**Slide 2:**

* Reinforcement learning is a machine learning training method based on rewarding desired behaviors and punishing undesired ones. In general, a reinforcement learning agent -- the entity being trained -- is able to perceive and interpret its environment, take actions and learn through trial and error.
* The agent perform an action in the given environment which leads to the interpreter that will provide the agent with a reward (positive or negative) and the next state.

**Slide 3:**

* Lunar Lander is part of the Box2D Gymnasium environments which is a classic rocket trajectory optimization problem.
* The game have 4 possible actions: do nothing, fire left engine, fire main engine and fire right engine.
* Here, the state is an 8 dimensional array that indicate the horizontal and vertical positions, orientation, linear and angular velocities and state of each landing leg.
* The reward has been customized so the purpose of the game is not just to land, but a more realistic landing where the ship constantly reduces its speed so that it touches the ground slowly.
* An episode is completed when the lander touches the ground or crashes.

**Slide 4:**

* Here, we will see an agent taking random actions to land the ship.

**Slide 5:**

* To solve this kind of problems, the Q-Learning algorithm has a basic concept of value-action function (the function in the right top corner) which is updated at each step (or action taken).
* Being an off-policy method means that this algorithm have no “brain”, just some “Q-values” (that's where that Q or Quality comes from) and a value corresponds to a state - action pair.
* These values are stored in a Q-table and when the agent is in a certain state, it search the state in the table and uses an ε-Greedy strategy to choose an action.
* The ε-Greedy strategy consists of generating a random number, if the number is less than epsilon, the agent takes a random action, otherwise it chooses the action from the table with the higher Q-value. Epsilon decreases from 1 with each episode.
* Nevertheless, there is a problem with this Q-table, we can’t store it because it will be too big having all the possible states of the ship.

**Slide 6:**

* The problem can be solved by using a neural network.
* A neural network is a group of artificial neurons that are interconnected and act like a brain for the agent. It is organized in layers (input layer - we put here the information we have), several hidden layers (where the information is processed) and the output layer (where we can see the result).
* The neural networks is used as an approximation for the function previously presented. As information we give the state and as a result we get 4 Q-values for our 4 actions.
* An efficiency problem appears when we compare 2 consecutive states. There is not much difference between them, so the agent can’t learn something new from both. Therefore, experience replay is involved. In a replay memory will be stored 100.000 states and during training a small batch of random states will be selected to train the neural network with.
* Even though the agent has a brain, this won’t solve properly the problem. When it will approximate the Q-values, there will be no reference that will tell the agent if it did a good job or not. Without a reference, it will be like a cowboy trying to catch a caw, but the cow is running as fast as he is. Hence, it needs a second opinion (a second neural network). The parameters of the main network are copied from time to time to the second one. After the training is over, the two networks will be approximately identical.

**Slide 7:**

* This is roughly what the neural network used looks like.
* You can compare this neural network with a function with 173.100 parameters. Every connection is a number named “weight” and every neuron have a bias that represent an activation number.
* More details about this neural network and the algorithm presented can be found on my github page, but I will give you more details at the end.

**Slide 8:**

* Now we will watch how the agent really performs actions during training and how it evolves.

**Slide 9:**

* Most of the improvements brought by me to this project are in the area of rewards.
* After every step a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode.
* For each engine firing the reward is decreased.
* For each leg that is in contact with the ground is increased.
* The episode receive an additional reward for crashing or landing safely respectively.
* But the problem is that the goal is just to land, in real life a lander reduces its speed according to the distance to the ground, so in the customized reward environment, the focus is on that
* The agent receive points if it adjusts the speed to the remaining distance, also when it does not exceed a certain horizontal speed.
* There are several improvements on this idea, related to the angle of the ship, angle velocity, combinations of all these, etc.
* However, I noticed that an agent learns faster and better when his reward decreases significantly for certain mistakes, than when it increases for what he does well.
* You can see that during learning the default environment have those spikes (this means that the ship crashed or landed badly). In the custom environment this problem disappeared.
* After 50000 episodes played, the performance is as follows: …
* Using the same neural network and algorithm, in the default environment the agent needs …

**Slide 10:**

* This training time is caused by the episode steps needed to land or crash and by number of episodes needed to have a well-functioning agent…
* The fuel consumption is improved as well in the custom environment…

**Slide 11:**

...

**Slide 12:**

Q&A