**1.Methodology Introduction:**

For both austria and circle_cw maps, I used a td3 model to train. In our assignment 4, we used the td3 model to play an Atari game, Car Racing, it was a tremendous success for me. Reaching 800+ points in all the seeds at the demo day, even 904 points in maximum score. As a result, td3 model would be a best choice for me to work on this final project. Besides, i thought that both racecar and Car Racing were racing games, so they were kind of similar in reinforcement learning.
Last but not least, the reason why choosing td3 is that td3 is a preferred model for continuous action space.
TD3 (Twin Delayed Deep Deterministic Policy Gradient) is an advanced reinforcement learning algorithm designed for continuous action spaces. There are some key features such as Twin Q-functions, Delayed Policy Updates, Target Policy Smoothing, Deterministic Policy. This algorithm is particularly suitable for continuous control tasks like autonomous racing because it provides stable learning and efficient exploration in continuous action spaces.

**2.Experiment Design and Implementation**
The training process consists of initial exploration phase (first 10,000 steps), main training phase with iterative policy improvement, regular evaluation every 500 steps, automatic saving of best models based on evaluation performance.
I used the stable baseline3 package which has full pre-defined models. I can just setup my environment and use the model sb3 offered directly. The environment initialization is done by importing the class RaceEnv from racecar_gym/env. I also defined a custom CNN architecture and frame stacking for better training result. To ensure the adequate exploration, the OU noise is important, it could help to train a more stable model.
In the implementation of RaceEnv, I also did some modification, especially in step function, the modification was mainly about reward shaping, note that we could get some information via variable info returned from env.step(), I utilized these information such as 'progress', 'lap', 'velocity', 'wall collision', 'opponent collisions', 'wrong way', 'acceleration', 'dist_goal', 'obstacle' to modify the reward returned from the environment. For example, counting the progress reward by info['progress']*5.0 to encourage the agent move forward.

**Neural Network Architecture:**

```
Actor network architecture:
Actor(
  (features_extractor): CustomCNN(
    (cnn): Sequential(
      (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
      (1): ReLU()
      (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
      (3): ReLU()
      (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
      (5): ReLU()
      (6): Flatten(start_dim=1, end_dim=-1)
      (7): Linear(in_features=3136, out_features=512, bias=True)
      (8): ReLU()
    )
  )
  (mu): Sequential(
    (0): Linear(in_features=512, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=2, bias=True)
    (5): Tanh()
  )
)
```

```
Critic network architecture:
ContinuousCritic(
  (features_extractor): CustomCNN(
    (cnn): Sequential(
      (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
      (1): ReLU()
      (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
      (3): ReLU()
      (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
      (5): ReLU()
      (6): Flatten(start_dim=1, end_dim=-1)
      (7): Linear(in_features=3136, out_features=512, bias=True)
      (8): ReLU()
    )
  )
  (qf0): Sequential(
    (0): Linear(in_features=514, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=1, bias=True)
  )
  (qf1): Sequential(
    (0): Linear(in_features=514, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=1, bias=True)
  )
)
```

**TD3 Algorithm Parameters:**

```
model = TD3(
    buffer_size=100000,    # Replay buffer size
    learning_rate=0.001,   # Learning rate
    batch_size=256,        # Batch size for updates
    train_freq=1,          # Update frequency
    policy_delay=2,        # Policy update delay
    learning_starts=10000, # Initial exploration steps
    target_policy_noise=0.2,  # Target action noise
    target_noise_clip=0.5,    # Noise clipping range
    gamma=0.98             # Discount factor
)
```

**Exploration Parameters:**

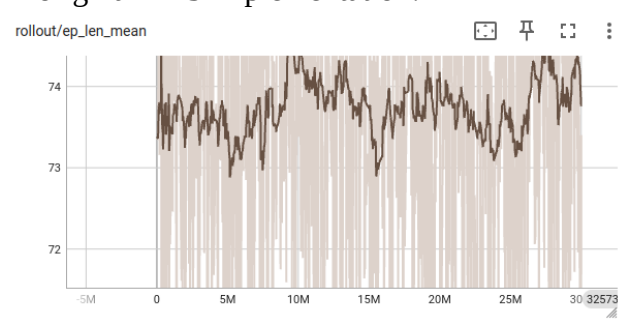
```
action_noise = OrnsteinUhlenbeckActionNoise(
    mean=np.zeros(n_actions),
    sigma=np.array([0.2, 0.1]),  # Different noise levels for steering/acceleration
    theta=0.15             # Mean reversion rate
)
```

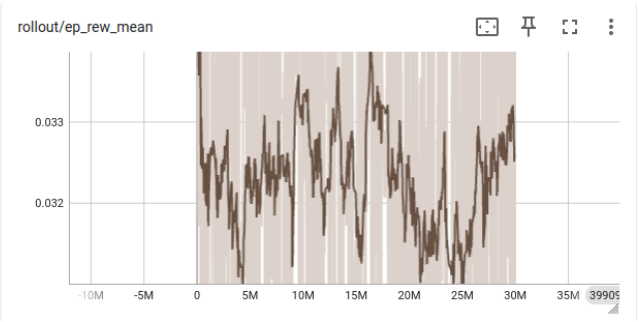
**List of packages, tools, or resources used:**

gymnasium
numpy
stable_baseline3
torch
RTX 4090 GPU
Visual Studio Code

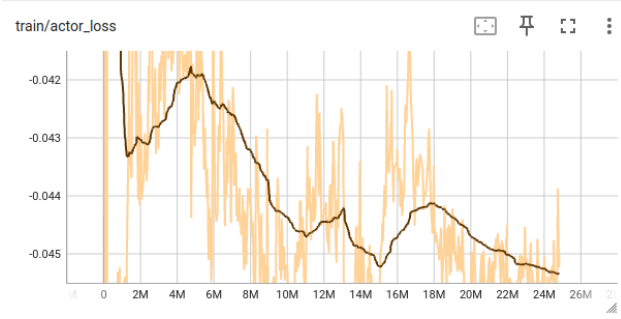
**Method Comparison and Evaluation**

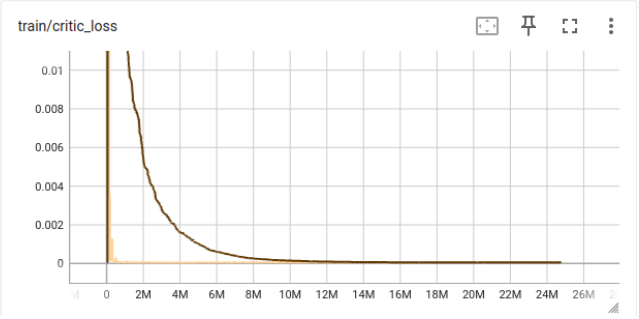
In original TD3 implementation:



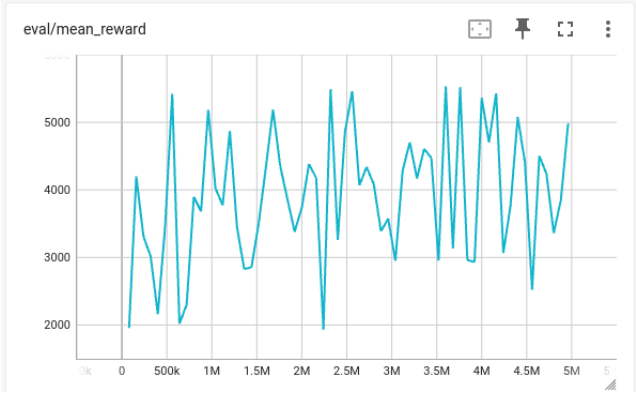

the reward didn’t arise well, check the actor and critic loss for verification.

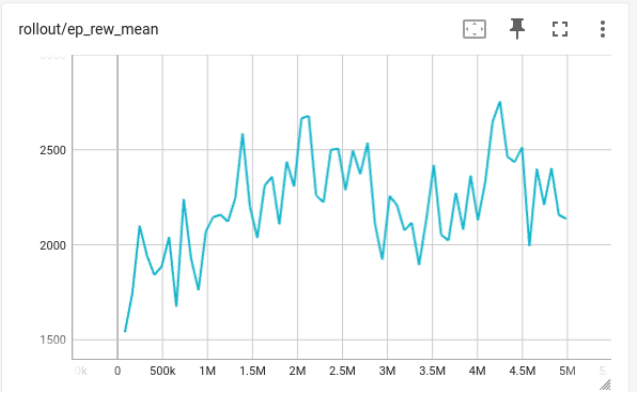



Both loss have converged, I think that maybe the model traps into local optimal, lacking of exploration.

After increase some noise and reward functions:




the reward increase well.

The reward function is shown below:

```
modified_reward = reward if reward is not None else 0.0

# Stronger progress reward
lap_multiplier = info['lap']  # Reward scales with laps
progress_reward = info['progress'] * 5.0  # Increased from 2.0
modified_reward += progress_reward * (1 + 0.3 * lap_multiplier)  # Incr

velocity = info['velocity']
forward_speed = velocity[0]  # Using x-axis velocity as forward speed
angular_speed = velocity[5]  # Using angular z velocity for rotation
# Stronger speed rewards
speed_reward = forward_speed * 1.0
rotation_penalty = -abs(angular_speed) * 0.6
modified_reward += speed_reward + rotation_penalty

# Much stronger penalties
if info['wall_collision'] or len(info['opponent_collisions']) > 0:
    modified_reward -= 5.0  # Increased from 2.0
if info['wrong_way']:
    modified_reward -= 6.0  # Increased from 3.0
# Add acceleration reward/penalty
acceleration = info['acceleration']
if forward_speed > 0:
    acceleration_reward = np.clip(acceleration[0], -1, 1) * 0.5
    modified_reward += acceleration_reward

# Add progress delta reward
cur_progress = info['progress']
progress_delta = cur_progress - self.last_progress
self.last_progress = cur_progress
if progress_delta > 0:
    modified_reward += progress_delta * 3.0
```

Taking progress, lap, velocity, wall_collision, acceleration into account.
And give some reward according to specific tasks.

The key observations and insights are the noise. After increasing the noise to make model explore more, it shows a better result.

**Challenges and Learning Points:**

The biggest challenge is that it shows steering in initial game and results in collision, no matter how do I adjust the progress reward and collision reward in the yaml file and reward function.

The learning point is that taking rotation penalty into account, which is lied in velocity[5].
The key point is about if it tries to turn left or right, there will be some penalty.

**Future Work:**

**Advanced Architecture Enhancements**

- Implement attention mechanisms to help the agent focus on important visual features
- Add recurrent layers (LSTM/GRU) to better handle temporal dependencies
- Explore vision transformers for more efficient visual processing

**Reward Function Refinements**

- Design dynamic reward scaling based on learning progress
- Implement inverse reinforcement learning to learn from expert demonstrations
- Add multi-objective optimization for balancing speed and safety

**Multi-Agent Racing**

- Extend to multi-agent scenarios with competitive and cooperative racing
- Study emergent racing strategies in multi-agent environments
- Develop robust collision avoidance in competitive scenarios
- Investigate team racing strategies