Reinforcement Learning Agents on Gym Car-Racing Environment

Félix CHALUMEAU, Ilan COULON, Pierre FERNANDEZ

***Abstract*—This report describes and explores different designs of autonomous agents on the OpenAI gym Car-Racing-v0 environment, which is a random 2D top-down racing environment. The problem is very challenging since it requires computational resources, given the high state space. Here, we will compare different algorithms to solve the environment.**

I. INTRODUCTION

Our goal is to compare different reinforcement learning strategies to get the best score at Car Racing. Using extensive parametrization and empirical simulations, we will show and discuss the performance obtained, providing insight and reflections and we will compare best scores as well as learning time.

The specificities of this environment are (see part III for more details):

- a high dimensionality observation/state space: 256^{27648}, similar to other atari games.
- a continuous action space: [Steering, Gas, Breaks].

Many decision processes are not suitable for this kind of environment, given that the observation space has high dimensionality. We decided to focus on Deep Q-Learning as well as Direct Policy with gradient and genetic algorithms to optimise the policy. One key point of this project is the simplification of the observation space into a smaller one, giving new algorithm's possibilities.

Our code can be found here.[1]

II. BACKGROUND AND RELATED WORK

Gym is a Python library created by OpenAI. It is composed of a collection of environments designed to test and develop reinforcement learning algorithms. Whether it is to learn this technique or to perfect its algorithms, Gym allows its users to focus only on the algorithmic and not on the design of the environment which can sometimes prove to be time-consuming. Among the many environments available are games (Taxi Driver, Frozen Lake, the Atari suite, etc.) but also 2D or 3D robotic simulations. The Car-Racing algorithm seemed like a good target to focus on reinforcement learning algorithms, since the environment has continuous features, that can be or not be discretized, while having a fun video-game-like side.

Before going into our precise work, let us introduce the two big types of agents that we have worked with: direct policy and deep Q-Learning, which are both popular to deal with huge state spaces.

[1]Code found here: https://github.com/felixchalumeau/INF581-project.

A. Deep Q-Learning

Deep Q-Learning is a very popular algorithm which generalise basic Q-Learning. As a reminder, Q-Learning is an off-policy algorithm [1]. This means that it doesn't estimate the Q-function of its current policy but it estimates the value of another policy which is the optimal one. To do so, it uses the following update rule:

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(r + \gamma \max_b Q_t(s',b) - Q_t(s,a))$$

where at a certain time step s is the state, s' the next state, r the reward, α is the learning rate, and γ is the discount factor.

In Deep Q-Learning, instead of storing values in a table, we use a neural network to learn those values:

$$Q(s,a) = q_w(s)_a$$

where q_w is a (deep) neural network parametrized by w [2].

The steps are the following ones:

Algorithm 1 Deep Q-learning

1: Preprocess and feed the observation to the DQN, which will return the Q-values of all possible actions in the state
2: Select an action a using the ϵ-greedy policy
3: Perform this action in a state s, move to a new state s' to receive a reward r and store (s,a,r,s') in the memory
4: Select some random batches of transitions from the memory do gradient descent steps, computing the loss, known as: $L = (r + \gamma \max_b Q_w(s',b) - Q_w(s,a))^2$

There exist some improvements to this algorithm. Amongst them, the Double Deep Q-Learning (DDQN) in which another network is used to prevent overestimates of the Q-value. Indeed, the simple DQN agent is trying to fit a target value that it itself defines and this can result in the network quickly updating itself too drastically in an unproductive way. To avoid this situation, the Q-values to update are taken from the output of the second target network which is meant to reflect the state of the online network but does not hold identical values. The target value becomes: $r + \gamma Q_w(s', \text{argmax}_b\, Q_{w'}(s',b))$.

An other improvement of this method, consists in combining Imitation Learning and Deep Q-Learning. The method used in this paper looks similar to the one one can find on [3], called DQfD. The idea is to use user data to generate the tupple (s,a,r,s') mentioned before. In other words, it uses the same principle as in DQN, but instead of choosing the action using an ϵ-greedy policy, it chooses it from the user experience.

B. Direct Policy Search

In a Direct Policy search technique, the idea is to learn directly the best policy instead of learning the values of the states. This technique is known to work better when the spaces are continuous (observation and action) [4]. The policy is defined thanks to a parameter θ and two different techniques were used to find the parameter that will maximise the final reward [2].

1) The Policy Gradient algorithm: The idea relies on maximising the "expected" reward following a parametrized policy : $J(\theta) = E\pi[r(\tau)]$ Then, under performance metric $J(\theta)$, we want to do gradient ascent: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

A classic way to do so is to use a Policy Gradient algorithm called REINFORCE [5]. It is based on the policy gradient theorem, which tells that

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta(s,a) Q^{\pi_\theta}(s,a)]$$

and on the use of a stochastic gradient ascent and replacing $Q_{\pi_\theta}(s_t, a_t)$ by a Monte Carlo estimate $R_t = \sum_{t'=t}^{H} r(s_{t'}, a_{t'})$ over one single trajectory.

Algorithm 2 REINFORCE

Initialise θ arbitrarily
while True **do**
  Generate an episode $a_1, s_1, r_1, a_T, s_T, r_T$
  for each step $t \leq T$ **do**
    $G = \sum_{k=t+1}^{T} R_k$
    $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) G$
  end for
end while

2) Evolutionary method: The evolutionary algorithm used here is the most basic one [6]. The idea is to pick N random θ. Then, for each generation, keep the T best candidates (by evaluating $J(\theta_i)$ $\forall i \in \{1, ..., N\}$) and mutate them with a slight random perturbation in order to get a new generation of N candidates.

In order to perform the random perturbation, we just add $\sigma \mathcal{N}(0, I)$ to every weight (with $\sigma = 0.02$ in our case).

In order to avoid losing some really good candidates, an *elite* is chosen (the best candidate) and kept for the next generation without any mutation.

Algorithm 3 Genetic Algorithm

Initialise θ and σ (noise standard deviation) arbitrarily
while True **do**
  for $i \in 1, ..., n$ **do**
    Compute $J(\theta_i)$
  end for
  Keep the T best θ_i
  Create N new θ by mutating the T candidates
  Keep the best θ_i without mutating it as the *elite*
end while

III. The Environment

As mentioned before, the Car-Racing-v0 environment of Gym OpenAI is a common environment because of both its simplicity and its complexity. That is to say the actions and the observation are quite easy to understand but many different methods can be implemented and tested in such an environment. Here are the reasons why:

A. Classical Gym environment

- State space : As we can see on figure 2, the agent can only see a 96×96 RGB grid, including the score bar with information such as the current score, or the controls
- Action space: what the agent gives is a vector of 3 numbers $(s, a, d) \in [-1, 1] \times [0, 1] \times [0, 1]$ corresponding respectively to the steer (-1 is hard left and $+1$ is hard right), the acceleration and the deceleration.
- The reward is given by -0.1 for every frame and $+1000/N$ for every track tile visited, where N is the total number of tiles in track (a tile is the colour discontinuity that can be seen in the photo). As an example, if there are 300 tiles, the car has seen all the tiles, and the game finishes at frame 100, then the score is $300 \times \frac{1000}{300} - 0.1 \times 900 = 910$
- The environment is deterministic and partially observed.
- The main challenge of this environment is the very high dimension of the observation space. But we will see how this issue can be tackled.
- This kind of environment can lead to real-world application for AI in games but also for self-driving cars (of course the observation space would be a lot bigger).

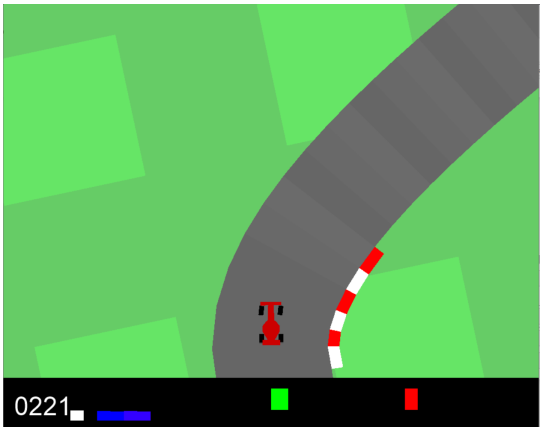

Fig. 1. Observation of the environment

B. Our personal modifications

First, we decided to simplify our environment as done in [7]. Indeed, all the channels of the image are not important, since the green one encodes much of the information the agent needs. Moreover, the score is useless for the agent. Thus, we only take a small part around the car. Then, we removed the colours by having only pixels that are the road (1) or that are the grass (0). This was done by taking the green part of the

observation and taking only the pixels that would be above a certain threshold (150).

We also created a simplified action space by discretizing it the previous one. In this action space, the agent only has 4 actions available:

$$\mathscr{A} = \{\text{left}, \text{right}, \text{accelerate}, \text{nothing}\} \quad (1)$$

The problem then becomes a classification problem and will be closer to what we have seen during the INF581 course. This is the space that is used by the Deep Q-Learning agent.

Since *Direct Policy Search* is known to work better on continuous action spaces, the discretization will not be used for those methods.

IV. The Agents

As a start, we chose to try many different agents, but all of them will not be presented in details because they were not efficient. Amongst them, the double deep Q-Learning and a neural-network based policy approach. We later focused on 3 different agents. The two first ones (Policy Gradient and Evolutionary) do not use all the information the observations provide but may be efficient and less costly. The third one (DQN agent) would be the *state-of-the-art* technique but is actually very expensive to compute. In this section, we discuss why we select them this way, how we implemented them, and what are their main advantages and drawbacks.

A. The Direct Policy agents

As mentioned, the agents do not use all the provided information. The main idea was to reduce the observation space to a smaller one. To do so, we created "detectors" placed on the car. Those detectors create 4 numbers, the distance to the grass to the left, to the right and two distances for the upper right and the upper left corner of the car. Those 4 detectors allowed us to reduce the 96x96x3 input shape to a simple vector of size 4.

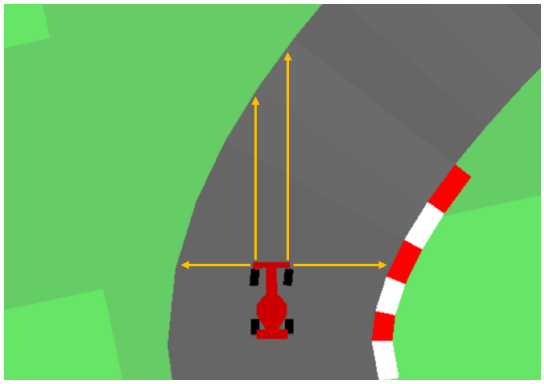

Fig. 2. Visualisation of the values measured by the detectors

Moreover, to reduce the action space in the first approach, we got rid of the break and acceleration process. The agent accelerate once every four frames and only has to decide whether it should go to the right or to the left. Again, it allowed to simplify the methods for the first approach.

1) Policy gradient: As the model is very simple (only 8 numbers for θ: the measures captured at actual and previous instants), we added some features to the agent behaviour in order to improve the learning process. For instance, the episode is stopped whenever the car gets outside of the road for too long, or it gets negative rewards when it touches the grass. These modifications were made for 2 main purposes:

- make the learning process faster
- prevent bias in the learning because of a too long duration in the grass

Finally, there are other parameters that had to be taken into account. Amongst others, the horizon H that is present in the Monte Carlo estimate formula of II.B. and the learning rate α had a great importance.

- The horizon represents how many steps should be used to compute the rewards, it is quite similar to the use of a discount factor γ. By choosing a too big horizon H, rewards that are not related to an action could be associated. By choosing a too low one, the car would not be able to anticipate turns.
- The choice of α is also a compromise. The higher the α the faster is the learning process. However, if it is too low, the solution will hardly converge, so we had too choose a middle value for α. We used a little decay as well.

In short, the PG agent is an agent that is easy to compute given that it uses a 4-coordinates vector to encode the information data and that learns quite fast some interesting behaviours, as we will see later on.

2) Evolutionary algorithm: The model used to map the observations to the actions was a one-hidden-layer neural network with a ReLU activation and a sigmoid in the end in order to get two numbers: the steer and the acceleration.

The number N of candidates per generation and T of candidates kept did not have such an impact on the results so were kept at $N = 30$ and $T = 5$.

The main limiting factor here was the execution of the episodes. Since evolutionary algorithms are highly parallelizable, we launched several environments in parallel and this trick made the training much faster.

Since our environment has some part of randomness (each track is randomly generated), $J(\theta)$ was evaluated 3 times for the selection of the best T candidates and 5 times for the elite selection. Evaluating more times would have been more accurate but was much slower. Plus, evaluating twice as much was tried and did not have that much impact on the selection process.

The reward function did not need to be changed.

B. The Deep Q-learning agent

The Deep Q-learning is here supposed to be the more suitable agent for this kind of environment[7]. Indeed it is a state-of-the-art method[2] that works pretty well in very large observation spaces thanks to the use a neural network. However, the training of this neural network, especially since ours takes images as input and has to use convolutional

networks to gives sufficient results, is a really expensive process.

For this agent, we first used the *keras-rl* library to speed up the coding process even if some code in it is deprecated. The results were not at all convincing so we decided to recreate a DQN agent from scratch.

To predict the Q-values, the chosen neural network is composed of two convolutional layers followed by 2 dense ReLu-activated layers. The final layer is a linear activation and the loss function is the mean squared error. The neural network was coded using the tensorflow keras library. (cf. Fig. 3)

```
Model: "sequential_20"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_39 (Conv2D)           (None, 20, 20, 32)        2080
_________________________________________________________________
dropout_39 (Dropout)         (None, 20, 20, 32)        0
_________________________________________________________________
conv2d_40 (Conv2D)           (None, 9, 9, 64)          32832
_________________________________________________________________
dropout_40 (Dropout)         (None, 9, 9, 64)          0
_________________________________________________________________
flatten_20 (Flatten)         (None, 5184)              0
_________________________________________________________________
dense_40 (Dense)             (None, 128)               663680
_________________________________________________________________
dense_41 (Dense)             (None, 4)                 516
=================================================================
Total params: 699,108
Trainable params: 699,108
Non-trainable params: 0
_________________________________________________________________
```

Fig. 3. Our deep neural network architecture

The main advantage is the fundamental non-linearity of the model and this could lead to very interesting and efficient strategies. On the contrary, the huge disadvantage is how expensive it is in term of computation.

Once again, the results were not convincing, since the agent only learned how to accelerate but not how to turn. Therefore, we decided to "teach" the agent with sort of a DQfD algorithm:

- With a python version of the game, users can play and the data is collected.
- Then, instead of letting the agent play itself by taking an action, we directly train the q-network by selecting (s, s', a, r) from the user database.
- In several learning epochs, the agent learns a coherent Q-Value.

V. Results and Discussion

Although there are some major improvements to make, we already have convincing results for both types of agents.

A. Direct Policy search

1) Policy gradient: The policy gradient agent already showed some specificities and interesting behaviours:

- It learns really fast. Three episodes are enough to learn to the car that it should turn when the road turns, and it quickly gives score near to 300 (still far from the 900 needed for the environment to be considered solved)
- Given that the agent is only able to choose between right or left, it leads to a quite unnatural oscillation effect of the car. This would be removed or decreased if we added the action "go straight forward" or if the speed was given as an input.

We could have spent more time on this agent but we knew, as already mentioned, that this wasn't the best agent for this environment. For instance, we could have moved on a neural network and have add inputs (more detectors and the current speed). In any case, this first simple approach provided interesting results and an average performance of 314.5 with only 5 minutes of training. Some performances reaching more than 380. This approach is also a good baseline to evaluate our other agents.

2) Evolutionary: The different rewards during the selection are shown Fig. 4. We see that we hit a plateau around 400 after around 10 generations.

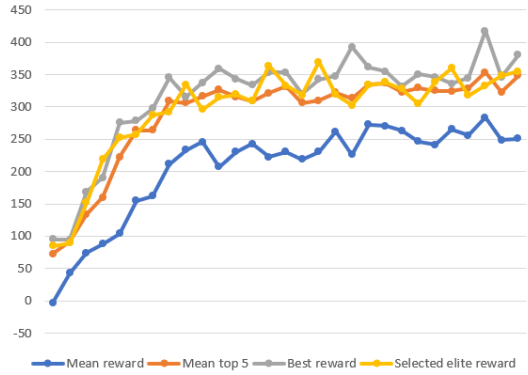

Fig. 4. Reward evolution during evolutionary process

Watching the agent play, we notice that this plateau is caused by the fact that after some distance, the car has too much speed so it can't take too sharp turns. As seen with the Policy gradient technique, this is without any doubt caused by the fact that the agent has no way of knowing its current speed. Therefore, we can guess that this agent could perform better if we included some speed informations in its observations.

B. The Deep Q-Learning agent

This agent is by far the one that gives the best results. Moreover, it is still quite quick to train (less than 30 minutes). Here is an overview of the training process:

We can see that the dropout layers, here to prevent from overfitting the user data, have a negative impact on the learning process (which is coherent since what is learned is partially unlearned at each step).

The final trained agent achieve an average score of 814.0 (on an average of 100 trials), which is far from the 900 needed to solve the game, but is already quite a good score.

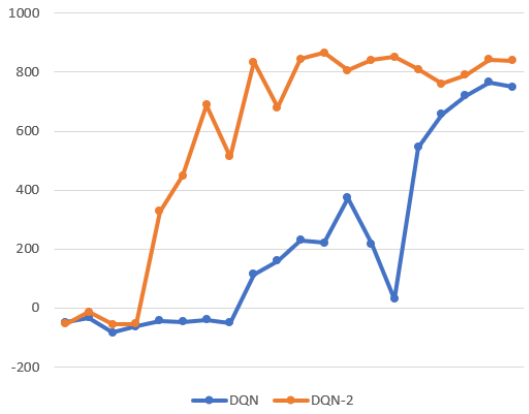

Fig. 5. Reward evolution during the learning phase of 2 different DQN agents: the score was computed on an average of 10 games, every 50 epochs of the learning process for a batch size of 100 frames. In blue with dropout layers at 0.3, in orange with dropout layers at 0.1.

C. Another (Unsuccessful) Agent

As mentioned before, some agents were not successful. Amongst them, an agent that learned the policy directly from a neural network. This neural network was similar to the one used by the DQN agent, but used a categorical cross-entropy loss and encoded actions as one hot vectors (i.e. [0,1,0,0] represents action 2). We realised that the agent could not work since for a given state, the user acts intermittently (doing accelerate - do nothing on a straight road for instance). Therefore, the network can not classify states in actions.

D. Sum Up

To sum up the interesting results, here is a sum up of the obtained average results, computing on 100 games:

Policy Gradient	Evolutionary	Deep Q-Learning
314.5	352.8	814.0

VI. Conclusion and Future Work

To conclude with, we already achieved good results (with the Deep Q-Learning agent particularly). They showed that, conditionally upon having a way to collect user data and to play the game, the Deep Q-Learning algorithm trained with user data is by far the one with more advantages. It learns really fast (compared to the 30-hours needed with a simple DQN [7]), and approaches the top results of the environment with a score of 814.0. In addition, we have shown with the Direct Policy Agents that our reduction of the state space is encouraging, and, even though the results are not as good as the DQN ones, they demonstrated the importance of the preprocessing part.

However, there is still many methods that seem promising according to the literature and the lectures, that we have not tested out; and some others that might work by changing some assets of our algorithm: for instance the Double Deep Q-Learning.

References

[1] As mentioned in Lecture VI - Reinforcement Learning II. *INF581 Advanced Topics in Artificial Intelligence*, 2020.

[2] As mentioned in Lecture VII - Reinforcement Learning III. *INF581 Advanced Topics in Artificial Intelligence*, 2020.

[3] Hester et al. Deep Q-learning from Demonstrations, *Google DeepMind*, 2018 https://arxiv.org/pdf/1704.03732.pdf

[4] El-Fakdi A. et al., *Direct Policy Search Reinforcement Learning for Autonomous Underwater Cable Tracking* https://doi.org/10.3182/20080408-3-IE-4914.00028

[5] As mentioned in Reinforcement Learning Lab III: Policy Gradient. *INF581 Advanced Topics in Artificial Intelligence*, 2020.

[6] Felipe Petroski Suc et al., *Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative forTraining Deep Neural Networks for Reinforcement Learning* https://arxiv.org/pdf/1712.06567.pdf

[7] Pablo Aldape et al. Reinforcement Learning for a Simple Racing Game, *Stanford University*, 2018

[8] Zoltán Lőrincz, *A brief overview of Imitation Learning* https://medium.com/@SmartLabAI/a-brief-overview-of-imitation-learning-8a8a75c44a9c