Deep Learning: Optimization. Coursework

Victor Ceballos-Espinosa & Orestis Makris

{ Victor.Ceballos-Espinosa@city.ac.uk, Orestis.makris@city.ac.uk }

1. Basic Q-Learning implementation

1.1 Domain and task

The chosen task belongs to the domain of pathfinding problems. To be more specific, it consists in carrying a passenger from an origin location to a destination in a 4x4 cells grid world where there are walls which cannot be overpassed, a fixed passenger's origin (green cell) and destination (orange cell) and random starting points for the taxi agent (figure 1).

The way we modeled this task as a reinforcement learning problem is that the location of the agent in the environment encodes the different states (A-P without the passenger inside the taxi & A'-P' with the passenger inside it) and the possible actions include go up, go down, go left, go right, pick up the passenger and drop the passenger. Every reinforcement learning problem needs rewards, and in this case, we have decided to give the agent a -1 reward at every allowed step, a -5 reward if it performs a forbidden action, a +10 reward whenever it picks the passenger in the right cell and +10 reward drops the passenger in the right cell. Lastly it is important to remark that once the agents drops the passenger at state D', it will reach a terminal state called END, finishing the current episode.

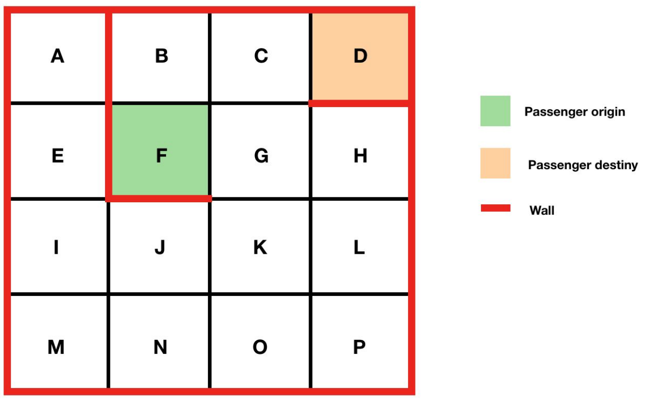


Figure 1. Environment.

1.2 State transition and reward function

State transitions are very simple. At every state, the agent is able to perform one of the six possible actions (go right, go left, go up, go down, pick passenger or drop passenger). As an example, if the agent is in J and it executes the action to go to the right, it will move to state K. Or if it is in state F and it executes the action to pick the passenger it will transit to state F'. Nonetheless, if the agent attempts to execute a forbidden action, it will remain in the same state. Forbidden actions include trying to go through a wall, picking a passenger where there is none and dropping the passenger anywhere outside the destination cell.

Regarding the reward function, every action execution will lead the agent to perceive a -1 reward. However, there are some exceptions. If the agent tries executing a forbidden action, it will receive a -5 reward. Additionally, when it picks the passenger and when it drops it in the right positions, it will receive 10 points for each of the actions.

State transitions and reward function were encoded as excel sheets and then imported as dataframes in Python using Pandas.

1.3 Graph representation and original reward matrix

Every Reinforcement Learning problem can be represented as a Markov Decision Process. In this case, the resulting MDP can be observed in figure 2.

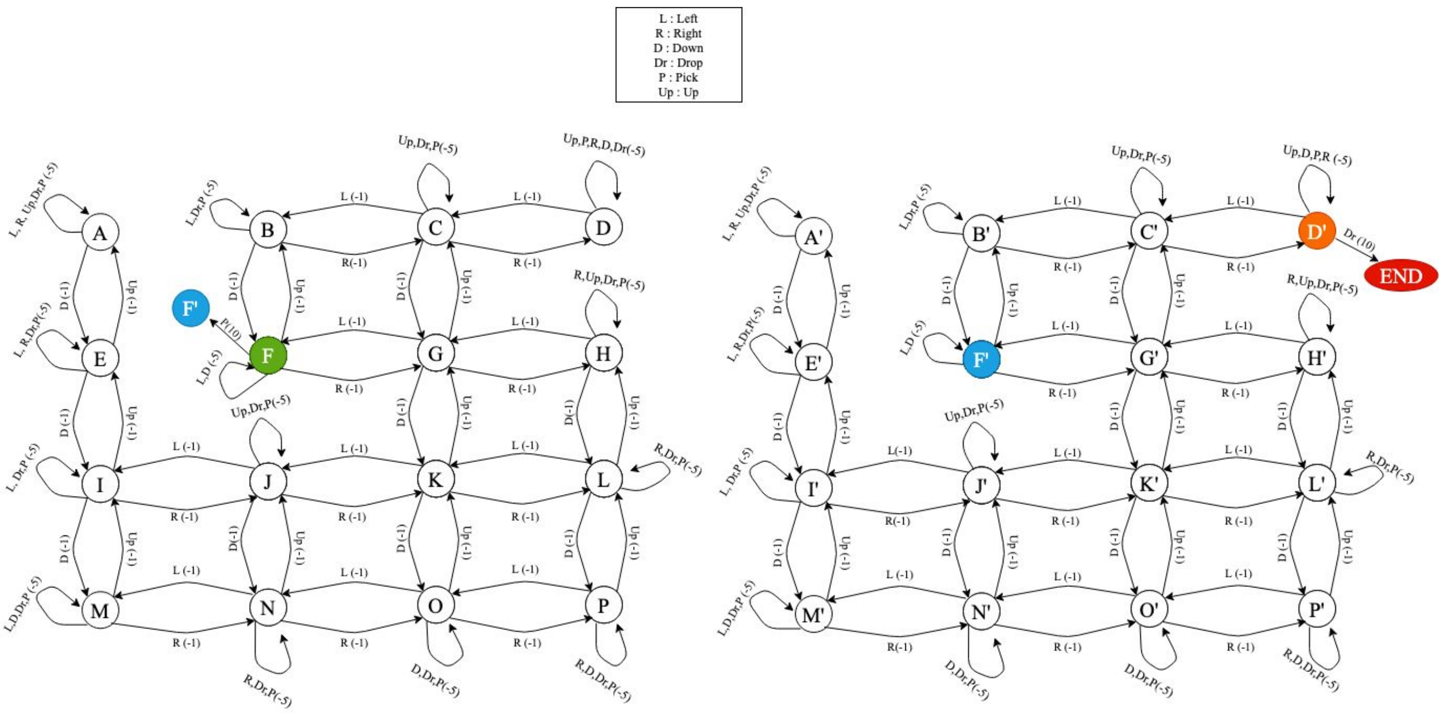
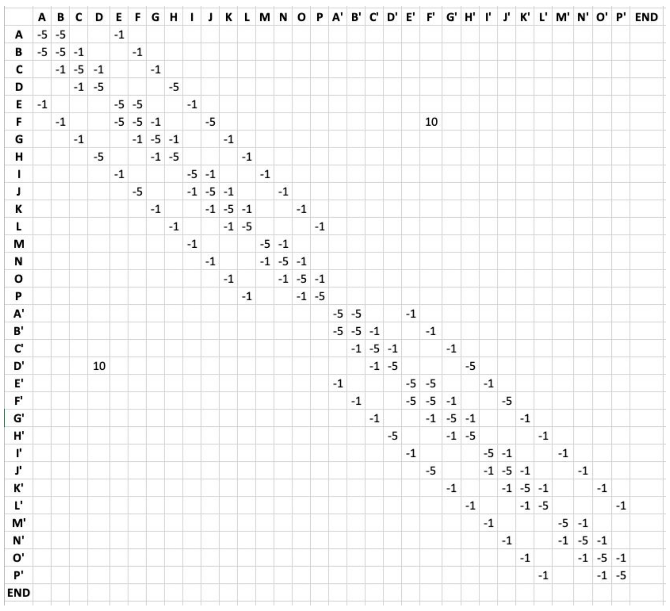


Figure 2. Left: MDP for states without the passenger. **Right:** MDP for states with the passenger.

The MDP on the left shows the agent's transitions while the passenger has not been picked up. However, once the passenger has been picked up (transition from F to F'), the agent will move to a different states space represented by the MDP on the right. Figure 2 should be observed as if F on the left MDP and F' on the right MDP were connected with a transition executing the PICK UP action with a +10 reward.

Finally, in figure 3, the original reward matrix is displayed.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	A'	B'	C'	D'	E'	F'	G'	H'	I'	J'	K'	L'	M'	N'	O'	P'	END
A	-5	-5			-1																												
B	-5	-5	-1			-1																											
C		-1	-5	-1			-1																										
D			-1	-5				-5																									
E	-1				-5	-5			-1																								
F		-1			-5	-5	-1			-5												10											
G			-1			-1	-5	-1			-1																						
H				-5			-1	-5				-1																					
I					-1				-5	-1			-1																				
J						-5			-1	-5	-1			-1																			
K							-1			-1	-5	-1			-1																		
L								-1			-1	-5				-1																	
M									-1				-5	-1																			
N										-1			-1	-5	-1																		
O											-1			-1	-5	-1																	
P												-1			-1	-5																	
A'																	-5	-5			-1												
B'																	-5	-5	-1			-1											
C'																		-1	-5	-1			-1										
D'				10															-1	-5				-5									
E'																	-1				-5	-5			-1								
F'																		-1			-5	-5	-1			-5							
G'																			-1			-1	-5	-1			-1						
H'																				-5			-1	-5				-1					
I'																					-1				-5	-1			-1				
J'																						-5			-1	-5	-1			-1			
K'																							-1			-1	-5	-1			-1		
L'																								-1			-1	-5				-1	
M'																									-1				-5	-1			
N'																										-1			-1	-5	-1		
O'																											-1			-1	-5	-1	
P'																												-1			-1	-5	
END																																	

Figure 3. Reward matrix.

It is important to note that the rewards from the state END are all nulls as once this state is reached, the episode is finished so there are no possible transitions from END to any other state.

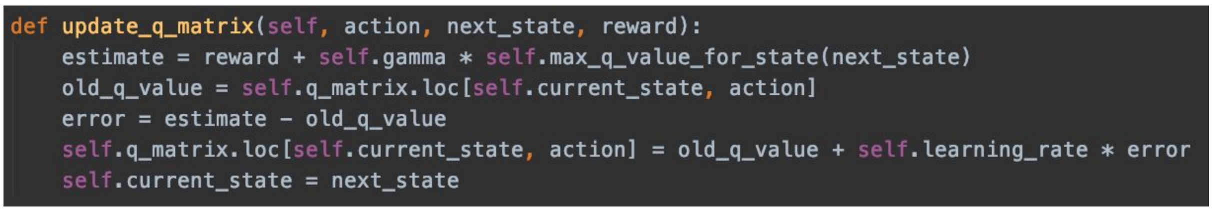
1.4 Q-Learning parameters and update process

In order to update the Q-matrix, Q-Learning update rule was used. Such update rule is defined by the following equation:

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t,a_t)}_{\text{old value}} \right)$$

Figure 4. Q-Learning update rule equation. Source: (Wikipedia, 2020)

As it can observed, a learning rate and discount factor are needed. After performing a hyperparameter tuning, the best combination consisted in a learning rate (α) of 0.05 and a discount factor (γ) of 0.9, which were the values used for the final model. The update rule was encapsulated on a method in the written code called *update_q_matrix* as it can be observed in figure 5.



```
def update_q_matrix(self, action, next_state, reward):
    estimate = reward + self.gamma * self.max_q_value_for_state(next_state)
    old_q_value = self.q_matrix.loc[self.current_state, action]
    error = estimate - old_q_value
    self.q_matrix.loc[self.current_state, action] = old_q_value + self.learning_rate * error
    self.current_state = next_state
```

Figure 5. Code for the Q-Learning update rule.

The update process consists in the following steps:

1. Select an action according to a policy
2. Execute the action, get the immediate reward and the state$_{new}$
3. Calculate the maximum Q-Value from the new state
4. Calculate an estimate for the Q-Value based on the equation: (estimate = r + γ * max$_a$Q(state$_{new}$, a))

5. Calculate the error based on the equation: (error = estimate - Q_{old})
6. Calculate Q_{new} as ($Q_{new} = Q_{old} + \alpha * \text{error}$)

In order to show how these steps work in the implementation, three consecutive update processes different from the initial ones were selected and shown below. It has been avoided to show the initial ones as the Q-Values are initialized to zero and not a lot can be extracted from this in terms of observing the update process.

Update process' sample 1
From state G', executing action "DOWN", the agent goes to state K'
Q_{old}(G', DOWN) = -0.03940399
r = -1.0
Q-values for K' are: [-0.0199, -0.01, -0.01, -0.0199, -0.05, 0.0]
error = ((r + γ * $\max_a$Q(K', a)) - oldQ) = (-1.0 + 0.9 * 0.0) + 0.03940399 = -0.96059601
$Q_{new} = Q_{old} + \alpha$ * error = -0.03940399 - 0.01 * 0.96059601 = -0.0490099501

Update process' sample 2
From state K', executing action "PICK UP", the agent goes to state K'
Q_{old}(K', PICK UP) = -0.05
r = -5.0
Q-values for K' are: [-0.0199, -0.01, -0.01, -0.0199, -0.05, 0.0]
error = ((r + γ * $\max_a$Q(K', a)) - oldQ) = (-5.0 + 0.9 * 0.0) + 0.05 = -4.95
$Q_{new} = Q_{old} + \alpha$ * error = -0.05 - 0.01 * 4.95 = -0.0995

Update process' sample 3
From state K', executing action "UP", the agent goes to state G'
Q_{old}(K', PICK UP) = -0.0199
r = -1.0
Q-values for G' are: [-0.029701, -0.0490099501, 0.0, -0.030235350691, -0.0995, -0.05]
error = ((r + γ * $\max_a$Q(K', a)) - oldQ) = (-1.0 + 0.9 * 0.0) + 0.0199 = -0.9801
$Q_{new} = Q_{old} + \alpha$ * error = -0.0199 - 0.01 * 0.9801 = -0.029701

1.5 Hyperparameter tuning

When it comes to perform the hyperparameter tuning, there are mainly three parameters to tune: the policy, the learning rate (α) and the discount factor (γ). In order to avoid the exponential number of combinations when building the different combinations, it has been separated the policy tuning and the learning rate and discount factor tuning.

As it can be observed in figure 6 Boltzmann policy outperforms ε-greedy policy in terms of episodes needed to converge. It is clear that because of the uniform random selection performed by e-greedy, there is a lot of noise in its performance with respect to the episodes. However, when experimenting, we could see that by changing epsilon's value in the ε-greedy policy to a lower value, the line became closer to Boltzmann policy's one, converging faster in approximately 500 episodes. Eventually as Boltzmann policy showed to be more consistent, it was selected.

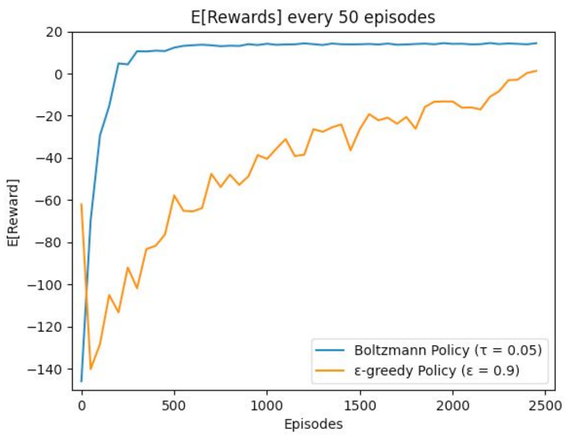
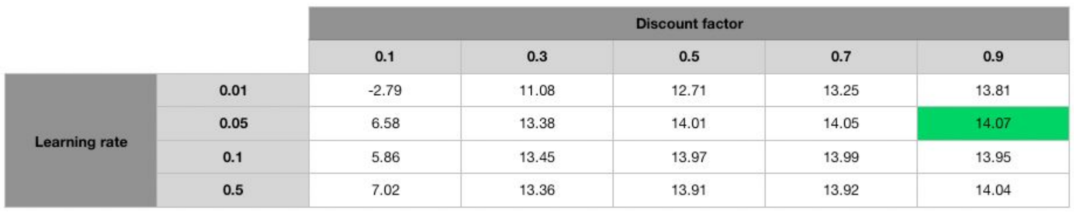


Figure 6: Policy tuning comparing performance w.r.t episodes.

As it can be seen, after running each combination ten times, the best combination of learning rate and discount factor consist in 0.1 and 0.7 respectively, producing 14.07 as expected reward on average. Therefore these were the values used for the final agent. This result can be seen in the table 1, in which the results of averaging the executing of each combination ten times have been recorded.



		Discount factor				
		0.1	0.3	0.5	0.7	0.9
Learning rate	0.01	-2.79	11.08	12.71	13.25	13.81
	0.05	6.58	13.38	14.01	14.05	14.07
	0.1	5.86	13.45	13.97	13.99	13.95
	0.5	7.02	13.36	13.91	13.92	14.04

Table 1: Learning rate and discount factor hyperparameter tuning using a Boltzmann policy.

1.6 Analysis

As it can be observed in figure 6 in which the performance with respect to the episodes is shown, in the beginning, the agent using the best policy and hyperparameters combination is performing extremely bad, obtaining a negative reward of around -120. This means that a lot of forbidden actions were executed. However, after approximately 250 episodes, the expected reward surged to 0. This is explained by the fact that the agent was executing way more accepted actions than forbidden actions and it is completing the goal. From this point on, the agent's performance continued rising until converging to the expected value of roughly 14.

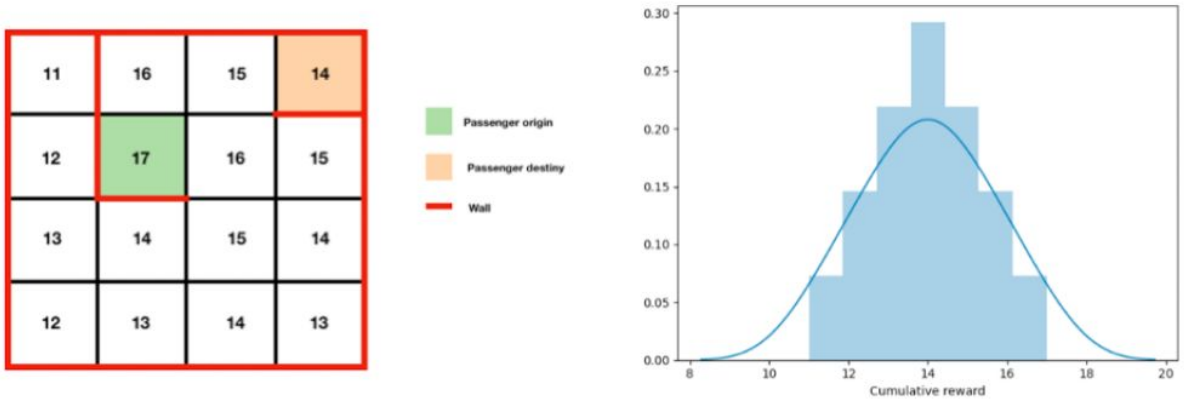


Figure 7. Left: Cumulative reward starting at each cell following the optimal policy. **Right:** Histogram and KDE of cumulative rewards

As shown in figure 7, before the agent was trained, it was already expected that the agent would converge close to a expected reward value close to 14. On the left image in figure 7, the cumulative rewards obtained following the optimal policy starting from each cell is displayed. Additionally, on the image on the right, the cumulative rewards' histogram together with a Kernel Density Estimation is shown and it is clearly visible that it follows a normal distribution shape with 14 as mean. Therefore, it could be inferred that the average expected reward would be 14.

All the previously discussed qualitative behaviour is explained by the used of the Boltzmann policy. In Boltzmann policy, the probability of executing an action is based on their Q-Values given the state. This way, every time an action needs to be executed, it is sampled from that distribution. In this sense, the problem of balancing between exploration and exploitation is solved as all actions have a probability of been selected (exploration) but the ones with a higher Q-Values have more chances to be selected (exploitation).

Tau values have a critical impact in Boltzmann distribution equation. High values of tau makes all actions equal to each other because it diminishes the impact of Q-value (i.e. it becomes closer to a uniform distribution). In the opposite way, as tau value decreases, actions with high Q-values have higher probabilities of being selected. In the case of the implemented agent using Boltzmann policy, this tau value was fixed to 0.05, obtaining the results presents at figure 6. This explains the fast convergence observable in figure 6 as having a very low tau value will make the agent to exploit a lot. This together with the fact that the problem is not extremely difficult, makes the performance to converge that fast.

2. Advance implementation

2.1 Domain and Task

The 'Lunar Lander' environment from openAi gym was used as a second part of our coursework. It is worth mentioning that the purpose of this game is to successfully land the rocket on a landing pad to the "moon". Especially, the rocket consists of two legs, the main body, and three jet engines which are responsible to decrease the falling speed and stabilize the ship. Jet engines can change the acceleration of the rocket thus, these actions do not affect the movement instantly. The distinctive difficulty of the lunarlander_v2 is that the shape of the surface is constantly changing, and the lander must adapt to all these adjustments.

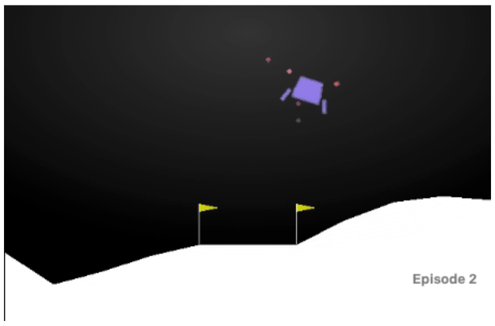


Figure 8. LunarLander-v2 environment. Source: (OpenAI, 2020)

To continue with, as the previous reinforcement learning task in the basic implementation, Lunarlander_v2 has a state space, possible actions, and rewards. The state space is a continuous 8-dimensional vector that contains position (x, y), velocity (v_x, v_y), lander angle θ, angular velocity, and right leg, left leg as ground-contact sensors. The discrete actions are: fire bottom (main) engine, fire right engine, fire left engine and do nothing.

A key point to remember is that all possible rewards that the agent receives at each step are determined by the openAI gym. Specifically, an accurate landing inside the pad can give a reward in the range between 100 and 140 points, depending on the received amount of penalties which are generated when the lander moves away from the landing target. Moreover, firing the main thruster costs 0.3 points per frame from the total reward while each leg that touches the ground, nets the lander with 10 points. Finally, the episode finishes when the ship either crashes or flies out of bounds with a negative reward of 100 points, whereas the agent gains 100 points with a successful landing (OpenAI, 2020).

2.2 Policy Gradient

Many implementations of RL algorithms are based on building models that can represent value functions such as DQN. However, there are other approaches like Policy Gradient in which the policy is represented by a neural network which is optimized using backpropagation. For the advance implementation of this project, Policy Gradient has been selected. In order to train the neural network representing the policy it is necessary defining a loss function. In this case, the objective (loss) consists in the expected reward throughout different trajectories in the environment.

$$J(\theta) = \mathrm{E}\Big[\sum_{t=0}^{H} R(s_t, u_t); \pi_\theta\Big] = \sum_{\tau} P(\tau; \theta) R(\tau)$$

Equation 1. Policy gradient loss function. Source: (Hui, 2018)

In equation 1, the loss function defined for Vanilla Policy Gradient is presented. However, VPG suffers from high variance and low convergence. In order to deal with these two disadvantages, three improvements were applied to VPG:

- **Mini-batch training:** Initially, the policy gradient was updated every episode. This was producing a lot of variance in which once the agent managed to perform around 200 as expected reward, it started decreasing until a point in which the expected reward turned negative. After the code was changed to use mini-batch training, the variance was extremely decreased.

- **Discount rewards:** A discount factor was applied to the obtained immediate rewards in order to weight them so that more recent rewards are taken more into account and further-in-time rewards are taken less into account. This helped reducing the variance as well.

- **Use of advantage-like normalization:** Finally, the last improvement was applied to VPG in which once the rewards were discounted as explained in the previous bullet point, they were normalized subtracting the mean of the discounted rewards and dividing by the standard deviation.

It is important to mention that the gradients to update the neural network representing the policy were not calculated manually. Instead, the loss was calculated and Pytorch was used to calculate the gradients based on the loss function and the topology of the neural network.

2.3 Hyperparameter tuning

As explained in (Islam, 2015), how fast gradient ascent (and gradient descent) convergence depends on the learning rate. In this advance implementation, gradient descent was used in order to optimize the policy given the loss. Therefore, the learning rate is an important hyperparameter. In (Islam, 2015), the decision of implementing a decaying learning rate was taken. However, this lasted two days just to find the best decaying factor for the learning rate. This is an example of how in more advance implementations, hyperparameter tuning could become extremely difficult. As a result and because of the lack of computation power and time, for this project the hyperparameter tuning for this advance implementation was performed in a simpler way in which specific combinations of the hyperparameters were tried. As based on the initial analysis it could be observed that there were variance and low convergence problems, it was decided to tune the following hyperparameters:

- **Learning rate:** Which is used by Pytorch's optimizer in order to update the weights smoother.
- **Mini-batch size:** Which is used to perform batch updates instead of updating the weights after every episode.

Only two combinations of hyperparameters were tried as it was managed to solve the environment with the second combination of hyperparameters faster and reducing the variance drastically. The first tried combination consisted of a learning rate of 0.1 and a mini-batch size of 50. Nevertheless, it could be seen that the performance eventually got out of a local minimum. As a result, a smaller learning rate 0.01 and mini-batch 20 was tried, obtaining the results shown in figure 9.

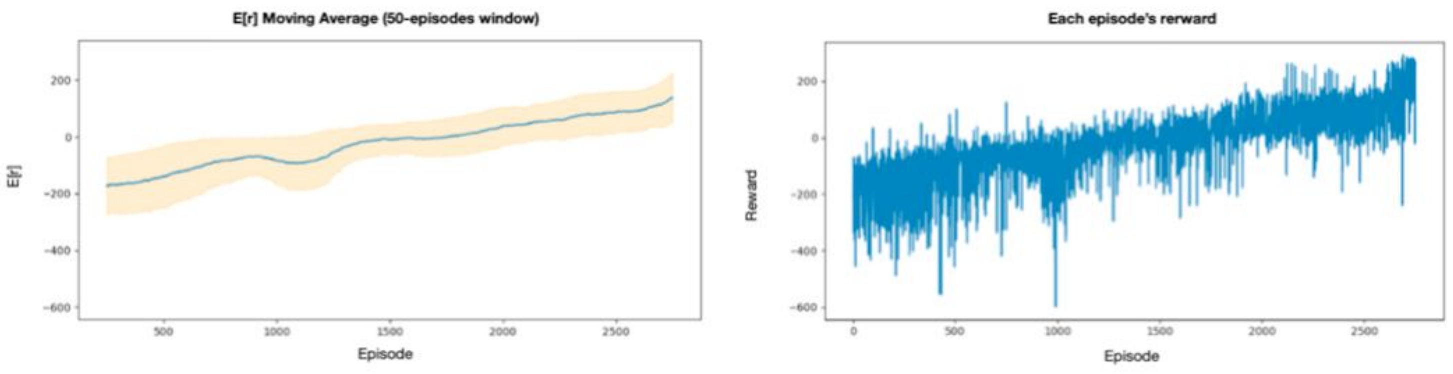


Figure 9. Left: Expected reward Moving Average**. Right:** Reward of each episodes.

2.4 Analysis

Initially, the advanced implementation consisted in Vanilla Policy Gradient, which suffers a lot from variance and low convergence and this was noted as it could be observed that once the agent managed to converge to an expected reward of around 200 points, its performance started to decrease rapidly getting negative rewards. Therefore, we had to think about ways to improve it in order to reduce the variance and to increase the convergence. To this end, it was decided to apply the improvements explained in the Policy Gradient section, which consisted of discounting and normalizing the rewards as well as using mini-batch training and early stopping. Once these improvements were implemented, we could observe a huge upgrade resulting in the performance shown in figure 9, in which it is observable that the moving average of the expected reward converges close to 200 points and it does not decay after a certain number of episodes. Apart from this, it was decided to add dropout layers after each linear layer in the neural network representing the policy, improving the robustness of the policy resulting in the network topology described in figure 10.

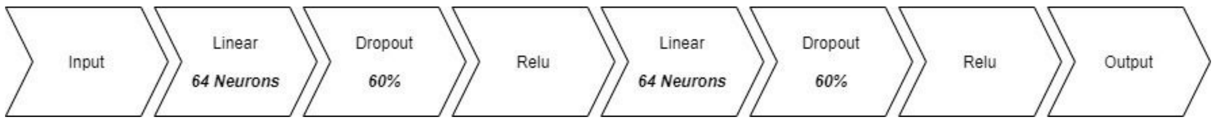


Figure 10. Neural network topology

The environment is solved at 200 points as stated in OpenAI website (OpenAI, 2020). Nevertheless, the agent can achieve more than 200 points when the spaceship is generated close to landing pad, being able to obtain a maximum of 260 points. This happens only when the lander is spawned just above the flags and the landing is “free fall”. In figure 9 this behaviour is displayed at the end of the training when the agent managed to obtain rewards higher than 200.

As it can be seen in figure 9, different from the performance of Q-Learning shown in figure 6, the performance of Policy Gradient increases slower. Here, it is clearly observable that even though a lot of improvements have been applied to Policy Gradient, the low convergence speed is still present.

From a qualitative point of view, in the first episodes, the ship was constantly crashing, not being able to understand that it needed to start the thrusters in order to decrease the falling speed. Nevertheless, after approximately 1500 episodes, it learned to use the engines to that the falling speed is decreased as well as keeping the right direction so that both ship’s legs touched the ground. Finally, after approximately 2600 episodes it successfully learned to correct its direction in order to land within the landing pad marked with two flags.

References

Hui, J., 2018. RL - Policy Gradient explained. [online] Available at: <https://medium.com/@jonathan_hui/rl-policy-gradients-explained-9b13b688b146>

OpenAI, 2020. LunarLander-V2. [online] Available at: <https://gym.openai.com/envs/LunarLander-v2/>

Wikipedia, 2020. Q-learning. [online] Available at: <https://en.wikipedia.org/wiki/Q-learning>

Islam, R. (2015). *Improving Convergence of Deterministic Policy Gradient Algorithms in Reinforcement Learning*. [PDF] p.69. Available at: <https://riashatislam.files.wordpress.com/2016/11/riashat-islam-report.pdf>.