Project 2: Lunar Lander

Rodrigo De Luna Lara

October 29, 2018

Last commit hash: ***edf64ab01caa785a40f03f1886b90096264ab8fc***

The task for this project was to create a Reinforcement Learning agent to succesfully solve the "Lunar Lander" environment in OpenAI gym, by using any algorithm. The MDP describing this environment consists of a mostly continuous state space S with 6 continuous variables and 2 discrete variables, a discrete action space with 4 possible actions and a reward function.

The state is described with a vector:

$$\langle x, y, v_x, v_y, \theta, v_\theta, ll, rl \rangle$$

In which x and y are the coordinates of the lander, v_x and v_y the velocity components, θ the angle and v_θ the angular velocity. ll and rl correspond to the 2 discrete states, which indicate if either the left leg (ll) or the right leg (rl) is touching the ground.

The action space is defined with 4 possible actions:

Id	Description
1	Do Nothing
2	Fire left engine
3	Fire main engine
4	Fire right engine

The total reward for successfully landing is between 100 and 140 points depending on the placement of the lander on the pad, and is penalized for moving away from the pad. An episode finishes when the lander crashes or comes to rest, receiving -100 or 100 points respectively. Each leg contact with the ground grants 10 points, and firing the maing engine results in a penalty of 0.3 points each time. As per the environment description the problem is considered solved if a mean reward of 200 is achieved over 100 episodes.

The proposed method to solve this problem is Q-Learning, as it's relatively easy to implement and has good results. However, out of the box, standard Q-Learning was designed for discrete state and action spaces, and the Lunar Lander has a continuous state space. The first approach to this problem was to try to discretize the states to a number of bins, each of the continuous elements of the state vector would be assigned to a bin based on its value, and the number for the bin would be the discretized state.

This approach immediately has two problems: first, the state space is technically infinite, so the choice of bins is extremely hard if not impossible to make accurately; second, the resulting Q-table grows exponentially as more discrete states are added, in this particular case if 10 bins per continuous element of the state vector were used (for a resutling discrete state between 0 and 9), the resulting discrete state would run from 00000000 to 99999911, resulting in a ~400 million item Q-table, which would require around 2GB of space **per-table**. Even with a very coarse grid search this would make training require a staggering amount of processing power or an extremely long time (or memory errors in Python).

Having discarded standard Q-Learning, other reinforcement learning algorithms such as Deep Q-Learning came into mind, but without proper knowledge on deep learning and neural network configuration, a good result may have been achieved but without very good understanding of why (which seems a common problem in deep learning, things work in the end but they are not very interpretable). Therefore, sticking to Q-Learning required solving the problem with this environment, which is the continuous state space.

Researching on this problem, value function approximation seemed to be the best candidate solution, as it was designed specifically for problems with large state spaces in which a table representation is inadequate. The main idea behind this method is to represent the Q-table as a parametrized function, in which the number of parameters should be small compared to the number of states. The parameters for the function are learned from experience, and the updates based on observations of one state will update similar states.

The approximation used in this analysis has two steps, first a Scaler and Radial Basis Function samplers are used to transform the features in the state space, this transformation is defined with multiple samples of states. Then a Stochastic Gradient Descent regression with constant learning rate is used to create a model per each action (akin a Q-table), which is used as the value function approximation.

The first issue with this in the Lunar Lander environment was the sampling of the state space. Several gym environments allow sampling of the state space by calling a sample method. However, the bounds for the state space in this particular case are $[-\infty, \infty]$, so it's not possible to directly sample it. As a workaround for this, the environment was run with random actions for several episodes in an attempt to get a good representation of the state space. These samples were then used to create the feature transformation object by fitting a Standard Scaler (by removing mean and scaling to unit variance) and then concatenating the result of several Radial Basis Function approximations with different kernel shapes. The resulting object could be called directly on a state from the environment to get it's approximation.

The models in the Q-Learner were defined with Stochastic Gradient Descent regressors from scikit-learn with constant learning rate. Each step in an episode the partial fit method was called on the models to update the functions representing the Q-tables. An ϵ greedy policy is used to determine the action that the agent will take at each step. The value for ϵ decays each episode by a constant factor. The discount factor is used on the estimation from the approximation models along the reward to update the models at each iteration.

Based on this implementation there are 3 hyperparameters, the discount factor γ, the exploration factor ϵ, and it's decay. Given the lengthy training times, a very coarse grid search was defined with these levels:

$$\gamma \in [0.96, 0.97, 0.98, 0.99]$$

$$\epsilon \in [0.40, 0.50, 0.60, 1.00]$$

$$\text{decay} \in [0.50, 0.75, 1.00]$$

To evaluate performance, the mean of the 100 last episodes was taken as main metric, as well as the net change in reward between the last 100 episodes and the first 100 episodes. For each combination of hyperparameters the agent was run 1000 episodes using the same state approximation function based on the same sampling of the state space. Figure 1 shows the resulting mean reward for the last 100 episodes for each hyperparameter combination.

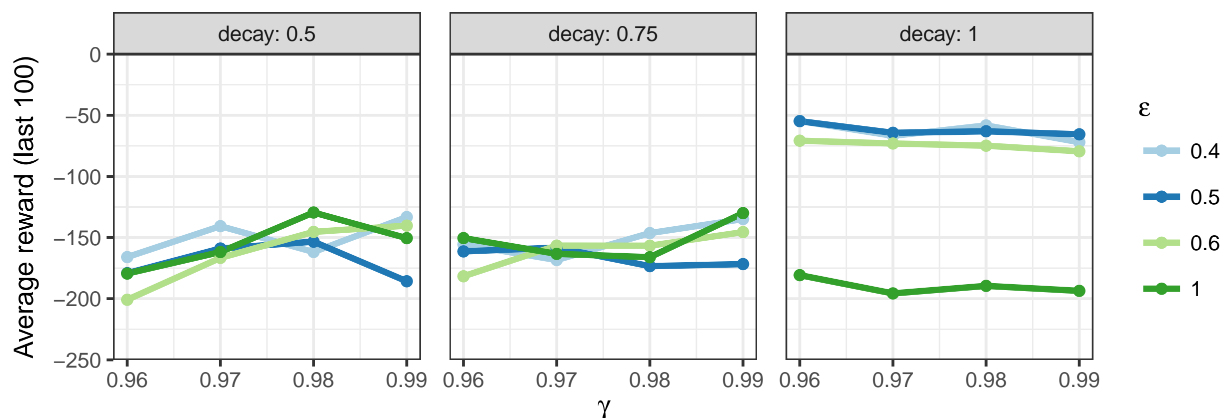

Figure 1: Hyperparameter Selection (Means)

Figure 2 shows the difference between the means of the first and last 100 episodes, as a measure of how much learning there was for each hyperparameter combination.

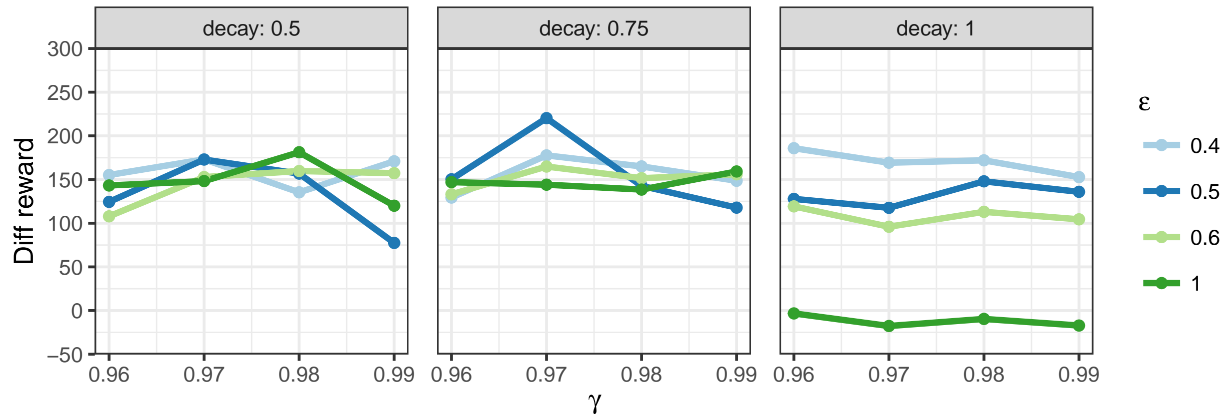

Figure 2: Hyperparameter Selection (Learning)

Based on these plots, the selected hyperparemters were $\gamma = 0.96$, $\epsilon = 0.40$ and decay $= 1.00$, given that it has the largest mean reward for the 100 last episodes and comparable learning to the other combinations. These hyperparemters were used to train the model another 4000 episodes, with the initial model approximation being the resulting from the hyperparameter selection and with the sample feature transformation sampling. Figure 3 shows the resulting episode rewards for the 4000 episodes.

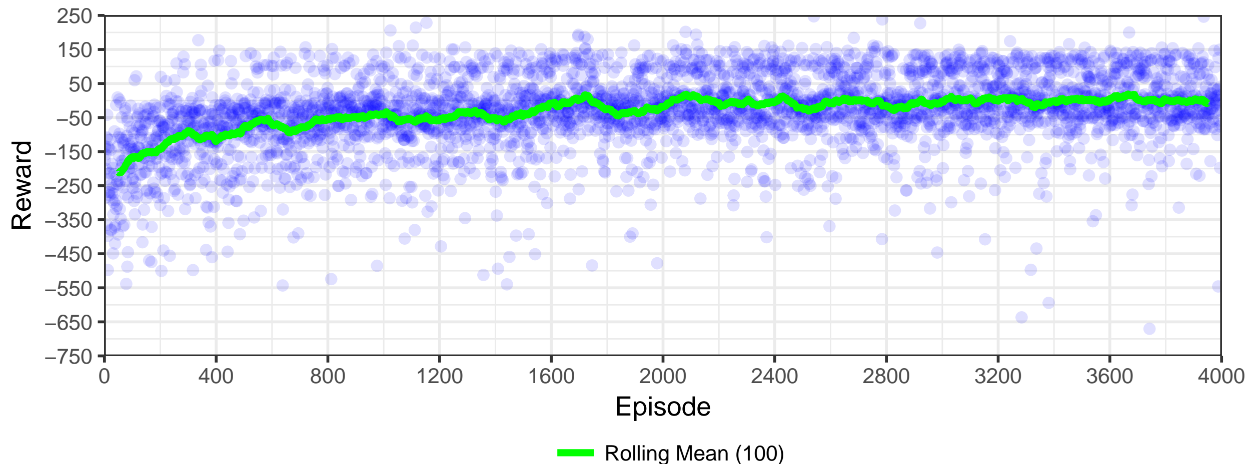

Figure 3: Episode Rewards

It can be seen that although it is extremely slow, the agent is learning, given that the initial mean reward over 100 episodes is near -250 and in the end it's around -50. Also the number of episodes with rewards between 50 and 150 is more frequent near the ending episodes. However, there is a large spread on the rewards, of about 300 points along the whole training of the learner.

Figure 4 show the distribution for the number of steps and the time per episode for this learner, as it can be seen a lot of episodes timeout at the maximum number of steps. These episodes may not be contributing to the learning that much as the agent is *lost* in them, wandering around for enough time without either crashing or landing. As per the training times, this approach doesn't seem to be computationally costly, although over several thousand episodes, training times of seconds can add up quickly.

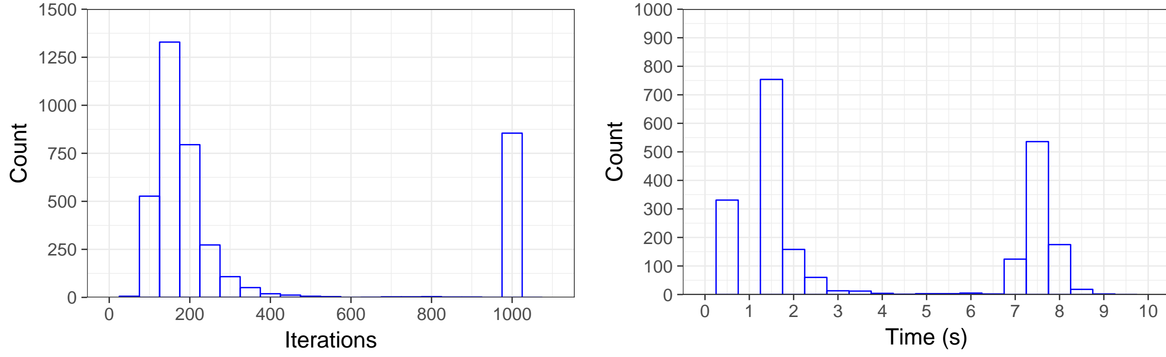

Figure 4: Steps and Times for the Learner

Finally, Figure 5 shows the results of this trainer with 100 episodes just evaluating the value approximation function models and getting the optimal action according to that. It is interesting that the results here don't match the average results during the training phase, in which the average reward was around -50. This could be indicative that the method used to evaluate the models to determine the best action is incorrect, or that the updating of the models during training doesn't reflect on the saved models file, which seem unlikely given that the best action is determined in the same way as during training, where it was clear there was learning as the mean reward tended to increase with more episodes, albeit extremely slow.

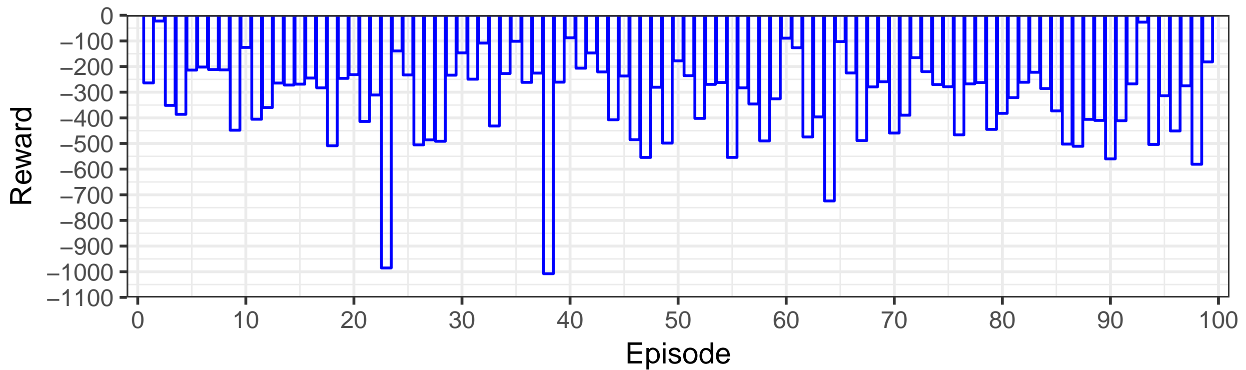

Figure 5: Episode Rewards After Training

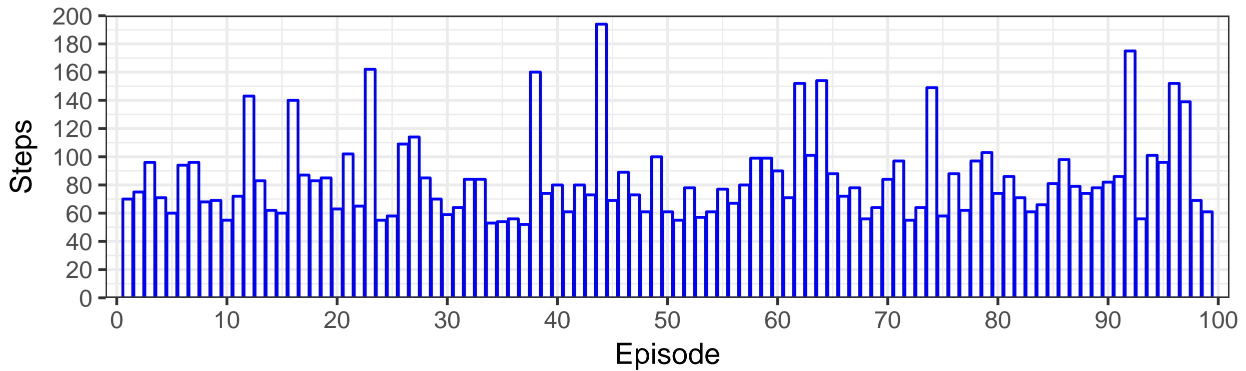

Figure 6: Episode Steps After Training

The steps shown in Figure 6 show that for all episodes the learner stops before 200 steps, but it seems to be crashing on every episode. Inspecting the generated GIFs, in some cases the lander is going up from the starting position, which is weird, and it also seems to not be doing anything for a lot of “states”. This seems to indicate an incorrect function for the value approximation, as it seems a lot of states are not being mapped to actions. As it was shown in some figures above, there is learning as the reward tends to be higher, but that could just be the lander crashing sooner instead of hovering around firing the engines losing points.

The current approach was insufficient to solve this problem, which is most likely due to the continuous state space, and the inability of the proposed method to create an appropriate value function approximation. Another possibility is that the way to evaluate the function approximation once the training is done is incorrect, or that the states visited during the 100 evaluations with the trained agent weren't visited before, due to the aforementioned restrictions.

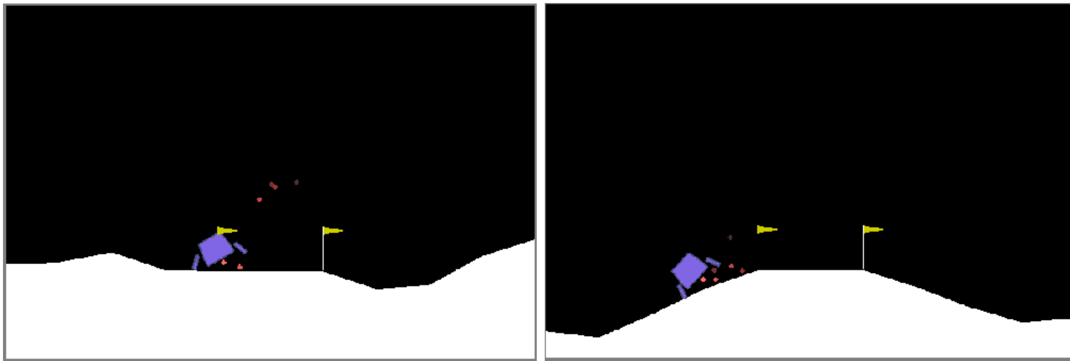

Figure 7: Example of Semi-Successful Landings

In this regard, perhaps implemententing Deep Q-Learning was not a bad idea, the Neural Network architecture should be able to better map the continuous states to a discrete action space, given how a multi-layer network is in theory able to represent any arbitrary function between the inputs and outputs. In this case sticking to the traditional Q-Learning was not enough to solve the problem. One quick browse online confirms this as most existing solutions to the Lunar Lander environment use DQN.

Figure 7 shows a couple of examples in which the landing was almost succesful, but it crashed. It can be seen that the surfaces are very different, so if the agent is not landing near the landing pad portion which is constant, then the visited state is not very useful for the long-term learning goal. The agent must then have more successful landings in the landing pad for it to learn, but the current method is too slow for this, given the complexity of the MDP.

The main issue here was the approximation of the continuous state space, and the large number of possible states, which tends to infinity given that there are 6 parameters in the state that are floating point numbers, which again is a point in favor for deep learning. Deep learning is most commonly used for problems with large inputs and complex domains (like image processing), and it tends to outperform traditional methods, being also quicker to converge. The final recommendation is then to stick to traditional Q-Learning only when the MDP is not very complex and the state space is not so large, and prefer deep learning methods like DQN in these cases.