CS181 Artificial Intelligence I Final Report: TORCS Based on DDPG

Jiadong Tu Peilin He Wentao Wang Heyang Li Qingyi Lu

Abstract

In this project, we implement reinforcement learning algorithms and evaluate them by their performances of playing the open racing car simulator. We combine Q learning in class and deep neural network to implement deep deterministic policy gradient, and it plays well in most tracks.

1. Introduction

This project is based on TORCS[2] (the open racing car simulator), which is a highly portable multi platform car racing simulation. It can be used as ordinary car racing game, as AI racing game and as research platform. It runs on Linux, MacOSX and Windows. The source code is licensed under the GPL ("Open Source").

TORCS features many different cars, tracks, and opponents to race against. It is possible to drive with the mouse or the keyboard. Graphic features lighting, smoke, skid marks and glowing brake disks. The simulation features a simple damage model, collisions, tire and wheel properties (springs, dampers, stiffness, ...), aerodynamics (ground effect, spoilers, ...) and much more. The game play allows different types of races from the simple practice session up to the championship.

Here are some basic elements of our reinforcement learning agent:

- Action: steer, accelerate, brake
- State: angle, trackPos, speed, damage...
- Reward: a function of the mentioned state elements
- Evaluation: travelled distance, time to the end, car damage

The basic reinforcement learning model is Q-learning. Since car racing is a continuous problem, we discretize the basic elements of our agent, like the action and the state. However, discretization leads to a large number states, so the model could be imprecise. Thus, we combine neural network and introduce deep deterministic policy gradient to deal with the problem.

Figure1 shows the visualization of the game and the perspective of the player.

Figure 1. Visualization of the Game

2. Methods

We mainly implement a deep Q learning algorithm named DDPG[1].

DDPG, a policy learning algorithm aimed at continuous problems, combines deep learning and deterministic policy gradient. Given the probabilistic distribution function $\pi_\theta(s_t|\theta^\pi)$, sample the action to get the certain best action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$. Such model-free policy saves computation sources. DDPG simulates policy function μ and function Q by feed-forward neural networks.

In the training process, we weigh the trade-off over exploration and exploitation. In order to explore the potential better policy, random noise is introduced by Uhlenbeck-Ornstein random process. This step is called behavior policy β, as shown in figure2.

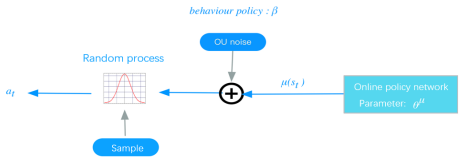


Figure 2. Behavior policy

The Q function is defined as the expected reward $r(s_t, a_t)$ at state s_t, take action a_t and continue carrying

out policy μ by Bellman Equation:

$$Q^{\mu}(s_t, a_t) = E[r(s_t, a_t) + \gamma Q_{\mu}(s_{t+1}, \mu(s)t + 1))]$$

An *actor-critic* architecture (figure3&4) is implemented. It stores the transition model in the replay memory buffer as the data set to train the online network. For each iteration of the mini-batch, N transition samples are randomly selected.

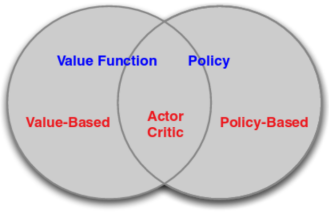


Figure 3. Actor-Critic Architecture

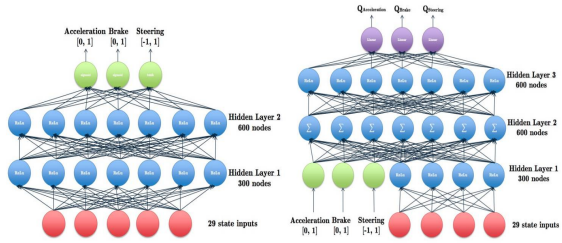


Figure 4. Actor-Critic Architecture

We use back-propagation to train the weights of the Q-network, namely θ^{Q}. The label/target-Q-value y_i is calculated as

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta')$$

, then we use Mean Squared Error between target Q value and computed Q value as the loss function of Q-Network.

To train thee policy netowork, we need to calculate the policy gradient. It is denoted as the gradient of the performance objective function J to θ^{μ}. According to Monte-carlo method, we can get an unbiased estimation of the gradient:

$$\nabla_{\theta^{\mu}} J_{\beta}(\mu) \approx E_{s\sim\rho^{\beta}}[\nabla_a Q(s, a|\theta^{Q})|_{a=\mu(s)} \cdot \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})]$$

$$\approx \frac{1}{N}\sum_i (\nabla_a Q(s, a|\theta^{Q})|_{s=s_i, a=\mu(s)} \cdot \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s=s_i})$$

Finally, use back-propogation to update policy network, namely θ^{μ} and use running average to soft update the parameters of the target actor/critic network.

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

Figure5 indicates the structure of our algorithm.

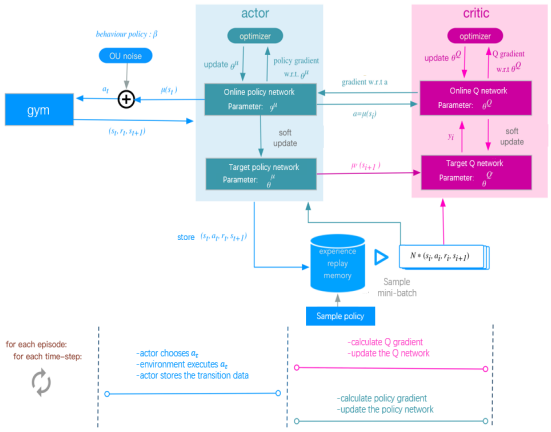


Figure 5. Complete DDPG Algorithm

3. Implementation Details

For Actor network, we use two hidden layers with size 300 and 600. For Critic network, we use three hidden layers with size 300, 600 and 600. ReLu is used as the activation function of some hidden layers. We tested the effect of BatchNormalization after ReLu layers, but the we can hardly see the improvement in performance.

Angle, track, trackPos, speedX, speedY, speedZ, wheelSpinVel and rpm are chosen as states, which composes 29 dimensional state vectors. Accelerate, steer and brake compose 3 dimensional action vectors.

For training, we set γ to 0.99, τ to 0.001 to guarantee soft update. To do exploration, ϵ is set to 1.3 in the beginning and is reducted by $\frac{2}{100000}$ every time step. The Replayer buffer size is set to 100000. Initial learning rate of Actor and Critic network are 0.0001 and 0.001 respectively. We train the model for 1500 epochs on GPU with batch size 32.

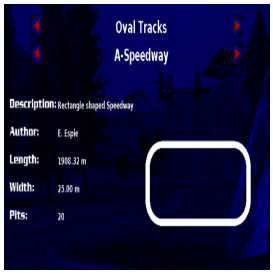
4. Performance

The training results of our agent on different tracks are shown in table1. The training process takes 1500 epochs. If the agent can reach the destination, then cost time is the evaluation; otherwise, the distance our agent can go will be the evaluation. From the table, it is intuitively shown that our DDPG agent works well for most tracks.

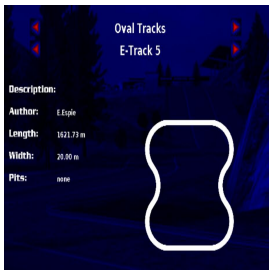
Map	Difficulty	Time(s)	Distance	Damage
A-speedway	Easy	41.59	/	0
E-track5	Median	42.61	/	0
CG-track	Hard	130	/	0
Alpine1	Extreme Hard	/	421.357	0.2

Table 1. Training result

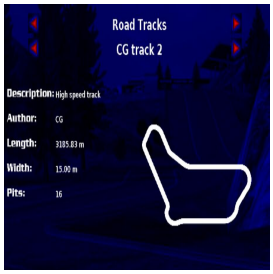
The details of racing maps are shown is figure6.



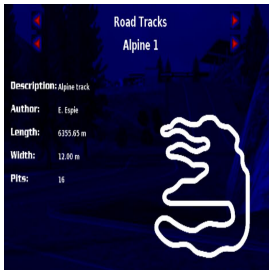
(a) A-speedway



(b) E-track5



(c) CG-track2



(d) Alpine1

Figure 6. Map Info

5. Thoughts and Improvement

5.1. Reward Function Adjustment

At first, we consider three main indicators: V_x (the speed parallel to the direction of the car), $angle$ (the angle between the car direction and the direction of the road axis) and $trackPos$ (the distance between car and road axis).

$$reward = V_x * cos(angle) - abs(V_x * sin(angle))$$

$$-abs(V_x * trackPos)$$

However, our agent would drift when turning the direction. So we add V_y (the speed perpendicular to the direction of the car) in the reward function. This solves the problem of drifting, but the speed when taking a sharp turn is still too fast.

$$reward = V_x * cos(angle) - abs(V_x * sin(angle))$$

$$-abs(V_x * trackPos) - abs(V_y * trackPos)$$

To deal with the problem, we set different ϵ for different actions, especially for braking. Specifically, ϵ_{brake} is increased to 1.5 and other functions remains $\epsilon_{action} = 1.3$. This time our agent can turn at a proper speed.

5.2. Advantages of DDPG

DDPG is a kind of deep Q Network learning. We utilize this algorithm because of its following advantages.

Actor-critic is a process of iterative training strategy network and Q network through the interaction of environment, actor and critic under the condition of cyclic episode and time steps.

The feed-forward neural network is used to simulate the strategy/policy function and Q function, and trained by the deep learning method. It is proved that the nonlinear simulation function in reinforcement learning methods is accurate, high-performance and convergent.

The DDPG actor first stores the transition data into the experience replay buffer, and then randomly samples the mini batch data from the experience replay buffer during training, so that the sampled data can be considered irrelevant. Otherwise, the network may overfit and be hard to convergent.

The use of target network and online network makes the learning process more stable and ensures the convergence.

6. External Resources

The implementation of reinforcement learning algorithms is done by ourselves, but we use open source TORCS as our game environment. The algorithm details are implemented by pytorch.

6.1. Python Library

- Pytorch
- Numpy

6.2. Tools

- Gym-torcs Environment: https://github.com/ugo-nama-kun/gym_torcs
- Snake Oil (Library interacting with Torcs): http://scr.geccocompetitions.com/

References

[1] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[2] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Simulated car racing championship: Competition software manual, 2013.