CSE579 Project Report
Drive Safe and Save

Toby Ouyang (ouyang36), Ziyuan Wang (wangzi), Zhaoyu Gong (zhaoyg)

December 15, 2022

1 Introduction

Autonomous driving receives ever-increasing attention with the development of computer vision, accumulated knowledge in vehicle dynamics, and the availability of novel sensor modalities, which together provide solid fundamentals that are seemingly able to ensure promising results. Governments and companies have already invested tremendous resources into this field of practice. In this project, we focus on two main challenges - safe and save - in the framework of the autonomous driving system that researchers and practitioners are faced on daily basis.

The first and probably the most important challenge preventing autonomous driving from becoming reality is the safety concern. The fatality caused by the failure of autonomous driving systems is simply unacceptable. Without the safety issue being addressed, autonomous driving will never come into play in the future of humanity.

Even though the proviso of the actual realization of autonomous driving is stringent, its potential of facilitating the development of human civilization is still tempting. One lure of autonomous driving to the public is the efficiency it can bring to our daily life upon its realization - taking one from point A to point B as fast as possible. If every agent on road can act in its optimal way based on global information, tons of time can be saved, let alone emission reduction and other benefits. Furthermore, if safety is guaranteed, autonomous driving is able to in turn avoid the fatalities caused by unprofessional human driving behaviors.

The goal of this project is to explore some trendy techniques, get some hands-on experience, and possibly make some intellectual contributions to this field of study. More specifically and quantitatively, we explore and test methods and algorithms that ensure the automated car stays on track all the time (safe) and at the same time moves as fast as possible (save).

2 Environment

Environment Setup:
conda create -n car_racing python=3.9
mkdir car_racing
cd car_racing
(copy env_carracing.yml to the folder)
conda env update -n car_racing -f env_carracing.yml
conda activate car_racing

For this project, we will be using an open-source environment called Box2D Car Racing from OpenAI Gym. It is a top-down racing environment that allows users to test their control and/or learning algorithms from pixels.

The map consists of three regions, which are track, grass, and dead zone. The track is depicted as the gray region in Fig.1. The racing car will have larger friction if it's running on the track compared to the

grass region, which is shown in green in the figure. The black region is the dead zone which is outside the boundary of the map. If the car is running into this region, the game will end immediately and a new episode will start. The map will be generated randomly in every episode with all of the sharp turns marked out. At the beginning of each episode, the car will default to somewhere at the center of the road at rest.

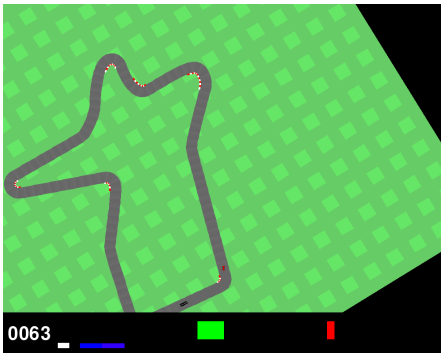


Figure 1: Illustration of the map of the game

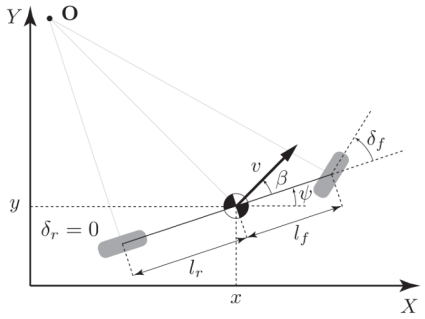


Figure 2: Standard bicycle kinematics

If working in a continuous space, the control/action users have over the racing car are gas, brake, and steering. If in a discrete space, the control/action will become steer left, steer right, gas, brake, and do nothing. The racing car is set to be a rear-wheel drive vehicle for which the power only applies to the rear wheels and the steering command only applies to the front wheels but the brake will be applied to all four wheels.

The observation window is 96x96 pixels and contains only a portion of the map information. Users will have access to the racing car's states including but not limited to: the longitude velocity, lateral velocity, and current position in the world coordinate.

3 Model-Based Control

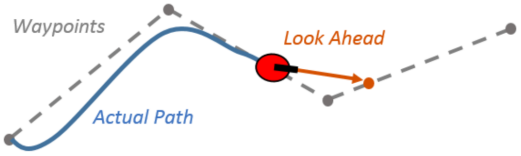
One strategy to accomplish the goal of the project is model-based control. We divided it into two parts. First of all, we will run a model-free path-following algorithm that drives the car along the centerline of the map for the first lap. In the meantime, driving data, such as velocity and steering, will be recorded. After the first lap, we will use the recorded driving data to construct a dynamic model for the racing car to describe its state evolution. Next, we will develop a trajectory generator that creates a set of potential trajectories and choose the optimal one based on some reward metric as the target trajectory to track. Once we figure out the optimal trajectory, we will then use the model that we construct in the first part to design some model-based controllers to do the trajectory tracking task. More specifically, MPC will be designed to track the generated optimal trajectory.

3.1 Pure Pursuit

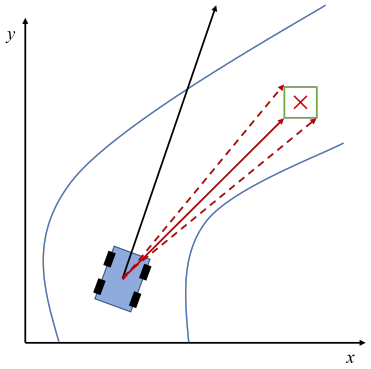
Pure pursuit is a simple model-free path following an algorithm. It computes the steering command that moves the vehicle from its current location to some look-ahead checkpoint as shown in Fig.2. The vehicle's linear velocity is assumed to be constant from all-time tracking the pre-determined path.

For our project, the implementation is first we accelerate the car for a certain amount of steps until it reaches the desired velocity. Once it reaches the velocity, the checkpoint pursuit is activated to move the vehicle toward the current checkpoint, which is depicted in Fig.3b. We compute the angle between the vertical axis of the world coordinate (y-axis) and the line connecting the car and the current checkpoint. Then, we compare the aforementioned angle with the racing car's heading angle to determine if the car is heading straight toward the current checkpoint (red arrow direction). If it's off the desired direction, we give

the car corresponding commands to correct its heading (black arrow direction) within some tolerance range (red dashed arrow). If the car passes the 4x4 pixel square area (green boxed area) where the center is the current checkpoint position, we update the checkpoint to the next checkpoint. For example, in Fig.3b, the car will receive a steer right command to correct its heading until the heading falls into the range between two red dashed arrows.



(a) Illustration of pure pursuit algorithm



(b) Illustration of the proposed pure pursuit path following algorithm

Figure 3: Pure pursuit illustration

3.2 Trajectory Planning

Previously, we have already developed a pure pursuit path following algorithm, which allows us to track some pre-determined trajectory (e.g. centerline of the track). In this section, we will discuss how to generate such trajectories with limited map information in order to mimic the real-life driving scenario, and how to select the optimal one to track in order to drive safe and save.

The idea we proposed is to generate a set of arcs that pass the vehicle and choose the optimal one with the highest reward based on the local map information as the target trajectory to follow. First, each arc is one-fourth of a circle of which the radius is set to be the distance to the center of the racing car. The center of the circle is on the horizontal extended line that passes through the center of the racing car. As such, by varying the location of the center of the circle close or away from the car's center, the arc trajectories can be generated easily with different curvatures.

It is obvious that some of the trajectories will not be considered (e.g. trajectories that direct the car to move out of the track). In order to systematically select the optimal trajectory, we create a scoring metric, and then we can quantitatively evaluate the goodness of each trajectory. For each trajectory, we give positive rewards for new tiles visited, and give negative rewards for each frame elapsed and the occurrence of out-of-the-track events. The function of the reward can be summarized in the following equation,

$$r(\{P_i\}) = 1000 * N_{tiles\_to\_visit}/N - 50 * N_{pts\_outside\_track} - 0.1 * length(\{P_i\})/v \quad (1)$$

where $\{P_i\}$ is a trajectory (expressed by several discrete points $P_i, i = 0, 1, 2, ...$), N is the total number of tiles composing the whole track, $N_{tiles\_to\_visit}$ is the number of tiles that the trajectory will visit, $N_{pts\_outside\_track}$ represents the number of points in the trajectory that is the outside track, and v is the velocity of the car.

3.3 System Identification

In order to yield better performance, it will be helpful to make use of some underlying physics of the system when giving the control input. It's often referred to as a model-based control. The first step is to construct

a mathematical model that is able to capture the dynamics of the target system, which is called system identification.

For our case, we use the standard bicycle model as an approximation of the kinematics of our racing car system, which is shown in Fig.4. From the kinematics, we construct a system of equations that describes the dynamics of the racing car system augmented with the local map information as follows [6, 2]:

$$f(x_t, u_t) = \begin{bmatrix} \theta_{v_x}^{(1)} + \theta_{v_x}^{(2)} \dot{\psi}_t v_{y_t} + \theta_{v_x}^{(3)} a_t \\ v_{y_t} + \theta_{v_y}^{(1)} \frac{v_{y_t}}{v_{x_t}} + \theta_{v_y}^{(2)} \frac{\dot{\psi}_t}{v_{x_t}} + \theta_{v_y}^{(3)} \dot{\psi}_{y_t} v_{x_t} + \theta_{y_t}^{(4)} \delta_t \\ \dot{\psi}_t + \theta_{\dot{\psi}}^{(1)} \frac{v_{y_t}}{v_{x_t}} + \theta_{\dot{\psi}}^{(2)} \frac{\dot{\psi}_t}{v_{x_t}} + \theta_{\dot{\psi}}^{(3)} \delta_t \\ e_{\psi_t} + \Delta t(\dot{\psi}_t - \frac{v_{x_t}\cos(e_{\phi_t}) - v_{y_t}\sin(e_{\psi_t})}{1 - e_{y_t} c(s_t)} c(s_t)) \\ e_{y_t} + \Delta t(v_{x_t}\sin(e_{\psi_t}) + v_{y_t}\cos(e_{\psi_t})) \\ s_t + \Delta t \frac{v_{x_t}\cos(e_{\phi_t}) - v_{y_t}\sin(e_{\psi_t})}{1 - e_{y_t} c(s_t)} \end{bmatrix} \tag{2}$$

where $x_t = [v_{x_t},\ v_{y_t},\ \dot{\psi}_t,\ e_{\psi_t},\ e_{y_t},\ s_t]^T$ is the state, and $u_t = [a_t,\ \delta_t]$ is the input. v_{x_t}, v_{y_t}, and $\dot{\psi}_t$ are the longitudinal velocity, lateral velocity, and yaw rate, respectively. e_{phi_t} is the heading angle error. e_{y_t} is the lateral distance error. s_t is the distance traveled along the centerline of the track. For the control input, a_t is the longitudinal acceleration, and δ_t is the steering angle. $c(\cdot)$ is the road curvature. θ's are the parameters that we need to identify to define our system.

During the first lap, we run the proposed model-free pure pursuit algorithm to track the centerline of the track and we record the control input and state at each discretization step. The data will then be used to identify the parameter vector theta to recover the original nonlinear dynamics $f(x_t, u_t)$. Note that the original system $f(x_t, u_t)$ is nonlinear, but we will still use the linear regression technique (ridged regression) to find our parameters by putting all nonlinearity into the regressor vector (e.g. $\frac{v_{y_t}}{v_{x_t}}$) by solving the following optimization problem $\theta^* = \text{argmin}_\theta \sum_{i=1}^{n} ||y_i - z_i^T\theta||^2 + \lambda||\theta||_2^2$ that has the closed-form solution takes the form of $\theta^* = (Z^TZ + \lambda I)Z^TY$, where $z_i = [x_i, u_i]$, $y_i = z_{i+1}$, and λ is the regularization parameter.

Once we obtained the dynamic model of the racing car, we can linearize it by taking Jacobian to figure out a linear model (A & B matrix) that we can use for designing model-based controllers.

3.4 Model Predictive Control

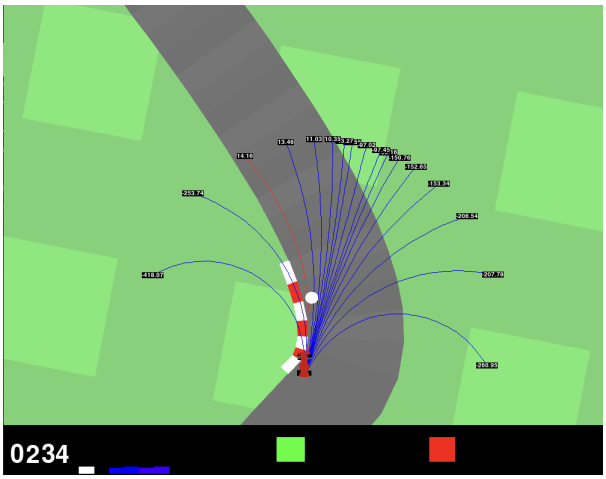
From the system identification, we extract the racing car dynamics from the recorded input and output data. After linearization, we get the A and B matrix that allows us to design some model-based controllers for trajectory tracking. In this section, we discuss the details of how the MPC controller is designed for our case. The MPC (constrained optimization) problem for our racing car is formulated as follows:

$$\begin{aligned} \min_{x,u} \quad & J = (x_N - x_{r,N})^T Q (x_N - x_{r,N}) + \sum_{k=0}^{N-1} (x_k - x_{r,k})^T Q (x_k - x_{r,k}) + (u_k - u_{r,k})^T R (u_k - u_{r,k}) \\ \text{s.t.} \quad & x_0 = x(0) \\ & x_{k+1} = A_k x_k + B_k u_k \\ & u_{min} \leq u_k \leq u_{max} \\ & x_{min} \leq x_k \leq x_{max} \end{aligned} \tag{3}$$

where $Q \succeq 0$, $R \succ 0$, N is the prediction horizon, and $x(0)$ represents the initial states.

To solve this constraint optimization problem, one efficient way is to reformulate it into a generic quadratic programming form as follows:

$$\begin{aligned} \min_{x,u} \quad & \frac{1}{2} z^T H z + g^T z \\ \text{s.t.} \quad & lb \leq \Lambda z \leq ub \end{aligned} \tag{4}$$



Therefore, we can solve the MPC problem efficiently in real-time with some open-source QP solvers (e.g. CVXGen, OSQP, etc.) to get the optimal tracking actions.

3.5 Result

Pseudo code:
While not finishing one round:
    Generate potential trajectories, Trajs
    Calculate the rewards of Trajs, RWDs
    Find optimal trajectory, Optimal_Traj
    Track the Optimal_Traj

Step 1: The algorithm will generate evenly distributed arc trajectories based on the current car state and track every half second. The total length of each arc trajectory is calculated by the predicted time times current car speed. Here, we found 2 seconds prediction time can achieve a high success rate. The blue arcs are the potential trajectories.
Step 2: After the potential trajectories are generated, the algorithm will calculate the reward of each trajectory and find the optimal one. The red arc in the figure is the current optimal trajectory.
Step 3: After the optimal is found, the pure pursuit controller will be implemented on the car to track the arc. The white point on the red arc is the current pursuit point.
Result: Play 10 episodes until the car finished a lap (i.e., revisited the start point) and count the reward.

Episode	1	2	3	4	5	6	7	8	9	10
Reward	616	645	782	630	765	707	799	615	688	621
Out-of-track	False	False	False	False	False	False	False	False	False	False
Out-of-boundary	False	False	False	False	False	False	False	False	False	False

Car speed = 25 mph, Success Rate = 100% (10/10), Average Points: 686.8
Discussion: This model can achieve a 100% success rate if the car speed has uniform speed which is less than 25 mph. However, the model works not well if the car speed is high since the algorithm doesn't consider decreasing the speed if the car is too fast to turn the next curve. This is one of our future works. Our idea is to implement the trained dynamics model to determine the maximum safe speed of the current optimal arc trajectory. If the car speed is less than this maximum speed, then input a gas. Input a brake if the car speed is greater than the maximum speed. Due to the time limitation, this part is not finished (system identification and constraint optimization not codified).

4 Learning-Based Control

4.1 Proximal Policy Optimization (PPO) [3]

Alternative to designing the driving agent by studying its underlying model, we also designed the agent by learning the lap data. In this idea, the car was controlled by a stochastic policy engine $\pi_\theta(a|s)$, whose behavior is decided by parameter θ. After driving the car and getting some data (s, a, A, s'), where A is the advantage function, θ will be updated to encourage advantageous policy while discouraging the opposite. The learning algorithm we chose was the PPO from an open source reinforcement learning toolkit-Stable Baselines 3. Three reason for choosing PPO: (1) It is an online learning algorithm, (2) PPO is a robust algorithm that uses clip to avoid overestimating a noisy observation. (3) PPO supports continuous action space and image-based observation space.

PPO update policy by maximizing a well-designed objective function $L\left(s, a, \theta_k, \theta\right)$, [5]

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmax}}\, E_{s,a\sim\pi_{\theta_k}}\left[L\left(s, a, \theta_k, \theta\right)\right] \tag{5}$$

where $E_{s,a\sim\pi_{\theta_k}}$ means an empirical average over a finite batch of samples on the collected trajectory, and L has the following form, [5]

$$L\left(s, a, \theta_k, \theta\right) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), g\left(\epsilon, A^{\pi_{\theta_k}}(s, a)\right)\right) \tag{6}$$

where [5]

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases} \tag{7}$$

and $A^{\pi_{\theta_k}}(s, a)$ is the advantage function under the policy π_{θ_k}.

The advantage function is a combination of discounted rewards (happened) and the value function (reward-to-go). To optimize θ, we should increase the possibility $\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} > 1\right)$ of positive A (i.e., encourage the advantageous actions), but decrease the possibility $\left(0 < \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} < 1\right)$ of negative A (i.e., discourage the disadvantageous actions).

However, the observation is noisy. Positive A does not always show advantageous action, and vice versa. Chance is that a bad action results in a fairly high A value, which pulls the π_θ too much to recover. To solve this problem, PPO introduces the concept of *clip*. When $A \geq 0$, the objective function reduces to $L = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1+\epsilon)\right) A^{\pi_{\theta_k}}(s, a)$, which means that the promotion of π_θ is capped at $(1+\epsilon)$. When $A < 0$, the objective function reduces to $L = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon)\right) A^{\pi_{\theta_k}}(s, a)$, which means that the suppression of π_θ is capped at $(1-\epsilon)$. The clip of the update can reduce the impact of noisy observation, and slow down the convergence before the agent visited enough states. And right because the optimization of the policy is clipped within a narrow range, it is called *proximal* policy optimization.

4.2 Train our PPO model

We used an open source module-Stable Baselines3 [1] to train our model.

Pseudo code:
Construct a PPO model with preset hyperparameters.
While less than max number of steps:
    Collect set of trajectories

Calculate the advantage estimate
Update the policy
Fit new value function
if the mean reward over the past 60 episodes is higher than ever:
    Save the model

Hyperparameters:
Hyperparameters are important to train a model efficiently. We largely reuse the hyperparameters recommemded on rl-baselines3-zoo [7]. For example, the batch size was set 128, meaning the trajectory for each learning round contains 128 steps. The clip range (ϵ in equation 6) is set 0.2 (a quite common choice). And the policy type is CNN policy, given that the state returned by the step function is an image instead of a structure of numbers. We customized the number of steps in each episode. In the first $1e5$ steps, we use 512, while we use 900 from then on. The reason for choosing a smaller number at the beginning is that the agent is too random at the beginning. And a smaller number is good for the agent to learn how to stay on track without spending excessive time on the grass. Later this number was tuned up, because the agent has been good enough and a larger number will encourage the agent to explore more positions.

Training record:
The training process was shown in the following figure. It can be observed that the reward grew rapidly from the beginning, and reached kind of a steady state since step 380,000. The reward then oscillated between 400-800 and did not improve anymore. Therefore, we thought the training of the model has converged.

Figure 4: The record of training process, quantified by the average reward

4.3 Evaluate our PPO model

After the training converged, we evaluated the model by playing 10 episodes. The length of each episode is 900 steps. In each episode, the total reward was counted, and the occurrence of any out-of-track event (drive onto the grass) or out-of-boundary event (drive out of the playground) was recorded. The result is shown in the following table

Episode	1	2	3	4	5	6	7	8	9	10
Reward	798	854	750	694	860	867	320	167	493	8
Out-of-track	False	False	False	True	False	False	True	True	False	True
Out-of-boundary	False	False	False	False	False	False	False	False	False	False

In the 10 episodes, the car sometimes went out-of-boundary, showing the stability of our model. In about 40% percent episodes, the car went out-of-track. So we concluded that the success rate of the car completing a lap is 60%. In the successful episodes, the average reward is 770. Note that the max reward in a game is

1000 deducted by each step elapsed times 0.1. So the maximal possible reward is 910 given we have 900 total steps. Therefore, considering the success rate and the reward in the successful episodes, our model performs pretty good.

For the readers to understand the power of our agent, we attached three video samples in our submissions. The NormalLap.mp4 shows the car completing a lap of normal difficulty. The HardLap.mp4 shows the car completing a hard lap with multiple sharp turns and S turns. The Recover.mp4 shows the situation that the car went out of track but recovered itself back to the track, showing the robustness of our model.

4.4 Discussion

We got a feeling from this instance that a RL model is very unstable and hard to train. The target of the training is keeping the car on track but driving as fast as possible. But these two goals are intrinsically controversial, because the car is more prone to side slip when turning at a high speed. Although PPO is an excellent tool to train the model to go as fast as possible in general, we had totally no idea on how to teach the agent some sophisticated skills. For example, it recovers faster if you fully brake and resume when the car goes out of track, and drifting can let you brake less and pass a sharp turn faster.

From the evaluation we concluded that our current model has a satisfying performance, but there is still room for improvement. According to others' model [4], the training can be improved by stacking multiple frames, because a stacked frame sequence is like a smoothing filter and can provide the CNN more information in one single gradient decent. In our work, we use a single frame, but the literature [4] and the RL Baselines Zoo group [7] use a stack of four consecutive frames. Moreover, parallel computation can be adopted. Stable Baselines3 offers a feature of vectorized environments. With this feature, the model can be trained with the output from multiple parallel environments. Not only the training speed can be improved, but the output from independent environments can smooth the noisy observation and update the policy more accurately.

References

[1] Stable Baselines 3. *PPO*. 2022. URL: `https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html#notes`.

[2] Maximilian Brunner et al. "Repetitive learning model predictive control: An autonomous racing example". In: *2017 IEEE 56th annual conference on decision and control (CDC)*. IEEE. 2017, pp. 2545–2550.

[3] Schulman John et al. "Proximal Policy Optimization Algorithms". In: *arXiv* (2017).

[4] NotAnyMike. *Solving Car Racing with Proximal Policy Optimisation*. 2019. URL: `https://notanymike.github.io/Solving-CarRacing/`.

[5] OpenAI. *Proximal Policy Optimization*. 2018. URL: `https://spinningup.openai.com/en/latest/algorithms/ppo.html#id1`.

[6] Ugo Rosolia, Ashwin Carvalho, and Francesco Borrelli. "Autonomous racing using learning model predictive control". In: *2017 American Control Conference (ACC)*. IEEE. 2017, pp. 5115–5120.

[7] RL Baseline3 Zoo. URL: `https://github.com/DLR-RM/rl-baselines3-zoo/blob/1f9cfcfa3b4a12f374aef12d0cbc1hyperparams/ppo.yml`.