Robot Learning

Mary Keenan, Cedric Kim, and Siena Okuno

1 Overview

The goal of this project was to teach Neatos to follow other Neatos using computer vision and machine learning. The leader would be teleoperated, with a follower using our machine learning algorithm as depicted in Figure 1.

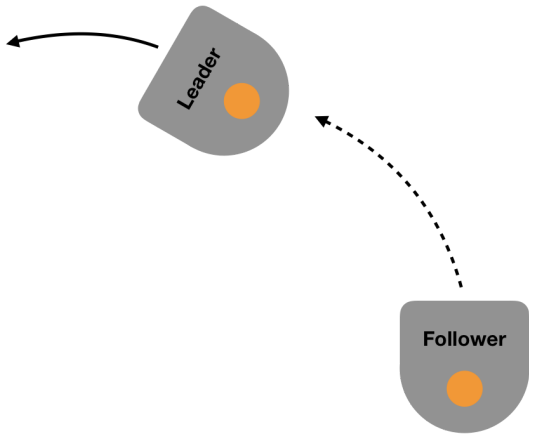

Figure 1: Neato Following

We collected data (a few thousand photo-to-motor command data points) of a Neato following another teleoperated Neato in an office and used that to train our models to take image inputs and output linear and angular motor commands. Not all of the data we collected showcased ideal behavior so used a script to filter out any “bad” data out. To achieve our goal, we investigated two algorithms: gradient boosting and convolutional neural networks. We were unable to fully implement gradient boosting, but the neural networks compared very favorably to our non-machine-learning baseline (88% to 50% accuracy in classifying images to motor commands in testing). Qualitatively, however, it performed poorly in real life.

2 Comparing Algorithms

We tried to meet our goal with two algorithms: gradient boosting and convolutional neural networks (CNN).

2.1 CNN

For the CNN, we utilized Keras’ Sequential Model and applied two convolutional layers to the input data. The training input data (image) was slightly compressed (because we were getting memory errors) but had no other processing. The training output data (linear/angular motor command tuple) was discretely categorized since there were only five possible values (forward, forward-right, forward-left, right, and left all at a fixed speed) and then hot-encoded before being passed into the model for training. Since the vast majority of the data points (about 5,000 of the 7,000) were the robot going straight forward, every data point with an angular

velocity was used twice, with the second version flipped left-to-right (along with the sign of the angular velocity) to introduce more diversity to the dataset.

The CNN model performed fairly well when applied to datasets it wasn't trained on. When trained on the second_collection and third_collection datasets and tested on the first_collection dataset, it had an accuracy of 88.03%. Compared to the baseline method, which had an accuracy of 47% on the first_collection dataset, this is very good. In real life, however, the CNN model qualitatively seemed to perform less well. With frequent encouragement from the lead Neato, the Neato running the code generally was able to follow the lead Neato around a room. Every now and then, however, it would find a pair of table legs or a wall it inexplicably preferred to follow and no matter what the lead Neato did, the Neato running the code continued towards its new fixation. So, the CNN model works well in theory but only passably in real life. Figure 2 shows a frame of the testing dataset and the model's accompanying saliency map. It looks like the model is paying attention to the Neato's shadow, it's white top, and unfortunately, the objects behind it. This attention to non-Neato objects may be why the model didn't perform well in real life. (click here to see a video of the Neato half-heartedly following a stationary Neato)

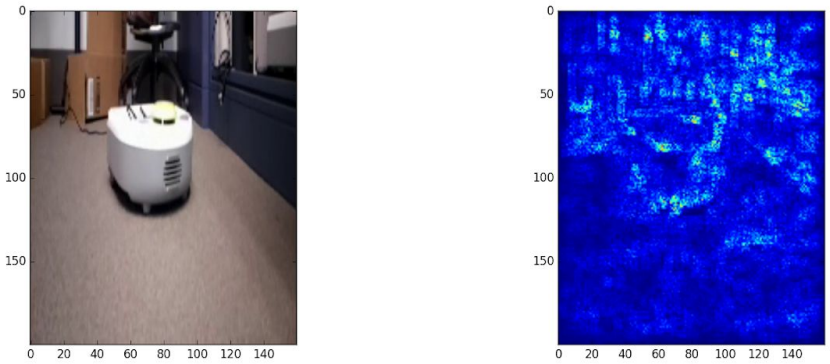

Figure 2: CNN Model Saliency Map for Image in Dataset

2.2 Gradient Boosting

Gradient boosting seemed like an attractive option for machine learning. It is known to be very accurate when given enough data and could be used for regression. Therefore, we used AdaBoostRegressor, which is a specific type of gradient boosting in scikit learn that is a slight adaptation on the classic AdaBoostClassifier. For this model, the predictor was a flattened vector of the image from the camera while the output was a tuple containing the linear and angular velocity.

There were two different iterations of the AdaBoostRegressor used for the project. The first was ran on Google Colab Notebooks. On this platform, our model was able to iterate through about 1500 image and velocity pairs when n=100. As a trial run, we first used the

bump-detection data to see if the model would be applicable. When k-folds cross validation was implemented, the ROC AUC score was around .96 for the trial run. Therefore, we decided to proceed with gradient boosting and moved the script from a Colab notebook to a normal .py file so it could be implemented on the Neatos with other code without a lot of trouble. However, this ended up being trouble as well. Because of the decrease in memory and CPU power, the final AdaBoost model could only be run with 500 data points when n=10. The test data was 1/10 of the size of the training set, but ROC AUC only works with classification so there was a problem with determining how accurate the model was. The R^2 value ended up as -.027, most likely due to the very small data sets. This means that in the end, our AdaBoostRegressor model was not successful in predicting the output velocities as in sklearn, negative R^2 values are possible if there is almost no correlation between input and output based on the model.

2.3 Baseline

We compared our two algorithms to a baseline. Since we were short on time, we decided to implement Connor Novak and Gretchen Rice's visual tracking code as a baseline. Their code finds contours from the camera image to determine the location of the neato. After finding the center location of the neato's contour, it is compared to the center of the image. If the location is in the left side of the camera image, the follower is told to turn left, and if the location is in the right, the follower is commanded to turn right.

Since our neural network training data uses command velocities from the tele-op mode, and their baseline code use continuous motor commands (PD control loop), we compared the output angular velocities to see if they were going in the same direction.

Running one of our data sets through the baseline code, the accuracy of the baseline follower came out to be 50% (355 F / 710). One reason for this low yield could have been due to lighting conditions, where the baseline follower was unable to follow the contours of the neato. Another could be from the delay in motor command velocities in reference to the output images from the neato's camera. When recording the data, motor commands would be delayed by a few frames because of the time it takes for the signal to get to the motors, time for the motors to spin up, etc. so this delay could have had an effect when the robots contour was near center in our field of vision.

3 Design Decisions

Various design decisions were made over the course of our project, two of which our outlined below.

3.1 Dataset Input

One of the biggest design decisions we made was the choice to use linear and angular velocity as the model output instead of the Neato's location in the inputted image. One consideration for using the Neato's location within the image frame was that we would have to create a script in order to track the Neato's location, or we would have to manually track the neato with our mouse in order to get the Neato's position. It would also introduce some noise

(bad data) that we would be unable to control and we would have to post-process. In addition, we would have to create our own control algorithm to move and turn the robot, which is fairly simple, but we wanted as few steps as possible.

One downside to using the motor command velocities is that our training data model would only have non-continuous inputs: left turn, right turn, forwards, and backwards with some extra combinations of those states. This would tell our algorithm that it does not matter how far away the neato is from the center of the image, the robot would just turn at a constant direction that it thinks it needs to go in. In addition, our training data would only be as good as the operator, which could include some bad data that we would have to filter out later. Regardless of the downsides, we decided to use the motor commands in order to skip the step of object tracking -- fewer things could go wrong if there were fewer steps.

3.2 Algorithms

Another decision we made was to try two algorithmic approaches to this problem. We had read that CNNs were a goto for processing visual data, which made the decision to implement one easy. However, we decided to also investigate a classic machine learning algorithm less commonly used for visual processing, AdaBoost, which is known for its accuracy and flexibility (as well as winning Kaggle competitions), but might have been finicky because it is typically used for classification problems, not regression. We were interested in seeing the difference between a classical machine learning algorithm and the more recently-emerging field of neural nets.

4 Challenges

4.1 Debugging

One challenge we faced was debugging performance issues. The CNN performed relatively well (75-88% accuracy depending on how the photos were processed) when tested on an unseen existing dataset, but qualitatively, it had less-than-ideal performance in real life, and it was a challenge to figure out why (we never did). There were two suspects: the training data and the model's implementation (or both). The data was all recorded in Paul's office on the same day, and we were live-testing in design studios, so it's possible the data was not representative enough of real-life environments. On the other hand, it's also possible that the CNN was just poorly implemented. To debug this issue, we tried to create a live-stream saliency map to help us understand what the model was seeing when the robot made the wrong (or right) decisions, but we were unable to get it implemented before the end of the project. Mystery unsolved.

4.2 Data Post Processing

Another challenge was taking all our data and filtering out any unwanted images and sensor data. We created a script in order to manually shuffle through our data. It may seem like a lot of work to do the data filtering manually, but a series of considerations were taken to make the process easier. Key strokes were used in order to increment frames one by one or by 10

frames. Spacebar was used to “omit” and image, and the comma and period key were used to do group omissions. Command Velocities were displayed to show the user if the movement of the neato corresponded to what movement we thought would be good to input into our training set. Visual indicators and a omission bar were both displayed to show which frame was omitted and how much of the data in the data set was bad. In addition, since garbled data (grey images from bad internet connect) was usually grouped together, it was simple to omit a big chunk of garbled data. Figure 3 shows our data processing GUI

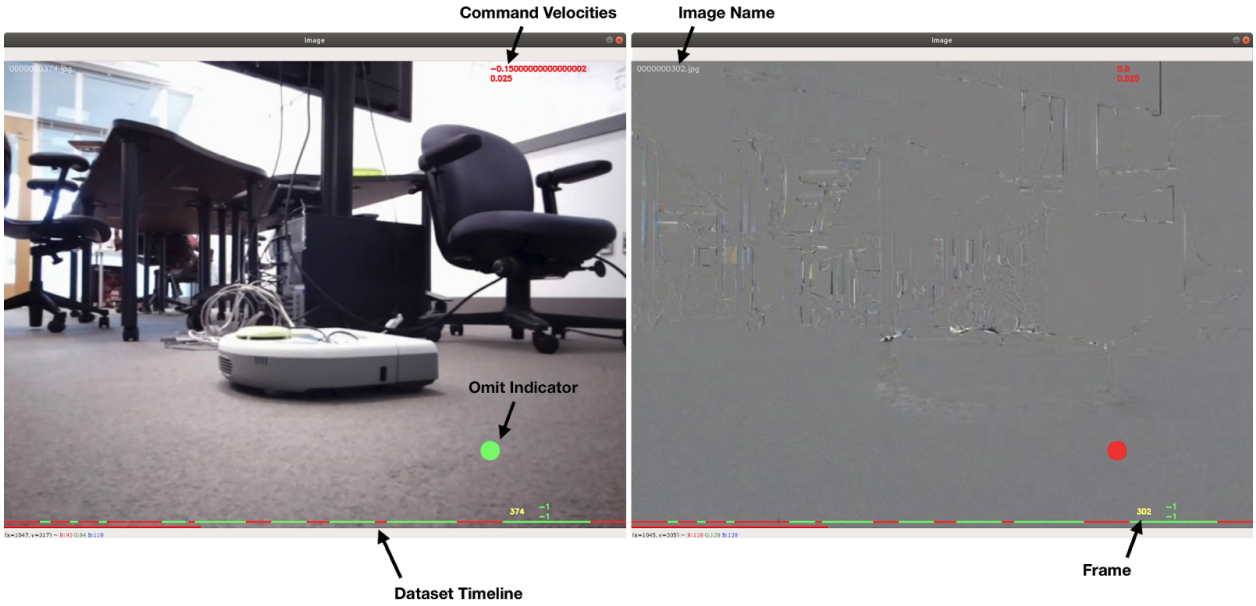

Figure 3: Data Processing GUI

5 Future Work

Given more time, there are three things we would try to improve.

Our data processor could have been more flushed out. The thought was to make a quick script in order to filter out some unwanted images, but the amount of bad data was more than we had expected, we ended up creating our own GUI in order to omit any bad frames, but in a rush, the code structure was not very good (non-object oriented script). In the future, we would use object oriented programming to create our GUI.

With more time, having a working AdaBoostRegressor model (with a positive R^2 value) would have been very good for comparison to the CNN model. This also could have helped with our hesitancy about the data being good or not as multiple models would have given some indication as to whether or not the data could be trusted.

One thing we could have done given more time is to investigate other types of datasets. Since our output from our machine learning algorithm was fixed speeds, we would use object tracking to create our data set and our algorithm could predict where it thought the Neato was in

the camera's field of vision. This way, we could use a controls algorithm that would be able to give variable motor commands so that the farther the robot is to the sides of our camera's field of vision, the faster the robot would try to turn to center it.

6 Lessons

Our final product was not what we had hoped for, but the process of trying and failing to get it had some good lessons:

- Make sure your training data is good data! Debugging the CNN performance issue was difficult in part because we didn't trust the training data we had collected. If we had been sure that we our training data was comprehensive and diverse, we could have ruled out a bad training dataset as the cause of the performance issue.
- Communicate with your team. We tried to split up the work as much as possible, but we did a poor job of communicating what work we had done that might be applicable to the other parts of the project. For example, the two models preprocessed data very differently, and having one uniform method of preprocessing would have streamlined the models' development and made it easier to troubleshoot problems.
- Use Colab notebooks as your computational powerhouse, pickle the model, and then switch to py files. Our laptops alone were having a lot of memory issues, but by using Colab we might have been able to get around this issue while still using py files for ROS.