AA 274: Principles of Robotic Autonomy
Problem Set 2: Perception
Problems 1 & 2 Due Feb 6th 11:59PM
Problems 3, 4 & 5 Due Feb 13th 11:59PM
TurtleBot Demo Due before Feb 13th ROS OH

Starter code for this problem set has been made available online through github; to get started download the code by running `git clone https://github.com/StanfordASL/AA274_HW2.git` in a terminal window.

You will submit your homework to Gradescope. For each submission (Feb 6th and Feb 13th, your submission will consist of (1) a single pdf with your answers for written questions (denoted by the symbol) and (2) a zip folder containing your code for the programming questions (denoted by the symbol).

To avoid any git merge conflicts from getting the submission script from the repo, run the following lines in your `AA274_HW2` folder to zip up your code:

```
$ zip -r hw2_part1.zip Problem_1 Problem_2
$ zip -r hw2_part2.zip Problem_3 Problem_4
```

Your written part must be typeset (e.g., LaTeXor Word).

Introduction

For this homework, you will explore different elements of the perception module. In particular you will investigate the following:

1. Camera Calibration
2. Line Extraction from Lidar data
3. Machine learning (SVM) for pedestrian detection
4. Classification and sliding window detection
5. Object (stop sign) recognition and integrating this in a Finite State Machine (FSM) in ROS.

Further, in terms of software development, you will

- Get into the hairy details of Python's `numpy`. For some basic information about numpy, read:
  - https://stanfordasl.github.io/aa274/pdfs/recitation/python_review.pdf
  - http://cs231n.github.io/python-numpy-tutorial/#numpy

  - https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html

- Implement a recursive function
- Use Tensorflow and its estimator framework
  - Read https://www.tensorflow.org/guide/estimators for an introduction to Tensorflow's Estimator framework.
- Learn about retraining a pretrained model for image recognition
- Use Tensorboard to visualize a neural network model and observe the training process
  - Read https://www.tensorflow.org/guide/summaries_and_tensorboard for more information about Tensorboard
- Implement multiple rosnodes that integrate the control and perception modules within a FSM.

  We strongly encourage you to take a look at all the code, even if it is not sections that you will be explicitly working on.

Problem 0: Final Project Lab 1

As we progress into Homework 2, it is time to start working with the TurtleBots! If you have not already done so, get yourself into a group of **5-6** people by adding your name to the following list: https://docs.google.com/spreadsheets/d/1XqVtdUDBXzeQyYXiddsOWzccT1WCfhCyTkoyhZa-B5o/edit?usp=sharing.

As a part of Homework 2, we will require at least one member from each team to come to the ROS Office Hours (Wed/Thurs from 5-7pm in Skilling) to demonstrate key components of Homework 1 on an actual robot. With a successful demonstration, and being able to answer the questions, you and your team will get checked off.

For this homework, you will familiarize yourselves with the interfacing with the TurtleBots. Namely, the goals of this question are:

- To connect to the TurtleBot from your VMs.
- To remotely teleoperate the TurtleBot.
- To run the parking controller from HW 1 Problem 5 on the TurtleBot.
- Familiarize yourself with the running ROS nodes across multiple computers.

Getting asl_turtlebot

We have made significant changes to the asl_turtlebot repo. Since the only changes you made in homework 1 was in controller.py, please copy this somewhere safe, delete the repo, and clone asl_turtlebot again.

1. In asl_turtlebot, copy your controller.py file to somewhere safe and not in asl_turtlebot.
2. Delete the asl_turtlebot folder
3. Reclone the asl_turtlebot folder

```
$ git clone https://github.com/StanfordASL/asl_turtlebot.git
```

4.
```
$ cd ~/catkin_ws && catkin_make
```

Connecting to the TurtleBot

First, we must take some steps to configure the VM in order to be able to connect to a TurtleBot. You will see `rostb3.sh` and `roslocal.sh` in the `asl_turtlebot` folder. These files are important for telling your computer where `roscore` lives. Take a look at the files. Can you figure out roughly what is it doing?

1. Copy those files in your home directory.
2. Connect to the correct network. (The TA will tell you which one it is.)
3. Edit `rostb3.sh` accordingly.
4. Edit your `.bashrc`. Whenever you open a terminal, it runs all the commands in this file.

   ```
   $ gedit ~/.bashrc &
   ```

   Add the following lines at the end of the file,

   ```
   alias rostb3='source ~/rostb3.sh'
   alias roslocal='source ~/roslocal.sh'
   export TURTLEBOT3_MODEL=burger
   ```

   IMPORTANT: This will create an alias for `rostb3` and `roslocal`. If `roscore` is to run on the TurtleBot, and you want to run nodes from your computer (not ssh), you must type `rostb3` EVERY TIME you open a terminal window. Otherwise if you want to run things locally on your machine (e.g., doing Problem 5), you should run `roslocal`.[1]
5. For these modifications to take effect in the current terminal, run

   ```
   $ source ~/.bashrc
   ```

Next, in a terminal window, ssh into the TurtleBot using

```
$ ssh aa274@<TurtleBot Name>.local
```

with the password `aa274`. This remotely logs into the onboard robot computer. The necessary ROS packages and drivers for TurtleBot operation have been pre-installed so we can go ahead and run

```
$ roslaunch turtlebot3_bringup turtlebot3_core.launch
```

to launch core packages to start up the TurtleBot.

1. ssh into the TurtleBot from another terminal window. We can start exploring the existing ROS topics. What are all the messages that are being published right now? In particular, look at the `\odom` topic. What is the message type being published to this topic and what information is contained within these messages? HINT: `rostopic info \odom` might help.
2. In the same (ssh-ed) terminal window, begin teleoperating the robot by running:

[1] Alternatively, you can copy the lines from `rostb3.sh` to your `~/.bashrc` to automatically run them when you open a new terminal. Just remember to reset it when you aren't working with the turtlebot and working locally.

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

3. Now try teleoperating the TurtleBot, but without ssh-ing into the TurtleBot. How would you control it from your machine? Afterwards, teleop the TurtleBot back to (0,0,0).

4. Now let's try to run the parking controller from Homework 1 Problem 5 on the actual TurtleBot. In the `asl_turtlebot` package, you should have a node called `pose_controller.py`. Edit this file. (You should paste the relevant controller code from hw1 into the `get_ctrl_output` function.)

5. Take a look at `hw2_control_demo.launch`. What does this launch file do?

6. On your local terminal, launch the hw2 demo launch file. (What must you make sure before doing this?)

```
$ roslaunch asl_turtlebot hw2_control_demo.launch
```

7. You can publish messages to a topic through the command line. In another terminal, publish the desired goal pose to the topic `/cmd_pose`.

```
rostopic pub /cmd_pose geometry_msgs/Pose2D "1.0" "1.0" "0.0" -r 10
```

8. Another way to set desired goal poses is to use Rviz! Open up Rviz by going into the Rviz folder, then running

```
$ rviz -d signs.rviz
```

   You should be able to see an axis frame (you will learn about tf frames in the next homework) that corresponds to the TurtleBot, and the odom frame. Click `2D Nav Goal`, and set a desired goal pose (not too far away!). We have written a node (`hw2_demo.py`) that subscribes to the goal poses from Rviz, and published them to the `/cmd_pose` topic. Feel free to take a look. This is representative of what you should make for your final project.

9. (Optional) What is a `rosparam`? Type:

```
$ rosparam get sim
```

   (Remember to type `rostb3` before you do this!) Take a look at `pose_controller.py` and see how `rosparam` is used.

10. (Optional) Take a look at `http://wiki.ros.org/rviz/DisplayTypes/Marker`. Experiment with different markers. Rviz subscribes to various marker topics and can show them. Can you make a marker to represent the TurtleBot?

Problem 1: Camera Calibration

In this problem, the objective is to estimate the intrinsic parameters of a camera, which will allow you to accurately project any point in the real world onto the pixel image output by the camera.

To accomplish this, we will be using a popular method proposed in Z. Zhang, "A Flexible New Technique for Camera Calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2000 (there are a couple of versions online; use the version here: `citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.220.534`). This method uses images of a known pattern on a 2D plane, such as a chessboard, captured at different positions and orientations. By processing at least 3 (and preferably many more) images, the camera parameters can often be accurately estimated using a direct linear transformation as described in lecture.

In performing this calibration, it will be important to keep the relevant coordinate frames in mind (the paper by Zhang will be the main reference, so note any differences in notation from Lecture 5):

- (X, Y, Z) A point in a world coordinate system (attached to each checkerboard)
- (x, y) Ideal, distortion-free, normalized image coordinates
- (u, v) Ideal, distortion-free, pixel image coordinates
- $(\breve{x}, \breve{y})$ Real, distorted, normalized image coordinates
- $(\breve{u}, \breve{v})$ Real, distorted, pixel image coordinates

The observed points we extract in the $(\breve{u}, \breve{v})$ frame for calibration can be denoted by $(u_{\text{meas}}, v_{\text{meas}})$.

The scripts `cam_calibrator.py` and `cal_workspace.py` are given to provide a framework for the calibration. You will be editing methods in the `CameraCalibrator` class which `cal_workspace.py` calls. Please take a look at the code and see how `cam_calibrator.py` and `cal_workspace.py` interact before you begin.

This part uses the `camera_calibration` ROS package, please make sure you have this installed by running the following command

```
$ sudo apt install ros-kinetic-camera-calibration
```

To take a look at the chessboard images that you will be processing, run `./cal_workspace.py`. The corner grid is 7×9 and the side length of each square is $d_{\text{square}} = 20.5$ mm. The corner locations $(u_{\text{meas}}, v_{\text{meas}})$ for each chessboard are extracted for you using OpenCV. You should see something like in Figure 1 (click on the image to go to the next image).

Let's begin!

Note: You do not need to include these images in your write-up (unless you want to for your own future reference). While grading, we will run your code and these images should be generated.

(i) Modify `genCornerCoordinates` to generate the world coordinates (X, Y) for each corner in each chessboard. It is important that the ordering corresponds exactly to the points in $(u_{\text{meas}}, v_{\text{meas}})$!

(ii) Next modify `estimateHomography`, using the singular value decomposition (SVD) method outlined in Appendix A of [1] to estimate the homography matrix H for each chessboard.

(iii) Use SVD again in `getCameraIntrinsics` to estimate the linear intrinsic parameters of the camera, using the homographies H. These parameters should be packed into a single matrix A (see section 2.1 and Appendix B). As a sanity check, the skewness parameter γ should be small ($|\gamma| \ll \alpha$) and the principal point (u_0, v_0) should be near the center of the image pixel dimensions.

(iv) Next modify `getExtrinsics`, use your estimated A and the H for each chessboard to estimate the rotation R and translation t of each chessboard when the images were captured. (Note that your initial R estimates will likely not be genuine rotation matrices! Once again, SVD comes to the rescue — see Appendix C in [1] for details.)

(v) You are now in a position to create some important coordinate transformations. Implement `transformWorld2NormImageUndist` and `transformWorld2PixImageUndist` in order to switch from (X, Y, Z) to (x, y) or (u, v) in the undistorted image frames. It will be helpful to make use of homogeneous and inhomogeneous coordinates.

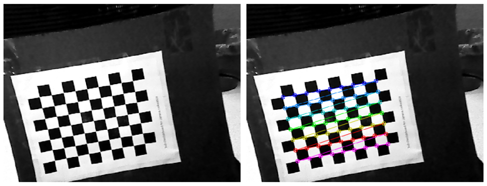

Figure 1: Corner extraction of chessboards.

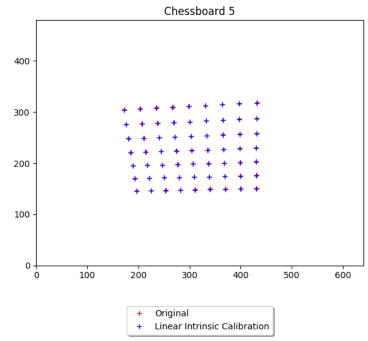

Figure 2: Example plots for (v)(a).

(a) Now you can check to see how well you are doing! Pass your estimated camera matrix A and chessboard extrinsic parameters R and t into the `plotBoardPixImages` function (leave the k argument unspecified) to see where your calibration is mapping the corners, compared to the original measurements. Refer to Figure 2 to see what the expected results should be. (Click on the image to move to the next image.)

(b) As a second check, pass your extrinsic parameters to `plotBoardLocations` to see the estimated locations and orientations of the chessboards relative to the camera. Refer to Figure 3 to see what the expected results should be. (Hit enter in the terminal to move to the next image.)

(vi) You will finish your camera calibration by applying radial distortion parameters k. These can be estimated using least-squares or nonlinear optimization, as outlined in Section 3.3 of [1]. However, we will simply give you some reasonable parameters $k = [0.15, 0.01]$ to use. Fill in the final transforms `transformWorld2NormImageDist` and `transformWorld2PixImageDist` to make use of k.

(a) Finally you have reached the true payoff! First, check the new movement of the corners, once again using `plotBoardPixImages`, but this time including k as an argument. (Your plots should look similar to Figure 2, though with an extra set of crosses, click on the image to move to the next image.)

(b) Now pass your estimated camera parameters A and k into the `undistortImages` function to apply your calibration to the original chessboard images. You should be able to compare the original (left) to the image after applying A (center) and finally after adjusting for radial distortion (right). Your plots should look like Figure 4 (click on the image to move to the next image). You should see that the edges on the right figure are straight, while there should be a slight curvature (depending on the checkerboard angle) on the left and center image.

In ROS, the camera calibration parameters you have just calculated are often sent to a `set_camera_info` service broadcast by the package running a given camera. They are then packed into a `.yaml` file in a standard location from which they can be automatically loaded whenever the camera starts. Pass your calibration parameters into the `writeCalibrationYaml` function to generate this configuration file.

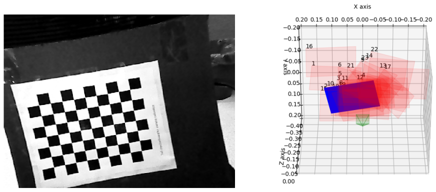

Figure 3: Example plots for (v)(b).

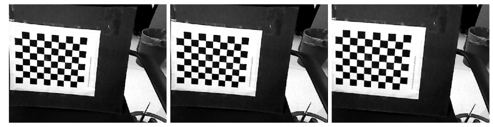

Figure 4: Example plots for (vi)(b).

Problem 2: Line Extraction

In this problem, you will implement a line extraction algorithm to fit lines to (simulated) Lidar range data. Consider the overhead view of a typical indoor environment:

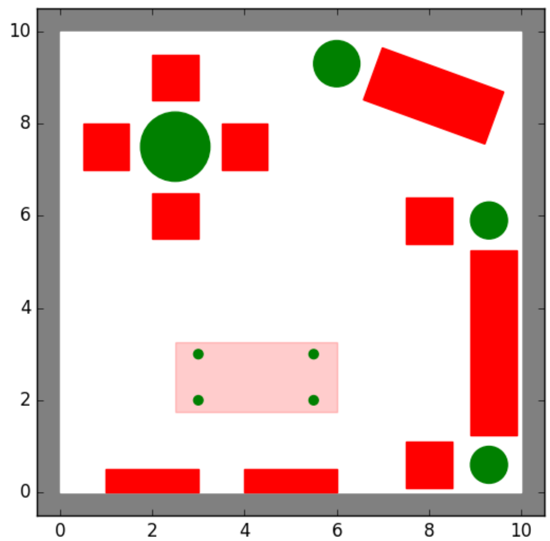

Figure 5: 2D Schematic of a typical 10 m x 10 m indoor environment.

A mobile robot needs to explore and map this room using only a (2D) LIDAR sensor, which casts equally spaced laser beams from the center of the robot to form a 360° view. The first step in mapping is to extract meaningful information from the range measurements. *Line Extraction* is a common technique used to fit a series of straight line segments to range data in an attempt to define the border of objects in the environment.

Line Fitting

A range scan describes a 2D slice of the environment. Points in a range scan are specified in a polar coordinate system with the origin at the location of the sensor. It is common to assume that the noise on measurements follows a Gaussian distribution with zero mean, some range variance and negligible angular uncertainty. We choose to express a line using polar parameters (r, α) as defined by the line equation (1) for the Cartesian coordinates (x, y) of the points lying on the line

$$x \cos \alpha + y \sin \alpha = r, \tag{1}$$

where $-\pi < \alpha \leq \pi$ is the angle between the x-axis and the shortest connection between the origin and the line. This connectionâĂŹs length is $r \geq 0$ (see Figure 6). The goal of line fitting in polar coordinates is to minimize

$$S = \sum_i^n d_i^2 = \sum_i^n \left(\rho_i \cos(\theta_i - \alpha) - r\right)^2 \tag{2}$$

for the n data points in the set. The solution of this least squares problem gives the line parameters:

$$\alpha = \frac{1}{2}\text{arctan2}\left(\frac{\sum_i^n \rho_i^2 \sin 2\theta_i - \frac{2}{n}\sum_i^n \sum_j^n \rho_i \rho_j \cos\theta_i \sin\theta_j}{\sum_i^n \rho_i^2 \cos 2\theta_i - \frac{1}{n}\sum_i^n \sum_j^n \rho_i \rho_j \cos(\theta_i + \theta_j)}\right) + \frac{\pi}{2}, \qquad r = \frac{1}{n}\sum_i^n \rho_i \cos(\theta_i - \alpha) \tag{3}$$

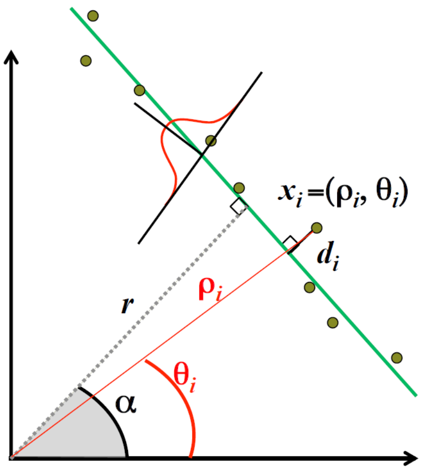

Figure 6: In polar coordinates, a line fitted to data (θ_i, ρ_i) can be uniquely defined by (α, r). We make the assumption that there is Gaussian noise on the range measurement (ρ_i) but none in the angle (θ_i).

Line Extraction

There are many algorithms that have been successfully used to perform line extraction (e.g. Split-and-Merge, Line-Regression, RANSAC, Hough-Transform, etc.). Here, we will focus on the "Split-and-Merge" algorithm, which is arguably the fastest, albeit not as robust to outliers as other algorithms. See Algorithm 1 below and Section 4.7.2.1 in the textbook [2] for more details.

Algorithm 1: Split-and-Merge

Data: Set S consisting of all N points, a distance threshold $d > 0$
Result: L, a list of sets of points each resembling a line
$L \leftarrow (S)$, $i \leftarrow 1$;
while $i \leq len(L)$ **do**
  fit a line (r, α) to the set L_i;
  detect the point $P \in L_i$ with the maximum distance D to the line (r, α);
  if $D < d$ **then**
    $i \leftarrow i + 1$
  else
    split L_i at P into S_1 and S_2;
    $L_i \leftarrow S_1$; $L_{i+1} \leftarrow S_2$;
  end
end
Merge collinear sets in L;

The scripts ExtractLines.py and PlotFunctions.py are provided to structure and visualize the line extraction algorithm. You will be modifying/adding functions in ExtractLines.py to perform the Split-and-Merge line extraction.

There are three data files provided, rangeData_<x_r>_<y_r>_<n_{pts}>.csv, each containing range data from different locations in the room and of different angular resolutions, where <x_r> is the x-position of the robot (in meters), <y_r> is the y-position, and <n_{pts}> is the number of measurements in the 360° scan. The provided function ImportRangeData(filename) extracts x_r, y_r, theta, and rho from the csv file. Figure 7 illustrates these three data sets.

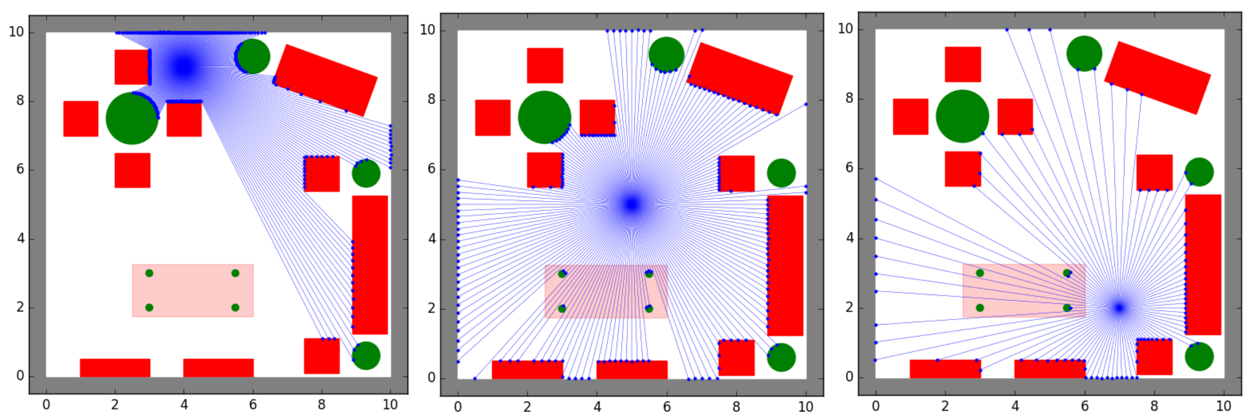

Figure 7: Lidar range data for three different locations in the room and three different resolutions, corresponding to rangeData_4_9_360.csv, rangeData_5_5_180.csv, and rangeData_7_2_90.csv, respectively

(i) For each of the three data sets, run ./ExtractLines.py to extract line segments from the data and plot them on the map. The main ExtractLines function has been provided for you. Your job is to populate the SplitLinesRecursive, FindSplit, FitLine, and MergeColinearNeigbors functions. More details can be found in the script comments.

There are four suggested parameters to control segmentation:

- `LINE_POINT_DIST_THRESHOLD`: The maximum distance a point can be from a line before the line is split
- `MIN_POINTS_PER_SEGMENT`: The minimum number of points per line segment
- `MIN_SEG_LENGTH`: The minimum length of a line segment
- `MAX_P2P_DIST`: The maximum distance between two adjacent points in a line segment

These parameters act as knobs you can tune to better fit lines for each set of range data. You are welcome to add other parameters/constraints as you see fit.

NOTE: There is not one correct answer to this problem. Slightly different implementations of the algorithm may produce different lines. However, *better* results will, of course, smoothly fit the actual contours of the objects in the room and minimize the number of false lines (e.g. that jump between objects). Also feel free to edit the `ExtractLines` function or any of the plotting in `PlotFunctions.py` if you'd like.

(ii) Submit three plots showing the extracted lines for each of the data sets; include your segmentation parameter choices with each plot.

IMPORTANT: Install Additional Software Dependencies

You will require a few additional software dependencies to complete the remaining problems. First, install TensorFlow (https://www.tensorflow.org/), an open-source library for dataflow programming that is very popular amongst machine learning researchers and roboticists:

```
$ sudo pip install tensorflow
```

You will also be making use of Jupyter notebooks (https://jupyter.org/) to interactively explore the results of your machine learning experiments:

```
$ sudo pip install --upgrade jupyter matplotlib
```

Note: To execute a cell, press [Shift]+[Enter]. Also look at the shortcuts options in the help menu. For those familiar with Matlab, running each cell is similar to running sections in your code. Restarting the kernel in Jupyter clears all assigned variable, similar to the `clc` command in Matlab.

Next, run the provided script to download the dataset we'll use for the next two problems:

```
$ sh download_datasets.sh
```

In case the script doesn't work, you can manually download the files from:
https://stanford.box.com/s/7uccz78ikgqnvckq0y2m3tb3z46j3mvf
The file `pedestrian_dataset.npz` should be placed in the `Problem_3` folder. Unzip `P4_cats_and_dogs.zip`, and place the resulting `datasets` folder in the `Problem_4` folder.

Problem 3: Tensorflow and HOG+SVM Pedestrian Detection

The field of computer vision includes processing complex and high dimensional data (up to millions of pixels per image) and extracting relevant features that can be used by other components in a robotic autonomy stack. Nowadays, many computer vision techniques rely on deep learning and machine learning algorithms for classification and depend heavily on computational tools that can efficiently process, learn, and do inference on the data.

In this problem you will familiarize yourself with TensorFlow as a tool for machine learning. You will learn the basics of TensorFlow by implementing a Support Vector Machine (SVM), a typical machine learning classification algorithm, on Histogram of Oriented Gradients (HOG) image descriptors to identify pedestrians in images. HOG is a technique for detecting and extracting edges of objects in an image. By detecting the edges, HOG produces features of an image which can be used for many machine learning algorithms.

Support Vector Machines (SVM) are supervised learning models for classification of data. In its simplest form, an SVM finds a decision boundary between two categories of data and classifies each datapoint $x^{(i)}$ based on which side of the boundary it lies on. In this problem we will assume that the decision boundary is a hyperplane $xw = b$ (a linear SVM)[2], and thus the prediction is of the form

$$\tilde{y}^{(i)} = sgn(x^{(i)}w - b)$$

Suppose the data we have is labeled with either +1 or -1 (pedestrian or not a pedestrian, cat or dog, tree or signpost, etc.). This can be visualized in Figure 8 where the black dots represent a label of +1 and the

[2] In Python-based machine learning computation frameworks, the convention is for datapoints to be row vectors so that a list of datapoints $[x^{(1)}, x^{(2)}, \ldots, x^{(n)}]$ corresponds to a matrix with the datapoints as rows. This is why in this problem we write xw instead of $w^T x$.

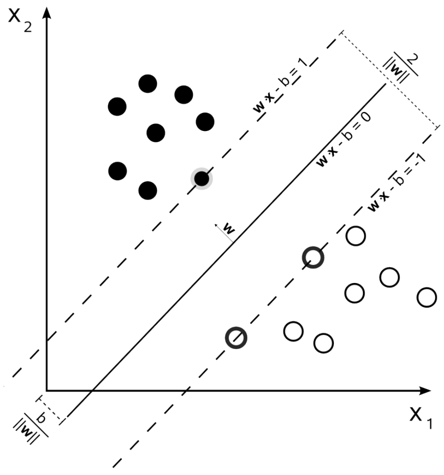

Figure 8: Illustration of a linearly separable dataset. Image from

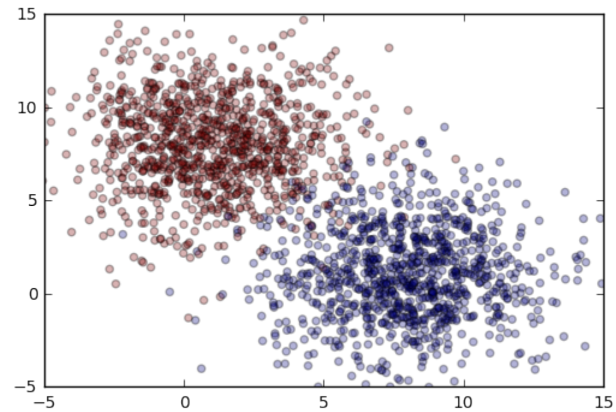

Figure 9: Toy dataset that is not linearly separable.

white dots represent a label of -1. For simplicity, we will first consider the case where the data is separable (no overlap) and can be separated by a straight line.

This means we can select two parallel hyperplanes that separate the two classes such that the distance between them is as large as possible. The region between the two hyperplanes is called the "margin" (dashed lines in Figure 8) and the hyperplane that lies halfway is called the maximum-margin hyperplane. Maximizing the region is equivalent to solving the following optimization problem:

$$\min_w \|w\|_2^2 \qquad \text{subject to} \qquad y^{(i)}(x^{(i)}w - b) \geq 1$$

Now suppose that the data is not separable (see Figure 9). Notice that there are some points that overlap in the data so the notion of maximum-margin hyperplane as optimal cannot be applied. In this case, optimality can be defined with respect to the following. First we introduce the hinge loss:

$$\ell_{hinge}(x, y) = \max(0, 1 - y(xw - b))$$

This loss function penalizes points for being inside the margin $(xw - b) \in [-1, 1]$ and especially penalizes points on the wrong side of the hyperplane, i.e., when $(xw - b)$ and y have opposite signs. Thus now not only do we want to maximize the margin, but also minimize the hinge loss.

Thus, given a dataset $(x^{(1)}, x^{(2)}, ..., x^{(n)})$ with correct labels $(y^{(1)}, y^{(2)}, ..., y^{(n)})$, the "soft-margin" linear SVM optimization problem is

$$\min_{w,b} \frac{1}{n} \sum_{i=1}^{n} \max(0, 1 - y^{(i)}(x^{(i)}w - b)) + \lambda \|w\|^2 \tag{4}$$

where λ is a hyperparameter which gives the relative weighting for the two terms. It is important to note that although Equation (4) describes a linear classifier, we can achieve nonlinear behavior by selecting feature vectors $x^{(i)}$ as arbitrary functions of the true input data. Examples include adding higher-degree monomials in addition to xy-position values for 2D data, learning a linear classifier on HOG features extracted from images instead of on the raw pixel values themselves, etc.

(i) ✎ Using the notation in Figure 8, prove that the perpendicular distance between the two planes is equal to $\frac{2}{\|w\|}$ (and thus minimizing $\|w\|$ is equivalent to maximizing the margin).

(ii) ✎ Explain in a few sentences why you think TensorFlow uses the workflow of setting up a computation graph of mathematical operations, then running these computations when fed with input data. How does this compare to numpy (where you get the results of computations instantly) and why does this paradigm make sense for machine learning, and possibly on robots?

(iii) Take a look at `svm_estimator.py`. Edit the function `model_fn`. In particular, during the training case, compute y_{est}, and the soft-margin SVM loss function, and for the prediction case, compute y_{est} and the label.

(iv) Run your Tensorflow estimator by running `python svm_estimator.py --type toy`. How does your SVM do? Explain why it makes the misclassifications the way that it does. Include the plot in your write up.

(v) In the previous part, identity features were used, i.e., x_1 and x_2 were the features used for the SVM. If instead we used the following features x_1, x_2, x_1^2, $x_2^2, x_1 x_2$, would this improve or reduce your classification accuracy? Explain your answer.

(vi) Fortunately for you(!), we have implemented HOG feature vector extraction [3] in TensorFlow for you.[3]

You can explore the dataset and the HOGs in `P3_visualization.ipynb`. Explore the dataset, and the associated HOG features. Notice that you need to run a TensorFlow session in order to get results out of the HOG tensor for an input image. Your task is to set up the datasets for training, evaluation and prediction in the `get_hog_data` function in `svm_estimator.py`. Make sure your labels correspond to the correct images. You do not need to worry about shuffling the data, this is automatically done within the Estimator workflow (see the `train_input_function`).

(vii) Run your SVM using HOG features for pedestrian classification, by typing `python svm_estimator.py --type hog`. What is the classification accuracy for this model? This should save your model in the `training_checkpoints/hog/model` folder. Visualize and explore your results in the notebook. Remember to specify your model number in the notebook! Notice that the HOG elements with positive weight correspond vaguely to the outline of a pedestrian (see Figure 6 in [3]).

[3]See `HOG.py`. Typically this step might be implemented in numpy as the HOG descriptors are used as inputs $x^{(i)}$ to the learning process, but aren't actually involved in the training process (i.e., extracting HOG feature vectors is similar to using custom features, rather than the identity feature which would be raw pixels for the case of images. But we wanted to show you that TensorFlow can be a general-purpose computation framework. And if you're really ambitious, you might even try putting in TensorFlow variables instead of the constant x/y-convolution kernels to make this a "deep" neural network!

Problem 4: Classification and Sliding Window Detection

Even with the vast reduction in model parameters achieved by convolutional neural networks (CNNs), compared to fully connected neural networks, training modern visual recognition models from scratch can still take weeks on immensely powerful computing hardware. But by leveraging the feature-extraction prowess of a pre-trained image classification CNN, in this case Google's Inception-v3 [4], even those of us with without a supercomputer[4] (and with a homework deadline!) can train a high quality image classifier on our own custom image data.[5]

Problem Setup: We will be using the open source TensorFlow library (`https://www.tensorflow.org/`) to perform the numerical computations involved in training and evaluating neural networks in this problem. After downloading `P4_cats_and_dogs.zip` from canvas.stanford.edu and unzipping it in the Problem_4 directory, the files for this problem should bea organized as:

- `datasets/` → labeled images from the PASCAL Visual Object Classes Challenge 2007 [6]
  - `datasets/train` → training images with labels for supervised classification learning
    - `datasets/train/cat` → pictures of cats!
    - `datasets/train/dog` → pictures of dogs!
    - `datasets/train/neg` → pictures of neither (mostly planes, trains, and automobiles)
  - `datasets/test` → test images with labels to evaluate the performance of our model
    - `datasets/test/cat` → pictures of cats!
    - `datasets/test/dog` → pictures of dogs!
    - `datasets/test/neg` → pictures of neither (mostly planes, trains, and automobiles)
  - `datasets/catswithdogs` → pictures with both! (for testing rudimentary detectors)
- `inception_graph.txt` → Inception-v3 graph specification (visualize with TensorBoard)[6]
- `retrain.py` → CNN classifier retraining script
- `utils.py` → TensorFlow computation graph input/output utilities, feel free to take a look!
- `classify.py` → image classification test script
- `detect.py` → object detection three ways, *the only file you must modify*

Image Classification

First, we concern ourselves with the task of *image classification*. That is, given an image belonging to one of a number of classes (here, "cat", "dog", or "neg"(ative) for neither) we would like to associate with each class a probability of the image's membership.

(i) (This part is heavily inspired by `https://www.tensorflow.org/how_tos/image_retraining/`.)

Here's the plan: we (a) download ~ 25 million pre-trained model parameters, (b) chop the pre-trained model off at the layer right before final classification, where it has produced concise vector summaries

[4] If your laptop/VM setup is particularly computationally limited, we have also provided the option to use one of Google's MobileNets [5] in the place of Inception-v3. This model has far fewer parameters (~ 1.3 million) and performs noticeably worse on the computer vision tasks in this problem, so we recommend sticking with the default choice, but to go this route uncomment line 1187 (MODEL_TYPE = 'mobilenet_0.50_224') in `retrain.py`.

[5] In this problem we'll be classifying cat and dog pictures; technically our model, pre-trained on ImageNet (`http://www.image-net.org/`) datasets and classes, is particularly well-suited to extracting features relevant to small animal classification. If this seems a bit cheat-y, feel free to try this problem with your own truly custom dataset, minding the tips under the "Creating a Set of Training Images" heading at `https://www.tensorflow.org/how_tos/image_retraining/`.

[6] If you're going the MobileNets route (see above), load this model with the "Upload" button on the GRAPHS tab.

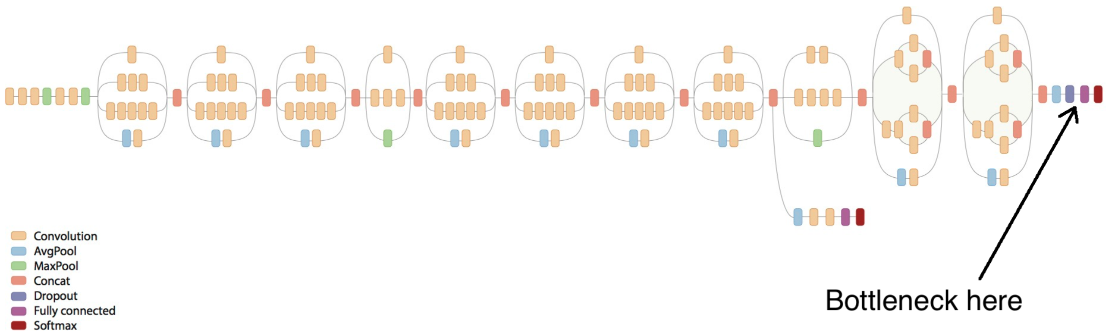

Figure 10: A visualization of the Inception-v3 CNN classifier ($\sim$ 25 million parameters) [4]. MobileNets [5] strive to achieve a similar level of accuracy with far fewer parameters.

of input images (the "bottleneck" layer, see Fig. 10), (c) implement a linear classifier[7] that takes these summaries as feature vectors and outputs a probability vector over our classes, and (d) train just this final classifier on our regular computer. Ready, set,

```
$ python retrain.py --image_dir datasets/train
```

After all of the bottleneck feature vectors have been computed/cached by the retraining script (this might take a few minutes as your computer has to compute a few billion feed-forward operations, i.e., the vast majority of layers depicted in Fig. 10, for each input image) we can visualize the progress of the training process by starting up the TensorBoard visualizer in another terminal window:

```
$ tensorboard --logdir=retrain_logs
```

and navigating to http://127.0.0.1:6006 in your browser. Look at the GRAPHS tab in TensorBoard[8] to see the model structure; our linear classifier lives in the `final_training_ops` subgraph.

What is the dimension of each "bottleneck" image summary?[9] How many parameters (weights + biases) are we optimizing in this retraining phase?

After training, we can evaluate the performance of our classifier on images it hasn't seen before.

```
$ python classify.py --test_image_dir datasets/test/
```

Pretty good, eh? Note the filenames of a few of the misclassified images; we'll revisit them in part (iv) of this problem.

Object Detection and Localization

Near-human-level image classification is pretty neat, but as roboticists it is often more useful for us to perform *object detection* within images (e.g., pedestrian detection from vehicle camera data, object recognition and localization for robotic arm pick-and-place tasks, etc.). Traditionally this means drawing and labeling a

[7]See http://cs231n.github.io/linear-classify/ for a good overview.

[8]If you're not running Inception-v3 (see footnotes on previous page), for the purposes of this question you may load `inception_graph.txt` in the GRAPHS tab — it's a bit easier to see what's happening there than in the MobileNets graph.

[9]The "?" in the first dimension of `input/BottleneckInputPlaceholder` indicates that this graph node can take an arbitrary number of bottleneck vectors as input (recall that row vectors correspond to data points). This question is asking for the dimension of each bottleneck vector.

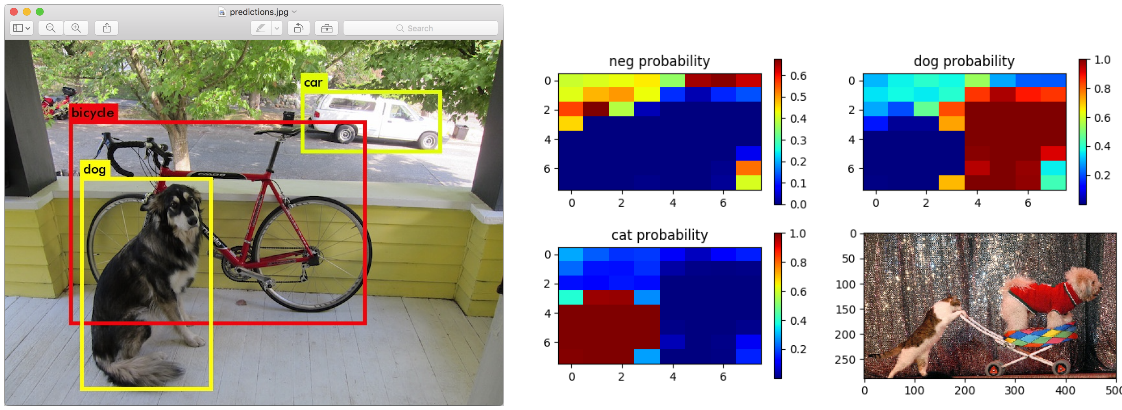

Figure 11: Object detection. On the left, YOLO [7]. On the right, us (sliding window classification).

bounding box around all instances of an object class in an image, but we'll settle for a heatmap today (see Figure 11). In practice, achieving state-of-the-art performance in object detection requires training dedicated models with clever architectures (see YOLO [7], SSD [8]), but in the spirit of bootstrapping pre-trained models we can convert our image classifier into an object detector by applying it on smaller sections ("windows") of the image.

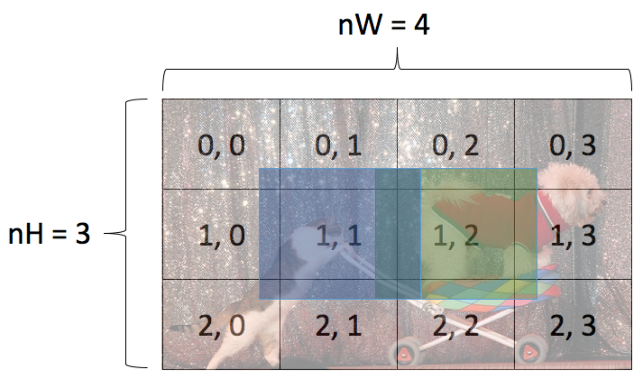

Figure 12: Sliding window with padding (part (ii)). Running a classifier on the blue window might yield an answer of "cat"; running the same classifier on the green window we might expect "dog."

(ii) In `detect.py` complete the `compute_brute_force_classification` function. The arguments `nH` and `nW` indicate how many segments to consider along the height and width of the image, respectively. Evaluating the classifier on the blue window in Figure 12 will yield a probability vector that there is a cat vs. a dog vs. neither at window $(1, 1)$. Pad your windows by some amount of your choosing so that the impacts of convolutional edge effects are reduced. Run the detector with the command:

```
$ python detect.py --scheme brute --image <image_path>
```

iii) In addition to filling out `compute_brute_force_classification`, include the detection plot for your favorite image in `datasets/catswithdogs/`.

(iv) Messing with indices and computing sliding windows is not only a lot of work for you, but computing *on* them is a lot of work for your computer! There's a slicker way. In the convolution/pooling process associated with running the classifier on the image as a whole, the final image features are *already* being computed for image sub-regions. That is, instead of running the classification model `nH · nW` times, we can run it just once and achieve comparable results. There's nothing to implement beyond some computation graph I/O (see the function `compute_convolutional_KxK_classification`[10]); run this detector with the command:

```
$ python detect.py --scheme conv --image <image_path>
```

Include in your writeup the detection plot for your favorite image in `datasets/catswithdogs/`.

(v) In the TensorBoard GRAPHS tab, find the output of the final convolutional layer (`mixed_10/join:0`)[11]. What operation does it feed into? How is the feature vector for the image as a whole computed from the feature vectors for each image region?

Another simple approach to object localization (finding the relevant pixels in an image containing exactly one notable object) is *saliency mapping* [9]. The idea is that neural networks, complicated and many-layered though they may be, are structures designed for tractable numerical gradient computations. Usually these derivatives are used for training/optimizing model parameters through some form of gradient descent, but we can also use them to compute the derivative of class scores (the output of the CNN) with respect to the pixel values (the input of the CNN). Visualizing these gradients, in particular noting which ones are largest, can tell you for which pixels the smallest change will affect the largest change in class evaluation.

(vi) Read Section 3 of [9] and implement the computation of M_{ij} (described in Section 3.1) in the function `compute_and_plot_saliency`. The raw gradients w_{ijc} are provided as a starting point.

(vii) In addition to filling out `compute_and_plot_saliency`, include in your writeup the results of running the command:

```
$ python detect.py --scheme saliency --image <image_path>
```

on both a correctly and incorrectly classified image from `datasets/test/`. In particular, for the incorrectly classified image, you may be able to gain some insight into what the CNN is actually looking at when getting it wrong!

[10] The effective (nH, nW) are defined by how the model does its final pooling operation; for Inception-v3 it's (8, 8) and for the MobileNet it's (7, 7).

[11] If you're not running Inception-v3 (see footnotes on previous page), for the purposes of this question you may load `inception_graph.txt` in the GRAPHS tab.

Problem 5: Stop Sign Detection and FSM in ROS

IMPORTANT: We have made significant changes to the asl_turtlebot repo. Since the only changes you made in homework 1 was in controller.py, please copy this somewhere safe, delete the repo, and clone asl_turtlebot again.

1. In asl_turtlebot, copy your controller.py file to somewhere safe and not in asl_turtlebot.
2. Delete the asl_turtlebot folder
3. Reclone the asl_turtlebot folder

   ```
   $ git clone https://github.com/StanfordASL/asl_turtlebot.git
   ```

   Put controller.py back in the folder.

You will also want to rebuild your workspace because we have provided you with a new custom message type (DetectedObject.msg) that needs to be compiled.

```
$ cd ~/catkin_ws
$ catkin_make
```

Let's now get back to building our stack for the TurtleBot. In the context of ROS, a *stack* usually means a collection of nodes and other software that is used on a robot. In the problem, you will be asked to implement a part of your stack that will allow the TurtleBot to move towards a goal and *stop* if it sees a stop sign on the way there. This is similar to what you would expect an autonomous car to do on the road.

This is an example of how your perception algorithms (computer vision/object detection) would be used to inform your path planning and control modules.

Here is a description of the new code you have just pulled and also a bird eye's view of what this problem is about.

- pose_controller.py The controller node. This is similar to your homework 1 script but desired goal points are not hard-coded in.
- detector.py The image detector node. This node takes in camera images, camera information and laser scan information in order to estimate the location of the stop sign in the TurtleBot's reference frame. We have pretrained a Convolutional Neural Network (CNN) to classify Gazebo stop signs and calculate bounding boxes for them.[12] However because most of you will be running this code inside the virtual machine, the CNN detector is disabled by default and the node instead uses a simple color thresholding to detect stop signs (IMPORTANT: this second detection method will NOT work on the real robot or in any more complicated scenario, but we opted to also implement this in order to reduce the computational burden of running the simulation on your laptops inside virtualization - your real TurtleBots will be running the CNN though so poke at it!).
- supervisor.py The supervisor node. This will encode your Finite State Machine (FSM) for your robot. Your robot will be in different *states* and your state machine will encode how your robot may transition between these states.

[12] In general, when you are deploying robots in the real world, you will need to train a network on images relevant to your robot. Fortunately, there exists pretrained CNN classifiers (Inception, MobileNet) where you need to only train the last few layers for your specific dataset. In this case, we have collected a dataset of Gazebo stop signs and trained the network using those.

- turtlebot3_signs_sim.launch The launch file that will launch Gazebo (with the Turtlebot3 and a stop sign), detector.py and pose_controller.py. By default, this launch file does not start Gazebo's graphical interface (which can be enabled by passing gui:=true to the launch file) and instead starts a simplified 2D visualizer gazebo_plot.py. The detector should also start another window, which corresponds to the camera images being sent by the robot (it should look mostly gray). Feel free to have a poke around this launch file to get an idea of how launch files work. It is likely you may write your own as part of your final project.

We have equipped the TurtleBot with a virtual camera (in the shape of a sleek white box). Gazebo allows us to receive simulated camera images as well as messages with camera calibration information similar to what we have worked with in Problem 1. In the detector node, we can subscribe to these messages, and uses the raw images to detect and identify stop signs, and use the calibration information to compute the direction of the stop sign relative to the TurtleBot. We can combine this with depth information from the laser scanner to estimate how far the stop sign away, giving us the relative position of the stop sign from the robot, which we can use to have the robot act intelligently around stop signs.

In this problem, you will be drawing upon your knowledge from Problem 1 and also design your own state machine for the TurtleBot to obey traffic rules (stop at a stop sign before moving past it.) The goals of the problem are:

- Incorporate perception into your decision-making.
- Learn about and use Finite State Machines (FSM) to command your robot to execute non-trivial tasks.
- Gain more familiarity with ROS by working with multiple ROS nodes that publishes/subscribes to multiple topics.
- Learn about different ROS messages and how to use them.

(i) Take a look at supervisor.py. What information does it publish? What is the message type?

(ii) Edit pose_controller.py so that it subscribes to a topic of desired goal poses. More precisely, add one subscriber with the appropriate arguments. The required information for this is in the comments.

(iii) Fortunately, the simulated camera provides its own calibration so you do not have to implement Problem 1 again (we promised you would have to go through the pain only once!). But you want to extract the necessary information from this calibration in order to transform objects from pixel coordinates to a heading angle. This information is encoded in the CameraInfo ROS message. Read the documentation http://docs.ros.org/api/sensor_msgs/html/msg/CameraInfo.html to help you extract the focal lengths and principal points in the camera_info_callback function in detector.py. More information is given in the code. Pay close attention to the dimension of the field you may be accessing in the message.

(iv) Given the focal lengths and principal points, edit project_pixel_to_ray in detector.py to compute the ray/direction (as a unit vector) of the stop sign in the camera's frame of reference. You may find Figure 13 helpful (note that in our case u in the code would be $\tilde{x}$ in the diagram, cx in the code would be $\tilde{x}_0$ in the diagram and that we have different focal lengths for each axes of this camera).

You will now implement a FSM for your Turtlebot. A FSM (https://en.wikipedia.org/wiki/Finite-state_machine) is defined by a finite list of states, and conditions that describe how you can transition in and out of a state. You must also specify your initial state. This can be viewed as a mathematical model for your logic.

Initially, the Turtlebot should just be navigating to its desired pose (in POSE mode). If it sees a stop sign though, it must then stop for *3 seconds* (see STOP_TIME variable) when it is sufficiently close to it (approximately 0.5 meters - see `STOP_MIN_DIST` variable). Then it shall "cross the intersection" (i.e. move while ignoring any stop sign it might be seeing) and then eventually transition back into its regular pose seeking state (the mode denoted as `POSE`).

(vi) Draw out the state diagram for a FSM respecting the description above and that you think you could implement. Think carefully about what the transitions should be and think whether or not the Turtlebot will get "stuck" in an undesirable loop.

NOTE: In `supervisor.py`, currently NAV and POSE mode are essentially the same. However, in future homework, NAV represents the higher level goal pose, while POSE represents intermediate way-points that help acheive the final NAV goal. Feel free to change the current skeleton code to fit your FSM design.

(vii) Code up your FSM. Note that if you run the code before editing `supervisor.py`, the Turtlebot will just move towards $(x_g, y_g, \theta_g) = (1.5, -4, 0)$ without stopping at the stop sign (traffic violation!). To help you get started, we have given you a template of what a FSM could look like and a list of states that we think may be useful (see the `Mode` class). From this template, you would need to implement two new states (look at the `loop` method) and one callback (look at the `stop_sign_detected_callback` method) for it to accomplish the task. However, you are free to modify the code beyond this.

To test your FSM and see how your turtlebot performs type

```
$ roslaunch asl_turtlebot turtlebot3_signs_sim.launch
```

If you think your computer can handle the graphical interface (you don't need it though), add the `gui:=true` argument. This one command will run Gazebo, `pose_controller.py`, `detector.py` and `gazebo_plot.py`. You should also see a window that simulates what the camera is seeing, which should be nothing initially. This might take a few seconds to load.

Alternatively, you can run Rviz, a ROS message visualization tool that will become very handy later on. This is more lightweight that Gazebo as it is not running rendering a physics simulation.

Navigate to the `signs.rviz` file in the `rviz` folder. Type

```
$ rviz -d signs.rviz
```

You should be able to see two little axis frames (initially on top of each other), one labeled `odom` (the world frame given the initial starting pose), and the other, `base_footprint` (the TurtleBot's frame).

To test `supervisor.py`, in a new terminal window, run `supervisor.py` (remember how to run a node from homework 1?).

Now you need to publish the desired goal pose. In another terminal window, type

```
rostopic pub /nav_pose geometry_msgs/Pose2D "1.5" "-4.0" "0.0" -r 10
```

You may need to alter that command depending on how you implement your FSM.

If done correctly, your Turtlebot should start moving. The terminal window in which you are running the supervisor should also be printing mode changes.

Alternatively, you can click `2D Nav Goal` in Rviz and place a desired goal pose that way. `supervisor.py` subscribes to this topic (`/move_base_simple/goal`) and converts it into a desired pose command. You can use to test how well your pose controller and supervisor performs.

(viii) Take a screenshot of the path and velocity profile that should get drawn in one of the windows. We should be able to see the turtlebot navigate to its goal location, and most importantly see the 3 seconds of stopping in the velocity profile (corresponding to stopping at the stop sign).

$$\tilde{x} = f\,\frac{X_C}{Z_C} + \tilde{x}_0, \qquad \tilde{y} = f\,\frac{Y_C}{Z_C} + \tilde{y}_0,$$

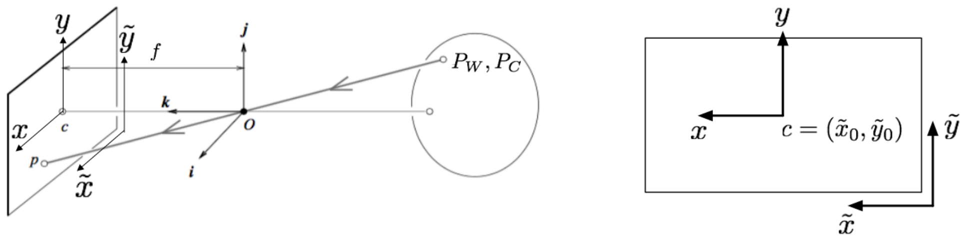

Figure 13: Projecting a point from pixels to camera frame

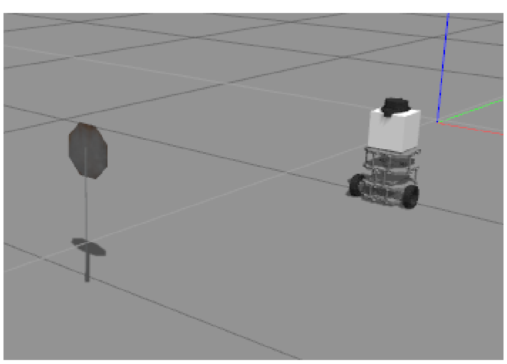

Figure 14: Turtlebot and stop sign in Gazebo.

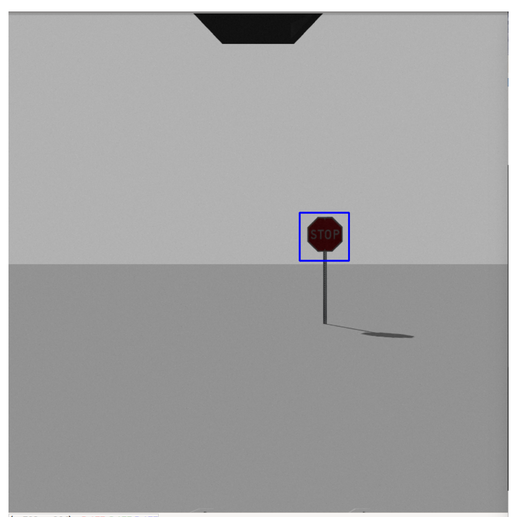

Figure 15: Camera's view of the stop sign.

References

[1] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000.

[2] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots*, 2nd ed. MIT Press, 2011.

[3] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 886–893.

[4] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception architecture for computer vision," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[5] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017, Available at https://arxiv.org/abs/1704.048614.

[6] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results," Available at http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 2016, pp. 779–788.

[8] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proc. European Conf. on Computer Vision*. Springer, 2016, pp. 21–37.

[9] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," 2013, Available at https://arxiv.org/abs/1312.6034.