Project 2: Lunar Lander

Latest hash commit: 27162a21c3cac374463e1278e82623dcbbc41691

Author: Poujon Adrien

I. Introduction to the Lander problem

In this report I will present my results on solving the Lunar-Lander-V2 gym environment with an agent training using Reinforcement Learning. The Lunar-Lander-V2 environment is a 2D environment in which the aim is to land a ship on a given flat area delimited by two flags, centered around coordinates (0,0). For that purpose, the agent can act in four different ways: it can do nothing, fire the left orientation engine, fire the right orientation engine, and finally fire the main (down) engine. Note that fuel is infinite so that there is no gestion of powering. The agent also has access to an observation vector of the environment it evolves in, which consists of the following vector of length 8:

$$state = \begin{bmatrix} x \\ x \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ leg_1 contact(Bool) \\ leg_2 contact(Bool) \end{bmatrix}$$

A reward system is also implemented within the lunar lander V2. This system rewards about 100..140 points when the lander lands on the landing pad and has zero speed. Points can also be gained by making the legs of the lander touch the ground (+10). Otherwise, if the lander goes afar from the landing pad, it loses points. Points losses also happen when firing the engines, representing a -0.03 loss for the side's engines, and -0.3 for the main engine. Note that if the lander crashes it loses 100 points, whereas if he lands, even outside the landing pad, he gains 100 points. Consequently, the environment is considered as solved by the lander when there is a total reward greater or equal to 200 points on an episode.

II. Q-learning : explanations and limits

For solving this environment, I use a reinforcement learning based agent. The problem thus takes the classical form of reinforcement learning, involving an agent that acts in an environment and which can then observe this environment and get rewards according to his actions. It is hoped that this cycling of action, evolution, reward, and observation will lead the agent to learn how to act in ways that maximize -or at least improve- its rewards through time.

Classically, such learning procedure is implemented via Q-learning. The Q-learning procedure is described by the following update equation:

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha * (R_{t+1} + \gamma * \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$

In this procedure, we update a Q function which maps a state (S) and a given action (A) to a given value, which can be seen as the expected reward for taking action A in state S. The issue with this method in our case is that the Q function is usually conceived as a table containing the expected results for all actions in every state of the environment. This is manageable in configurations with few states, but in the case of the lunar lander, the 2D-space is a grid of 600x400 which are 240000 space states. And for each of these space states (x,y), one must consider every possible configurations of the six other parameters in the observation vector. Concretely, this means that this approach is impractical.

One solution to this problem could be tile coding as exemplified in chapter 9 of Sutton's Book [2]. In tile coding, you discretize the environment in few bins. There is no need for a linear tiling of the 2D space, so one could, for instance, increase the resolution of the tile grid above the landing pad and at the center of the 2D-space, while using only a few different tiles for parts of the space that are away from the center and from the landing pad. Note that this solution

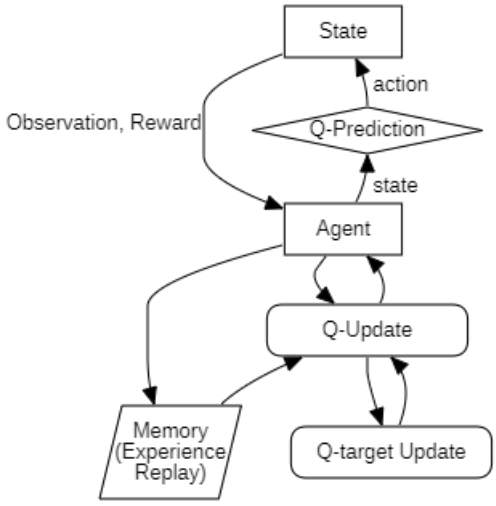


Fig. 1. Principle for DQN, Q is a Neural Network

would probably require conjectures and refutations regarding the tiling, which is probably difficult to do and might take at lot of time and efforts.

As the Q-table approach seems to be difficult, we need to find alternatives for defining the Q function. For that purpose, one thinks of course of function approximation. Here again, several choices must be made. The linear approximation appears to be doomed to failure because the Lunar environment is most likely not linear, and the behavior of the ship probably depends on the combination of multiple inputs. We could however overcome this issue by expanding the observation vector with computed combinations of the original values in the observation vector (so we would have quadratic and polynomial, non-linear terms relatively to the original observation vector, and these terms would be used as virtual observations to approximate a linear function). This would probably work but, as well as in the case of tiles coding, would require a lot of conjectures and refutations to choose the right combinations of the elements of the original vector.

III. Understanding Deep-Q Network (DQN)

From all the exposed reasons above, we must turn to non-linear approximations of the Q-function. These can be done quite efficiently by neural networks. Two approaches are possible here for the non-linear approximation. Either we choose to take two arguments as inputs, the state and the action, and we predict an associated Q(S,A), or we choose to consider only one input, the state, and we predict the outputs for all actions $Q = [Q(S, a0), Q(S, a1), Q(S, a2), Q(S, a3)]$. In what follow I choose the second alternative.

As for me, it was the first time I had to implement a neural network for function approximation, and I faced some difficulties in choosing the right way to do it. In the final work, I use the tensorflow library with a neural network composed of four layers of which two are hidden layers. The global structure of the neural network providing the best results is reproduced in Fig.2. The resulting global procedure has been used in Mnih [1][3], and is called DQN-learning. The Fig.1 shows the schematic principle: the agent makes an action based on the maximum expectation from the outputs of the neural network, the environment evolves in response to the action taken and the agent is thus in a new state, having new observations, and receives a specific reward. Combining these information, the agent can update the structure of his decision procedure by fitting the neural network to the previous set of action/reward.

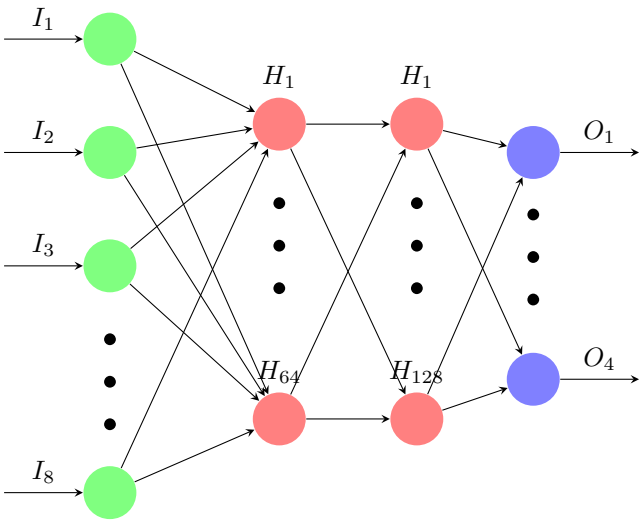


Fig. 2. Neural Network Architecture

My neural network consists of a dimension-8 input layer, followed by two hidden layers of respective sizes 64 and 128. The last layer is the output layer of dimension-4. I choosed the Adam optimizer which is a stochastic gradient descent method based on adaptive moment estimation. This optimizer is widely used in deep learning models and is useful as it allows not to converge to a local minimum to fast. I also use a simple mean squared error function.

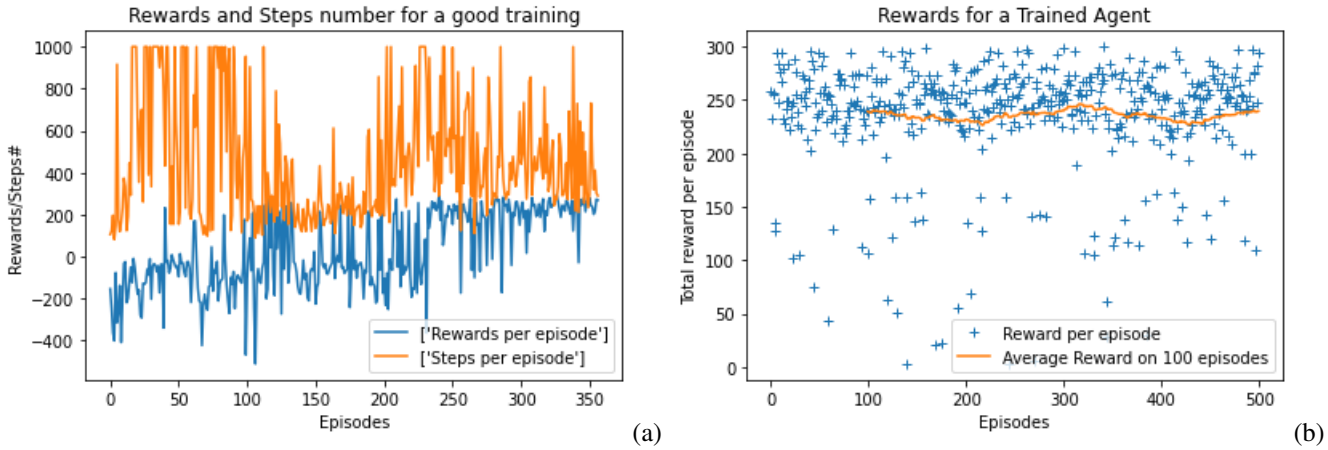
The process used for solving this environment is thus quite simple. We start by creating the neural network, then for each state we end up in, we feed it to the neural network and look at the provided outputs of which we retain the maximum to realize the associated action. While an action is taken, the agent receives a reward and the observation vector. We can use then use the previous state we were in and the rewards we gained to make supervised learning for our neural network. This procedure, associated to the use of a target neural network (see section IV), is detailed in Algo.1.

In this environment, the presence of wind, modelized by an INITIAL RANDOM parameter, enables a good exploration of the states. But, for further increase of this exploration I use an epsilon greedy policy to decide which action to take when a Q prediction is provided. As I want my agent to train rightly, it is however required that this randomness introduced by the epsilon greedy ceases to be predominant after some episodes. For that purpose, the epsilon greedy policy is coupled with an exponential epsilon decay that allows epsilon to reach low values after a given number of steps and thus reduces the randomness in the agent action decision process.

IV. Improving DQN stability

As will be shown in the next parts of the report, the learning of the agent is quite unstable. For trying to improve the stability of the learning, I thus implement two supplementary methods. These two methods will in fact ensure that the experiences we provide to the neural network are independent and identically distributed. This two additions are inspired by the Mnih [1][3].

The first of these additional features of the learning algorithm is the experience replay from a replay buffer. As shown in Fig.1, this consists of adding a memory to the agent that will allow it to remember old experiences. In the algorithm, only a random batch of defined size from the past experiences is selected for a given update of the neural network. This is though to reduce the bias in the state sampling and will help to decoupling experiences. With this addition, we can



(a) (b)

Fig. 3. Training (a) and testing (b) for a good set of hyperparameters

Algorithm 1 Neural Net Update Algorithm

$E \leftarrow$ batch of past experiences ▷ E is filled with quadruplet : (S,A,R,S')
$predictionQ \leftarrow$ prediction on all S from E
$predictionQ\hat{} \leftarrow$ prediction on all S' from E
initialize $input, output$ as tables of zeros
initialize $i = 0$
for each experience e in E **do**
▷ e is a given quadruplet : (S,A,R,S')
  $P = predictionQ[i]$
  if S' is the end of episode **then**
    $P[A] \leftarrow R$
  else
    $P[A] \leftarrow R + \gamma * argmax(predictionQ\hat{}[i])$
  end if
  $input[i] \leftarrow S$
  $output[i] \leftarrow P[A]$
  $i \leftarrow i + 1$
end for
Supervise learning for Q with $input/output$
if Number of steps is a multiple of N **then**
  $\hat{Q} \leftarrow Q$
end if

thus pretend to have independency of the experiences which is prevalent for the good functioning of a neural network. When the replay buffer is full, we start to overwrite the oldest values with the new acquired ones. This means that our agent will progressively forgot about experiences that happened "far" in the past, which are though not to be very good experiences as they were memories from when the agent was not trained. Note that however we remember the experiences, we are still producing a learning that is model free : there is no transition or reward function that is taken into account, we are not modeling the environment but simply remembering past experiences.

The second addition we can make to our algorithm, is the use of a target neural network. This was not implemented in this original Atari paper [1], but was used in later's one of 2015 [3]. The principle of adding a target network is that we will have two neural networks that will evolve simultaneously but at different paces. The original target network will be update at each step based on previous experiences and relatively to predictions from the target network, whereas the target network is updated only every N steps by copying the weights in the original network, and fixed the rest of the time. So, the target network is used for training the original network and is though to be useful as it will allows the identically distributed properties for the experiences in the training. Hence, using both experience replay and a target network we can ensure that the experiences used to train the original neural network will be independent and identically distributed (iid), which will help for the stability of the learning.

V. Results and Analysis

A. Results

Now, taking the best obtained results with my algorithm presented in Fig.3, I can say that the agent seems to learn. On the training Fig.3 (a), the agent achieves to learn to have good scores starting episode 120. However, we see that soon after this, the agent starts to "unlearn" for another hundred episodes. It is only after the episode 240 that the agent actually starts to learn and stays at high scores for the remaining training episodes.

On this same picture Fig.3 (a), we can also look at the number of steps for each episodes during the training. What we see is that there is a first period -episodes [0,120]- during which the agent learns how to fly. This is when the number of steps reaches 1000, which is the step limit for the Lunar Lander environment. After that period of learning how to fly, the number of steps quickly decreases with the increase

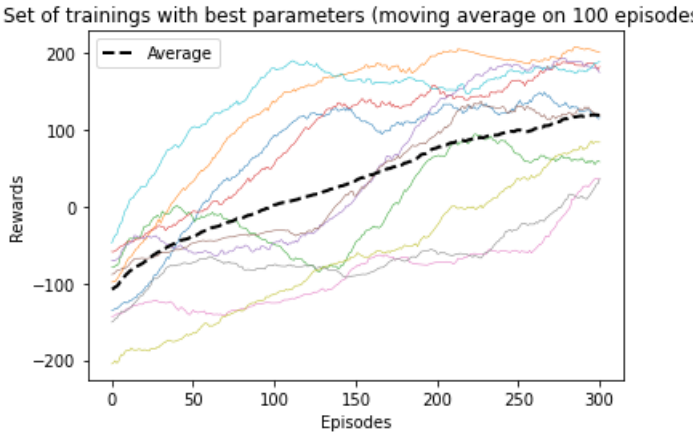
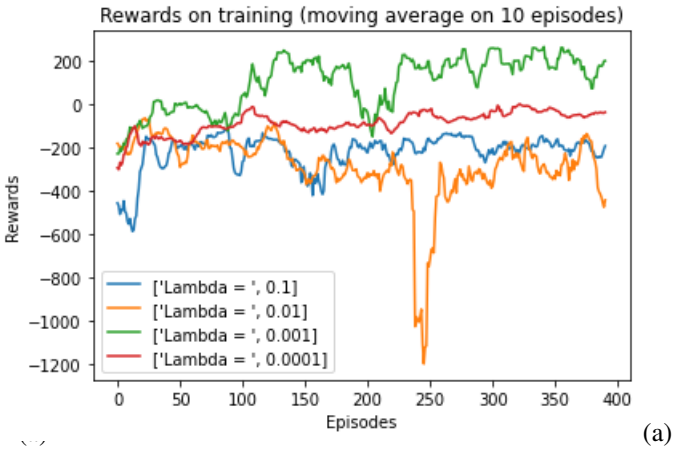


Fig. 4. Sutton's 88 Figure 3 (a) reproduction (b)



(a)

Fig. 5. Training for different learning rates

of reward per episodes -episodes [120-140]. When the agent seems to forget about what it has learned -episodes [140,200], the number of steps is very low, and we can think that it has typically learned to let himself fall without using much of the engines; This is corroborated by the reward per episode falling down to -100, which is around what we get when always taking no action at all. Thereafter, the agent restarts learning how to fly -episodes [200-260]- and then how to land properly [260-360]. In this last period, we see that the number of steps per episode also falls, but not as low as what was observed in the period when the agent forgot about what he has learned.

Finally, on graph (b) of Fig.3 we see the obtained result for the trained agent on 500 episodes. The average reward is a moving average on 100 episodes, and shows that we are way above 200 rewards per episode -the actual value is 245. It must however be underlined that even if the average reward is greater than 200, there still are some episodes for which the agent does not correctly land on the pad. These episodes when the agent fails to land in the adequate area might be reduced in their number with an eventually longer training session or with a further fitting of the hyperparameters. I unfortunately did not manage to produce better outcomes.

On Fig.4, I display a set of trainings for the best parameters. For reasons of time limitations, I stopped each training after 400 episodes, and for readability I use a moving average over 100 values. We see that there is huge differences between the trainings, some of them having attained the goal of reward +200 over 100 episodes before the training episode 200, while others being at an average reward of a few dozens at the end of the training. However, when plotting the average of all training rewards, we see that there is a global tendency of increasing rewards, traducing that the agent is actually globally learning in all cases.

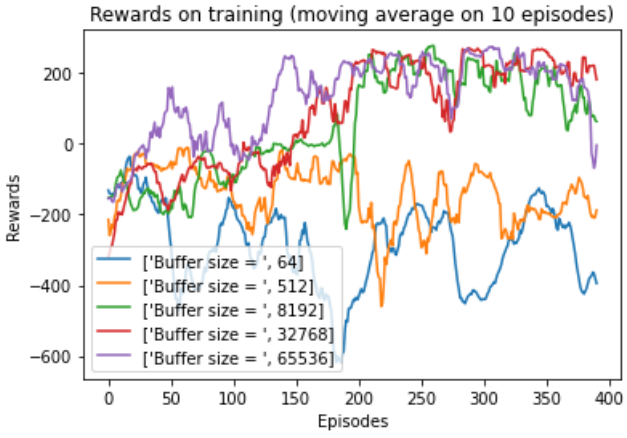
B. Analysis : learning rate

For better understanding of this learning procedure, I decided to do variations of some of the used hyperparameters. I focus on the study of three hyperparameters which are the learning rate used for updating the neural network, the size of the memory, which I call stack of past experiences, and finally the size of the two hidden layers in the neural networks.

The first parameter, the learning rate, is very important as it serves the purpose of making the neural network approximate a good Q-function. Theoretically, the larger the learning rate, the faster the learning, but also the most likely not good convergence. Conversely, for small learning rate, we might learn more slowly and stays in local minima.

What is seen on Fig.5 is that very high values for the learning rate (0.1 and 0.01) do not offer good training of the agent. With these values, it can happen that the agent starts learning something, but it will quickly forget about it and the learning curves are unstable and stays in negative rewards.

C. Analysis : memory size



(a)

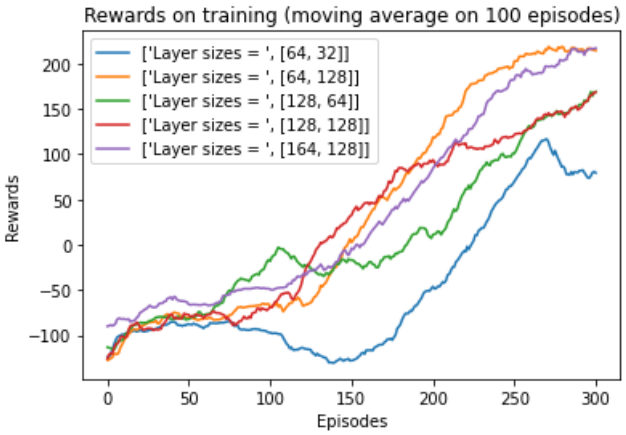
Fig. 6. Training for different sizes of buffer replay

The second parameter, the size of memory, is crucial as it represents the size of the pool an agent will learn from. As was explained in section IV on improving the DQN stability, the buffer replay contains past experiences and these are overwritten by new experiences when the buffer is full.

Consequently, taking a very small replay buffer leads to a learning that will be very close to no memory of the past at all. On the contrary, a large buffer replay will potentially remember all the past experiences of the agent since the very beginning of the training.

On Fig.6, we see that a very small buffer replay leads to a very bad learning. This perfectly illustrates the importance of the buffer replay and how a large pool of past experiences is important for independancy of the training data for the neural network. What is also seen is, that for large buffer replay, the larger the buffer, the earlier the acquisition of good results, and the steadier the learning curve. From these observations we can deduce that remembering even the experiences from the times when the agent was not well trained matters very much in how good the agent will learn. For a human agent, this might translates as follow : remembering all the times you fell off your bike is as important to your cycling skills as remembering only the last few times you were able to ride your bike perfectly.

D. Analysis : Hidden layers size



(a)

Fig. 7. Training for different sizes of layers

The last parameter I wanted to study is the configuration of the neural network in terms of the size of the hidden layers. There are two hidden layers in my neural network. Intuitively, if there are more nodes in a neural network, it will be more accurate in its predictions as it can combines the inputs in more complex ways. But at the same time, multiplying the number of nodes must lead to an increase in the number of steps and episodes required to learn as there are more weights to update. If, contrarily, the neural network is smaller, there will be less possible combinations between the inputs, and it is possible that the network will not be able to correctly estimate the function. I also wanted to know if starting with a large hidden layer and following with a small hidden layer gives the same result as doing conversely, starting with a small hidden layer and following it with a large one.

As anticipated, the smaller network of hidden layers of sizes [64,32] does not produce a very good learning and, although it reaches positive rewards, stays well under +200. We also see, that increasing the sizes of the layers results in best learning performances but we do not see an actual decrease of the learning kinetic with the increasing size of the layers. However, it seems that the architecture consisting of a small layer followed by a large one gives the best results of all.

VI. Remarks on implementations

During the coding of my program, I was confronted with some questionings, mainly about how to implement the neural network. In particular, I wondered about the necessity of normalization of the inputs of the neural network. By printing the input I could see that some of the values were very high whereas others were not. Normalization is not implemented in the provided script, and is not used in obtaining the graphs in this report, but I did an implementation of the normalization of the inputs and it did not reveal any different behaviors for the few training I was able to launch.

In the Letter from 2015 [3], it is said that stability can be further improved by clipping the error between -1 and +1. I tried to implement this but the obtained results were worse than those presented in this report, so this approach was abandoned thereafter.

In the presented algorithm, the memory starts empty. I have tried to fill the memory with many experiences by starting with launching a given number episodes with actions taken at random. Far from improving the results, it sometimes leads to absence of learning, possibly due to the low relevance of these experiences.

VII. Conclusion and possible further work

I thus have achieved to develop a learning algorithm that is good enough for training an agent to land properly in the Lunar Lander environment and get a reward greater than 200 on average for 100+ episodes. This learning algorithm is also consistent in the sense that it always ends up by making the agent learn. However, we have seen that the kinetic of the training fluctuates from a training to an other. With more time, experiments on the hyperparameters could have been refined on narrower ranges of values and coupled with statistical analysis of the results. But this would require a lot of time because each training is equivalent to several dozens of minutes and for meaninful statistical analysis we shall require at least 30 iterations for each parameters -allowing to assimilate the results with normal distributions.

I would have like to try implementing an agent using tile coding and compare its performances to DQN-learning. Finally an other possibility would be to implement what is called DDQN (and further DDDQN) which consist in decoupling the estimations of the expected values and the action to take.

References

[1] V. Minih et al., Playing Atari with Deep Reinforcement Learning, 2013.
[2] Richard S.SUTTON and Andrew G. BARTO, Reinforcement Learning, An Introduction (2sc Edition), The MIT Press, 2020
[3] V. Mnih et al., Human-level control through deep reinforcement learning, Letter, Nature VOL 518, 26 february 2015.