Free Space Detection and Trajectory Planning for Autonomous Robot

By: Zachary Winger

Advisor: Dr. John Seng

Department of Computer Science and Software Engineering

College of Engineering

California Polytechnic State University, San Luis Obispo

June 8, 2020

Table of Contents

Introduction

An important ability of autonomous robots is being able to know what’s around them and where it is safe to move to. Determining these two things can be accomplished, most of the time, without any special sensors or equipment by using a computer vision system; a system that produces output from an image. There are many different ways to create a computer vision system, all suitable to their own applications. But nowadays one technique seems to stand out, neural networks. Given enough time and data, a neural network can be designed to predict most anything, including a robot’s surrounding and safe driving paths. Along with the help and advice of Dr. Seng, we aim to develop such a system for his autonomous robot, Herbie.

Problem Statement

For this project, there were two main goals we wanted to accomplish. The first of the two goals is to create a neural network model to predict the free space surrounding Herbie as it drives around in real time. In this project, we are defining free space to be the area surrounding Herbie that is unobstructed by people or objects, and that Herbie is able to drive through without driving into anyone or anything. As such, it is restricted to only paved pathways.

The second goal of this project was to use the predicted free space from the neural network model to determine what trajectory Herbie should take in order to continue driving safely. Since this goal requires us to have a predicted free space boundary to use for our trajectory calculations, we needed to first develop the free space detection model and get it into a state where it could be used to accurately predict the free space surrounding Herbie.

Software

All code written for this project was done using Python3, and the model was created using Keras to aid in creation and training. Image processing was performed using OpenCV for Python3, and running the model for testing was done using Robot Operating System (ROS). Using ROS allowed us to use ROS .bag files to read in video and output video to its own ROS topic for viewing.

Data Processing

Creating a new neural network based model requires a significant amount of data to get the model into a state where it will predict the desired output with high accuracy. Because we wanted Herbie to be able to wander around Building 14 and the surrounding area, we needed to collect our own data to use in creating the model since this data was not readily available to us. This data was collected by taking frames from video taken around Building 14 and the surrounding area (Figure 1).

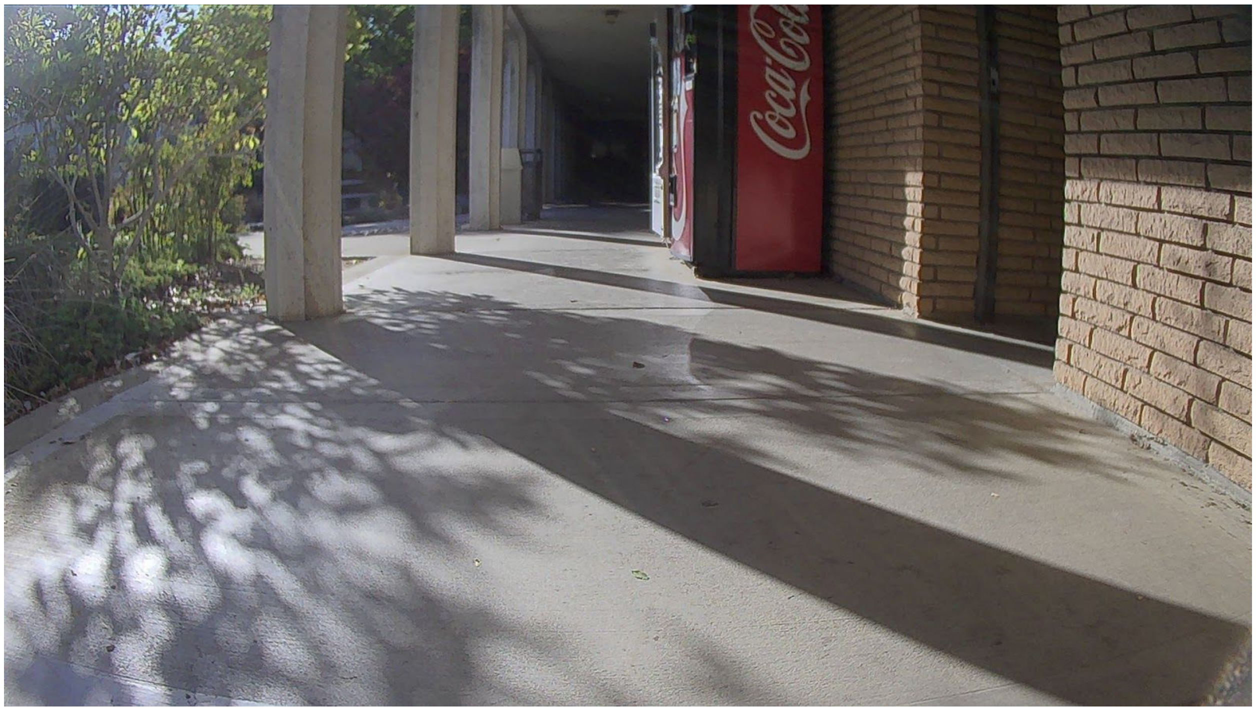


Figure 1: Original Image captured from Herbie

Once the data had been collected, it wasn't quite ready to be used yet. The images still needed to be labeled with what the expected free space was in each image. To do this labeling we used the website Labelbox. Labeling the free space in each image was done by drawing a red polygon that encompassed all areas in the image that were considered to be free space (Figure 2).

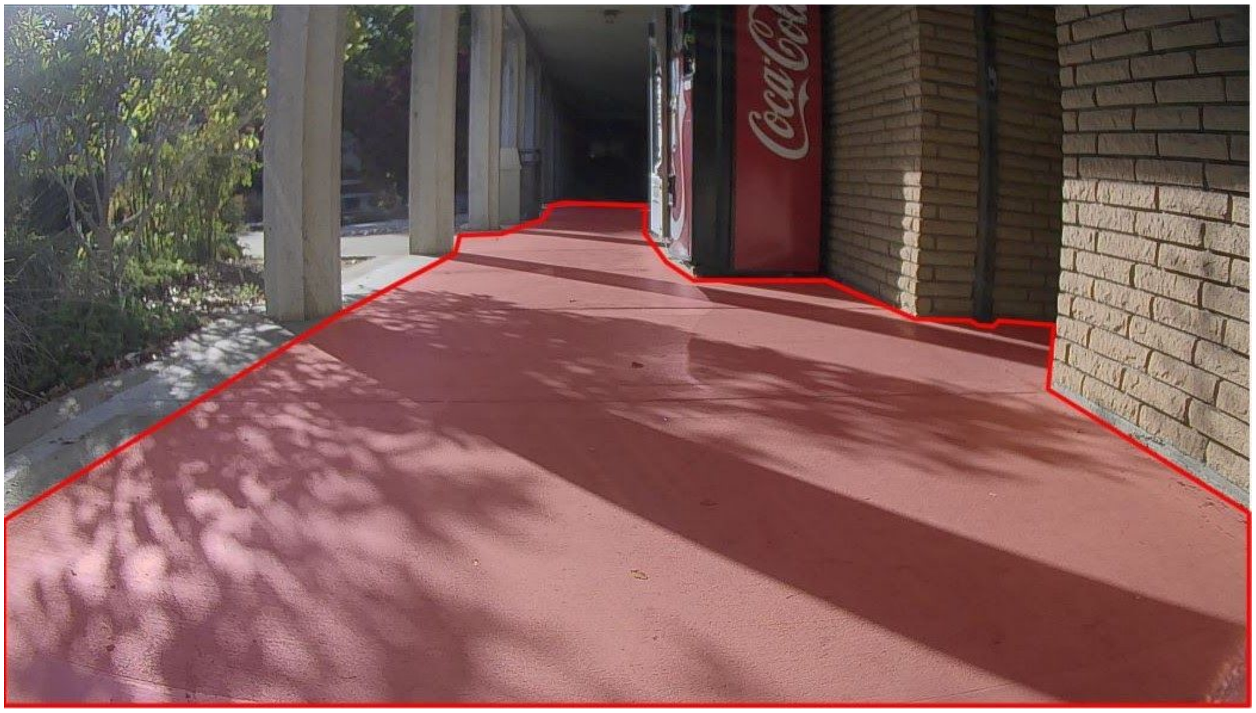


Figure 2: Expected free space shown in red

In order to extract any information about the free space polygons, the labels were also represented as a black and white mask, where white was the expected free space and black was not (Figure 3). Doing so allowed us to more easily determine the free space boundary without having to worry about the actual contents and colors of the original image.

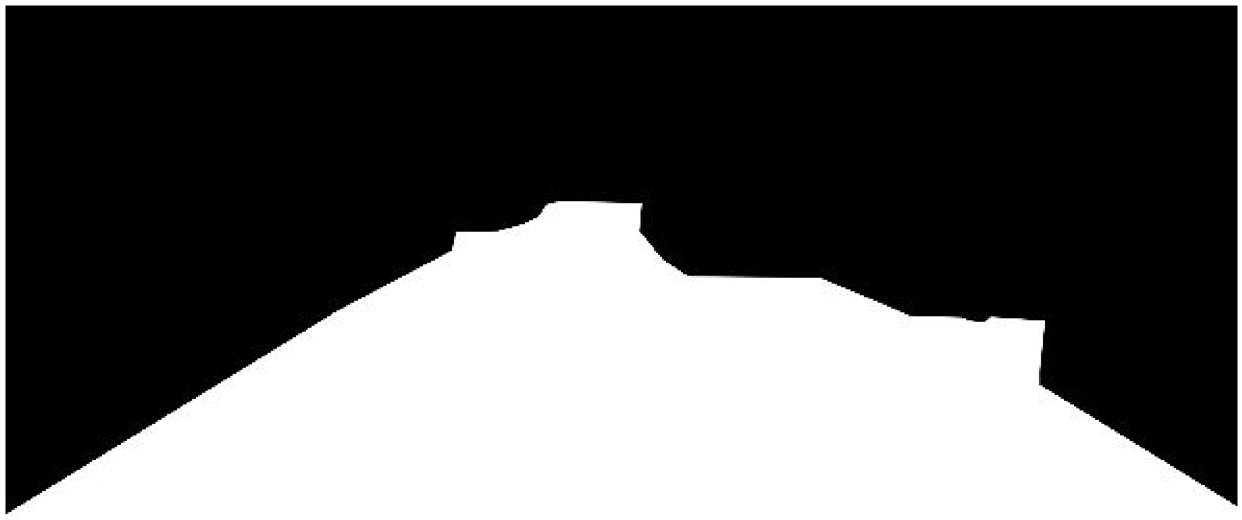


Figure 3: Free Space Boundary Mask

After the collected data had been labeled, there was the issue of processing it into a state that could be used to train a neural network model. By using Labelbox to label our data, once all the data was labeled a .csv file was created that contained the metadata associated with each image, including a URL to the original image and an array of geometric points describing the polygons(s) of free space. This format gave me an easy way to create a Python3 script that would download all of the images and their metadata, and recreate the image masks to extract the free space boundaries. In order to recreate the mask for the free space, I created a blank image with equal size to the original image using numpy and used OpenCV to draw the free space polygons onto it in white. After the mask had been created, the mask and the original image were then resized to a resolution of 640x360 pixels to match the input size of the free space detection model.

With the mask resized to its correct resolution, I could go about extracting information to describe the free space boundary. Since the model was not going to output images depicting the free space, the free space was going to be instead be represented by a feature vector of 128 elements. In order to create this vector for each image, the script I had written traversed each mask from the bottom of the image to the top every 5 columns. At each run through the loop, when the pixel color switched from white to black, the current Y-coordinate was added to the free space descriptor. The transition from white to black marked the extent of the free space in that given column of the mask. So, by performing this process for every 5 columns of the mask, a feature vector describing only the boundary of free space could be produced. This process could have been done for every column in the mask, resulting in a 640 element feature vector, but for the purposes of this project, 128 points was sufficient enough to obtain good predictions from the model. In order to see that the correct boundaries were being produced, I used OpenCV to draw the point onto the original images in green (Figure 4).

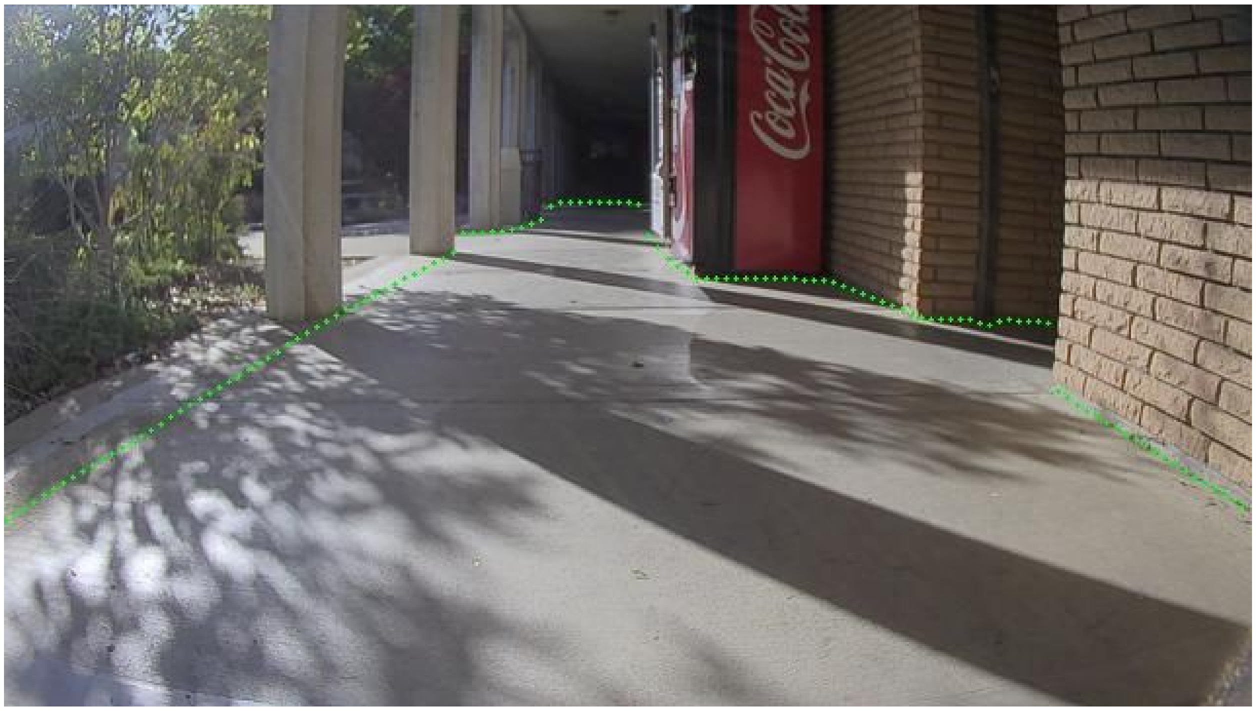


Figure 4: Extracted free space descriptor shown in green

In order to use the created feature vectors in training the model, there was still a little more processing that needed to be done. The first thing that needed to be done was to normalize the values in each feature vector to between 0 and 1.0. This was done because the model itself will produce values between 0 and 1.0, so when training the model we need to do the same, and in order to use the output from the model the values can then be scaled back up to between 0 and 360. The last thing that needed to be done was to split the data into two groups, training and validation. 85% of the data was put into the training set, and the remaining 15% was placed into the validation set.

Since the development of the model was an evolving process, this script was also written to accommodate that. The script uses as a command line argument a file containing the specifics for processing the data, such as the resolution to scale images to, the number of feature points to extract, etc. This allows the single script to process the data in a multitude of ways to accommodate many different potential model configurations. Having the script is also beneficial in that anyone who has the .csv file and the script can download all of the data onto their machine to use. It also allows for new data to be downloaded at any time, without having to re-download any of the old data. Or, the user can decide to re-download all the data from scratch. Each time the user downloads data, the newly downloaded data is split into the training and validation sets in an 85:15 split, which can also be modified to whatever the user wants using the command line arguments. The script also notifies the user of any images that may have been mislabeled which may not be suitable for training.

Free Space Model

Once data had been collected and processed, we could begin working on developing a model for predicting free space in an image. Neural network models can become large very quickly, containing millions of parameters, however, because this model needed to be running in real time, on Herbie, it needed to be kept relatively small so it would perform faster, yet still be accurate enough to correctly predict the free space in a given image.

The type of neural network model we ended up choosing to use for the free space detector was a MobileNetV3 model. The advantage of this type of model is that it was designed to be used for cell phone CPUs, so it was small, around 2 million parameters, and fast, which is what we wanted, and the original purpose of the implementation we used was for object detection. While object detection is not quite the same as free space detection, the two problems are similar, so a model made for one should perform well for the other. This type of model also has two variations, a small and a large, which change the number of parameters in the model. Since we wanted to keep the number of parameters as small as we could, we opted to use the smaller variation of the model. Many open source implementations of this type of model have been created, so we were also to use one of these implementations[1] as a starting point for our model, although it did require some modifications to get working with our data.

The first major change that needed to be made to the original model was to change the model's expected inputs and outputs. The model already expected images as its input, but we needed to change the resolution of the images to be 640x360 pixels. The next major change was to modify what the model output as predictions. Since the original purpose of the model was for object detection, it originally output an array of 1000 elements containing a prediction of the likelihood each type of object was in the image.

[1]https://github.com/godofpdog/MobileNetV3_keras

We instead wanted the model to output an array of 128 Y-values representing where the boundary of free space was in the image. To do this, the model needed to instead limit the array of 128 numbers between 0 and 1.0. There was also a training script that came along with the original model, so once the modifications had been completed, training could begin; which was carried out using the Keras library.

Training the model did not prove as successful as predicted, so instead, we found a new implementation of the MobileNetV3 model[2], which was able to easily replace the old one in the training script. The same necessary changes needed to be made to the new model as the old one, but it was still compatible with the training script from before. This new model trained faster and produced better results than the old model. Once we had been successful in getting the model to begin training on our data, we began to work on decreasing the total number of parameters in the model to help speed up the performance. The original model proved to be larger than was necessary, so in each layer of the network, we began to trim down the number of nodes to decrease the overall number, which would speed up its performance. We could not decrease the number of nodes too much, however, otherwise we would begin to see a hit to the accuracy of the model.

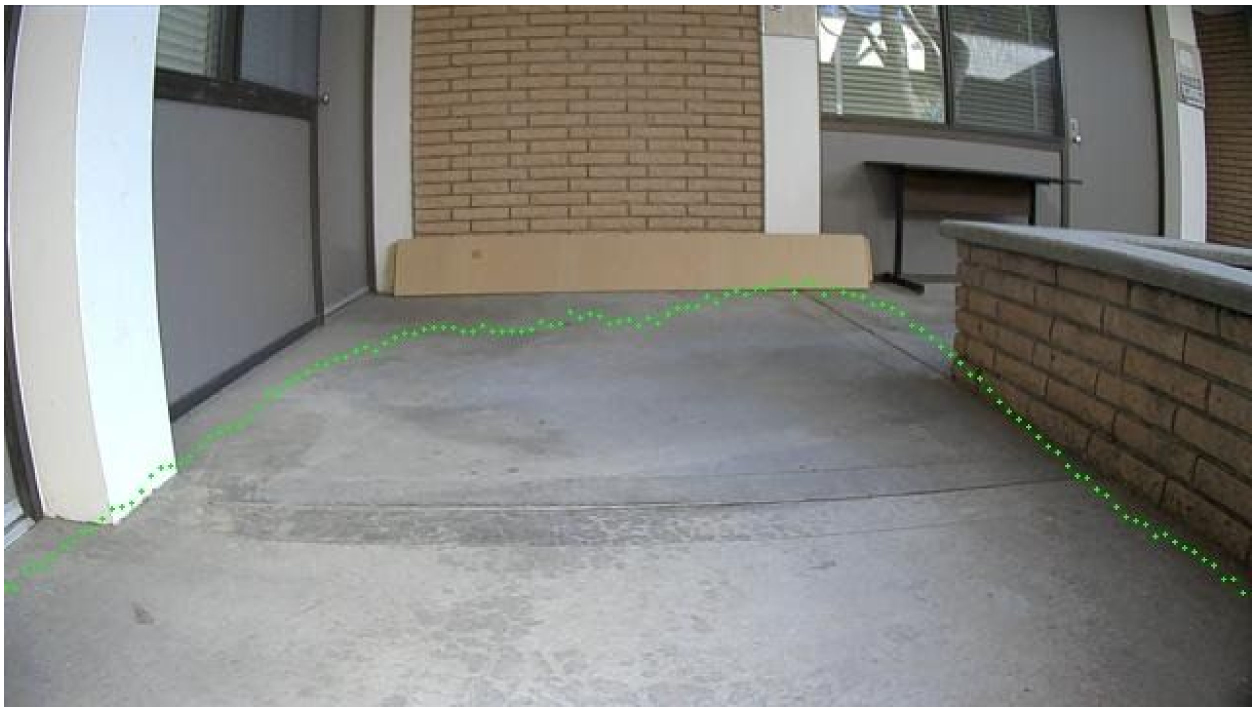


Figure 5: Model prediction with “wavy” noise

[2]https://arxiv.org/abs/1905.02244?context=cs

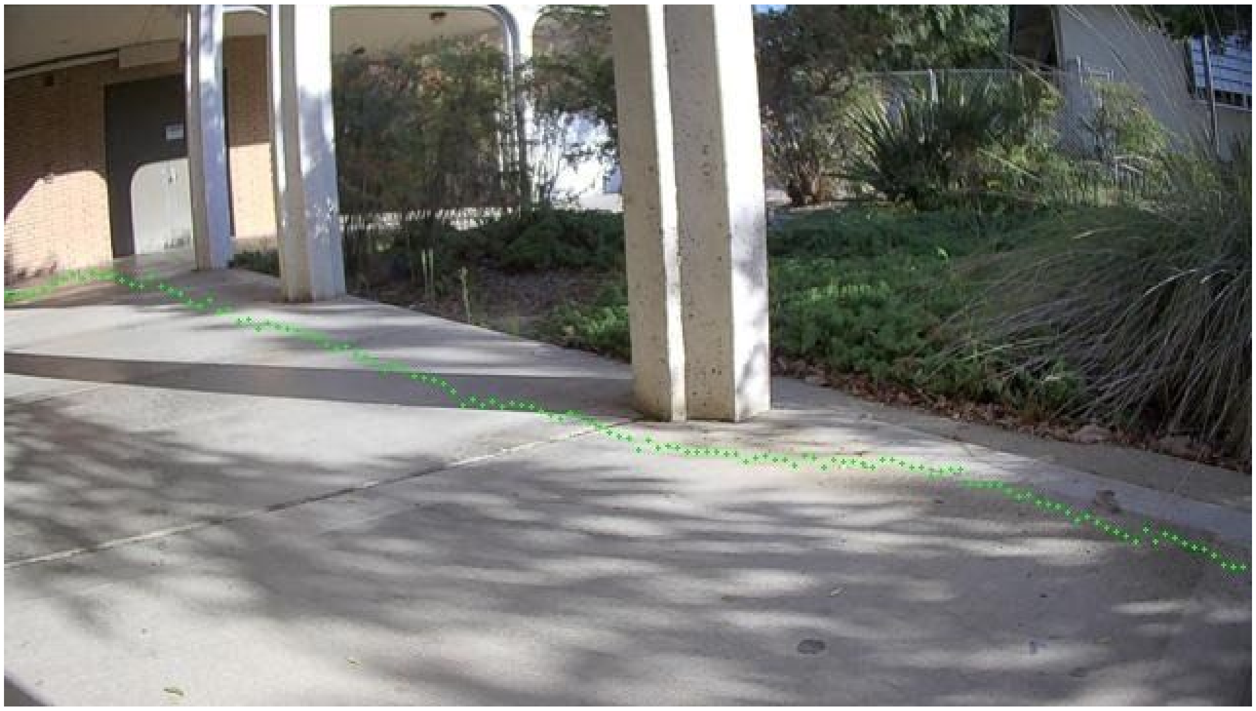


Figure 6: Model Prediction with “wavy” noise

After training the newly modified model on our data, we noticed there was still a fair amount of “wavy” noise in the resulting free space boundaries (Figures 5 & 6). One solution we took to this problem was to also train the model with dropout. Doing this would randomly ignore some nodes in each layer of the model. The idea behind doing so is to help reduce the chances of overfitting the model to training data, and improving its ability to generalize. This implementation did improve the models ability to remove some of the “waviness” in its predicted boundaries. During training we also noticed that just a limited amount of data was preventing us from training the model to the best it could be. As more data was collected and processed, we found the model to perform more accurately than before (Figures 7 & 8). To help increase the amount of data that could be used for training, we not only collected new data, but implemented image augmentation to augment the images we already had to be used as new training data. By doing this, we could take the data that had already been processed, augment it slightly, such as a rotation, which would create a new image, and add this new image to the training set.

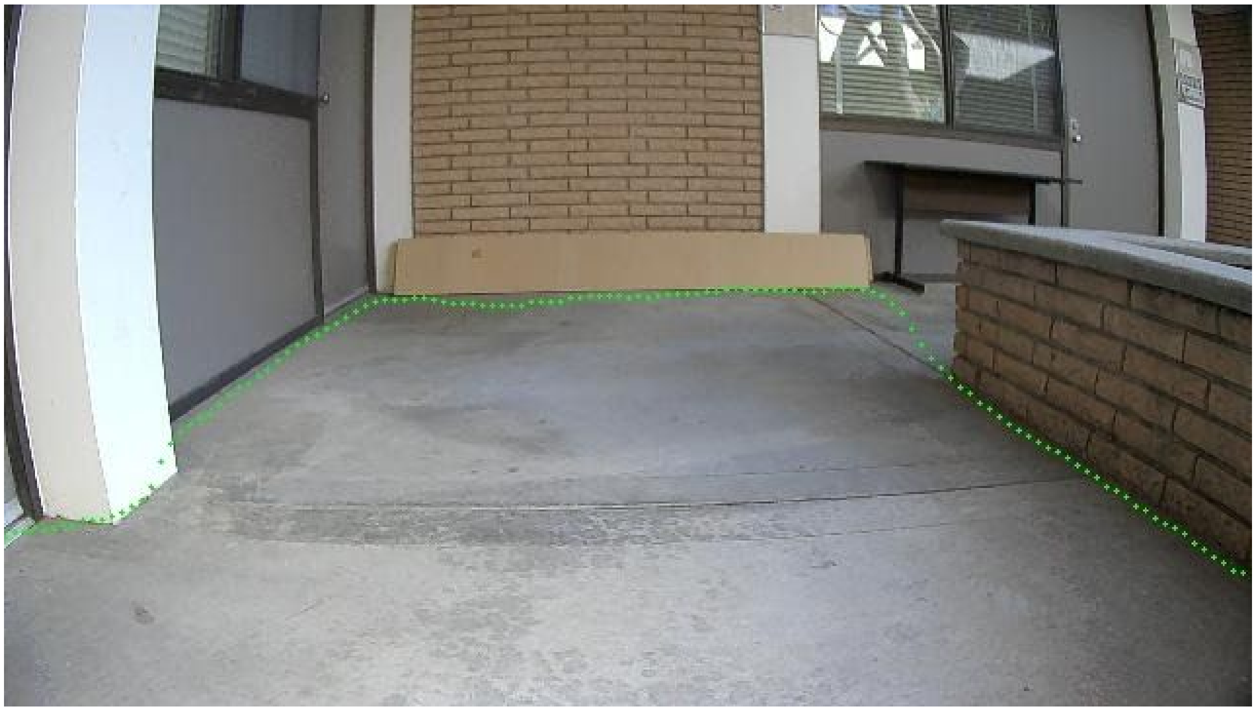


Figure 7: Model Prediction with reduced "wavy" noise

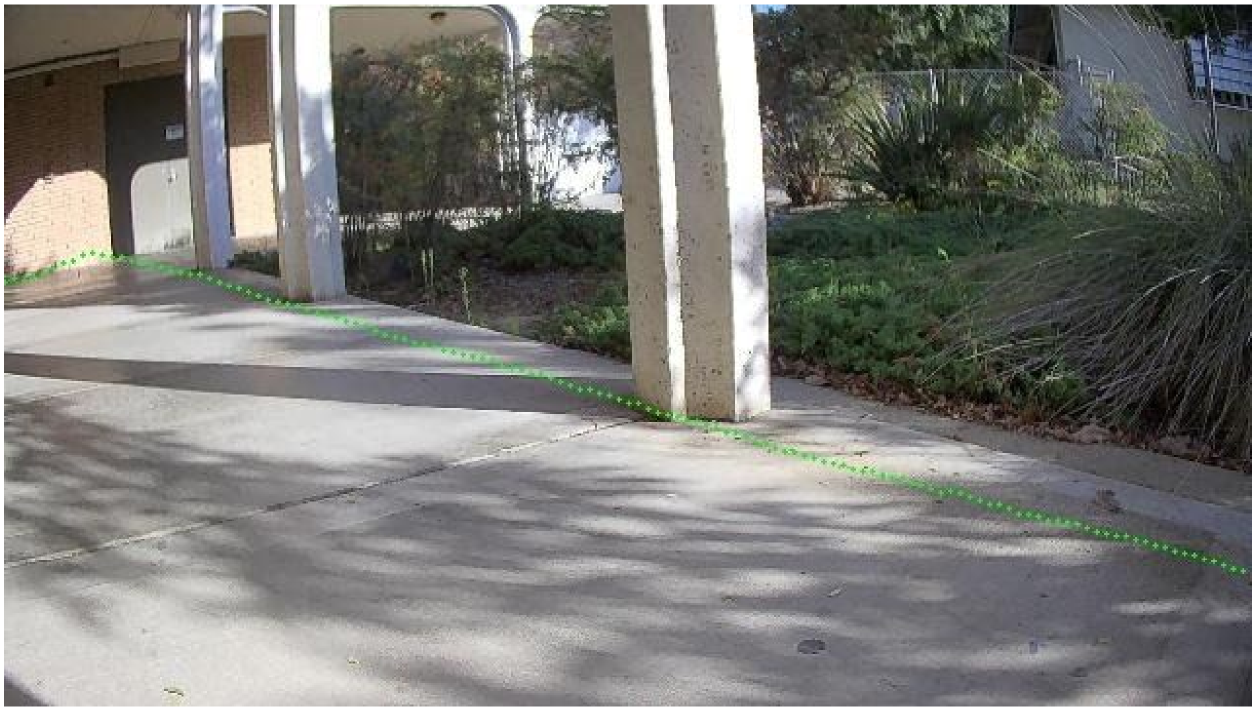


Figure 8: Model Prediction with reduced "wavy" noise

In order to view the results of the model after each time it was done training, I developed a Python3 script to visually display the predicted boundary on the image used as input. To do so, the script used a directory of images as input, each of which it would run through the model. After an image had been run through the model and its free space boundary predicted, each of the 128 points representing the boundary were drawn onto the image using OpenCV. This script was later improved to predict the free space in each frame of a video. To do this, video frames were read from a .bag file which contained video recorded directly from Herbie. Each image was then run through the model, as before with the still images, and the resulting boundary was drawn on top using OpenCV. The resulting image was then published to a new ROS topic, which could be recorded itself and viewed live as the video ran through the model, or saved in its own .bag file for use later.

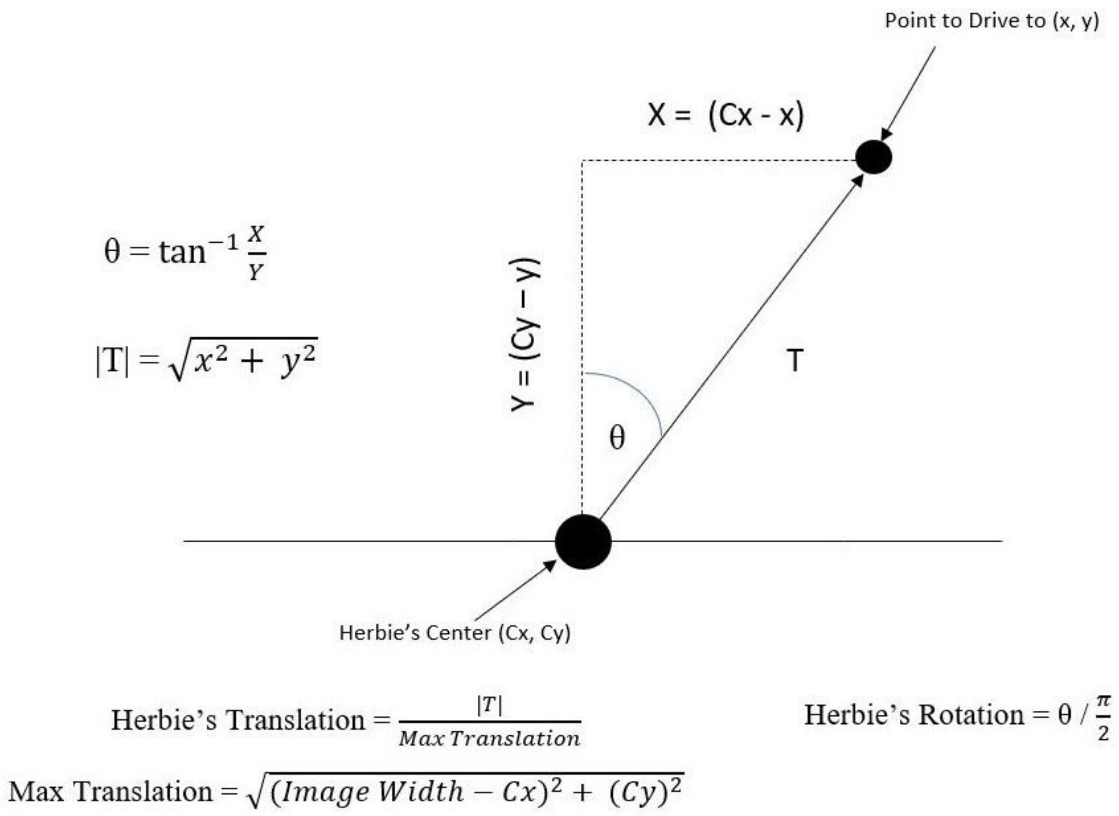
I also created a Python3 script to aid in determining what new data should be labeled when it was collected. When new data was collected, there needed to be a way to separate out the types of images the model had a difficult time accurately predicting from the ones it could accurately predict easily. To do so, the free space model was trained on our existing data 4 times to give us 4 different, yet similar, models. By using 4 different models, each of the new images could be run through the 4 models, and the images that produced the most varying results from all the models would be the ones that ended up getting labeled. If all the models predicted very similar results for any of the images, those images most likely did not need to be added to the training set. To determine which images were the most difficult, each image was run through each of the models, and the resulting predictions recorded. The average of the predictions for the image was then calculated and the squared error between the average prediction and each individual prediction was found, all the squared errors for the image were then added together. The sum of squared errors was then used to determine the images ranking among the rest of the new data. Once all the data had been ranked, the top 20% of images with the worst sum of squared errors would be the images that were labeled and added to the training set.

Driving Policy and Trajectory Calculation

After the free space model was predicting well, the next step was to determine a driving policy for Herbie and use the model's predictions to calculate the trajectory Herbie should take in order to follow this driving policy. The initial driving policy I created was to have Herbie drive toward the point in the predicted free space furthest from itself. If there was an obstacle in its path, turn to left or right, whichever direction has the furthest point away from itself.

In order to determine the correct trajectory for Herbie to take, I created a function that would take in the predicted free space vector from the model, and return the translation and rotation for Herbie take, represented as percentages. These values were represented as percentages since they would be used as the motor speeds for Herbie. In these calculations, Herbie's center is represented by the bottom center of the image. Since part of Herbie is also in front of and to the sides of the camera recording the video, this area is also considered when performing the calculations.

If there are any points in the model's predicted free space within Herbie's region, the function determines there is an object directly in front of Herbie, and therefore Herbie must turn to the left or to the right. The direction in which Herbie turns is determined by which side has the furthest point away from Herbie. This decision is made because Herbie will most likely be able to go further by traveling to the side with the furthest point after it has cleared the obstacle. In these situations, Herbie will have no translational movement and a rotational movement of 1, full speed left, or -1, full speed right. If no obstacles are detected to be directly in front of Herbie, the furthest point away from Herbie's center in the model's prediction will be chosen to drive towards. To determine the translational speed percentage, the magnitude of the vector from Herbie's center to the furthest point is taken and divided by the max translation magnitude Herbie could take; which is from Herbie's center to either of the top corners of the image. The angle at which Herbie should turn is calculated by taking the arctangent of the distance the point is from Herbie's center in the X direction divided by the distance the point is from Herbies center in the Y direction. This angle is then divided by $\frac{\pi}{2}$ to get the rotational speed at which Herbie should turn (Figure 9). For a visual representation, I used OpenCV to draw the trajectory vector, along with the free space boundary and Herbie space onto the image used in the calculations. (Figures 10-12)



$$\theta = \tan^{-1}\frac{X}{Y}$$

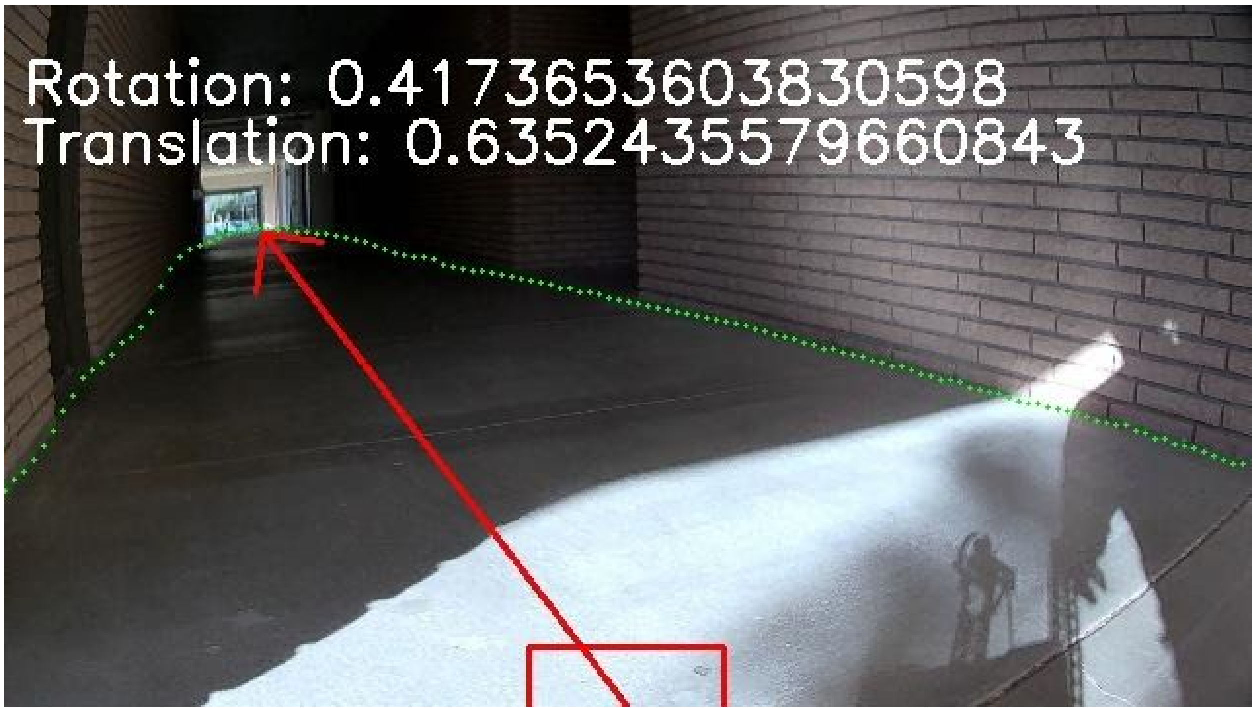
$$|T| = \sqrt{x^2 + y^2}$$

$$\text{Herbie's Translation} = \frac{|T|}{Max\,Translation}$$

$$\text{Herbie's Rotation} = \theta \,/\, \frac{\pi}{2}$$

$$\text{Max Translation} = \sqrt{(Image\ Width - Cx)^2 + (Cy)^2}$$

Figure 9: Trajectory calculation when Herbie is not blocked by an obstacle



```
Rotation: 0.4173653603830598
Translation: 0.6352435579660843
```

Figure 10: Predicted trajectory with no obstacles

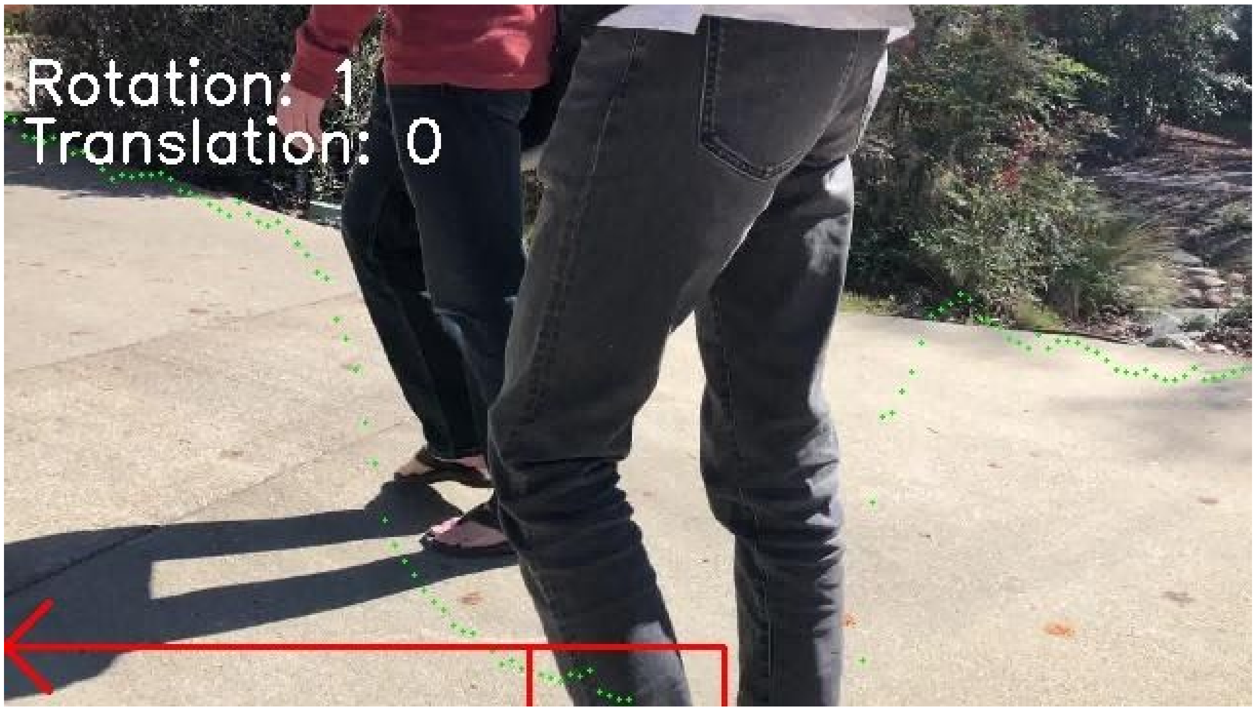


Figure 11: Predicted trajectory left to avoid the obstacle

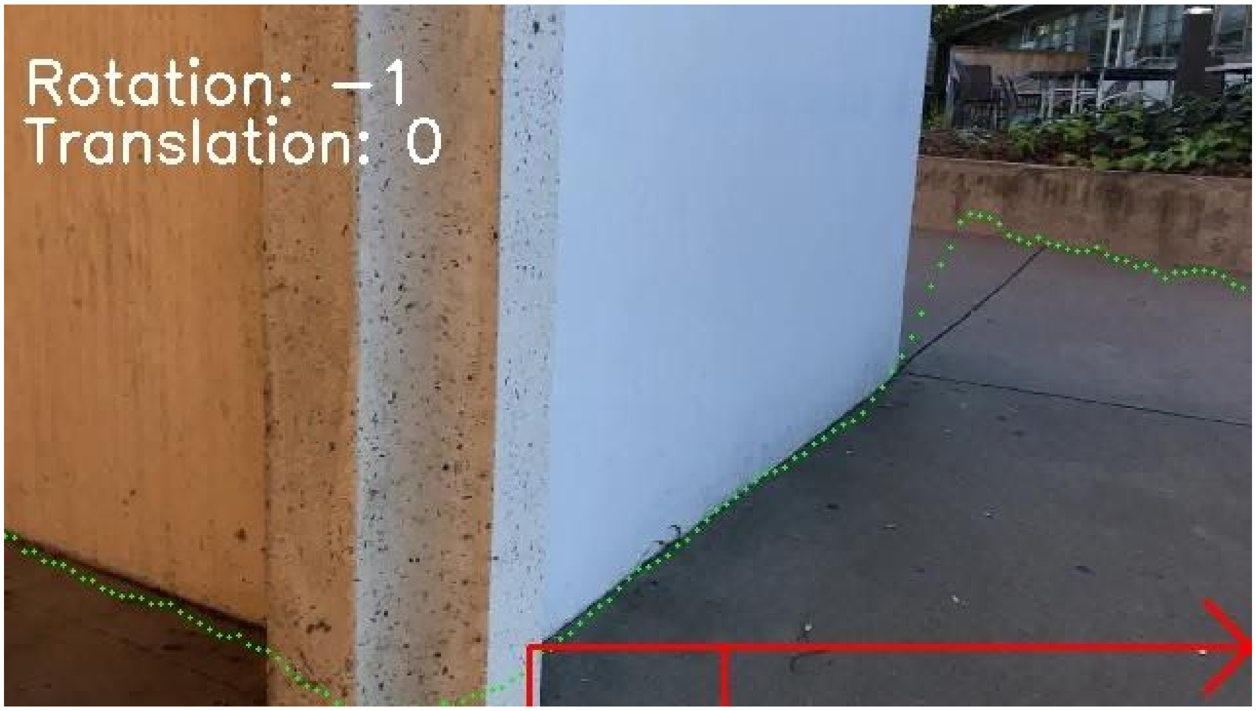


Figure 12: Predicted trajectory right to avoid the wall

Similar to when testing the model, at first I only tested the driving policy on still images, so the next step was to test it on an actual video. To do so, I modified the script I used to perform inference on video when testing the model's predictions. After running each video frame through the model and drawing the predicted boundary, I needed to then calculate the trajectory and draw the trajectory information onto the frame using OpenCV. Although this approach did use a pre-recorded video so not all frames had a trajectory that matched the actual movement of Herbie, it worked well enough to see what the program was doing when given a video.

Although the program worked for the video, I was able to see that the calculated trajectory would jump around the image when given a fairly straight, horizontal free space boundary, such as the one shown in Figures 13 & 14. This was due to the fact that there was noise in the model's predictions. Training the model on more data did reduce this noise, but there is only so much that can be done to reduce it and there will always be some noise in the prediction. Since the policy was to drive towards the furthest point, with the added noise in the model, this point would not always be the same, so the trajectory vector would jump from side to side of the boundary line. In order to smooth this out, I made a slight change to the driving policy. Instead of always selecting the furthest point to drive to, if there were a line of points close enough together, as with the horizontal boundary lines, Herbie would drive towards the median of the points. This was accomplished by keeping an array of any point that was -2 and +5 pixels away in the Y-direction of the furthest point. If any point was further away, this became the furthest point, and all other previous points would be discarded. Once all the points in the model's prediction had been gone through, the median point in the array was chosen as the point to drive towards.

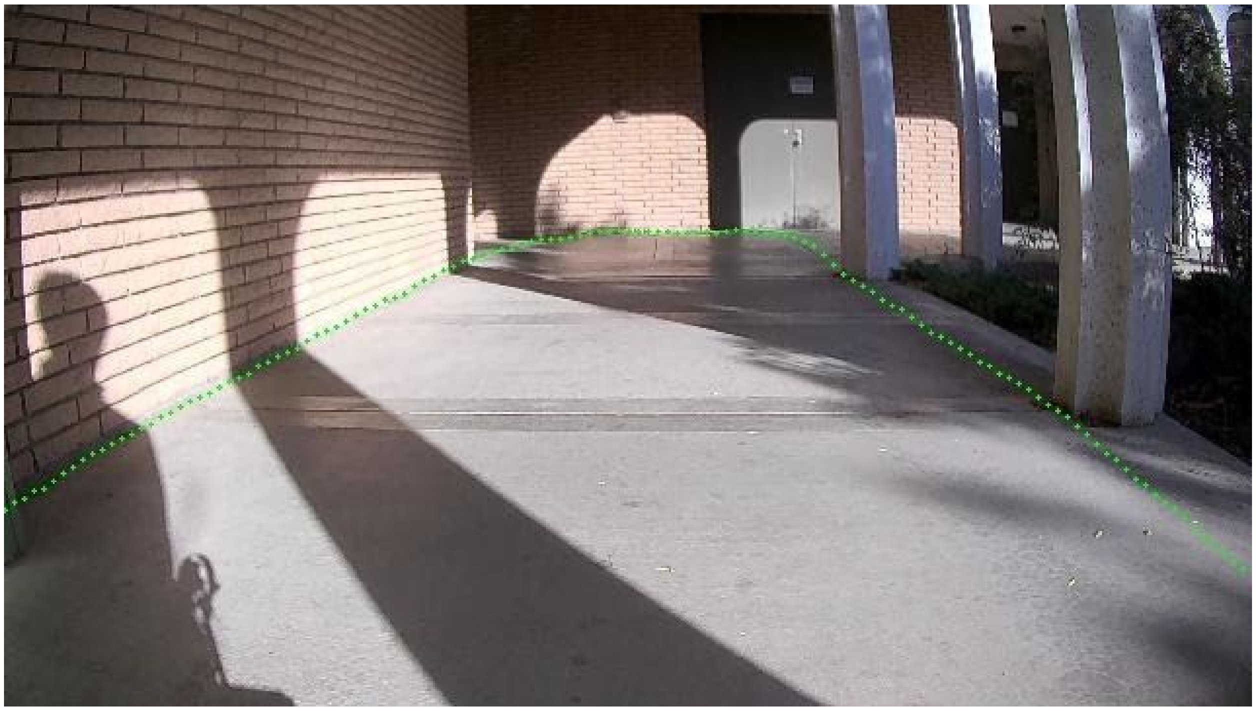


Figure 13: Prediction with horizontal segment

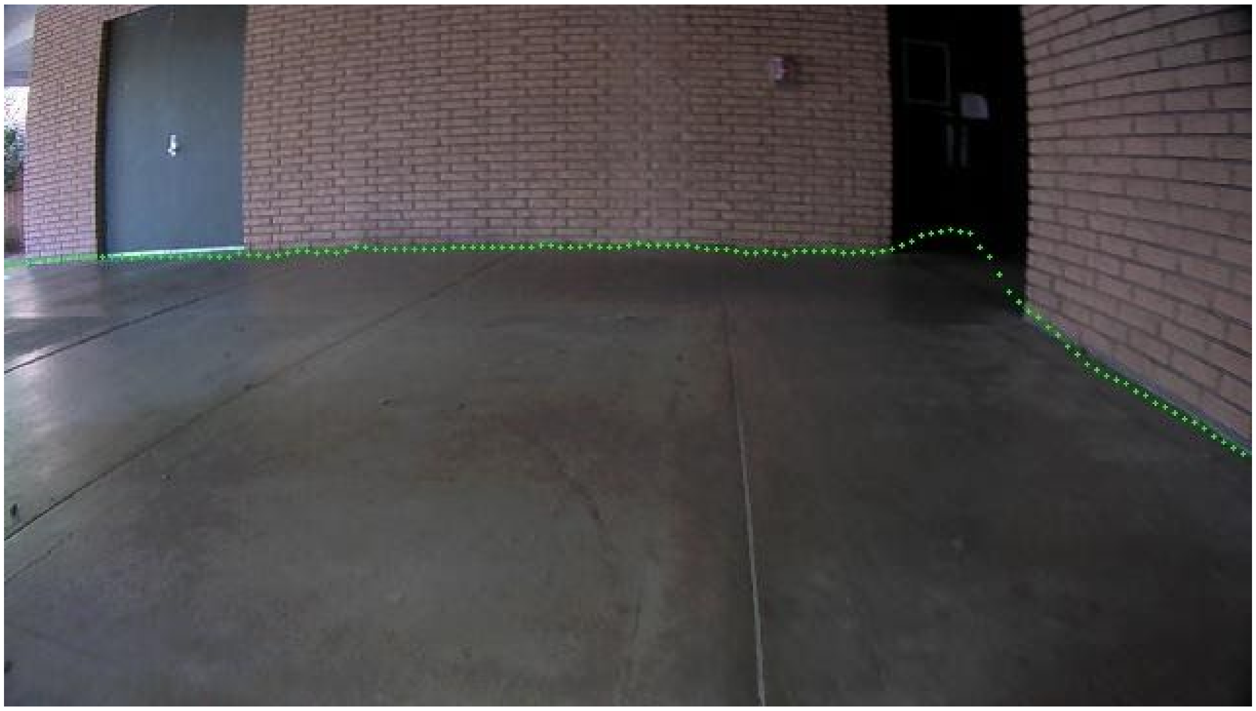


Figure 14: Prediction with horizontal segment

In order to see if this change did anything, I measured the overall difference in rotation between frames of the video I had been using and compared the results of the original driving policy and the new one. I found that there was a significant decrease in overall change in rotation using the new policy, confirming I was headed in the right direction to smoothing the “jumpiness” of the trajectory calculations. I tried testing with different tolerances for what would be considered close enough to the furthest point, but found the -2 and +5 pixel tolerance to perform the best.

Lessons Learned

Before working on this project, I knew that neural networks could be powerful, but I never truly realized just how powerful these models could be. The ability to create a model that is capable of a high degree of accuracy for our problem amazes me. It is even more amazing when the inner workings of the model are essentially a black box. I have previously taken a computer vision course in the past which focused on more traditional techniques for solving these types of problems and clearly defining all aspects of the models, but when compared to a neural network approach, it’s hard to fairly compare the two. While there still are situations where a more traditional approach would work better, more complex problems seem to benefit from a neural network approach.

Taking the approach of using a neural network did pose its own challenges at times though. Since the model acts like a black box in that we don’t know what it is actually doing in between the input and the output, it can be difficult to debug. Early on in the development of the free space detection model, even though the model was predicting free space boundaries, the predicted boundaries contained a fair amount of noise in them. At the time, it wasn't known if there was a problem with the model itself, or just not enough data to train on. The addition of more data began to reduce the noise, which would make it seem the noise was due to not having enough data. But further modifications to the model, such as drop out, also decreased the noise, so perhaps a mix of both caused it. Either way, the exact cause of the noise in any model can be difficult to pinpoint due to this black box nature.

One thing I also didn’t realize was just how much data is needed to be able to train a model to predict with high accuracy. I knew training a neural network was a data driven process, but I never thought about how much was needed. The first iterations of the model were being trained on around 300 images, which in the beginning seemed like a lot to me, but as mentioned previously the model predictions still contained a fair amount of noise in them. But as more and more data was added to the training set, the noise began to reduce. The final iteration of the model I tested with was trained on around 1300 images, which looking at it now is not very much considering the model was only trained on images around Building 14. If we wanted to expand the model’s capabilities to more areas, images of those areas would need to be collected, further increasing the size of the data set.

Actually training and running the model showed me that in order to do so in a reasonable amount of time, a GPU is a necessity. Due to the calculation intensive nature of neural networks, running solely on a CPU takes significantly longer than running on a GPU. When I ran the model on my own computer, which does not have a GPU, to see the model's predictions, what should have taken a couple minutes would take nearly half an hour. It just is not feasible to run these types of models in real time on a CPU, and training them is no different. Training these types of models on a GPU can already take a large amount of time to do. In the beginning, when the data set contained only around 200 images, training the model by running through the data set 10 times would take a few minutes running on a GPU. But when I attempted to train it using my CPU, it took a couple hours. Trying to train the model on the current data set of around 1300 for 400 iterations on my CPU would be insane.

Whenever I've thought about neural networks in the past, I've only thought about what it is the model would output, in this case the free space boundary, but never how to actually use the result. Since our model predicted free space, in the beginning I would only think about how that would appear on an image so I could have a visual representation of it; I had a difficult time trying to see how I could actually use these predictions for anything other than just seeing them on an image. However, actually using the model's predictions to create a driving policy for Herbie, made me think about the prediction in more than just a visual sense. Doing so helped me start thinking about what the prediction actually means and how it could be used by Herbie.

Conclusion

The model that was created to predict free space in an image was a success. There is still some "way" noise in the predictions, but with the addition of more data to train on, this noise will be reduced. The driving policy created for Herbie also appears to be successful. However, due to the COVID-19 pandemic during Spring quarter 2020, we were unable to test the driving policy live on Herbie. Instead, we could only test it using pre-recorded video. But, in testing with this video, the policy behaves as expected, and appears it would work in a live setting.

Overall, this project was successful, and quite enjoyable to be a part of. Robotics, especially autonomous robotics, is something I've been interested in since before this project. So, being able to be a part of a project for an actual autonomous robot was an amazing experience that taught me so much about neural networks and their application to autonomous robotics.

Appendix: GitHub Repository

All the code used in this project can be found in out GitHub repository here[3]

[3] https://github.com/zwinger/Senior-Project