

Ensembling Deep Deterministic Policy Gradients trained networks in Torcs

Nicola De Cao - 11286571 – Marco Federici - 11413042
Luca Simonetto - 11413522

University of Amsterdam

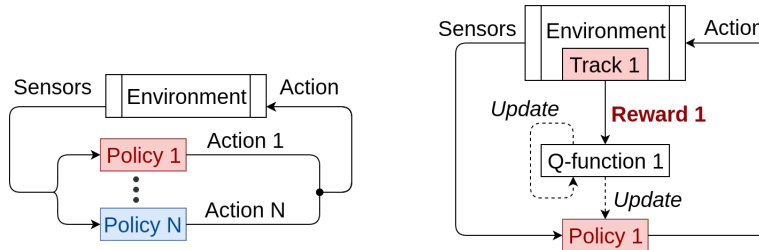
1 The controller

The principal component of our car controller consists in a feed forward neural network which uses 29 inputs (track angle, track position, speeds along 3 axis, RPM, 4 wheel spin velocities and 19 proximity sensors) to predict 2 values from -1 to 1 that represent the steering and the change in acceleration (-1=full brake and 1=full accelerate) respectively. We choose to merge both acceleration and brake into a single output in order to reduce the number of tunable parameters along with the reduction of the action space size. Making acceleration and brake mutually exclusive experimentally led to comparable results.

1.1 The training algorithm

In order to train our model we decided to use a state of the art reinforcement learning technique rather than supervised ones. The DDPG (Deep Deterministic Policy Gradients) algorithm [3] resulted to be the best candidate for our problem since it grants the advantages of Deep Q-Learning techniques in a continuous action domain [4] (a brief explanation in appendix A).

Furthermore, the necessity of flexibility resulted in the creation of multiple smaller models, each one trained on a different kind of track with a specific reward function, rather than a single one. These trained model were merged into a better performing controller by combining the outputs during the racing phase. Some genetic-tuned rule-based heuristics increase the robustness of our driver.



(a) Diagram of racing setup.

(b) Diagram of training setup on track 1 with the reward function 1 for the policy 1.

1.2 The implementation

Our implementation and adaptation of the DDPG algorithm relies on python Keras-rl module¹ which is built on top of the Keras framework² for Tensorflow³. After the implementation, we focused on finding the best reward function and hyper-parameters for the algorithm following basic guidelines from similar works⁴. Hardware, software and time limitations led to simplifications of the models and radical changes in the training phase. The introduction of *curriculum learning* [1] is just one of the strategies described in this section which we applied to decrease the training time.

The reward functions We used two different reward functions to train our neural networks (three out of five are trained with r_1 , the other two with r_2). r_1 leads to a stable behavior by rewarding the speed of the car, its position and alignment with the track axis.

$$r_1 = \begin{cases} -500 & (if\ |p| > 0.99) \\ s \cdot (\cos(\theta) - |\sin(\theta)| - |p|) & (otherwise) \end{cases}$$

where θ is the angle between the car direction and the direction of the track axis, s is the current speed of the car in km/h and p is the distance between the car and the track axis normalized from -1 to 1.

Then r_2 allows the car to be in the edges of the track (up to -0.85 and 0.85) without being punished for that. This is less safe but allow the car to learn to steer sharply and cut simple turns.

$$r_2 = \begin{cases} -5 & (if\ |p| > 0.99) \\ (d_t - d_{t-1}) \cdot (\cos(\theta) - |\sin(\theta)|) & (if\ |p| > 0.85) \\ (d_t - d_{t-1}) \cdot (\cos(\theta) - |\sin(\theta)| - (|p| - 0.85)^2 / 0.0225) & (otherwise) \end{cases}$$

where the factor 0.0225 is just $1/(1 - 0.85)^2$ which is needed to normalize the off-track-center factor from -1 to 1 and d_t is the distance from the start in time instant t in meters.

The controller were punished with a negative value, by both of the rewards, if the car goes off the track. The same negative reward is given if the car bumps into guardrails in order to remove the wrong behavior of using them to finish the track.

DDPG Hyper-parameters In order to tune the convergence factor τ and both the learning rates for the ADAM optimizer [2] we followed the original values described in the cited DDPG paper. In our specific case we used Ornstein-Uhlenbeck process to generate temporally correlated noise (using Brownian motion). The noise was centered in 0 for the steering and 0.2 for the acceleration/brake output to explore increases in speed values.

¹ <https://github.com/matthiasplappert/keras-rl>

² <https://keras.io/>

³ <https://www.tensorflow.org/>

⁴ <https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html>

Curriculum Learning A limitation of our work is the neural network size and training time. Since for this project we trained under strict time constraints using simple laptops, we experimentally verified that the training time could be reduced by limiting the maximum allowed speed. Starting from 80 km/h we gradually increased the upper limit by 10 km/h for every completed lap. Following this procedure, the networks quickly learned how to brake and turn properly. Picture 1.2 shows the effect of our training methodology on the average speed of the car in different episodes during the training phase. The x axis represents the iteration number while y values show the average speed of the car for that episode. The collected data underlines the effectiveness of the curriculum approach while showing the 2 different phases in the process (blue line in Picture 1.2). The average speed gradually increases while the controller is learning with a fixed upper speed limit, then it drops instantaneously when the speed limit is incremented to stabilize again after a few episodes.

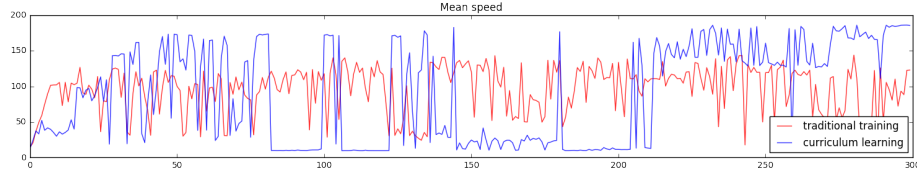


Fig. 2: Effect of curriculum learning on average speed in the first 300 iterations.

Ensemble methods Instead of training a single network on different tracks, we chose to train multiple models on a representative sample of them using 2 different reward functions. This option suited best our needs since it allowed us to train multiple models at the same time (each of them can be trained independently on different machines) while reducing the variance of the resulting model and making it less prone to over-fitting.

Every predicted action is the result of a weighted average of different trained networks. The steering is calculated by taking the average of the steering values, however, the acceleration/brake action is driven towards a more safe behavior. If at least one of the model predicts to brake, its weight is increased, along with every other agreeing model. If no models predict a brake action, we take a simple average of the suggested accelerations. Each weight has been tuned in order to find a balance between safer behaviors (r_1) and dangerous but better performing ones (r_2).

Rule based heuristics In order to to increase the stability of the controller and to make it perform better, we added some basic rules:

1. if the car is too slow or not moving at all, then the acceleration is set to the max value and the brake is regulated to the minimum (the threshold for triggering this behavior were properly tuned). This rule has been added to force a movement when the car is blocked.

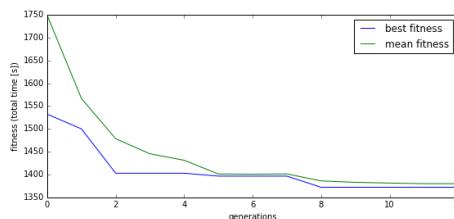
2. if the distance between the car and the border is less than the braking distance, we do not allow the car to accelerate anymore and we increase the braking force ($brake_{Force}$) by a fixed factor. In order to do so, we empirically computed the value of the minimum distance required to completely stop when driving at the current speed ⁵) and we added a tunable offset o to regulate the braking distance.
3. if the speed exceeds a fixed threshold the car steering is elevated to the power of 2, in order to help to increase the stability when the car is racing in a speedway).

2 Parameters tuning with genetic algorithm

Since the rules created to regulate the acceleration/brake actions resulted to have a major impact on the performance of the controller (allowing it to cut corners cleanly or avoiding going out of track when performing sharp turns), we decided to tune only the two parameters that have the highest influence.

The two most influential parameters were experimentally seen to be the offset o , and the $brake_{Force}$ from heuristics rule 2 (see section 1.2). This optimization problem has been approached by implementing a single objective genetic algorithm using The Watchmaker Framework⁶. The fitness function is the sum of one lap times for a subset of 6 tracks in which the car is tested, namely alpine-1, alpine-2, b-speedway, corkscrew, aalborg and e-track-4. As the time needed to calculate each individuals fitness is moderately large, a compromise has been made between evaluation time and population size: each generation is composed by 15 individuals, that produce off-springs via a 1 point crossover and random mutation (2% probability). In order to get a reasonable convergence time we used elitism (of 1) and to leave enough time to search for solutions we used a stagnation termination of 10 generations (if no improvements have been made in the last 10 generations the algorithm stops). Results of this operation can be seen in figure ?? and table ??.

Fig. 3: Results of the genetic algorithm run.



3 Racing against opponents

Our controller resulted to be stable enough to compete with other cars without particular changes. Since opponent sensors were ignored during the training

⁵ the value has been calculated experimentally with data from Torcs, and resulted to be $0.851898 \cdot 10^{-3} \cdot s^2 + 0.104532 \cdot s - 2.03841$

⁶ <http://watchmaker.uncommons.org>

phase our controller ignored other cars which resulted in unwanted outcomes. Because of that, we added other rules to the controller in order to avoid hitting opponents. Furthermore we implemented a simple overtake procedure and a behavior that aims to avoid being overtaken by the opponents. This seems not a good option since we override the behavior of the controller but this choice was motivated by the fact our driver is not able to do this kind of action by itself.

A possible solution we did not explore but would be great for future work is to take into consideration opponents during the training phase. This approach would lead to a slower training but would allow the controller to learn how to behave among other cars by itself.

4 Performance evaluation

We evaluated our controller performing both alone but also with opponents in all the 38 tracks available on the Torcs package. We measured whether the controller was able to perform at least one lap (with and without touching the border) and the sum of lap times all over the tracks. As we see in table ?? with a basic controller trained on only one track (alpine-1) the car was able to finish almost all the tracks but it hit the border in most of them. With the ensemble controller, over 5 networks, the controller performed worse considering the lap times. That is because the car tend to accelerate more. However, differently from the basic controller, the car was able to do not hit the border in more than half of the tracks. Adding heuristics the total lap time all over the tracks dropped of one third and the controller was able to do all the tracks except one. Tuning the hyper-parameters with a genetic algorithm allowed the controller to finish all the tracks and to do not hit the border in more than 80% of them. Moreover the controller gained more than 5 minutes in the sum of the track lap time.

| Method | Tracks | All tracks | alpine-2 | spring | dirt-5 | forza |
|----------------|-----------------|--------------|---------------|---------------|--------------|---------------|
| Basic | 6-35/38 | 5339s | 125.1s | (H) 619.8s | (H) 73.8 s | (H) 138.6s |
| Ensemble | 25-33/38 | 5735s | 118.9s | 554.6s | (H) 43.0s | 125.9s |
| Heuristics | 22-37/38 | 4083s | 116.3s | 554.0s | (H) 47.8s | 124.5s |
| Genetic | 31-38/38 | 3758s | 111.9s | 510.2s | 53.6s | 108.6s |

Table 1: Controller evaluation, 4 tracks and overall: (H) means whether the car hit the border of the track. The first number in tracks means how many tracks the car finished without touching the border, the other the total.

5 Conclusions

Our work shows how using reinforcement learning on continuous domains can lead to very good results even with a relatively small feed-forward network trained with limited computational resources. We show how curriculum learning technique boosted the convergence of the training process. Furthermore we highlight how averaging multiple networks both reduces the variance, increases precision and allows parallel training which drastically lowers the training time.

A DDPG

The main idea behind the Deep Deterministic Policy Gradient is to use two different models for the reinforcement training phase.

The first model "*Actor*" represents a deterministic policy function $\mu : s \rightarrow a$ from the state s (values of all the sensors) to an action a (combination of values for steering and acceleration/brake), while the second one "*Critic*" takes as input both the predicted action and the state to compute the maximum discounted future reward R following the given deterministic policy $Q^\mu : s, a \rightarrow R$.

The total discount for future reward R is defined as:

$$R = r_1 + \gamma r_2 + \gamma^2 r_3 + \dots + \gamma^n r_n$$

. Where $r_1 \dots r_n$ represent the reward in states $s_1 \dots s_n$ respectively. Since we want our policy μ to maximize the Q-function, we can compute the gradients with respect to the parameters θ^μ of the actor using the chain rule:

$$\Delta_{\theta^\mu} = \frac{\partial Q^\mu(s, a)}{\partial \theta^\mu} = \frac{\partial Q^\mu(s, a)}{\partial a} \frac{\partial a}{\partial \theta^\mu} = \frac{\partial Q^\mu(s, a)}{\partial a} \frac{\partial \mu(s)}{\partial \theta^\mu}$$

In order to update the critic model, we can use the SARSA (State-Action-Reward-State-Action) rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, \mu(s_{t+1})) - Q(s_t, a_t)]$$

Since reinforcement learning algorithms make the assumption that samples are independently and equally distributed, instead of training directly on the sequence of states, the algorithm randomly samples from a replay buffer. Each entry of the buffer is a tuple (s_t, a_t, r_t, s_{t+1}) representing the current state s_t , the corresponding action a_t , the current reward r_t and the next observed state s_{t+1} . These values are first used to update the critic network and, then, the target one.

Since both networks are prone to divergence, instead of directly updating the weights during the back-propagation step, DDPG uses 2 copies (*targets*) of each network and updates the weights θ with a soft update equation:

$$\theta' \leftarrow \tau \theta + (1 - \tau) \theta'$$

The exploration of the neighborhood of the selected action policy is a fundamental part of the whole process, in fact, random noise \mathcal{N} is added to the action of the *Actor* at every step during the training phase.

Bibliography

- [1] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.
- [2] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.
- [4] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. 2014.