Land lunar lander, Land!

Khalil Nouili

Abstract

In this project, I will be helping the lunar lander to learn how land by its own using reinforcement learning technique called Deep Q-learning.

In this paper, I will try to solve a reinforcement learning problem in a continuous state environment. I will be discussing the specific work environment, details about the solution implemented and finally some analysis about the algorithm results.

Environment description

In this section, I will be describing the lunar lander objective and how he is interacting with the environment.

Lunar lander rules!

The goal of the lunar lander is to land in the landing spot in a short time.
In order to do so the lander has to pick the right action at the right moment.
Before diving deeper on how he is going to pick the right action (policy), let's understand first the different parts that are shaping this problem.
As a way to describe this problem, Markov chain process comes to the rescue.

Markov decision process

Inspired from Markov chain, Markov Decision process is as follow:

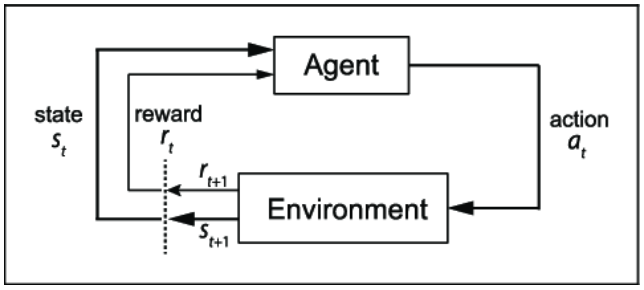

Figure 1: Markov Decision Process

At each time-step t, our agent with a state s_t will perform an action a_t that can be both discreet ($a_t \in$ our space of actions) or continuous according to a certain policy. Thanks or due to this action, the environment will reward the agent with a value $r_t \in R$.
As we can see this value can be both positive or negative so that we can tell the agent that his action was a good action or disastrous.
The agent receives then the new state s_{t+1} which represents the consequence of the action he took.

In other words, the goal of the reinforcement learning is to maximize the agent's future rewards given how it did perform in the past.

More formally a MDP in this environment is defined by:
Agent: The lunar lander
States: 8-dimensional state:
x, y: The agent coordinates in the 2-dimensional space.
$\dot{x}$, $\dot{y}$: Horizontal and vertical angular speed.
θ: Angle of the lander.
$\dot{\theta}$: Angular speed of the lander.
leg_L, leg_R: Indicator whether lander's leg are on the ground or not.
Actions: 4 actions:
Do nothing.
Fire the left engine.
Fire the right engine.
Fire the main engine.
Rewards:
leg-ground contact reward: +10.
Fire main engine: -0.3.

Fire left/right engine: -0.03.
lander crashes: -100.
lander landed: +100.

After understanding the lunar lander objective and environment, let's start discussing the method I am going to implement to hep our little lander.

Technical state of art

One of my major concern is an algorithm that is capable to learn everything by itself so that it can both learn from its mistakes and tune better its actions. One of the best and fast solution to implement that came into my mind is applying deep Q-learning learning.

In this section I will talk more about why did I choose this method, what it is and how did I manage to implement it.

Why did I choose this method?

In order to tackle this problem, many methods came to my mind and most of them can solve it but not at the same time or with the same accuracy.
First thing that came into my mind is the use of model-free reinforcement learning methods such as Monte-carlo or Q-learning. The Q-table (cheat sheet) created thanks to Q-learning can help our model to map the correct action for each state but in our case, we have a huge amount of states and huge amount of possible actions that our model can take (6 continuous variables), this would create a very long table with billions of cells which will cause :

- the need of a high memory to both save and update the Q-table.
- a huge amount of waiting time required to map the whole table to both read it and update it.

Due to these cons, I was in between making the states more discrete and we lose some information or switching to an other approach.
I couldn't afford wasting time and resources to work on this algorithm but thanks to deep Q-learning we could approximate these Q-values with neural networks.

Deep Q-learning

Thanks to the neural networks, we can approximate the Q-value function as showed in the Figure 2 :

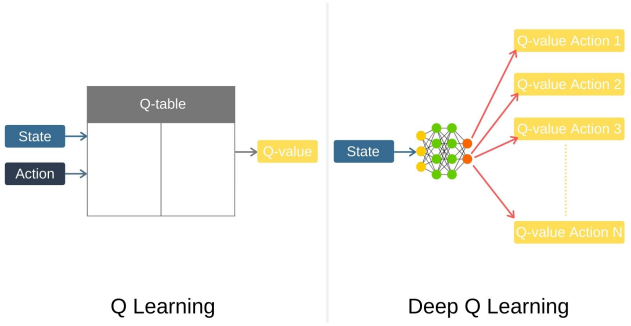

Figure 2: Q-learning Vs deep Q-learning

Here instead of having a very long table that we need to map every time we need to get the action a_t with the maximum reward r_t in state s_t, we can use neural networks instead which

- input: state
- output: a probability distribution of the actions that are more likely to give us better reward.

Here our model doesn't need to save everything that he encountered during the training, instead the neural network updated weights will create the best fit function to map the states into actions.

Deep Q learning principal

This part provides an in-depth explanation of the proposed solution and algorithm that is used to provide a better understanding about how we did apply the deep Q learning technique.

As discussed in the last section, the deep Q-learning is an upgraded version of the Q-learning. Instead of the use of the Q-table, we use here a neural network. All of the other steps regarding Q-learning remains unchanged:

- The agent selects the action suggested by the maximum output of the Q neural network.
- The agent interacts with the environment using that action.
- The agent receives a reward regarding the action taken.
- The agent updates his model weights using that reward
- This steps will be repeated until the weights of the model finally converge.

Here, in the third step we need to calculate the loss of our neural network.
In the beginning, the target needs to be calculated every time:

$$Target = R_{t+1} + \gamma argmax(Q(S_{t+1}, a)) \quad (1)$$

In the last equation, γ represents a number between 0 and 1 and thanks to it we define whether we are going to take into consideration long term rewards rather than short ones.

Finally after the target calculation we can update to the neural network weights as follows:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha * mse(Target, Q(S_t, A_t)) \quad (2)$$

As good this method as may seem, I encountered several problems while implementing it:

- The same model is calculating both the predictions and targets which may cause to its divergence.
- The exploitation/exploration trade off.

In the next parts, I will talk more about some ways I used to solve these two issues.

Experience Replay

In this part, I tried to solve the model divergence problem by simply dividing the two jobs to two models instead of only one:

- The online model will be responsible of the actions that the model is taking overtime and will be updated at every step.
- The target model will be the one that calculates the future reward at every step with which we will update the online model weights. The weights of this model will be updated at every episode by copying the online model weights.

In order to update the online model, we need to use the target model result and then perform the mean squared error to calculate the loss and update the weights. Each time when updating the online model and instead of only fitting the weights with the recent step results, we can store the past experienced history $(s_t, a_t, r_t, s_{t+1}, done)$ and fit it in a batch each time.

Epsilon-greedy

As for the solution of the second issue and since the agent here is the responsible either for the gain or the loss, it focuses to find a balance between exploration (of new experiences) and exploitation of current knowledge (previous experiences) to get the maximum reward. Our model needs not only to exploit the previous data but also to explore actions which may lead to better outcome and better rewards.To solve this exploration-exploitation trade off, we used variable named ϵ-greedy that is decaying overtime which allow us to tell our model to explore more at the beginning and less at the end.

Algorithm 1 epsilon-greedy

$\epsilon \leftarrow 0.99$
decay $\leftarrow 0.001$
for `step t ∈ {1,2,.....,T}` **do**
  $a \leftarrow$ randomly generated value between [0,1]
  if $a > \epsilon$ **then**
    $a_t \leftarrow$ randomly generated action to explore
  else
    $a_t \leftarrow \pi_\theta(a_t|s_t)$
    $epsilon \leftarrow epsilon$ - decay
  end if
end for

Deep Q-learning pseudo code

Algorithm 2 Actor critic

Input The learning rates, λ_1 and λ_2 for the two models, T: number of episodes

procedure DEEP Q-LEARNING
Initialize neural networks wights parameters θ and ω
  for `episode t ∈ {1,2,.....,T}` **do**
    $done \leftarrow$False
  $D \leftarrow []$
    while not done **do**
      $action \leftarrow$epsilon-greedy()
      $newState, reward, done \leftarrow takeStep(action)$
      D.append($state, action, reward, newState, done$)
      calculate targets for every state in D
      model.fit(modelResults, targets)
      state = newState
    end while
    Copy online model weights into target model weights
  end for
end procedure

Experimentation and analysis

In this section we will be testing the deep Q-learning on the lunar lander openai environment.

Agent

The trained agent hyperparameters is as follows:

- Online and target model:
  - Fully connected network
  - 1 Input layer
  - 3 Dense hidden layers (32*32)
  - 1 output layer
  - learning rate = 0.0005
- $\epsilon = 1$
- $\epsilon - decay = 0.0005$ until $\epsilon = 0.1$
- $\gamma = 0.8$
- experience replay size = 3000
- sample from experience replay size = 100

Analysing the results

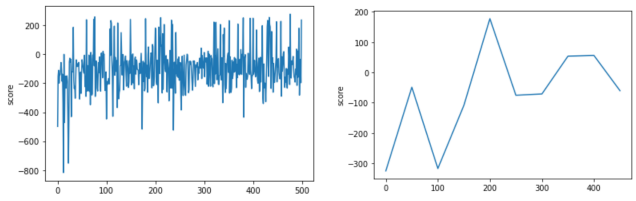

Figure 3: agent score throughout the time

The figure 3 .represents the training process of our agent and how it improves with time. Due to time management reason and since the model takes a long time to train, I could not finish until reaching the objective (avg 200 score in 100 consecutive episodes)in time.

- $[0, 100]$: At the beginning, the agent is starting to train with randomly initialized model weights The agent is favouring exploration over exploitation thus the bad scores. Even though the scores are bad, we still can see a slight improvement.
- $[100, 600]$: In the middle of training, we can see that the agent's decisions become more stable and improves highly with time as the scores are getting higher. But still the agent's policy is not optimal yet.
- $[600, end]$: At the end, the agent learns how to interact with the environment as he stops to loose and knows how to win.

Different learning rates and their effect

** I had some technical issues related to this project and I lost the data, I didn't have time to recalculate the values so that I did a look alike plot using dummy values **

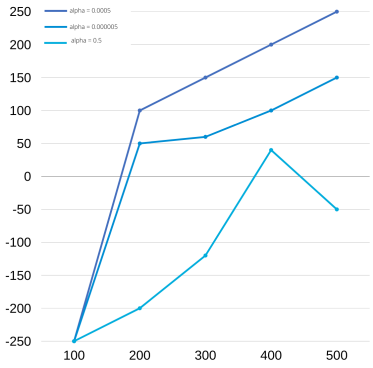

Figure 4: Different learning rates and their effects on the training

The learning rate α represents the step by which our agent is training. A we can see in figure 3 the learning rate is a very important for training:

- A big learning rate such as $\alpha = 0.5$ results in an agent that fails overtime. This can be explained by the fact that the bigger the steps are, the bigger the update of the weights are. This way the model can never find the global minimum of the loss thus a failing model.
- With a very small learning rate such as $\alpha = 0.0000005$, the agent will eventually find the global minimum thus a successful agent but the time it takes in order to get this result is very high.
- Thanks to a medium learning rate $\alpha = 0.0005$, the agent can both find the optimal policy while being faster than any other learning rate studied.

Different Gammas and their effect

** I didn't have time (due to bad time management and since the model takes a lots of time to train), to try the model and give real results. Instead these are the plots of how it should look like. **

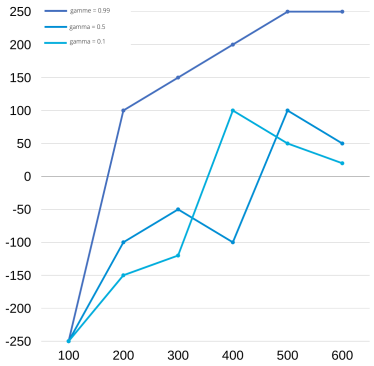

Figure 5: Different gammas and their effects on the training

The γ here is responsible for making the model short sighted or long sighted. The bigger γ is, the higher we take into consideration long time reward and vice versa. The figure5indicates that the agent performs best while γ is high. The agent works very well if it takes the future reward into high consideration thus impacting the decision of the action.

Different sizes in the experience replay sample and their effect

The experience replay sample is the sample from the historical data of the agent we use to retrain the online model at each time step. As the figure **** suggests, the higher the experience replay sample is, the optimal the model is and the higher time it takes to reach that value.

Conclusion

In this paper, not only I described the lunar lander environment and shaped it into a Markov decision process but also

I did a walk-through on how I help him to attain his objective with some analysis of the results. Thanks to deep Q-learning, we could solve the Q-learning problems but here also we face some problems: we have to wait until the end of the episode to train the model. And due to this problem we may be lead to wrong conclusions for specific actions in specific states.
example: Let's assume we got at the end of the episode a high reward $R(t)$, the agent will conclude that all of the actions he took were good but that's not the case as showed in the Figure 6 .

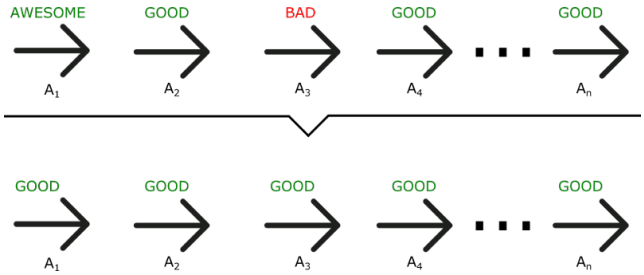

Figure 6: policy gradient problem within an episode

Seeing these limitations and inspired from Deep Q-learning, other methods can be more helpful in this case such as: Actor-critic,..