Reinforcement Learning and Lane tracking

Vikram Sriram
Udacity Capstone Project
Dec 3rd, 2019

Project Overview and Problem Statement:

As computational power and amount of data increases, machine learning will become an increasingly viable option for automation. I am especially fascinated by Reinforcement Learning due to the nature of a reward function. We can exploit this human-like behavior to develop and find new solutions to problems. In the near future, reinforcement learning will play a significant role in many autonomous related tasks.

For my test environment, I chose to investigate methods to move from pointA to pointB autonomously. From factories to consumer products, the applications have great potential to enhance life. Although Lane Keep Assist technology is available in many modern cars, it is still an open problem. There are numerous edge cases that still need to be solved. A continuously learning system, such as Reinforcement Learning, could be a potential solution to these problems. I wanted to explore three different methods: Reinforcement Learning (with DDPG and DQN), OpenCV PID, and a CNN from Nvidia's paper. There are model-free reinforcement learning methods as well as the ubiquitous control scheme, PID, to tackle this problem.

Description of some tools I will be utilizing:

- CNN or convolutional neural networks are a class of deep neural networks that can take advantage of spatial information to visually analyze and extract key features from image data.
- OpenCV is a powerful open-source computer vision library used for processing images in real-time. PID or proportional-integral-derivative controller is a control loop that operates using a feedback loop.
- Reinforcement Learning is a semi-supervised learning model in which there is an agent that interacts with an environment by taking actions to maximize a specified reward function. The reward function is designed to accomplish a specific goal or task.

Reinforcement Learning often accompanies a simulation to be trained and many iterations. There also needs to be an explorative component to find the optimum action that maximizes the reward function.

Due to the many environmental variables that need to be considered, building a simulation itself can be challenging. Furthermore, transferring the model from simulation to real life doesn't always yield similar results. Rather than a simulation, I chose to train the car in real-time.

Outline of steps:

- Create a stable and modular test car.
- Develop a technique to process the image in real-time and locate ideal tracking heading as an error function. Then use this error as feedback to create a control loop.
- Implement and optimize Nvidia end-to-end CNN model.
- Implement **DDPG and DQN to train in real-time.
- Evaluate results.

Note: ** DDPG originally planned, but currently postponed due to time and budget constraints.

Metrics:

I evaluate the different approaches using lap-time as a global metric. Using a constant throttle system, the different model's five lap-times are recorded. Additionally, I control the throttle myself in an attempt to push the limits of the model for the best lap-time.

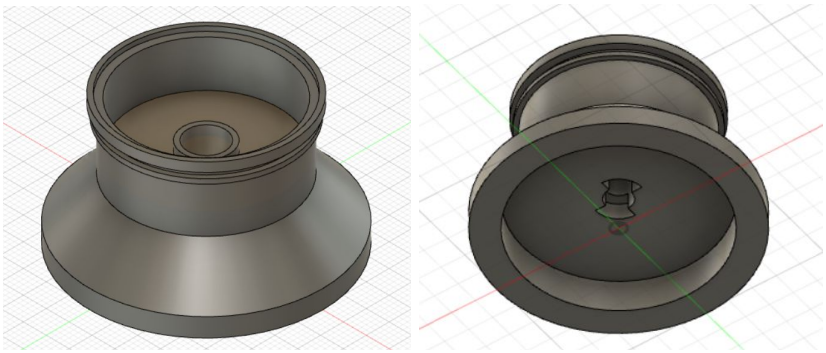
The neural net based approaches will be compared using training time and architecture details. The average training FPS and testing FPS will be recorded as well.

And finally, I judge how well the approach can generalize by altering environmental conditions such as lighting.

Building The Car: Obstacles and details

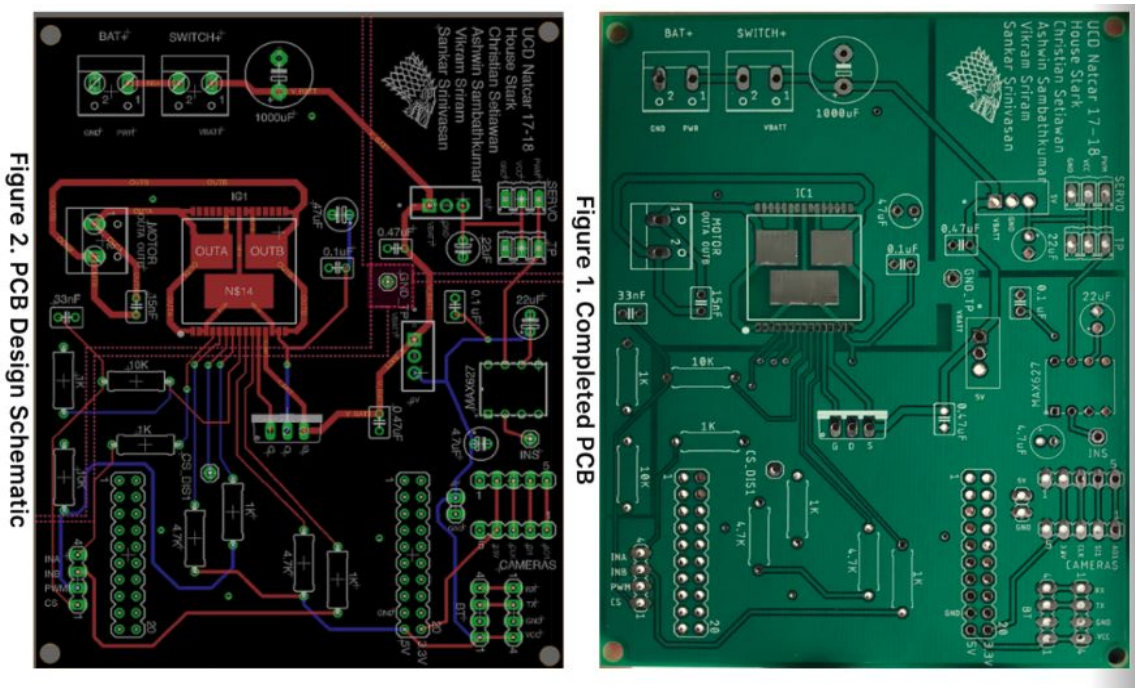
The chassis of the car was removed from a model car assembly kit.

The ground clearance decreased significantly due to the weight of the boards and stress on the suspension. Larger wheels had to be designed and fitted to accommodate this change.



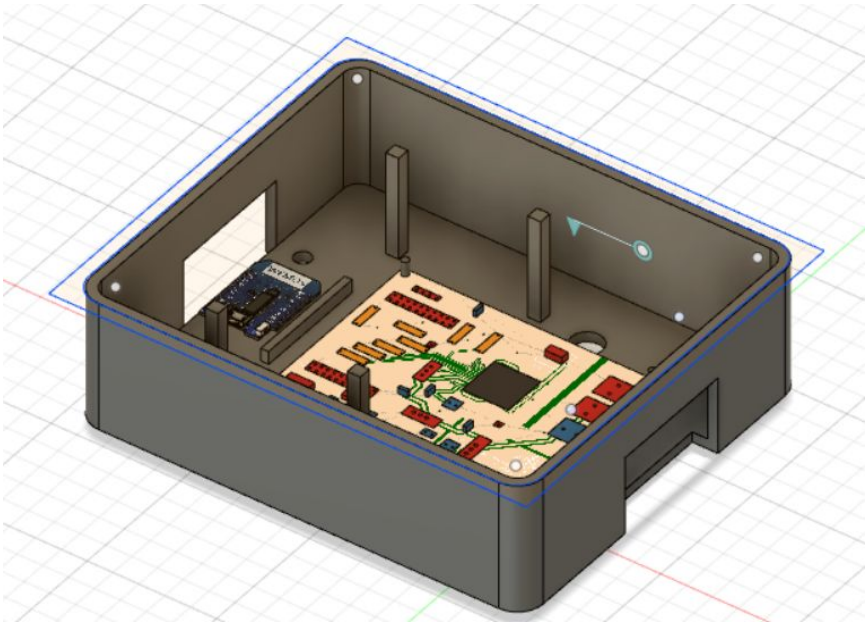
Model of bare wheel before rubber

The car is a rear-wheel-drive system with a DC Motor for acceleration and Servo for steering control. To drive signals to these devices, a Motor Control PCB is used.

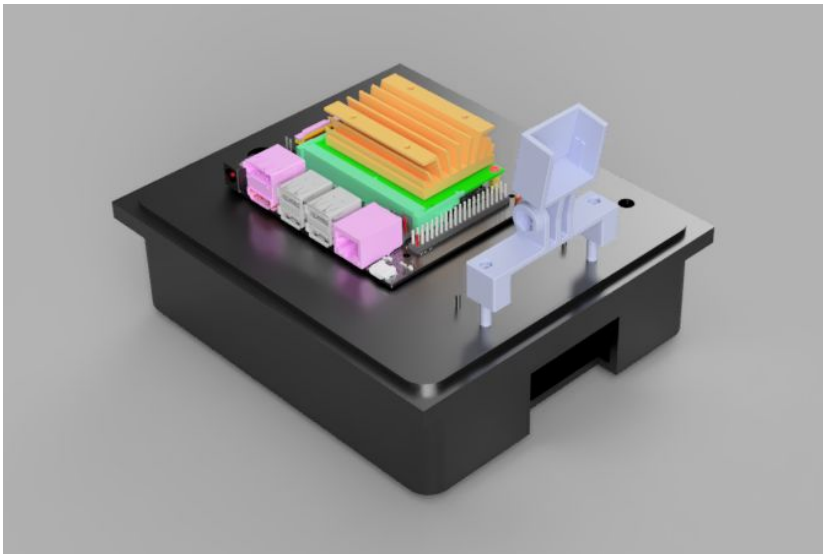


Motor Control PCB

The next step was to create an enclosure that houses the electronics. To accomplish this, I designed a simple box with holes for wires and airflow. The Jetson Nano mini-computer board was placed on top of the board, secured by stubs.



3D model developed in Fusion 360
The battery was to be placed on the sides



3D Render

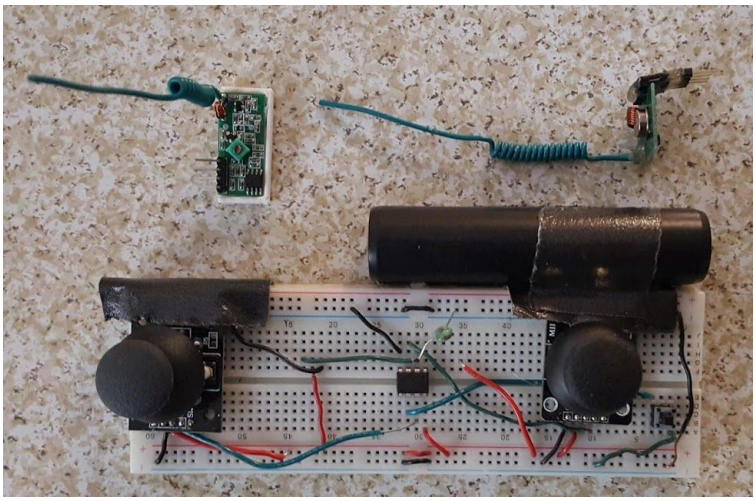
(Raspi-Cam Camera Mount Source:)

Signal Processing:

An Arduino Nano is used to generate the real-time hardware PWM signals. The microcontroller simplified training by creating a sort of a raw input black box where the steering (0-180) and throttle (0 to 180) are fed. As an added benefit, the main mini-computer is relieved of some computational stress.

To control and assist training, I needed a remote control wireless communication with the mini-computer. In an effort to keep the controller processing on the Arduino Nano and reduce latency, my first attempt involved some time trying to develop my own controller.

A small remote controller was developed on a breadboard with two Joystick analog inputs for steering and throttle. I chose Attiny85 to process the signals due to its low power consumption and the availability of two 10-bit ADCs. RF 433Mhz was used for wireless communication between the Arduino Nano and the Attiny85 controller.



However, this method suffered from interference and lacked range. I was successfully able to extend the range using a loaded coil design antenna, but the wireless communication peripherals connected to the Jetson Nano interfered.

After scouring the internet to find low latency communication methods, I found a library that can process signals from popular game controllers called Approximate Engineering's Python Game Controller library. Using this library, a PS4 controller paired to the Jetson Nano now transmits signals to the Arduino Nano. The availability of many buttons on the controller became enormously useful for training control.

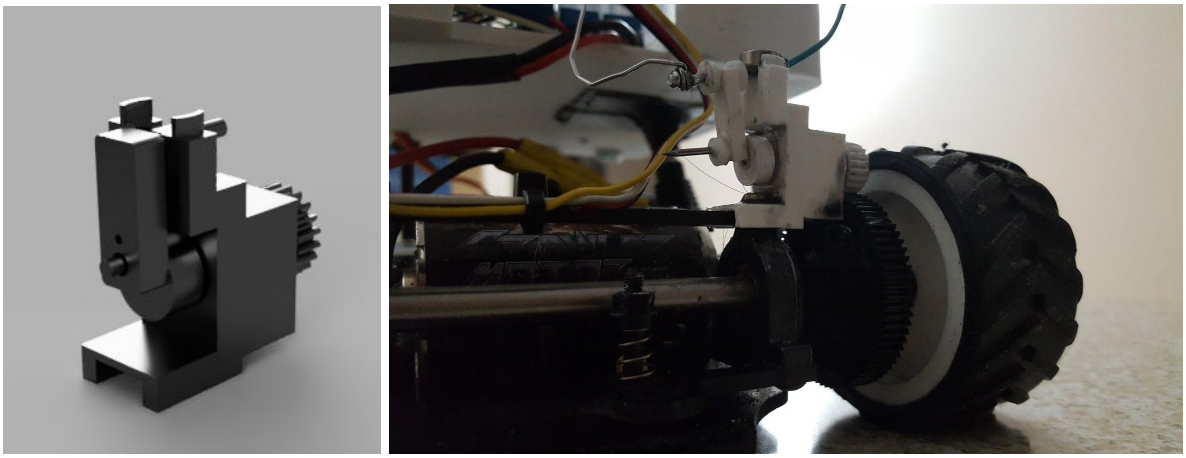
Moving forward, the keymap config file for the PS4 controller needed to be altered due to the default erroneous mapping. I was able to open context managers in Jupyter to gain wireless controller access, though I needed to run Jupyter as root. The continuous outputs from the PS4 controller are rescaled to range from 0-180, while the discrete outputs had a time-held down value.

```
# PI Cam 160 FOV 60 FPS @ 1280x720 (GSTREAMER SAYS 120?)
with serial.Serial('/dev/ttyUSB0', 1000000, timeout=1) as ser:
    with ControllerResource() as joystick:
```

Context managers

In-between testing and training methods, the car needs to move at a constant, stable speed. Positive values are interpreted as manual control values, while negative values are interpreted as constant value mode. The absolute value of the negative term is used as the target rpm in a mini control loop (with just a proportional term) in the MCU.

Initially, I used a digital hall-effect sensor with neodymium magnets on the wheel interiors to measure rpm, but unfortunately, I found this method to be inaccurate and unstable at higher speeds. In an effort to find a more precise, inexpensive method to measure rpm, I designed a small crank shaft mechanism that could be fitted to the DC-motor gear in the back of the car. Although this method produced better results, the additional gear introduced too much torque. Ultimately, I reverted back to the hall-effect and reduced the car speed.



Gear spins the shaft which will make contact with the magnet (connected to ground).

The raw input from the PS4 controller is sent over to the Arduino via USB Serial to avoid noise from the DC-motor. Serial communication produced excellent results, giving precise control as well as very little lag (at 1 million baud). Pyserial library is used in Jupyter Notebook to initiate an instance of Serial Communication.

However, the notebook server needed to be run as root the user running needs to be added to dailout group to allow Serial access, which could create some security concerns.

Jetson Nano:

The Jetson Nano board is a miniature computer running a light-weight Ubuntu OS with all the essential ML tools installed. The board is set up in headless mode with a 64GB micro sd with a majority of the packages installed via pip3. The tutorial from here is a guide I referenced.

Peripherals:

A Wifi and Bluetooth adapter needed to be purchased due to the lack of onboard. The visual input to the mini-computer is a Pi Camera connected via CSI cable.

Camera:

The CSI interface on the Jetson Nano board is limited to a few different sensors. I initially used the Pi Camera v2.1 (stock Sony IMX219 image sensor) with a horizontal FOV of 62.2 degrees, but I soon realized that this FOV was far too small to capture the lane adequately. To solve this problem, I kept the same driver board and replaced the image sensor with IMX219 which has a FOV of 160 Degrees. Securely fastening the Camera and increasing the height of the lid allowed for a much larger and more stable view of the lane.

I found myself frequently switching Camera libraries for the CSI interface due to the many problems such as the inability to rotate the camera, and the capture thread refusing to terminate. After searching through Nvidia's repositories projects demonstrating CSi Cameras, I was able to restructure some of the code to fix the library. Here is the version used.

Power:

The power for the DC Motor, Servo, and Arduino Nano comes from a 7.2 NiCd battery that is connected to a linear voltage regulator on the Motor Control PCB.

Motor control PCB battery

The Jetson Nano needed to have a power supply capable of delivering 5V at 3A. This is critical because the Jetson can consume 2A just without any peripherals, so 3A+ is ideal for some GPIO outputs, USB peripherals, and the Pi Camera attached directly onto the board. I ran the Jetson Nano at 10 Watt mode with forced maximum clock speed without any throttling, which increased power consumption.

I initially attempted to power the board using a 1200mAh 7.2V NiCd battery connected to an adjustable buck converter that is capable of delivering 2.5A, but I found that the battery was depleting rapidly and reaching the 2.5A limit easily- causing unexpected shutdowns.

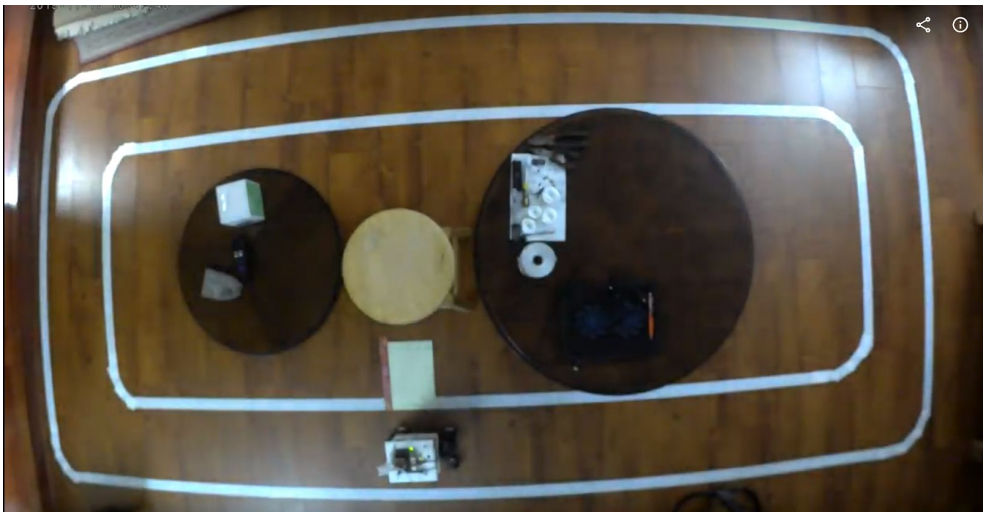
I instead opted for a USB 10,000mAh battery bank that is capable for of 5V @ 3.4A. This setup is reliably able to supply power for ~5-6 hours. However, this battery was much bulkier, so some adjustments had to be made to accommodate the large battery bank.

Final version of car

Analysis:

Environment Exploration and Visualization:

The environment is an oval race track made using White Duct Tape. The track is designed to be slightly larger than the width of the car at approximately 1 foot wide. The light reflection and track's tight corners introduced additional challenges during image processing. Even with high FOV camera sensor, the car could only pick up a single line in the corners. To combat this, I extrapolated from previous values and current slope. Eventually, after tweaking OpenCV parameters, I was able to isolate the lines from reflections. This process is explained further in detail further down.



Top down view of track

The Camera library, is configured to output a 244x244 BGR image (BGR is OpenCV output). To reduce the size of the network and improve performance, the raw images were further processed and resized.

The input space varied depending on the method deployed. Generally, they were image tensors of varying dims: 1 x height x width x channels (Keras with TF backend).

Algorithm, Techniques, and Benchmarking:

DQN, DDPG, OpenCV, and PID are used to explore the problem.

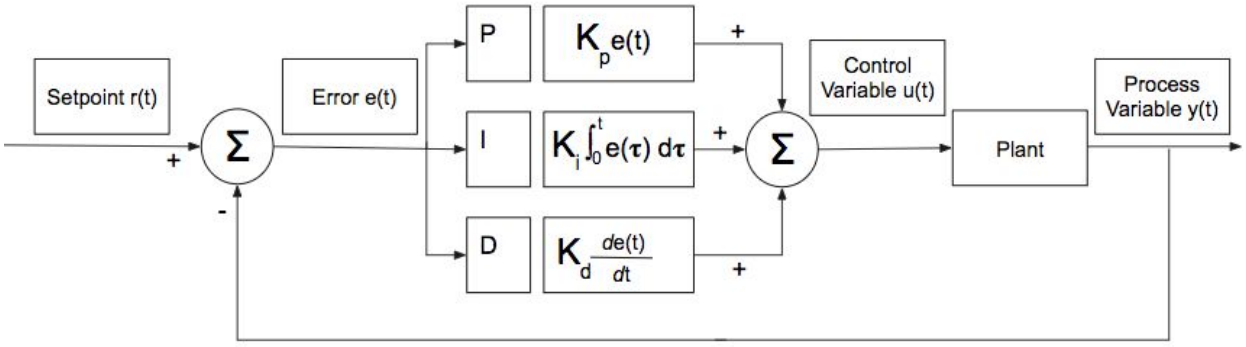
Deep Deterministic Policy Gradient (DDPG) is an Actor-Critic method that is capable of outputting continuous state space and continuous action space which is very beneficial for my application. Even though the soft off-policy target network update allows for a

more stable output, DDPG is highly sensitive to hyperparameter changes which are difficult to tune. DDPG can also be taxing to memory due to the 4 networks.

Q-Learning is a reinforcement learning algorithm where the goal is to learn the optimal policy by maximizing the expected value of the total reward. Deep Q-Learning (DQN) is an advancement of Q-Learning with Deep Neural Networks which allows learning from high dimensional states such as pixels in an image.

DQN is not ideal for continuous action spaces, so I needed to discretize my action space to 19 linear activated Q-Values that map to 0-180 degrees. A traditional epsilon greedy policy is not needed since I am able to manually introduce some variability.

PID control loop mechanisms are widely used in industrial control applications and are relatively easy to set up. The coefficients were found using a heuristic method of PID tuning called Ziegler-Nichols method. A modification of the output will serve as the reward function.



Proportional-integral-derivative control loop

OpenCV is an open-source library that can quickly process images in real-time. The OpenCV version that comes with the Jetson Nano image provided by Nvidia is not optimized for GPU. Instead, I am using an optimized OpenCV4 installation from this article for increased performance. This open-source library has many powerful utilities that can be used to extract and analyze image data efficiently.

As a benchmarking model, a plain regular CNN with a continuous output space (0-180) is used. I came upon the model from this article which utilizes a CNN model from Nvidia's research paper. The input space here is a 1x200x66x3 image tensor with YUV color space (converted using OpenCV).

Finally, the models are judged against one another to compare the performance using the metrics discussed earlier.

Methodology, Implementation, and Refinement:

OpenCV PID:

The first challenge was to develop a way to track and locate the ideal heading in a Lane from this picture similar to this.

Raw BGR image

I explored many different approaches and discovered some useful approaches here and here, but these were unable to tackle the tight, rounded corners on my track. The linked methods utilize OpenCV's Hough Line Transform, which is typically used to detect straight lines.

Fortunately, OpenCV has a contour detection method. Using this, I was able to locate and isolate different contours in an input image with hierarchy. I then filtered out the irrelevant contours based on known characteristics of lane lines.

Using regression on the contour points to find the best fit line was only useful for vertical lines, so I performed a rough calculation with the contour points to estimate the center of the lane. In the case that the Camera only found one line, I used the slope of the line to determine whether it was a left lane or right lane. Then a static offset based on the width of the lane is added to approximate the center of the lane.

The pipeline is as follows:
Input image 244x244x3 BGR
Convert BGR to grayscale
Run cv2.Canny() edge detector
Run cv2.findContours() to return a list of contours

Loop through contours (default hierarchy)

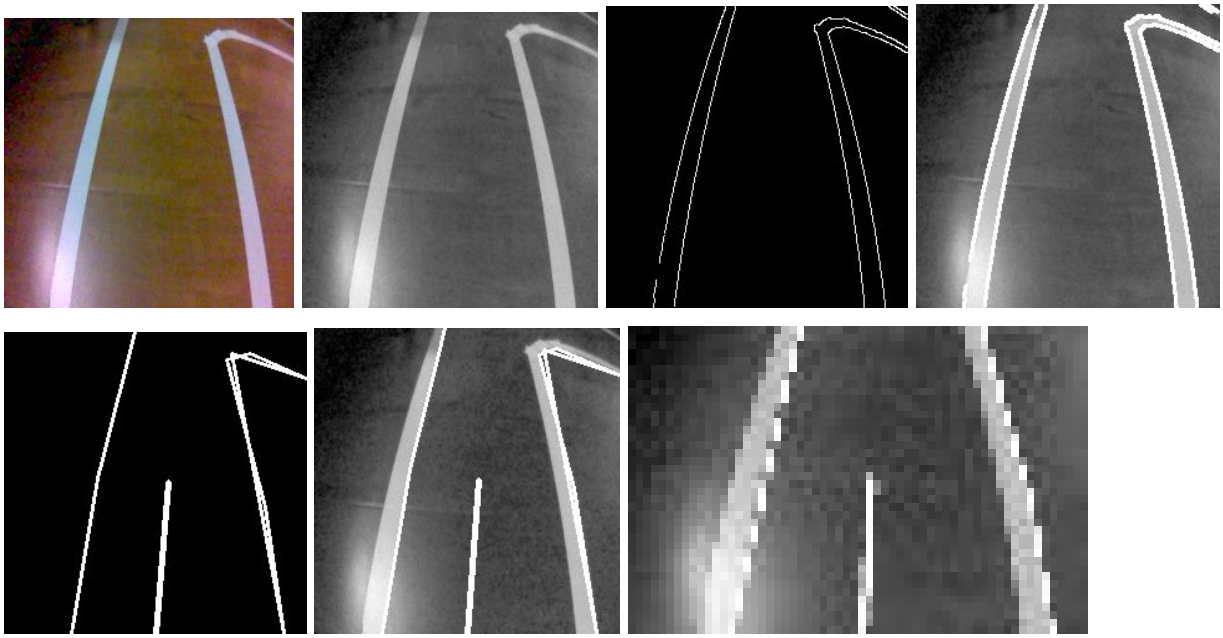
- Get best-fit rectangle and calculate the area using cv2.minAreaRect()
- Calculate arc length using cv2.arcLength()
  - After Isolating contours using the above criteria
    - Determine if left line or right line using box coordinates
      - Calculate slope
      - Approximate polygon using approxPolyDP()
      - Store contour line with largest arc length for left and right

- If both lines are detected
  - Draw both lines
  - Calculate heading coordinate point X using contour points
- If only a single line is detected
  - Draw a single line
  - Based on the slope, determine if left line or right line
  - Use a static offset to approximate center point X using contour points

- Calculate the angle to the center using atan
- Run through PID control loop and np.clip output (0-180)

The steering output is the final value that is fed to the Arduino over Serial.

Visual of pipeline:

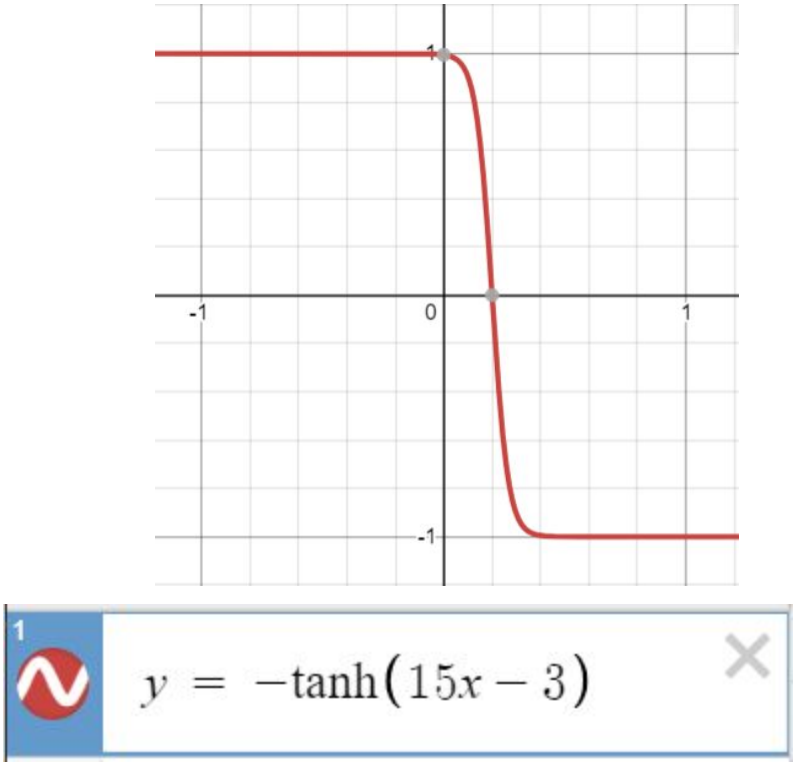


DQN:

Computing Reward Function:

The agent takes steps with a small delay to allow time for the next step to take place and to avoid repeat states. The PID output is computed to serve as a reward function. Due to the oscillations from PID, the values are rounded to the nearest multiple of five and normalized. This function below then converts the normalized value to range from -1 to 1 with a tight_tanh behavior.

tanh function:



Graph generated from [desmos]

DQN Learning step:

State space and Action space:
The state is the normalized black and white image tensor (1x80x80x1).
The action is periodically sourced from manual control, the PID output, and a random integer generator.

Since the Camera frame rate is high, there were many duplicate SARSA. This can cause overfitting and instability due to an unbalanced input. To solve this problem, DQN learns with a fixed delay to avoid overfitting and allow the environment to transition to the “next_state.” The learning step computes the Q_target to fit the model. During training, the car moves relatively quickly through the straights and slows at the curves which was advantageous for equal samples.

The DQN Network:

The input space is an image tensor 1x80x80x1 (1 x height x width x channels for Keras with TF backend). The image is single-channel because the image is back and white. The action space is discretized 19 (0-18 inclusive) linear activated Q-Values that map to 0-180 degrees by multiplying the output by 10.

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           (None, 80, 80, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 19, 19, 32)        2080
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 64)          32832
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 6, 6, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               590080
_________________________________________________________________
Q_Values (Dense)             (None, 19)                4883
=================================================================
Total params: 666,803
Trainable params: 666,803
Non-trainable params: 0
_________________________________________________________________
None
```

DQN param details

```
def _build_model(self):
    # Neural Net for Deep-Q learning Model
    # Define input layer (states)
    states = layers.Input(shape=(self.state_size), name='input')
    c1 = layers.Convolution2D(filters=32, kernel_size=8, strides=4, activation='relu')(states)
    c2 = layers.Convolution2D(filters=64, kernel_size=4, strides=2, activation='relu')(c1)
    c3 = layers.Convolution2D(filters=64, kernel_size=3, strides=1, activation='relu')(c2)
    l1 = layers.Flatten()(c3)
    l2 = layers.Dense(256, activation='relu')(l1)
    Q_val = layers.Dense(units=self.action_size, name='Q_Values', activation='linear')(l2)
    # Create Keras model
    model = models.Model(inputs=[states], outputs=Q_val)  #actions
    model.compile(loss='mse', optimizer=optimizers.Adam(lr=self.learning_rate))
    self.get_conv = K.function(
        inputs=[model.input],
        outputs=model.layers[1].output)
    return model
```

Final DQN model

Drew some inspiration from [the Atari paper]

The get_conv was a side experiment in an attempt to visualize the convolution layer.

Hyperparameters:

self.gamma = 0.99
self.epsilon = 0.2
self.epsilon_min = 0.01
self.epsilon_decay = 0.995
self.learning_rate = .0001
self.tau = 0.1

```
if(TRAIN or DEBUG):
    next_state = main_cv.preprocessed_img
    if(count % 5 == 0): # Frame Skipping
        if(DEBUG == 0):
            target = reward + car_agent.gamma * \
               np.amax(car_agent.target_model.predict(car_agent.conv_to_tensor(next_state))[0])
            target_f = car_agent.target_model.predict(car_agent.conv_to_tensor(state))
            target_f[0][int(action/10)] = target
            car_agent.model.fit(car_agent.conv_to_tensor(state), target_f, epochs=1, verbose=0)
            car_agent.batch_id += 1

            # TAU UPDATE
            if(count % 60 == 0):
                car_agent.target_train()

        # SARSA at this moment (for STATE)
        state = next_state.copy()
        action = train_STEER
        reward = tight_tan(abs(train_STEER-cv_action)/180)
    count += 1
```

Learning step

DQN Refinements:

I initially used a learning rate of 0.001 with an image size of 40x60, but this resulted in exploding Q-values and poor performance. I observed that the entire Q-table was significantly affected despite reducing the learning rate and modifying the reward function.

After some trial and error, I expanded the network by changing the image size to 80x80, implemented frame skipping, created a target network that updates at a rate of tau, and reduced the learning to 0.0001. These changes significantly improved the result.

Nvidia Model: (To be used as a benchmarking model)

Nvidia Model CNN implemented in the article:

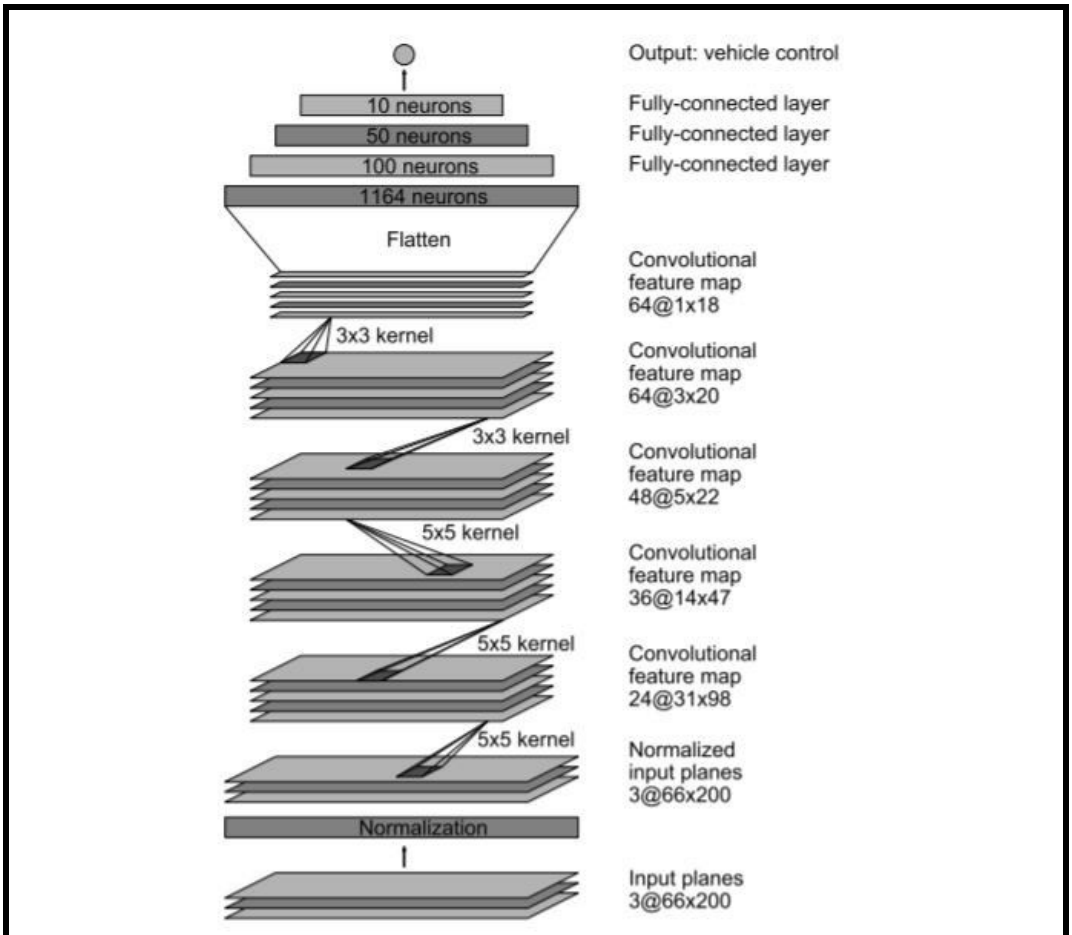


Figure 4: CNN architecture. The network has about 27 million connections and 250 thousand parameters.

The image was pre-processed using Nvidia's recommendations.

```
# Source: https://towardsdatascience.com/deeppicar-part-5-lane-following-via-deep
# Convolution Layers
from keras import layers, models, optimizers, regularizers

frame = layers.Input(shape=(66, 200, 3), name='input')

c1 = layers.Convolution2D(filters=24, kernel_size=5, strides=2, activation='elu')
c2 = layers.Convolution2D(filters=36, kernel_size=5, strides=2, activation='elu')
c3 = layers.Convolution2D(filters=48, kernel_size=5, strides=2, activation='elu')
c4 = layers.Convolution2D(filters=64, kernel_size=3, activation='elu')(c3)
c5 = layers.Convolution2D(filters=64, kernel_size=3, activation='elu')(c4)

l1 = layers.Flatten()(c5)
l2 = layers.Dense(100, activation='elu')(l1)
l3 = layers.Dense(50, activation='elu')(l2)
l4 = layers.Dense(10, activation='elu')(l3)

output = layers.Dense(1)(l4)

model = models.Model(inputs=[frame], outputs=output)

# we use MSE (Mean Squared Error) as loss function
optimizer = optimizers.Adam(lr=1e-3) # learning rate
model.compile(loss='mse', optimizer=optimizer)
```

Nvidia model implemented in Keras

The pipeline is as follows:

Original image from Camera 244x244x3 BGR
Convert color space to YUV
Resize the image to 200x66x3

Pipeline Visual:

The final image tensor that is fed into the network is 1x66x200x3

Nvidia Refinements:

I found that this model was highly prone to overfitting. I suspected the high frame rate and imbalance of training images to be the cause. So, I stored the samples into two buffers: turn_buffer and regular_buffer by using the predicted PID output. When it was time to train, I randomly drew from one of buffers. This allowed for a balanced training samples and thus which solved the problem.

```
STEERING = cv_action
d = experience(main_cv.preprocessed_img, STEERING)

# Frame skipping
if(count % 30 == 0):
    if(DEBUG == 0 and (len(turn_buffer) > 5 and len(regular_buffer) > 5)):
        chance = rn.randint(0, 1)
        if(chance==0):
            rnd_idx = rn.randint(0, len(turn_buffer)-1)
            loss = (model.fit(turn_buffer[rnd_idx][0], np.array([turn_buffer[rnd_idx][1
        else:
            rnd_idx = rn.randint(0, len(regular_buffer)-1)
            loss = (model.fit(regular_buffer[rnd_idx][0], np.array([regular_buffer[rnd_

        ID += 1
```

Nvidia training step

```
# Compute Validate Score
if(joystick['circle'] is not None): # (X)
    print("TRAIN STOP")
    all_times.append((time.time()-loop_start))
    TRAIN = False
    AUTO_THROTTLE = False

    # Stop Car
    output = "{:05d}-{:05d}\n".format(int(90), int(STEERING))
    ser.write(bytes(output,'utf-8'))
    ser.write(bytes(output,'utf-8'))
    time.sleep(0.2)

    Y_true = []
    Y_pred = []
    if(len(validate_memory) >= 30):
        print("Memory Size:", len(validate_memory))
        for i in range(max(len(validate_memory),100)):
            rnd_idx = rn.randint(0, len(validate_memory)-1)
            Y_pred.append(int(model.predict(validate_memory[rnd_idx][0])))
            Y_true.append(validate_memory[rnd_idx][1])

        MSE = ((np.array(Y_pred) - np.array(Y_true)) ** 2).mean()
        print(MSE)

        mse = mean_squared_error(Y_true, Y_pred)
        r_squared = r2_score(Y_true, Y_pred)
        print("mse    {:1.3f} ".format(mse))
        print("r_squared    {:1.3f} ".format(r_squared))
```

A method to monitor progress using MSE and r_squared

Results:

Model Evaluation and Validation:

	PID	network_size:	NA
lap	time	total train time sec:	NA
1	15.5	train_iterations:	NA
2	15.8		
3	15.8	train_FPS:	NA
4	14.8	test_FPS:	110.77
5	15.2	avg_lap sec:	15.42
manual throttle lap:	7.8		

	DQN	network_size:	666,803
lap	time	total train time sec:	2273.35
1	12.8	train_iterations:	16935
2	14.3		
3	14.6	train_FPS:	51.45
4	15.5	test_FPS:	74.56
5	15.5	avg_lap sec:	14.54
manual throttle lap:	8		

	NVIDIA	network_size:	252,219
lap	time	total train time sec:	13366
1	12.9	train_iterations:	15928
2	12.8		
3	13.2	train_FPS:	50.7
4	14	test_FPS:	65.31
5	15	avg_lap sec:	13.58
manual throttle lap:	6.2		

FPS was recorded to get a general sense of computational load.

The models were trained until they were able to successfully navigate the track autonomously and consistently. (a quick lap completion test was done along with a MSE computed from randomly stored samples).

- test_FPS:
    - The minimum setup required to run the model (see demo notebook).
- train_FPS:
    - Involves all the tools required to train. Nvidia, and DQN had PID computation running alongside during training.
- avg_lap sec:
    - The average lap time of the five completely autonomous constant throttle lap times in seconds.
- total train time sec:
    - The total time spent only on training.
- train_iterations:
    - Number of times model.fit() is called (on a single data point in all cases).

Even though PID had the highest FPS, it was the slowest performing of the three. This method was highly sensitive to environmental and speed changes. Changing the lighting affected the ability to identify the lane. As I increased the throttle, the path-finding algorithm struggled to keep up and escaped the lane lines. This was expected because the algorithm was only tuned to perform well in slow conditions.

The model was from Nvidia's paper had the best average lap time, however, the training time was long ~3.7 hours compared to DQN which only needed ~40 minutes to pass the lap completion test. The Nvidia model was resistant to changes in lighting and high acceleration. While the car was running on the track, I randomly shift the steering drastically to see if it was capable of recovering. Both DQN and Nvidia were able to recover and re-adjust to the center of the track.

From this experiment, I was able to conclude that Reinforcement Learning can expedite learning and can perform just as well as traditional methods.

Justification:

The Nvidia model had a continuous output, which gave it an advantage over DQN's discretized output. The performance of DQN with a lower training time is closely riveled the Nvidia model's performance and was equally as robust when there were changes in the environment.

In addition to constant-throttle training, I included some random steering to learn recovery methods. I was pleased to observe that the model was able to adapt this recovery method to other sections of the track.

There are a few more variables that need to be tackled and approaches that need to be studied to make any further conclusions. Nevertheless, this was an incredible learning experience.

Conclusion:

Visuals:

I've recorded the first person view as well as a bird's eye view of the five lap times from each of the different methods.

Youtube link: https://www.youtube.com/watch?v=aXGxy_qaXkY

The video also briefly shows a timelapse of the training process for Nvidia's model and DQN.

Some observations from video:

Note: The "speed" values are the RPM from the microcontroller. Since the value is unstable and inaccurate, this value was not used.

The first person view exported via OpenCV has some export issues due to incorrect settings, as a result, the video could not be aligned with the top-down as well.

- DQN:
    - Car appears to slightly cut some of the corners, this could be due to the sudden shift to 170 in these locations.
- NVIDIA:
    - The corner steering appears to be smooth and somewhat continuous, which could explain it's higher performance.
    - The driving behavior also looks very similar to the PD version.

Reflection:

Many aspects of the project were challenging, but tuning DQN and PID parameters were especially challenging and time-consuming. Coordinating all the hardware to store and train efficiently in real-time was challenging as well. Although I very much enjoyed working on this, I severely underestimated the time needed to complete the project (~1.5 months). In the end, I had hoped that Reinforcement Learning would significantly outperform the other methods, but this was not the case.

Improvements:

There are quite a few improvements that I initially planned, but was unable to come to fruition due to time constraints. I plan to implement them at a later time.

Car related hardware:

- ❏ As I upgraded parts, the original design could no longer accommodate them, as a result I had to make some make-shift adjustments.
- ❏ Rather than a single center camera, dual cameras placed on the left and right ends of the car could provide better lane tracking.
- ❏ Automatic throttle system needs to be rewritten for stability and consistency. The hall-effect sensor interrupt triggers currently setup is not sufficient for accurately measuring rpm at higher speeds. Perhaps another wheel attached to the crank-shaft mechanism can be an alternative way to accurately measure rpm without introducing additional torque.
- ❏ Investigate Jetson Nano's onboard PWM pins to drive signals to the motor control PCB, rather than through the Arduino Nano. A dedicated thread to handle the control signals could lead to better performance.

PID:

- ❏ The algorithm currently is designed around a specific environment and I suspect it might not be able to generalize under different conditions. There are definitely some improvements that could be made to the core algorithm. I plan to exploit more well known characteristics of Lane Lines to identify them better.
- ❏ The throttle system needs to be improved to perform consistently for a more fair comparison of the approaches. Perhaps one that is based on the steering angle could yield better results.

Reinforcement Learning:

- ❏ I was really looking forward to implementing DDPG, which is capable of outputting continuous action space. I am interested in observing the performance of this method for further analysis.
- ❏ In addition to steering, I want to include throttle control output as well. I have read about successful attempts of a reward function using the cross track error with the car speed. However, to accomplish this, I needed accurate rpm measurement.

The opportunity to dive into many fields and improve my knowledge in the many involved fields made this project incredibly rewarding. Once the revisions are applied and everything is tuned, I plan to recreate the project at a later time with a larger car on a full-sized road. Stay tuned.

Feel free to contact me at vsksriram@gmail.com for details, comments or concerns. Thank you for reading.

Relevant Links:

1. Achieved using a Soft-Actor Critic in simulation.
   https://towardsdatascience.com/learning-to-drive-smoothly-in-minutes-450a7cdb

2. Achieved using a Convolution Neural Network in simulation.
   https://blog.coast.ai/training-a-deep-learning-model-to-steer-a-car-in-99-lines-of-code-ba94e0456e6a

3. Full end to end self-driving implementation using small scale car powered by Raspberry Pi. Nvidia's CNN was utilized here.
   https://towardsdatascience.com/deeppicar-part-5-lane-following-via-deep-learning-d93acdce6110

4. Achieved using Double Deep Q Learning in simulation.
   https://flyyufelix.github.io/2018/09/11/donkey-rl-simulation.html

5. Atari DQN with improvements.
   https://becominghuman.ai/beat-atari-with-deep-reinforcement-learning-part-2-dqn-improvements-d3563f665a2c

6. Concept Links:
   a. https://sergioskar.github.io/Deep_Q_Learning/
   b. https://ml-cheatsheet.readthedocs.io/en/latest/
   c. https://towardsdatascience.com/reinforcement-learning-w-keras-openai-dqns-1eed3a5338c
   d. https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning
   e. https://medium.com/@cacheop/pid-control-for-self-driving-1128b42ab2dd

A big thank you to those involved in the above links. They were enormously helpful and inspiring.

Project files link:
https://gitlab.com/engineer6080/Education/blob/master/Machine_Learning/Autonomous_Car/