Self driving cars in TORCS using Deep Deterministic Policy Gradient

Adrian Andersson [1], Joel Paulsson [2], Samuel Svensson [3]

Abstract
This paper presents a research project where an autonomous driver, or agent, was implemented for the AI-racing game TORCS by using *Deep Deterministic Policy Gradient* (DDPG). The implementation was made in Python using two main frameworks; TensorFlow and Keras. DDPG is a method that can handle larger state spaces as in this case and it makes use of an Actor-Critic architecture of neural networks. The behaviour of the agent is ultimately defined by the reward function and in this context the agent should stay on track and try to finish the race as fast as possible with a stable training process. Therefore, the reward function rewards high longitudinal speed and penalizes driving off the road.
At this point the agent is trained for 150 episodes which is enough for it to complete easier tracks, meaning tracks with few and long corners but it is not able to finish complex tracks with more winding roads.

Source code: *https://github.com/AddeAndersson/TNM095*

Video: *https://www.youtube.com/watch?v=V_kxQbAES2E&feature=youtu.be*

Authors
[1] *Media Technology Student at Linköping University, adran117@student.liu.se*
[2] *Media Technology Student at Linköping University, joepa811@student.liu.se*
[3] *Media Technology Student at Linköping University, samsv787@student.liu.se*

Keywords: Deep Deterministic Policy Gradient — DDPG — TORCS — autonomous driving — AI racing

Contents

1. Introduction

The objective of this project is to explore how an agent can be implemented and trained to race in *The Open Racing Car Simulator* (TORCS) using reinforcement learning and *Deep Deterministic Policy Gradient* (DDPG). This project is inspired by a blog post by Lau [2016].

TORCS is an open-source 3D racing game and research platform that can be used to implement and race your own computer controlled drivers but also by using the keyboard Wymann [2020].

Self driving cars are more popular than ever and a lot of research is going into both fully autonomous driving and driver support. Both are connected and can help reduce the number of fatal traffic accidents, improve fuel economics, and reduce traffic congestion and therefore lower emission. Self driving cars are based on AI techniques which takes sensor input from the vehicle and outputs actions determined by the agent. Testing different configurations and techniques in real world scenarios can be both expensive and dangerous, hence simulations in a platform like TORCS can be a good tool for tuning the model.

2. Theory

2.1 TORCS

TORCS consists of multiple tracks and various agents to choose from. There are options to control the steering, braking, acceleration, and gear shifting in the game. A custom

agent can be built for TORCS with the help of the *simulated car racing* (scr) patch and *Snakeoil* python wrapper. Snakeoil is a python framework to interact with the server that is created from the scr patch. The server will retrieve sensor input from the game and then send what action to perform to TORCS. The sensor input contains information that is very useful for building a self driving agent, see Table 1.

Table 1. *Available sensor inputs.*

Name	Range	Description
angle	$[-\pi, +\pi]$	The angle between the car's main axis and the road's main axis.
track	$[0, 200]$	Vector of range indicators to the road's edge in 19 directions.
trackPos	$[-\infty, +\infty]$	The distance between the car and the center of the road.
speedX	$[-\infty, +\infty]$	The car's speed along it's longitudinal axis.
speedY	$[-\infty, +\infty]$	The car's speed along it's transverse axis.
speedZ	$[-\infty, +\infty]$	The car's speed perpendicular to the road.
wheelSpin	$[0, +\infty]$	Vector of size four representing the wheels' rotation.
rpm	$[0, +\infty]$	The number of rotations per minute of the car's engine.

2.2 Deep deterministic policy gradient

Because TORCS is designed for continuous actions, the state space of possible action combinations is substantial. One way to reduce the state space is to discretize the actions, together with a *deep deterministic policy gradient*, referred to as DDPG. The DDPG is a combination of several other AI techniques such as *deterministic policy gradient* (DPG) and *deep Q-network* (DQN). The hierarchy of the DDPG is an Actor-Critic approach where the Actor network proposes an action and the Critic network predicts if the action is good or not. Two other techniques that are used in DDPG is target networks and experience replay. The target networks are used to add stability to the agent's training by slowly updating the target network and learning from these estimated targets. The target network consists of a copy of both the Actor and Critic networks from where the final target weights are applied. The experience replay is used to learn from samples from all accumulated experiences that are gained during training, compared to only sampling and learning from the most recent experiences Keras [2020].

These methods defines a policy network that the agent will learn how to race from. This policy-based reinforcement learning is based on a parameterized policy, as in Equation (1).

$$\pi_\theta(s,a) = P[a|s,\theta] \tag{1}$$

Here, s and a represent the state and actions, and θ is the policy model parameters. These parameters will be adjusted accordingly using policy objective functions so that the agent primarily avoids getting stuck at a local maxima, but also to avoid straying too far from the race track in order to receive a higher reward. In Q-learning, the action-value function usually works by choosing an action which results in a maximum amount of reward, as in Equation (2) where s_t and a_t are the state and action values for each time step, and R_{t+1} is the reward for the subsequent step.

$$Q(s_t, a_t) = max(R_{t+1}) \tag{2}$$

2.2.1 Actor-Critic network

The Actor-Critic method works by having the Actor network select actions depending on a given policy and the Critic network will then criticize these actions, providing the policy from which the Actor will follow, with a *Temporal-Difference* (TD) error.

The Critic network must always follow whichever policy the Actor network is currently following. This type of method is an extension of reinforcement comparisons used in TD learning. The Critic is a state-value function which evaluates the new state (after an action has been performed) to determine whether the new state leads to better results or not. In practice, this function is calculated using the current state and reward from the TORCS environment, and outputs the TD error to the policy that the Actor network utilizes. The Critic network also uses an Adam optimizer for updating its network weights as a stochastic optimization method.

An overview of the Actor-Critic relation can be seen in Figure 1 where the blue circle represents the value based Critic network and the pink circle represents the policy based Actor network.

An iterative method is used to solve the Q-function using the transition between one state and another. To do this the Actor-Critic network approach is used and the Q-function is replaced with a neural network. The approximation can be seen in Equation (3) where w represent the weight of the network along with the state and action as before.

$$Q^\pi(s,a) \approx Q(s,a,w) \tag{3}$$

A loss function is defined using the mean squared error from which is used to try to minimize the error loss by optimizing the Q-function, see Equation (4). The Q-value determines the value estimation from the current Actor policy and γ is the convergence factor.

$$Loss = [r + \gamma Q(s',a') - Q(s,a)]^2 \tag{4}$$

Since the actions are continuous the gradient of the deterministic policy, $a = \mu(s)$, can be written as in Equation (5).

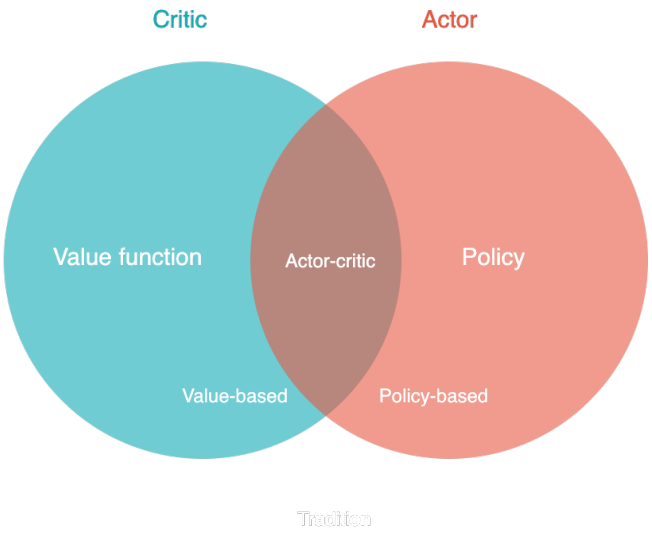

Figure 1. *Venn diagram depicting how the Actor- and Critic networks share input from both the value function and the current policy.*

$$\frac{\partial L(\theta)}{\partial \theta} = \frac{\partial Q(s,a,w)}{\partial a}\frac{\partial a}{\partial \theta} \tag{5}$$

The full Actor-Critic architecture can be seen in Figure 2.

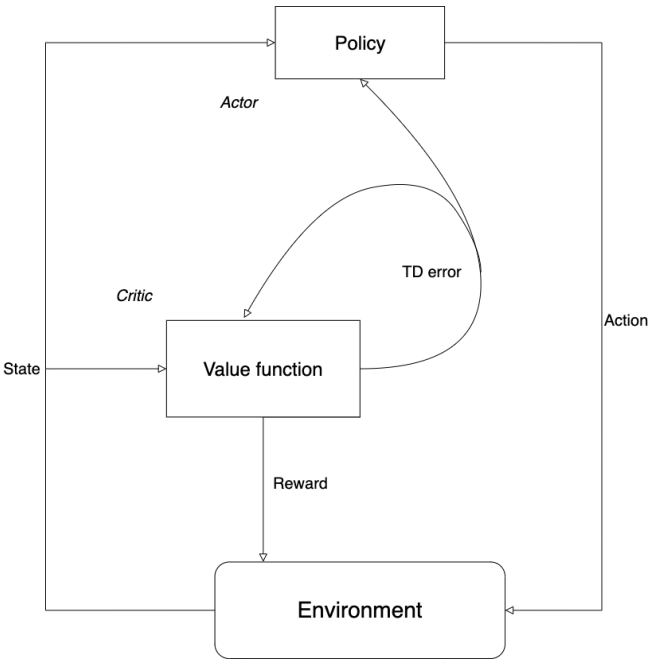

Figure 2. *The Actor-Critic architecture.*

2.2.2 Target network

The target network are time-delayed copies of their original network that slowly track the learned Actor-Critic network. The main reason for using the target value network is that it greatly improves the stability in learning. Values calculated from the Actor-Critic network are used to calculated target values and store them as weights. The weights are progressively updated whilst tracking the learning networks, resulting in a final network which the agent will drive with.

2.2.3 Replay Buffer

As a mean of progression, the agent needs to memorize past experiences to perform better actions for the upcoming episodes. This can be achieved by using a *Replay buffer* which stores all sample experiences gained (state, action, reward etc) for each episode. The replay buffer samples random mini-batches of experience from the buffer when updating the value and policy network.

2.2.4 Ornstein Uhlenbeck process

Adding noise to the actions will prevent the agent from being stuck in local maximas. If the agent randomly chooses which action to perform from a e.g, uniform random distribution, the result would become nonsensical. An example of this would be to suddenly steer 90 degree left whilst giving no acceleration. To counter this, noise is added to the action output following the *Ornstein-Uhlenbeck* process which is a stochastic process commonly used in *DDPG* algorithms. The generated noise acts as an off-policy exploration strategy which is being added to every action to prevent certain actions from cancelling the overall dynamics of the agent. The process is mean-reverting, meaning that over time, it will drift towards its mean value. The Ornstein-Uhlenbeck process defines three parameters Θ, μ and σ for every action in Equation (6).

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t \tag{6}$$

The parameter θ determines how fast the process will revert towards its mean function, μ represents the mean value (or equilibrium) and σ decides the degree of volatility of the process.

Since the *DDPG* method has continuous control it is usually better to have temporally correlated exploration which allows for smoother transitions between actions, rather than using pure Gaussian noise. The process was also used in the original *DDPG* paper Lillicrap et al. [2015].

3. Method

The following chapter will present the project methodology and implementation of the AI-agent and its corresponding policy-network defining what actions to perform and how the reward function was defined. The implementation was made in Python using two main frameworks; TensorFlow and Keras.

3.1 Reward function

The reward function will ultimately define the behaviour of the agent. In this context the agent should stay on track and try to finish the race as fast as possible, to reflect this the reward function should reward a high longitudinal speed and penalize

driving off the road. The used reward function can be seen in Equation (7).

$$R = S_x * (cos(\theta) - |sin(\theta)| - |trackPos|) \quad (7)$$

In Equation (7) the notation for longitudinal speed is S_x, *trackPos* is the distance between the road's center line and the car, and θ is the angle between the car and the direction of the road. This will enforce a behaviour where high speeds along the direction of the road and staying in the middle of the road is rewarded. If the car has driven off the road the episode ends with a negative reward of 500 and the process is restarted.

3.2 Actions

Lau suggests using three different actions for the networks to predict, steering, acceleration, and brake Lau [2016]. To simplify the action dimensions only two actions were used for our agent. The acceleration and brake are both continuous in the domain $[0, +1]$ in Lau's article while this method utilizes only one single action for both acceleration and brake in the domain $[-1, +1]$, meaning -1 corresponds to a full brake and $+1$ to full acceleration.

3.3 Setup for the Actor-Critic networks

The design of the Actor-critic networks can be seen in Figure 3 and 4 below. The Actor network contains of input states and a two layer design with 300 *ReLu* nodes in the first layer followed by 600 in the second. It outputs acceleration and steering. In the critic network, the actions are made visible in the second layer of Sigmoid (Σ) nodes which is connected to a first and third layer of *ReLu* nodes.

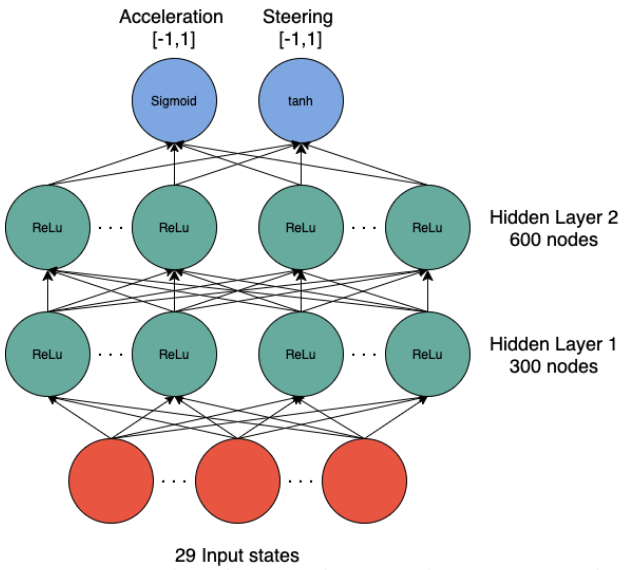

Figure 3. *The schematic design of the actor network.*

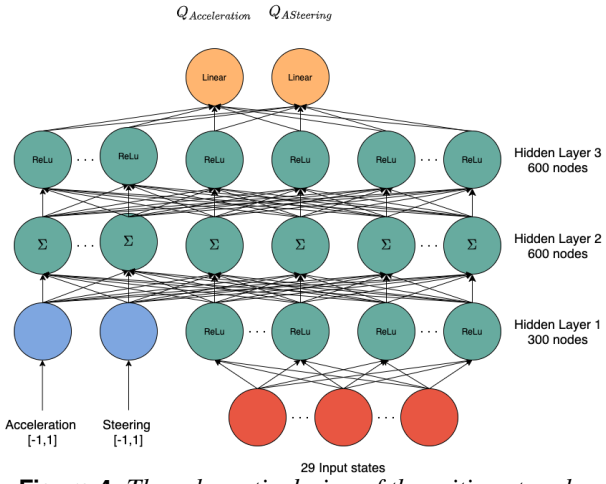

Figure 4. *The schematic design of the critic network.*

4. Result

Using the Actor-Critic network architecture, the agent was trained for a number of episodes, learning through experience replay to finally being able to drive around by itself through the entire track. The graph that shows the rewards that were accumulated after 150 episodes can be seen in Figure 5.

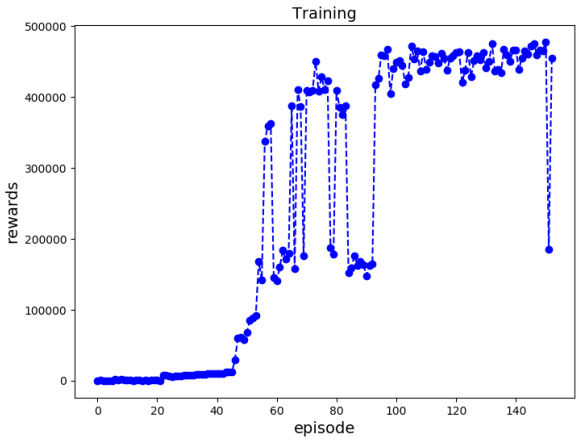

Figure 5. *The rewards accumulated over approximately 150 episodes.*

Adding noise to the action inputs is essential to avoid unwanted behavior resulting in a better off-policy exploration strategy. The Ornstein-Uhlenbeck process was used with the following parameters seen in Table 2.

Table 2. *Chosen parameter values for the Ornstein-Uhlenbeck process.*

Action	θ	μ	σ
Steering	0.6	0.0	0.30
Acceleration	1.0	0.5	0.30

Table 3 displays some of the pre-existing built-in agents

and their lap times in TORCS over two different tracks; CG Speedway 1 and Aalborg, which can be seen in Figure 6, and compare them to our agent by the name DDPG Agent. The time measures are taken from agents using the same car to complete one full lap from the same starting position, without any other agents present on the track.

Table 3. *Comparison between different agents in TORCS.*

Agent	Track	Time (s)
berniw 3	CG Speedway 1	45:20
	Aalborg	1:20:56
inferno 3	CG Speedway 1	44:43
	Aalborg	1:19:72
lliaw 3	CG Speedway 1	44:43
	Aalborg	1:19:72
olethros 3	CG Speedway 1	44:20
	Aalborg	1:26:58
bt 3	CG Speedway 1	46:64
	Aalborg	1:22:28
DDPG Agent	CG Speedway 1	58:19
	Aalborg	DNF

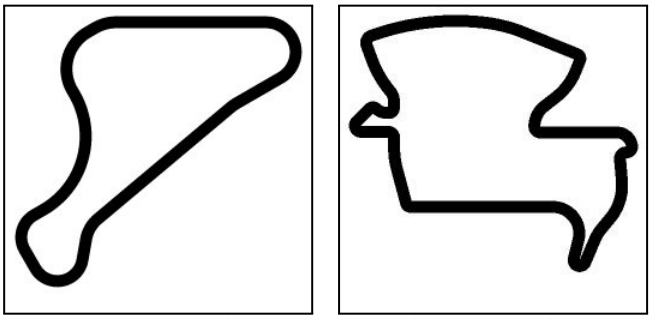

Figure 6. *A map of the two tested tracks, CG Speedway 1 to the left and Aalborg to the right.*

5. Discussion & Conclusion

The following chapter presents a reflection of the results, our conclusions and ideas of how to improve the performance of the agent.

5.1 Training

To thoroughly test our method, more training is required. The agent has trained on the simple course *CG Speedway 1* which can explain why it fails to complete the more difficult track *Aalborg*, see Table 3 and Figure 6. Aalborg consists of several straight segments followed by sharp turns. Since the agent has not yet learned to utilize the brake, these circumstances are particularly hard. Training on a more difficult track could prove useful, as the agent would be exposed to complex track segments such as sharp turns. However, that would require more training episodes and consequently, the stability would decrease.

5.2 Optimizing the reward function

The reward function creates a behaviour where the agent is following the middle of the track in order to maximize its rewards. This discourages following the optimal trajectory around a track and thus reduces performance. The other agents presented in Table 3 follows the optimal trajectory and can thus reduce their lap time drastically. However by rewarding the agent for staying in the middle of the track the stability is improved and with the short amount of time this project is being developed stability is much appreciated.

De Cao et al. suggest a slightly more sophisticated approach that could potentially enable the agent to learn to take the apices of a corner to maintain a higher speed through corners, thus following the optimal trajectory around a track. The proposed reward function will instead encourage staying closer to the edges of the road, but still penalize the agent if it drives off the road. However, De Cao et al. states that this approach is less safe and was therefore discarded for the project.

5.3 Stochastic braking

As of today, braking is not used by our agent. However, it is an interesting aspect of racing, especially when optimizing the agent for sharper turns. In an ideal scenario the agent should drive as fast as possible into a turn (yielding a high reward), brake as late as possible and then accelerate out of the corner. At the same time, the agent will associate braking with a lesser reward and thus not utilize it. One clever way to learn the agent how to use the brake is by applying the brake stochastically Lau [2016].

5.4 Final thoughts

In conclusion, there are many different parameters to keep in mind when optimizing a TORCS agent so that it performs well on the track. TORCS provides a good basis for developing and training agents because of the large amount of sensor inputs. Using DDPG seems to be the one of the best reinforcement methods for this particular problem since it is designed to handle large state continuous spaces, avoiding the curse of dimensionality. Regarding narrow turns the reward function would most likely have to be researched further, incorporating stochastic braking into the action outputs. Another aspect is a better definition of what the test data should consist of, to result in a better overall performance of the agent. A combination between training on both simple and difficult to learn the agent both basic and complex driving behaviour while maintaining stability.

References

Nicola De Cao, Marko Federici, and Luca Simonetto. Ensembling deep deterministic policy gradient trained networks in torcs, 2017. URL *https://github.com/nicola-decao/Torcs-with-DDPG/blob/master/ensembling-deep-deterministic.pdf*.

Keras. Deep deterministic policy gradient (ddpg) example for an inverted pendulum, 2020. URL *https://keras.io/examples/rl/ddpg_pendulum/*.

Ben Lau. Using keras and deep deterministic policy gradient to play torcs, 2016. URL *https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html*.

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. *ICLR 2016; arXiv preprint arXiv:1509.02971*, sep 2015.

Bernhard Wymann. About torcs, 2020. URL *http://torcs.sourceforge.net/index.php?name=Sections&op=viewarticle&artid=1*.