Reinforcement Learning Project

Victor BARBERTEGUY, Marc-Antoine OUDOTTE, Clément GARANCINI

***Abstract*—In this project, we created a new unity environment inspired by the Rider game, to implement an agent and train it with reinforce algorithm. The agent's observations include its position, velocity, and angle along the z-axis. We penalize the agent if it falls or fails to reach the goal and reward it if it succeeds. We face some challenges with the current implementation, such as the long time it takes to play an episode and the difficulty of training the agent with REINFORCE, which we plan to address. We also plan to implement the DQL algorithm and fine-tune the parameters in the following weeks.**

I. Introduction

The use of RL in games has been skyrocketing over the past few years, and efficient algorithms can now solve environments with continuuous observation spaces. There are well-known environments in gym which have such observation spaces. This is the case for the CartPole Environment [1] or with the Mountain Car Environment [2], where a car has to efficiently accelerate forward or backward to reach the top of a hill. Our approach is to create a new environment inspired from the Mountain Car environment in which a motorbike has to cross obstacles to reach a goal. The motorbike has only two actions: do nothing or accelerate forward. The main feature of this game is that the motorbike will accelerate if it is on a slope, but will turn on itself if it is in the air (the mechanics of the game is similar to that of KETCHAPP's Rider game). In such games (RL with MarioKart [3] or obstacle games), the model tends to accelerate everytime to reach the goal as fast as possible, or introduces a notion of energy consumption as a constraint (like the Mountain Car). The mechanics of our game thus induces a new challenging tradeoff for such RL model:

1) Reach the goal as fast as possible

2) Flip and land backwards (i.e failing) if it accelerates too much

We will study how REINFORCE and Deep Q-Learning (DQL) algorithms will solve the Rider environment and we will discuss their efficiency.

To that end, we built from scratch an environment in Unity [4] where a human player can control the car and get hands on the game's mechanics.

Once the game was functional, we implemented the ml-agents [5] framework to define our agent settings (action and observation spaces, reward shaping). Even though the ml-agents package contains some RL algorithms (such as Proximal Policy Optimization), we did'nt use them as they did not fit well to our continuuous observation spaces. We preferred to use the 'UnityToGymWrapper' to wrap our unity environmnent into a gym environment. In this gym environment, we could implement the REINFORCE algorithm studied in [1] to train our agent.

We implemented only one simple level and trained our agent with REINFORCE and Deep-Q Learning (DQL). Furthermore, we tested our environment with `StableBaselines3`[6] algortihms (state-of-the-art Reinforcement Learning (RL) algorithms) to compare with our implementations.

You will find all the files in the github (there is a README that explains how to run the files).

II. Background

A. Building the environment

Each action chosen will modify the state of the agent, thus changing its observation that will induce a reward defined by the **reward function**. In our case, we penalize the agent (giving him negative rewards) if he falls or if it lands backwards. Likewise, we penalize it if it has not reached the goal at the end of the episode. On the contrary, we reward our agent if he gets closer to the goal, and a bonus if it solves the episode; This is all the preprocessing that we need to implement our learning algortithm.

The following section will first detail how we built our 2D environment. This 2D environment is a game with which an agent (the motorbike) will interact and try to reach its goal. Thee placed at an initial position on a slope and its goal is to reach the flag (*fig. 1*) without falling in the holes or landing backwards on a slope after a jump. To do so, the agent has only two possible actions: do nothing or accelerate forward (which will make it flip if it is in the air (*fig. 2*)). When playing the game in human mode, the bike will accelerate with a left click on your mouse/trackpad. The bike has also access to some information about its environment. As a logical prolongation of [2], the agent observations will be: its position (x and y), its velocity(vector of size 2) and its angle along the z axis (to get its rotation).
Each action chosen will modify the observations and will induce a reward according to a **reward function**. In our case, we penalize the agent (giving him negative rewards) if he falls, or at each step if he doesn't reach the goal (as in [2], in order to get the agent finishing the level as fast as possible) and we reward it if he reaches the goal, and a bonus if it solves the episode;
This is all the preprocessing that we need to implement our learning algortithm.

Fig. 1. The first level of our game

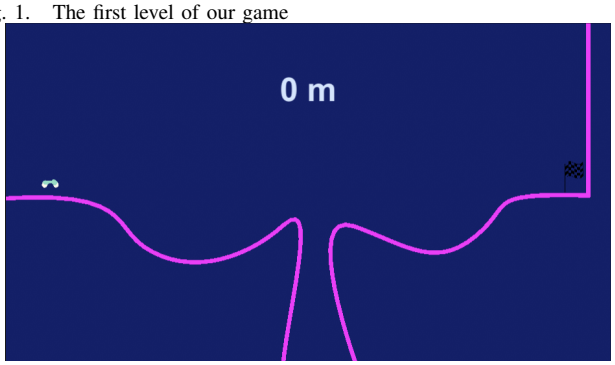
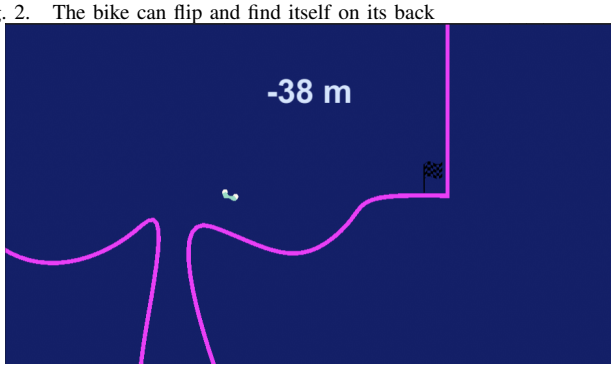


Fig. 2. The bike can flip and find itself on its back



B. RL algorithms

As our observation space is continuous, we had to carefully choose our algorithms. Using techniques such as Q-Learning or SARSA is unthinkable because state's space is too wide (we can't calculate the Q-table nor use dynamic programming). We decided to implement gradient descent - namely **REINFORCE algorithm** and **Deep Q-Learning**.

1) REINFORCE algorithm [extracted from [1]]: This method performs gradient ascent in the policy space so that the total return is maximized.
We will restrict our work to episodic tasks, *i.e.* tasks that have a starting state and last for a finite and fixed number of steps H, called horizon.

More formally, we define an optimization criterion that we want to maximize:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{H} r(s_t, a_t)\right],$$

where $\mathbb{E}_{\pi_\theta}$ means $a \sim \pi_\theta(s, .)$ and H is the horizon of the episode.

In other words, we want to maximize the value of the starting state: $V^{\pi_\theta}(s)$.

The policy gradient theorem tells us that:

$$\nabla_\theta J(\theta) = \nabla_\theta V^{\pi_\theta}(s) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\ Q^{\pi_\theta}(s, a)\right],$$

With

$$Q^{\pi}(s, a) = \mathbb{E}^{\pi}\left[\sum_{t=1}^{H} r(s_t, a_t) | s = s_1, a = a_1\right].$$

Policy Gradient theorem is extremely powerful because it says one doesn't need to know the dynamics of the system to compute the gradient if one can compute the Q-function of the current policy. By applying the policy and observing the one-step transitions is enough. Using 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, we end up with a special case of the REINFORCE algorithm (see Algorithm below).

REINFORCE with Policy Gradient theorem Algorithm

Initialize θ^0 as random
Initialize step-size α_0
$n \leftarrow 0$
WHILE no convergence
  Generate rollout $h_n \leftarrow \{s_1^n, a_1^n, r_1^n, \ldots, s_H^n, a_H^n, r_H^n\} \sim \pi_{\theta^n}$
  $PG_\theta \leftarrow 0$
  FOR $t = 1$ to H
    $R_t \leftarrow \sum_{t'=t}^{H} r_{t'}^n$
    $PG_\theta \leftarrow PG_\theta + \nabla_\theta \log \pi_{\theta^n}(s_t, a_t)\ R_t$
  $n \leftarrow n + 1$
  $\theta^n \leftarrow \theta^{n-1} + \alpha_n PG_\theta$
  update α_n (if step-size scheduling)
RETURN θ^n

2) DQL [extracted from [1]]: TODO The main idea behind Q-learning is that if we had a function $Q^* : State \times Action \rightarrow \mathbb{R}$, that could tell us what our return would be, if we were to take an action in a given state, then we could easily construct a policy that maximizes our rewards:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s, a) \tag{1}$$

However, we don't know everything about the world, so we don't have access to Q^*. But, since neural networks are universal function approximators, we can simply create one and train it to resemble Q^*. To address this issue, an option is to make use of the power of deep learning to estimate the Q-value of a state-action pair even if it was never encountered before. A deep neural network is used to estimate the Q-value of each possible action in a given state.

This is the update rule for DQN:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Initialize $\mathcal{A}$ the agent neural network
Copy $\mathcal{A}$ weights to initialize the target neural network $\mathcal{T}$
Initialize batch_size B
For i in nb_episodes
While not done
Generate experience

$exp \leftarrow [s_1^1, a^1, r^1, s_2^1, done^1, \ldots, s_1^n, a^n, r^n, s_2^n, done^n]$
If len(exp) $>$ B
Sample minibatch $\sim$ exp
$q_1 \leftarrow \mathcal{A}(s_1)^B$
$q_2 \leftarrow \mathcal{T}(s_2)^B$
q_target = reward_batch $+ \gamma \cdot q_2 \cdot (1 -$ done_batch)
Update $\mathcal{A}$
Every j iterations, $\mathcal{A} = \mathcal{T}$

III. Methodology/Approach

A. Preprocessing the environment

1) Game objects: We used Unity [4] to create an environment similar to the KETCHAPPs game Rider. The objects to create are quite simple (slopes and a bike) but have properties that requires to be set with attention. The velocity is a key parameter in the training of the agent: is if the bike accelerates too fast, it will jump directly to the goal. The slopes must be designed in order tu put the agent in difficulty. This first level (*fig 1*) is only a jump but if the car only accelerates, it will flip and possibly end on the back and die. As the new feature of this environment (compared to [2]) is to be able to jump off the slope and flip, the gravity and the torque are also crucial. The coding of the game is similar to this YouTube tutorial [7] but we set the parameters by tuning and testing them.

Then, we had to tranform this gaming environment into a RL environment thanks to the `mlagents` package. This consists in creating a new script `agent` which overrides the Agent class of `ml-agents`. We override some functions to set the observation and action spaces, as well as the reward function. The action space is discrete (either 0 or 1), but the observation space is an array containing :

position x: *(float)*,
position y: *(float)*,
velocity: Vector2 *(float)*,
angle: *(float)*.

We built three different levels for the game (*img. 3*):

1) **Level 1** a simple hole to reach the goal. We target no precise feature of the game.
2) **Level 2 - Speed Control**: no holes or jumps (so no risk to land backwards), but the agent has to control its speed very carefully
3) **Level 3 - Flip control**: there are three holes: the agent can speed once on the slopes, but has to control its torque when it is in the air to land in the right position.

2) Reward Shaping: Tweaking the reward has played a crucial role in this study, as changing some features of the reward functions had striking consequences over the performance of the agent. We successively implemented these rewards:

1. The reward function **r1** has three cases for an action `a`

$$\begin{cases} \text{-if the goal is reached, `r(a) = 10`} \\ \text{-if the agent falls in the hole or land backwards (collides with slope), `r(a) = -10`} \\ \text{-if it is still running the game, `r(a) = -1`} \end{cases}$$

This reward is almsot the same as the Mountain car. This does not work very well because with this, the agent wants to solve the env fast (to avoid penalties) so tends to accelerate and lands backwards (speed vs flip)

2. The reward function **r2**

$$\begin{cases} \text{-if the goal is reached, `r(a) = +100`} \\ \text{-if the agent falls in the hole or land backwards (collides with slope), `r(a) = -50`} \\ \text{-if it is still running the game,} \\ \text{`r(a) = this.position - goal-position`} \end{cases}$$

As the goal of this study is to get the car going as far as possible (like in the real KETCHAPP game), it is a bit cheating to give him a goal . Yet we kept the idea to give greater reward for reaching the goal and a greater penalty when dying (giving bonus rewards of the same range as the game rewards makes it harder for the agent to understand its goal/what it has to avoid).

3. The reward function **r3**

$$\begin{cases} \text{-if the goal is reached, `r(a) = +100`} \\ \text{-if the agent falls in the hole or land backwards (collides with slope), `r(a) = -50`} \\ \text{-if it is still running the game,} \\ \text{`r(a) = -this.position + initial}_{position}\text{`} \end{cases}$$

This is better than the previous (it only knows where it is witht relation to the initial position), but the agent faced a crucial problem: it had great poisitive reward (+80 near the goal) so when reaching the goal, it gained +100 but it wasn't worth it compared to waiting near the goal and being rewarded +80 until `max-steps`

4. The reward function **r4**

$$\begin{cases} \text{-if the goal is reached, `r(a) = +200`} \\ \text{-if the agent falls in the hole `r(a) = -100`} \\ \text{-if the agent lands backwards `r(a) = -50`} \\ \text{-if it is still running the game,} \\ \text{`r(a) = this.position - max}_{position_reached}\text{`} \end{cases}$$

This seems to be the good reward: it is negative if it goes backward (during a climb) and is slightly positive when moving forward (so there is a big bonnus when reaching the goal). To avoid having the agent moving too slowly and gather small rewards, we penalized it if `max-steps` was reached.

IMAGE DU JEU EN COLORIANT LES DIFFERENTES ZONES DE REWARD POUR LE NIVEAU 1 PAR EX TODO

B. Implementing the Algortihms

The agent and environment is preprocessed and is ready to be trained. As it will be discusses in 4., the algorithms implemented in the `ml-agents`package were not corresponding th those we sought to implement, so we decided to wrap our environment in a gym environment to work with the functions and methods seen in [1] (the lab 4 and 5 were more adapted to discrete states rathen than continuous). One of the brand new features of the latest release of `ml-agents` is `UnityToGymWrapper` which did exactly what we wanted.

1) REINFORCE algorithm: Once in our gym environment, we implemented the REINFORCE algorithm detailed in 2. which is well suited to our continuouds observation space. Following the steps of [1]:

1. Implemented the `sigmoid` function defined by:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

in order to scale our input between -1 and 1.

2. Implemented the *logistic regression* function that implements the logistic regression. This function returns the probability to draw action 1 (”accelerate forward”) w.r.t the parameter vector θ and the input vector s (the 5-dimension state vector).

Thanks to this, we were able to implement the *draw action* function. Then, the sigmoid policy has the following property (the proof will be in the Appendix 1):

$$\nabla_\theta \log \pi_\theta(s, \text{RIGHT}) = \pi_\theta(s, \text{LEFT}) \times s$$
$$\nabla_\theta \log \pi_\theta(s, \text{LEFT}) = -\pi_\theta(s, \text{RIGHT}) \times s$$

This will help us implement the *compute policy gradient* function.

Then, we introduced the paremeters:

EP_DURATION = 50
ALPHA_INIT = 0.1
NUM_EP = 100
STAND = 0
ACCEL = 1

Finally, thanks to all the auxiliary functions detailed above, the coding of the `train` function is quite simple.

2) Deep Q-Learning: We did not implemented the vanilla Deep Q-Learning, but we added some features to increase the efficiency of the network.

First, as in [1], we added a **Replay Buffer** to our network. This method consists to store the `BATCH-SIZE` last steps played. Once the buffer is full, all the samples are randomly shuffled and our agent learns on this shuffled samples. It brings stability to the learning process as the agent trains on more past experience and break harmful correlations. In our prject, we decided to use the $Adamoptimizer$, but if we change to $StochzsticGradientDescent(SGD)optimizer$, the data must be independant and identically distributed. Consequently, implementing a Replay Buffer is essential. The size of the Replay buffer -that's to say `BATCH-SIZE`- is not a parameter we changed for this project. We left it to `BATCH-SIZE = 72` (emprically found).

Subsequently, we implemented a second neural network which is used as a **Target Network** (TN) for our agent. Instead of updating the weights every time the agent learn, we update this TN only every `sync-freq`. As explained in [8], in Q-Learning, we *”update a guess with a guess”*. By using the weights and biases of the TN to train the main Network brings more stability to the learning process. Therefore, we can increase the initial value of the **learning rate** (LR). The learning process is fastened without becoming too unstable.

Finally, as we did our first experiences, we were stuck by the fact that sometimes the agent could reach the goal and, the episode just after this success, fall in the first hole or land backwards whereas we expected it to reach the goal again or to fail close to the latter. We implemented a function `dynamicLearningRate()` that updates the learning rate by multiplying it by a decay < 1 when the goal is reached.

C. Assessing the performances of our algorithms

To see if our algorithms are efficient, we had to know the performance of a state-of-the-art RL algorithm. To do so, we decided to use `StableBaselines3`, which is a Python library that contains well-designed, performant RL algorithms. The DQL is not yet available in this library, so we will compare our results with an Actor Critic algorithm which is **A2C: Advantage Actor Critic** [9].

Nevertheless environment we wrapped in gym was not inherited from the `gym.Env` class, but was of type `UnityToGymWrapper`. We implemented a new class named `CustomGymEnv()` that multi-inherits from both classes. All the methods of `gym.Env` are implemented thanks to the methods of `UnityToGymWrapper` using *super()*. As such an environment is an instance of `gym.Env`, it is accepted by the StableBaselines library.

Unfortunately, we were not able to run the StableBaselines algorithms. Indeed, before running the RL algortihm itself, the StableBaselines library wraps our environment in a vectorized environment in order to make it possible for the user to run on multiple environments. However, there is a problem when wrapping our environment: the agent plays actions but those are not rendered in the environment. We tried to modify the library, but without success for the moment. The file `CustomGymEnv` contains the code that implements the new class and the instructions to (theoretically) run the A2C algorithm.

IV. Results and Discussion

***NB**: all the learning curves in thus section have a point (0,0) that we did not manage to get rid of. The learning process starts just after this points*

A. *REINFORCE: "determnism" after first action*

The results obtained with REINFORCE are quite disappointing. The learning curve we obtained is the *fig 3*, when running on 25 episodes on the level 1

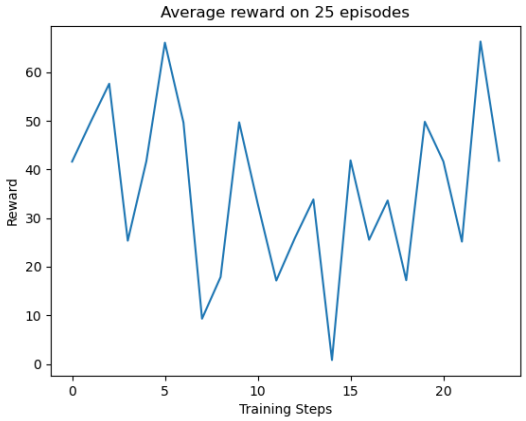


Fig. 3. Results for REINFORCED algorithm

The results are quite random. On all the tries we did, we had similar results. The problem seems to come from the choice of the action. Indeed, at the beginning of an episode, the logistic regression outputs a very small value for `prob-accel` (around 10^{-40}).

Consequently, the policy gradient does not vary and one action is always preferred over a hole episode: either the agent is almost only accelrating, either it stands at x=0. As the first level is quite simple, we managed to have some successes, but it was highly unstable. Here is a piece of the code for the `compute-policy-gradient` function:

```
prob_accel = logistic_regression(episode_states[t], theta)
a_t = episode_actions[t]
R_t = sum(episode_rewards[t::])
if a_t == STAND:
   g_theta_log_pi = - prob_accel * episode_states[t] * R_t
else:
   prob_stand = (1 - prob_accel)
   g_theta_log_pi = prob_stand * episode_states[t] * R_t

   PG += g_theta_log_pi
```

We tried to give more importance to the reward by mutliplying by `np.exp(R_t)` and to clipthe probabilities between a minimum and a maxiumum value, but we could not reach the optimal, policy this way because even if the agent gets closer to the optimal policy, it will still take random actions and fail the level. As we did not managed to solve the issue, we decided to use Deep Q-Learning whose results will be described in the next section.

B. *Deep Q-Learning (DQL)*

The results with DQL were far more satisfying than with the REINFORCE algorithm. With good parameters, the agent succeds to reach to goal 9 times on 10 on average after approximately 150 episodes. In the $DeepQLearning.ipynb$ file, you will find the commented code for DQL and the results for the three levels with the optimal parameters (learning curves, number of successes, GIFs,...).

1) Results on the three levels with the same parameters: In this subsection, we study the performance of our Deep Q Network (DQN) on the three levels (*img. 3*) with the same parameters which are:

EPISODES = 200
LR = 0.001
MEM_SIZE = 10000
BATCH_SIZE = 72
GAMMA = 0.95
EXPLORATION_MAX = 1.0
EXPLORATION_DECAY = 0.999
EXPLORATION_MIN = 0.001
sync_freq = 5

The *fig. 2* dislays the results obtained

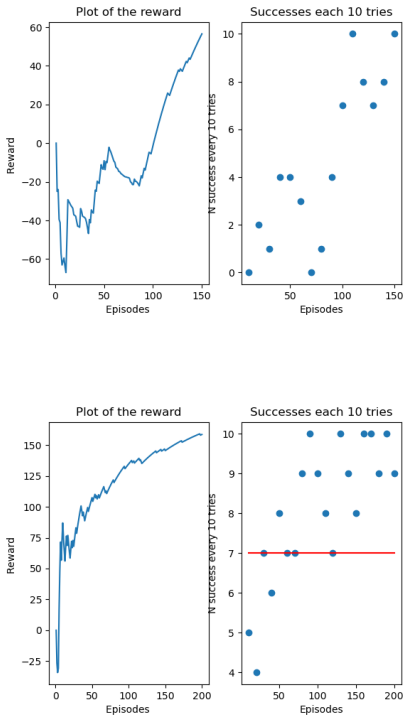
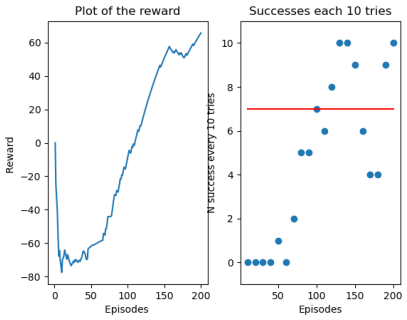


fig 2 - The Three levels with their initial parameters

The agent manages to have a high rate of succes on each episode after some training time (between 100 and 150 episodes). Yet, what is surprising is that the 'easiest' level for the agent is the Level 2. The little sinusoid during the 50 first episodes corresponds to the period between the first success of the agent when goes very carefully and very slowly to the goal and when it optimizes its speed to go as fast as possible to the goal. We had anticipated this results but we thought that the agent would take more time before this 'transition'. On the contrary, the drop in the graph for the level



1 was unexpected.The agent starts to have some successes but then falls in the hole several times before definitely reaching successes. The level we designed as the easiest was in fact the hardest for the agent! The third level takes the same time of training as the first level, but is more stable, so we consider it easier. This behavior is the reason why we implemented a dynamic learning rate.

2) Fine-tuning the GAMMA factor and the Learning Rate: The main parameter we chose to study is the discount factor GAMMA. This choice comes from the "speed vs spin" tradeoff. Indeed, GAMMA modulates the importance of the immediate reward with respect to the expected (future) reward. So by setting a high GAMMA, we expect to have an agent that is very careful and that does not spin much. The following tables indicates the mean reward obtained with different discount factors.

NB: some parameters are also optimized directly within the DQN with an Adam Optimizer

Level\Gamma	0.9	0.95	0.99
Level1	-7	85	-90

For the Level 1, we stopped gamma at 0.9 because it did already give bad results. *The optimal GAMMA that we found is thus 0.95.*

Level\Gamma	0.5	0.7	0.8	0.9	0.95	0.99
Level2	-80	135	190	175	160	90

The agent has correct results even with low GAMMA values. *The optimal GAMMA that we found is thus 0.8.*

Level\Gamma	0.7	0.85	0.93	0.95	0.97	0.99
Level3	-80	53	75	63	120	-70

For the level 3, *The optimal GAMMA that we found is thus 0.97.*

The overalll results are clear. Setting a too high or a too low value of GAMMA induces a bad performance in the training. If it is too low, it gives more importance to direct reward, thus accelerating and flipping backward. On the contrary, if it is too high, the agent slows and is more careful. Nonetheless, the optimal GAMMA is different for the three levels. It has similar values for the level 1 and 3 (the learning curves of *4.2.1* had also the same trend). GAMMA must be quite high in order to cross the obstacles (0.95 and 0.97). For the level too, a lower GAMMA (0.80) is optimal as there are no holes to jump over. It fits with our interpretation. There are not major obstacles in level 2 so the agent can take more risks. In levels 1 and 3, the

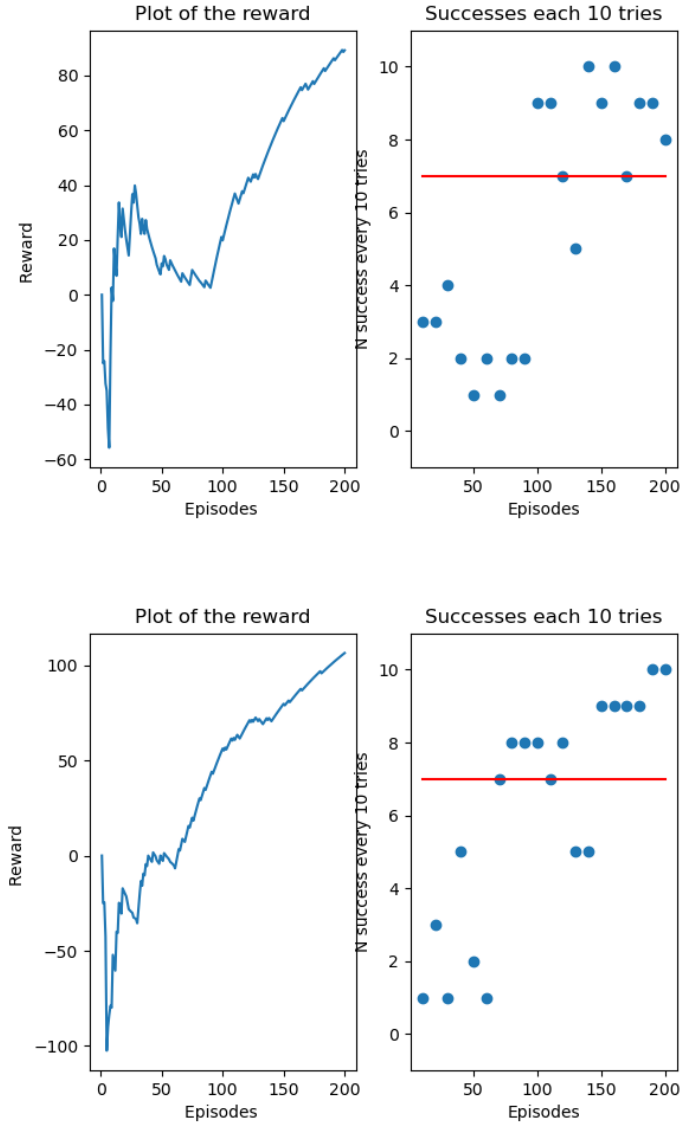


Fig. 4. Comparison for Level 1 with constant learning rate (up) and with dynamic learning rate (down)

agent has to be more prudent and to accelerate less to avoid flipping backwards (in other to maximize the future reward: the goal).

Now that we have tuned the discount factor, we assessed the effect -based on the observation of *4.1*- of setting a dynamic learning rate. The idea is to do 'big steps' during the beginning of tthe training and then increasingly shrinking the learning rate to be sure to converge direclty to the optimal policy (without having a drop in the average reward). Again, we had different results for the three levels.

The results obtained with the same parameters as in 4.2.1, but with the optimal discount factor for the level 1 are presented in the *fig. 4*.

The results are quite positive. Even if the end of the training is supposed to be slower because the learning rate is ten times slower, the overall training is faster and more stable inasmmuch as we don't have this drop in the average reward

around 4Oth episode. Dynamically changing the learning rate has a striking impact on the overall performance of our agent. However, the results are not that positive for the other levels. As shown in *fig 4*, the agent underperformed both in level 2 and 3. The main concern is not the 50 first episodes as they depend on the randomly chosen actions of the first episodes. Moreover, we don't modifiy the learning rate while there are no successes. What is disappointing is that once the first goal is reached, the reward increases less fast than with a constant learning rate. We end up with an average reward of 83 compared to 170 in the level 2, respectively with and without a dynamic learning rate. As for the level 3, it is 45 vs 120.

We think that we modified the learning rate too early for levels 2 and 3. Indeed, as the level are longer and more complex (as we had imagined in the beginning !), the agent has to take more episodes to set up a good Q-table (or to get acquainted with its environment). With more accurately tuned parameters, the level 1 is easier and faster to solve than the others. To have a better result with a dynamic learning rate, one idea could be to leave it constant for the first 100 episodes, and then to start decrease it.

V. Conclusions

This article presented a Reinforcement Learning project based on the game RIDER. We have explored all the steps, from the creation of the game to finetuning the parameters of our custom RL algorithms and comparing the performances with state-of-the-art methods. Building the game from scratch has allowed us to go back and forth during the project and to modify the environment to fit it to our study. We then implemented two algorithms, namely REINFORCE and Deep Q-learning. Although we faced problems with the REINFORCE algorithm, we managed to have conclusive results with the Deep Q network. Our agent (the motorbike) solved all three levels we had designed.
As our main motivation was to explore how the agent would behave in a situation where it has to accelerate to reach goal, but not going too fast to avoid flipping (the 'speed vs spin' tradeoff), we focused on fine tuning the discount factor GAMMA (that modulates the importance of direct reward versus future reward). To this point, we can conclude that the DQN adapted well to this tradeoff, as when it was in the air, it stopped accelerating (it 'understood' that it only induced more risks to die landing backward on the slope).

As a prolongation of this work, we can imagine two possible scenario:

1) Implementing more accurate algorithms. Having an algorithm that both implements a Replay Buffer (thar randomly shuffles previous steps to learn) and a recurrent layer (that keeps memory of the past to avoid failing too early after a success) could be more effective. To that end, we could try to code Recurrent Replay Distributed DQN[10] (R2D2).
2) Doing Reinforcement learning on a more realistic Rider Game, that's to say wihtout a goal ton reach. We only train the agent on a fixed number of episodes and see how far it can goes on a procedurally generated track. To do so, we would have to change the observation space, and give the agent a 'visual input' (an array of the pixels in front of it). Using a Convolutional Neural Network[11] could therefore be more efficient as there would be a huge amount of data.

References

[1] Read. LAB VI - Reinforcement Learning III. In *INF581 Advanced Machine Learning and Autonomous Agents*, 2023.
[2] Mountain Car Environment, https://github.com/openai/gym/blob/master/gym/envs/classic_control/mountain_car.py
[3] SethBling *MarIQ – Q-Learning Neural Network for Mario Kart* https://www.youtube.com/watch?v=Tnu4O_xEmVk&ab_channel=SethBling,
[4] Unity, cross-platform game engine, Unity Technologies, https://unity.com/
[5] *Unity ML-Agents Toolkit*, 2022, https://github.com/Unity-Technologies/ml-agents
[6] *Unity ML-Agents Gym Wrapper*, 2020, https://github.com/gzrjzcx/ML-agents/blob/master/gym-unity/README.md
[7] Brackeys, *How to make RIDER in Unity* https://www.youtube.com/watch?v=9Ztd1XXmUGI&ab_channel=Brackeys,
[8] Nicholas Renotte, *Reinforcement Learning for Gaming — Full Python Course in 9 Hours*,https://www.youtube.com/watch?app=desktop&v=dWmJ5CXSKdw&ab_channel=NicholasRenotte
[9] Asynchronous Methods for Deep Reinforcement Learning *Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu*
[10] RECURRENT EXPERIENCE REPLAY IN DISTRIBUTED REINFORCEMENT LEARNING *Steven Kapturowski, Georg Ostrovski, John Quan, Re mi Munos, Will Dabney*
[11] https://en.wikipedia.org/wiki/Convolutional_neural_network

APPENDIX

A. Appendix 1 (according to [1])

First, let's define $\pi(s, RIGHT)$ and $\pi(s, LEFT)$:

$$\pi(s, RIGHT) = \sigma(s^\top \theta)$$
$$\pi(s, LEFT) = 1 - \sigma(s^\top \theta)$$

Then, let's compute the derivative of the sigmoid function:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

1. Compute $\nabla_\theta \log \pi_\theta(s, \text{RIGHT})$:

$$\begin{aligned}
\nabla_\theta \log \pi_\theta(s, \text{RIGHT}) &= \nabla_\theta \log \sigma(s^\top \theta) && (2)\\
&= \frac{1}{\sigma(s^\top \theta)} \nabla_\theta \sigma(s^\top \theta) && (3)\\
&= \frac{1}{\sigma(s^\top \theta)} \sigma(s^\top \theta)(1 - \sigma(s^\top \theta))\ s && (4)\\
&= (1 - \sigma(s^\top \theta))\ s && (5)\\
&= \pi_\theta(s, \text{LEFT}) \times s && (6)\\
& && (7)
\end{aligned}$$

2. Compute $\nabla_\theta \log \pi_\theta(s, \text{LEFT})$:

$$\begin{aligned}
\nabla_\theta \log \pi_\theta(s, \text{LEFT}) &= \nabla_\theta \log(1 - \sigma(s^\top \theta)) && (8)\\
&= \frac{1}{1 - \sigma(s^\top \theta)} \nabla_\theta (1 - \sigma(s^\top \theta)) && (9)\\
&= \frac{-1}{1 - \sigma(s^\top \theta)} \sigma(s^\top \theta)(1 - \sigma(s^\top \theta))\ s && (10)\\
&= -\sigma(s^\top \theta)\ s && (11)\\
&= -\pi_\theta(s, \text{RIGHT}) \times s && (12)\\
& && (13)
\end{aligned}$$