Reinforcement Learning
M2 X-HEC 2020

Team Lovelace

Thomas de Mareuil - Paul-Emile Dugnat - Paul Lafforgue

Tommy Tran - Etienne Windels

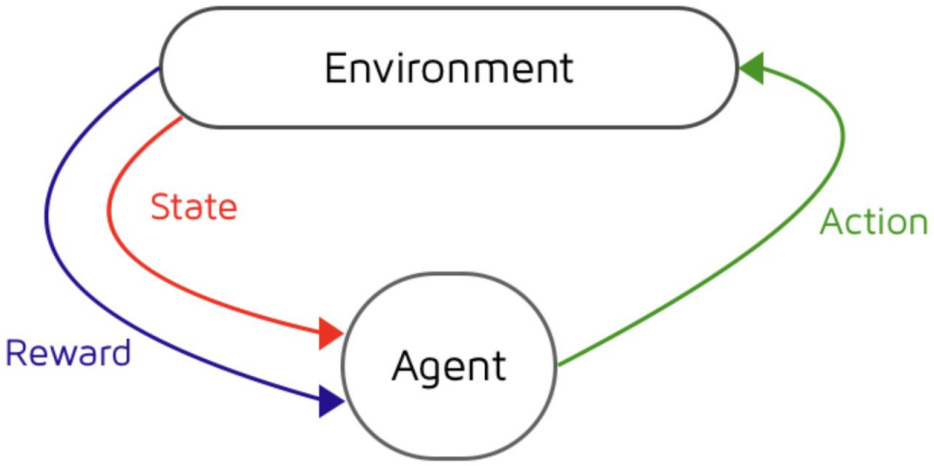
I. **Problem stating**

The goal of this hackathon was to successfully land a SpaceX rocket on its ship.

To do this, we trained a RL agent using the PPO (Proximal Policy Optimization) algorithm, tesnorforce and the OpenAIGym toolkit.

a. Setting

For this project, we put ourselves in a Reinforcement Learning setting. As pictured in the diagram below, our Agent (the rocket) navigates through an Environment (the simulation). It performs Actions which change its state, and receives variable rewards. Learning happens by finding the policy that maximizes total rewards.



b. Actions

For each step in the environment, the agent carries out actions based on the current state it is in. In this project, the rocket can take actions modeled by the following variables:

- **Thrust** : float between 0 and 1, corresponding to the main engine power
- **Left(/Right) booster** : float between 0 and 1, corresponding to lateral engines power
- **Gimbal** : float between -1 and 1, corresponding to the orientation of the rocket power.

c. States

In a simple environment, states could be represented by a simple matrix, while more complex environments could use the pixel data from the screen as the current state. In our case, the state

corresponds to the rocket's positioning in the air, as well as it's current speed and trajectory, modelled by the following variables:

- **Angle, x and y**, corresponding to the orientation and position of the rocket in the environment (horizontally and vertically)
- **distance** : float between 0 and 1, corresponding to the euclidean distance between the ship's center and the rocket.
- **first/second leg contact** : boolean, corresponding to whether or not the first/second leg of the rocket has touched the ship
- **velocity x/y**, corresponding to the speed of the rocket (negative when going away from the ship, positive when going towards the ship)
- **angular_velocity** : the rate at which the rocket turns around its center (negative when turning left, positive when turning right)
- **landed** (boolean) if the rocket has touched the ground, and finally **landed_full** if it has stopped moving for more than 60 time steps.

d. Rewards

The rewards are the feedback the agent gets from interacting with the environment. We explain the way we shaped our reward function in section III. The agent receives a positive reward for landing closer to the target zone, and a negative reward for moving further away from the landing zone. Also, the agent receives a small negative reward every time it carries out an action. This is done in an attempt to teach the agent to land the rocket as quickly and efficiently as possible. If we were to simply give it a reward for landing the rocket, the agent could do it, but as often in reinforcement learning, it could learn a sub-optimal strategy - e.g. take much longer than it should, using excess fuel, just because there is no downside to doing so!

II. Agent parameters

To improve training, we tweaked the hyperparameters which govern the agent's learning. To be honest, as we had little prior knowledge about what parameters were best, we adopted a test-and-learn approach and looked online for best practices. We ended up using a PPO agent with the following main parameters:

```
discount = 0.99
learning_rate = 1e-5
batch_size = 10
likelihood_ratio_clipping = 0.1
```

exploration = 0.1

A discount factor (gamma) very close to 1 (here, 0.99) makes sure that learning considers "looks far into the future", and considers future rewards with greater weight. We also found out by experience that learning rates one order of magnitude lower than the default PPO parameters yielded better results. We also chose to leave some exploration possibilities to the agent (0.1, vs 0 by default) in order to avoid getting caught in suboptimal policies. For the other parameters, we went through several experiments and took time to understand the PPO method, and we ended up keeping the default values.

III. Reward shaping

Reward shaping was a big deal in this project, and concentrated most of the work. Our reward shaping strategy started from the postulate that learning would be more efficient if we provided rewards to our rocket **at each time step**, and **not only when it lands successfully**. Therefore, we tried to shape rewards that provide **gradual feedback** to the agent and let it know it's getting better and closer to the desired final state. A key to making this work was being able to **decompose** the reward function into relevant components.

a. Rewards in case of failure

Before all, we started by setting the reward value to 0, and by defining a 'groundcontact' variable, based on the pre-coded 'first_leg_contact' and 'second_leg_contact' variables.

```
groundcontact = first_leg_contact or second_leg_contact
reward = 0
```

We then considered that the simulation ends up in a failure when the rocket doesn't land fully, which includes either ground contact (the rocket touched the platform but didn't stabilize) or no ground countact (the rocket missed the platform). Another case with no ground contact is simply when the rocket is still flying. For all these scenarios, we decided to allocate negative rewards (details and code below):

- When the rocket is **still flying**, we want to incentivize it to **go towards the center** and to **stabilize**. Therefore we decrease the reward mainly based on rocket **angle** and **horizontal distance** from the center of the platform. We also decrease the reward based on **velocity** and **angular velocity**, in order to avoid the rocket arriving too fast and crashing on the platform. As the rocket accumulates negative points at each time step, it also has to stabilize as quick as possible and to limit actions in order to avoid penalties.

- In case of **ground contact**, we set the reward to 0.5 (as ground contact is good, base reward is positive), and we subtract a value depending on angle, horizontal distance, velocity and angular velocity, i.e. the variables we want to bring to 0 (ingredients for a successful landing!). We clip this value to 1, so that we can't go under -0.5.

- In case of no ground contact and y distance = 0, i.e. **missed platform**, we allocate the **lowest possible reward**: -1.

```
if not landed_full:

    if groundcontact:
        # case in which the rocket landed (one or both legs), but didn't stabilize (broken).
        # -> we set the reward to 0.5 (as ground contact is good), and substract a value depending on angle,
        # horizontal distance, velocity and angular velocity, i.e. the variables we want to bring to 0
        # (ingredients for a successful landing!). We clip this value to 1, so we don't go under -0.5.
        reward += (0.5 - min(1, (velocity + abs(angle) + abs(x) + abs(angular_velocity))))
        print('\rlanded improperly: {}'.format(reward), end='')

    elif (not groundcontact) and (y == 0):
        # case in which the rocket missed the platform altogether.
        # -> worst case, reward set to -1
        reward += -1
        print('\rmissed platform: {}'.format(reward), end="")

    else:
        # case in which the rocket is still flying.
        # -> we want to incitate the rocket to go towards the center and to stabilize, so we
        # start from reward = 0 and we substract a value that we want to be minimized. We clip
        # this value to make sure the reward doesn't go under -1.
        reward -= min(1, (abs(x) + abs(angle) + abs(angular_velocity) + throttle/y))
        print('\rflying: {}'.format(reward), end='')
```

b. Rewards in case of success

We considered that the simulation ended up in a success when the rocket lands successfully, i.e. stabilizes on the platform for more than 60 time steps. However this includes again 2 situations :

- The rocket **didn't land in the center of the platform**: we set the reward to 1 (success) and we subtract a value depending on the **distance from the center** of the platform, but **not going under 0**.

- The rocket achieved a **fully successful landing**: we chose to **over-reward** this situation, in order to make sure that the reward **supersedes all the negative rewards accumulated over the previous time steps**. We wanted our rocket to learn that this situation is strongly superior to the others. Through trial and error, we observed that the usual number of time steps in the simulation was 400-450, clipped to -1 per time step, so we set the top reward to 500.

```
if landed_full:

    if distance > 0:
        # case in which the rocket didn't land in the center.
        # -> it's a success: we set the reward to 1 and we substract a value depending on
        # the distance from the center of the platform, but not going under 0
        reward += (1 - min(1, abs(x)^2))
        print('\rlanded uncentered: {}'.format(reward), end='')
    else:
        # full successful landing, right in the center!
        # -> Highest reward, +1
        print('\rlanded successfully: {}'.format(reward), end='')
        reward += 500
```

c. Things that we tried but didn't work out so well :/

A strategy we tried was to use an **exponential increase for rewards**. Indeed, something we thought was very important was that the first epochs are far less important than the last one, in that it is possible to recover from a bad start, whereas it is not possible to land if the last steps are bad. In order to account for this, we designed an exponential function with a very low rate and we weighted the raw reward by this factor. We also tried directly to **weight the reward with the value of the time step**. We thought this type of processing would help the rocket focus on the last rounds and improve success rate, but it yielded disappointing results.

Another idea we had was to have each reward **depend on the value of the previous reward**, in order to **over-penalize bad strategies** and to push the agent to focus on good strategies (i.e. strategies with high rewards for several time steps in a row). However, in order to transmit the value of the reward from each time step to the next one, we would have had to adapt a number of elements in the environment, and we lacked some time to do so in the short time frame of the project.

IV. Results

Despite the previous steps and experiments that we described, we ended up in first place of the hackathon (starting from the end :/).

We identified two reasons that we think might be the cause of our rocket's under-performance:

- A reward function **too complex** that didn't allow us to fully understand how it was affecting the rocket's behaviour, despite the fact that we built it as methodically as possible, starting with basic actions and scenarios.

- As we were all discovering the topic of reinforcement learning, we did not manage to **split the work** in a way that would have allowed us to improve performance more efficiently. We were also a little destabilized by the fact that the Reinforcement Learning approach is more related to **"test and learn"** than what we usually do - there is no secret sauce, you just have to try and adjust things until it gets better ! (aren't we reinforcement learning agents ourselves, in the end..?)

As a conclusion, we think that this project was a very good occasion to see the challenges of working as a group on a totally new topic. We learned about how important **communication** is on such a project, especially in the face of new methods that we discover together. As we were all "learning by doing", we saw that communication was key to make sure that individual findings and practical knowledge was regularly shared within the group. In the end, our rocket did not perform exactly as expected, but we were glad to "get our hands dirty", and this project definitely motivated us to look deeper into Reinforcement Learning in the future!