UNIVERSITÉ DE LIÈGE

Faculté des Sciences Appliquées

Reinforcement Learning Project

Florent Hervers s201931
Jimmy Walraff s201500

9 mai 2025

Année Universitaire : 2024-2025

Table des matières

1 Domain

The `CarRacing-v3` environment, provided by the OpenAI Gym library and built using the Box2D physics engine, is a continuous control task designed to evaluate reinforcement learning algorithms in a dynamic driving context.

Objective

The agent must control a car and maximize the cumulative reward by driving as far as possible along the track while staying within its boundaries.

Environment Structure

— **Track** : Each episode presents a procedurally generated closed-loop track composed of interconnected road segments with varying curvature and direction. This setup forces the agent to generalize rather than memorize.
— **Observations** : The agent receives a top-down RGB image of size $96 \times 96 \times 3$, representing the local area around the car. The image includes road tiles, grass areas, and red/white borders indicating the track edges.
— **Actions** : The environment has a continuous action space consisting of three components :
  — `steering` : $[-1, 1]$ (left/right steering),
  — `acceleration` : $[0, 1]$ (gas),
  — `brake` : $[0, 1]$ (braking).

Reward

The reward is composed of two components :
— A negative reward of -0.1 is applied at every frame to promote efficiency and discourage unnecessary actions.
— A negative reward of -100 is applied if the car go outside of the playfield.
— A positive reward of $\frac{1000}{N}$ is granted for every new track tile visited, where N is the total number of tiles in the track.

For example, if the agent completes the track in 732 frames and visits all tiles, the total reward is :

$$1000 - 0.1 \times 732 = 926.8$$

Episode Termination

An episode terminates if :
— The car go outside of the playfield.
— All track segments are visited (lap completed).

2 External Library Algorithm

We decided to use the `PPOLSTM` algorithm from the Stable Baselines3 library. This algorithm is a variation of PPO (Proximal Policy Optimization), which is known to perform well in many reinforcement learning tasks. In our configuration, an LSTM (Long Short-Term Memory) layer is added to the actor network only. When the LSTM is enabled, the network architecture is : `Input` $\rightarrow$ `CNN` $\rightarrow$ `LSTM` $\rightarrow$ `Linear` $\rightarrow$ `Output`. If no LSTM is used, the architecture simplifies to : `Input` $\rightarrow$ `CNN` $\rightarrow$ `Linear` $\rightarrow$ `Output`.

As we have access to a simulator of the environment thanks to the gymnasium library. As the environment have a continuous action space, policy-based algorithms are the ones to considered to solve the problems. We have the possibility to choose either off-policy or on-policy algorithm. We choose to go with on-policy algorithm as it's simpler to use. They are no need to think about how to collect the transitions to train the algorithm and the simulator is fast enough on the hardware available to us. For the on-policy algorithm choice, we have chosen PPO as it's considered as the best default on-policy for continuous domain environments. For the choice of library, we have chosen to go with stablebaseline3 as it provides a nice and complete documentation while being very easy to use and integrate with the environment and weight and biases. We have chosen `PPOLSTM`

after comparing results published by other users on HuggingFace [1]. The LSTM version of PPO showed better results than the standard PPO model [2]. We believe that the usage of the LSTM will help to understand the car dynamics as the LSTM have the possibility to retain information about the car state with his hidden state.

To prepare the input, we first resize each image to 64×64 and convert it to grayscale. We also stack frames and create multiple parallel environments using built-in functions. We ran 18 experiments, adjusting hyperparameters in order to evaluate the impact of the hyperparameters will believe as being important to train a good model. To avoid overloading the submission zip archive, we only provide the best performing models for question 2 and 3. To be as transparent as possible : every trained model can be found on our github `https://github.com/Florent-Hervers/RL-project/tree/main`).

In the 18 different configurations, we chose to vary five specific hyperparameters : max_episode_length, N_STEPS, N_ENVS, FRAME_STACK, and GAMMA. The first hyperparameter we update is the max_episode_length. It's the number of step before which the environment stops to avoid never ending episodes that are very likely to appear during the start of the training. By default, the value is set to 1000 steps which is very small when you take into account the fact that the model is running at 50 FPS. It's very unlikely that even an expert model will end in such a short time. We have then tried to set the value to 3000 steps, but a second problem arised. Due to the reward structure, the minimum possible reward is -400 (by going off the map at the 3000^{th} step without discovering any tiles) which is according to us, not enough. We decided to use the final 12000 steps that makes the cumulated reward more or less uniform when a stronger penalizing for the worse models. The model using 3000 steps was much more straightforward. It seems to not care at all of the track and just tried approximately to follow the track. We think that such behavior is due to the fact that due to the too small timelimit, it couldn't penalize the model for the few remaining tiles as the episode was cut too early. The model with 12000 steps is the complete opposite : he tried to follow the track much more carefully which results in a model that performs well on our ten tracks used for the evaluation.

Next, we have N_STEPS determines how many time steps are collected per environment before each policy update. Increasing N_STEPS can lead to more stable and informative policy updates, as it provides a longer temporal context and reduces the variance of the advantage estimates. The two values use for N_STEP are 256 and 2048. 256 was the default value we wanted to try and 2048 comes from the switch from config 5 and 6 where we wanted to reduce the number of environments from 8 to one while conserving the number of steps constant for the training of the models. That's how we came with the 2048 values and decided to stick with it as our high N_STEP value. The value of 410 for the N_STEP that can be seen in configs 9 and 10 came from the misunderstanding on how the frame stack works. We believe initially that the frame stacking performed the number of step of the stack and then provides the observations. As we wanted to conserve the number of steps for the update of the models, we came with N_STEP of $2048/5 \approx 410$. In practice, frame stacking works by caching the previous observation and to provides them along the new observation of the current step. As the reasoning that yield the value of 410 was wrong, we decided to not use this value once we understood our mistake.
Models with N_STEP set to 256 are very unstable, often alternates small turns even in straight lines, and was doing random action whenever the car when out the track. In the opposite, models with N_STEP set to 2048 are very stable, controlled and are almost never lost when going out of the track. This models seems like it is confident while the ones with less N_STEPS are much more shaky and uncertain of their actions.

The third important parameters is N_ENVS refers to the number of parallel environments used to collect data. Raising N_ENVS helps speed up data collection, reduces variance by averaging over more trajectories, and improves sample efficiency. For example, when we use only one environment, the resulting model was staying at the border of the track. This illustrate that the model only learned the minimal behaviour to work as the whole car doesn't need to be on the track to discover tiles. But this yield to poor generalization as the car misses much more tiles when they are lots of small turns in the tracks. Using 4, 8 or 12 environments during the training allows to model to see many type of turns and tracks and yield better model as much more cases were

1. `https://huggingface.co/sb3/ppo_lstm-CarRacing-v0`
2. `https://huggingface.co/vukpetar/ppo-CarRacing-v0-v3`

covered during the training. However, it comes at the cost of higher CPU or memory usage.

An other advantage of the two previous paragraphs is that we increase the number observations provided to train the model. Increasing the number of data on which we train the models is a great advantage, deep learning models like the LSTM and the CNN are often better as they are trained on more data. More accurate models will provide more accurate predictions which can help to enhance the quality of the training.

The next hyperparameter studied was the GAMMA, the discount factor that determines how much future rewards influence the current decision. A higher GAMMA (closer to 1) encourages long-term planning, which is important for environments where delayed rewards are critical. The value chosen 0.9975 to reach more or less the same penalty using 2048 steps than using a GAMMA of 0.99 with 512 steps. Conversely, a lower GAMMA prioritizes short-term gains, which might speed up convergence but could result in suboptimal long-term behavior (helps learning not to cut a turn). The chosen value was 0.98 to not reduce too much the reward compared to the best configuration. Unfortunately, these experiments didn't improve the resulting model. Increasing the gamma made the model not follow the track whenever the turn was too sharp and decreasing the gamma made the model much less stable than the best one.

Finally, we tried to modify the FRAME_STACK. It controls how many consecutive frames are stacked together to form a single observation. A higher value provides the agent with temporal information about motion, which is especially useful in partially observable environments. This allows the agent to better infer velocity and direction (learning how to turn), but also increases the dimensionality of the input and the computational cost. Our experiments showed that the frame stacking did help to better control the speed of the car as is looked like the model became much more careful to not go to fast. Unfortunately, these models seems like to forget that their goal is to discover 95% of the track. It often miss some tiles which prevent them to finish the track. It's a bit disappointing as these models drives objectively way better than without frame stacking. It is maybe cause by a too short training : as we increase by a lot the number of data during each training, the 2 million horizon may be too little for the model to learn the perfect model. We also remark that the model using 4 frames stacking seems to be less qualitative than the one using 2 frames despite seeing the opposite in the quantitative results.

All other parameters were kept constant across all configurations. We used tanh as the activation function and applied orthogonal weight initialization to help prevent vanishing gradients during training. A linear learning rate decay strategy was employed, starting from an initial learning rate of 0.0001. This approach gradually reduces the learning rate throughout training, enabling the agent to perform larger updates early on and more precise refinements in later stages. The following parameter's value were set and not modify (inspired from the blog linked during TP5 [3] and online configuration linked in the begining of this section) : ENT_COEF, N_EPOCHS, VF_COEF, CLIP_RANGE, GAE_LAMBDA and MAX_GRAD_NORM.

Final Configuration

To determine the best configuration for ppolstm algorithm, we evaluate every model on ten tracks not used during the training during a maximum of 12000 steps to evaluate whether the model has truly learned how to drive and finish the tracks. The summary of the evaluation can be found in the Results/Q2 folder of the archive. The best model was selected by taking the best mean summed reward on the 10 evaluation tracks. The best configuration was the following one (which correspond to configuration 13 in the Results/Q2 folder) :

- **Total timesteps :** 2,000,000
- **Max episode steps :** 12,000
- **Environments :** 12
- **Observation preprocessing :**
  - Resize to 64×64
  - Convert to grayscale (channel dimension preserved)
  - Frame stack : 1

3. https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/

— **PPO Hyperparameters :**
  — Learning rate : 0.0001 (linear schedule)
  — Batch size : 64
  — Steps per update : 2048
  — Number of epochs : 10
  — Discount factor (γ) : 0.99
  — GAE lambda : 0.95
  — Clip range : 0.2
  — Value function coefficient : 0.5
  — Entropy coefficient : 0.0
  — Max gradient norm : 0.5
  — Normalize advantage : True
— **Policy Network Configuration :**
  — Activation function : Tanh
  — Orthogonal initialization : True
  — LSTM hidden size : 128
  — Initial log standard deviation : -2
  — Optimizer ε : 1e-5
  — Critic LSTM : Disabled

The results of on the 10 evaluation tracks are presented in the table 1 along with the standard deviation of the 10 runs, the maximum and minimum summed reward.

Track	**Reward**
1	689.89
2	907.40
3	414.58
4	753.04
5	894.81
6	874.21
7	545.59
8	868.71
9	611.55
10	752.94
Overall Mean	**731.27**
Standard Deviation	**157.75**
Minimum	**414.58**
Maximum	**907.40**

TABLE 1 – Performance results for PPOLSTM (stable baseline3)

3 Personal algorithm

For the custom algorithm, we wanted to implement a on-policy algorithm for the reasons explained in the previous section. As there not really a better version of the PPO algorithm that we could implement, we have thus decided to implement an older algorithm. We have decided to implement the the A2C (Advantage Actor-Critic) algorithm. As we have seen this algorithm during the theoretical lectures and seems to be suitable for the car-racing environment. To include an other source of information, we also used the original paper presenting A2C [4] to better understand the algorithm. The first version tried to implement the algorithm presented in details in the paper.

A2C is an actor-critic method, which means that we train two components : a policy (the actor) and a value function (the critic). Both are represented using neural networks. The actor takes an observation (state) as input and outputs a probability distribution over possible actions. The critic also takes the same observation and outputs a single value, which is the expected return (future reward) from that state.

The actor is trained using policy gradients. This means it is encouraged to increase the probability of actions that lead to higher rewards. The critic is trained to predict the actual returns,

4. `https://arxiv.org/pdf/1602.01783`

helping the actor to learn more effectively by reducing variance in the updates.

Our neural network architectures include a convolutionnal neural network (CNN) to process visual observations, as our environment provides image-based states. CNNs are well suited for extracting spatial features from images, which helps the agent understand the visual structure of the environment. After the CNN, we use fully connected (linear) layers to map the extracted features to outputs. We also experimented with adding an LSTM layer between the CNN and the linear layers, but only for the actor. The goal was to allow the policy to capture temporal dependencies in the sequence of observations. This is particularly useful in partially observable environments, where a single frame may not contain enough information to make optimal decisions, and memory of past observations can improve performance.

The training process of our A2C (Advantage Actor-Critic) implementation follows the typical n-step update scheme used in many on-policy reinforcement learning algorithms. At a high level, the agent interacts with the environment for a fixed number of steps, stores the collected experience, and then performs a single update of its neural networks using that batch of data. This process is repeated until a predefined number of total timesteps is reached.

During training, multiple parallel environments are used to improve data efficiency and stabilize learning. For each update cycle, the agent collects data over a horizon of n steps in all environments simultaneously. At every step, the actor selects an action based on the current observation, while the critic estimates the value of that state. The environment returns the next observation, reward, and information about whether the episode has ended. All relevant data (including observations, actions, rewards, log-probabilities, and value estimates) are stored for further processing.

To compute the advantage function, which guides the learning of the actor, we experimented with two approaches : using the Temporal Difference (TD) error and using Generalized Advantage Estimation (GAE). GAE helps to balance bias and variance in the advantage estimates by incorporating both immediate and future rewards. Once the n-step episodes is completed, the advantages are computed accordingly and then normalized to stabilize the policy gradient updates.

The actor is trained by maximizing the expected return using policy gradients, which are computed by weighting the log-probabilities of the chosen actions with the estimated advantages. The critic is trained to minimize the error between the predicted value and the actual return. Additionally, an entropy term is added to the total loss to encourage exploration by preventing the policy from becoming too deterministic too early. These three components : actor loss, value loss, and entropy bonus are combined into a single loss function that is used to update the model via backpropagation and gradient descent. To improve stability, we apply gradient clipping and optionally reduce the learning rate over time using linear scheduling.

Regarding the choice of hyperparameters, we kept most of them consistent with those used in Question 2. We experimented with two different approaches for computing advantages : Temporal Difference (TD) and Generalized Advantage Estimation (GAE). Additionally, we upgraded our network architecture by introducing LSTM layers in the actor network. For this recurrent architecture, we tested configurations with 1 and 3 LSTM layers, and hidden sizes of 128 and 256. When verifying the script during the redaction of the report, we have discovered major mistakes in the code. The lstm states weren't properly managed during the training which impact the models during the training. The results obtained with LSTM, that are better than without, aren't thus considered in the reasoning of section 4 due to the mistake in the scripts. It makes no sense to reason about models which results from mistaken algorithm.

We also focused on modifying some of the most impactful parameters, namely N_ENVS, FRAME_STACK, and N_STEPS. All other hyperparameters were once again adopted from the blog linked during TP5 and a online configuration [5]. The first and only goal driving the change of parameters was to search for a configuration that succeed in training a working model using the A2C algorithm. Unfortunately, we couldn't achieve any correct results using the algorithm. We never make the algorithm succeed to finish a track completely. The models didn't really try to follow the track but it rather try to spin on itself, a bit like the "donuts" we can see in the classical motor sports. The model is thus far from trying to complete the track. We discuss about

5. `https://huggingface.co/sgoodfriend/a2c-CarRacing-v0`

the reasons of this behavior in the section four.

Final Configuration

We reuse the same evaluation protocol than in the previous section. The best model was still selected by taking the best mean summed reward on the 10 evaluation tracks. Despite being trained with the faulty LSTM training algorithm, the best configuration was the following one (which correspond to configuration 12 in the Results/Q3 folder) :

- **Total timesteps :** 2,000,000
- **Max episode steps :** 12,000
- **Environments :** 12
- **Observation preprocessing :**
  - Resize to 64×64
  - Convert to grayscale (channel dimension preserved)
  - Frame stack : 1
- **PPO Hyperparameters :**
  - Learning rate : 0.0001 (linear schedule)
  - Steps per update : 512
  - Discount factor (γ) : 0.99
  - GAE lambda : 0.95
  - Value function coefficient : 0.5
  - Entropy coefficient : 0.1
  - Max gradient norm : 0.5
  - Normalize advantage : True
- **Policy Network Configuration :**
  - Activation function : Tanh
  - Orthogonal initialization : True
  - LSTM hidden size : 256
  - LSTM layers : 3
  - Initial log standard deviation : -2
  - Optimizer ε : 1e-5
  - Critic LSTM : Disabled

The results of on the 10 evaluation tracks are presented in the table 2 along with the standard deviation of the 10 runs, the maximum and minimum summed reward.

Track	Reward
1	-1190.48
2	-629.96
3	-865.39
4	-1165.31
5	-1106.21
6	-1088.91
7	-1200.15
8	-626.70
9	-1133.14
10	-1128.27
Overall Mean	**-1013.45**
Standard Deviation	**212.04**
Minimum	**-1200.15**
Maximum	**-626.70**

Table 2 – Performance results for A2C (own implementation)

4 Discussion

Both algorithm shares some common properties : they are both actor-critic methods that computes advantages and they share the same loss function for the critic networks. The main difference between the algorithm resides in the loss used for the actor model. For A2C, the loss to maximize

is

$$\mathbb{E}_t \left[\log(\pi_\theta(a_t|s_t))\hat{A}_t \right] \tag{1}$$

where $\hat{A}_t$ is the estimation of the advantage used in the algorithm. For PPO, the loss to maximize follows the following expression (from the PPO paper) [6] :

$$\mathbb{E}_t \left[min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t,\ \ clip\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon \right)\hat{A}_t \right) \right] \tag{2}$$

Where ϵ is a hyperparameter to be chosen.

Now let's study the behavior of the two loss to understand their differences. To start, as $\pi_\theta(a_t|s_t)$ define the probability of taking action a_t given the state s_t. The values of $\pi_\theta(a_t|s_t)$ must be in the interval [0,1] by definition of a probability and it implies that $\log(\pi_\theta(a_t|s_t)) \in [0; -\infty[$. Now, let's get back to the equation (1). As the equation is very simple, we have only two cases arises : when $\hat{A}_t$ and $\log(\pi_\theta(a_t|s_t))$ are negative, the loss is maximized when $|\log(\pi_\theta(a_t|s_t))|$ is as big as possible indicating that the probability $\pi_\theta(a_t|s_t)$ should be close to zero. In other word, if the advantage is negative and thus the action taken isn't a good one, we look to minimize it's value. The second case arise when $\hat{A}_t$ is positive. As $\log(\pi_\theta(a_t|s_t))$ is still negative, the loss is maximized when $\log(\pi_\theta(a_t|s_t))$ is close to zero meaning that $\pi_\theta(a_t|s_t)$ is close to one. If the action lead to a positive advantage, then the probability of taking this action should be maximized.

As the expression of the loss described in (2) is more complex than the loss of A2C, we have more cases to analyze in order to fully understand the loss used in PPO. For the sake of brevety, the notation $r_t(\theta)$ is introduced and defined as $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} = r_t(\theta)$. Unlike in PPO the term containing the probabilities, $r_t(\theta)$, must stay in the interval $[0, \infty[$. The trivial case where $\theta = \theta_{old}$ (ie no update was made) yield to a ratio $r_t(\theta_{old}) = 1$ for every time step t. When $r_t(\theta) \in [1-\epsilon, 1+\epsilon]$, then both terms in the min function are the same and the loss to maximize can be reduce to

$$\mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t, \right]$$

The behavior of the optimization when the update of the policy is small is as follow. When $\hat{A}_t$ is negative, $r_t(\theta)$ should be close to zero meaning the updated model should decrease the probability of taking this action. When $\hat{A}_t$ is positive, $r_t(\theta)$ should be as big as possible meaning the probability of taking the action should be bigger than in the old model. Now lets study the case where $r_t(\theta) \notin [1-\epsilon, 1+\epsilon]$, we can remark that due to the min expression, the clipping isn't used in the cases where the unclipped ratio make the expression $r_t(\theta)\hat{A}_t$ smaller than the clipped version. It means that the clipping isn't used when the update of the model is large and don't change the probabilities in the correct direction. This make sure that when the update is very wrong, the update is not canceled by the clipping (by setting the gradient to zero) and so it ensure that the model can correct itself. The clipping does only affect the loss when the update is large and change correctly the probabilities. The clipping makes such the gradient of the loss is zero and thus it prevent any updates to the model when in that case. The only way to update the model in the proper direction is by making updates that are small to fall back in the first case described above.

Now that the loss are well understood, we can use that knowledge to try to understand why we aren't able to use A2C to train a successful model on the Car Racing environment. We believe that the main reason is that the reward structure of the environment is very sparse. The model is always penalized except when discovering new tiles of the tracks. As the reward is provided since the car went on the tiles whatever the direction and the speed of the car. If the model discover the first tile when turning slightly to the left, the A2C model increase the probability of turning left. But as the update is very large, the model became very biases toward turning left early and keep forcing this behavior with makes the car do these donuts we can see on almost all A2C models. PPO in the other side, will only perform updates when the ratio is small enough. It will thus prevent doing big update unlike A2C which shield the model from bad behavior learned too quickly. The model learned by PPO will thus learn very progressively until a point where the model is close enough from a good policy to begin to integrate the good action to perform given the state.

6. `https://arxiv.org/pdf/1707.06347`