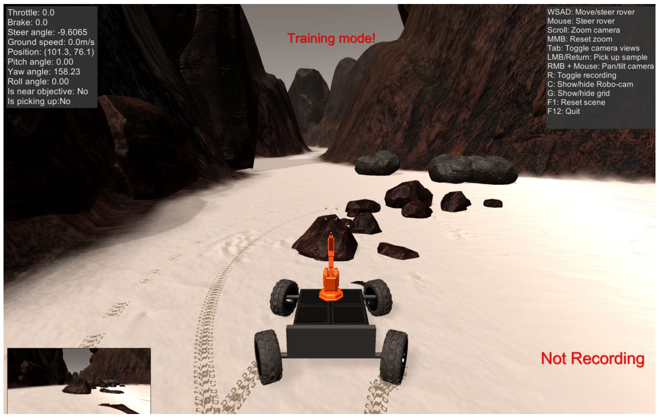
Martian Learning: Imitation Learning on a Mars Rover

By Brian Xu and Hairuo Sun



Throttle: 0.0
Brake: 0.0
Steer angle: -9.6065
Ground speed: 0.0m/s
Position: (101.3, 76.1)
Pitch angle: 0.00
Yaw angle: 158.23
Roll angle: 0.00
Is near objective: No
Is picking up:No

Figure 1. NASA Search and Sample Challenge, 2016

Abstract

A Mars rover's ability to safely conduct free-space mapping of unknown Martian terrain with undrivable areas and obstacles is essential for improving autonomous exploration robustness and extending the rover's life-expectancy. This project proposes using imitation learning to train the rover to traverse in a simulated Mars environment. 2 different sets of expert data, including actions and corresponding images, which are front camera images and images that went through perspective transformation and color thresholding, are collected for training the rover. The transformed image is an optimization of the front camera image, showing the amount of free space within the camera angles. This project has 2 goals: first, allowing the rover to traverse autonomously by following the mountain walls; second, implementing obstacles avoidance or end of the route collision by steering. Our experimental result shows that, despite the difficulty in distinguishing between wall obstructions and end of the route when making the steering decision, the rover can still manage to complete wall following, obstacle and collision avoidance in the Mars simulator for over 80 percent of the time.

Introduction

Large parts of Martian terrain haven't been explored and several past rover missions were concluded because rovers were immobilized in undrivable terrains. In order to navigate in the past, rovers were either given manual commands with a latency of more than 13 minutes (that's a ping of ~780,000!), or in more recent times, driven autonomously using a collection of sensors. As autonomous driving methods improve here back on Earth, using neural nets and using more powerful computer vision techniques have found their place at the top. Many of these implementations include identifying obstacles and un-passable terrain from camera images in order to determine the best course of action. Therefore, by essentially mapping the freespace area using computer vision techniques and training the rover to classify and identify un-drivable terrains and obstacles, we can help the rover navigate its surroundings safely, potentially preventing a rover's demise before it is all but guaranteed.

Related Works

In the Udacity Mars Rover Challenge, monocular image analysis is the most essential component for completing the autonomous navigation goals. The preprocessing procedure is implemented in 2 steps: first, implementing monocular image perspective transformation; second, adding occupancy grid. After proper scene understanding, 2 different decision-making processes have been implemented in the past: first, the finite state machine; second, path-planning algorithms, such as A-star algorithms and Djikstra's algorithms. [2]

For the preprocessing procedure, implementing the perspective transformation of the images allows the Rover to obtain a Top Down view of the Rover front camera image. [1] Then, adding an occupancy grid on this top down view image allows the rover to obtain the distance information for navigable terrain and obstacles. [4]

With all necessary information obtained, the finite state machine, with a specific set of states, including wall following state, wall finding state, wal/obstacle avoidance state, getting unstuck state, parking state, is implemented to ensure fluent autonomous traversal. However, the limited state transitions may fail if it encounters new scenarios, such as areas without walls, or areas with more wall obstructions. In addition, this method keeps the rover at a specific distance from the wall and doesn't allow the rover to explore other areas of interest.

The path-planning algorithms can also be used to plan autonomous traversal, such as Unicycle models; however, it works best when the rover has already known the obstacle positions on the map ahead of time and where exactly it will be traversing to. Therefore, it doesn't work well with unknown obstacles, and it also takes time for the rover to process the information and adjust its course to the final destination, slowing down the autonomous navigation speed. [6]

Therefore, a good autonomous navigation method should be able to combine the detailed scene understanding, which uses the analysis of a front camera monocular image or perspective transformed image, with a proper learning agent that can adapt to various scenarios without failing and can respond to its surroundings and take the next actions immediately.

Methods

Udacity Mars Rover Simulator

We will use Unity 3D to simulate a sample environment on Mars. The simulation is based on an existing Mars Search Robot Project developed for NASA Search and Sample-Return challenge (2016). In order to implement this environment, we first download miniconda in order to be consistent with the package versions used when the simulator was first released. We use an *environment.yml* template provided in order to create and activate the RoboND environment required to develop and test with the simulator.

The way the simulator works is that the assets in Unity used to create the simulator are connected to a Flask server using Socket.io. Every step through the environment, the python files used to drive the rover will emit messages through the Flask server in order to communicate with the objects in the simulator. These messages include commands and processed data, both of which are sent to and from the rover in order to aid the rover in autonomous traversal.

Inside the simulator there are two different modes, a training mode and an autonomous mode. The training mode allows the user to record observations and does not set up the socket connection between the python code and the simulator. In order to test any scripts that aid in navigation on the rover, we use the other mode, called autonomous mode, which sets up the connections between the simulator and the server and allows us to use methods and properties of the rover object in the simulator in order to make decisions when navigating the terrain.

Creating Expert Data

The first step in any imitation learning implementation is the creation of expert data. This expert data is used to train our neural network and used as reference by our rover when it is trying to make decisions in the field.

Creating this expert data can prove to be very difficult as it is a key component in the success of our rover's traversal and it is a very fickle process. For our implementation specifically, we have a "Training Mode" implemented into our simulator which will record data every frame/ step through the environment. While recording, we have the rover run through some planned course including numerous different navigation challenges commonly seen in our mars terrain simulation. This includes safely traversing valleys/ paths, avoiding collision with rocks, walls, and maintaining a safe velocity and control while traversing. All these goals include very similar input but call for different output of actions. This is a problem which we will elaborate on in the Optimization Motives section.

This data is sent to a CSV file with records of correlated actions and observations called *robot_log.csv.* We then use the python package Pandas in order to parse and organize the actions data from the CSV and the python package Pillows in order to convert JPEG camera images into numpy arrays.

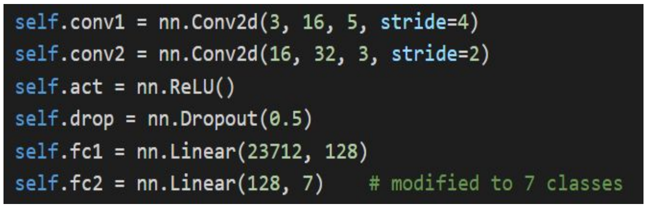
Training Neural Network

Once we have obtained expert data, we use this collection of images and corresponding actions in order to train a convolutional neural network (CNN). The task that this CNN is to take in the experimental camera image from our rover and return the probability of the rover's current state based on 7 pre-ordained classes of action. For our specific implementation, we chose to have 7 different classes corresponding to unique combinations of rover properties for the front camera images as our observations:

Class	Action [steer, throttle, brake]
throttle	[steer = 0, throttle >0, brake = 0]
brake	[steer = 0, throttle = 0, brake >0]
left	[steer >0, throttle = 0, brake = 0]
right	[steer <0, throttle = 0, brake = 0]
throttle left	[steer >0, throttle >0, brake = 0]
throttle right	[steer <0, throttle >0, brake = 0]
keep	[steer <0, throttle >0, brake = 0]

Table 1. Seven Imitation Learning Classes for Front Camera Images

The architecture of our network is the same and is as follows:



```
self.conv1 = nn.Conv2d(3, 16, 5, stride=4)
self.conv2 = nn.Conv2d(16, 32, 3, stride=2)
self.act = nn.ReLU()
self.drop = nn.Dropout(0.5)
self.fc1 = nn.Linear(23712, 128)
self.fc2 = nn.Linear(128, 7)    # modified to 7 classes
```

Figure 2. EasyNet Neural Network Layers

We keep the architecture relatively simple in order to have a faster training speed, so we used EasyNet. This is so that we can train many different models in our given time so that we can experiment with different methods while recording expert data. The expert data has more control in the performance of the rover so we prioritize perfecting that instead of having a more complex CNN.

Imitation Learning Procedure

Below is a graphic of how data traverses our pipeline and is used to make decisions in our experimental runs:

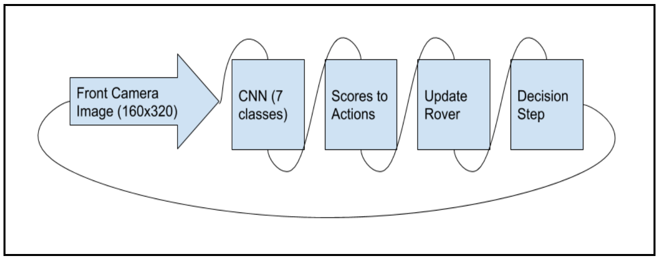


Figure 3. Front Camera Images Rover Traversal Steps

From Figure 3 & 4, we can see the 2 rover traversal steps. When the rover starts its experimental run, it will supply a stream of images and each image is passed through the trained network. The networks returns a score/probability of which class the rover is in and then returns whichever class has the highest score/probability. This score is interpreted into a set of actions and the rover is given a set of instructions based on how the expert reacted in a similar scenario.

Transformed Image Imitation Learning Optimization Method

There are 4 procedures to go through in order to get the useful transformed top down view images for training and for rover's autonomous traversing.

throttle	[steer = 0, throttle >0, brake = 0]
throttle left	[steer >0, throttle >0, brake = 0]
throttle right	[steer <0, throttle >0, brake = 0]
keep	[steer <0, throttle >0, brake = 0]

Table 2. Four Imitation Learning Classes for Images with Perspective Transformation and Color Thresholding

First, we re-designed the neural network in rovernet.py file so that it only trains on 4 action classes: throttle_left, throttle_right, throttle, keep(no action), in table 2 above.

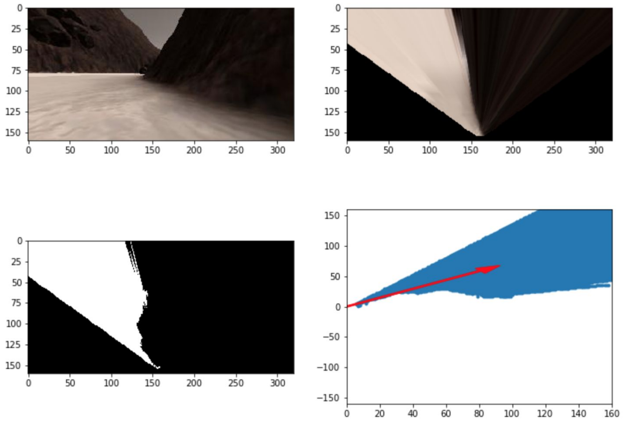


Figure 4. Perspective Transformation (top right) & Color Thresholding (bottom left) of the Front Camera Images [1]

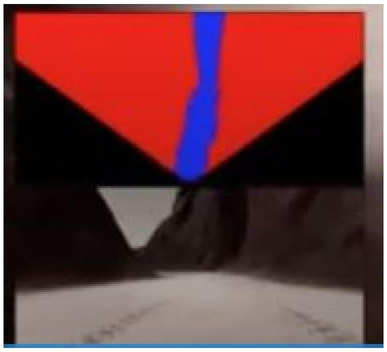


Figure 5. Final Output after Image Transformation

Second, we modified the roverimitations.py file so that the agent is trained on the front camera images that have gone through perspective transformation and color thresholding. We utilized some of the functions supporting_functions.py provided by the Udacity Rover challenge and modified it with hw1 imitation evaluate code to transform these images.[1] Figure 4 shows the examples of these images' transformation for us to obtain useful freespace information to train on. For us to visualize the terrain better, we add red color the obstacles and blue color to the freespace shown in Figure 5.

Third, we modified the preprocecssing.py file so that we could shuffle and randomly drop different amounts of collected datasets to balance the keep, throttle_left and throttle_right classes.

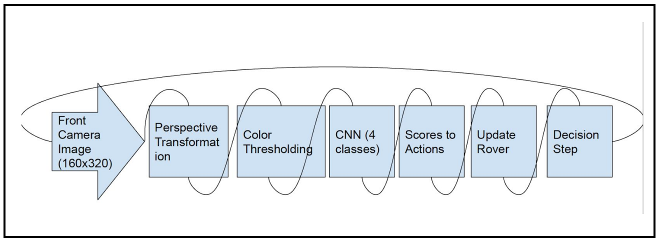


Figure 6. Optimization - Images with Perspective Transformation and Color Thresholding Rover Traversal Steps

Fourth, we modified the decision.py file so that the rover, when making a decision, fed the trained agent with the similarly transformed images shown in Figure 5. When driving the rover, it utilizes the trained agent following the steps in Figure 6.

Results

Initial Model

In order to create expert data, we initially started recording with the following goal: Traverse a pre-planned course without hitting an obstacle or colliding with a wall. After taking several laps of our pre-ordained path, we trained our network and found that our rover did not perform as smoothly as we thought it would.

The initial model did not successfully do any of the goals we set for it and it actually just kept throttling and never doing any other different actions. In addition, the loss plot (see appendix) further proved to us that the model did not actually learn anything. The loss never converged and was highly irregular as it also started at a super low value.

In order to fix these errors, we decided to perform a series of optimizations to the network and experiment in order to obtain a better performance from our model. These included data preprocessing, changing the expert data, splitting up the training goal into different subtasks and training on front camera images that went through perspective transform and color thresholding.

Optimization Motives

Upon seeing the behavior in our initial model, we decided to take a closer look at the expert data our model was being trained upon. So, we created an analysis script (roveranalyze.py) in order to see what the distribution of classes our expert data contained. One thing we noticed as that the majority of our data was all pure throttle data. This follows our results as the most commonly returned set of actions was throttle and the rover had a lot of trouble ever classifying an observation as anything but that. In order to combat this, we decided to pre-process our expert data before we trained our network. This took the form of a data reduction function in our preprocessing script (roverpreprocessing.py) where we cut some portion of the data which we find is in excess. In this case, it was throttle data in the original model.

In addition, we also realized that because we were trying to train our model to do multiple different tasks and handle those situations differently, we needed to organize our data in such a way that the model would not just get confused by the multiple tasks it was asked to solve. So, we broke the goals we wanted into different modules and planned on creating separate expert data for both situations and combining them after we found success in them individually. The different modules we decided to create were an obstacle detection module and a wall/path following module.

Another optimization we chose to do was a change in the input we were getting from the rover simulation. We noticed that the front camera image was not the best angle to look at our data in order to make decisions such as turning when there is a wall in front of you. Since we had no access to additional hardware assets such as distance sensors or other cameras, we decided to do a perspective transform and color thresholding of our front camera data into a top-down view which was able to tell if there was free space in front of the rover.

Optimization Results

Data Reduction/Balancing

The outcome of our data reduction scripts were all positive and the changes this preprocessing step did on our models were clear as night and day. In our initial model, we originally had a rover which would only throttle and never turn, brake, or go into any other class of action. Our analysis script showed that throttle data was the majority category of data by more than 200% as we had 898 instances of throttle data. The second most common class, keep, only had 400 instances. After training a model on our data with a 50% reduction in throttle data, we began to see that the rover started to go into other classes of action and no longer was always stuck in the throttle class.

Because the outcome to this step was so successful, we utilize this script with all of our models using the front camera image after the initial model. The balance of data is very important so that the scenario where a single class of data drowns out all the training we do to teach the rover to go into other classes of action never happens.

Obstacle Avoidance

The first module which we decided to split our overall goal into was the idea of having our rover never collide into objects. This is a key feature that is necessary for a mars rover to have as any collision could be fatal on a desolate planet with nothing there to help you.

When recording the expert data for this module's model, we chose to approach a rock in the center of our environment and just keep approaching the rock and turning when we got to a certain proximity of the rock. We tried to start turning when the rock got to the size of at least half the camera image's height. In order to keep our training robust, we would approach the same rock from 5 different angles to ensure that any face of the rock would be able to be classified as an object. Our most optimal model was trained on 2171 images after data reductions. The original dataset was probably ~3k images of size 160x320. The balance of data for the module leaned towards having a majority of turn and brake data. Throttle data was seen in very low numbers due to the static nature the rover would take while actively avoiding obstacles.

Immediately after training, we could tell that our simplification and splitting of tasks affected the learning that our model was capable of doing. Just from looking at the loss plot (see appendix), we could see that the loss was now behaving correctly and stabilizing after a few epochs.

While testing this model, we found that the rover had great success in classifying objects in its camera view and to be able to avoid them. Upon seeing a rock in the view of the camera, the rover would brake and start rotating to the right until the rock was not in the front camera image anymore. Unfortunately, the rover would have trouble traversing normally as the expert data was just based on avoiding the rocks instead of actually traversing the environment. In addition, the influx of brake and turn data outweighed the throttle data in our expert captures. This made it so that throttling was very rare and would only occur in small amounts. Either way, brake data would always come into play more often and it would only take a few steps of brake actions in order to cancel out numerous environment steps of throttle.

In addition, sometimes the coloring of the rocks in the foreground of our camera image would match the color of the walls in the background. This would make classifying the rock from the wall difficult from our single image and would sometimes let the rover not see that the rock was indeed in the foreground and not part of the background wall. We talk about how we overcome this issue in our transformed image imitation learning optimization experiment.

Wall/Path Following

After achieving the goal of obstacle avoidance, we needed the rover to be able to traverse safely in general in order to minimize the situations which had a change to end in collision or a need to do obstacle avoidance.

When recording the expert data for this module's model, we would set a specific course in the environment where the amount of turns and straight shots were balanced so that our expert data in turn would also be balanced. We would train by lapping this course 4-5 times in order to get a dataset of 2572 images after data processing. Similarly to the previous module, the data beforehand was ~3k images of 160x320. The majority of this data was in the throttle or keep class as most of the path following involves the rover moving along the path at a controlled velocity. Adjustment classes such as throttle left or throttle right were all equally balanced and made up a small portion of the data. Our data reductions focused on reducing throttle and keep data in order to have the rover learn how to turn and not get drowned out by all the traversal data.

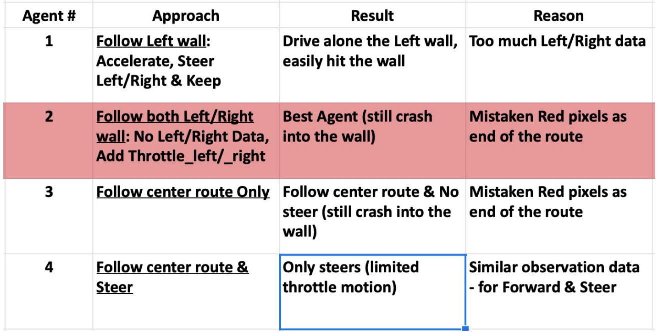
Upon testing our model, we found success in reaching the goal set for this model. The rover's behavior would be ambiguous when just travelling in the middle of an open area but once it saw a wall in proximity of the camera angle, it would "latch" onto that wall and start approaching it and using it as reference as a path. Once it caught a wall it would use a reference, it could be observed that the rover would throttle along the wall and adjust accordingly to the curvature of the wall. In situations where the wall had sharp curves, the rover would usually make the correct decision and would come to a complete stop and rotate till it was aligned with the next section of wall. However, there were situations where the rover would have too much initial velocity to make the turn without colliding with the wall first. An example of this situation is if the rover has a straight shot for a long distance and then has a sharp turn ~90 degrees. This is not deemed a failure in our goal though due to the controlled speed that we train our rover to take. The speed the rover traverses is around 1-2 mph. In more gradual turns, the rover would have the most success as small adjustments were no problem for the rover.

Recombination of Modules

When combining the two expert datasets and training a new model to combine our individual tasks, we found that the performance was not satisfactory and did not achieve the goals of the two different modules combined. In fact, the ability to perform either task was lost and the rover started behaving just like our initial model did once again. When given the obstacle detection course, the rover did not classify obstacles and just got stuck in the brake class while in front of a rock. In the path following course, the rover would move a little and then eventually get stuck in the keep or brake class once again before it had the chance to traverse or catch a wall to follow.

We think that this happened because of several reasons. One of those reasons is because the data balance we had achieved individually from the modules were once again altered and it showed from our rover being stuck in the same couple of classes. We performed additional data processing in order to balance this once again but it still did not get unstuck from those classes. Another reason we think this happened is because the balance between the classes must be too different in order to combine. The make up for the optimal performing individual modules had drastically different data balances and changing it to be a combination of the two modules made it so that neither of the tasks could be completed or learned from the rover's network. We talk about possible fixes in the Future Works section later.

Transformed Image Rover Traversal



Agent #	Approach	Result	Reason
1	Follow Left wall: Accelerate, Steer Left/Right & Keep	Drive alone the Left wall, easily hit the wall	Too much Left/Right data
2	Follow both Left/Right wall: No Left/Right Data, Add Throttle_left/_right	Best Agent (still crash into the wall)	Mistaken Red pixels as end of the route
3	Follow center route Only	Follow center route & No steer (still crash into the wall)	Mistaken Red pixels as end of the route
4	Follow center route & Steer	Only steers (limited throttle motion)	Similar observation data - for Forward & Steer

Figure 7. Representative Trained Agents using Transformed Front Camera Images

We collected around 15 groups of datasets and trained around 16 agents. Agents are trained on both 7 classes and 4 classes. Different groups of datasets are collected using distinct or slightly modified strategies, different number of classes and are trained with different percentages of data dropping rate. Figure 7 presents the best model model and other representative agents' data collection strategies, results and reasons, we will also explain them in more detail below.

In the beginning of the datasets collection, 7 action classes are used and several different agents are trained using strategies, such as wall following, centerline following and so on; however, the trained agent performs badly.

For instance, Figure 7 agent 1 includes a small amount of data for throttle_left & throttle_right and lots of left and right data classes, though it drives along the left wall fine but easily crashes into the wall; and the reason is that there is too much left/right data. For agent 3 and 4, if we collected data for the rover to traverse only the center line, it has a hard time recognizing the side walls and crush into those walls often; however, if we add left/right or throttle_left/throttle_right data with the original centerline traversal data, the rover usually keep steers left/right if 7 classes is implemented or throttle_left/throttle_right if 4 classes is implemented.

Therefore, For this optimization procedure, we eventually settle for training on 4 action classes, which are throttle_left, throttle_right, throttle and keep(no action), shown in table 2. The reason is to avoid confusing the rover; if we have additional data on steering left and right action classes, the rover often just steers indefinitely without moving forward when it encounters walls/obstacles ahead. We also reduce the value for our steering control in our "scores_to_action" function in rovernet.py so that the rover does not oversteer and misses the window to throttle into freespace. In addition, we eliminated the brake action class so that the rover will not brake when it is supposed to move forward or steer left/right; since there is probably for the rover to brake or throttle with a similar transformed image, sometimes the rover doesn't move at all. However, we did save the keep action class; in fact, we balanced the keep, throttle_left, throttle_right data so that each has the similar amount of data, allowing the rover to maintain a lower speed and slow down significantly when it reaches end of the route or obstacles without needing to brake.

Following vaiours strategies, we were able to train 3 satisfactory agents. Our best trained agent 2 in Figure 7 chooses to train by following both the left/right wall as well as steering at the end of the route with 4 action classes described above. Agent 2 is capable of throttling forward, constant steering adjustment, traversing on both the original training route and new routes by following the left and right wall, steering at the end of the route. Our trained agent is able to freely explore the route on the new test route as well. However, the agent has a hard time dealing with is distinguishing between the end of the route and the wall obstructions, and the reason is that the transformed images exhibits mostly red pixels when encountering both situations, thus making it impossible for the rover to distinguish between those two, and making false decision to steer. In addition, the wall following performance when traversing the left route is not perfect, the agent sometimes needs human intervention and drives a little bumpy.

Another discovery during training is that the best data size is between 1000 to 2500 when it comes to training a good agent, along with balanced data for throttle_left, throttle_right and keep(no action) action classes. For our best agent, we utilized 1374 datasets of quality data as our expert data.

Conclusion

To conclude, we successfully conducted 2 different image training using imitation learning: the front camera image training, and the perspective transformed and color thresholded front camera image training.

For the front camera image, the modularization of the wall following and obstacle avoidance tasks drastically improve agent performance so that the rover can traverse following the left wall and turn right when encountering obstacles, and data reduction/balancing improves expert data quality. However, the combination of 2 datasets with different tasks introduces problems in terms of data balance and confuses the rover, preventing it from executing any expected task.

We also optimize the rover training by implementing the perspective transformed and color thresholded front camera image as our observation. Our final best agent is trained on a smaller dataset and smaller number of classes. The trained agent still has a hard time distinguishing between the end of the route and the wall obstructions and the rover sometimes stuck on the side of the wall because the transformed images look similar in both cases and exhibit solely red pixels. However, we were able to successfully train an agent that can traverse following the left or right wall with steady low speed, turn at the end of the wall and adjust steering so that it can reduce the possibility of stucking beside a wall. This agent also generalizes well on both its original training route and new test routes that it has never been on before.

Future Works

There are 4 potential future implementation and optimization strategies.

First, implementation of stereo camera and stixel world can help the agent discover side wall obstructions and plan an ideal exploratory path within the free space.[5] The stixel method can help prevent misclassification of actions when the rover reaches the end of a route or encountered side wall obstructions and avoid crashing into the wall.

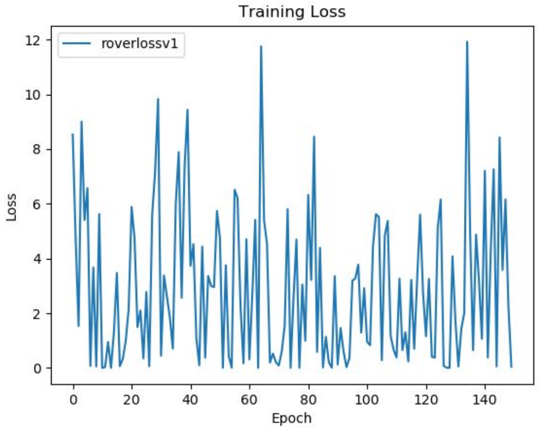
Second, alternative learning methods, such as reinforcement learning (deep q learning), can be used in this simulation to train the rover by exploring the environment.

Third, combining the modular implementation with imitation learning so that the rover can switch between different tasks more easily. For example, introducing different states for the agent to switch from so that the rover can use different trained agents with different training weights for various tasks.

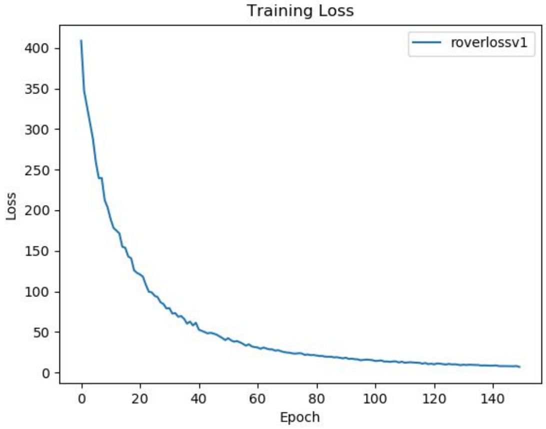
Fourth, the simulation can also be modified to introduce real Mars terrain patterns and the rover can be trained to distinguish between different terrain types to determine which terrain is safe to traverse and which terrain is dangerous.

Appendix

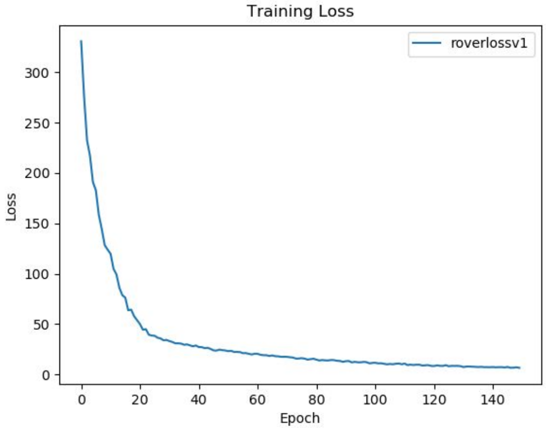
I. Training Outputs



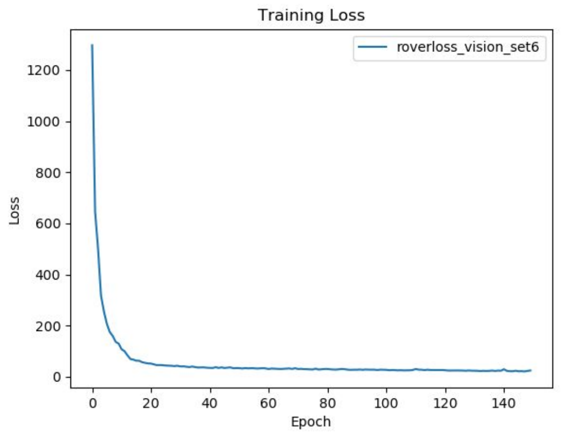
Initial Model Loss Plot



Obstacle Detection Loss Plot



Wall/Path Following Module Loss Plot



Front Camera Image with Perspective Transformation and Color Thresholding Loss Plot (stabilizes at around 13)

II. Sample Outputs GIFs & Videos

Front Camera Rover Traversal GIFs

- GIFs of performance can be found in the presentation slides 10 & 11 (GIFs do not load on PDF)
- Presentation slides link: https://docs.google.com/presentation/d/1nsBOyPENV18Kdcy4F2FEK5CfuT1mkPPBEbo74Y_DHa4/edit?usp=sharing

Transformed Image Rover Traversal Videos (youtube links):

- Following right wall (original training route): https://youtu.be/zkZdn1vhk5M
- Following right wall & turn at the end (new test route): https://youtu.be/FLxTmbcKRXk
- Following left wall (original training route) (need human intervention during traversal): https://youtu.be/6yhUoRV45ks

III. Work Distribution

Member	Tasks
Brian Xu	- Implement rovernet.py + rovertraining.py - Implement roverpreprocessing.py and decision.py - Implement Front Camera Image Wall-following experiment + optimizations - Created Presentation slides (1-12, 18, 20) - Write Report (Introduction, Methods, Front Camera Results, Appendix)
Hairuo Sun	- Implement 2 roverimitations.py files (front camera & transformed images optimization) - Implement decison.py + analysis.py - Modify rovernet.py (optimization) - Implement Front Camera Image Obstacle Avoidance Experiment - Implement optimizations: front camera image perspective transforms and color thresholding (perception.py) - Implement Transformed Image Wall-following & End route Steering Experiment - Complete Presentation slides (13 - 17, 19, 20) - Write Report (Abstract, Related Works, Transformed Image Imitation Learning Optimization Method & Result, Conclusion, Future Works, Appendix)

References

[1] “Robot Search and Sample” *Github.com*, 4th October 2020, https://github.com/ryan-keenan/RoboND-Python-Starterkit

[2] “Mars Search Robot” *Github.com*, 4th October 2020,https://github.com/Salman-H/mars-search-robot

[3] Chapter 9 of Probabilistic Robotics Book, by S. Thrun, W. Burgard, D. Fox

[4] Free Space Computation Using Stochastic Occupancy Grids and Dynamic Programming In Workshop Dynamical Vision ICCV 2007, by H. Badino, U. Franke and R. Mester

[5] The Stixel World - A Compact Medium Level Representation of the 3D-World DAGM 2009, by H. Badino, U. Franke and D. Pfeiffer

[6] Technical report on Optimization-Based Bearing-Only Visual Homing with Applications to a 2-D Unicycle Model, by R. Tron and K. Daniilidis.