Deep Reinforcement Learning for Autonomous Racing Track Agent (Project Cars 2™)

Michael Hatchi
mhatchi@vrona.io

Human + Machine
#UnitedByRacingSkills

initial release: 01/08/2020 - update: 02/01/2020

Abstract

Contextually, autonomous vehicle is a hot topic for years. The *future of racing* is a related topic on which mainly car manufacturers are trying to anticipate guided by software, energetic and safety developments.
Even though full autonomous technology is not yet widely accepted by day to day drivers around the world, motor sports field acts as a full-scale lab and makes the advent of self-car driving even closer.
Indeed, the launch in 2019 of *Roborace Season Alpha* sets autonomous racing standard and is a remarkable achievement.

Technically, reinforcement learning enables engineers to develop machines that could learn on their own within a relatively simple deterministic space. Thus, for more complex state and action spaces the combination of deep learning with reinforcement learning is required as the nature of spaces changes to continuous.
On 2015/2016, a team from Google has published the Deep deterministic policy gradient (DDPG) [1] algorithm. Starting from it, DDPG algorithm and some of its variants, has helped to solve highly complex problems in continuous spaces.

VRONA® BOT88 is a self-driving car project that leans toward building an autonomous racing driver model based on plain input vision (no lane and surface detectors preprocessing). It learns to control a Porsche 911 GT3 R on (Barcelona) Catalunya GP racetrack in the notorious high quality racing simulation (sim-racing) title, *Project Cars 2*™.
This project aims to answer the following question: how to build a simulated racing track driver that would learn to drive based on vision with the intuition of a racing line?
The underlying idea here, is that like racing driver that uses his/her senses (then skills) to perform, the model should learn in the same context as a driver.

DDPG algorithm has been, then, used with an asymmetric actor-critic.

This publishing is focused on BOT88's training phase.

Special thanks to the Quarante family, Khelil, Damien and my family for their unfailing support.

I. Introduction

Motor sports, and especially racing car, are in permanent evolution. In less than 5 years, a tipping point has come.
Indeed, each area of the sport has pushed its own boundaries due to technologies advances and progressive change in mentalities:

- Powertrain: hybrid, full electric,
- Audience/Broadcast: Multi-screens, 5G, …
- Safety: real-time driver's health data,
- Environment: real, virtual (Esport),
- Driver: human, machine (autonomous).

Development of real-world autonomous vehicle required an important structure and infrastructure.
As training an agent is based on try and errors scheme. Up to a certain level of generalization, real-world training cannot be considered because of physical damage and collateral costs.

One solution is to develop autonomous racing car 'agent' within a simulated racing environment.
This solution would nurture purposes like (non-exhaustive list):

- Additional support for autonomous racing track championship which, as a reminder, represents a deep interest for 'mobility' industry (automotive, aerospace, defense, rescue).

- Simulation of race accident which helps to take into consideration critical points upstream real driver races in deterministic scenario.

Nowadays, very accurate replica of racing environments like[1]: *rFactor2* (developed by Image Space Incorporated and Studio 397) or *Project Cars 2™ (*developed by Slightly Mad Studios) provides real racetrack data, car states and motions data.

Technically, deep learning, reinforcement learning and the fusion of these two have proven spectacular successes of learning and solving complicated tasks. Google Deepmind team with *AlphaGO* [3] on Go game and with another model on *Atari 2600 games* [4] have shown the way.

By using Deep Reinforcement Learning, in 2018, a team of researchers from *INRIA*, has made an impressive research - 'end to end *race driving with deep reinforcement learning'* [2] - and demonstrated the validity of using a high-quality video game at training stage before using the agent control model into real self-driving car experiments which produced great results.

Their agent had been trained on WRC6™ (World Rally Championship - FIA) video game AAA title as this would provide replicas of real-world roads (see *about WRC*[2]) with different surfaces and weather conditions.
They employed an Asynchronous Advantage Actor Critic (A3C) algorithm to learn lateral and longitudinal control from a virtual front camera. And trained it on multiple game instances which run on different machines.

Another team of researcher from Carnegie Mellon University has published - *Deep Reinforcement Learning for Autonomous Driving* [5] - in 2018, as well. They describe how their autonomous car agent solved the problem of control within another simulated environment: TORCS (The Open Racing Car Simulator) with full states inputs (localization and motion data) and Deep Deterministic Policy Gradient (DDPG) algorithm.
And finally, a third team released in a former robotics study - *Asymmetric Actor Critic for Image-Based Robot Learning* [7] – an interesting deep reinforcement learning model where image and full states are both used to solve robotic tasks.

Fig. 1: BOT88 drives a Porsche 911 GT3 R from Project Cars 2™.

Thus, in the same vein of these three research works, VRONA® BOT88 solves a reinforcement learning problem where it interacts with a racetrack environment (state S, in the form of screen inputs), due to actions (actions a among: accelerating, turning left or right and braking) and receives a reward (r) from reward function made of motion data.
The first two researches are using the middle of the track as a guideline. The latter is using a goal localization where the robotic arm should be set to succeed its task.
In our case, the racing line[3] has been given to the agent as a goal localization.

Because of the high-dimension environment and the action continuous space at each state, BOT88 uses deep neural network (deep learning) to map directly raw images input to control outputs. The agent shown interesting progress with Deep Deterministic Policy Gradient with asymmetric actor-critic at this stage (sec. Method and [7]).

II. Background

A. Reinforcement Learning

As said previously, reinforcement learning (RL) is an agent which needs to interact with its environment to learn and solves the problem is dealing with.
In main features, RL is based on Markov Decision Process (MDP), where the future is independent of the past given present.

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, S_2, S_3, \dots, S_t)$$

MDP is a tuple of five components (S, A, P, R, γ) where:

$S \rightarrow$ a set of states in the environment,
$A \rightarrow$ a finite set of actions (deterministic or in our case, continuous) an agent can do in the environment,
$P \rightarrow$ the transition probability matrix (maps S to A),
$R \rightarrow$ reward component from reward function for given (S, A),
$\gamma \rightarrow$ discount factor where $\gamma \in [0,1]$ which weights distant future R to immediate future R.
By using MDP, the goal is to obtain the maximum expected cumulative reward (G_t) at timestep t for a given state until the ultimate state.

$$G_t = \sum_{m=0}^{\infty} R_{t+m+1}$$

Thus, in a stochastic context, during every episode an agent starts at initial state S_0 from S, makes an action a_0 from A due to an initial transition probability $\pi(a|S) = P(a_t = a, S_t = S)$, observes the new state S_1 and get in return a reward $r_0 = r(S_0, a_0)$ from R.

1 These are consumer racing simulation. Gran Turismo, Assetto Corsa are other titles.

2 https://www.wrc.com/en/wrc/about-wrc/what-is-wrc/page/673---672-.html

3 The ideal trajectory that deal with maintaining highest speed as late as possible before touch the apex of a curve and retrieve high speed as soon as possible when exiting this curve.

It pursues this mechanism at each timestep until the terminal state (which ended an episode), which enable to obtain:

- the State-Value $V_{(s)}$, computed due to value function following policy π:

$$V_{\pi(S)} = E\,[G_t|S_t = S]$$

- the Action-Value $Q_{\pi(S,a)}$ computed due to action-value function following policy π:

$$Q_{\pi(S,a)} = E_\pi\,[G_t|S_t = S, a_t = a]$$

The main goal is to find the optimal Policy π^* when the policy maximizes the expected reward for all state in S.

$$\pi^*(a|S) = argmax\ Q_{\pi(S,a)}$$

B. Deep Learning (DL)

DL acts like features extractor from high-dimensional data where the input data are vectorized and nurtures non-linear activation functions which output next inputs of layer of activation function and so forth.
The multi-layer tends to minimize a given loss function on a training set (or a designed reference marker) using backpropagation which then, updates the weights of the network.
In BOT88 case, DL is used to extracted features from image inputs (straight or curved lines, texture, …) due to CNN structure (see. below) and to approximate policy function (see. C. DDPG) and action-value function from low dimension inputs (car motion data and network weights).

- Convolution neural network (CNN).

RGB image inputs (like here from sim-racing environment) have 3 dimensions: width, height and depth which are high-dimension data.
CNN is a type of neural network intrinsically made upon the said 3D shape data.
The asset of such structure is to extract sum of convolution from precise region from the sliding of a filter window over the input data. These are then stored in a feature map.
CNN uses pooling layer, as well, right after convolution layer in order to reduce the dimensionality as convolution layer tend to enlarge it (because of filters). Max pooling layer enables to reduce the size but helps to keep the relevant data.

CNN, obviously, allows to work on large dimension inputs with less training time compared to a multi-layer perceptron (MLP). MLP would request much more nodes and layers to achieve the same task.

- Deep Reinforcement Learning

Deep Q network (DQN) is a well-known deep reinforcement learning algorithm that performs in discrete action space. It performs then in few actions' possibilities context as its deep network approximates the action-value function ($Q_{\pi(S,a)}$) at each state based on the action that has the maximum value. In other words, it combines the mechanism of RL to find the optimal policy π* (due to optimal Q^*) with DL capacities to deal with non-linear environments. Deep RL bridges the gap between machine and human capacities.
DQN follows two methods:

- model-free: transition probability S to S' is initially unknown,
- off-policy: one policy is evaluated 'Target Policy', another 'Behavioral Policy' is followed for exploration.

C. Deep Deterministic Policy Gradient (DDPG) algorithm

Autonomous racing car, and then obviously day to day car, evolves in continuous spaces.
Indeed, starting from 0 velocity accompanied with extreme acceleration up to high velocity with immediate steering reaction (with continuous angle possibilities) until a strong brake[4] before a curve represent to deal with continuous action spaces while piloting precisely.

[4] in order to transfer the mass of the racing car onto its front wheels to maximize grip and to reach the highest possible speed.

Thus, to learn and behave in a racetrack environment, the racing car agent needs a RL algorithm with characteristics that can handle the mandatory continuous action spaces.

DDPG and DQN have in common to look for $Q^*(s,a)$ which helps to find optimal action given state $a^*(s)$. But in a continuous action space which is wide, it would be untenable. Thus, a gradient-based learning rule for a policy $a = \mu(s)$ is key as instead of $max_a Q(s,a)$ (see. DQN) in DDPG we approximate it: $max_a Q(s,a) \approx Q(s,\mu(s))$.

DDPG integrates an Actor-Critic architecture.
The Actor tends to find the optimal parameters for a policy function approximator which maps $S\ to\ A$. And Critic evaluates the function policy by approximating action-value function like DQN does.
Actor learns a deterministic policy $\mu_\theta(S)$ and needs a gradient ascent from Critic (with policy parameters only) which solves:

$$\max_\theta E_{s\sim D}\,[Q_\theta(s,\mu_\theta(s))]$$

D: set of transition (S,a,r,S_{t+1},d) where d indicates whether S_{t+1} is terminal.
Both, DQN and DDPG, use experience replay buffer fulfill of D (as a cache of finite size to avoid dependence between exploration) from which they learn.

DDPG algorithm involves four neural networks:

- Actor network: Deterministic policy function with weights θ^μ, maps the states to actions (acceleration, left, right, brake) instead of outputting the probability distribution across a discrete action space.
  It provides the agent's current policy $\mu(s|\theta^\mu)$ for the Q function (aka action-value function).

- Critic network: Q network with weights θ^Q, approximates the action-value function $Q(s,a|\theta^Q)$.

- target policy network $\theta^{\mu'}$and target Q network $\theta^{Q'}$are respectively copies of Actor and Critic networks. But have different network weights θ' and are time delayed in order to slowly track the learned networks $\theta' = \tau\theta + (1-\tau)\theta'$.

τ, is an hyperparameter that decays the learning rate linearly and which constrains the target values with a slow updating speed. This enhance stability while learning.

Note that to enable DDPG policy explore better, noise is added to action. But still, it is the deterministic target policy which is evaluated.

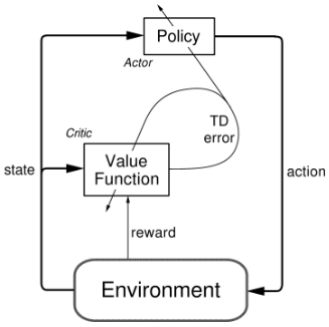


Fig. 2: basic Actor-Critic flow where Critic evaluates the new state due to TD error (Temporal-Difference).

III. Method

Before going further through the method applied in this project, several challenges have been raised. Many trials dealing with these obstacles and part of the method explained in *'Asymmetric Actor*

Critic for Image-Based Robot Learning' [7] have led to the method used in this project.

- Racing line

Basically, a racing driver follows a racing line which maximizes the overall pace on racetrack. It implies for some sequences of turns to 'sacrifice' the 1st curve by avoiding its apex in order to keep high speed when heading into the 2nd curve and to reach the highest speed at the exit as soon as possible.

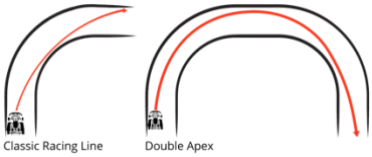


Fig. 3: examples of racing line conditioned to the number of curves and apexes (source: kartworldbelmont.com.au)

In terms of vision, this makes mapping input to action (output) more complex.
Case: suppose at turn 1 the racing line is on the left side of racetrack, at turn 2 on the right side, at turn 3 in the middle of the track to finally be on left side at turn 4. Without racing line, the context would be simpler: always in the middle of the racetrack.
Moreover, if you take into consideration the changing context off-track: grass, wall, deceleration surface, gravel, curbs presence or not, this does not help the agent model to generalize rapidly.

Racing Line True

Racing Line False

Fig. 4: vision rendering of True / False racing line positioning.

Thus, to bring more contextualization information, identical state_vision input to DDPG's Actor-Critic networks is avoided.
Instead, Asymmetric Actor-Critic is used as Actor network waits state_vision input and Critic network waits for state_motion input [7].

- Motion data

The chosen sim-racing API does not indicate localization of racing car within a racetrack Euclidean plan (simulated world Euclidean plan instead), nor distance from racetrack's edges or angle between racing car orientation and racetrack orientation.

This has led to consider Racing Line as a Goal that agent needs to tend to every meter of the racetrack. Then, it is used as input to Critic network and finally stored in the replay buffer [1][7] alongside state_vision, state_motion, action, reward, next_state_vision, next_state_motion.

- DDPG asymmetric Actor-Critic

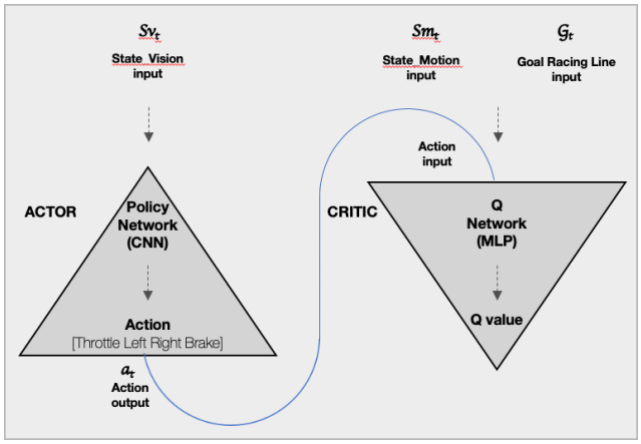


Fig. 5: asymmetric Actor-Critic mechanism in Bot88 case.

Algorithm DDPG from [1] with **asymmetric actor-critic** [7]

Randomly initialize critic network $Q(Sm,\ G, a|\theta^Q)$ and actor $\mu(Sv|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state_vision Sv_1, state_motion $Sm_1$and goal (racing line) G_1
  for t = 1, T **do**
    Select action $a_t = \mu(Sv_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t, observe reward r_t , new_state_vision Sv_{t+1}, new_state_motion Sm_{t+1}, new_goal G_{t+1}
    Store transition $(Sv_t,\ Sm_t, G_t,\ a_t, r_t,\ Sv_{t+1},\ Sm_{t+1}, G_{t+1})$ in R
    Sample random minibatch of N transitions $(Sv_i,\ Sm_i, G_i,\ a_i, r_i,\ Sv_i,\ Sm_{i+1}, G_{i+1})$ from R
    Set $y_i = r_i + \gamma Q'(Sm_{i+1}, G_{i+1},\ \mu'(Sv_{i+1}|\theta^{\mu'}) \mid \theta^{Q\prime})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(Sm_i, G_i, a_i \mid \theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(Sm, G, a|\theta^Q)|_{Sm=Sm_i,\ G=G_i,\ a=\mu(Sv_i)} \nabla_{\theta^\mu}\mu(Sv|\theta^\mu)|_{Sv_i}$$

    Update the target networks:

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

  end for
end for

Pseudocode 1: DDPG with asymmetric actor-critic adapted to Bot88.

Recall from Background C. that DDPG follows Off-Policy method:

- Actor network with $(Sv_t, a_t, r_t, Sv_{t+1})$ learns a deterministic policy $\mu(Sv|\theta^\mu)$ which followed a stochastic policy for good exploration only (behavioral policy),
- Critic network with $(Sm_t, G_t,\ a_t, r_t,\ Sm_{t+1}, G_{t+1})$ evaluates the deterministic target policy.

Basically, the mechanism is:
Raw RGB vision input of (635, 800, 3) shape is processed to be resized and outputted into (89, 120, 3) shape with a black mask on top third part of the image. This reduce irrelevant data.
In parallel, motion input of (15,) shape is retrieved from simulated environment's API and goal (racing line) input of (3,) shape is available from a preprocessed dataset which matches with racing car motion in real-time.

At each timestep t of each episode, DDPG algorithm receives state_vision Sv_t of (89, 120, 3) shape via Actor. Predicts action a_t which is added to noise $\mathcal{N}_t$ from Ornstein-Uhlenbeck process.
It observes r_t , Sv_{t+1}, Sm_{t+1}, G_{t+1} and stores $(Sv_t,\ Sm_t, G_t,\ a_t, r_t,\ Sv_{t+1},\ Sm_{t+1}, G_{t+1})$ in replay buffer R.
Critic estimates the deterministic policy due to state_motion Sm_t, goal G_t and action a_t.
Once full, a sample from R is used to enable Critic to learn which provides the action that maximizes action-value (performs gradient ascent) to Actor which learns.

IV. Implementation

As numerous motor sports companies test their innovations on its route, the chosen racetrack for experiments is: (*Barcelona) Catalunya GP*.

Combining accurate racetrack data, advanced replica racetrack rendering and relevant telemetry data, all developments have been made on sim-racing license *Project Cars 2*™.

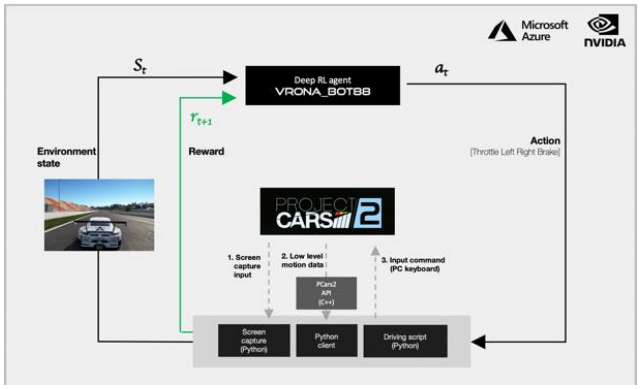


Fig. 6: materialization of the dedicated architecture and flow of this project.

A. Data

Nature	Name	Data Type - Units	Utilization	Origin
Vision input (Actor)	-	(800, 635, 3) resized to (89, 120, 3)	State actor	Real-time processing Data
Motion input (Critic)	Goal = racing_line	World Space X Z Y	State critic	Preprocessing data manual driving
	mWorldPosition	World Space X Z Y	State critic	PCars2 API with python client
	mOrientation	Pitch, Yaw, Roll (Euler angles)	State critic	
	mLocalVelocity	X Z Y (-inf, inf) Metres per-second	State critic and Reward	
	mAngularVelocity	Pitch, Yaw, Roll (radians / s)	State critic	
	mLocalAcceleration	X Z Y (-inf, inf) Metres per-second	State critic	
Other	mCrashState	Status 0 None; 1 Off-track; 2 Large prop; 3 Spinning; 4 Rolling.	Nextstep function + Monitoring	
	mCurrentLapDistance	(0, TrackLength) meters	Monitoring	
	mOdometerKM	(0, inf) meters	Monitoring	
	racing_line_delta	|racing_line – mWorldPosition|	Nextstep function + Reward	Real-time processing Data

Table 1 - Overall data (NB: PCars 2's 3D Euclidean axes order is X Z Y)

While sim-racing instance runs, the gameplay displayed on screen is captured in real-time due to Python script.
In parallel, telemetry data are delivered from Project Cars 2 API (C++) via shared memory. A dedicated python client helps to retrieve these motion data and other monitoring data.

Racing line coordinates have been gathered after manual driving and data have been preprocessed in order to create a real-time matching guideline to BOT88 wherever it is located on the racetrack.

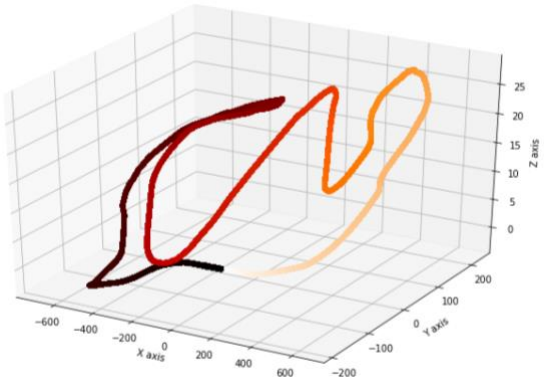


Fig. 7: 3D plot of ideal Racing Line coordinates retrieved from upstream manual driving. Data used later on to provide goal to Critic network and compute delta distance (between bot88 and goal) for NextStep function and Reward function.

B. Devices

One instance of *Project Cars 2*™ license has been installed on one Microsoft Azure Virtual Machine which has 6 cores CPU, 56GB memory and one Nvidia Tesla M60 GPU managed with Windows Server 2016 Datacenter OS.

C. Controller

BOT88 uses PC keyboard command to control the racing car. A dedicated script enables the agent to PressKey or ReleaseKey on the following keys: [↑ , ← , → , ↓]
which correspond to: [accelerating, left, right, braking]

Concretely, BOT88 agent outputs one vector of 4 elements. [Acceleration, Turn-left, Turn-right, Brake] where they value are all contain between 0.1 second (almost no action) to 1.5 seconds (of action). This particularity is from the fact that time sleep function is a component PressKey and ReleaseKey functions.

D. Network

- Actor

Is a composion of CNN network of 4 layers which embed 2 Conv layers with filters=64, kernel_size =2, strides= 1 and 2 Conv layers with filters=32, kernel_size =2, strides= 1 (with one bottleneck on the 3rd layer), all with relu activation function, MaxPooling and BatchNormalisation.

And 3 Dense layers: 150, 300, 300, all with relu activation function and BatchNormalisation.

It outputs (action) a concatenation of 4 unique Dense where accelerating and braking are computed with relu activation function and turn-left, turn-right are computed with 'tanh' with a kernel initializer=Variance Scaling of scale 1^{-2}.

- Critic

Is composed of three pathways: state motion, Goal (racing line) and action. They all have the same structure: 3 Dense layers: 150, 300, 300, all with relu activation function and BatchNormalisation.
They are merged to nurture a 600 Dense layer which nurture a last Q_value Dense layer.

- Reward function

Is as simple as: $r(S_t, a_t) = V - \alpha * |racing\ line\Delta|$
Or an alternative:

$$r(S_t, a_t) = V - \alpha * |racing\ line\Delta| - \beta * crashState$$

V is the forward velocity, $|racing\ line\Delta|$ is the absolute value of distance between BOT88 position and racing line (goal) position and $crashState$ is the crash state (1 for Off-road, 3 for spinning, …).

V. Results & Discussion

Due to trials and the hyperparameters settings (below), the average reward evolved in a promising way. It has been monitored, along 3 laps of 4655 kms each.

Algorithm	Hyperparameters	
Actor network	Adam optimizer	learning rate 0.001
Critic network	Adam optimizer	learning rate 0.0001
target policy & target Q function	Tau τ	0.01
	discount factor γ	0.98
Noise (Ornstein-Uhlenbeck)	mu μ	0
	theta θ	0.8
	sigma σ	0.3
Replay buffer (memory	Size	100000
	batch	32

Table 2 - DDPG's (asymmetric A-C) hyperparameters

These results were obtained with around 300 episodes which is very little and represents +1 hour of training.

We can notice that at every beginning and ending of the lap, the avg. reward is higher as the start/finish line is located in the middle of a long straight. This peak happens during another straight, as well.

During Lap 1 (blue), the agent acts pretty agilely but then the replay buffering and sample extractions runs intensively with all the computation underlying which slows down the action's execution. It creates a lag between two execution of Turn-Left and Turn-Right. The immediate impact is that the pace of the racing car is drastically reduced, and it tends to crash more often.

During Lap 2 (orange), the agent learns at slow pace and finally Lap 3 it converges a little bit more. One supposition is that after hours of training and exploration the policy converges slowly but surely.

Long short-term memory (LSTM) network has been tested as it makes sense to use memory to predict next state and next actions. But huge lags have been observed when training and the agent did not converge at all.

Advantage Actor-Critic (A2C) and Asynchronous Advantage Actor-Critic (A3C) would handle the LSTM with more comfort as they don't use replay buffer. They are algorithms which are probably more indicate to tackle this end to end control challenge on an external sim-racing environment with a 'restricted' hardware architecture.

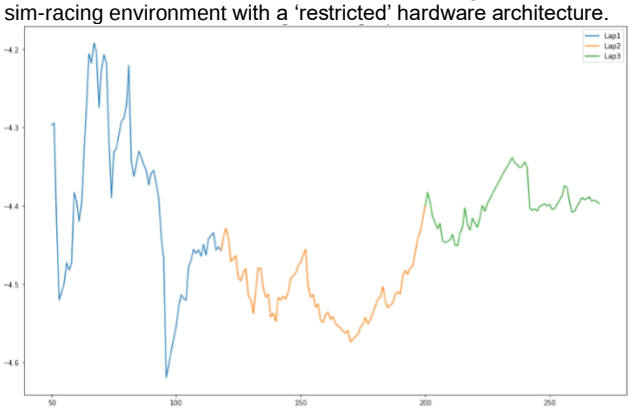


Fig. 8: Average reward over 3 laps in a row.

VI. Conclusion

End to end control is a hard challenge more over when focusing on raw pixel input only and trying to stick to a racing line trajectory.
Lateral control is the more complex task to handle, this is where LSTM could help in addition to a more specific work on vision input.
A more agile execution of action is key.
DDPG algorithm works perfectly and rapidly with TORCS environment and its dedicated data but within a context with more uncertainty and limited hardware architecture, it is more favourable to choose a different variant of Actor-Critic algorithm.
On the other hand, Multi-Agent DDPG can handle the task.

References

1 *Continuous_control_with_deep_reinforcement_learning*
Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra
https://arxiv.org/abs/1509.02971

2 *End-to-End Race Driving with Deep Reinforcement Learning – INRIA*
Maximilian Jaritz, Raoul de Charette, Marin Toromanoff, Etienne Perot and Fawzi Nashashibi
https://team.inria.fr/rits/files/2018/02/ICRA18_EndToEndDriving_CameraReady.pdf

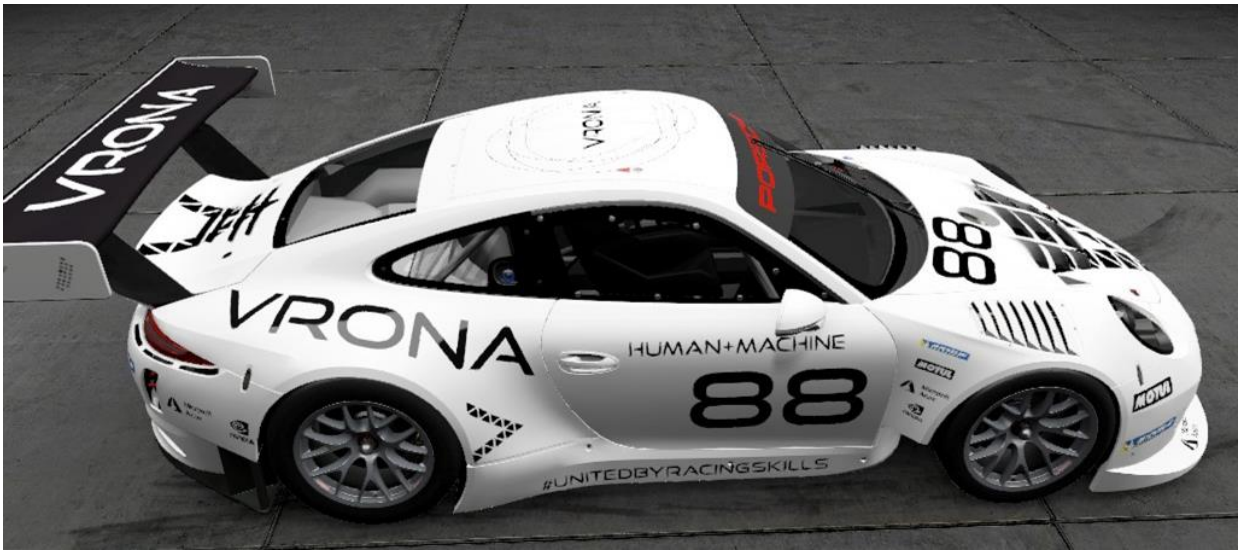
3 *Mastering the game of go with deep neural networks*
D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot
https://www.researchgate.net/publication/292074166_Mastering_the_game_of_Go_with_deep_neural_networks_and_tree_search

4 *Playing Atari With Deep Reinforcement*
V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller.
https://arxiv.org/pdf/1312.5602.pdf

5 *Deep Reinforcement Learning for Autonomous Driving*
SenWang, Daoyuan Jia, Xinshuo Weng
https://arxiv.org/abs/1811.11329

6 *Vision-based Deep Reinforcement Learning*
Anirudh Vemula, Debidatta Dwibedi
https://pdfs.semanticscholar.org/23cc/423a4c3c69b9470aac4f401f6225441a6e6d.pdf

7 *Asymmetric Actor Critic for Image-Based Robot Learning*
Lerrel Pinto, Marcin Andrychowicz, Peter Welinder, Wojciech Zaremba, Pieter Abbeel
https://arxiv.org/abs/1710.06542


VRONA
VRONA
HUMAN+MACHINE
88
#UNITEDBYRACINGSKILLS
88