robot running. For this checkpoint, we are only testing image classification performance.

You may design any solution of your choice to identify the signs. If you want to hand-code a solution, that is acceptable. The best results, however, will likely be achieved with a classification algorithm, and we are providing two suggested solutions:

Use KNN and OpenCV: You are already somewhat familiar with OpenCV, so one solution is to use the tools provided within. We are providing example code for the use of KNN, although you will need to improve it to get good performance. The current code reads the images in grayscale, and represents them simply as an array of raw pixel values. The resulting model achieves low accuracy. Suggested improvements to the model (in order of importance):

- Crop the image to focus classification only on the sign portion instead of the entire image.
- Incorporate other high-level features (color, shape, gradients, etc.)

Use scikit-learn toolkit (Python only): scikit-learn provides a wide range of machine learning tools, including classification algorithms. For those who don't mind adding another tool to their software arsenal, scikit will likely provide the greatest benefit and best overall performance.

You may also use a different third-party library or tool if you wish. Also feel free to collect more training data to train your model. Remember, for your final, this classification algorithm must be able to run in real time on your computer (passing images from the Turtlebot to your PC) or onboard the Turtlebot entirely.

Getting Started

Read through this section carefully to understand assignment requirements and expected submission guidelines. Failure to ensure correct use of the provided grading and submission files and improper submission formatting will directly impact our ability to grade your submissions.

Files Provided

- `YYYYFimgs/` — A directory containing example images of robot signs.
- `YYYYFimgs/labels.txt` — A text file with image names and their corresponding class labels.
  - **labels:** 0: empty wall, 1: left, 2: right, 3: do not enter, 4: stop, 5: goal.
- `knn_example.py` — An example KNN classifier using OpenCV.
- `README.md` — Instructions for running `knn_example.py`.
- `model_grader.py` — The grading script into which you will integrate your model.
- `generate_requirements.py` — A helper script to generate a minimal `requirements.txt`.

First Steps

Before you begin, in addition to **python3** and **pip**, make sure to install all the prerequisites:

- Required Python packages are `opencv-python`, `numpy`, and `pipreqs`. Install these using:

  ```
  pip install opencv-python numpy pipreqs
  ```

- It is highly recommended that you use a virtual environment to manage your project dependencies. For detailed instructions on how to create and use a virtual environment, please refer to this guide: https://realpython.com/python-virtual-environments-a-primer/.

Dataset Overview

The dataset provided for this lab consists of a directory of images (e.g., `YYYYFimgs/`) and a corresponding `labels.txt` file. The `labels.txt` file contains lines in the following format:

```
222, 1
345, 4
```

In this example, the line `"222, 1"` indicates that the file `222.png` belongs to class 1, and `"345, 4"` indicates that `345.png` is in class 4.

Preparing the Dataset

It is your responsibility to prepare the data for effective model training and evaluation. This involves:

- **Splitting the Dataset:** It is strongly recommended to split the dataset into training and validation sets. A common practice is to use an 80/20 or 70/30 split, ensuring that the model is trained on a majority of the data while retaining a validation set for ensuring performance.
- **Data Augmentation:** To improve the robustness of your model and mitigate issues related to a limited dataset, it is highly recommended that you employ data augmentation techniques. Common augmentation methods include:
  - Random cropping
  - Rotation
  - Horizontal/vertical flipping
  - Scaling
  - Color jittering

These techniques help simulate variations in real-world conditions, thereby increasing the diversity of your training data and reducing overfitting. You can use openCV tools to create augmented dataset. Alternatively, existing python packages such as **albumentations** are also great for this purpose.

- **Collecting Additional Data:** If you find that the provided dataset is insufficient for achieving high accuracy, you are encouraged to capture additional images and manually label them. Ensure that any additional images follow the same naming and labeling conventions as the provided data.

Key Points:

- It is your job to make the dataset better and more usable by creating a proper split between training and validation sets.
- Improving the dataset through augmentation or additional data collection is essential for building a robust model.
- Always use relative paths when referencing data files in your code.

Training Model

Your training code should output a trained model file (if applicable) that can be loaded during grading. This code should take the form of a standalone notebook or python script that you must create yourself. It should be completely separate from the `model_grader.py` script.

Scoring Model

For grading purposes, you must integrate your trained model into the provided grading script (`model_grader.py`). If you had split the data set into train and validation sets, using the validation set for the grading script may provide results closer to what you can expect when your submission is graded with the withheld data set. **You are only allowed to modify the following sections in `model_grader.py`:**

1. **imports:** Add necessary package imports you need for your model and to perform predictions.
2. **initialize_model(model_path=None):** Implement this function to construct and load your trained model. Use the optional `model_path` argument to load the pre-trained model file that can be passed in via command line argument, if applicable.
3. **predict(model, image):** Implement this function to perform inference on a single image (provided as a NumPy array) and return the predicted class label as an integer.

The function arguments and the remainder of the grader script must remain unchanged. When running the grader script, use the following command-line flags:

- `data_path` to specify the validation dataset directory (which must contain a `labels.txt` file and image files).
- `model_path` (if applicable) to provide the path to your trained model file.

For example:

```
python3 model_grader.py --data_path ./val_dataset --model_path ./my_model.pkl
```

Prepare for Submission

Before submitting, you must generate a minimal `requirements.txt` file that lists only the packages your project depends on. We provide a helper script, `generate_requirements.py`, which uses `pipreqs` to scan your project directory and create this file. It also prepends your Python version as a comment at the top. To run the script, execute:

```
python3 generate_requirements.py
```

This will create or update `requirements.txt`. Make sure to check over the generated file manually to ensure successful execution of this script.

Submission

Submit a single zip file named `LastName1_LastName2_Lab6.zip` containing all of the following at the root level:

- Your training code (e.g., `your_training_code.py`).
- Your trained model file (if applicable).
- `model_grader.py` (with only the two designated functions modified).
- The generated `requirements.txt` from the `generate_requirements.py` output.
- A `readme.txt` with instructions on how to run your code.

All file paths in your code must be relative (do not use hardcoded absolute paths).

Verifying Submission Functionality

Testing your submission in the correct folder layout is essential. You can verify the submission formatting and functionality by ensuring that the `model_grader.py` script is able to run without error. From the submission directory, run the grader script as follows:

```
python3 model_grader.py --data_path ./val_dataset --model_path ./my_model.pkl
```

Expected Submission File Structure

Below is an example of the expected directory structure for your submission:

```
Submission_Directory/
|-- model_grader.py
|-- requirements.txt
|-- your_training_code.py
|-- your_trained_model_file (if applicable)
|-- readme.txt        % Optional: To explain any caveats we should know when grading
|-- supplemental/     % Optional: For helper functions/additional files if needed
```

Grading

Your grade will be based on the classification accuracy of your algorithm on the withheld test set. For example, if your model achieves an accuracy of 93%, your grade will be 93. The grading script will output a confusion matrix and overall accuracy. Your submission will be graded using a withheld dataset that you will not have access to.

Points of Emphasis and Additional Notes

- You are only allowed to modify the designated sections of `model_grader.py` (i.e., imports, `initialize_model()` and `predict()`).
- Ensure that your code for the `model_grader.py` uses relative paths only.
- Test your submission thoroughly in the expected directory layout.
- Failure to include a proper `requirements.txt` file or to follow the directory layout may result in issues during grading.

Improving Model Results

Proper data preprocessing is a crucial step in both training and prediction. Effective preprocessing can drastically improve the performance of your model by reducing noise, highlighting relevant features, and ensuring consistency in the data that the model sees. Consider the following key points:

- **Image Cropping:** Focus on the sign portion of the image rather than the entire frame. Cropping reduces background noise and ensures that the model primarily learns from the relevant region.
- **Normalization:** Standardize pixel values (e.g., scaling them to the range [0,1] or normalizing to have zero mean and unit variance). This helps the model converge faster and improves numerical stability.
- **Feature Extraction:** Instead of relying solely on raw pixel intensities, consider extracting features like edges, gradients, or color histograms. For instance, applying a Sobel filter to extract edges or using histogram equalization can enhance contrast and detail in the image.

3 Final Demo (Due April 21^{st})

For the final demo, your robot should navigate the maze set up in the lab. To aid in developing code for this component, we have created an augmented version of the Gazebo environment provided in lab 5 that may be used for for testing. The Gazebo environment is identical to the one in the lab, and provides an idealized environment for testing. The files to use it can be found here (https://github.gatech.edu/7785-Intro-to-Robotics-Research/Lab6_Gazebo), We highly recommend debugging in Gazebo before starting on the robot as there is only one real maze for the entire class to share development time in.

The figure on the right (Figure 2) shows an example map of the maze your robot would need to navigate. For testing, the robot could start in the bottom left, facing South or West. The goal is the star on the right side of the maze, third gridcell from the bottom, facing North. A significant number of signs such as the ones shown in the map will be present to guide the robot to the goal.

If your robot is lost, consider having it turn in place to check other orientations. Note that we recommend that the robot check for signs only when stationary and facing a wall directly in front (which can be verified using the LIDAR data).

The world will contain only right angles. As a result, you can assume that a right/left turn arrow indicates that the robot should perform a 90° turn and drive to the next wall from there. The stop and do-not-enter signs signal that the robot should turn around and drive to the next wall. Note, there will be errors in our (read, the TAs and Sean's) creation of the maze. We will do our best to make 90° angle walls but in reality they may be slightly off from these ideal orientations.

Some parts of the world near the wall can be a bit dark due to the lighting in the room. If needed, you can use a cell phone or flashlight to help light up the area when your robot is running.

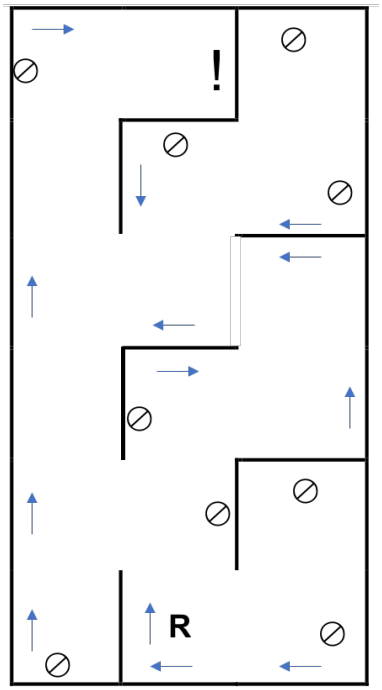


Figure 2

3.1 Grading

Consider the example map of the world as a large 3x6 occupancy grid. Project grade will be determined based on the following formulas:

If the goal is reached:

$$grade = 1 - (|r - n| * 0.01) - 0.1 * c$$

If the goal is not reached:

$$grade = (1 - \frac{d}{n}) - (|r - m| * 0.01) - 0.1 * c$$

where:

- n is the length of the optimal path to the goal when following the signs.

- r is the length of the path taken by the robot.
- d is the distance remaining from the robot's furthest progress point to the goal.
- m is the length of the optimal path to the robot's furthest progress point.
- c is a boolean value (0 or 1) indicating if the demo was terminated due to a collision with a wall or the user stopping the experiment without the robot indicating it reached the goal.

All above distances are Manhattan distances. The robot's furthest progress point is defined as the point along its path that is closest to the goal. Note that the above equations are basically identical if you consider that $d = 0$ and $m = n$ in the case that the goal is reached.

Example grading scenarios given the map on the previous page (incorrect robot actions are underlined for clarity):

- The robot correctly turns right two times, drives north, turns right, drives to the east wall, turns left, drives north, turns left, drives to the west wall, turns right, drives north, turns right, drives east, and finds the goal.

$$grade = 1 - (|9 - 9| * 0.01) - 0 * 0.1 = 1(100\%)$$

- The robot correctly turns right two times, drives north, turns right, drives to the east wall, turns left, drives north, turns left, drives to the west wall, <u>turns left</u>, drives south, turns around, drives north, turns right, drives east, and finds the goal.

$$grade = 1 - (|15 - 9| * 0.01) - 0.1 * 0 = 0.94(94\%)$$

- The robot correctly turns two times, drives north one block and cannot see a sign so begins to spin, sees the do not enter sign on the right, <u>turns around</u>, drives straight, turns right, drives straight, turns right, finishing in the goal cell but does not classify the goal sign correctly and the student terminates the experiment.

$$grade = 1 - (|7 - 9| * 0.01) - 0.1 * 1 = 0.88(88\%)$$

or

$$grade = (1 - \frac{0}{9}) - (|7 - 9| * 0.01) - 0.1 * 1 = 0.88(88\%)$$

- The robot correctly turns right two times, drives north, turns right, drives to the east wall, turns left, drives north, turns left, drives to the west wall, <u>turns left</u>, drives south, then stops and makes no further progress constantly spinning.

$$grade = (1 - \frac{3}{9}) - (|9 - 6| * 0.01) - 0.1 * 0 = 0.67(67\%)$$

- <u>The robot doesn't move</u>.

$$grade = (1 - 9/9) - (|0 - 0| * 0.01) = 0(0\%)$$

3.2 Demo and Grading Process

The world will remain unchanged for the grading demo, so you can continue using the same map you created for testing, if you created a map. However, we will change the robot's start and goal locations on the day of the demo, as well as the locations of the signs as appropriate. You will be able to localize your robot at the beginning of your run.

Demos will be conducted during **pre-assigned 20 minute time slots**. Each person will sign up for a time using the spreadsheet below and will be graded during this time. Only one team may demo at a time and only those who are demoing will be allowed into CoC 050. You may set up a robot in CoC 050 away from those demoing if you are signed up for one of the next two slots (i.e. you can start setting up 40 minutes before your session). Please work quietly and respect the work environment of the students demoing (they have priority).

This demo process means you should not plan on any last-minute hacking, since neither robots nor the maze world will be available for testing.

Each team will have one demo session in which to test. A second session will only be made available later in the day if there is a major hardware failure during the first run (i.e. LIDAR stops working). **NO LATE DAYS MAY BE USED ON THE FINAL PROJECT!**

Sign up sheet:here
https://docs.google.com/spreadsheets/d/1EA2pOLzfFvBbz6ZeNCjqhnkLkGSIDisQIy3zmMkWuPA/edit?usp=sharing

3.3 Submission

You have two required submissions for the final.

1 Your ROS package in a single zip file called `LastName1_LastName2_Final.zip` uploaded on Canvas under Assignments–Final Project.

2 A live demonstration of your code to one of the instructors during your specified timeslot.