Training an Agent to land the Lunar Lander from OpenAI Gym

SAHIL DHINGRA

CS-7642 Reinforcement Learning
Project 2
GitHub hash – *7d43ff7*

1. Introduction

In this article, we would discuss the implementation and training of a model in the lunar lander environment from the OpenAI Gym library. In this environment, our goal is to train the agent to autonomously land, with a cumulative sum of rewards of 200 or higher over 100 consecutive runs.

As an RL problem, the MDP process attributes can be summarized as follows –

- *State space*: 8 space attributes – XY space coordinates (x, y), velocity components (vx, vy), angular position & velocity (θ, vθ) and if the two lander legs are touching the ground (left-leg, right-leg)
- *Action space*: 4 actions– do nothing or fire any of the three engines
- *Rewards* – 100-140 for moving from top of the screen to the landing pad, -0.3 for firing the main engine each time, -100 for crashing and 100 for successfully landing.
- *Deterministic transitions*

2. Experimental Design

The problem can be summarized as a continuous space RL problem, where we need to find the optimal policy – represented in a 4-dimensional continuous action space – corresponding to a continuous 8-dimensional state space.

$$\boldsymbol{\pi_t}\ (\boldsymbol{s_t}) = \boldsymbol{a_t}$$

where, $\boldsymbol{s_t}$ = (x, y, vx, vy, θ, vθ, left-leg, right-leg)

$\boldsymbol{a_t} \in$ [do nothing, fire the left orientation engine, fire the main engine, fire the right orientation engine]

To find the optimal policy, DQN (Deep Q-Networks) algorithm was used. Using DQN, we follow the Q-learning approach in a continuous state space. In this approach, a deep learning model is used, which takes as input the state vector and predicts the corresponding Q-values for

each available action as output. The maximum of Q-value predictions after convergence would define the optimal policy.

While training, we train the model with *y* as target and *Q(s, a)* as input. *y* is equal to *Q(s, a)*, however the Q-values corresponding to the actions in the batch are updated with the actual observed rewards plus the maximum discounted lagging Q-values, as predicted using $Q_{targ}(s',a')$.

$$y = r + \gamma \max_{a'} Q_{targ}(s',a')$$

For Q-value prediction, a neural network was used, with 8 input nodes for the 8-dimensional state space, 1-hidden layer with 8 nodes and 1-output layer. *Rectified Linear Unit* (ReLU) was used as activation function for the input and hidden unit, while *linear* activation was used for the output layer, for predicting Q-values corresponding to each action. For stochastic gradient descent, *Adam* optimizer was used; mean squared error was used for the loss metric. Similar architecture was used for target Q-network.

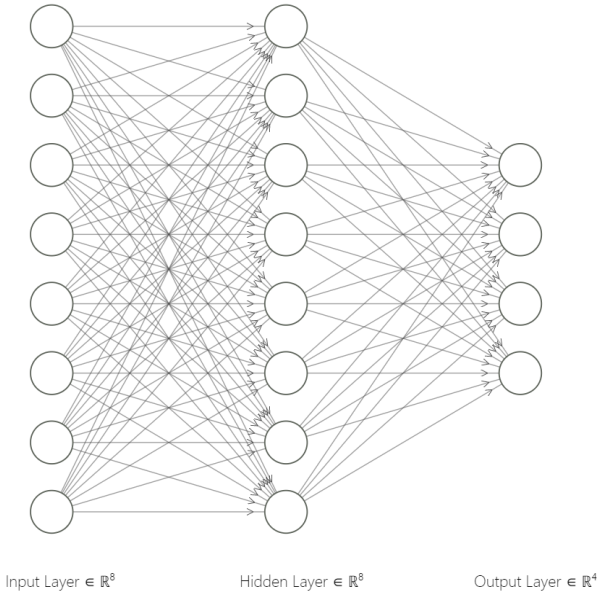

Fig. 1 A representation of the neural network used

3. Implementation

3.1. Algorithm

DQN was implemented in Python 3.7, as described in the DeepMind paper *'Playing Atari with Deep Reinforcement Learning'* and provided below.

Algorithm – Deep Q-Learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value functions Q and Q_{targ} with Adam optimizer, learning rate = lr and random weights
For episode = 1, 1000 do
    Initialize sequence
    $\varepsilon = \varepsilon_{min} + (1 - \varepsilon_{min})\, e^{-\lambda * \text{episode}}$
    for $t = 1, T$ do
        With probability ε select a random action a_t
        Else, select $a_t = \max_a Q(s_t, a)$
        Execute action $\boldsymbol{a_t}$ and observe reward r_t
        Set $s_t = s_{t+1}$ and store transition (s_t, s_{t+1}, a_t, r_t) in D
        Sample random mini-batch of transitions (s_t, s_{t+1}, a_t, r_t) from D
        Set $y = \begin{cases} r & \text{for terminal states} \\ r + \gamma \max_{a'} Q_{targ}(s', a') & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(s_j, a_j))$ for mini-batch elements with indices j
        If T%8=0, set $Q_{targ} = Q$
    end for
end for

4. Results

1. **Training rewards** – The algorithm starts converging after episode 600 with learning rate 0.0006 and epsilon decay 0.01

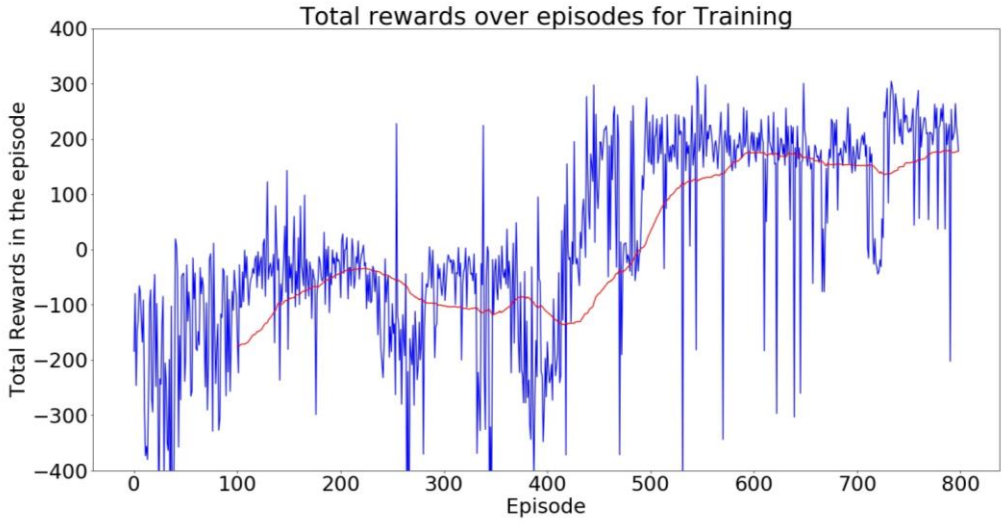

Fig. 2 Total rewards accumulated over each episode during training. The agent is able to successfully land after 600 episodes, observing 200+ rewards eventually.

2. **Test rewards** – The agent is able to score an average of higher than 200 over 100 trials. However, it does fail occasionally, possibly due to states not seen while training.

Fig. 3 Performance of the agent over 100 consecutive trial runs, with an average reward higher than 200

3. **Effect of hyperparameters** – Higher values of learning rates seem to diverge to worse scores over episodes learning a sub optimal behaviour while lower rates take too long before the exploration window has expired, i.e. the agent doesn't learn from the initial states and cannot explore the states near the launching pad well enough

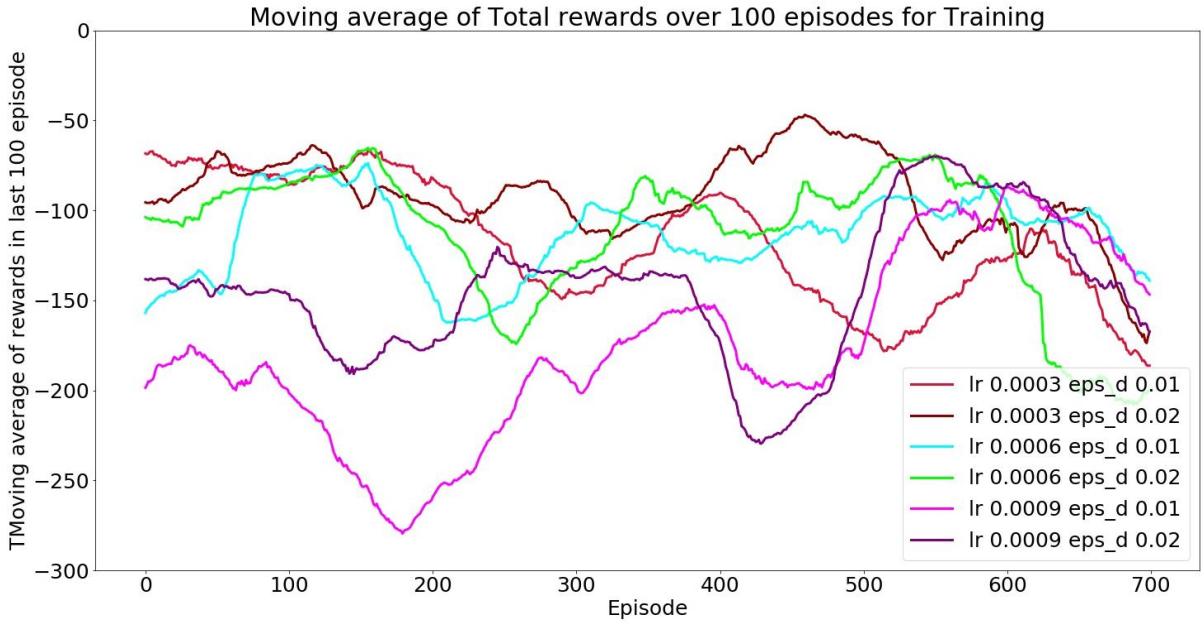

Fig. 4 The effect of hyperparameters on the rewards curve after training

5. Analysis of results

5.1. Sensitivity to hyperparameters – The algorithm is extremely sensitive to the hyperparameter values – learning rate, epsilon decay and minimum epsilon – with only a narrow range of values leading to convergence.

5.2. Number of hyperparameters – Apart from the key hyperparameters, a suitable value for the other hyperparameters – memory buffer size, batch size, Q_{target} update frequency – also needs to be determined for the particular environment, as these parameters have been observed to affect the agent's behaviour. A short memory can lead the agent to forget positive experiences while a long enough memory would lead it to keep learning from the same experiences several times over. Similarly, the target network should not change too quickly or too slowly.

5.3. Gamma (γ) – As long as the gamma is high enough, such that the agent doesn't mind taking the negative rewards in order to get higher rewards in the future, gamma did not seem as important. So, a constant value of $\gamma = 0.99$ was used while tuning the model.

5.4. Exploration - exploitation – Given the large state space, a huge amount of exploration is needed initially and we need to start improving on those experiences gradually. However, as a result of that exploration, the agent might never converge if the exploitation is not optimal. Once we have explored enough and the agent has learned from the experiences, the epsilon minimum should allow only limited random actions, so that the trained agent does not take too many random actions. Therefore, a right epsilon decay and epsilon minimum need to be found through tuning and testing.

It is also important to successfully land during early exploration, before it's too late that we cannot take random actions with high probability. This is also sometimes random, as the agent needs to encounter the nearby states during landing. It was observed with the same hyperparameters that the agent failed to converge sometimes when no early landing was observed.

5.5. Variance in results due to randomness – Due to randomness in the exploration process being dependent upon different random seeds (for action, batch sampling, environment initialization), there is a significant variation in algorithm's convergence. This variation can be drastic even with the same set of hyperparameters and the algorithm might converge in one training run and not in the other.

5.6. Effects not studied:

5.6.1. **Neural Network architecture** – The effect of neural network architecture – number of nodes, hidden layers, activation functions – was not studied. The choice of these parameters could also affect the agent's performance.

5.6.2. **Other algorithms –** Due to time constraints, other algorithms were not tested.

References

[I] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou. Daan Wierstra, Martin Riedmiller. Playing Atari with Deep Reinforcement Learning