Deep Q Learning In Lunar Lander OpenAI Gym Environment

Petros Portokalakis AM:2014030021[*a]

[a] *Technical University of Crete, Electrical and Computer Engineering Department*

e-mail:pportokalakis@gmail

ABSTRACT
In this project, within the context of Autonomous Agents course taught by M.G. Lagoudakis, a DQN network was developed to tackle the OpenAI gym version of the well known lunar lander game, inspired by the famous Atari collection.

KEYWORDS
Deep Q Networks; Deep Reinforcement Learning; Lunar lander; OpenAI gym

INTRODUCTION

Reinforcement learning (RL) is a subdomain of machine learning, dealing with how agents should behave in unknown environments. Unalike supervised or unsupervised learning, RL algorithms learn to use an often optimal policy, in order to maximize an unknown reward function which is defined by the environment. Even though the birth of RL comes from 1989 with the discovery of Q-learning (*Watkins, 1989*), most algorithms struggled when the state-action space went out of hand. An also old conception, neural networks were concieved in the late 1940's based on human brain neural plasticity. Neural networks are actually non linear function approximators. In one of the most important breakthroughs of the AI community last decade, (*Mnih et al 2015, DeepMind*) combined Q-learning with neural networks to tackle the problem of inefficiently large Q-tables required by Q-learning for problems with large state action space. Rather than using algorithms like value iteration to compute Q-values, DQN uses neural networks to approximate the optimal Q-function. The problem to be examined will be lunar lander as seen in the picture below.

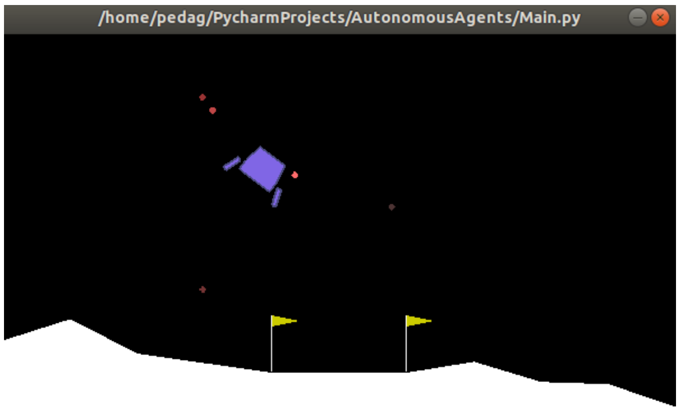


Figure 1. Lunar lander in mid-flight. The agent has four possible actions in each state in order to land the vechicle smoothly between the two yellow flags. Landing beyond the two flags is possible but with a reduced reward

1 LUNAR LANDER

1.1 *State and action space*

The "LunarLander-v2" environment provides a total of 8 observations in each state, hence the state space vector is 8 dimensional. The state vector elements are as follows: 1) x-axis position 2) y-axis position 3) x-axis velocity 4) y-axis velocity 5) angular position 6) angular velocity 7) left landing pad touching ground and 8) right landing pad touching the ground. The last two (7 and 8) are boolean variables. The action space is 4 dimensional and is as follows: 1) fire left engine 2) fire right engine 3) fire main engine 4) do nothing. It is useful to note that the landing pad is always at coordinates (0,0).

1.2 *Rewards*

The episode finishes if the lander crashes or lands successfully resulting to a reward of -100 or +100 respectively. Each leg ground contact is +10, and firing main engine is -0.3 per frame. The problem is considered solved if the agents achieves a reward of +200 over a batch of consecutive episodes. In this work, the considered batch is set to 100 consecutive episodes.

2 BACKROUND

2.1 *Reinforcement Learning*

The general RL formalism if used in this work where an agent interacts with the environment. The environment is fully observable for the sake of simplicity. An environment is described by a set of states S, a set of actions A, a distribution of initial states $\Pr(s_0)$, a reward function $R : S \times A \to R$ transition probabilities $\Pr(s_{t+1}|s_t, a_t)$ and a discount factor γ.

A deterministic policy is a mapping from states to actions $\pi : S \to A$. Every episode starts by sampling the initial state s_0. At every frame (or step) of the environment, the agent executes an action $a_t = \pi(s_t)$, which in turn results in a reward r_t provided from the environment. The agent's goal is to maximize its expected return $\mathbf{E}_{\mathbf{s_0}}[R_0|s_0]$, where R is defined as $R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$ which is a discounted sum of future rewards. The Q-function is defined as $Q^{\pi}(s_t, a_t) = \mathbf{E}[R_t|s_t, a_t]$. An optimal policy π^* is any policy s.t $Q^{\pi^*}(s, a) \geq Q^{\pi}(s, a)$. The optimal policy is obtained via the *Bellman* equation:

$$Q^*(s, a) = \mathbf{E}_{s' \sim p(\cdot|s,a)}[r(s, a) + \gamma max_{a' \in A} Q^*(s', a')]$$

2.2 *Deep Q-Networks*

Deep Q-Networks (DQN)(Mnih et al. 2015) is a model-free RL algorithm for discrete action spaces. In general, a neural network Q is maintained in order to approximate Q^*. An obvious policy to choose, is the ϵ-greedy $\pi_Q(s) = argmax_{\alpha \in A} Q(s, a)$. An ϵ-greedy policy is a policy where with probability ϵ the agent executes a random action sampled from A, and consequently with probability $1 - \epsilon$ takes the action $\pi_Q(s)$. In general, the input layer of the Deep Q-Network has as many inputs as the dimensionality of the the state vector $|S|$, followed by some number of hidden layers, and the output layer has as many outputs as the agent's possible actions. The output layer computes the $|A|$ Q-values of the available actions.

When the agent is training, episodes are generated using the ϵ-greedy policy mentioned above, using the current approximation of the action-value function Q. After every environment transition, the tuple $(s_t, a_t, r_t, s_{t+1}, done)$ is stored in the replay buffer. The replay buffer is a data set of the agent's experiences at each time step. The main reason the agent is not trained with the sequential events of the its experience is to break the correlation between consecutive samples. Also, the generation of new episodes is interleaved with neural network training. The network is trained based on the loss function $L = \mathbf{E}(\mathbf{Q}(\mathbf{s_t}, \mathbf{a_t}) - \mathbf{y_t})^2$ where $y_t = r_t + \gamma max_{a' \in A} Q(s_{t+1}, a)$ and the 5-tuples mentioned above are sampled uniformly from the replay buffer. The loss function mentioned is the key of *Deep Q-Networks*. It incorporates in the training of the neural network, the well known Bellman equation discussed back at section *2.1*.

Also, another teqnique is used to further improve the quality of the algorithm. A seperate neural network called the target network is created. The target network updates its weights slower than the main network. The use of the target network is to make the optimization procedure more stable.

Algorithm 1 Deep Q Learning with replay buffer and target network

```
1:  Initialize replay buffer capacity
2:  Initialize the policy network with random weights
3:  Clone the policy network, and call it the target network
4:  for each episode do
5:      Initialize the starting state
6:      for each time step do:
7:          Select an action via exploration or exploitation
8:          Execute action
9:          Observe reward and next state
10:         Store experience in replay buffer
11:         Sample random batch from replay buffer
12:         Pass batch of states to policy network
13:         Calculate loss between output Q values and target Q values (now pass batch to the target network)
14:         Gradient descent updates the weights to minimize loss
15:         After C time steps, copy policy network weights to the target network
16:     end for
17: end for
```

2.3 *OpenAI gym*

By using the OpenAI gym environments, there is no need to code an RL environment from scratch, or preprocess any data, as the gym framework provides all information exchange needed through its API. Most information is exchanged by the following line of code.

$$observation, reward, done, info = env.step(action)$$

The function $.step(action)$ is called to advance the progress of the environment by one step, given the fact that the agent executes $action$. Then, the next state vector, the reward for that action, and a boolean variable $done$ indicating if it is time to reset the environment are returned, along with $info$, a python dictionary useful for debugging.

IMPLEMENTATION DETAILS

Both target network and replay buffer functionalities are implemented. **Table 1** below describes all hyperparameters chosen for the specific problem.

Hyperparameter	Value
Starting ϵ	1.0
Minimum ϵ	0.01
Decay factor of ϵ	0.99
Discount factor γ	0.99
Learning rate α	0.001
Batch Size	64
Replay Buffer	1000000

Table 1. Hyperparameter Values

The decay factor of ϵ is used after each episode the following update takes place:

$$\epsilon \leftarrow \epsilon \cdot 0.99$$

The replay buffer is implemented with a *deque*, a high performance container datatype provided by python. Deques are list-like containers with fast appends and pops on either end.

The neural network is implemented with tensorflow. The input layer has eight input nodes (dimensionality of state space), followed by the first hidden layer consisting of a fully connected layer of 64 nodes, which is followed by the second fully connected hidden layer of 128 nodes. The output has four output nodes, the argmax of which indicates the predicted action. The output layer can be though of as a one hot encoding layer. More fully connected layers were tested but they dropped the performance and they were hence removed.

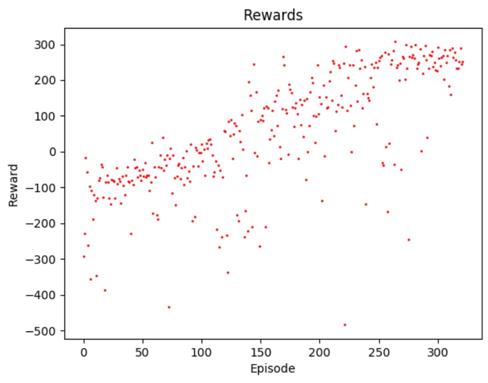
There are two ways for the training to end. Firstly, if the agent achieves the expected behaviour. In the lunar lander setting, the agent is considered well trained if it can achieve a mean reward of 200 over 100 consecutive episodes. Secondly, there exists a flag as explained bellow, indicating the maximum number of episodes available for the agent to train. The training is then forced to end, breaking the training loop. After the end of training, performance graphs are plotted using package *matplotlib*.

Bellow are some other administrative flag variables created to easily alter the agent's behaviour.

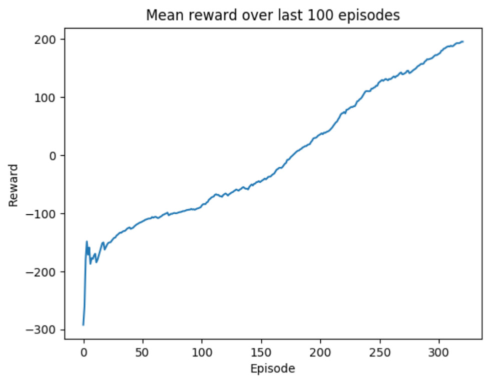
- `USE_TARGET_NETWORK` (boolean): Whether to use or not the second neural network introduced
- `UPDATE_TARGET_NETWORK` (integer): Amount of episodes before updating target neural network with the weights of the main
- `STOP_AFTER_N_EPISODES` (integer): Stop training regardless of the agent's performance. This is particularly helpful to examine untested hyperparameters and network architecture
- `RENDER_EVERY` (integer): Render one in `RENDER_EVERY` episodes. A rendered episode last for approximately 3 seconds. In the training phase it is infeasible to render all episodes as the training will take a substantial amount of time.

RESULTS

In this section performance graphs are presented and commented.

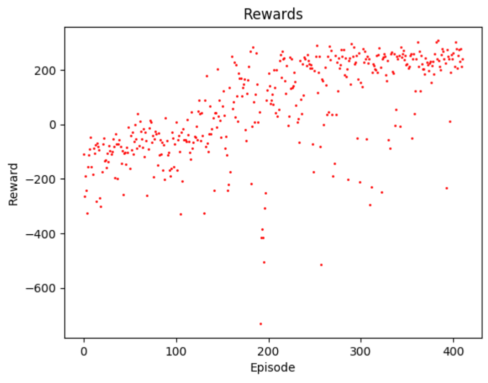


(a) 1a

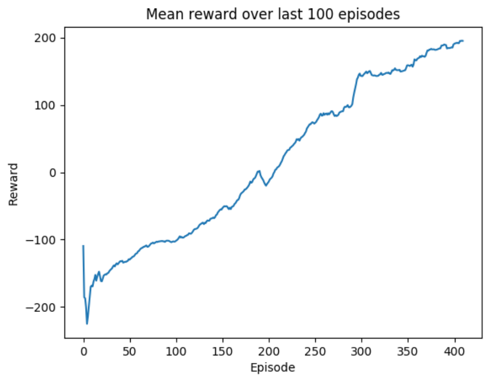


(b) 1b

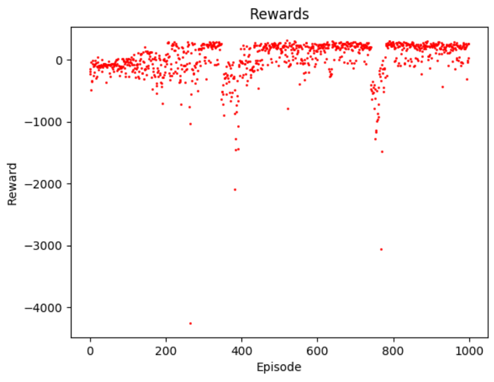
Figures (1a) and (1b) correspond to two fully connected hidden layers, 150 and 120 nodes respectively, whereas figures (2a) and (2b) correspond to two fully connected hidden layers of 64 and 128 nodes each. The last two figures (3a) and (3b) correspond to three fully connected hidden layers of 64, 128 and 128 nodes each. Obviously, the input layer has 8 nodes and the output layer has 4. By examining the six figures presented in this section, it is obvious that a simple architecture with two hidden layers is not only sufficient to train the lunar lander agent, but by adding a third hidden layer the agent is unable to get trained as seen bellow.



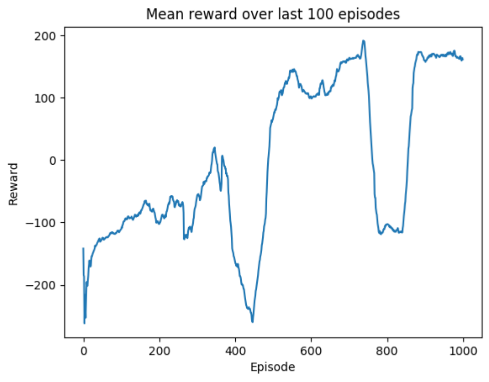
(a) 2a



(b) 2b



(a) 3a



(b) 3b

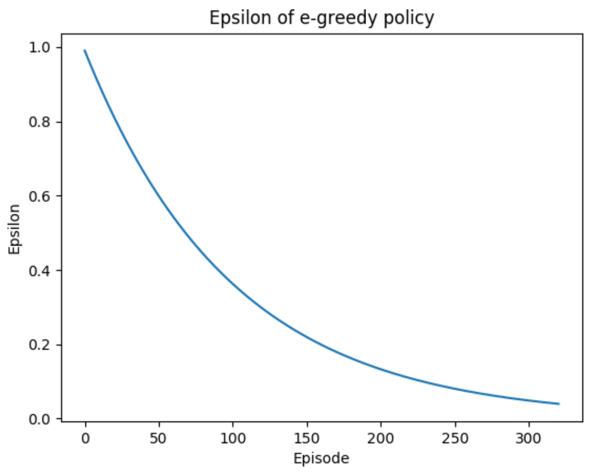


Figure 5. Epsilon decay as the episodes increase

Figure 5 shows the ϵ value decaying. The decaying of ϵ always ends after 459 episodes with a final value of $\epsilon = 0.0099$ although it stops having an effect on the agent around episode 350 where $\epsilon = 0.02$.

DISCUSSION

2.4 *Regarding Q-Learning*

The original intention was to test both Q-learning and Deep Q-Learning on the Lunar Lander game. As the state space is continuous with six of the eight state variables ranging from ∞ to $-\infty$ (as the two of the state variables are boolean and describe if the two landing pads touch the ground or not) the state space must be discretized. The original discretizations chosen raised memory errors in the workstation, indicating *MemoryError: Unable to allocate 11.4 GiB for an array with shape (25, 25, 24, 20, 20, 16, 2, 2, 4) and data type float64*. When the discretization was loosened, the state matrix was of shape [18,18,16,16,16,16,2,2], meaning that every action sampled from the environment was discretized and put into *buckets*. Given the fact that the agent has four possible actions in each state, the size of the q-table to be generated is computed to be $size = 18 \cdot 18 \cdot 16 \cdot 16 \cdot 16 \cdot 16 \cdot 16 \cdot 2 \cdot 2 \cdot 4 \approx$ 340.000.000. It seems infeasible, but due to time limitations and the long time of training needed to populate the q-table with good enough q-values it could not be tested.

FUTURE WORK

In the near future, i aim to implement the Q-learning approach to have a more robust comparison between Q-learning and Deep Q-learning. Also, i aim to implement *hindsight experience replay*, a novel architecture which reinvents failure. In this approach, when an agent does not achieve the intended goal, it stores the sequence of actions it executed. It treats failure as a new virtual goal, as if the agent in fact wanted to achieve another goal.

REFERENCES

1. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg Demis Hassabis (2015) *Human-level control through deep reinforcement learning*
2. Watkins, C.J.C.H. (1989). *Learning from delayed rewards. PhD Thesis, University of Cambridge, England*