AI3603 2023 Project 1: Autonomous Driving Simulation Experiment Report

1st Yizhe Feng
SEIEE
Shanghai Jiao Tong University
Shanghai, China
yizhe.feng@sjtu.edu.cn

1st Yijin Chen
SEIEE
Shanghai Jiao Tong University
Shanghai, China
st.czzz@sjtu.edu.cn

1st Gonghu Shang
SEIEE
Shanghai Jiao Tong University
Shanghai, China
shanggonghu@sjtu.edu.cn

***Abstract*—This is report for AI3603 project 1: autonomous driving simulation experiment. We run SAC on four gym environments: highway, racetrack, intersection and parking to simulate different road environment. To make the car move as expected, we did a lot of work, including: reward engineering, abstract obs, hyperparameter tuning, algorithm improve and so on. You can find our code at https://github.com/Fizzfyz/AI3603_Project.**

***Index Terms*—Autonomous Driving, Highway, Reinforcement Learning, SAC**

I. Introduction

In this project, we run autonomous driving simulation experiment. Our task is implement reinforcement learning to control the vehicle move as fast as possible while ensuring safety to accomplish specified tasks such as tracking and parking.

A. Understanding the HighwayEnv

We use highway-env [**?**] as simulation environment. Our goal within highway, intersection, and race track scenarios is to move as quickly as possible while maintaining safety. The task in parking scenarios involves parking the vehicle at specific locations.

To understanding the goal Highway-env environment, we focus observation and action here. The details (reward, info and so on) about sub-environments will be described in respective sections.

1) observation: There are two types of observation we used in the project. Occupancy grid(for highway, intersection, and race track) and Kinematics(for parking). For Occupancy grid observation, variable obs is a three-dimensional array of 8*11*11. 8 stand for the number of feature of the obs.

> "features": ["presence", "on_road","x", "y", "vx", "vy", "cos_h", "sin_h"]

11*11 is the size of view grid of our vehicle . The feature of our vehicle is located in the center and others is nearby vehicle's feature. For Kinematics observation(parking), obs is a dict like:

> OrderedDict([('observation', array([0, 0, 0, 0, 0, 1])), ('achieved_goal', array([0, 0, 0, 0, 0, 1])), ('desired_goal', array([0, 0, 0, 0, 0, 1]))])

Each array stand for the features of the observation.

> "features": ["x", "y", "vx", "vy", "cos_h", "sin_h"]

2) action: The action has two components: [longitudinal action, lateral action]. Both actions is among [-1,1]. Longitudinal action will change the speed of the vehicle and lateral action will change the direction of vehicle. If we set longitudinal to false(for racetrack), the action only has one component: [lateral speed]. It means that the vehicle cannot change its speed, only its direction can be changed.

B. Algorithm

We use SAC [**?**] for the four environment. SAC means **Soft Actor-Critic**.It's an off-policy network. It features in the **maximum entropy** in the actor network. It use the flexible parameter α and an entropy regularization to control the randomness in the actor network and therefore balance **utilization** and **exploration** of the network. The target of SAC is to optimize the policy π:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_t r\left(s_t, a_t\right) + \alpha H\left(\pi\left(\cdot \mid s_t\right)\right)\right]$$

We've also compared SAC among different algorithms. We've considered DDPG [**?**], PPO [**?**], A3C [**?**] and TD3 [**?**]. We prefer **off-policy** algorithm, because we've learn the difference between Q-Learning and SARSA in Class. Considering that we're going to do the **ranking**, we decided to use off-policy algorithm to attain a **greedier** result. So we gave up PPO. Then, A3C is a **asynchronized** algorithm. We don't think it easy to implement, so we gave it up as well. Then we chose SAC among DDPG, TD3 and SAC, because we suppose that the flexible maximum entropy may make better balance of utilization and exploration, and finally helps **convergence**(and also because the SAC's paper is quite fresh(2018), of course).

We used the SAC version on OpenAI.[1] The SAC algorithm is shown in Fig **??**:

C. Experiment Environment Setup

All the code runs on Ubuntu 22.04, and we use conda to manage the Python environment. You can see the 'README.md' in the source code to setup the environment and run the experiment.

[1]https://spinningup.openai.com/en/latest

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1-\rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Fig. 1. SAC Algorithm

II. TASK1: HIGHWAY

A. Introduction

The goal of the highway is to drive as fast as possible while avoiding collisions. The configure of the environment is the same as eval_files.

We have learned about the composition of the reward by consulting the source code.

$$r = (-r_{collision} + 0.4 * r_{high\_speed} + 0.1 * r_{right\_lane}) * r_{on\_load}$$

Where $r_{collision}$ equals 1 if a collision occurs, otherwise 0; r_{high_speed} varies linearly when the vehicle's speed is between 20 and 30; r_{right_lane} depends on whether the vehicle is driving in the right lane and can take values of 0, 0.333, 0.667, or 1; and r_{on_load} is 1 if the vehicle is driving on the road, otherwise 0.

We can obtain the value of each sub reward from the info, laying the background for reward engineering.

B. Difficulty and Solution

We'll discuss our work on the highway in a problem-solution format. The methods we employed include vector environments, reward engineering, limiting max action, simplifying observations, change network structures, and tune hyperparameters, and so on.

1) Slow training speed: The original environment had a slow training speed, it takes about 30 hours to train 200K steps. After debugging, we found that most of the time was spent interacting with the environment. To accelerate the training process, we implemented **parallel environments** using the built-in vector environments in Gym. This approach accelerated our training speed to four times the original rate.

2) Off-road, spin in place and reverse: In the original environment, the car is easily off-road, leading subsequent steps in that episode meaningless. So we **end the episode** if the car is off-road, which solve this problem. Another problem is the car may spin in place or reverse to avoid collision and cheat reward. So I **limit the max value of lateral action to 0.2** to avoid excessive steering action. We also set the reward to 0 if there is no other cars in the observation. After these improvements, the car can move forward as we expect.

3) Struggle to learn the policy strategy: We find that the car is keeping away from the car ahead. It's speed is low and get a low reward. We hope the car to learn lane changing and overtaking.

We guess the observation is too complex for a MLP to understand. So we add a RNN to the network of SAC. But this didn't working. One idea strikes us. To some extent, the network employed to represent observations aims to extract features from observation. Can we manually reduce the dimension of observations to reduce the workload on the network, potentially improving performance?

We'll put the idea into practice. We tried several methods for selecting new observations and ultimately settled on the following. The new observations is 31-dim, consist of the following parts: the 'cos_h' and 'sin_h' of our car; the 5 bool vector to shown if the five lane around our car is on road; the 3*4 vector to represent the feature of the closest car ahead in each of the three lane around our car. We only choose the features of 'X', 'y', 'v_x', 'v_y'. If there is no cars in the lane, we set the features to zero; We also use 3*4 to record the information about the vehicle is behind and close(within 2 grids) our car, to avoid collision.

After simplifying observations, the car has learnt lane changing and overtaking.

4) Strike a Balance between Conservative and aggressive Strategies: We start reward engineering and fine-tuning. To prevent negative rewards resulting from collisions that prompt the car to drive off-road, we've set the reward to right lane reward. Simultaneously, we've adjusted the ratio between high-speed rewards and right lane reward within the general reward scheme. We've found that a higher ratio cause the car fearing collisions and maintaining a safe distance. Conversely, a lower ratio reduces the concern for collisions, leading to reckless driving and an increased likelihood of collision. Lowering gamma exacerbates the former, while raising it exacerbates the latter. Our goal is to strike a trade-off between these two factors to achieve a balanced outcome.

After many attempts, we find γ=0.99, and

$$r = 0.7 * r_{high\_speed} + 0.3 * r_{right\_lane}$$

get the best results.

5) Other Unresolved Problem:

i) **Unsteady action** The actions of the trained car were ultimately very unsteady, frequently exhibiting slight up-and-down movements. We attempted to address this by **adding penalties for angles of car** in the reward function, but this didn't work.

C. Shortcomings of Highway

After lots of attempts, we still haven't achieved satisfactory results. After analyzing crash videos and debugging, we believe there are certain shortcomings in the environment, which are preventing us from achieving satisfactory results.

1) Shortcomings of Observations: Firstly, there's some problem with observations. The rewards provided by the environment heavily rely on speed. The Reacting time to lane changes if there are cars in front is also depends on speed. However, we can't find information about the speed of our car in the observations provided by the environment. Our car can only judge whether to accelerate or decelerate based on the relative speed with other cars. But the speeds of other cars are random. In discrete situations, this isn't a problem since they can produce good outcomes by only maxing out longitudinal action.

2) Couldn't keep in lanes center: Unlike racetrack, the reward of highway don't related to the car's position relative to the lane center. This means the car can position itself between lanes, leading to confusion in determining the correct lane. Figure **??** illustrates a bad situation. The matrix in Figure **??** represents the on-road information. It indicates whether the car's position, as well as positions one and two lane widths above and below, are on the road. Logically, this situation should not occur regardless of which lane the car is in.

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Fig. 2. the reward curve of without Her(purple) and with HER(black).

3) Unavoidable Collision: In the environment, there are certain collisions that cannot be avoided. Here's an example: our car detects another vehicle ahead and attempts a lane change. However, after the car changes lanes, the vehicle ahead also switches to the same lane, resulting in a collision that is unavoidable.

D. Result and Conclusion

The highway environment exhibits the highest level of randomness among the four required environments. The most important work is simplifying the observations. Adjusting the reward and gamma to **strike a balance between aggressive and conservative strategies** was also a time-consuming task. Our result is shown in Figure **??**. We can get an average reward of nearly 14/20, with an average step length reaching 34/40. Although there remains a probability of collisions, we have done our best.

```
ep_ret  Mean: 13.9253, Std: 5.2112
ep_len  Mean: 34.77, Std: 10.5668
```

Fig. 3. Result of Highway

III. Task2: racetrack

A. Introduction

The goal of this task is to control our yellow car to drive along the track while avoiding collisions with the blue car driving on the same track.

In this environment, we're not allowed to modify the **linear velocity** of our car, which means the action space is actually one-dimensional. We can only modify the **lateral velocity** of our car by giving it lateral acceleration. And that is what "action space" stands for in this environment.

The **initial rewards** in this environment include action reward, lane-centering reward, collision reward and on-road reward. Driving on road and in the center of the road brings positive reward, while taking actions, colliding brings negative reward. If the car drives out of the road, the reward for the current step will be reset to zero, but the driving process will **NOT** truncate immediately. And that will cause problems while training. We'll talk about it later.

B. Problems in the Original Settings

We implemented the **SAC model** for this environment. We used the default hyper-parameters given by cleanRL and the original obs(size 8*11*11) to train our SAC model. It turned out that the agent was good at **cheating**, which means, the agent wasn't act as the ideal way yet still attained good rewards. After training, though we've tried different learning rate and different expected entropies, the agent still act like this(shown in Fig. **??**):

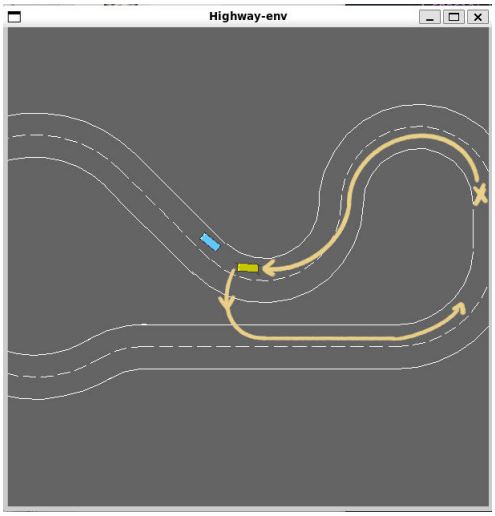


Fig. 4. the action taken by the agent during evaluation.

It's still explainable. When the agent drove to the bottom of the U-shape track, sometimes it will be very close to the blue

car. So if the agent kept on following the blue car, it would be very likely for the agent to **crash into** the blue car, which will make the **Q-value** there quite **low**. Moreover, driving out of the road at that point may result in some loss of rewards, but the subsequent potential rewards are greater. And that makes the **Q-value higher**, which caused the agent to drive out of the road. We'll argue about it here**??**

Therefore, there's much we can do towards the original settings:

1) the original obs is too **sparse**...If we use MLP, then we'll have to throw a $11 \times 8 \times 8 = 968 - dimensional$ vector into the MLP, with only quite a few of the elements conveys the useful information(we have only **2** cars here). So we decided to modify the **observation-space**.

2) the **rewards** and the **truncation settings** are also very unreasonable. On one hand, If we make our agent learn that collisions has lower Q-value than driving out of road, then the problem mentioned above will occur. On the other hand, if we let the agent learn that driving out of the road is more awful than collisions, then the agent may collide more in order to avoid driving out of the road, which is also unacceptable. So we're actually asking for a **trade-off** between **collision** and **driving out of bounds** here. We have to reconstruct the original reward and the truncation settings.

3) There's still another problem. The agent tends to take large lateral accelerations, which means the agent often does **snake-driving** on the road, or **shivers** intensively from time to time. In some cases it even **makes a U-turn** and drove in the opposite direction. We'll try to fix that.

C. Solutions

Let's first jump to the pleasant part, since the exploring process maybe too boring to read. We finally made the following improvements, which are proved to be effective.

1) **obs:** We obtain our obs manually. First of all, The 8-dimensional(short as 8-d in the context) vector that represents the **agent's state** is chosen. Then, we scanned the 5×5 grids around our agent, and pick the **"on-road"** feature out of the 8-d vectors. Finally, we scan the whole 11×11 obs, and choose the 8-d vector representing the **state** of the **blue car**, then concatenate them together to form a $8 + 5 \times 5 + 8 = 41$-dimensional vector as our new obs. If the blue car isn't in the 11×11 obs, we replace the 8-d vector with a 8-d zero vector. Then we throw away all the useless information and make the obs much more compact.

2) **reward-engineering and truncation settings**: In order to solve problem 2) and 3) mentioned in section **??**, we adopt new reward function when training, which is:

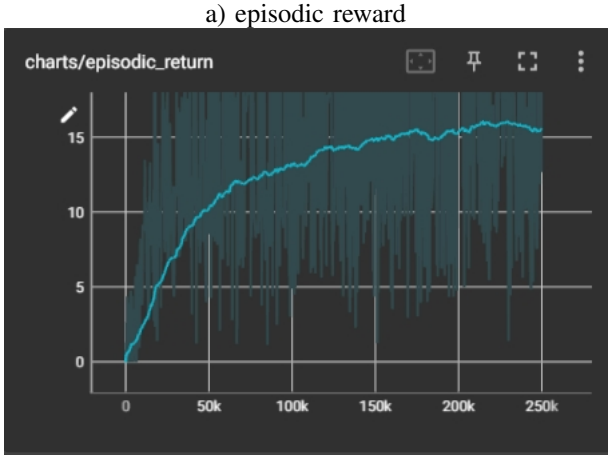
$$r_{new} = r_{on\_road} \times (r_{lane\_center} - 0.4 \times r_{action} - r_{collision})$$

Meanwhile, we truncate the driving process as soon as the agent drive out of bound. We did this to prevent the agent from receiving any future reward if it drives out of bound, and that successfully reduce the Q-value of the grids that are not on road.

3) Moreover, we decrease γ for training. We did this to make the agent "forgetful". We want our agent to "forget" the collision punishments properly so as to avoid acting like **??**.

After tuning other parameters, we finally got a good result. The agent would drive correctly and get sufficiently good results(shown in Fig. **??**):

a) episodic reward



b) scores when evaluating

```
ep_ret  Mean: 17.8932, Std: 6.7517
ep_len  Mean: 22.92, Std: 8.3877
```

Fig. 5. results

D. Constrained RL in Racetrack

In this section, We aim to further reduce collisions between agents based on the results obtained in section **??** We referred to the paper Worst-Case SAC [**?**]. In this paper, the authors introduced CVaR(Conditional Value at Risk) to represent the "cost" in RL problem. They introduced another critic net called "safety-Critic" to estimate the CVaR on the map, just like what critic network do in reward estimating. They also introduce a new concept called "risk-level", controlling the extent for the agent to take risks. Combine them together and we will form the whole algorithm.

The authors also made comparison with pure SAC-lagrange in the paper. He claims that WC-SAC is better than SAC-lag(Fig. 3):

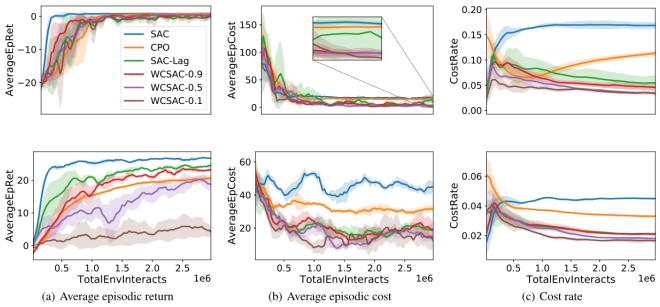


Fig. 6. The comparison between SAC and WC-SAC

So given the model parameter θ risk-level α, the limit CVaR d the step-wise obs s_t and action a_t the optimization problem in WC-SAC becomes:

$$\begin{aligned} \max_{\theta} \quad & episode\ return \\ s.t. \quad & CVaR_{\alpha}(a_t, s_t) \leq d, \ \ \forall t \end{aligned} \tag{1}$$

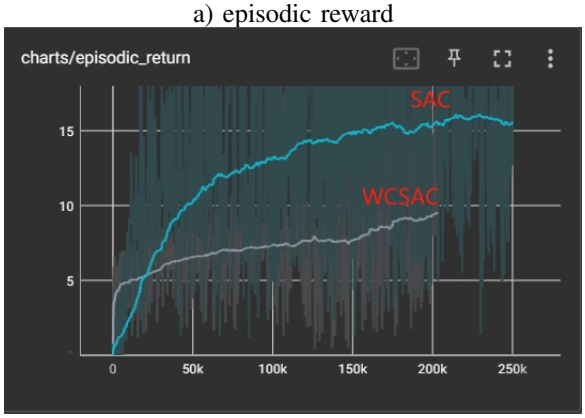
where $CVaR(s, a)$ is defined as:

$$CVaR_{\alpha}(s, a) = Q_{\pi}^{c}(s, a) + f_{PDF-CDF}(\alpha, s, a) \tag{2}$$

So WC-SAC will **maximize the reward** and **keep the "risk" low** at the same time. We hope this will help us solve the **collision** problem. Since We don't have the definition of "cost" in this environment, we have to design our own "cost" in order to make use of WC-SAC. We put collision, out-road, take actions into the cost. Then we started to tune the hyper-parameters.

The result isn't very pleasant. We again witnessed many forms of cheating given by our agent. The **best** of them has an average episode reward like(Fig.**??**):

a) episodic reward



b) scores when evaluating

```
ep_ret  Mean: 11.3813, Std: 4.2389
ep_len  Mean: 20.03, Std: 7.1616
```

Fig. 7. results of wc-sac

and the agent drives like(Fig. **??**):

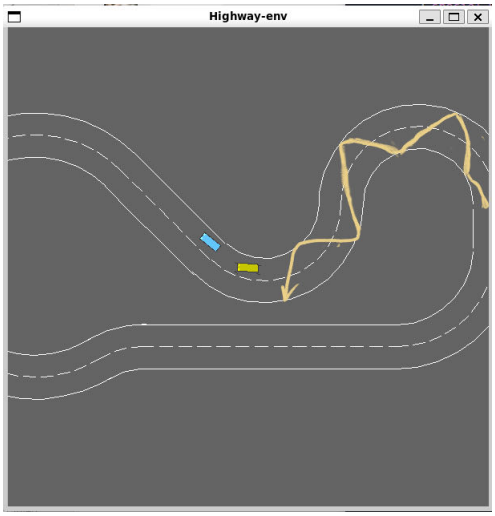


Fig. 8. The way that the agent drive in wc-sac

Well, at least it means that the agent try to do snake-driving to avoid getting any closer to the blue car, thus reduce collision. We've compared the collision times between WC-SAC and SAC version and finds out that the WC-SAC version agent collides about 35/100 times, while the SAC version collides about 50/100 times, though with higher reward.

E. Attempts that Are Proven to Be Useless

For all the results shown in section **??** and **??**, we've tried plenty, plenty of hyper-parameters around one single improvement, including both SAC and WC-SAC.

Fig. 9. The checkpoints that we've saved

As you can see, there are totally 163 checkpoints for racetrack saved in my computer. 4 to 5 hours are required to train one. So that's almost 700 hours in total...

Plus, the agent is extremely good at cheating. Here are 2 other ways to cheat that we have faced with(Fig. **??**):

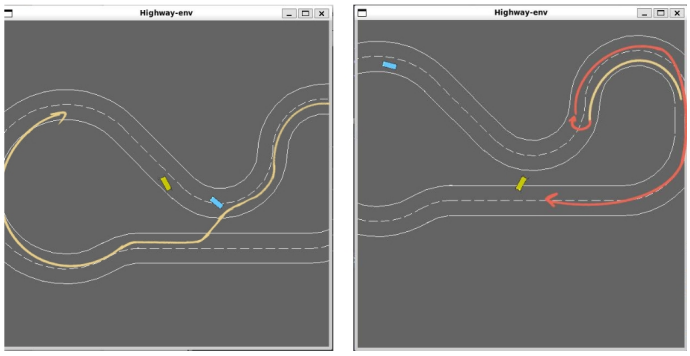


Fig. 10. ways to cheat

Punishing out-road by giving **negative reward** and truncate the process at the same time does no good. I set the out-road reward to be -1, -10 and even -100, and it turned out that out-road reward = 0 is enough to constrain the agent on road, while negative reward will cause frequent collisions.

Punishing actions **inappropriately** by either adding the "cost" or reducing the "action reward", or simply make a constraint to the max_action variant is dangerous. If the punishment is too benevolent, then the agent will either do **snake-driving** or simply **make a sharp U-turn**(shown in Fig **??** left). If its too strict, then the agent will not be able to pass the U-shape track and will drive like Fig **??** right

Behavior cloning is hard to implement in racetrack, because we can only control the **lateral acceleration**. It's hard for human controller to race by only controlling acceleration.

We've also tried **PPO** on race track, and it didn't show any advantages over **SAC**.

F. conclusion and ablation study

We'll **draw our conclusion** first. In this environment, I have to mention that, because there are loads of ways to cheat, so there's **no plug-and-play algorithm** here. What's worse, because the cheating routes don't have collision problems, if we do nothing towards the original settings, then the agent will get **more rewards** if it's cheating rather than driving as we wish. So our improvements in section **??** are necessary(We'll illustrate that later). No matter what algorithm is implemented, we have to modify the original settings to find a **elaborate balance** between those factors in order to make the agent act as we wish.

Moreover, any improvement towards the algorithm will break the balance to some extent, and may cause the agent to fall into another weird **cheating logic**, as we've shown in the WC-SAC case.

In all, we suppose that the key to solve the racetrack problem is to modify the settings to find the **"balance"**, rather than finding new powerful algorithm to improve it. Plus, pure SAC is good enough already.

Then we did some **ablation study** to prove that, the three key improvements that we proposed in section **??** are all necessary. And it turned out that any absence of the three improvements will cause the agent to drive like:

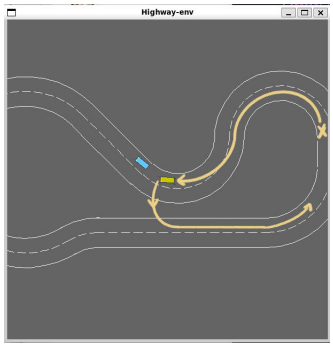


Fig. 11. the action taken by the agent during evaluation.

and the reward during evaluation is like:

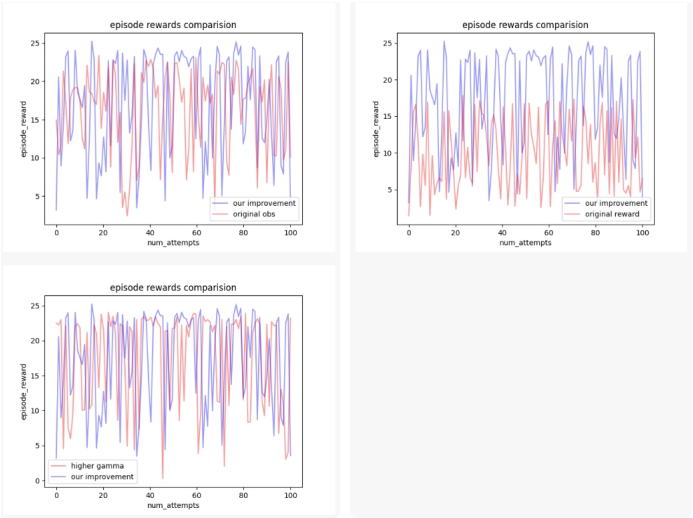


Fig. 12. reward comparison

It shows that using original reward will lead to **unfitting**, while the other two will lead to **overfitting**. And that supports the correctness of our improvements.

IV. TASK3: INTERSECTION

In this task, the agent is expected to pass through the intersection safely and quickly(Fig. **??**).

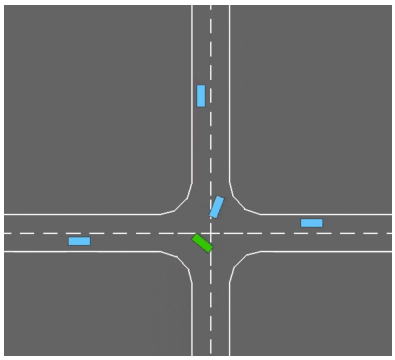


Fig. 13. Intersection

A. DQN and related methods

The original action space is continuous. If we discretize it to discrete action space, it becomes smaller and iterable. Therefore we tried transforming the continuous action space into a discrete action space and trained the agent using DQN (Deep Q-Network). Here, we will document a useful trick employed during the training process with DQN.

1) A useful trick: As mentioned earlier, a discrete action space is iterable. An intuitive idea is to let the agent traverse the action space and select the action with the highest reward. With this approach, the agent can quickly find the endpoint. This approach improves training efficiency by reducing a lot of useless exploration(actions with low rewards have already been identified and excluded during the iteration process)(Fig. **??**).

```
if global_step < args.learning_starts:
    rewards_cp = {}
    for action in range(envs.action_space.n):
        env_copy = copy.deepcopy(envs)
        obs_cp, reward_cp, done_cp, truncated_cp, info_cp = env_copy.step(action)
        rewards_cp[action] = reward_cp
    best_action = max(rewards_cp, key=lambda x: rewards_cp[x])
    actions = np.int64(best_action)
```

Fig. 14. Traverse the action space and calculate rewards

However, we abandoned DQN because the DQN model is prone to getting stuck in local optima and requires meticulous tuning.

B. SAC and related methods

We use **SAC model** to train the agent, with some additional methods:

- Action Mask
- Reward Engineering
- Time Limits Awareness
- Predefine Trajectories
- Ban lateral acceleration

1) Action Mask: Since the action space is not limited, the agent can take any action when driving. However, before the agent arrives the intersection, according to traffic rules, it should not turn to other lanes. One method is to develop corresponding reward functions to enable the agent to learn this constraint. However, the training cost is very high, and it is extremely easy to cause the model to not converge. We can implement this behaviour using action masks:

$$\begin{aligned} action[1] = action[1] * \\ (not(obs[1][5][4] == 1 \text{ and } obs[1][5][6] == 0)) \end{aligned} \tag{3}$$

$obs[1][5][4] == 1$ represents that the left side of the agent is on_road. $obs[1][5][6] == 0$ represents that the right side of the agent is off_road. Then, the agent would only go straight until reaching the intersection.

2) Reward Engineering: The action mask method is only suitable for the first straight lane segment. After crossing the intersection, it is necessary for the agent to take lateral actions to keep staying on road as shown in Fig. **??**.

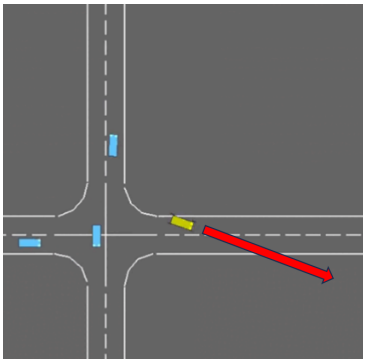


Fig. 15. If the agent does not take later actions, it would be off road

We can design rewards based on the angle of the vehicle to teach it to move straight along the direction of the road:

$$\begin{aligned} reward\_angle = &(90 - angle)/90 * \\ &(np.sum(obs[1][5]) == 11) \end{aligned} \tag{4}$$

$np.sum(obs[1][5]) == 11$ is also a mask to ensure that $reward\_angle$ only takes effect after the car turns left/right.

3) Time Limits Awareness: Intersection is a time limited scenario. According to the paper **Time Limits in Reinforcement Learning** [**?**], if the observation(obs) includes infomations about the remaining time, the agent will take some risky actions at the end of time for higher rewards. It is easy to verify this idea by adding the remaining time to the origin obs.

4) Predefine Trajectories: One motivation to use this method is that, it takes a lot of time for the agent to randomly explore the endpoint. The roads in this Intersection scenario are very narrow, making it easy to go off-track or have accidents. The agent can hardly make any mistakes to reach the finish line. Similar to the idea of Imitation Learning, we can predefine a series of trajectories that lead the agent to the destination. For example, here is a feasible action_list that lead the agent to turn right:

$$\begin{aligned} actions\_right = [\\ &[0, 0], [0, 0], [0, 0], [0, 0], \\ &[0, 0.9], \\ &[0, 0], [0, 0], [0, 0], [0, 0] \\ ] \end{aligned} \tag{5}$$

Drawing inspiration from data augmentation, we can add a bit of noise to each action in the action_list:

$$\begin{aligned} noise = \\ np.random.normal(mean, std\_dev, len(action)) \end{aligned} \tag{6}$$

In the early stages of training, we can have the agent take these pre-defined actions, instead of exploring randomly. Experiments have shown that this method can enable agents to quickly learn how to turn right(although the success rate is relatively low).(Fig. **??**)

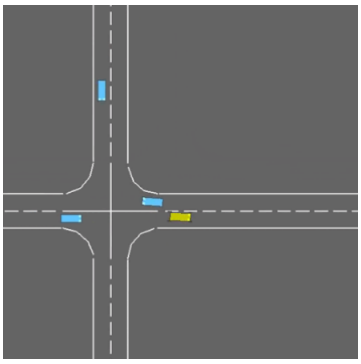


Fig. 16. Agent learns how to turn right

5) Ban lateral acceleration: In the intersection scenario, there are three endpoints(left, right and straight). During training, even if we have the agent to fully explore these three different endpoints, it still tends to converge to one single endpoint(going straight). Lateral acceleration complicates the vehicle's motion, and going straight to pass the Intersection does not require lateral acceleration. Hence we can ban lateral acceleration.

$$action[1] = action[1] * 0 \tag{7}$$

The experimental results indicate that this method enables the agent to learn to pass the Intersection faster and achieve higher rewards compared to other methods. Therefore, this model is used for the final evaluation(Fig. **??**, Fig. **??**).

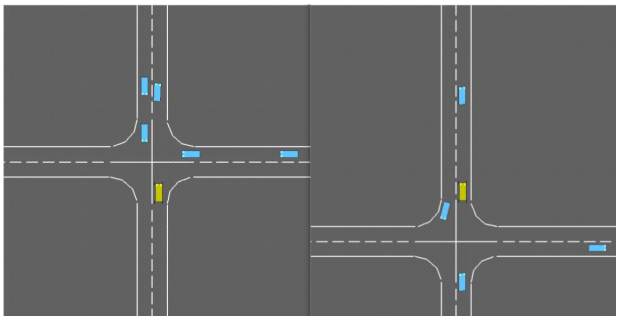


Fig. 17. Agent learns how to pass intersection

```
Moviepy - Done !
Moviepy - video ready /home/fizzthinkplus
99.mp4
ep_ret  Mean: 7.8711, Std: 2.4026
ep_len  Mean: 8.52, Std: 2.6626
```

Fig. 18. Results of Intersection

V. TASK4: PARKING

A. Introduction

The goal of this task is to control the vehicle to arrive the parking spot. The configure of the environment is the same as eval_files.

We have learned about the composition of the reward by consulting the source code. Different from other goal-conditioned scenarios, the reward structure for parking is consistent with HER (Hindsight Experience Replay) [**?**].

$$r = -||s - s_g||^p_{W,p} - r_{collision}$$

where $p = 0.5$, $r_{collision} = -5$, $s = [x, y, v_x, v_y, \cos\phi, \sin\phi]$, $s_g = [x_g, y_g, 0, 0, \cos\phi_g, \sin\phi_g]$. We use weighted P-norm to calculate state differences, where p = 0.5, and W represents the weights for each dimension [100, 100, 5, 5, 1, 1]. It means that the closer to the goal, the higher the reward.

B. Implement HER

HER (Hindsight Experience Replay) is a technique used in environments with sparse rewards. It modifies the traditional reinforcement learning framework by leveraging hindsight information from failed attempts.

To use HER in a sparse reward setting, two steps need to be taken. Firstly, modify the reward to be based on the distance from the goal, which is already the form of reward used in the parking scenario. Secondly, record the observations, next observations, actions, and done status for each step of the episode. If the episode fails, select a new goal and recalculate observations, next observations, and rewards for each step, storing them in the replay buffer. For the new goal, we choose the penultimate position along the trace(the last point goes out of bounds). Adjusting observations and next observations only requires change the goal information. As for the reward adjustment, refer to the original reward structure, replacing the target with the new goal.

C. Tuning Hyper-parameters

When it comes to Hyper-parameters tuning, we focus on four key parameters: learning rate (where both actor and critic share the same rate), gamma, buffer size, and batch size.

Starting with gamma, as our environment penalizes going out of bounds and we aim to avoid such failures, we need to consider long-term rewards. Therefore, we set gamma to 0.9, 0.95, and 0.99, running the program separately for each. Ultimately, we found that 0.99 yielded the best results.

Regarding the learning rate, after testing values of 0.001, 0.0005, and 0.0001, we observed that 0.001 led to the most optimal performance.

As for buffer size and batch size, we concurrently increased and decreased the two values, eventually determining that a buffer size of 50,000 and a batch size of 256 produced the best result.

D. Result and ablation study

The parking environment is relatively simple. After implementing HER and conducting some basic parameter tuning, we've achieved a success rate of over 98% and the evaluation result is shown as Figure **??**. By the way, the final evaluation results in rewards ranging between -6 and -7 because of significant randomness of environment.

We also did some ablation study. Figure **??** showcases the reward curves without and with HER. It's evident that the curve converges more rapidly whit HER. (By the way, it's possible that our misuse of TensorBoard caused the initial graph to appear sparse.)

```
ep_ret  Mean: -6.492, Std: 2.2921
ep_len  Mean: 18.05, Std: 3.7239
```

Fig. 19. Results of Parking

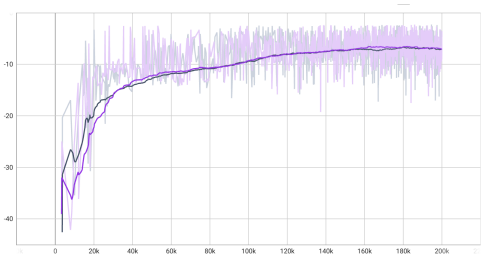


Fig. 20. the reward curve of without Her(purple) and with HER(black).

VI. Conclusion

In this project, we explored four autonomous driving scenarios: highway, racetrack, intersection, and parking. We tried using PPO, DQN, and SAC, and ultimately selected SAC to train our model. We also tried a variant of SAC: Worst Case SAC. To accelerate training, we employed vectorized environments and the HER algorithm. Additionally, we applied numerous techniques to address challenges in different scenarios, including: modify obs, modify reward, modify hyperparameters, reduce maximum acceleration value, mask actions, and predefine trajectories. We made numerous attempts to train the agent and the agent could get relatively high rewards.

Acknowledgment

Thanks for the guidance from Teaching Assistant Jingtian Ji and Teacher Yue Gao!

References

[1] Leurent, Edouard, et al. "An Environment for Autonomous Driving Decision-Making" GitHub, GitHub repository, 2018

[2] Haarnoja, Tuomas, et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor." International conference on machine learning. PMLR, 2018.

[3] Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." arXiv preprint arXiv:1509.02971 (2015).

[4] Schulman, John, et al. "Proximal policy optimization algorithms." arXiv preprint arXiv:1707.06347 (2017).

[5] Fujimoto, Scott, Herke Hoof, and David Meger. "Addressing function approximation error in actor-critic methods." International conference on machine learning. PMLR, 2018.

[6] Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." International conference on machine learning. PMLR, 2016.

[7] Yang, Qisong, et al. "WCSAC: Worst-case soft actor critic for safety-constrained reinforcement learning." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 35. No. 12. 2021.

[8] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, Petar Kormushev "Time Limits in Reinforcement Learning" ICML 2018

[9] Andrychowicz, Marcin, et al. "Hindsight experience replay." Advances in neural information processing systems 30 (2017).