COMP450/550 Introduction to Reinforcement Learning  
Rainbow SpaceY Project  
Final Report

Prepared by Space Pharaohs Team: Ahmed Gamal Masry & Mohammed Kazamel

# Abstract

Rainbow SpaceY is a game in which a rocket learns how to take off from one planet (the origin planet) and lands successfully on a target planet. The rocket can only learn these by trial and error which means this problem can’t be done in a supervised way and it must be solved using Reinforcement Learning. For the RL formulation, we have utilized some physical measurements as our state, engineered our own reward function based on distances and orientation, and rockets moves in four directions as our actions. In order to solve this problem, we have implemented the famous DQN, DDQN, Dueling DQN[1], and used the Prioritized DQN from an open source project. At the end, we integrated them into one agent called Semi-Integrated agent. Although all the algorithms succeeded in achieving the task, the Semi-Integrated agent achieved the highest success rate. This indicates that combining different Extensions of DQN can result in better performance, although it’s not guaranteed.

# Introduction

We started this project with the intention to solve the SpaceY (Bykhovskiy, 2018) problem. However, we didn’t attempt solving the problem directly, instead, we made simpler versions of it and we added complexity incrementally as we solved the simpler cases. This strategy made the problems as well as the debugging process significantly easier and manageable with limited computational resources. The time frame of the semester didn’t allow us to add complexity until reaching the original problem (game), so we will report our findings on a simpler version of the game.

The problem is simply building an RL agent to guide a rocket from the origin planet to the target planet. The locations of the two planets are randomly generated in a bounded 2d space. In each step, the rocket can take one of six available actions: take a step in any of the four directions or rotate left or right. The rocket landing is considered successful when the rocket lands on its back, with a clearance of 30 degrees in each direction. The agent fails when the rocket reaches the bounds of the space or when it hits the target planet with a wrong angle. This problem contains all the main elements of a reinforcement learning problem and being able to solve it helps us to solve more practical RL problems.

We decided to apply the methods mentioned in the Rainbow paper (Hessel et al. 2018) to solve this problem. The methods are improvements on the DQN algorithm (Mnih et al. 2015), so our solution method is q-learning with function approximation, with the deep networks as the approximators. The paper compared the performance of the base DQN algorithm and six different improvements. The authors finally combined all the improvements into one algorithm and called it Rainbow algorithm. Rainbow’s performance surpassed all the others by a huge margin.

In our case, we were able to implement three algorithms as well as the base DQN algorithm within the time frame of the semester. We finally combined these algorithms into one agent, which we will refer to in this report as the semi-integrated agent. The algorithms we implemented are namely Double Q-learning (Van Hasselt et al. 2016), DQN with Prioritized Experience Replay (Schaul et al. 2015), and Dueling Networks (Wang et al. 2016).

During the training process, our reward function and state representation had to be changed many times. We have experimented different hyperparameters values till the rocket learnt to take off and land successfully. Although our results are not that much satisfying for some algorithms, we believe if we had more time, we could have come up with a better reward function and state representation that improves our results.

In this report, we will first present the formal RL definition of the problem, then explain the five algorithms we used to solve it. After that, we will provide some detail about the implementation of the game environment and

the algorithms. Afterward, we explain how we evaluated the algorithms, the measures we used, and we discuss the results we obtained. Finally, we end with the conclusion of our report.

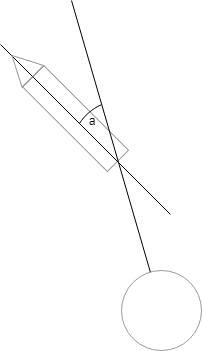


Figure 1: Angle between rocket and planet

# Formal Problem Definition

## States

- Normalized relative x position of the rocket

- Normalized relative y position of the rocket

- Orientation of the rocket with respect to the target planet

where a is the angle shown in Figure 1

## Actions

The set of actions is as follows: {move forward, move right, move backward, move left, rotate right, rotate left}

## Rewards

We used dense rewards in our training, and they are as follows:

- Nonterminal states reward:

where α= 0.1, β= 0.5

- Terminal reward for successful landing: 5

- Terminal reward for failed landing (hitting the target planet) or going out of the screen borders: -1

## Discount Factor

We used a discount factor of 0.99.

# Approach

Since our state space is continuous, we cannot use Tabular Methods for solving this problem. Instead, we used DQN to approximate our q-values. So, our formulation of the problem was model-free. We have implemented four of the algorithms mentioned in Rainbow paper (Hessel et al. 2018). Namely, DQN, Double DQN, Prioritized replay, Dueling DQN, and their combination. The network architecture is explained along with the algorithms.

## 1- DQN

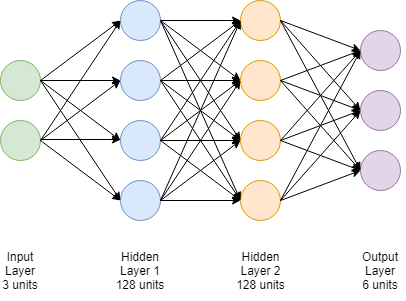


Figure 2: DQN Architecture

The input of the DQN is the state representation mentioned above. The output is q-values for each action of the six actions that may be taken in this state. So, at each step, the action that gives the maximum value in our neural network is selected e-greedily. Epsilon decays exponentially from 1 to 0.01. We used an exponential decay function for epsilon:

where 𝜖𝑓: final epsilon = 0.01, 𝜖𝑖: initial epsilon = 1, n: frame number, and d: epsilon decay

A transition (St, At, Rt+1, yt+1, St+1) is added to replay buffer (Lin 1992), that holds the last forty thousand transitions. We limit the number of frames in each episode to 500. The neural network is an MLP that consists of two hidden layers with ReLU activation function having 128 neurons each, is being trained using Adam optimizer on batches of size 256 with a 0.001 learning rate to minimize the following loss function:

where t is randomly selected from the replay buffer.

The gradient of the loss is back-propagated only into the parameters () of the online network (which is also used for action selection). The term represents the parameters of the target network; a periodic copy of the online network that’s not directly optimized. Finally, since there have been some limitations in the DQN performance, extensions of it were proposed to solve some of the limitations and increase the performance.

## 2- Double DQN

In this approach, the hyper-parameters are the same as in the DQN except that two networks for each Q are used: one for the selection of the action, and one for its evaluation as shown in the following equation:

This managed to reduce the overestimation bias in the DQN due to the maximization step over the q-values (Hasselt et al. 2016).

## 3- Prioritized Replay

In this approach, the hyper-parameters are the same as in the DQN except that instead of randomly choosing transitions from the replay buffer in the DQN, here we choose the transitions according to priorities assigned to each of them (Schaul et al. 2015). The priorities are relative to the last encountered absolute TD error as the following:

## 4- Dueling Networks

The main differences between this algorithm and DQN are as follows, this network features two streams of computation, the value and advantage streams, sharing a convolutional encoder, and merged by a special aggregator (Wang et al. 2016). So, the state-action values will be calculated as the following equation:

where *ξ*, *η*, and *ψ* are, respectively, the parameters of the shared encoder *fξ*, of the value stream *vη*, and of the advantage stream *aψ*; and *θ* = *{ξ, η, ψ}* is their concatenation.

### Dueling Network Architecture:

Our encoder consists of a single layer perceptron with 128 neurons and ReLU activation function. The input of the encoder is the shape of our state, and the output is fed into the value and advantage streams.

For the advantage stream, we are using a neural network that consists of one hidden layer with 128 neurons. The output shape of the network is the same as the number of actions, each representing the value (or advantage) of each action. For the value stream, we are using a neural network with one hidden layer that consists of 128 neurons as well. The output is just one value which represents the value of the current state. Then our aggregator computes the q-values as shown in the equation above.

## 5- Semi-integrated agent

We called the agent that combines the above algorithms the semi-integrated agent to differentiate it from the Rainbow algorithm which is called the integrated agent in the Rainbow paper (Hessel et al. 2018). The semi-integrated agent simply combines the above approaches together. Since the approaches are in different parts of the pipeline, there weren’t any incompatibilities or issues in combining them.

# Implementation Details

## The Environment

The original SpaceY environment was implemented using the Unity Framework (Bykhovskiy, 2018). We wanted to have some control over the environment, and since the OpenAI Gym Framework is simpler and more flexible, we reimplemented the game from scratch using the Gym Python interface. First, we implemented the rocket as a point going from the origin planet to the target, and when we could solve this case with our RL agent, we added the orientation of the rocket to the picture.

## The Algorithms

We implemented our algorithms using the PyTorch library version 1.1 and Python 3.7. We utilized the GPUs of our laptops for training the models.

# Evaluation Details, Results, and Analysis

## Environment

Our environment dimensions are 600x600 pixels. At the training, we have tried two approaches. The first one is to fix he target planet’s location at the center of the screen and to change only the origin planet location after every 40 episodes of training, since we are using the relative location of the rocket to the target planet in our state. We chose 8 locations for the origin planet which are separated by 200 pixels from each other formulating a square around the target planet. The reason is that they are enough to allow the rocket to explore most of our state space in the shortest time, since they are evenly distributed in our environment.

The second one is to generate around 50 random origin and target planets’ locations and use them for training the rocket. However, since our state representation doesn’t encode the location of the borders, the rocket sometimes crosses the borders and fails; the same relative location to the target planet can be in the environment with some target planet locations and outside the environment for other locations. That confuses the agent and degrades the performance.

Our neural network will learn to minimize the distance between the target and the rocket. Moreover, it will learn to minimize the relative angle between the rocket and target planet as stated above.

At each episode, the maximum number of frames the rocket can take before finishing and reaching a terminal state is 500 frames. After that, we reset the environment and start a new episode. To enhance the exploration at each new origin location we set the epsilon to 1 and start decaying it as shown above. This approach guarantees that the rocket will have the chance to explore most of the state space. Furthermore, we update our target model with the parameters of the current model every 1000 frames.

## Measures of Success and Metrics

After training our models, we tested each of them on a set of 50 randomly generated origin and target planet locations. The percentage of times the rocket succeeded in reaching the target planet successfully is used as our first measure of success. Our second metric is the number of time-steps the rocket takes to reach the target planet using each algorithm on the same randomly generated locations.

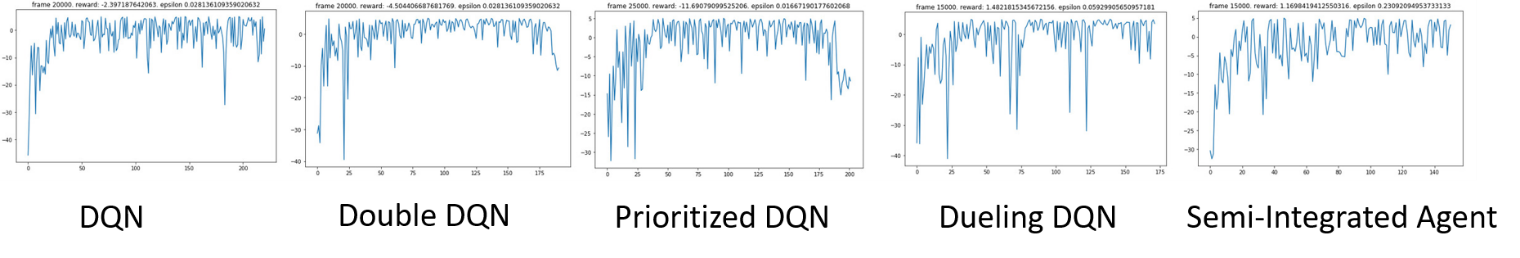


Figure 4: Total reward at each episode.

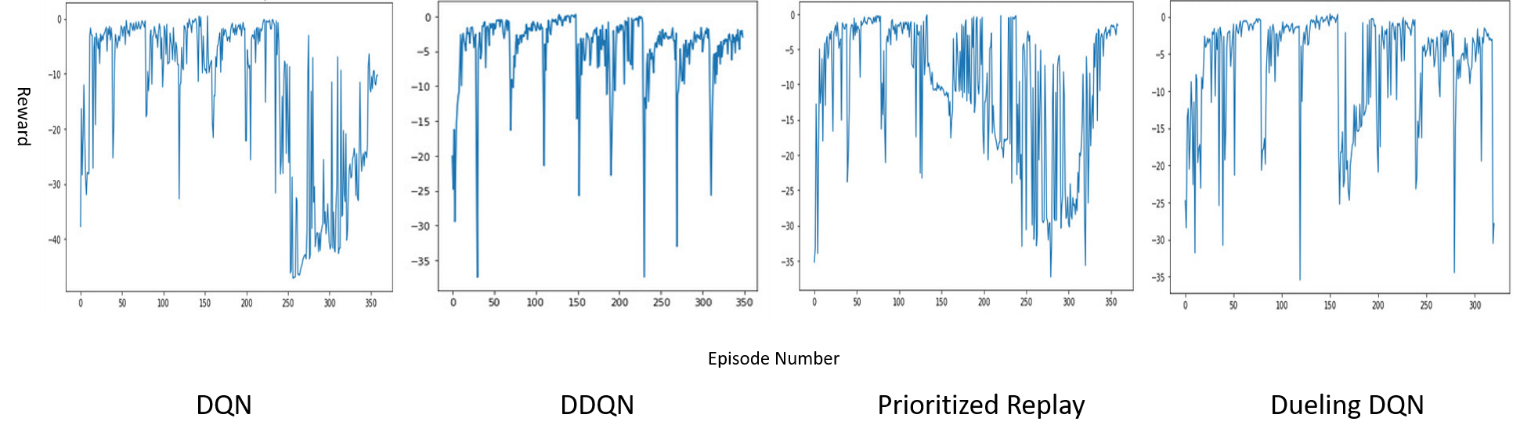


Figure 3: Total reward at each episode.

## Results

Here we present a plot of the rewards that the rocket gets during the training process as well as the success metrics on the 50 random planet locations. We use a learning rate of 0.001 for all our algorithms, and the architecture as explained before. As we stated above, we had two training processes with different approaches.

For the first one, after training our models on the 8 locations we obtained the rewards vs episodes plots shown in Figure 3. The rewards in the figure range from around zero to around -40 because we use negative rewards except for a successful landing. There is some cyclical behavior in the figures because we increased the epsilon with each new location. After the training, we tested the model with 50 randomly generated locations and got the success rates for each algorithm and average number of frames as shown in Table 1. For the second approach, after training our models on 50 randomly generated locations, we obtained the rewards vs episodes plots shown in Figure 4. When the reward is larger than -5 that usually means that the rocket succeeded in that episode succeeded, failed otherwise. After the training, we tested the model with 50 randomly generated locations and got the success rates for each algorithm and average number of frames as shown in Table 2. The rocket fails sometimes at the end of the training process, because of the problem we stated above; the state representation doesn’t encode all the necessary information, e.g. borders locations, so the rocket gets confused and cross the borders many times. In order to include the borders locations in the state representation, the training process will take much time since our computational power is limited.

Discussion

For the first approach, when we observed the rocket’s movement in the environment, we saw that the rocket usually gets close to the target planet quickly, but then it spends most of its time trying to minimize its relative angle to 0 (i.e. it keeps rotating left and right) without landing on the planet, although we allow a degree of freedom up to 45 degrees in both directions. We concluded that it's because our reward function ignores the degree of freedom during its calculations thus it tries to make the rocket lands perfectly perpendicular on the target planet.

We recorded the behavior of the rocket while testing on the random locations. The best performance was obtained from the DDQN which can be watched from this [link](https://youtu.be/e-xRGZ-yTUc). A worse performance was observed from the Dueling DQN and it shows the problem mentioned above. The video can be viewed from this [link](https://youtu.be/YUgcc-tyzHA). The videos have been slowed down, normally it is a lot faster.

During the second approach, the rocket learned fast in most of the algorithms. However, we observed that it fails many times after it learns. We concluded that it’s because of the lack of information about the borders in the state representation. The performance also degrades much if the training process continued for a long time; the neural network will get confused by the bad frames that results from the borders issue.

After comparing the results from both approaches, we have concluded that guiding the learning process by choosing good locations for the origin planets can make the training much better and achieves better results in some algorithms, however, in some of them it didn’t. Hence experimenting with the different algorithms is the best way to decide which one is the best.

# Conclusion

The advancements in the deep reinforcement learning algorithms have led to achieving great results in simple video games like Atari and our SpaceY. However, it requires a lot of reward function engineering. Small changes to the reward function can hugely affect the learning outcomes of the agent. The state representation also matters a lot; we must select a state representation that both encodes all the important information in the environment and facilitates the learning for the neural network. Furthermore, the exploration is also crucial for the success of the learning process. The agent must explore much of the state space so that it can store transitions that are good enough to learn the task. The problem of exploration can be solved by playing the game before ahead and storing the transitions in a file. Then, before the training process, we load the transitions from the file and store them in the memory replay. In this way, the human relieves the burden of exploration on the agent and the training process runs much faster. Finally, deciding which RL algorithm will perform better than the others is pretty much hard

|  |  |  |
| --- | --- | --- |
| Algorithm | Success Percentage | Mean Frames per Episode |
| DQN | 6% | 410 |
| DDQN | 97% | 78 |
| Prioritized Replay | 12% | 394 |
| Dueling DQN | 81% | 239 |

Table : Results of each model after the training scheme.

|  |  |  |
| --- | --- | --- |
| Algorithm | Success Percentage | Mean Frames per Episode |
| DQN | 63% | 183 |
| DDQN | 51% | 61 |
| Prioritized Replay | 79% | 111 |
| Dueling DQN | 77% | 146 |
| Semi-integrated Agent | 84% | 126 |

Table : Results of each model after the training scheme.

before experimenting. We have concluded that we should experiment all the different Deep RL algorithm and compare their results so that we can come up with the best one. Although the problem type can give us some insight about which algorithms can work better, we can’t guarantee that till we experiment. The Semi-Integrated agent achieved a higher success rate than all the other algorithms which means that integrating different extensions of DQN may result in a better performance, although it’s not guaranteed

# References

Bykhovskiy, M. (2018). *SpaceY*. Retrieved from Unity connect: <https://connect.unity.com/p/5a6dc943880c6400200e6af0>

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... & Silver, D. (2018, April). Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, *8*(3-4), 293-321.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*.