COMPSCI 9170B

Final Project:
Car Reinforcement Learning Agent

Submitted By

Shrita Gaonkar

Submitted To

Scott Leith

23rd April 2022

Table of Contents

Introduction

Autonomous vehicles are without a doubt the most popular, expensive, and profitable invention of the twentieth century. Waymo, Uber, and a slew of other companies have adopted self-driving technology, including Tesla although semi-autonomous as of April 2022. Speeding, inattentive, or inebriated driving are common causes of car accidents; adopting self-driving technology will lessen these car accidents. The majority of incidents can be prevented if self-driving automobiles are used. Not only that, but practically all self-driving cars run on battery rather than gasoline, making them a more environmentally friendly option than other vehicles available in the market that still run-on gas. These are some of the benefits, however the disadvantages much outnumber the benefits.

Autonomous vehicles are much more difficult to implement, and the risks associated are much higher. Waymo is the only firm that has developed a fully self-driving car, but they are still actively testing and assuring that it is suitable for market before releasing it. Even semi-autonomous cars like Tesla are still quite popular, demonstrating that there is a high demand for these vehicles. The well-known trolley problem has inspired new ethical research fields as a result of autonomous vehicles. Many businesses are racing to discover a solution to these problems and develop a completely autonomous vehicle that is viable.

In my final project for the reinforcement learning course, I want to combine what I have learned so far into a reinforcement learning agent that could drive a car without human interaction. I will be implementing my agents in three different scenarios. I attempted in making a reinforcement learning agent that can drive a car in a parking lot, a reinforcement learning agent that can drive on highways, and a reinforcement learning agent that can drive around and, in a roundabout, using three environments: merging, roundabout, and parking. For each circumstance, I compared the performance of various policies like Proximal Policy optimization, Hindsight experience relay and the Deep Q-networks along with an agent that takes greedy actions. The results were quite astonishing because the simplest of algorithms performed better than the more complicated ones for some cases. For each environment, I discovered that the performances of these algorithms change heavily, which I feel will be another problem when deploying an RL agent in an autonomous context.

I have also uploaded the project along with the project report to my Github repository. The link to which is https://github.com/Shrita10/CarRL

Architecture

For ease of implementation, the problem statement is written in Markov Decision Process.

State(s): The state space includes not only the position of the car, but also the positions of the cars around the RL driving car object. We also include information about the road the car is driving on. As a result, the state space expands dramatically.

Action a: The action space consists of the car driving straight, stopping, turning right and turning left, and accelerating the car to move faster or slower. More importantly, we define the behaviour of the car as a weight on the car's decision making/action. The product of the chances of crashing and the action taken would then determine an updated action set.

Updated action = chances of crashing × action a

Where chances of crashing = {very low, low, high, very high, medium}

Reward r: When the reinforcement learning hits a car or a wall, it receives a large negative reward. If the car successfully parks, passes the roundabout curve, or drives through the highway, we receive a positive reward at the end of the loop of the testing environment. We also give a positive reward for speed.

A large reward is given when the car parks in the blue box, indicating that it has parked in the designated area. When the car parks in an area where it is not supposed to park, a negative reward will be given. When the car collides with other cars on the highway in the merge environment, a high negative reward is given. The agent performed far too well in comparison to the environment's default reward values. To make the testing more thorough, I decided to add some noise to the environment by adding more cars, etc. This reduced the rewards and, as a result, the accuracy of the agents' performance.

Policy: We compare agents that make random greedy decisions to agents that make decisions based on policies such as Hindsight Experience Replay, Deep Q-networks, and Proximal Policy Approximation.

Project Set-up

I have included a separate section for the environment setup because it is a critical, clean step before we begin the project. Failure to do so, as well as running the project in any normal environment, resulted in numerous time-consuming errors. All the libraries required for the project must be installed individually in the Python anaconda environment. The stable baselines libraries are only compatible with Gym 0.21.0 and Python 3.7. A distinct environment was created as a result of this distinction in order to avoid any errors.

conda create --name intelcar python=3.7

conda activate intelcar

Following that, various libraries, such as gym, highway env, tensorflow, and others, are installed in this environment.

pip install tensorflow=1.15.0

pip install "gym==0.21.0"

pip install highway-env

pip install jupyter

We create a different directory for the project and store our files in that directory

mkdir car_project

cd car_project

jupyter notebook

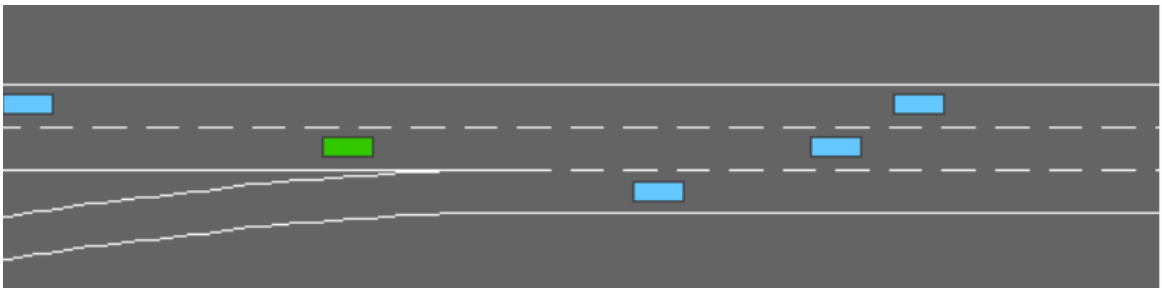
The final library required is OpenAI, which technically contains all of the algorithms such as Hindsight Experience Relay, Deep Q-networks, Soft actor critic, and proximal policy optimization algorithms.

Environment

Highway-env is the most commonly used and, most likely, the only environment for testing Reinforcement Learning car agents. In my project, I attempted to create reinforcement learning agents capable of parking a car, navigating a roundabout, and merging on highways. More information on how each of them works is provided below.

Merge

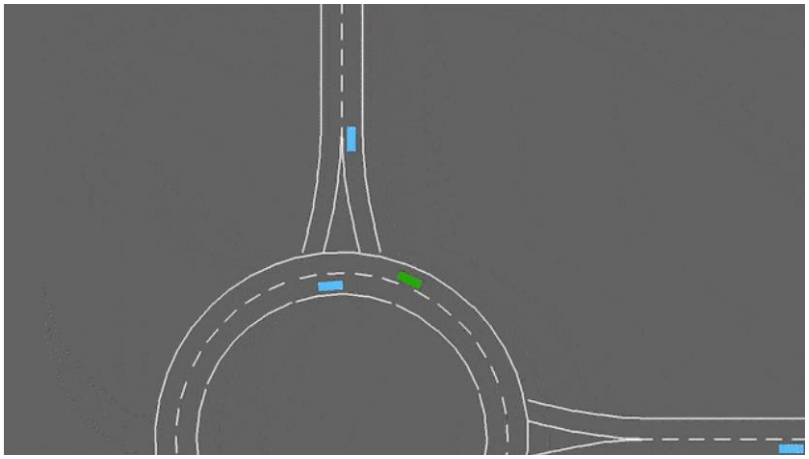
The ego-vehicle starts out on a major highway, but soon comes to a road junction with incoming vehicles on the access ramp. The goal of the agent now is to maintain a high speed while making room for the vehicles to merge safely into the traffic. The obstacles that the agent must overcome are the other cars on the highway. The agent must learn to change lanes.



Roundabout

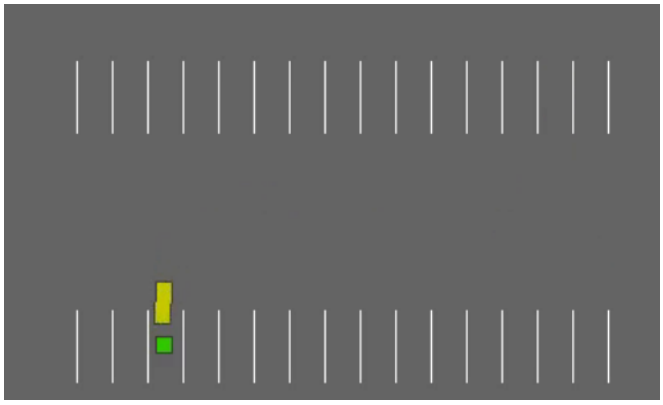
The ego-vehicle is approaching a roundabout with traffic in this task. It will automatically follow its planned route, but it must manage lane changes and longitudinal control to get through the roundabout as quickly as possible while avoiding collisions. The main challenge is to wait and see if other vehicles are approaching the road. When other cars collid, it has to come to a halt. Along with the ego-vehicle, there are nearly five vehicles present.

Another objective for the agent is to take turns as the road progresses.



Parking

A continuous control task with a goal in which the ego-vehicle must park in each space with the appropriate heading. There are plenty of parking spaces available. The environment, on the other hand, chose the parking spot for the ego-vehicle at random, and the RL agent must park the car exactly at that parking spot without hitting any other lines in the parking area. The parking environment's default configuration includes only one controlled vehicle. I added an extra controlled environment to increase the difficulty and make the environment more realistic.



Policies/Methods

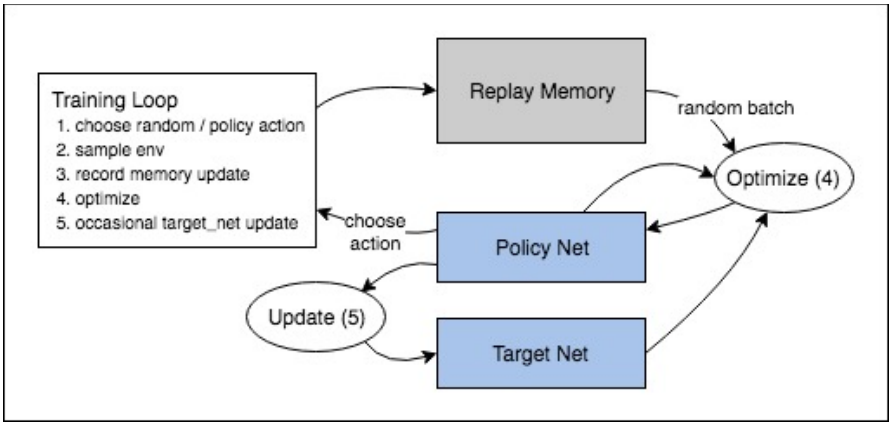
Deep Q-learning

Deep Q Learning (DQN) is a reinforcement learning algorithm first proposed in paper Playing Atari with Deep Reinforcement Learning (Mnih et al. 2013). The algorithm uses convolutional networks to learn using raw pixels, which the authors then utilize with Q Learning to train a model capable of playing ATARI games.

Since the technique relies on raw pixels of a scene, the complexity is decreased, by scaling the RGB pixels to four frames of (84 x 84) images, effectively providing us a (84 x 84 x 4) tensor. This tensor is then fed to a convolutional neural network returning a vector containing the Q value of every action. Afterwards, an exploration scheme (generally epsilon-greedy) is used, and an action is probabilistically chosen between highest Q value and a random action.

DQN consists of the following steps repeated until a desired outcome is received:

- Use current policy is to gather and store samples in a replay buffer
- Sample random batches of experiences from the replay buffer (Experience Replay)
- Update Q network using sampled experiences



https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Above, the authors effectively define experience replay as a sampling random experiences, and provide the explanation behind its use in contrast to sequential experiences by discussing the fact that sequential experiences hold high correlation with each other in a learning environment such as an ATARI game. Random sampling of experiences allows mitigation of this temporal correlation of averaging it over many of its discrete earlier states.

Additionally, authors describe the approach towards updating the Q network, as minimizing the mean squared error between current Q output and a target Q value (according to the Bellman equation). Mathematically,

$$L(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot)}[(y_i - Q(s,a;\theta_i))^2]$$

where $$y_i = \begin{cases} R_T & \text{for terminal state } s_T \\ R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') & \text{for non-terminal state } s_t \end{cases}$$

Finally, the DQN loss function can be defined as follows:

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot),s'\sim\mathcal{E}}[(R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)]$$

Using the DQN technique, the authors were able to get state-of-the-art results (at the time) and were able to open a different branch of reinforcement learning algorithms which performed very well on diverse set of applications.

Hindsight Experience Replay

Hindsight Experience Replay (HER) is a reinforcement learning technique first proposed in paper Hindsight Experience Replay (Andrychowicz et al. 2018). The algorithm works in addition to off-policy learning to learn ways to win in games that require long-term strategy and do not have immediate rewards available for an agent to learn from.

The description of the algorithm can be seen below:

Algorithm 1 Hindsight Experience Replay (HER)

Given:
- an off-policy RL algorithm $\mathbb{A}$, ▷ e.g. DQN, DDPG, NAF, SDQN
- a strategy $\mathbb{S}$ for sampling goals for replay, ▷ e.g. $\mathbb{S}(s_0, \ldots, s_T) = m(s_T)$
- a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$. ▷ e.g. $r(s,a,g) = -[f_g(s) = 0]$

Initialize $\mathbb{A}$ ▷ e.g. initialize neural networks
Initialize replay buffer R
for episode = 1, M **do**
  Sample a goal g and an initial state s_0.
  for $t = 0, T-1$ **do**
    Sample an action a_t using the behavioral policy from $\mathbb{A}$:
      $a_t \leftarrow \pi_b(s_t||g)$ ▷ $||$ denotes concatenation
    Execute the action a_t and observe a new state s_{t+1}
  end for
  for $t = 0, T-1$ **do**
    $r_t := r(s_t, a_t, g)$
    Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in R ▷ standard experience replay
    Sample a set of additional goals for replay $G := \mathbb{S}$(**current episode**)
    for $g' \in G$ **do**
      $r' := r(s_t, a_t, g')$
      Store the transition $(s_t||g', a_t, r', s_{t+1}||g')$ in R ▷ HER
    end for
  end for
  for $t = 1, N$ **do**
    Sample a minibatch B from the replay buffer R
    Perform one step of optimization using $\mathbb{A}$ and minibatch B
  end for
end for

https://arxiv.org/pdf/1707.01495.pdf

The core idea behind HER is to allow the agent to perform well despite of sparse rewards. In order to accomplish this, the authors suggest tweaking the general Q learning algorithm to follow the following details:

- Gather and store all states, goals, actions, rewards and next state from the policy as a tuple into an experience buffer.
- Sample goals from the states visited in the episode obtained at the earlier step, and for each goal, store data tuple into the buffer, repeating both steps through multiple episodes.
- Sample experience batches from the buffer and train the network, repeating training through multiple iterations to retrieve optimized results.

Using the HER method, authors were able to get excellent results on robotic object manipulation tasks, allowing the DQN algorithm to be extended further towards more challenging tasks with sparse rewards.

Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a reinforcement learning technique consisting of several policy optimization methods first proposed in paper Proximal Policy Optimization Algorithms (Schulman et al. 2017). The technique is an on-policy policy gradient method, which alternate between sampling data through interaction with the environment and optimizing a "surrogate" objective function using gradient ascent.

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

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

The objective function used by PPO is depicted as:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)\,]$$

where

θ is the policy parameter

$\hat{E}_t$ denotes the empirical expectation over timesteps

r_t is the ratio of the probability under the new and old policies, respectively

$\hat{A}_t$is the estimated advantage at time t

ε is a hyperparameter, usually 0.1 or 0.2

https://openai.com/blog/openai-baselines-ppo/

The PPO technique allows a significant improvement on policy gradient methods which are sensitive to the choice of step size and have poor sample efficiency, causing the model learning to take millions of steps to learn basic tasks. It accomplishes this through Trust Region update compatible with Gradient Descent, and further simplifying the algorithm by removing the KL penalty negating a need to make adaptive updates and providing fast, efficient way to learn easy tasks.

PPO, like other on-policy methods, strikes a balance between exploration and exploitation. It samples behaviour from the same policy that it optimises because it is on-policy. Initially, the exploration rate is higher than the exploitation rate in order to explore the state space. Eventually, the policy focuses on maximising the benefits that have already been discovered.

Results

So far in the report, we've discussed the environment, the environment's setup, the algorithms used in the project, and the problem formulation of the three tasks that my Reinforcement Learning agent intends to solve.

Let us now discuss how the algorithms performed on the various algorithms. I compared the algorithm's performance to that of the agent performing random actions. Let us begin by taking a look at the roundabout environment.

I ran each of the environments through 100 loops. Following that, I calculated the rewards at the end of each loop, yielding a total of 100 rewards at the end of 100 loops. Then I take the average of the 100 rewards. I ran this code five times to thoroughly test the performance of the algorithms. Then I compare the results to those of other algorithms. The following is a list of the average rewards I receive after running 100 loops.

Roundabout

First Run:

Algorithm	Average of rewards after 100 runs	Number of crashes
Greedy	0.95	37
PPO	0.89	26
DQN	0.91	57

Second Run:

Algorithm	Average of rewards after 100 runs	Number of crashes
Greedy	0.91	25
PPO	0.97	0
DQN	0.87	3

Third Run:

Algorithm	Average of rewards after 100 runs	Number of crashes
Greedy	0.94	36
PPO	0.93	23
DQN	0.93	31

Fourth Run:

Algorithm	Average of rewards after 100 runs	Number of crashes
Greedy	0.55	34
PPO	0.95	30
DQN	0.93	20

Fifth Run:

Algorithm	Average of rewards after 100 runs	Number of crashes
Greedy	0.55	27
PPO	0.89	1
DQN	0.88	0

Interpretation

In most cases, the Proximal Policy Optimization algorithm performed well in terms of fewer crashes and an average of rewards. There are times when the greedy algorithm has more rewards than PPO and DQN, but when we look at the number of crashes, the greedy algorithm has more than PPO and DQN. There is no way that we would pick a car which has more crashes.

Initially, when I did not change the environment's default settings, the greedy algorithm outperformed DQN and PPO. The greedy algorithm's performance decreased after I added more cars and increased the difficulty.

Another surprising change was the timesteps. I noticed that when the timesteps were set to 1000, the car did not wait at the intersection to see if another car was approaching. When the timesteps were increased to 10000, the car waited at the intersection to see if another vehicle was approaching, and it also waited when the other cars collided. This demonstrates that as the algorithm's timesteps increased, the car became more intelligent.

Parking

First Run:

Algorithm	Average of rewards after 100 runs
Greedy	-0.43
HER	-0.46

Second Run:

Algorithm	Average of rewards after 100 runs
Greedy	-0.51
HER	-0.45

Third Run:

Algorithm	Average of rewards after 100 runs
Greedy	-0.51
HER	-0.46

Fourth Run:

Algorithm	Average of rewards after 100 runs
Greedy	-0.62
HER	-0.47

Fifth Run:

Algorithm	Average of rewards after 100 runs
Greedy	-0.43
HER	-0.46

Interpretation

The agent's performance in the parking environment was dismal. According to the video in the jupyter notebook attached to the report, the agent was trained on the Hindsight Experience Replay policy, and the performance improved as a result of the agent taking greedy actions.

After I added two controlled environments to the parking environment, the rewards for the greedy agent were drastically reduced. The rewards for HER were lower with fewer timesteps but improved with more timesteps.

The HER algorithm outperformed the greedy algorithm.

Merge

First Run:

Algorithm	Average of rewards after 100 runs	Number of crashes
Greedy	0.93	25
DQN	0.97	0
PPO	0.87	3

Second Run:

Algorithm	Average of rewards after 100 runs	Number of crashes
Greedy	0.89	23
DQN	0.97	0
PPO	0.94	4

Third Run:

Algorithm	Average of rewards after 100 runs	Number of crashes
Greedy	0.84	22
DQN	0.97	0
PPO	0.90	0

Fourth Run:

Algorithm	Average of rewards after 100 runs	Number of crashes
Greedy	0.97	17
DQN	0.97	0
PPO	0.95	6

Fifth Run:

Algorithm	Average of rewards after 100 runs	Number of crashes
Greedy	0.95	25
DQN	0.84	0
PPO	0.91	3

Interpretation

The merge environment was less difficult to implement than the parking and roundabout environments. The values from the environment configurations worked perfectly fine, but I still went ahead and edited some of the configurations, and the performance was excellent in difficult situations as a result.

In many cases, the number of crashes was zero. The DQN algorithm outperformed the PPO and greedy algorithms.

Conclusion

In conclusion, I would like to talk about the environments I tried for the project, the results, and the challenges I encountered along the way.

There were a few obstacles in the project that I had to overcome. It was critical to set up the conda environment for the project. Initially, I ran it in the “base” conda environment, which resulted in a slew of errors. After some investigation, I discovered that the solution was to run the project in a different conda environment.

Another challenge was to improve the parking environment agent. After failing with a couple of algorithms, including DQN and PPO, I turned to HER. Moreover, algorithms like HER and PPO have a very high training time. It took me almost three hours to run the 10000 timesteps for all the algorithms.

The project began by changing the environment's default configurations; in roundabout environments, I increased the default collision reward to -8 to ensure that the vehicle receives a high negative reward when it collides. In addition, I increased the high-speed reward to 1. Because of the positive high-speed reward, the greedy agent may receive greater rewards. We would always prefer a car with low speed and low collision rates than a car with high speed and high collision rates. That is why, it can be noticed that in the interpretations of the results, I pick the algorithm with good rewards and a smaller number of crashes combined.

In the roundabout environment, the greedy algorithm outperformed more intelligent policies such as DQN and PPO in some situations; however, the number of times the car crashed into other cars and walls was much higher than DQN and PPO, and the rewards may be high due to other factors such as driving at a high speed. If we consider the average of the rewards from the 100 loops and five runs, as well as the number of crashes, the PPO algorithm outperformed the greedy and DQN algorithms in the roundabout performance.

The highway environment throws random roundabouts at the car; if we look at the range of the average of the rewards, greedy algorithm has a very wide range, which means it performs very well in some cases and very poorly in others, whereas algorithms like DQN and PPO have a smaller range, which means they may not always perform the best but will not perform poorly. They are specifically designed to avoid collisions at all costs.

The parking environment was difficult to implement. Without changing the configurations, the rewards were nearly equal to 0.07. I also tried DQN and PPO algorithms on the parking environment, but they performed horribly. In almost all cases, the greedy algorithm and the HER algorithm perform equally well. However, after adding the second controlled vehicle, the greedy agent's performance suffered greatly. The timesteps of the HER algorithm were also very important in the parking environment. Timesteps less than 1000 resulted in worse rewards than the greedy agent.

Initially, in the project, I implemented the highway-environment in which the ego-vehicle drives on a multilane highway crowded with other vehicles. The agent's goal is to travel at a high speed while avoiding collisions with other vehicles. It is also advantageous to drive on the right side of the road. Later, I discovered that the merge environment is the same as the highway environment, and we must also solve the merging problem on highways.

This was simple to implement because the car has no walls, and the cars are running at a greater distance from the ego-vehicle than the roundabout and parking environments. DQN and PPO policies were considered for this problem. When the number of crashes and the average of the rewards were considered, the DQN algorithm outperformed the PPO and the greedy algorithms.

REFERENCES

[1] https://colab.research.google.com/drive/1rP6HiZOCIDeRxtc8Bajvnqv1fDYFm_K8?usp=sharing#scrollTo=Y5TOvonYqP-g
[2] https://stable-baselines.readthedocs.io/en/master/modules/ppo2.html
[3] https://www.youtube.com/watch?v=HrapVFNBN64
[4] https://www.youtube.com/watch?v=Dvd1jQe3pq0
[5] https://medium.com/analytics-vidhya/advanced-exploration-hindsight-experience-replay-fd604be0fc4a
[6] https://towardsdatascience.com/hindsight-experience-replay-her-implementation-92eebab6f653
[7] https://towardsdatascience.com/reinforcement-learning-with-hindsight-experience-replay-1fee5704f2f8
[8] https://www.youtube.com/watch?v=tmjLvRc5ZsQ
[9] https://pradeepgopal1997.medium.com/mini-project-2-2cafa300895c
[10] https://github.com/eleurent/highway-env/blob/master/docs/source/environments/parking.rst
[11] https://eleurent.github.io/highway-env/environments/roundabout.html
[12] https://www.emerald.com/insight/content/doi/10.1108/JICV-01-2018-0003/full/pdf?title=markov-probabilistic-decision-making-of-self-driving-cars-in-highway-with-random-traffic-flow-a-simulation-study
[13] https://www.emerald.com/insight/content/doi/10.1108/JICV-01-2018-0003/full/html
[14] https://towardsdatascience.com/deep-q-networks-theory-and-implementation-37543f60dd67
[15] https://paperswithcode.com/paper/hindsight-experience-replay
[16] https://highway-env.readthedocs.io/en/latest/quickstart.html#examples-on-google-colab