Oranges?

Team 5

Final Report: *NEET Machine Learning Labs*

05/15/2019

Hector Castillo
Mitchell Guillaume
Claire Traweek
Daniel Wiest
Franklin Zhang

1. Overview and Motivations *(Authored by Claire Traweek)*

Machine learning is a hot topic in technology today. Recent developments in robotics use machine learning to implement camera-based object detection and navigation without the use of of other sensors such as LIDAR, thus lowering hardware costs and energy consumption. For the NEET Machine Learning Final Challenge, our team experimented with some basic machine learning techniques to autonomously drive our car in the stata basement. The first portion of our final challenge consisted of a gate detection lab, in which the team taught the car to drive towards gates using over 7 thousand images as data. The second portion of our final challenge consisted of the imitation learning lab, in which footage of the Stata Center basement was used to train the car to drive in a predefined loop using only a single camera. As a bonus to augment our understanding of the machine learning field, the team also experimented with reinforcement learning in a simulated environment.

1.1. Final Challenge Overview

During this lab, the team explored new topics in machine learning including gate detection, imitation learning and reinforcement learning. These abstract concepts were made somewhat more concrete through their implementation on the autonomous race car system.

1.1.1. Gate Detection Lab

In this lab, the team used a TensorFlow model we designed to recognize gates. After locating gates in an image with a heatmap, we created a basic controller that steers the car towards the gate. The end result was a robot that drove towards gates as they are recognized utilizing the three webcams mounted on the front of the car.

1.1.2. Imitation Learning Lab

In this lab, we collected data to train a PilotNet convolutional neural network (CNN) model, then performed a variety of augmentations to the data so that it was more representative of more general cases. We applied this strategy to a series of progressively more difficult scenarios.

1.1.2.1. First, we implemented imitation learning in the Udacity self-driving car simulation environment. Because environmental conditions remained consistent across runs, it was easy to test our model.

1.1.2.2. Then, we trained a model on the car, based off of data from all three cameras. The initial car path was simple--a short loop around part of the basement. The shortness of the loop made it possible to collect a large amount of data.

1.1.2.3. Finally, we implemented this scheme for the entire Stata basement. This required a large amount of data, and took a long time to train.

1.1.3. Reinforcement Learning Lab (Bonus)

As a final step, our team implemented reinforcement learning in a simulated environment, experimenting with three different algorithms, Proximal Policy Optimization 2 (PPO2), Soft Actor-Critic (SAC), and Deep Deterministic Policy Gradients (DDPG). This lab was mostly experimentation-focused, so all of our work was done in the *Donkey Car Simulator*.

1.2. Motivations

Machine learning plays a key role in characterizing and controlling complicated systems whose behaviors cannot be succinctly summarized with traditional analytical programming methods. Machine learning is particularly useful in cases where systems cannot be exactly modelled. Systems with inherent noise--like the race car and its tire skid--are perfect examples of these sort of challenging systems. Trained models can be trained to respond more quickly and accurately than human controllers in a variety of situations. Additionally, the creation of a robust model can allow for dynamic adaptation to situations that were not previously encountered. Because of this, machine learning promises to play a key role in both autonomous vehicles and the complex autonomous tasks of the future.

1.3. Goals

We aimed to fulfill the given requirements of the labs while also creating robust systems that can effectively generalize to conditions that are less than ideal.

1.3.1. Gate Detection Lab

On the most basic level, our robot needed to drive in the direction of the gate. We decided to extend this goal, and aimed to robustly detect gates from a wide variety of angles, then provide the steering angle necessary to drive through them.

1.3.2. Imitation Learning Lab

For this lab, we wanted to be able to drive a car autonomously in simulation, in a small loop in the stata basement, and then complete a large loop around the entire stata basement using an imitation learning network.

1.3.3. Reinforcement Learning Lab (Bonus)

Our aim in this lab was to drive the Donkey Car in the simulator using reinforcement learning. This lab was primarily learning-focused, so it was important that we understood how the changes we made affected the output. We also aimed to create our own VAE (Variational Autoencoder).

2. Proposed Approach

As was dictated by the final challenge descriptions, both the imitation and gate detection labs were implemented using the highly optimized machine learning tools available in TensorFlow. Because of the computational cost associated with training the enormous number of parameters on the large datasets our team created, specialized hardware was necessary. Our team was provided with a new car containing a Nvidia AGX Xavier onboard computer and three cameras. The specifications for this robotics platform are listed in Table 1. Additionally, an image of the new car is shown in Figure 1.

Table 1

Specification	Nvidia AGX Xavier
CPU Cores	8 Cores
Video Memory Quantity (VRAM)	16 GB (Shared with RAM)
Random Access Memory Quantity (RAM)	16 GB (Shared with VRAM)

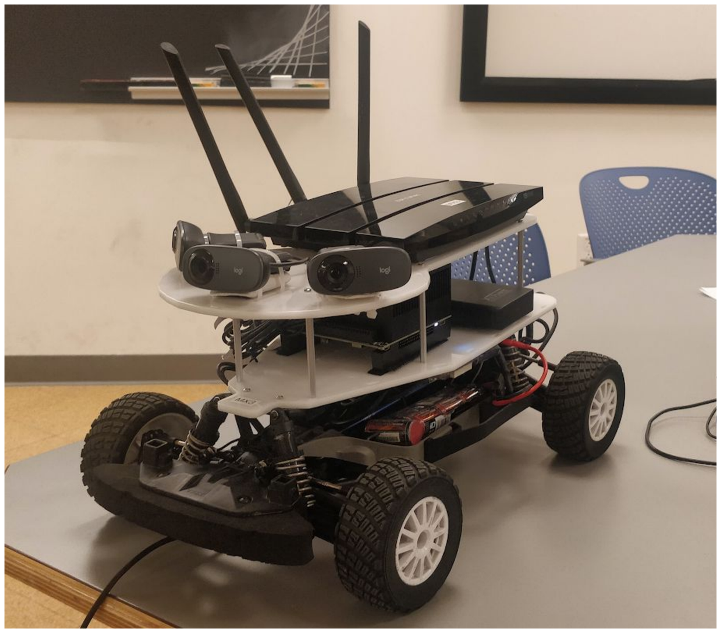


Figure 1: The new race car Model-X (MX) is pictured. It contains a Nvidia AGX Xavier onboard computer with the specifications listed in Table 1. Of primary importance is the fact that the MX contains only 3 webcams as sensors on its platform.

In addition to the new onboard computer and race car platform, a high performance network server was provided as a compute-capable machine on which our team's more complex models could be quickly and easily trained. The specifications for the network server are shown in Table 2.

Table 2

Specification	NEET Remote Server
CPU Cores	40 Cores
Video Memory Quantity (VRAM)	48 GB Nvidia RTX8000
Random Access Memory Quantity (RAM)	200 GB

2.1. Gate Detection Lab *(Authored by Daniel Wiest)*

At a high level, the goal of navigating through gates within the robot's field of view was accomplished in two steps. The first step was to utilize a convolutional neural network to determine whether or not a gate was visible to any of the robot's cameras. If a gate was detected by the front camera, the last layer of the convolutional neural network is activated and is used to find the location of the gate as well as the steering angle necessary to reach it.

2.1.1. Technical Approach

A convolutional neural network consisting of four convolutional layers followed by a single dense layer with two softmax output neurons was constructed to accomplish the task of both classification and localization. The classifier was trained on a dataset consisting of 7274 labeled images of which 2907 contained gates and 4367 did not. Additionally, a test set of 781 images (10% of the total dataset) was withheld from training in order to validate the generality of the model being trained. Among these training images, 323 contained gates and 458 did not contain any gates. An example of a subset of 25 images is depicted in Figure 2. The dataset consisted of images with a variety of different brightnesses and magnitudes of motion blur. These imperfect features were critical to obtaining a model which generalized well and performed satisfactorily during evaluation on the racecar platform.

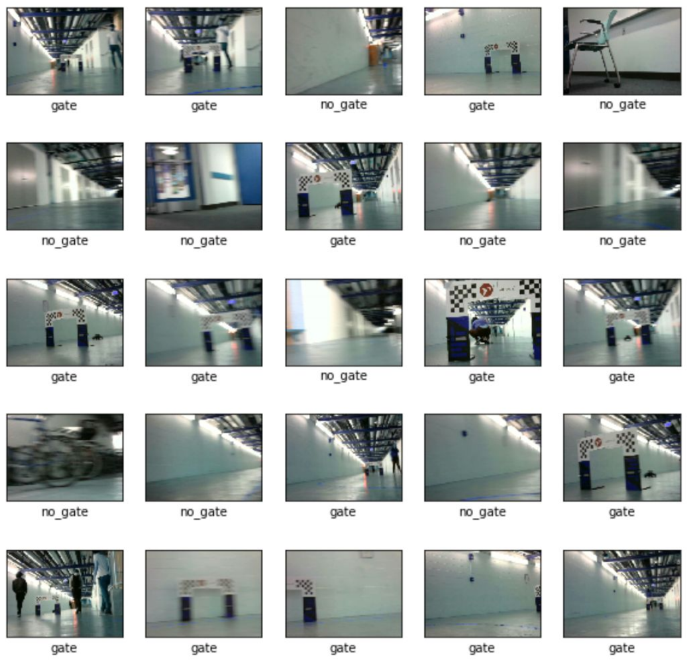
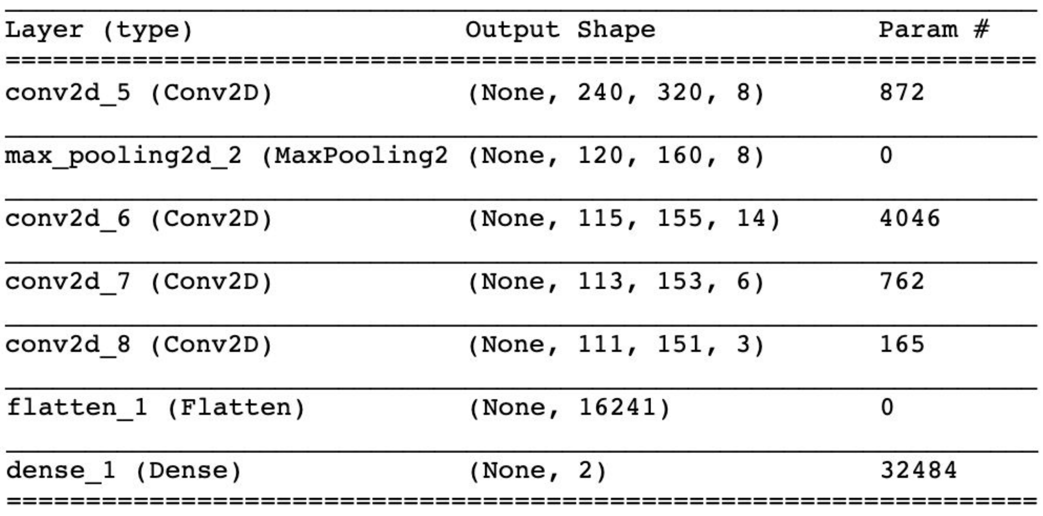


Figure 2: A subset of the dataset used for testing and training of the gate detection convolutional neural network is pictured. Of importance is the fact that many of the images are distorted or blurry. We found that including these poor quality data points greatly increased the overall robustness of our model.

Training was performed on the NEET Server with the specifications listed in Table 2. Our team was initially instructed that training directly on the car would be feasible. However, because of the limitations in available system memory, it was not possible to load the entire dataset into memory at full resolution. Although we could have utilized batch processing to make the neural network also train directly on the racecar, we found that utilizing the NEET Server was much faster and more predictable and therefore better addressed our needs.

The gate detection convolutional neural network consisted of four convolutional layers separated by max pooling layers followed by a single dense layer with two softmax output neurons. These two outputs corresponded to the two possible classes of output, with 0 being “no_gate” and 1 being “gate.” The structure for the convolutional neural network was based loosely off of the recommendations given in the paper *Is object localization for free? - Weakly-supervised learning with convolutional neural networks*, however the final structure was determined largely empirically. The exact structure can be seen in the TensorFlow model summary given in Figure 3. This structure was dictated partially by the accuracy of the model in binary classification, but was much more heavily dependent on the ability of the network to provide the spatial resolution and activations necessary for effective and accurate localization of the gates.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 240, 320, 8)       872
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 120, 160, 8)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 115, 155, 14)      4046
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 113, 153, 6)       762
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 111, 151, 3)       165
_________________________________________________________________
flatten_1 (Flatten)          (None, 16241)             0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 32484
=================================================================
```

Figure 3: The TensorFlow model summary for our team’s gate detection convolutional neural network is pictured. After training over the dataset for three epochs, the network achieved a classification accuracy of 98.56% on the test set. Most structural choices were made with the goal of extracting spatial localization from layer activation in mind.

One factor that had a large impact on the ability of our gate detection model to extract meaningful gate localization information from the images captured was the padding technique used for the convolutional layers. When the inputs to the convolutional layers were padded, this would result in regions of high activation at the borders of the image that greatly inhibited the network’s ability to accurately extract localization information. For this reason, all padding throughout the network was set to “valid” so that images were not altered in any way before being passed through the layers of the network.

The activations from the deepest convolutional layer in the network were used to produce the feature map (heatmap) for localization. This corresponds to averaging the activations across the 3 channels of the output resulting from the conv2d_8 layer defined in Figure 3. An example of the activations resulting from a given input are shown in Figure 4 with the corresponding image captured by the race car's camera. The heatmap has a resolution of 72 by 54 pixels, a reduction of around 4.5 times from the 320 by 240 pixel resolution image captured from the onboard cameras.

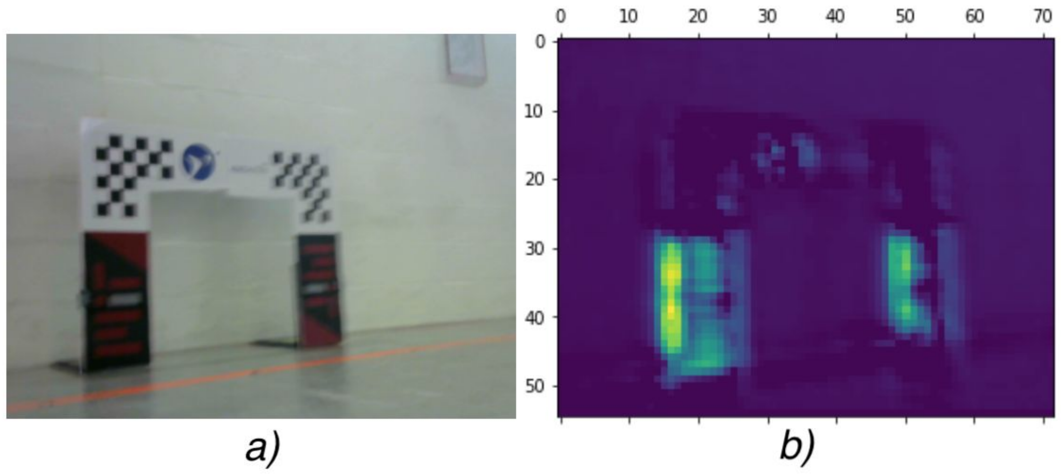


Figure 4: After inputting the image a), the average activation of the channels of the tensor resulting from the 4th layer of convolution in the gate detection neural network is pictured in b). Of particular interest is the fact that the network learned to recognize the gates by their pillars rather than the other features which are also present.

As seen in Figure 4, the network learned to recognize the pillars of the gates based on the features present and accordingly, these features have the largest activation at the final convolutional layer. For more evidence of this capability to detect pillars, see Figure 6. Because the goal of the challenge was to get the car to drive through the gate and not into one of the pillars, distinct design choices were made in order to process the resulting heatmaps and calculate the resulting steering command. The process of extracting the steering angle from the heatmap is comprised of a series of steps.

First, if a gate is detected in the image, the heatmap is cropped so that the ceiling is removed from the image and thresholded such that only the top 15% of pixel activations remain. A threshold value of 15% was chosen as it empirically seemed to provide a good balance between false positives and false negatives. Next, erosion is performed using OpenCV on the thresholded image with a kernel size of three such that isolated regions of high activation are eliminated. This step is taken because we know that a valid gate detection will likely consist of a large number of consecutive pixels rather than just a few pixels. The image is then dilated by three pixels and contours are identified using OpenCV's findContours function.

The number of contours are counted and if more than two are identified (corresponding to the left and right pillars of the gate), then the average of the centroids of the two largest regions is calculated as the pixel location of the center of the gate. If there is only one contour region detected, then this means that the gate is likely partially out of the frame or far away and the gate location is specified by the location of this single largest contour. This process is depicted graphically in Figure 5.

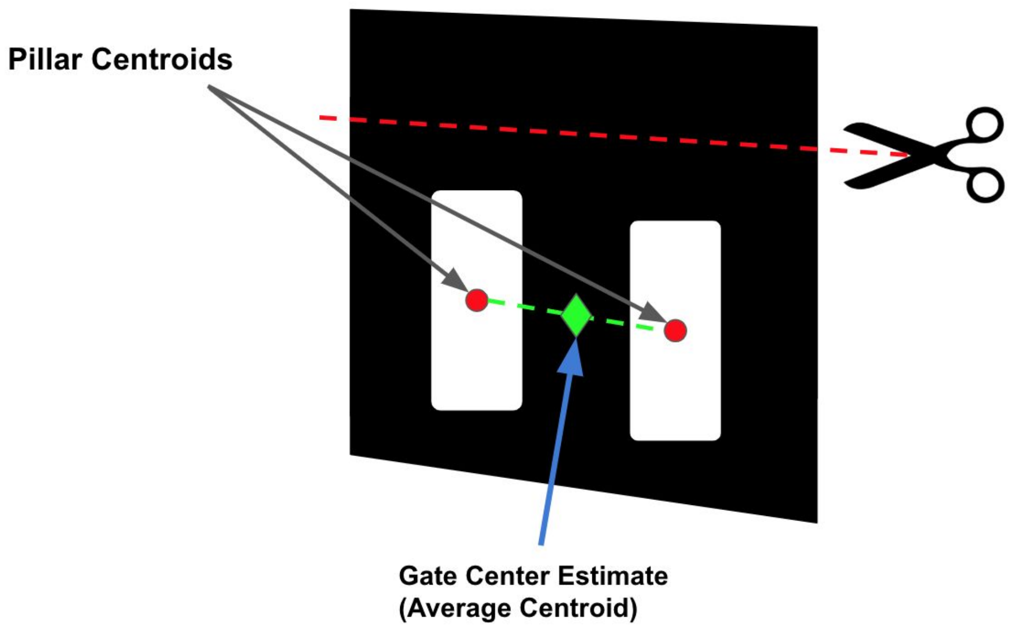


Figure 5: The process of extracting the gate center location estimate is shown for the two pillar case. The top region of the thresholded activation heatmap is also cropped out in order to prevent interference from the lights on the ceiling which often cause large activations in the convolutional neural network's final layer that can lead to erroneous gate localizations.

Once the gate location is calculated, it is mapped to a steering angle by projecting the width of the heatmap into the steering range of the car. This yields a steering range of -0.34 to 0.34 radians with 0.0 being a gate directly in the middle of the heatmap. A graphical depiction of the gate localization process is shown for 25 randomly selected images from the validation set in Figure 6.

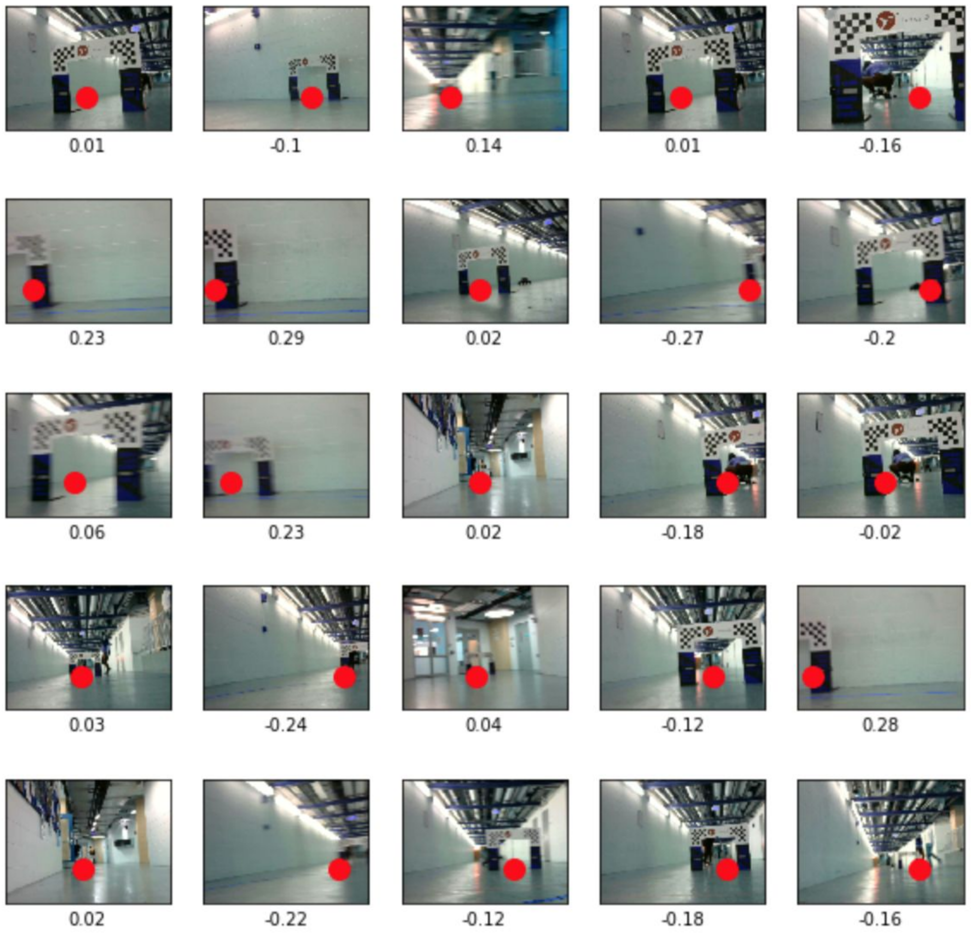


Figure 6: 25 images of gates are shown with the resulting steering angle recommendations shown as the label underneath and the estimated gate centroid shown with the red dot overlayed on top. These images depict the ability of the network to make reasonable guesses at the location of the gate in a variety of different situations.

In order to minimize the effect of high frequency noise in the steering angle calculations, a moving average is applied to the steering angle command, allowing for a much smoother traversal of the trajectory to the gate. A constant speed of 0.5 meters per second is used when moving towards the gates. If no gate is seen in the front camera, the left and right cameras are checked for gates. If a gate is present in a side camera, then a steering command is sent to turn the robot such that the gate will enter the field of view of the center camera. Figure 7 depicts the trajectory taken by a robot with the gate location beginning outside the view of the central camera.

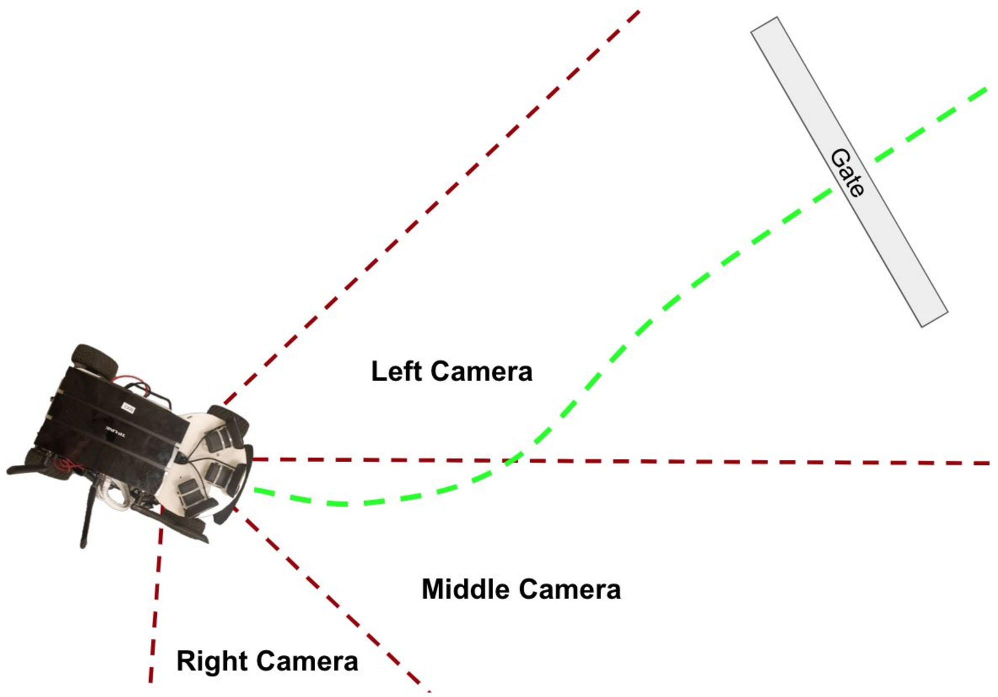


Figure 7: The trajectory taken by the racecar when the gate is located in the view of only the left camera is shown. In this example, the car turns left until the gate is visible in the central camera, permitting a wider range of gate detection tracking trajectories than would otherwise be possible using only the center camera.

2.1.2. ROS Implementation

TensorFlow operates only in Python3 whereas ROS operates only in Python2.7. For this reason, there was an extra degree of complexity added to the integration of the gate detection and localization algorithm with the existing ROS infrastructure. In order to overcome this incompatibility, ZeroMQ was used as a neutral message passing format to send the steering angle from the Python3 script which captured images from the webcams and evaluated them on the previously trained convolutional neural network to the Python2.7 script which used ROS to publish the steering commands to the control mux, allowing the race car to drive. This hierarchy is shown graphically in Figure 8.

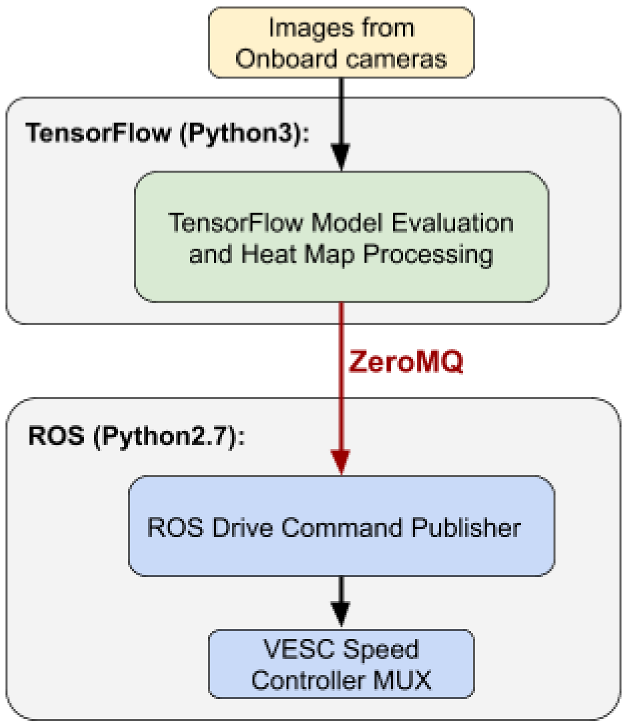

Images from Onboard cameras
TensorFlow (Python3):
TensorFlow Model Evaluation and Heat Map Processing
ZeroMQ
ROS (Python2.7):
ROS Drive Command Publisher
VESC Speed Controller MUX

Figure 8: A visual layout showing the flow of data between different subsystems of the gate detection lab is pictured. ZeroMQ is used to pass information between ROS and TensorFlow because of the differences in the versions of Python that each permits.

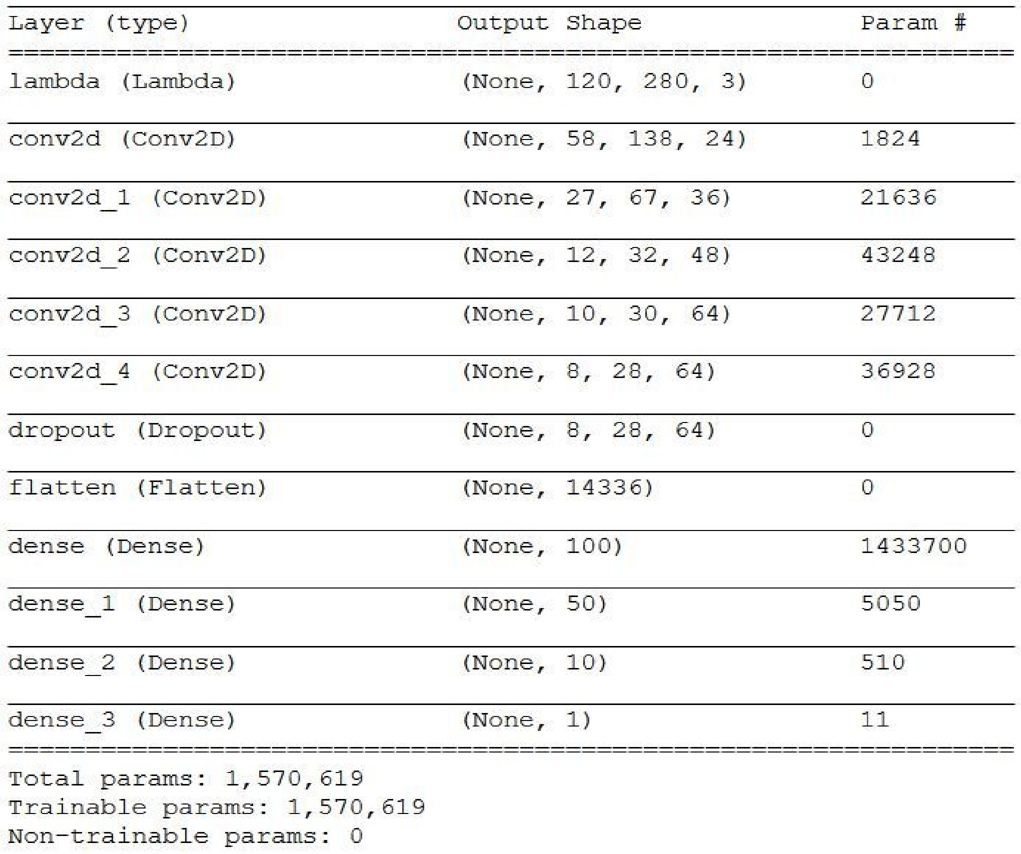
2.2. Imitation Lab *(Authored by Franklin Zhang)*

The imitation lab was implemented both in simulation and on the racecar platform. Training images that are labeled with a steering angle were collected and fed into a neural network. At a high level, the neural network takes in an image and returns a suggested steering angle.

2.2.1. Technical Approach

The imitation lab was implemented in steps. The model was implemented on a simulator, a small loop in stata, and then in the entire stata loop. By dividing the implementation into these steps we were able to learn along the way as we scaled up, building increasingly complex models.

To gain experience with imitation learning, we first implemented a simulated autonomous driving scheme using the Unity-based Udacity self-driving car simulator. This allowed the team to begin experimenting with imitation learning without the bottleneck of having to collect data on the physical car. We began by collecting data. We drove 28 laps around the simulator track, collecting a total of 131,508 images. We used these to generate a model, using batches to limit the number of images that were processed at once. The imitation learning model consisted of twelve layers, with an 320 by 240 normalized image from the webcam as the input and the steering angle as the output. The model includes four convolutional layers, a dropout layer, and several dense layers. This model is referred to as PilotNet and was developed by Nvidia for self-driving car applications. This topic is explored in further detail in the paper *End to End Learning for Self-Driving Cars*. A visual representation of the imitation learning model can be seen in Figure 9. To train the model, the team used a batch size of 20, 10 epochs, and 4000 steps per epoch.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda (Lambda)              (None, 120, 280, 3)       0
_________________________________________________________________
conv2d (Conv2D)              (None, 58, 138, 24)       1824
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 27, 67, 36)        21636
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 12, 32, 48)        43248
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 10, 30, 64)        27712
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 28, 64)         36928
_________________________________________________________________
dropout (Dropout)            (None, 8, 28, 64)         0
_________________________________________________________________
flatten (Flatten)            (None, 14336)             0
_________________________________________________________________
dense (Dense)                (None, 100)               1433700
_________________________________________________________________
dense_1 (Dense)              (None, 50)                5050
_________________________________________________________________
dense_2 (Dense)              (None, 10)                510
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 11
=================================================================
Total params: 1,570,619
Trainable params: 1,570,619
Non-trainable params: 0
```

Figure 9: This is an overview of the convolutional neural network that was used to train the imitation learning agent and the Udacity simulator. Its structure is known as PilotNet and was created by researchers at Nvidia for use in self driving car applications.

To get more value from the data recorded, we applied several augmentations to our images at random. These images offset, brightened, shrunk, or otherwise distorted our images so that the final model could react to errors in real life. This process added more variation to our training data, so that we could train a better model than we could with just the plain forward facing camera data alone.

Meanwhile, in the real-world racecar approach, we first implemented the imitation learning model on a smaller route to validate our process on a simple case. The car drove in a loop in a small corner of the stata basement as shown in Figure 10. Data was collected by running the car around this loop 7 times for a total of 6,830 labeled images. This data was used to train a imitation learning model on the neet server with a batch size of 20 and 1000 steps per epoch. The resulting model was able to drive itself around the loop, however the agent is not robust and has difficulty generalizing when it is placed in an offset starting position or noisy environment. The agent's difficulty in generalizing is further discussed in the evaluation section. Although this agent isn't robust, it was a good enough proof of concept to show that we could move on to test the model in the larger stata loop.

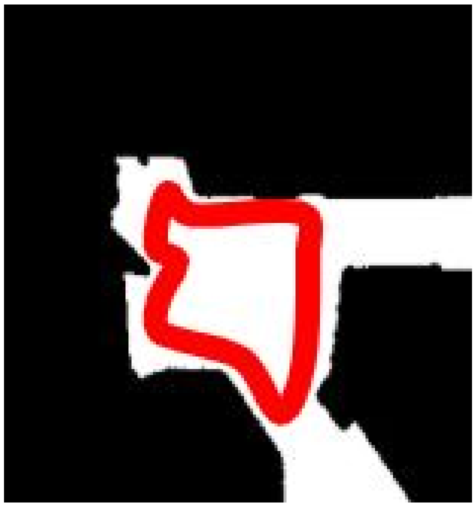


Figure 10: The imitation lab for the real world race car was done on a loop in this corner of the stata basement. The loop closely follows the walls and has right turns, left turns and straight paths. However, the majority of the loop in dominated by left turns.

The training data for the entire stata loop was done using 7 laps around the course, resulting in 14459 labeled images. In order to make our model able to generalize to more cases and to generate more data, the images were augmented by changing brightness, shifting, and distorting. The layers of the neural net consists of multiple convolution layers, densely connected layers and dropout layers like shown in Figure 9. The input is an image and the output is a single value corresponding the a predicted steering angle. The model was trained using 1000 steps per epoch, a batch size of 20 and 10 epochs.

The model was then put onto the car and used to generate steering angles so that the car can drive autonomously. The car was able to successfully complete a loop around stata completely autonomously. In order to evaluate the model, video from the camera and the steering angle output from the imitation learning agent was recorded.

2.2.2. ROS Implementation

The ROS implementation was more complicated than previous labs because TensorFlow uses Python3 and our ROS system uses Python 2.7. To bridge this gap, we used a ØMQ (Zero MQ) server.

Onboard the racecar, the overview of the autonomous driving process begins with the image taken from the camera that is feed into a Python3 file that uses a pre-trained model to provide steering angle commands. As mentioned earlier, ZMQ was required to send that command to a Python 2.7 file that publishes to a ROS topic. This system overview is shown in Figure 11.

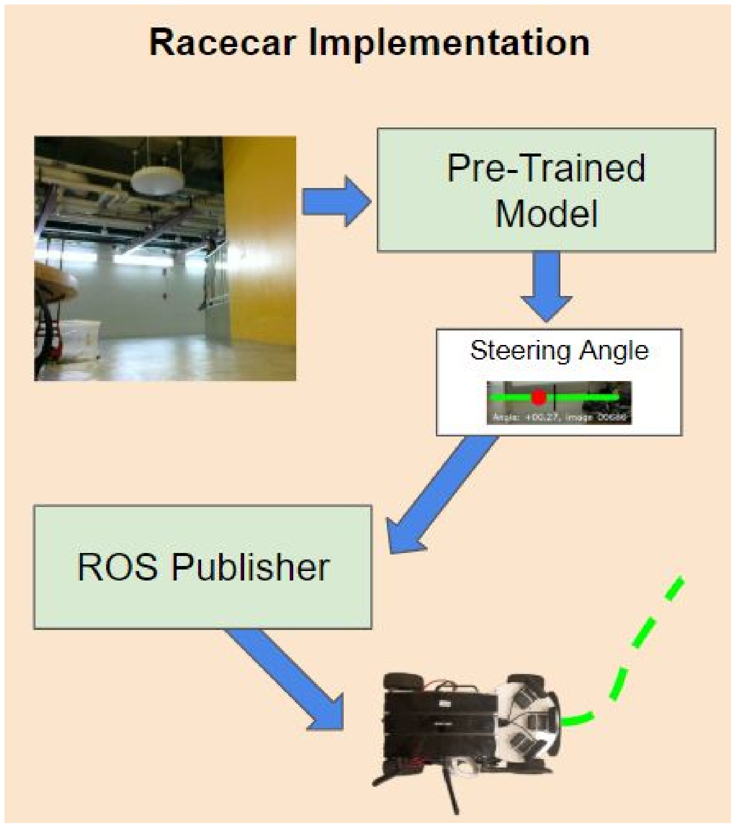


Figure 11: On the racecar, an image from the camera is feed into a pre-trained model that outputs a steering angle that is passed to a ROS publisher. The published message includes the steering angle that the robot uses to drive.

2.3. Reinforcement Learning Lab (Bonus)

This part of the lab focused primarily on a simulated Donkey Car in the DonkeySim program. The simulator provided a randomly generated road for the car to train on. We created and trained policies from three different algorithms: PPO2, SAC, and DDPG. Most of the code was provided for us, but it was important to understand the general structure so that we could swap out and evaluate algorithms. At the core of the system was TensorFlow and the Gym API. TensorFlow, along with Stable Baselines, a set of prewritten algorithms, trained the models, while Gym simplified the process of stepping through decisions and resetting upon failure.

Because the entire observable state was too large to reasonably process, we relied on a variational autoencoder, or VAE, which modified a limited number parameters to create a manifold of relatable scenarios. We used a pre-built VAE for the first part of the lab but later built our own based off of a few thousand images captured from the simulator.

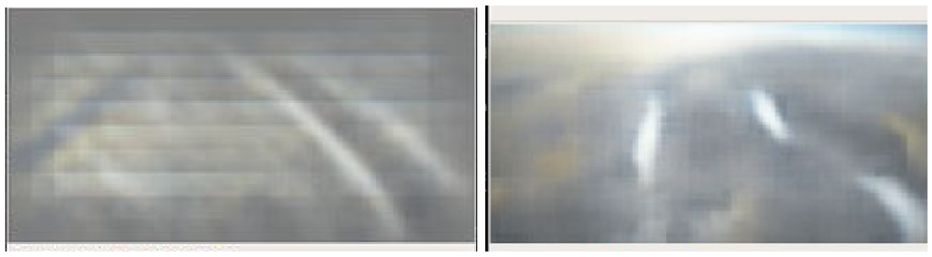


Figure 12: Our VAE, left, and the pre-generated VAE, right. Our VAE was generated from significantly fewer images and had 30 parameters, and was not as well-suited to the road simulation as the provided VAE.

3. Experimental Evaluation

The concept of evaluation on representative data is inherent to machine learning algorithms and is used to understand the optimality of the models during training. With this in mind, however, the abstracted nature of machine learning algorithms also makes it extremely difficult to dissect the internal logic responsible for producing the observed output of the system. This fundamental tension between the importance of evaluation and the abstraction of the team's systems was a significant hurdle in the team's ability to meaningfully evaluate the relative successes of the models in a quantitative manner.

Instead of being able to rely purely on the losses returned during training epoch, which measure the discrepancies between the ground truth labels and values predicted by the model, we found that these metrics were unrealistic and unrepresentative of the performance of the system when implemented on the physical race car. Even when evaluated on the validation data, the model's accuracy on these datasets was still not a tractable indicator of the model's success on the car. Because of the aforementioned difficulty associated with the precise quantitative evaluation of the algorithms, much of the team's evaluation was based on empirical metrics off which performance on the racecar could be more concretely evaluated.

3.1. Evaluation of Gate Detection *(Authored by Daniel Wiest)*

The process of gate detection necessitates a number of sub-processes, each of which contributes consecutively to the ability of the system to deconstruct an image, extract the gate location, then calculate the steering angles necessary to reach the gate. In order to address this sequential reliance, the evaluation of each sub-process was performed independently of the previous processes. The orthogonal nature of this evaluation method was beneficial in that it enabled us to easily explore a simplified parameter space, but also neglected the interdependencies of each step in their contribution to the overall effectiveness of the system.

The first sub-process which was evaluated was the convolutional neural network's (CNN) accuracy in binary classification. Our final CNN structure obtained an accuracy of 98.56% on the validation set consisting of 781 images that the network was not trained on. The binary classification sub-process, however, turned out to be one of the most variationally complex in the system because extremely high gate/no-gate prediction accuracy could be achieved with a variety of different network structures. During training, performance was preliminarily evaluated using the accuracy and loss metric on the dataset, shown in Figure 13. Viewing these metrics in isolation is not entirely representative of the desired results because they do not accurately reflect the influence of overfitting on the accuracy of the model on the validation set. In order to prevent overfitting, the model was continuously evaluated against the validation set. Training was stopped early after 5 epochs because this proved to produce the most generalizable model for images outside of the training set.

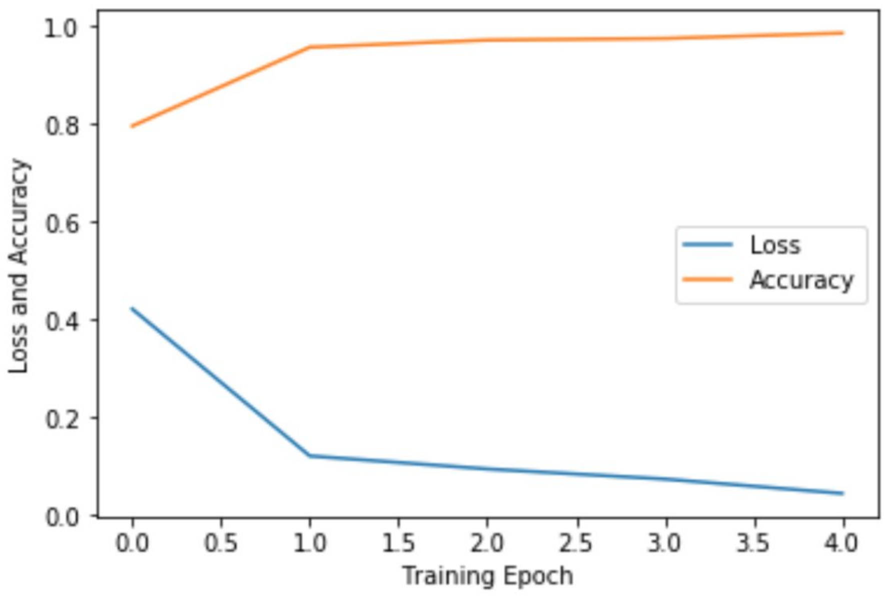


Figure 13: Training accuracy and loss are plotted versus epoch. Early stopping after 5 epochs was determined to provide a generalizable network which performed as accurately on the validation set as it did on the training set.

In the end, because of the ability of many different networks to the primary evaluation metric for this stage was the quality of the heatmap that the structure produced. Figure 14 shows 16 high quality activation heatmaps produced by the final network. This structure was selected because it provides a large amount of contrast between the pillars of the gates and the surrounding background.

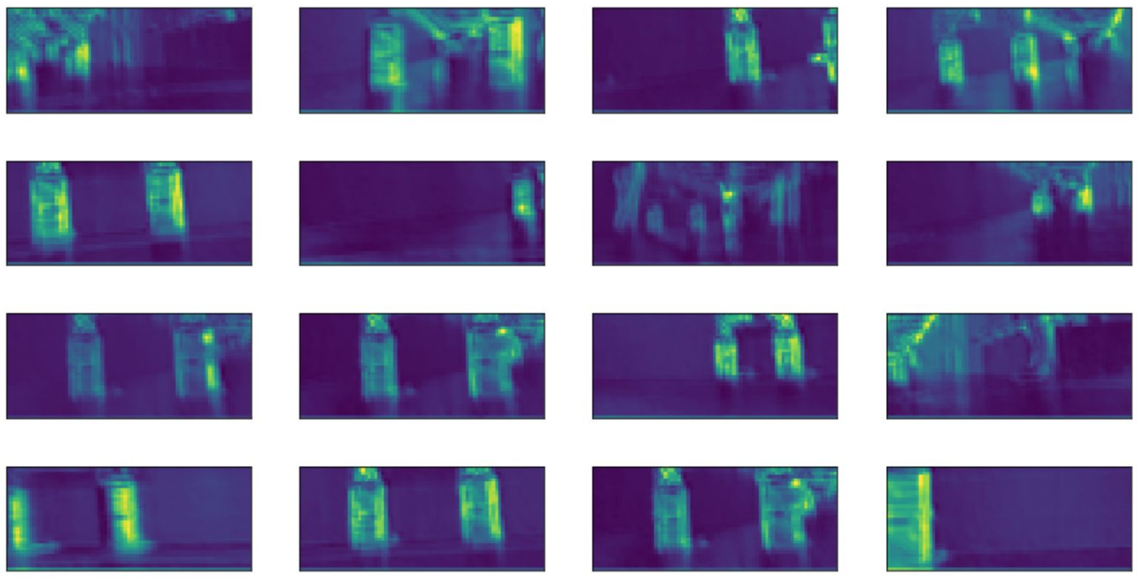


Figure 14: The activation heatmaps of the fourth layer of the convolutional neural network structure chosen for gate localization is pictured. This structure produces well defined outlines of each pillar, permitting localization to occur successfully in subsequent steps.

The second sub-process evaluated was the gate centroid extraction mechanism. Once again, because ground truth datasets were not available or feasible to construct, the evaluation of this sub-process was based on the quality of the estimates it returned on a series of 25 images of gates. The overall quality was assessed by eye by comparing the fraction of permissible centroid locations to those which were not accurate. An example of 25 evaluation images can be seen in Figure 6. Our finalized network attained approximately 85% success on locating a permissible centroid location given an random image of the gate.

Once the previously addressed sub-processes were evaluated, the integrated system needed to be evaluated on the robot. In preliminary tests, we found that while the car uniformly drove at the gate, it would command noncontinuous steering angles which caused unnecessary jittering. In order to combat this, an moving average was applied to the steering command, providing much smoother trajectories to be constructed to the gate. In addition, once the car got close enough to the gate, it would lose sight of one pillar then turn towards the pillar it could still see and run into it. This was a difficult issue to address because it arose from fundamentally correct behavior that under other circumstances would yield the correct results. In the end, through empirical evaluation, it was determined that the performance of the vehicle met the requirements of the lab the results were satisfactory. Videos of the car's performance can be found in the video section below this report on the team website.

3.2. Evaluation of Imitation Lab *(Authored by Franklin Z. and Mitchell G.)*

For our imitation implementation to be robust, the inference model needs to predict the turns accurately. We can evaluate the model both through quantitative methods and qualitative ones.

3.2.1. Udacity Simulator

Two different driving styles were used to collect training data for the Udacity simulator. In the first style, only the arrow keys on the keyboard were used to steer the car. This resulted in a much jerkier path, with long straightaways followed by short, sharp turns with a high steering angle. In the second style, a computer mouse was used to set the steering angle. Though it was harder to drive the car in this manner, it allowed for a much smoother path with much lower steering angles over a longer time. This was a good fit for the long sweeping turns of the simulated track. The resulting steering angle distributions are very different, with the mouse-only control having a much greater variety of values than the keyboard allowed. The two distributions of steering angle can be seen below in Figures 15 and 16.

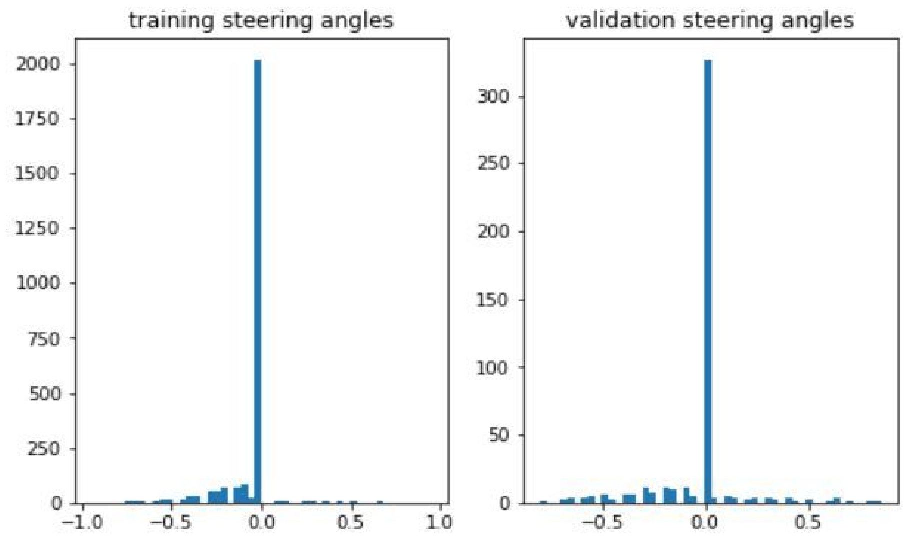


Figure 15: Steering angle and velocity distributions for training data for keyboard-only control. Note the discrete values of the steering angles.

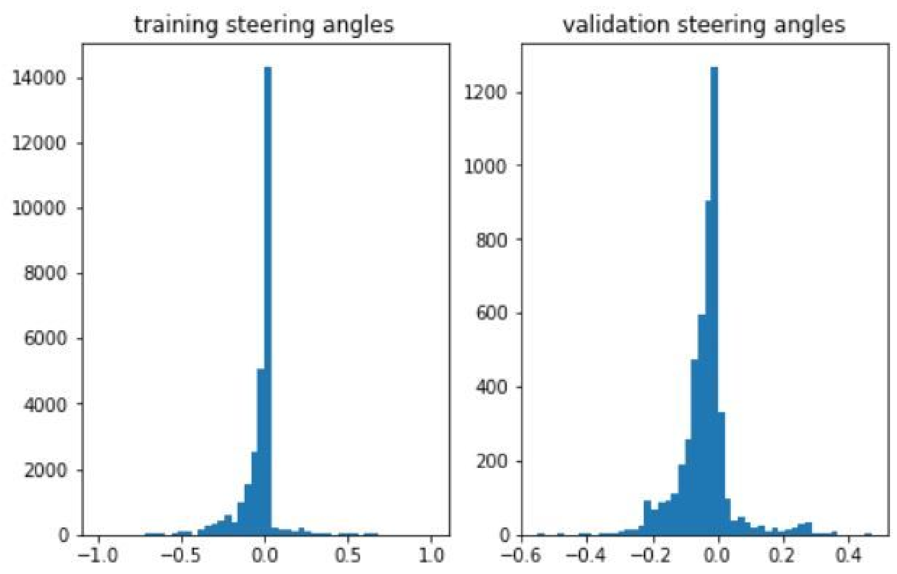


Figure 16: Steering angle and velocity distributions for training data for mouse-only control. Note the much broader range of values of the steering angle.

To determine which driving style should be used to generate our training data, four trial laps of training data were created for each method. Using this training data, 10 epochs were executed, and the loss and value loss of each model were used to evaluate which type of training data yielded a better model. For all epochs except the first, the model constructed with the mouse-only, continuous steering angle control had both lower loss and lower value loss. The losses for both methods of generating training data can be seen in Figure 17. Because the mouse-only control yielded much better models, all subsequent training data was generated that way.

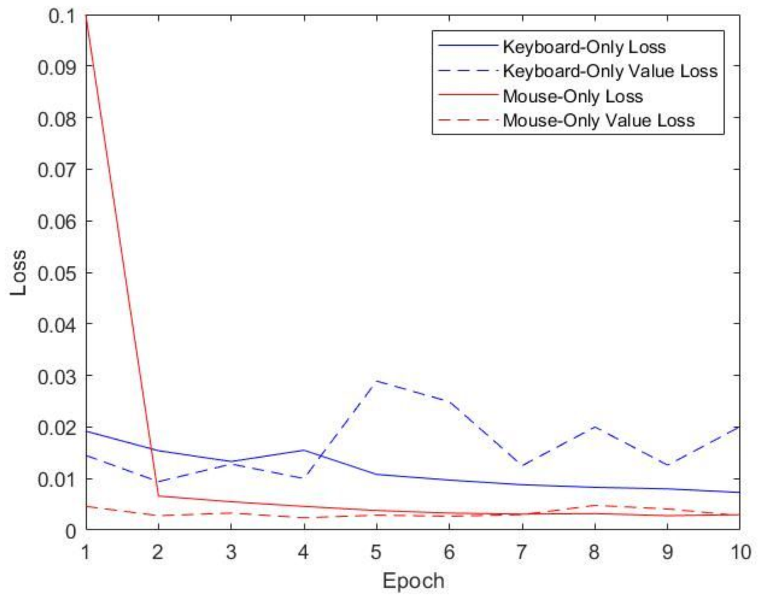


Figure 17: Losses and value losses for the keyboard and mouse control training data. The graph shows that for all cases (except the first epoch) the mouse controlled training data yields a better model.

To evaluate the performance of the imitation learning model, the model was tested against a new lap driven by one of the team members. In Figure 18, one can see the human driver's steering commands in orange, and the model's predicted steering commands in blue. The test shown in Figure 18 resulted in a root mean square (RMS) error of 4.03, or 6.2% normalized RMS error.

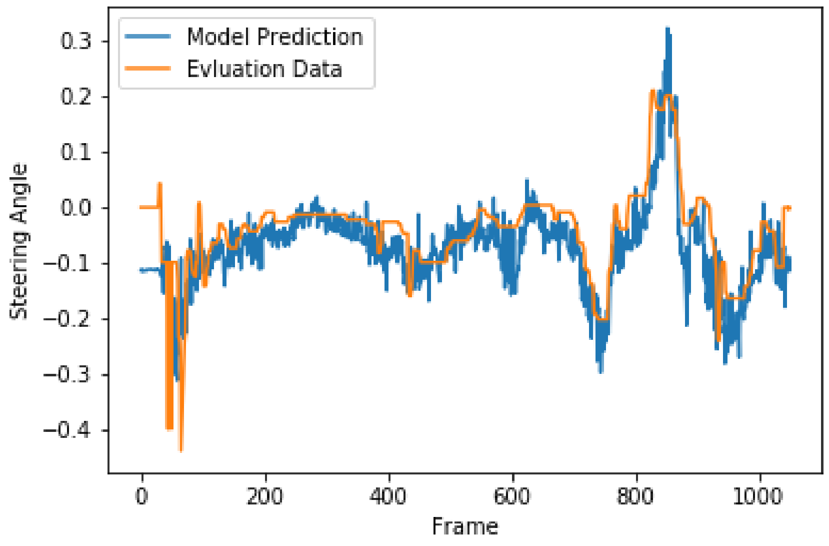


Figure 18: Model prediction and training data for steering angle from a lap of frames from the Udacity simulator. The orange line is the recorded input (human driver) and the blue line is the model's prediction. In this test, the model had a normalized RMS error of 6.2%.

Though the team was never able to create a model that was robust enough to complete an entire lap of the test track, the model was consistently able to make it to around the first corner of the track, and proceed until the bridge. The bridge was a unique feature of the test track, and it probably does not match the rest of the training data very well. As such, if the team wanted to make the model more robust, additional data should be collected for the bridge portion of the track.

3.2.2. Racecar Stata Loop

To evaluate our model quantitatively, we will compare the steering angles of a manual drive around the loop with the steering angles output from our model. We can then calculate the root mean squared error of all of the angle differences. It is very difficult to have a total error of zero because the distributions of the steering angle output from the model and the steering angles that are manually output are very different as shown in Figure 19. We can set a baseline error value to be the error that results from running this evaluation on the training set data. By comparing this baseline error value to the error of an evaluation of another dataset, we can see how well our model is able to generalize.

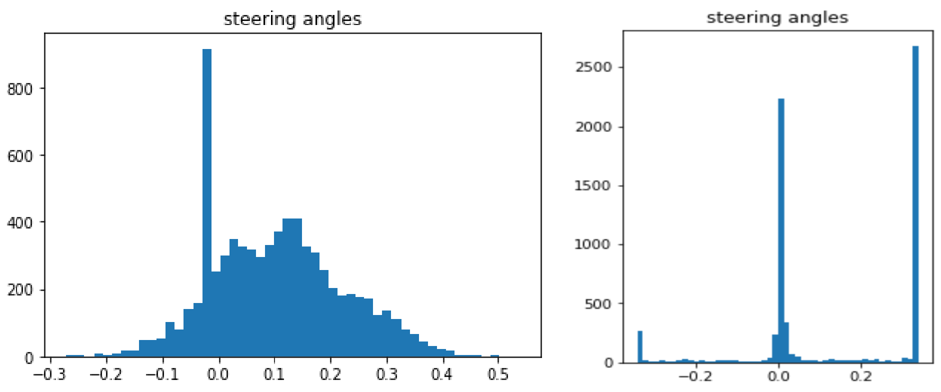


Figure 19: The left shows the steering angle inference made by the model. The right shows the steering commands inputs from the controller.

A qualitative approach to evaluation would be to see a video stream from the camera and then see the output of the inference model. The inference model should output a large steering angle when approaching a wall or making a turn. Figure 20 shows the inference model commanding a turn as the race car approaches a wall.

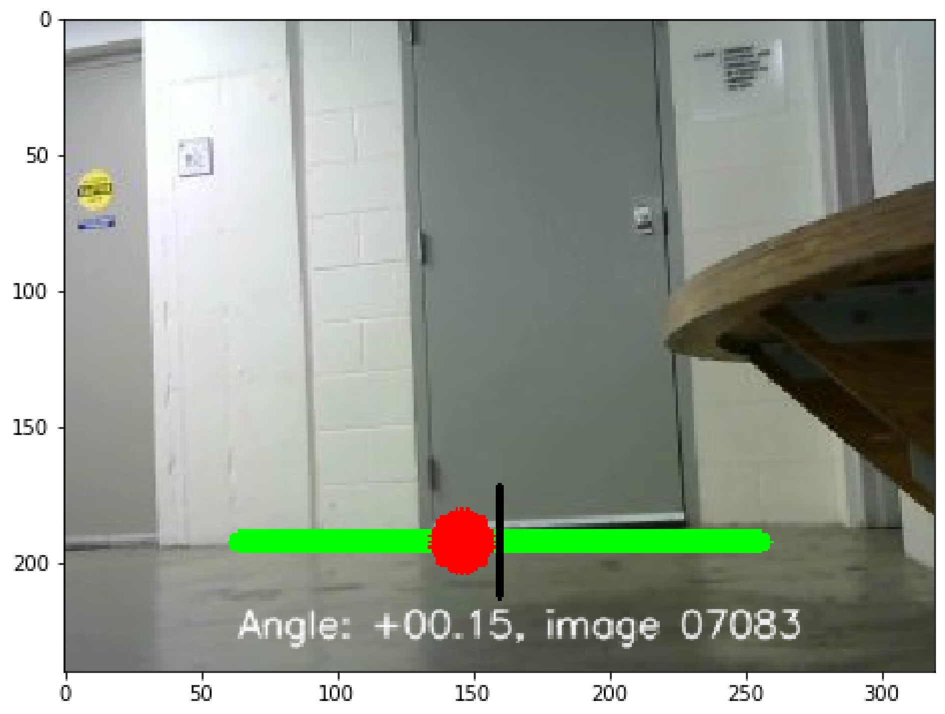


Figure 20: The trained model is commanding a steering angle as the race car approaches a wall. The red dot is located at on a 1D line graph where the center black line refers to a steering angle of 0. Falling on the left refers to a positive steering angle that points to the left while falling on the right refers to a negative steering angle that points right.

In order to more rigorously evaluate the model, the root mean squared difference between the steering angle from the model and the manual input was calculated. This evaluation method was chosen because we want to compare a model's ability to imitate manual driving. We graphed the steering angles from the controller and the imitation learning agent in Figure 21. The results of this image shows that the model trained on the small corner loop fits the training data well but doesn't generalize to other datasets as well. This explains why the agent's performance was observed to be sensitive even to the slightest changes in lighting, starting position and vehicle vibrations.

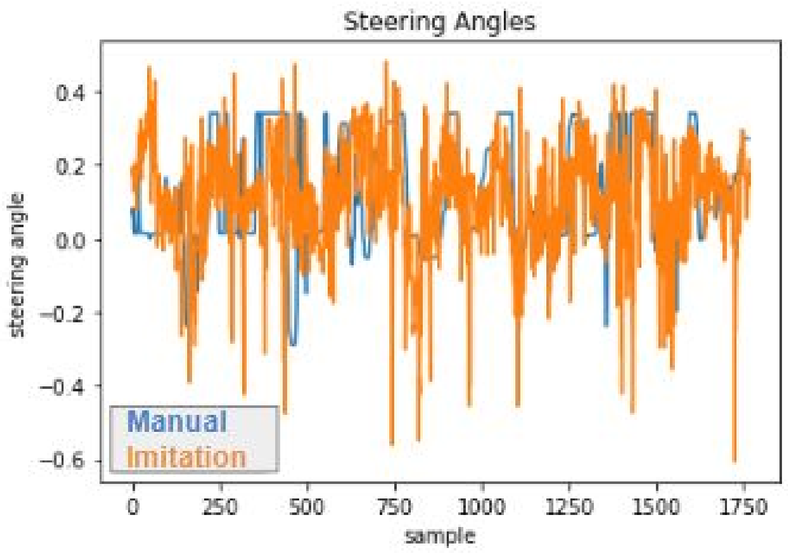


Figure 21: The steering angle from the remote controller in blue and from the imitation learning agent in orange for a series of sample images from a test run was graphed. The The root mean squared (RMS) error calculated with the testing data of for the small loop in stata was 0.19. The RMS error calculated from the training data was 0.07 and from a control with a constant output from the agent was 0.19. This graph and the RMS error shows that the agent doesn't do a very good job generalizing to images from non-training images.

The model trained on the entire stata loop is able to generalize better to new datasets. Figures 22 to 23 show how the agent is able behave similarly to manual control in both training and testing datasets. The imitation learning model isn't able to capture the large step discontinuities that are present in the steering angles of the manual control and instead opts to fall on more moderate values. Comparing both figures we can note similar patterns. Both figures have four relative minimums and one relative maximums laid out in the same order. These relative extremes in the steering angle map to the four right turns and one left turns in the driving route. Because the model is able to capture these extremas, it shows that the model is able to accurately predict when to turn. More quantitatively, the RMS errors of the training dataset, testing dataset, and control are 0.022, 0.026, and 0.047 respectively. This shows that the agent does imitate manual driving well because the RMS error of the control is much higher than the other two and that the agent can generalize to non-training datasets.

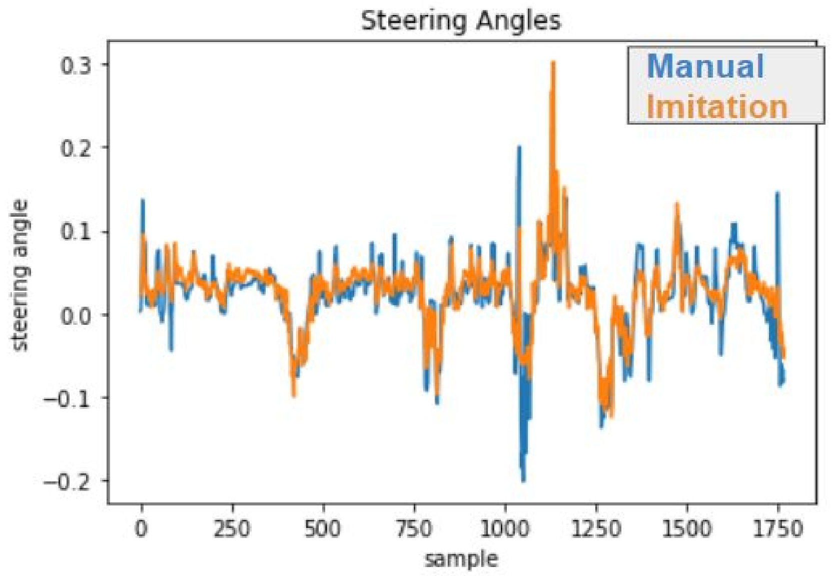


Figure 22: The graph shows the plots of the manual steering angle output and the imitation learning agent output for the training dataset. This data was the same data that was used to train the imitation learning model. The agent is observed to track the manual input quite well. The RMS error calculated on the training data for the entire stata loop was 0.022. This will be used as a baseline for comparisons.

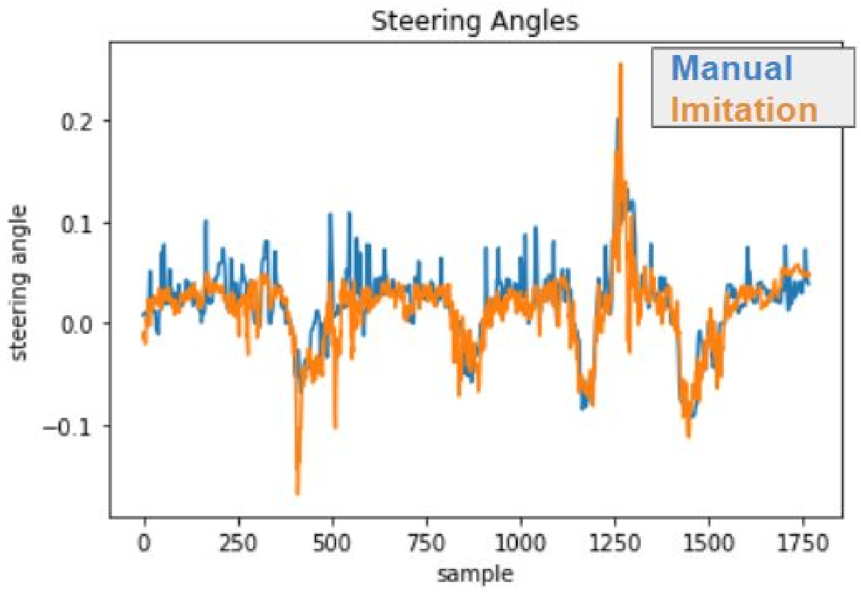


Figure 23: This graph shows the plots of the manual steering angle output and the imitation learning agent output for the validation set. This data was not used to train the imitation learning model. Once again, the agent is able to track the manual input quite well. The RMS error calculated on the validation data for the entire stata basement loop was 0.026. This is higher than the .022 value from the testing dataset, but still significantly smaller than the control value of .047, which was calculated by assuming a constant output form the imitation learning agent.

While these results are promising, there were some issues that we encountered. Although we were able to get the car to autonomously drive around the entire stata basement loop without crashing or human intervention, not every test was successful. There was one turn where the car steers too early and runs into a wall and another turn that the car sometimes complete ignores. These problems arise depending on slight changes in the physical environment, including things like humans, lighting changes, environment changes, and different starting positions. When more data was collected to continue to train the model, the results of the new model turned out to perform worse than the original model, resulting with the car requiring much more human intervention to complete the route. This might be because of the change in environment with time. The next day, the environment may be different enough that the model isn't able to make reasonable inferences from the image data.

Despite these issues, the evaluations of the imitation learning lab on the physical race car show quantitatively and qualitatively that the agent is able to imitate manual driving around the entire stata loop while generalizing to work on non-training dataset. Further improvements are required to make this more robust.

3.3. Evaluation of the Reinforcement Learning Lab (Bonus)

(Authored by Claire Traweek)

The only algorithm that, after only about 5000 iterations, created a policy that completed a significant portion of the track was SAC (Soft-Actor Critic). Even after 2,000 iterations, PPO2 (Proximal Policy Optimization) only consistently reached about 300 steps before failing. DDPG (Deep Deterministic Policy Gradient) routinely failed before 100 steps, even after 1,000 iterations of training. It is likely that further tuning would improve the performance of these other algorithms. In this case, SAC may be performing better because unlike the other two algorithms, it focuses on entropy rather than reward. Evaluating SAC using Tensorboard reveals that the entropy consistently decreases with iterations, as shown in Figure 24. Figure 25 shows the performance of an SAC policy over time. Episode reward, or the distance travelled each iteration, increases steadily.

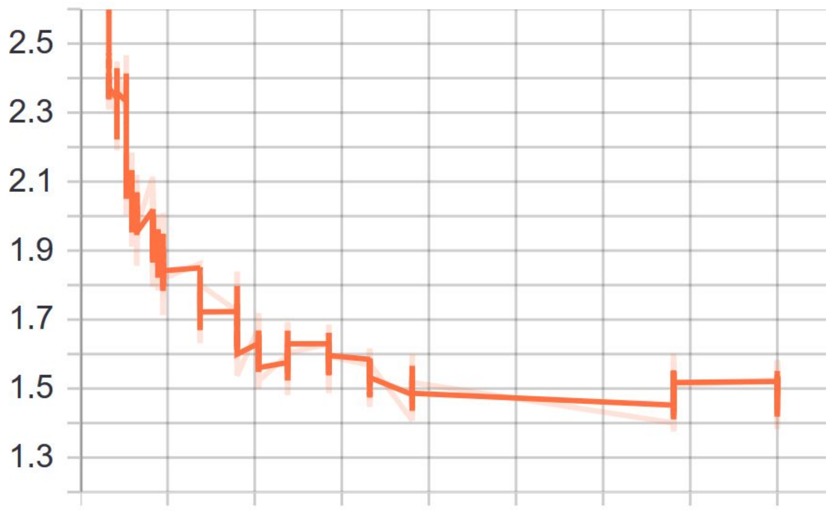


Figure 24: This Tensorboard graph shows the loss entropy of a policy trained using the SAC algorithm. As expected, the loss entropy consistently decreases with iterations. This policy performed exceptionally well in simulation, it is shown here after 500 iterations.

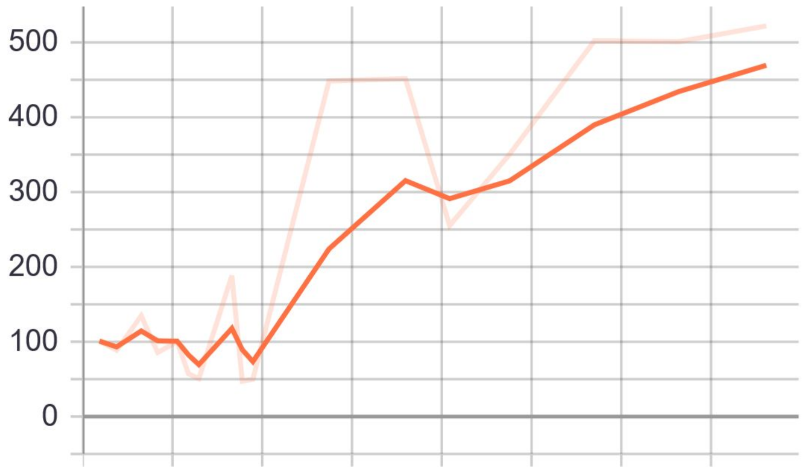


Figure 25: This Tensorboard graph shows the episode reward over time for a well performing policy, trained using SAC. Despite SAC's focus on entropy, the reward increases over time, meaning that the car is able to traverse further along the path.

4. Integration *(Authored by Claire Traweek)*

A key component of integrating discrete code was ØMQ, a system of servers and clients that maps inputs and outputs to different programs. Specifically, we used ØMQ to integrate Python2 and Python3, as our robot controller was written with ROS in Python2 but TensorFlow uses Python3. We created a server on the Python 3 side that, upon reception of a request, would capture images from the front webcams and then give them to the model to generate a prediction. This would be returned to the client, a ROS node that integrated with the rest of the hardware.

4.1. Lessons Learned *(Authored by Mitchell Guillaume and Claire Traweek)*

This week's lab was unique because it introduced very new software and hardware systems, requiring team management across non-integrated subsections. For many on the team, this was their first experience with machine learning. Another new aspect of this lab was the shared server. This was a new resource that had to be learned and shared by multiple teams.

4.2. Technical Lessons Learned *(Authored by Mitchell Guillaume)*

The team learned a lot about using TensorFlow to explore machine learning in a hands-on, non-theoretical way. We learned to use Python libraries to explore and augment our results and models

4.2.1. Types of Machine Learning

Three types of machine learning were introduced in this lab--convolutional neural networks (CNN), imitation learning, and reinforcement learning. Each of these methods has its advantages and disadvantages, so it's important to consider the context of the problem when deciding which method to implement. For example, imitation learning requires a lot of training data, whereas reinforcement learning doesn't require any. However, to properly implement reinforcement learning, once must define the set of rules governing the problem, as well as defining an objective cost function. The cost function is critical, as it will determine what the model will seek to optimize. CNNs are necessary for the models to recognize features and objects in images, making this an absolute must for implementing any of these machine learning labs.

4.2.2. Virtual Environments and Software Installs

This week's lab required the use of both new hardware and software systems. TensorFlow is a machine learning library that was used extensively. The team has not used TensorFlow previously in any of the labs, so there was a learning curve. Additionally, TensorFlow is only compatible with Python 3, while ROS is only compatible with Python 2. There were lots of small issues of this nature, like package dependencies, anaconda environments, Python versions, and more. The team also used Jupyter notebooks for the first time.

Another new tool introduced to the team this lab was the NEET server (specifications listed in Table 2). This is a powerful tool that allowed the team to log hundreds of GPU and CPU hours over the course of this challenge. However, it was a shared resource among three teams and an academic department. Having the server forced the team to plan ahead, communicate with other teams, and learn to use a very powerful piece of hardware remotely.

4.2.3. Technical Lesson Learned

This was the first and only challenge executed on the new “Model-X” version of the autonomous racecar platform. This race car is an updated version of the race cars utilized throughout the first six labs. The new cars feature a Nvidia AGX Xavier in place of the Nvidia Jetson TX2. Additionally, all of the sensors of the old car have been replaced by three webcams. As is expected when breaking in any new system, there were some slight teething problems. These included issues associated with the allocation of memory to the GPU and CPU by TensorFlow during machine learning algorithm training as well as permissions issues with Jupyter stemming from an improper initial install as the root user. By the end of the challenge, many screws had vibrated out of place and were lost, and many adhesive strips had failed due to heat from the processor. The team learned to appreciate the “debugging” of a new integrated hardware system and learned to expect unforeseen issues to arise.

4.3. Team and Communication-Related Lessons Learned

Because this week’s lab and associated challenges were not dependent on each other and could be developed in parallel, the team split up, and each team member worked on the part of the challenge they found interesting, including the final autonomous race. Despite this, there was still lots of good communication within the team, as members would ask for tips, exchange ideas, and help each other out when their own task was at a bottleneck.

For the first time, the team also had to share resources with other teams. This was the first time the team had to communicate externally with this level of intensity. There were very few miscommunications between teams, and the few that did arise were resolved fairly quickly.

4.3.1. Developing more nodes of Communication

This week, the chat we use to coordinate meetings unexpectedly failed. Due to this, we had to migrate our communication to Slack. We experienced some difficulties because not all team members checked Slack routinely, and not everyone was familiar with how Slack notifications worked. Working with this, we also contacted each other through other mediums and made sure to agree on clear plans while we could communicate in person.

4.3.2. Working on discrete labs concurrently

Our challenge was unique because it consisted of discrete subsections that never integrated but were related to each other. Our initial approach was to divide the parts into clean subsections, but it became clear that lessons learned from one part of the lab could be useful in another. As such, we found it useful to remain in contact and update each other as progress was made. As a result, multiple team members could focus on one part when problems arose and then return to their individual assignments once the problem was resolved. A downside to this is that nothing was completely finished early, but an upside was that everyone learned the project contents.

4.4. Individual Lessons Learned

The multitude of technical and teamwork related challenges faced during this lab contributed to the development of all members' individual skills.

"I learned the importance of keeping up to date with my teammates. Sometimes a problem that I had solved in one part of the lab cropped up again in a different part, and being aware of what was going on helped us speed our work up. Additionally, this was the first time I worked with Tensorflow, so it was interesting to see what it could do." -*Claire*

"Machine learning is an extremely popular topic right now in industry and academia, and by addressing this topic on a mobile racecar platform using cutting edge hardware, I was able to gain a much more intuitive understanding of the capabilities and limitations of the current technology and algorithms." -*Danny*

"Everyone is doing it, everyone is talking about it, and everyone wants to learn it. Of course what I'm referring to is machine learning. To some, it is a magic black box that somehow solves any problem. But there is a lot of data collection, parameter tuning and training involved in the implementation of machine learning algorithms and not many people are aware of that. This week we got our hands dirty doing just those things. " -*Franklin*

"This is my first exposure to the topic of machine learning. It had seemed like such a black box to me before this lab, especially as someone studying mechanical engineering. Fortunately, I now feel confident enough with my knowledge to have a conversation about machine learning and learn about it more in the future." -*Hector*

“In addition to experiencing machine learning for the first time, I’m also learning practical skills that are essential to large software development projects. It’s not just about writing code, it’s about managing and understanding an integrated system.” *-Mitch*

5. Future Work *(Authored by Hector Castillo and Claire Traweek)*

We identified several areas for improvement that could be implemented given more time and data.

5.1. Building Better Models with More Data

As with almost all models, more data would improve our robot’s performance. For instance, the imitation lab was trained on a small dataset only consisting of a dozen of so loops around Stata. The limits of training data is a limiting factor to how well our model can perform and generalize. Even in the same Stata hallway, when placed slightly off of the starting position, the race car has difficulty outputting reasonable steering angles. To address this issue in the future, during data collection, the car can drive with a variety of paths, each different in their turning radii and distances to walls/ obstacles. To further develop this model enable the race car to drive in other hallways, much more data is required in order to train a model that is fully capable of navigating hallways that it has never seen before. Additional augmentations could also extend the data we already have.

As encountered on demo day, the Stata basement looks a lot different at varying times of the day and depending on where objects are moved around. One particular complication was the number of people in the hallways on demo day. Very little of the training data collected contained people in frame, so the robot was thrown off. Collecting more training data with people in view and at different times of the day would make the model more robust to this environment.

In the case of the gate detection lab, a more refined model with more specific data could be trained to target the top of the gate, rather than the legs, improving the accuracy at which we can aim at gates and distinguish gates from gate-like objects like doors.

As is the case with most reinforcement learning implementations, running the Donkey Car in simulation for thousands more iterations would improve the smoothness and consistency with which it drives.

5.2. Identifying Gate legs individually

A known issue with our gate-identification algorithm is that the robot drives at the average position of the gate, so if only a single leg is visible, the robot will drive into it rather than into the gate. A potential solution is to train the model to identify the checked pattern at the top of the gate rather than the legs (as the heatmap shows it does). Ideally, the checked patterns would be obscured by the limited camera field of view simultaneously, so the car would continue driving straight at the center of the gate.

5.3. Driving at Higher Speeds

In both cases, we trained the car and implemented our solution at lower speeds for safety. An interesting augmentation would be to increase speed and include training data with slippage to build a path that's potentially faster than a human-driven trajectory. The controller used for autonomous racing in the basement of Stata calculates driving speed based on the steering angle. Therefore, the car should be able to easily perform at higher speeds by implementing this controller.

5.4. Training a different VAE

In the reinforcement learning lab, we used a pre-trained VAE. When we trained our own VAE, we used the same network architecture as the provided VAE, so further experimentation with newer methods could yield better results. Another option is forgoing the VAE altogether, and training with the image directly, potentially with some modification to the network.

5.5. Generalizing Imitation Learning

The imitation learning model implemented in this lab only works on environments that are similar to the environment that the training data was collected on. The data was only collected in two runs around the entire loop for a total of around 15 loops worth of images. These were recorded around the same time in the same day. This doesn't give the agent much to go off of when it receives an input image of the stata loop on a different day or an image of a different hallway. In order to generalize, much more data is required. This data should include other hallways besides the stata basement and should include a variety of obstacles.

5.6. Moving Reinforcement Learning to the Race Car

The next step for our reinforcement lab is to move to the physical racecar. This would entail constructing a VAE from images captured while driving, potentially sourced from the data used for the imitation lab. Then, the car would have to be watched and corrected for a few hundred trials in the Stata basement before it could learn to drive on its own.

This document was revised and edited by Mitchell Guillaume, Daniel Wiest, Claire Traweek, Hector Castillo, and Franklin Zhang