Senior Research Report

Manav Gagvani

May 29, 2024

Introduction

When I started out on this project, I intended to extend previous reinforcement learning processes for self-driving cars by executing them in real life on a physical system. To do this, I divided the project into 3 stages:

1. Developing Reinforcement Learning algorithms in simulation
2. Builiding the physical car and executing basic algorithms
3. Implementing reinforcement learning on the real car

I was able to successfully complete steps 1 and 2, although there is room for improvement on both. Thus, since the majority of the non-RL work has been done, such as assembling the car, future work can revolve around step 3. (Consider the inverted pendulum project, which had 3 people and a mechanism assembled from day one. This is what a future group who intends to work with the car should aspire to emulate).

Prior Approaches

Although my project revolved around reinforcement learning, this was done with prior knowledge of previous approaches to the self-driving problem. One popular approach which has previously been applied to miniature self-driving cars is supervised learning through a CNN (convolutional neural network). This approach originated with Nvidia's PilotNet CNN. At its core, the PilotNet CNN takes in front-facing images of a road, and outputs both the desired steering and throttle for the car to stay on the road. A human driver manually collects training data by driving a car, collecting road images along with canonically correct steering and throttle videos. Although this is not a good replica of how real self-driving cars operate, it is a good emulation of the desired behavior, and ties in quite closely with the development of the autoencoder I will describe later.

Stage 1: Reinforcement Learning in Simulation

Basic Reinforcement Learning with PPO

The first step of my project was to implement basic reinforcement learning using the

well-known PPO (Proximal Policy Optimization) algorithm. This was done through a simulator known as the DK Simulator which is used to simulate 1/10 scale self-driving cars. It implements the kinematic bicycle model and exposes camera images (front-facing, from the perspective of the car), LiDAR measurements, and spatial information (velocity, position, collisions) through the standard Gym API.

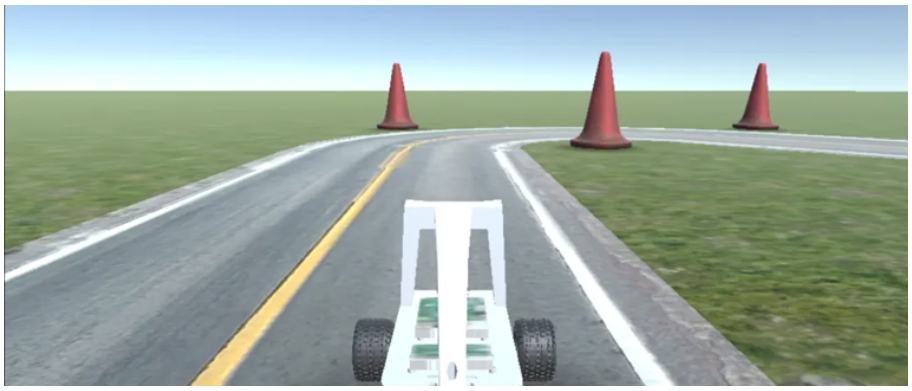


Figure 0.1: Screenshot of the DK Simulator. Vehicles can be customized to better represent their real-world counterparts.

I used Stable-Baselines 3 which is a community-maintained set of reinforcement learning algorithms implemented in PyTorch. Stable-Baselines made creating a proof-of concept for reinforcement learning extremely simple. Here is the code I used:

```python
import gymnasium as gym
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv, SubprocVecEnv
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.callbacks import CheckpointCallback
import f110_gym
import tqdm.rich as tqdm

# Define the Gym environment
env_id = 'donkey-warren-track-v0'
env = gym.make(env_id, render_mode='human_fast')

# Define the MLP policy
policy = 'MultiInputPolicy'

# Create the PPO agent
model = PPO(policy, env, verbose=1)

env.reset()
env.render()

# set up model in learning mode with goal number of timesteps to complete
model.learn(total_timesteps=10000)
obs = env.reset()
for i in tqdm.tqdm(range(10000)):
    action, _states = model.predict(obs, deterministic=True)
    obs, reward, done, info = env.step(action)
    try:
        env.render()
    except Exception as e:
        print(e)
        print("failure in render, continuing...")
    if done:
        obs = env.reset()
    if i % 100 == 0:
        print("saving...")
        model.save("sb3_mlp_policy_dk")


# Evaluate the agent
mean_reward, std_reward = evaluate_policy(model, env, n_eval_episodes=10)

# Save the agent
model.save("sb3_mlp_policy_f110")
```

Figure 0.2: Code used with Stable-Baselines 3 to implement PPO in simulation. Stable-Baselines 3 simplified the code significantly.

This code takes in front-facing camera images as input which are evaluated by the policy, which is a CNN. The two outputs are steering and throttle for the car. The reinforcement learning terminates when there is a collision or the car goes too far from the center of the road.

After about 2 days of continuous training on a GPU-accelerated machine (GTX 1070 GPU), the car was able to successfully navigate the entire mountain track. However, The simulation was already sped up 4x, thus resulting in what was effectively 8 days of training. The car was able to navigate the entire track in 21-23 seconds. However, it would occasionally crash, and it was extremely wobbly. This is because PPO has no "short-term memory": it's acting on whatever input it receives in the current frame, with no information being incorporated from preceding frames. Thus, I looked for ways to improve the reinforcement learning process such that it could be applied to a real car.

Incorporating a Variational Autoencoder & the TQC Algorithm

The idea of the autoencoder is that it compresses an image of the road into a small latent vector which is then fed into the reinforcement learning algorithm through a wrapper class. This is needed because using the entire image of a road is quite inefficient. There are millions of different road images that are all turning to the left. In all of these cases, the car should do the same thing, even though the road looks different. Compressing this information into only the parts that are relevant for a car to navigate is the goal of the autoencoder. However, before it can be used in RL it needs to be trained with a corresponding decoder. I started by collecting data manually by controlling the car with a joystick. The original implementation of the data recording used the arrow keys, but I wanted more control over my dataset, so I added the ability to control the car with a joystick while recording data. The resulting dataset consists of 976 pictures, but not the corresponding throttle/steering values. Here are the relevant details of the autoencoder: The encoder and decoder consist of 4 convolutional/transpose convolution layers with ReLU activations, except for the last layer of the decoder, which uses sigmoid. The input images are of size (160, 120, 3) which are cropped to (160, 80, 3) because the top 1/3 of any given picture doesn't contain anything very useful:

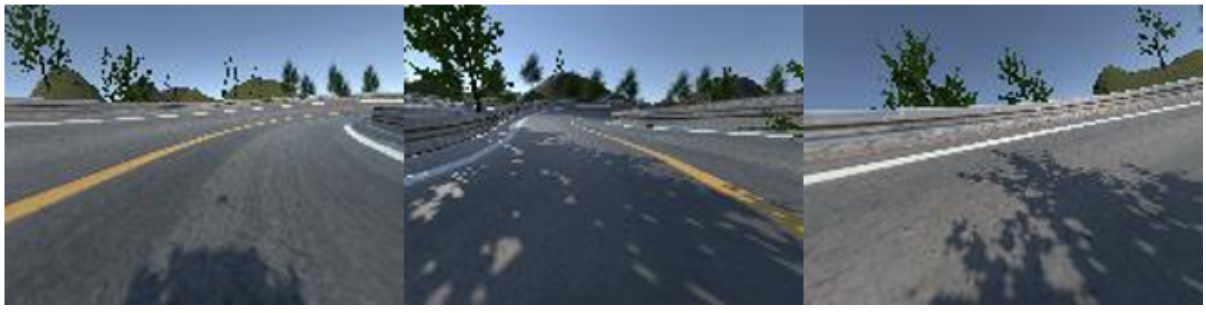


Figure 0.3: Road images from the simulator from manually driving the car. Note that the top portion of the images shows sky and trees, information that is irrelevant to the car's navigation.

In order to improve the quality of the reconstruction and create a more robust autoencoder, the data is augmented through random left-right flipping, random shadows,

Gaussian blurring, motion blurring, random pixel jittering, and cutouts (small rectangular portions of the image are replaced by the average of those pixels). The reconstructed images contain the yellow and white lines, but they don't generally have shadows or texture on the road due to the augmentations. Thus, they only contain the relevant information for the car to navigate on the road. Running on n=500 samples, the autoencoder runs at 140 Hz on an Nvidia GPU (RTX 3060). This means it should run above 20 FPS on the Nvidia Jetson Orin Nano since the auto-encoder is a relatively small neural network.

The next step was to create a wrapper around the environment which alters the observation space from an image to the latent vector generated by the encoder. The speed of the vehicle is also concatenated to the end of the latent vector, creating an observation space of shape (33,) in the wrapper. The policy was changed from a CNN to a simple fully-connected network with two layers of 256 neurons each – a significant advantage of preprocessing the images with the autoencoder. I was then able to run the reinforcement learning training using the autoencoder to pre-process the images taken by the car. This resulted in improved performance versus PPO using raw images as input. Lap times went down from roughly 21 seconds to 15.5 seconds, and mean time before a crash went from slightly over 2 minutes to 15 minutes:

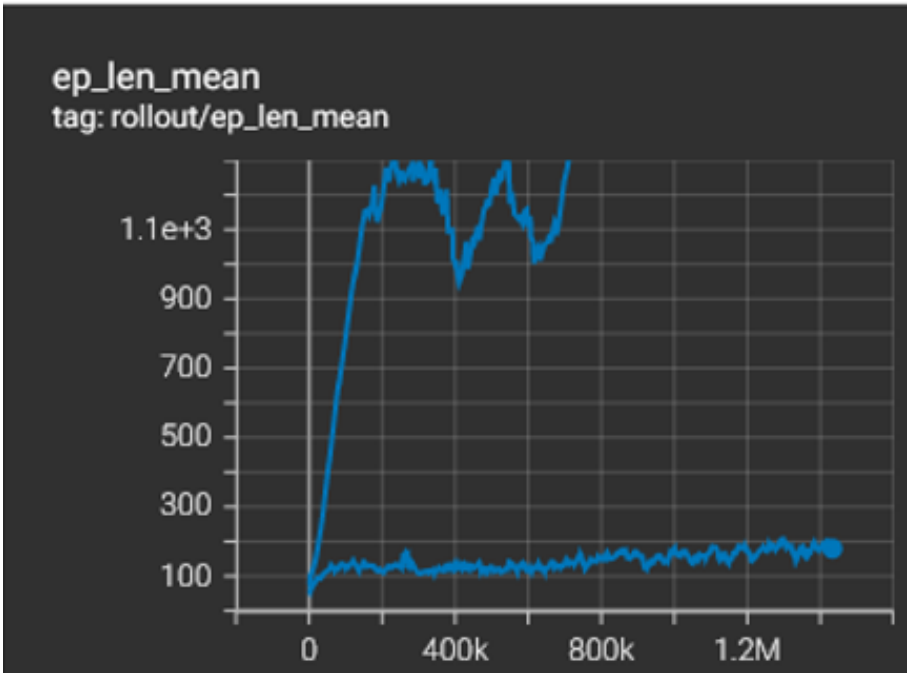


Figure 0.4: Results of training with PPO with and without the autoencoder. The bottom line is mean time before crash (episode length) without the autoencoder and the top line is the mean time before crash with the autoencoder.

I would recommend that future groups who want to work on the self-driving problem with reinforcement learning in simulation use the CARLA simulator which has a much more full-featured set of graphics, pedestrian interactions, and other useful features applicable to the real world.

Possible Implementations of Reinforcement Learning in Real Life

There are two key considerations for implementing the reinforcement learning process:

1. *Determining when an episode of training is over* - Both Dr. Gabor and I agreed we would have a track with two sets of parallel lines, most likely colored duct tape, on the ground. The goal of the reinforcement learning is to stay in between the lines. Thus, it is important to know when the car has gone off the track since it is no longer collecting useful information. The solution to this is to implement code which determines whether the boundary lines are visible on the ground. If they are not, we assume we have exited the track, and thus need to reset.

2. *Resetting the car back to a good position so that training can re-start* - The simplest solution is to maintain a "replay buffer" of previous throttle/actuation signals and then play them backwards so that the car would reverse itself into a position it was previously at. I intended to use the motor controller's RPM values in order to mitigate the problem of accumulated error, but after switching from the VESC to ESC, this was not possible to do accurately. However, there are more robust solutions. AprilTags, which are a fiducial markers, could be used as a possible solution. Given a calibrated camera (e.g. known focal length, image center, and distortion coefficients), which I do have in the OAK, there is readily available code to calculate the pose of an AprilTag relative to the camera. Thus, if I establish a set position for the AprilTag, I will know the pose of the camera. I can use an AprilTag, or multiple AprilTags, to re-localize in this way. By finding out the true position of the camera relative to the AprilTag, the car can navigate back to a known position within the track boundaries.

Stage 2: Building and Testing the 1/10 Scale Self-Driving Car

When I started this project, there was a half-disassembled car in the SysLab annex:

For reference, a group in 2016, Grey Golla and Charles Zhao, originally built and tested the car. From what I know, their intention was to implement the Artificial Potential Fields (APFs) path-planning algorithm on a car. I would not recommend future groups pursue this topic because cars are not *holonomic*: in essence, they cannot move in any direction equally well, while APFs are essentially implementing gradient descent in a contrived real-world scenario.

I disassembled the car, which already was stripped of many of its components, and started rebuilding it from the ground up. My goal was to replicate F1Tenth, an open-source design created by a group of universities for research purposes. However, given the enormous cost of some of the parts (e.g. $1500 for a LiDAR sensor), I decided to make do with some of the parts which I could already access. In the future, I would recommend not deviating from the recommended BOM (Bill of Materials) as it introduces complications in the future which take a significant amount of time to work around. Here is a table of the parts I started with at the beginning of the year:

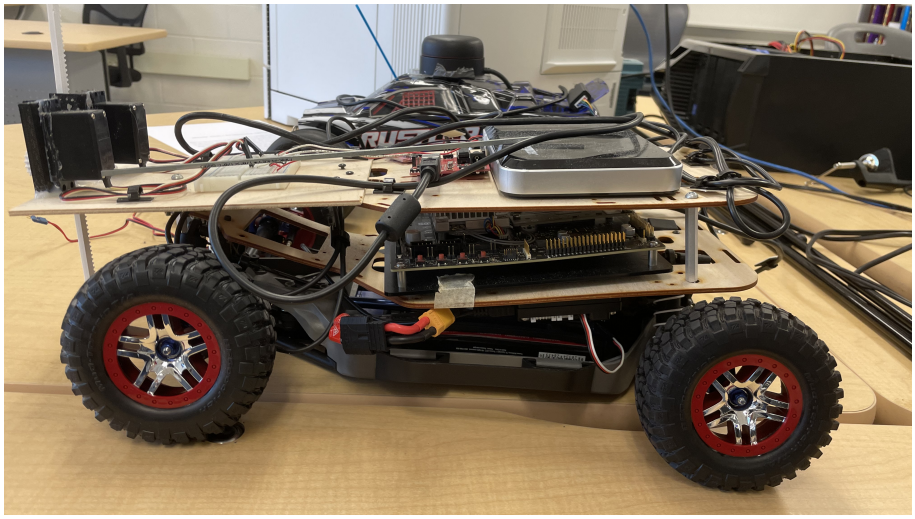


Figure 0.5: The car as I found it in the beginning of the school year.

Part	Quantity
Traxxas Slash Chassis	1
Traxxas 2S LiPo 5800 mAh	1
Traxxxas EZ-Peak Plus Charger	1
RPLidar A2	1
XPAL Power XP20000 Battery	1
Enertion FOCBOC VESC Motor Controller	1
AmazonBasics USB hub (7 port)	1
Nvidia Jetson TX1 Camera Module	1
Arduino Uno Clone Board	1
SparkFun 9DoF Razor IMU	1
Traxxas Slash Plastic Cover	1
Nvidia Jetson Nano	1
Nvidia Jetson TX1	1
Assorted wires and hardware	

Out of these, the chassis, charger, lidar, VESC, hub, Arduino, and IMU were usable. The rest were either broken, damaged, or unusable in some way. I attempted to follow the F1Tenth construction guidelines as well as possible, resulting in the following set of core components:

- Nvidia Jetson Orin Nano
- OAK-D Pro Active Stereo Vision Camera
- RPLidar A2
- 80A Brushless ESC
- PCA9685 breakout board
- Omnicharge 20 battery

(a) CAD (b) Laser-cut

Figure 0.6: First try at designing the layout of the car's mechanical assembly in Onshape and laser cutting it to mount on the car.

Note that the F1Tenth project, and others, such as MuSHR, use a VESC motor controller instead of a standard one such as the one linked above. I chose to use the normal ESC because of ease of operation with the software stack I already had in place. However, if you plan on using the standard ROS (Robot Operating System) stack for this hardware, the VESC will be much easier. I spent a few months attempting to interface with the VESC effectively using custom software. This included interfacing with the VESC over USB to the Jetson, and using an Arduino as an intermediary, where signals would be sent from the Jetson to the Arduino over USB, and then to the VESC through the servo PWM protocol (different than normal PWM). After failing to successfully control the throttle motor through the VESC, I switched to the normal ESC. The advantage of the VESC is that it detects small variations in the currents going through each pole of the brushless motor to determine the true RPM the shaft is spinning at. This is helpful when determining the true speed of the car. The VESC also has a built-in PID control loop which can be used to set a desired RPM value. It also has other telemetry, such as temperature monitoring, which is useful for detecting thermal overheating.

The first step I took was designing the mechanical assembly of the car. Since I was using non-standard components I could not use the provided laser-cutting files to mount the parts.

However, the layout in Figure 0.6 had some issues, particularly that there was no good space to mount the Nvidia Jetson board and that cable management was difficult with the USB hub. Thus, a few revisions were made until the final assembly was created, as seen in Figure 0.7.

The next step, once the car was assembled, was to do the wiring. Here is a basic summary of the wiring that was done:

- USB ports on the Omnicharge power the **power port** on the OAK Y-adapter and the **power port** on the USB hub.
- The AC plug on the Omnicharge powers the 19V power supply that is connected to the Jetson's **barrel plug**.
- Each of the 3 phase wires on the ESC were connected to the **motor**.

(a) Revised CAD (b) Assembled car

Figure 0.7: Final Onshape design and assembled car. Note the blue Jetson mount, which doesn't require any screws on the Jetson itself to be removed.

- The battery T-plug was connected to the ESC's **T-plug**.
- The 3-pin assembly of **signal, power, and ground** from the ESC is connected to the 1st set of header pins on the **PCA9685**.
- The 3-pin assembly from the **servo motor** is connected to the 2nd set of header pins on the PCA9685.
- The PCA9685 is wired to the Jetson according to the wiring guide here. Refer to the Jetson Orin Nano pinout on the specifics of the Jetson-side wiring.

Once the car was wired up and ready to go, I had to calibrate the steering and throttle. I used the "DonkeyCar" (DK) Python library, which contains utilities to work with common sensors/actuators and pull them together into an execution loop ran multiple times per second. In general, the vehicle pipeline (what is run in the execution loop), for any vehicle, can be described as follows:

1. **Observe**: Gather sensor readings (camera, lidar, IMU, etc.).
2. **Plan**: Use reinforcement learning or any other algorithm to process the data and generate motor commands.
3. **Act**: Send commands to the motors, and ensure they are doing what we expect them to do.

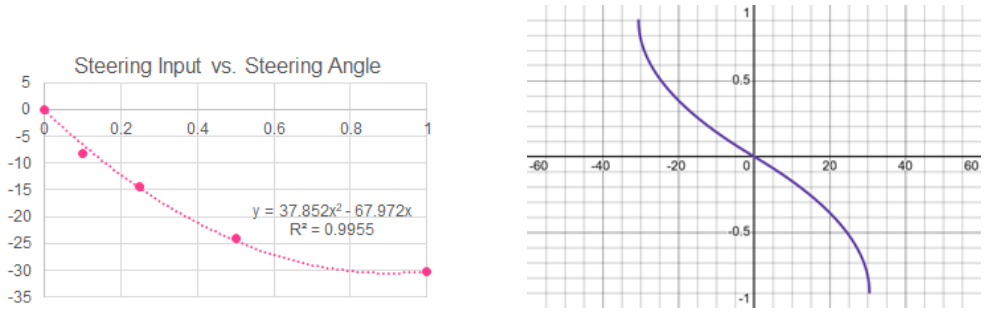
Note that DK installs TensorFlow and PyTorch in its own install. When I first installed DK onto the Jetson, the versions installed were incompatible with the version of JetPack (the preloaded binaries installed on the Ubuntu OS) and thus wouldn't be CUDA accelerated. I had to manually uninstall TensorFlow and PyTorch, look up the correct versions, and install them. I then verified that CUDA was being used for computations.

The next step was to calibrate the steering and throttle, since the car's steering servo motor and ESC driving the throttle motor are both commanded using a unitless PWM value. I had no reliable method of calibrating throttle, since I was using a normal ESC instead of the VESC, and didn't have access to calculated RPM. Furthermore, velocity would differ significantly even with the same RPM because of tire slip on different surfaces.

Thus, I calibrated steering only using the following procedure.

- Attach laser range-finder to the front wheel of the car and point it at a flat wall which is a known distance from the wheel (denoted x).
- Command the wheel to turn to a specified value (e.g. 0.1, or -0.5), and measure the distance from the rangefinder to the wall for each value (each measurement is z_v, where v is the steering value).
- Calculate the steering angle for each z_v with the formula $\theta = arccos(x/z_v)$.
- Fit a quadratic curve to the measured data. This represents the function that takes steering values as an input and returns steering angles. Compute the inverse of this function to obtain a function that accepts steering angles and returns steering values.

This is the end result for the calibration I performed:



(a) Quadratic curve fit to the measured data.

(b) Conversion from steering angle to steering value.

Figure 0.8: Results of calibration procedure.

Once I had done this, I attempted to run algorithms I had previously used in simulation that used the lidar. These included Gap Follower and Disparity Extender. However, these algorithms asked too much of the car: commanding it to turn to steering angles that were not physically possible. Although I am aware of research groups who have ran these algorithms on the exact same car as we have in the SysLab, I do not know how they achieved this. I pivoted to a simpler algorithm, wall following.

Figure 0.9: Diagram of how wall following works. There must be 2 well-defined walls for the algorithm to work as intended.

The wall following algorithm uses the LiDAR sensor for distance measurements. Two measurements, to the left and right of the car, are denoted d_l and d_r, respectively. In theory, these should be singular measurements from exactly 90 to the left and right of the zero point of the LiDAR. However, due to the slow update rate of the LiDAR mounted on the car, I calculated d_l and d_r as averages of the window of measurements in a 20-degree window (e.g 80 to 100 degrees). I then use a PID controller which outputs a steering value, to drive the error $d_l - d_r$ to zero. PID control theory is well-documented, and I found a PD controller with a minimal D coefficient could successfully navigate the car around the set of hallways behind room 200.

Acknowledgments

I would like to thank Dr. Gabor, my Senior Research teacher, and Dr. Torbert, who has been immensely helpful in buying parts needed to assemble the car. I would also like to thank Shreyan Dey and Christopher Lim for their help designing and fabricating the mechanical assembly.