TORCS-report

Bram Kooiman, Dmitrii Krasheninnikov, Akash Raj Komarlu Narendra Gupta

University of Amsterdam

1 Introduction

This report discusses concepts and experiments our team used to implement a driver for The Open Race Car Simulator (TORCS). The driver controlled by a neural network and several heuristics competes in the TORCS races autonomously, and has learned how to drive with external data or previous experience. The report consists of three parts:

- Learning to finish a lap with a single car without crashing;
- Improving performance of a single car (with and without opponents);
- Doing teamwork with two cars (against opponents).

In the first part we discuss subsumption architecture and two simple supervised learning approaches to train a driver: a feed-forward neural network (FNN) and an echo state network (ESN). In the second part we improve upon the basic controller using the Deep Deterministic Policy gradient (DDPG) algorithm. Even though DDPG is typically considered to be a reinforcement learning (RL) algorithm, we suggest that DDPG with parameter noise can be easily interpreted as an evolution strategy. The teamwork part consists of having the car that ends up ahead of the other teammate strive for the finish line (the champion), while the car behind (the bully) is mainly focused on slowing down and damaging opponents that threaten the championing car.

2 A basic controller

2.1 Subsumption Architecture

The racer is built using several competence levels. The low competence levels are rule-based and implement simple but hard to learn behaviors such as backing up when facing a wall and getting back to the track if we have gone off-track. The highest competence level uses a neural network (either a FNN or ESN trained in a supervised manner, or the FNN trained with DDPG) to control steering and acceleration. By default, the highest competence level is active. However, if the condition of a low competence level is met, such as the car going off track, the lower competence level takes priority over the higher competence level.

2.2 Supervised learning with a FNN and an ESN

In the first part of the assignment, the driver is trained to stay on track using a FNN and an ESN. The driver is trained to predict the control commands, a tuple of *(steering, acceleration, brake)*, from its current sensor data. For training the networks our team used the two small datasets provided by the course TAs.

In the recent years FNNs have proven to be very powerful in a variety of applications, including autonomous driving [1]. ESN is a NN architecture aimed at working with sequential data; it speaks to the intuition that remembering past events is important when driving a car. We test the validity of this intuition by comparing ESN to FNN.

Unfortunately, both methods turn out to not perform particularly well. ESN drives off-track very quickly and cannot complete any of the tracks; FNN performs generally fine, but will sometimes stop in the middle of the road due to an erroneous brake prediction and would often crash due to a failure to brake at a turn. On simple tracks the FNN is able to complete a lap, but the laptime is not impressive.

In order to fix the driver suffering from erroneous brake predictions and underpowered acceleration we decided to implement a thresholding function for braking and acceleration. Above some predicted value braking is set to 1, and is set to 0 otherwise. In case both values are 1, acceleration is given priority. This strategy results in a very reckless driver that nevertheless is able to finish most of the simple tracks. This experience helps us realize that crudely discretizing the provided continuous action space leads to poor results, and we decide to remove the thresholding function for acceleration. Knowing this, in the second part of the assignment we choose an algorithm designed for continuous action spaces – Deep Deterministic Policy Gradient (DDPG).

3 Learning to race with DDPG

In the second part of the assignment the goal to use an evolutionary algorithm (EA) to improve the driver's performance and learn how to navigate a race with opponents. Our team has decided to use a RL algorithm Deep Deterministic Policy Gradients [5]. This section describes the experimental setup, the DDPG algorithm, and interprets DDPG with parameter noise as an EA.

3.1 The TORCS environment as a Markov Decision Process

Markov decision processes (MDPs) are the most common way to represent the environment for RL algorithms. The TORCS environment is formalized as a MDP $< S, A, P, R >$, where:

- S is a set of states s; each s is a vector of sensor data received from TORCS.

– A is a set of actions:

$$A = \{a_1 = steering,\ a_2 = acceleration,\ a_3 = brake\};$$
$$a_1, a_2, a_3 \in \mathbb{R};\quad a_1 \in [-1, 1];\quad a_2, a_3 \in [0, 1].$$

– $P(s_{t+1}|s_t, a_t)$ is the transition probability between the states.
– $R(s)$ is the reward function. We define the R(s) as follows:

$$R(s) = s.vX \cos(s.angle) - |s.vX \sin(s.angle)| - s.vX|s.trackPos|,$$

where $s.vX$ corresponds to $s.speedX$ in the state vector. Additionally, the agent receives $r = -100$ during each timestep it spends off-track. Thus $R(s)$ incentivises the agent to not die, drive fast and stay on track.

To train the agent we run the TORCS race multiple times using randomly selected tracks, and refer to each run an *episode*. To control the interface between the agent and TORCS we use a version of the gym-TORCS client [9] modified to work for multiple tracks with opponents. The agent receives the rewards according to the $R(s)$, and tries to maximize the discounted sum of rewards per episode, also referred to as *return*: $J = \mathbb{E}_{s' \sim p(s'|s,a),\ a} \sum_{t=0}^{T} \gamma^t r_t$, where r_t is the reward received at step t of the episode, and γ is the discount factor $0 \leq \gamma \leq 1$.

3.2 Deep Deterministic Policy Gradient

We use DDPG with experience replay and parameter noise to learn a policy $\mu(s)$ that maps state vectors to the corresponding action vectors. The policy function is represented with an *actor* network, small FNN with two hidden layers of 32 and 16 neurons, and the RELU activation function. DDPG is an *actor-critic* algorithm, so in order to learn the parameters of the actor network, DDPG uses a *critic* network $Q(s, a)$ to estimate the return J given the current state and action. The architecture of the critic network is the same as that of the actor network, apart from the different output shape ($\mathbb{R}^3$ for $\mu(s)$ and $\mathbb{R}^1$ for $Q(s, a)$). DDPG is a *model-free* algorithm, which means that the reward function $R(s)$ and the transition probability $P(s'|s, a)$ are modeled by the critic network implicitly.

It is important to note that unlike in most RL algorithms, the policy used by DDPG is deterministic, that is, the actor network $\mu(\cdot)$ outputs only one action as opposed to a probability distribution over actions. The DDPG algorithm takes advantage of the Bellman optimality equation for a deterministic policy:

$$Q(s, a) = R(s) + \gamma \mathbb{E}_{r, s' \sim p(s'|s,a)} Q(s', \mu(s'))$$

To obtain the parameters θ^Q of the critic network $Q(s, a|\theta^Q)$ satisfying the Bellman optimality equation, the following loss is minimized with gradient descent:

$$L(\theta^Q) = \mathbb{E}_{s,a,r}\Big[\big(Q(s, a|\theta^Q) - y\big)^2\Big],$$
$$\text{where } y = R(s) + \gamma\big(Q(s', \mu(s'))\big)$$

The actor is trained with GD using the gradient of the return J w.r.t parameters of the actor θ^μ (recall that the critic $Q(s, a)$ approximates the return after taking action a in state s):

$$\nabla_{\theta^\mu} J \approx \mathbb{E}_{s\sim p(s'|s,a)}\big[\nabla_a Q(s, a|\theta^Q)|_{s,a=\mu(s)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_s\big]$$

The deterministic policy gradient theorem proved by [7] states that this is the gradient of the policy's performance.

The agent attempts to simultaneously maximize the return based on the existing knowledge and acquire new knowledge; these two processes are referred to as exploitation and exploration. To aid exploration, we implement a modification of DDPG that uses *parameter noise* $\theta^\mu \leftarrow \theta^\mu + \mathcal{N}(0, \sigma^2)$ [6], where σ^2 is adapted over time and by controlling variance in the action space induced by the noise.

In the *experience replay* setting the agent records a large number of tuples (s, a, r, s') it encounters, and uses these to update θ and θ^μ using minibatch gradient descent. This significantly improves data-efficiency of the training process compared to the online learning setting. We use an experience replay buffer of length 10^7 to store the (s, a, r, s') tuples, and overwrite the oldest entries in the buffer with new tuples when the buffer gets filled. We use the Tensorflow implementation of DDPG by OpenAI [2], adapt it to work with gym-TORCS, set up the parameter noise and train the agent with a NVIDIA GTX 1050 GPU.

3.3 DDPG with parameter noise as an evolutionary strategy

The motivation for using DDPG is that we expected it to perform better and be more data-efficient with the training than EAs such as NEAT and CMA-ES. In particular, both EA and RL are good at optimizing complex non-differentiable functions. In EA terms, algorithms such as NEAT [8] perform well largely due to the competition dynamic among the children and ease of parallelization; RL algorithms on the other hand excel largely by having few (or one) smart children that take advantage of an additive $R(s)$ and understand which particular actions lead to which particular rewards. We chose DDPG since we believed we can design a reward function that would give the agent rich feedback on which actions are good and which actions are bad (as opposed to feedback about the overall goodness of the policies on which EAs rely).

DDPG with parameter noise can be interpreted as (1+1)-ES: one parent θ_i^μ, the actor at epoch i; one child $\theta_{i+1}^\mu = \theta_i^\mu - \alpha\nabla_{\theta^\mu} J + \mathcal{N}(0, \sigma^2)$. DDPG here is very similar to a one child version of CMA-ES: both DDPG and CMA-ES adapt the noise level ("mutation strength") in the process of exploration; DDPG relies on having one smart child by using the $\nabla_{\theta^\mu} J$ as part of the mutation, and CMA-ES relies on having many children and selecting the best ones [3]. One interesting avenue for future work is adding a competition dynamic to DDPG – having multiple children using different noise samples, and only picking the best ones at the selection stage. This would result in a hybrid approach that uses the strengths of both EAs and RL, and is likely better at exploration.

4 Teamwork and Swarm Intelligence

We let two cars race together as a team with a champion/bully approach. The champion strives to be the first finisher and get the team a good score. The bully has an aggressive behavior: it tries to crash into the opponents to slow them down. The champion is the DDPG actor discussed in the previous section. The bully is trained in a similar manner with a reward function that includes speed and proximity to the opponents. If the distance between the two cars is greater than a threshold value, the bully becomes a champion and tries to decrease this distance (attraction rule). The two cars communicate by sharing their race position, their role and the distance from the start via a file.

The Swarm Intelligence paradigm might have fulfilled its potential better had there been more than two individuals. The Boids algorithm would have been more in place if finishing the race was a team effort with e. g.the score determined by the last finisher. PSO is a useful algorithm for a training particular neural net, but it does not teach the drivers to do teamwork.

5 Experimental results and conclusion

Results for FNN and DDPG, approaches that did end up working, are presented in Table 1 along with the results for default TORCS drivers *berniw* and *lliaw*. We can see that neither FNN nor DDPG are able to beat the rule-based bots. The main limitation of FNN seems to be the small amount of training data. If we were to pursue FNN as the main algorithm in this work, a trivial solution to that would have been collecting more data from e.g. *lliaw* and training the FNN on that. Determining the main limitation of DDPG is harder, but we think that more effort should be put into designing a well-suited reward function. For example, the DDPG driver ended up being very wobbly, and a small penalty for steering might have helped. Additionally, we did not use the subsumption architecture when training DDPG, so it may be the case that the agent performs worse when we do use it since the agent has no experience relying on the deterministic rules. Finally, the DDPG driver did not learn how to brake, likely since the reward diminishes immediately as the agent brakes (as reward is proportional to speed). We tried using the *stochastic brake* [4] technique to help DDPG explore braking, but were not successful.

Track	**FNN**	**DDPG**	**berniw**	**lliaw**
Forza	1:26	4:21	1:43	**1:21**
Michigan	2:45	1:46	0:47	**0:33**
A-speedway	1:30	1:28	0:39	**0:26**
Dirt-1	0:50	2:28	0:40	**0:34**

Table 1. Laptimes for different tracks and drivers

In this work, we built a controller that autonomously controls a driver in TORCS. We used supervised learning methods FNN and ESN, as well as a reinforcement learning algorithm DDPG. We learned a lot about all of these, and understood the connection between reinforcement learning and evolutionary algorithms.

Bibliography

[1] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[2] Prafulla Dhariwal, Christopher Hesse, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. https://github.com/openai/baselines, 2017.

[3] David Ha. A Visual Guide to Evolution Strategies. Web page, 2017.

[4] Ben Lau. Using Keras and Deep Deterministic Policy Gradient to play TORCS. Web page, 2016.

[5] 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. *arXiv preprint arXiv:1509.02971*, 2015.

[6] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

[7] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014.

[8] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[9] Naoto Yoshida. Gym-TORCS. GitHub Repository, 2016.