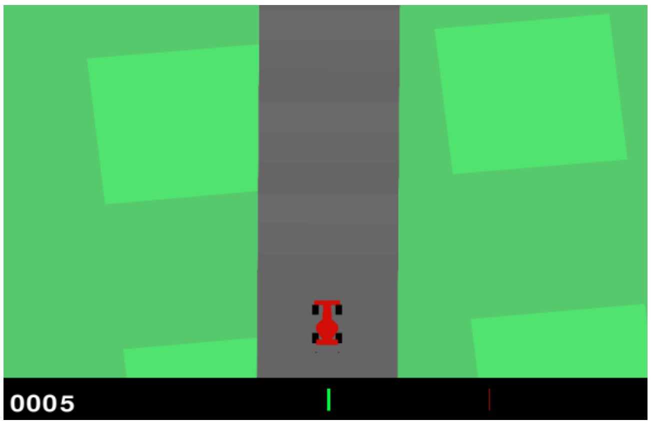
LIÈGE université
School of Engineering

Reinforcement Learning

Project : Racing car

Professor :
Ernst Damien
Assistant :
Louette Arthur

Authors :
Julien HANSEN
Adrien DE THIBAULT



1 Domain

latex2e The domain considered in this project is a simple car racing simulator from the `Gymnasium` library. This environment can be modelled as a Markov Decision Process (MDP), defined by the tuple :

$$M = (\mathcal{S}, \mathcal{A}, P, R, \gamma) \tag{1}$$

1.1 State Space

The state space $\mathcal{S}$ consists of observable image frames. At each time step, the agent observes a 96×96 RGB image of the car and the track. Since each image has 3 color channels and each pixel value is an integer in $[0, 255]$, the state space can be described as :

$$\mathcal{S} = \left\{ s \in \mathbb{N}^{96\times96\times3} \mid s_{ijk} \in [0, 255] \right\} \tag{2}$$

Where s_{ijk} is the pixel value. The initial state places the car at rest in the center of the road.

1.2 Action Space

The action space $\mathcal{A}$ is continuous and defined as :

$$\mathcal{A} = [-1, 1] \times [0, 1] \times [0, 1] \tag{3}$$

Each action $a_t \in \mathcal{A}$ is a 3-tuple : $a_t = (s_t, g_t, b_t)$, where $s_t \in [-1, 1]$ is the steering input at time t, $g_t \in [0, 1]$ is the acceleration (gas), and $b_t \in [0, 1]$ is the braking.

1.3 Transition Dynamics

The transition function P represents how the car moves on the track, defined by the physical equations defined in the environment. Since these are not required, they are not provided in this report.

1.4 Reward Function

The reward function R at each time step t is defined as :

$$r_t = \begin{cases} +\frac{1000}{N} & \text{if a new tile is visited at time } t \\ -0.1 & \text{time penalty at every time step} \\ -100 & \text{if the car leaves the track} \end{cases} \tag{4}$$

where N is the total number of unique tiles visited during the episode. If a new tile is visited at a given time step, the agent receives both the corresponding positive reward and the time step penalty simultaneously.

For the discount factor, we choose a value of 0.99 to encourage long-term planning by balancing immediate and future rewards.

The environment is deterministic, single-agent, and time-invariant, with partial observability since the agent only perceives the current image frame rather than the full track layout.

2 External Library Algorithm

The environment of this project has a continuous action space, partially observable environment with a high-dimensional observation space. These properties pose challenges such as the need for effective exploration, sample efficiency, and the ability to learn robust policies from visual input.

2.1 Methods

Because it is continuous, we decided to try different methods such as Proximal Policy Optimization (PPO), Deep Deterministic Policy Gradient (DDPG) or even Soft Actor-Critic (SAC). All methods are implemented in the library `stable_baselines3`.

PPO Due to the continuous action space, the possibility to perform on-policy learning, PPO was an obvious choice, as it is widely used in the literature. Its on-policy nature, while less sample efficient, can sometimes be advantageous in terms of reproducibility and learning dynamics. PPO also requires relatively minimal hyperparameter tuning to achieve reasonable performance, making it an excellent candidate for early-stage experimentation.

DDPG DDPG is a model-free, off-policy algorithm specifically designed for environments with continuous action spaces. It maintains separate actor and critic networks and uses deterministic policies, which can be beneficial in deterministic environments like CarRacing-v3. However, DDPG is sensitive to hyperparameters and exploration noise strategies. We may need to pay close attention when selecting those in order to get good results.

SAC Like DDPG, SAC is also an off-policy algorithm suited for continuous action spaces. However, SAC has the added advantage of stochastic policies and automatic entropy adjustment, which improves exploration, especially in environments with sparse or delayed rewards. SAC also benefits from better sample efficiency due to its off-policy nature and maintains greater training stability thanks to soft updates and entropy regularization. Given the high-dimensional observation space, SAC's robustness and ability to learn effectively with a replay buffer made it a suitable algorithm for our task.

2.2 Environment

We slightly modified the base environment for this project. The first added modification is to use a gray-scaled version. After that, we also use some frame stacking. This stacking is useful to add temporal information. Indeed, the decision of our agent is based on the observation of the screen. Without this stacking, the model will not be able to know in which direction the car is moving, what its speed is, if it turns, and so on. And finally, we transpose the image as it is needed to use a PyTorch CNN.

2.3 Training Phase

We train our models for a fixed number of steps. At regular intervals, defined by `eval_freq`, we evaluate the model's performance at that point in training. Evaluation is carried out over a set number of episodes (20 in this case), allowing us to compute the mean and standard deviation of the rewards. These statistics are useful for visualizing the training progress and assessing performance improvements.

If the mean reward from the current evaluation exceeds all previous results, the model is saved as the best so far. Regardless, the final model at the end of training is also saved. Ultimately, the version of the model used for deployment or further testing is the one that achieved the highest mean reward during training.

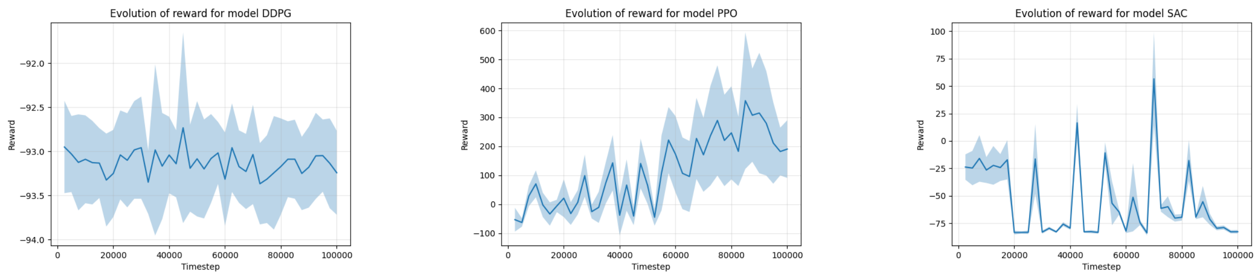


FIGURE 1 – Evaluation of the reward obtained for DDPG, PPO and SAC

3 Personal Algorithm

Following the results we obtained in Section 2, we decided to implement our own version of PPO and DDPG. Even though DDPG performed badly with the `stable-baselines3` implementation, we chose to re-implement it ourselves as other projects obtained good results with it.

Deep Deterministic Policy Gradient (DDPG)

The explanation and details provided in this section are inspired by **ddpg** [1] The DDPG implementation relies on four convolutional neural networks : two learning networks, an actor and a critic, and their corresponding fixed “target” copies. The actor network (θ) is a convolutional neural network (CNN) that processes the pre-processed image input through three convolutional layers, 32 filters at 8×8 stride 4, then 64 filters at 4×4 stride 2, and 64 filters at 3×3 stride 1 followed by a 512-unit fully connected layer. Its output heads produce a steering command via a tanh activation and an acceleration/brake vector via a sigmoid activation. The critic network (ϕ) shares the same convolutional feature extractor and a 512-unit state-encoding MLP, but then concatenates the action input and passes through a 256-unit MLP to yield a scalar

Q-value. The actor target and critic target networks are exact, frozen copies of θ and ϕ that are updated only via *Polyak averaging* (i.e. soft updates with factor τ) after each learning step, which greatly stabilizes temporal-difference targets. All networks use ReLU activations in their hidden layers, and both the actor and critic optimizers are Adam with learning rates of 1e-4 and 1e-3 respectively.

3.0.1 Action Selection

At each time step, the agent selects its action by passing the current observation through the actor network. To encourage exploration, we perturb this action using the *Ornstein-Uhlenbeck process* which generates temporally correlated noise. The relation between these noises is :

$$\epsilon_{i+1} = \epsilon_i + \theta_{ou}(\mu_{ou} - \epsilon_i) + \sigma_{ou}\mathcal{N}(0,1) \tag{5}$$

with θ_{ou} μ_{ou} and σ_{ou} set respectively to 0.2, 0.001, 0.25 .
The final action executed in the environment is then clipped to respect the action-space bounds of the *Car-racing* environment.

Prioritized Replay Buffer

Since DDPG is an off-policy actor–critic algorithm, it learns from transitions generated by any past policy. To store these transitions, we employ a replay buffer D of fixed capacity N, into which we store at each time step the transitions

$$(s_t,\, a_t,\, r_t,\, s_{t+1},\, d_t),$$

where $d_t \in \{0,1\}$ is the terminal indicator. Unlike a uniform buffer—which samples mini-batches of size B uniformly at random, we implement a *prioritized* replay buffer :

In our design, each stored transition i carries a priority

$$p_i = |\delta_i| + \varepsilon, \tag{6}$$

where δ_i is its temporal-difference error and $\varepsilon > 0$ is a small constant to avoid zero probability. Sampling is then performed according to

$$P(i) = \frac{p_i^\alpha}{\sum_{j=1}^{N} p_j^\alpha}, \tag{7}$$

with $\alpha \in [0,1]$ controlling the degree of prioritization ($\alpha = 0$ recovers uniform sampling). To correct for the bias introduced, each transition in the gradient update is weighted by an importance-sampling factor

$$w_i = \left(\frac{1}{N}\,\frac{1}{P(i)}\right)^\beta, \tag{8}$$

We linearly increase β from its initial value β_0 up to 1.0 during training, and after each update, we reset each sampled transition's priority to

$$p_i \;=\; |\delta_i| + \varepsilon.$$

This choice is motivated by the work of Yuenan Hou [2], who demonstrated that prioritized experience replay can significantly accelerate convergence, improve stability, and boost final performance compared to uniform sampling.

Algorithm 1 DDPG

```
 1: Initialize actor network $\mu_\theta$ and critic network $Q_\phi$, empty buffer $\mathcal{D}$
 2: Initialize target networks $\theta_{\text{tar}} \leftarrow \theta$, $\phi_{\text{tar}} \leftarrow \phi$
 3: for episode $= 1, 2, \ldots$ do
 4:     Initialize exploration noise process $\mathcal{N}$
 5:     Receive initial observation $x$
 6:     for $t = 1, \ldots, T$ do
 7:         Select action $u = \mu_\theta(x) + \epsilon$, where $\epsilon \sim \mathcal{N}$
 8:         Execute action $u$, observe reward $r$, next state $x'$, and done signal $d$
 9:         Store $(x, u, r, x', d)$ in replay buffer $\mathcal{D}$
10:         $x \leftarrow x'$
11:         if ready to update then
12:             for update $= 1, \ldots, U$ do
13:                 Sample minibatch $B = \{(x_i, u_i, r_i, x'_i, d_i)\}_{i=1}^{B}$ from $\mathcal{D}$
14:                 Compute target : $y_i = r_i + \gamma(1 - d_i) Q_{\phi_{\text{targ}}}(x'_i, \mu_{\theta_{\text{targ}}}(x'_i))$
15:                 Update critic by minimizing :
                        $$\mathcal{L}(\phi) = \frac{1}{|B|} \sum_{i=1}^{|B|} \left(Q_\phi(x_i, u_i) - y_i\right)^2$$
16:                 Update actor using the sampled policy gradient :
                        $$\nabla_\theta J \approx \frac{1}{|B|} \sum_{i=1}^{|B|} \nabla_u Q_\phi(x_i, u)|_{u=\mu_\theta(x_i)} \nabla_\theta \mu_\theta(x_i)$$
17:                 Update target networks :
                        $$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$$
                        $$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$
```

Proximal Policy Optimization (PPO)

Our PPO agent architecture is composed of a shared convolutional neural network (CNN) feature extractor followed by separate actor and critic heads. The agent processes preprocessed image observations through a CNN composed of three layers : 32 filters with kernel size 8×8 and stride 4, 64 filters with kernel size 4×4 and stride 2, and 64 filters with kernel size 3×3

and stride 1, each followed by a ReLU activation. The output is flattened and fed into both the actor and critic heads.

The actor network uses a Normal distribution over actions, whose mean is generated by an MLP composed of a 512-unit layer with ReLU activation followed by a linear projection to the action dimension. The log standard deviation of the action distribution is modeled as a trainable parameter vector, shared across states. The critic network is composed of a 512-unit fully connected layer with ReLU activation followed by a linear layer that outputs a scalar value function estimate. All layers are initialized using orthogonal initialization with a standard deviation of $\sqrt{2}$ except the final actor layer, which uses 0.01, and the final critic layer, which uses 1.

Action Selection

At each timestep, the agent samples a stochastic action from a multivariate normal distribution parameterized by the actor's output. The sampled action, clipped to respect the CarRacing environment's bounds, is executed in the environment. The log-probability of the action, the entropy of the distribution, and the value estimate are also computed and stored.

Advantage Estimation

To estimate advantages, we employ Generalized Advantage Estimation (GAE) with parameters γ (discount factor) and λ (GAE factor), computing :

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t), \tag{9}$$

$$\hat{A}_t = \delta_t + \gamma\lambda\hat{A}_{t+1}. \tag{10}$$

This method enables low-variance yet relatively unbiased estimation of the advantage function. The target return is then computed as :

$$\hat{R}_t = \hat{A}_t + V(s_t). \tag{11}$$

Optimization

Policy updates are done using the clipped surrogate objective :

$$L^{\text{CLIP}} = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right], \tag{12}$$

where

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{13}$$

is the probability ratio, and ϵ is the clip coefficient. This objective prevents large policy updates.

The value function is optimized using the squared error between predicted values and returns, with optional clipping :

$$L^{\text{VF}} = \mathbb{E}_t\left[\left(V(s_t) - \hat{R}_t\right)^2\right], \tag{14}$$

and the total loss combines the policy loss, value loss, and an entropy regularization term :

$$L = L^{\text{CLIP}} + c_1 L^{\text{VF}} - c_2 \mathbb{E}_t[\mathcal{H}[\pi(\cdot|s_t)]], \tag{15}$$

with coefficients c_1, c_2 controlling the balance between objectives.

Algorithm 2 PPO

```
1:  Initialize policy parameters $\theta$ and value function parameters $\phi$.
2:  Initialize $\theta_{old} \leftarrow \theta$.
3:  for $iteration = 1, 2, \ldots$ do
4:      Initialize empty set of trajectories $\mathcal{D}$.
5:      for $actor = 1, 2, \ldots, N$ do
6:          Run policy $\pi_{\theta_{old}}$ for $T$ time steps, collecting $\mathcal{D}_{actor} = \{(s_t, a_t, r_t, s_{t+1})\}_{t=0}^{T-1}$.
7:      Compute rewards $\hat{R}_t$ for all $t$ in $\mathcal{D}$.
8:      Compute advantage estimates $\hat{A}_t$ with Generalized Advantage Estimation (GAE)
9:      for $epoch = 1, 2, \ldots, K$ do
10:         for each minibatch $\{(s_t, a_t, \hat{R}_t, \hat{A}_t)\}_{t \in \mathcal{M}} \subseteq \mathcal{D}$ do
11:             Compute probability ratio
                    $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$    (16)
12:             Compute policy objective :
13:                 $L^{CLIP}(\theta) = \min\left(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)$    (17)
14:             Compute value function loss :
15:                 $L^{VF}(\phi) = (V_\phi(s_t) - \hat{R}_t)^2$    (18)
16:             Compute entropy bonus $S[\pi_\theta](s_t)$
17:             Minimize the total loss :
18:                 $L(\theta, \phi) = \hat{\mathbb{E}}_{t \in \mathcal{M}}\left[-L^{CLIP}(\theta) + c_1 L^{VF}(\phi) - c_2 S[\pi_\theta](s_t)\right]$    (19)
19:             Update $\theta$ and $\phi$ using gradient descent w.r.t. $L(\theta, \phi)$.
20:     $\theta_{old} \leftarrow \theta$.
```

Training Setup

Training occurs over multiple iterations. At each iteration, rollouts are collected across N parallel environments for T steps, forming a batch of size $N \times T$. The batch is divided into several mini-batches and shuffled. For each mini-batch, the agent is updated for multiple epochs. If enabled, the learning rate is linearly annealed across training steps. Observations are normalized to $[0, 1]$ and rewards are clipped to the range $[-10, 10]$. The Adam optimizer is used for all network updates with a learning rate set to 3e-4.

Additional Improvements and YAML configuration

We also develop another version of PPO with some implementation tricks presented in **ppoimplementatio**
. In this new version, we replace the normal distribution with a Beta distribution. This choice was motivated by the fact that the Beta distribution is naturally bounded in $[0, 1]$, making it particularly well-suited for environments with bounded action spaces such as *Car-racing* even though to scale the value return by the distribution to the actual action values range of the environment.

We also implement Frame stacking and GrayScale conversion using the luminosity technique, which is a weighted sum of the red, green, and blue channels. Similarly to our *Stable_ Baseline3* implementation.
Finally, to simplify the testing phase of our algorithms, YAML configuration file were designed, their purposes is to centralized every hyperparameters values in one place to faciliate the modification of a specific parameters if needed, these files can be found under the *cfg_ agent* folder.

4 Discussion

We compare the maximum rewards achieved by agents trained with Stable Baselines3 against those trained using our custom implementations. Table 1 summarizes these best-obtained returns.

Algorithm	**Stable Baselines3**	**Custom Implementation**
PPO classic	595	772
DDPG	-92.5	1
SAC	52	/

TABLE 1 – Comparison of best episodic rewards obtained by each algorithm.

In terms of training duration, the Stable Baselines3 implementations consistently converge faster than our custom agents. On average, PPO with Stable Baselines3 completed training in approximately 5 hours, whereas our custom PPO required around 6 to 7 hours to reach comparable performance. DDPG and SAC followed similar patterns, with our custom versions requiring roughly 20 to 30% more training time. These differences are likely due to the optimized C++ back-end and the parallelization features in Stable Baselines3, .