Reinforcement Learning - Project

Vladimir Balagula & Lea Setruk

May 24, 2021

1 Lunar Lander - Continuous

1.1 The problem

Lunar Lander is another interesting problem in OpenAIGym. The goal of the game is to land the space-ship between the two flags smoothly. The ship has 3 throttles in it. One throttle points downward and the other 2 points in the left and right direction. With the help of these, we control the ship.

There are 2 different Lunar Lander Environment in OpenAIGym. One has discrete action space and the other has continuous action space. We will solve the continuous one.

1.2 Related works

Reinforcement learning is a field in expension and we could see a lot of works related to the LunarLander problem. Each one using different techniques, parameters and approaches. We saw works that were solving the discrete problem and others that were solving the continuous one. A very few works took into account uncertainty of the environment. For example, in Adam Gjersvik's work [6], he added noise directly to the parameters of the learned policy and he is showing that it accelerates training and improve performance in comparison to action noise. As another example, to solve LunarLander, [7] uses a control-model-based approach to learn the optimal control parameters instead of the dynamics of the system.

1.3 The space

We will describe the environment, the states, and possible actions.

Observation Space: [Position X, Position Y, Velocity X, Velocity Y, Angle, Angular Velocity, Is left leg touching the ground: 0 OR 1, Is right leg touching the ground: 0 OR 1].

Continuous Action Space: Two floats [main engine, left-right engines]. Main engine: -1 to 0 off, 0 to +1 throttle from 50% to 100% power. Engine can't work with less than 50% power. Left-right: -1.0 to -0.5 fire left engine, +0.5 to +1.0 fire right engine, -0.5 to 0.5 off .

Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector.

We noticed that each game has a different land. It not deterministic so it can be harder for the model to learn. Also, the initial state is not always the same.

1.4 Discretization

To output discrete action space we have to quantize the actions into a finite number of states. We tried different discretizations, different numbers of actions, different functions of discretization and also hard-

coded discretization and we chose the one that gave us the best results. The hard-coded discretization (we chose the actions and wrote it on a list so the model will choose) gave us good results but we chose to build a function to get more precise results.

We built a function that divides the intervals (the possible values for each engine) to the number of points we want (to bins). For example, for the main engine, it is on when it is between 0 and 1. We want it to be represented by two numbers (weak power and full power) and we add 0 to represent the off mode. We chose to take the middle of the interval. We did the same for left and right engine. The model will choose between those actions. These are the different possible actions we have: [[0, 0], [0, 0.625], [0, 0.875], [0, -0.875], [0, -0.625], [0.25, 0], [0.25, 0.625], [0.25, 0.875], [0.25, -0.875], [0.25, -0.625], [0.75, 0], [0.75, 0.625], [0.75, 0.875], [0.75, -0.875], [0.75, -0.625]].

1.5 The rewards

Reward for moving from the top of the screen to landing pad and zero speed is about 100 to 140 points. If lander moves away from landing pad, it loses reward back. An episode is done when the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved problem is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Action is two real values vector from -1 to +1. First controls main engine, -1..0 off, 0..+1 throttle from 50% to 100% power. Engine can't work with less than 50% power. Second value -1.0 to -0.5 fire left engine, +0.5 to +1.0 fire right engine, -0.5 to 0.5 off. In this project, the goal is to get an average reward of 200 for at least 100 episodes.

2 The Strategies

2.1 Epsilon-Greedy

We'll use the exploration-exploitation trade-off strategy which is the Epsilon Greedy Exploration Strategy.

At every time step we choose an action. If the probability is less than epsilon, we choose a random action. Otherwise, we take the best known action at the agent's current state. As the agent takes more and more steps, the value of epsilon decays and the agent will choose better actions. Near the end of the training process, the agent will be exploring much less and exploiting much more.

2.2 Soft-Epsilon

Epsilon-Greedy has a disadvantage. Indeed, it chooses the random actions uniformly. But maybe some actions are better than others and we would want to choose them over others. Thus, Soft-Epsilon is a solution to it. Each action gets its probability to be selected according to its quality. An action has a probability of being chosen $\mathbb{P}(a) \geq \varepsilon$. We observed that this strategy doesn't really improve the results and it takes a lot of times to execute, that's why we decided to not use it for our models.

2.3 Experience replay

To ease the problems of correlated data and non stationary distributions, we use an experience replay mechanism which randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors.

We store the agent's experience at each time-step (the state, action, reward, next_state).

The reinforcement learning algorithms (like DQN) use Experience Replay to learn in small batches in order to avoid skewing the dataset distribution of different states, actions, rewards, and next_states that the neural network will see.

2.4 Soft-update

As we know, algorithms like DQN have an idea of Target Network. Q learning updates the Q values using this equation : $Q(s,a) = Q(s,a) + \alpha(\underbrace{R + \gamma \cdot \max Q(s',a')}_{target} - Q(s,a))$

Instead of updating weights after a certain number of steps, we will incrementally update the target network after every run using the following formula:

$Target_{weights} = Target_{weights} \cdot (1-\tau) + model_{weights} \cdot \tau$, where $\tau \in (0,1)$.

The target network maintains a fixed value during the learning process of the original Q-network, and then periodically updates it to the original Q-network value.

While implementing our algorithms, the convergence to the wanted reward (successful landing of the lander) was slow and sometimes wasn't happening. When we used the soft-update strategy, it helped us. The convergence was way faster.

3 The algorithms

Due to the changes of the land on every game and due to randomization, the number of episodes we need to converge to the wanted reward changes according to the game. That is why, we ran our models several times and the results we are going to expose in this report are average results.

3.1 DQN

DQN is an algorithm that combines neural network and Q learning. We initialize our Main and Target neural networks. We choose an action using the Epsilon-Greedy Exploration Strategy and we update our network weights using the Bellman Equation. To get good results, we have two choices: either we input several frames or a single one. For the simple algorithms (without noise), we chose to input a single frame. The output will be the value of Q according to every state/action pair.

Like every Neural Network models, we have hyper-parameters to choose. We choose them by "playing" with them.

We tried different architectures, with different numbers of neurons, different numbers of layers, we also tried different values for the batch size, and for the learning rate. We tried to add drop-out but observed that it didn't improve the results.

The architecture we chose is composed of 3 hidden layers that all have 256 neurons. We used ReLU function as the activation function, Adam optimizer and MSE loss.

We set learning rate to 0.0005, batch size to 64, discount factor to 0.99, τ to 0.01 (for soft update of target parameters). Also, target update to 4 (all how often to update the network). We also need to set parameters for the espsilon-greedy strategy, those are $\varepsilon_{start} = 0.99, \varepsilon_{end} = 0.01$ and $\varepsilon_{decay} = 0.01$.

With those parameters, we obtained that DQN converges to a reward of 200 after **252** episodes (until at least 352).

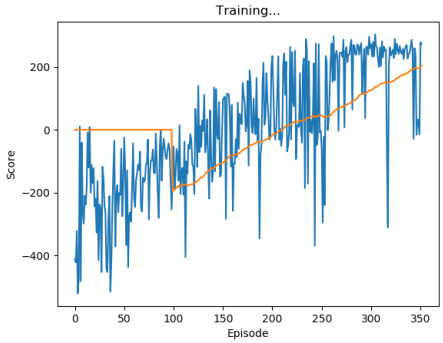

Figure 1: Average score by episodes of DQN

3.2 Double DQN

Since Q values are very noisy, when we take the maximum over all actions in DQN, we're probably getting an overestimated value.

The proposed solution is Double Q-learning. This algorithm uses two different function approximators that are trained on different samples, one for selecting the best action and other for calculating the value of this action, since the two functions approximators see different samples, it is unlikely that they overestimate the same action.

The difference between DQN and Double DQN is the way that we estimate the next action.

In Double DQN, the eval_net is used to estimate Qmax(s', a') in the Q_target and the Q_target is selected by the action estimated by the Q_eval. The Double DQN algorithm remains the same as the original DQN algorithm except for the weights updates.

$$y_j^{DoubleQ_1} = r_{j+1} + \gamma Q_2(s_{j+1}, \arg\max_a Q_1(s_{j+1}, a; \theta); \theta^-)$$

$$y_j^{DoubleQ_2} = r_{j+1} + \gamma Q_1(s_{j+1}, \arg\max_a Q_2(s_{j+1}, a; \theta^-); \theta)$$

The architecture we chose and the parameters are the same we used for DQN. Using MSE Loss or Huber Loss gave us similar results.

We obtained that Double DQN converges to a average of 200 after **190** episodes (until at least 290).

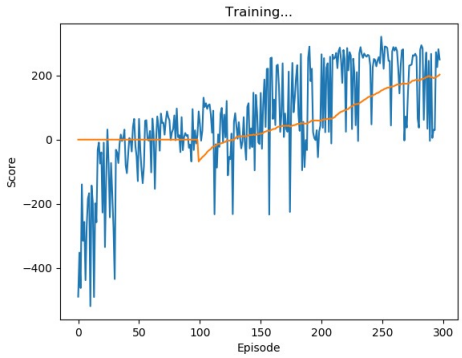

Figure 2: Average score by episodes of Double DQN

3.3 Dueling DDQN

Dueling DDQN divides the Q network into two channels: one output V which represents the Value, one output A which represents the Advantage of doing action a while in state s. Together we get Q. It separates the dominant and value functions before they are combined into the action function. The Q value of each action of Dueling DDQN is obtained by the following formula:

$Q(s, a) = V(s) + A(s, a)$

The dueling architecture can learn which states are valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way.

The dueling architecture shares the same input-output interface with the simple DQN architecture, the training process is identical. For the network architecture, we split the state-dependent action advantages and the state-values into two separate streams. We also define the forward pass of the network with the forward mapping.

After playing with the architecture and with the hyper-parameters, we got the best results with an architecture composed of the value stream with two layers of 128 neurons outputing a single neuron and the advantage stream with two layers of 128 neurons. We used the same parameters as in the algorithms above. Using MSE Loss or Huber Loss gave us similar results.

We obtained that Dueling Double DQN converges to a reward of 200 after **186** episodes (until at least 286).

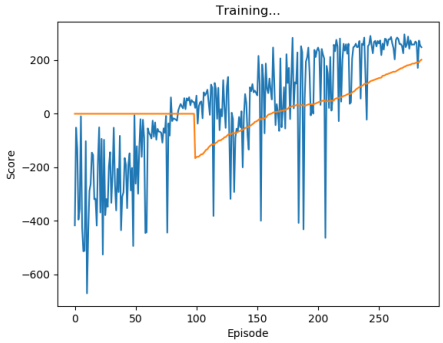

Figure 3: Average score by episodes of Dueling DDQN

3.4 Lunar-lander with uncertainty

After observing all of those algorithms, we'll do the observations in the real physical world that are sometimes noisy.

We add a zero-mean Gaussian noise with mean = 0 and std = 0.05 to PositionX and PositionY observation of the location of the lander. We ran the same algorithms, with the same parameters but with a noisy environment.

When we added noise, we had some convergence problems. Indeed, the model starts to learn and we can see improvements through the episodes, but then it starts to descrease, and re-increase and so on. We tried to use soft epsilon, to change the hyper-parameters etc. but it didn't help. Then, we added the history of the state, we saw that it helped, and the models improved. Thus, we decided to input several frames (memory_snap in the code), to take into consideration PositionX and PositionY of the three last steps. It stabilized the models.

We observe that the convergence is slower when we add noise. Indeed, it is quite more difficult for the model to learn. DQN converges to the wanted reward after **309** episodes (until at least 409), Double DQN after **362** episodes (until at least 462), and Dueling DDQN after **215** episodes (until at least 315). Even with noise, Dueling DDQN learns faster than simple DQN or than Double DQN. Here, we can see that it is a better algorithm. It learns quite fast even with noise.

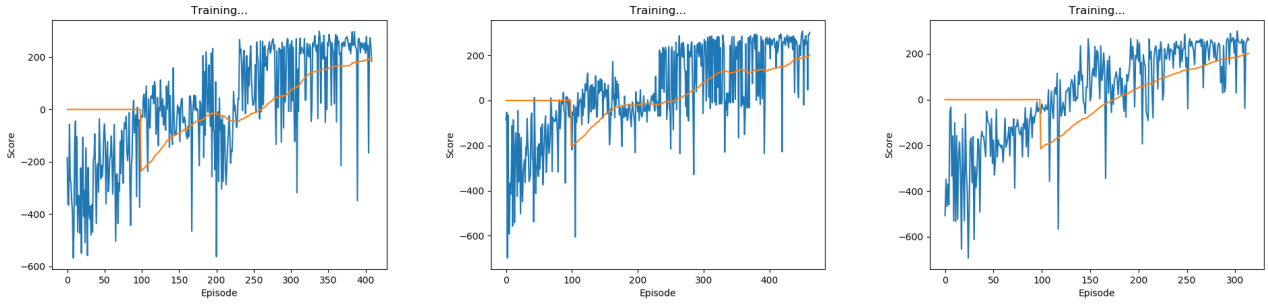

Figure 4: DQN, Double DQN and Dueling DDQN (in this order) with noise

3.5 Prioritized experience

As seen above, experience replay makes the agent remember and reuse experiences from the past. Experiences are uniformly sampled from a replay memory. However, some experiences may be more important than others and might occur less frequently. Prioritized experience changes the sampling distribution. It takes in priority the experiences which led to an important difference between the expected reward and the reward that we actually got. It means that we want to keep the experiences that made the neural network learn a lot.

When we begin the training, the lander will crash most of the times. However, rarely the lander will touch the ground without crashing, or land correctly. In that case, the difference between the expected result and the actual output would be significant, then there is a much higher probability for this experience to be sampled. We want to use this experience multiple times to train the neural network as an example of what is working and what direction we should take to improve the weights of the network.

We add in our replay buffer the magnitude of our TD error $|\delta_t|$. It would, then, be composed of $(s_t, a_t, r_t, s_{t+1}, |\delta_t|)$. We define the constant e that assures that every experience has some probability to be taken.

Priorities : $p_t = |\delta_t| + e$.

The priority is updated according to the loss obtained after the forward pass of the neural network. The probability is computed out of the experience priorities, while the weight is computed out of the probabilities. We get two more hyper-parameters α, which is the level of prioritization and β which controls how much the important sampling weights affect the learning (close to 0 at the beginning of learning and annealed to 1 to the end of learning). At the end of the training, we want to sample uniformly to avoid overfitting due to some experiences being constantly prioritized. We'll implement those equations:

$\mathbb{P}(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$, where p_i is the priority i.

$w_i = (\frac{1}{N} \cdot \frac{1}{\mathbb{P}(i)})^\beta$, where w_i is the weight i and N is the number of experiences.

After playing with the parameters, we chose $\alpha = 0.6$ and $\beta = 0.5$. We also remarked that changing the MSE loss to the Huber Loss gives us better results using PER because it is less sensitive to outliers.

We obtained good results. This strategy seems to be a good one, especially for DQN but it's not improving the results significantly. When we add noise, it doesn't help.The reason might be because it gives a higher probability to be samples to the experiences with high loss. And this can be caused by the noise.

DQN converges to the wanted reward after **269** episodes (until at least 369), Double DQN after **252** episodes (until at least 352), and Dueling DDQN after **171** episodes (until at least 271). We still observe that Dueling DDQN learns faster.

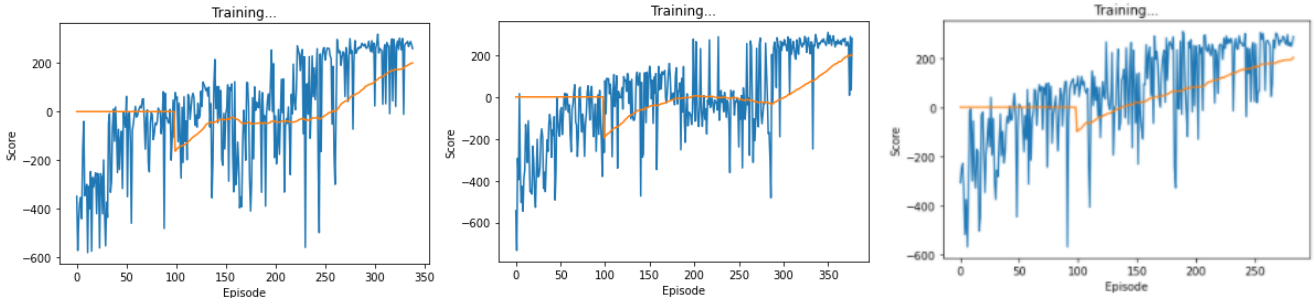

Figure 5: DQN, Double DQN, Dueling DDQN with PER (without noise)

4 Results and conclusion

After implementing, and running all of those algorithms, we can conclude that the results depend a lot on the parameters, the model and the algorithm. We could observe that the Dueling DDQN algorithm gives us great results and in a very few episodes. Double DQN gives us better results than DQN (without noise). Moreover, when implementing with uncertainty, we saw that the learning process is slower. We needed to take into account the history of the state to learn better and faster. Dueling DDQN learns fast despite the noise while Double DQN and simple DQN are slow learners. The strategies we used (greedy-epsilon, experience replay etc.) are very interesting. We could observe, through experiences that we need them to improve our models and our learning process. Prioritized experience strategy doesn't really improve the results, and is not helpful in an uncertain environment.

	DQN	Double DQN	Dueling DDQN
Simple	252	190	186
With noise	309	362	215
With PER (without noise)	269	252	171

As a further work, it would be interesting to add information about the frame (the form of the land), using CNN.

Our Colab Project (link)

References

[1] Q learning with Pytorch

[2] Soft update (GitHub)

[3] Dueling DQN - code

[4] Explanation on DDQN and Dueling DDQN

[5] Soham Gadgil, Yunfeng Xin, Chengzhe Xu, "Solving The Lunar Lander Problem under Uncertainty using Reinforcement Learning" , 2020

[6] Adam Gjersvik, "Landing on the Moon with Deep Deterministic Policy Gradients"

[7] Y. Lu, M. S. Squillante, and C. W. Wu, "A control-model-based approach for reinforcement learning," 2019.