Solving Lunar Lander Using DDQN

1. Introduction

Lunar Lander is an OpenAI environment, which requires agent to move in 8-dimensional continuous state space using 4 actions to land on pad. An average score of 200 for last 100 episodes is required for solution. In this project, DDQN (Deep Double Q-learning Network) algorithm, based on Mnih's "Human-level control through deep reinforcement learning" [1], is implemented to solve Lunar Lander.

2. What is DDQN?

DDQN is an improved algorithm over standard DQN. In standard DQN, same Q network is used to select and evaluate an action as follows:

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-).$$

Figure 1. Standard DQN target computation [1]

As shown, a small increase in Q could change policy and target Y. This non-stationarity induces an upward bias and divergence during learning [1]. To resolve this, DDQN decouples selection and evaluation into two networks. In addition, DDQN improves data efficiency by storing previous experiences in a memory buffer. At each training step, a mini-batch is randomly drawn from memory to be replayed and trained on main DDQN to reduce correlation between sequential samples.

3. DDQN Implementation

Implemented DDQN follows Mnih's description [1] closely. In brief, algorithm is implemented using python and keras. Networks are constructed in keras as 3-layer Sequential DQN. Input layer is dense with observation vector (size 8) as input. Hidden dense layers consists of 60 nodes using "relu" activation. Output is a value vector for the four actions. Loss is computed as MSE with Adam optimizer. At start of each training episode, main and target networks are initialized with same uniformly randomized weights, and an experience replay buffer (1.5M size) is created. For each step, action is selected either randomly with ε probability or deterministically by main DQN, according to:

$$\varepsilon = 0.99 - \frac{0.99 - 0.001}{N_A} * t \; ; \; t = \text{current step}$$

$$: a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$$

Figure 2. Epsilon-greedy formula (top) & Use main DQN's action value to select action (bottom) [1]

Agent then takes action. Resulting experience tuple (<s, a, r, s', done>) is stored in replay buffer. At each step, a random batch of experiences is drawn from memory and trained using main DQN. At batch training, target vectors (y_j) is computed as reward if state is terminal or by adding reward to discounted future values using target network as follows:

$$y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$$

Figure 3. Use target network to evaluate target value [1]

Where ø' and ø are weights of target and main DQN, respectively. Loss is then optimized using Adam as:

$$\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$$

Figure 4. Loss is defined as MSE between rewards between target network and main network [1]

Target DQN is then updated with main DQN's weight every C steps to reduce overestimation. Training ends when agent's last 100 episodes average to rewards of >= 200.

4. Why select DDQN?

Lunar Lander's complex continuous space state requires algorithms with a non-linear function approximator such as DQN to sufficiently extract spatial features. However, DQN tends to overestimate action value as it uses single function for action selection and evaluation, causing potential divergence

for complex problems. Therefore, DDQN is selected for implementation to reduce error in function approximation, sample correlation, and overestimation of action value from non-stationarity.

5. Experiment 1 – Training Model

First experiment is to develop a model to achieve average score of >= 200. To do so, DDQN is implemented as previously mentioned. There are 4 hyper-parameters (annealing size N_A, learning rate α, discount factor C, and batch size K), which are examined in this project.

5.1 Result

Setting N_A = 150, α = 0.0015, C = 2, and K=25 results are generated as follows (Avg. Reward is average of last 100 episodes' scores):

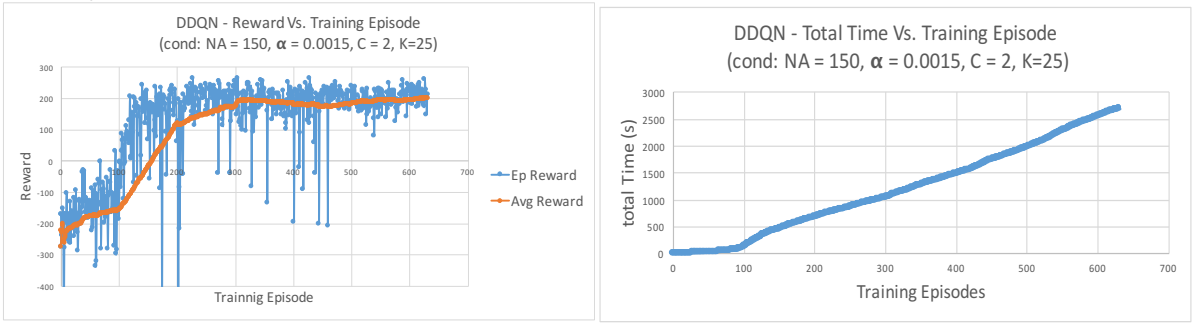

Figure 5. DDQN - Reward vs. training episodes (left) and Total Time (s) vs. episode (right)

As shown, this condition solves Lunar Lander in 640 training episodes using ~45min. However, this model is not fully optimized as it reaches score 194 as early as 325 episodes. Thus, it spends 306 episodes converging to 200. There is noticeable variance after 150 episodes. This is likely because agent is likely not sufficiently trained to random states. Increasing randomness N_A should resolve this.

6. Experiment 2 – Testing Model

In this section, previously trained DDQN model is tested for 100 trials. This experiment is implemented as follows: for each step of test episode, take action predicted by previously trained main DQN, and accumulate reward for each step until episode terminates.

6.1 Result

Testing previously trained model (N_A = 150, α = 0.0015, and C = 2), results are as follows:

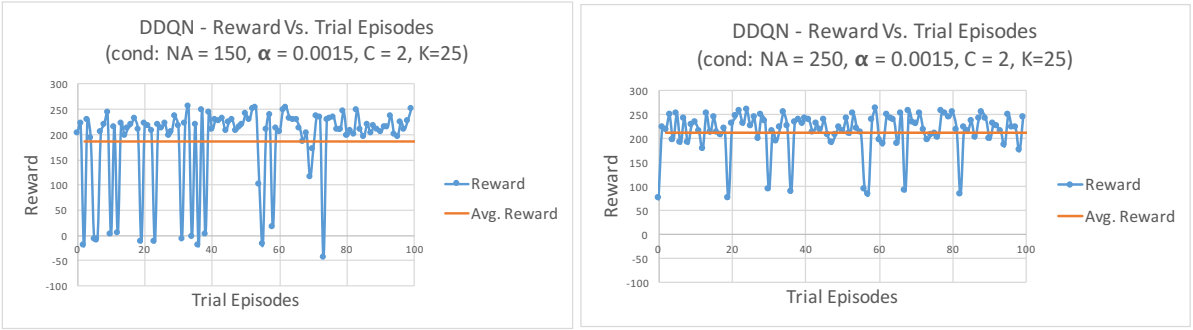

Figure 6. DDQN - Reward vs. trial episodes using previous model (left) and model with N_A = 250 (right)

As shown (left), 100 trial runs of previously trained model (N_A = 150) give an average score of 185.2 with high variance. This high variance is likely due to agent not exposed sufficiently with randomness during training. To demonstrate, Figure 2 (right) plots trial result for a trained model with N_A = 250 or increased randomness at training. As shown, increasing training on random states improves stability, resulting 212.8 average score for N_A = 250 model.

7. Experiment 3 – Hyper-parameter tuning

This section examines effects of hyper-parameters on DDQN. To do so, training performance for a range of randomness (N_A = 150, 250, 500), learning rate (**α** = 0.0010, 0.0015, 0.0020), update steps (C=2, 5), and batch size (K=25, 50) are examined. These parameters are chosen because: **α** affects valuation of new training experiences, N_A affects magnitude of randomness, K affects correlation of sample experiences, and C affects overestimation of action value. Ranges are chosen to solve problem within sufficient time.

7.1 Result

Randomness is analyzed by fixing **α**=0.0015 and C=2 while learning rate and C are examined by fixing N_A = 150. Results are generated as follows:

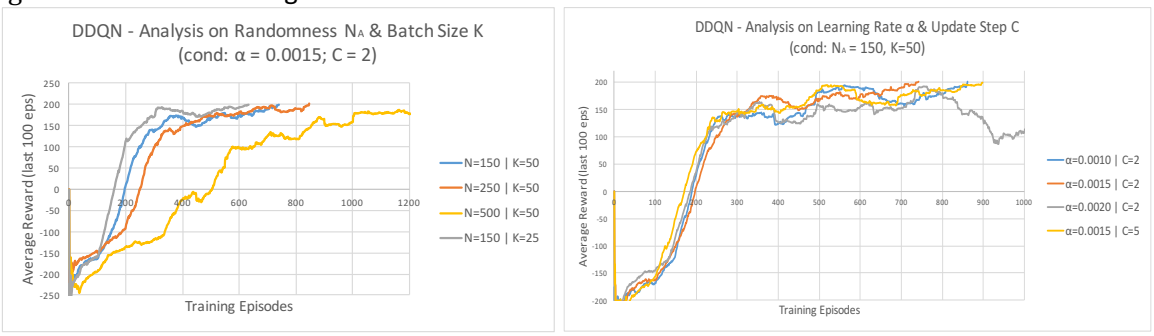

Figure 7. Analysis on randomness (left) and learning rate & C (right) using average reward at training

As shown (left), increasing randomness from N_A = 150 from N_A = 500 slows convergence significantly. This is because at higher N_A, agent performs more exploration, resulting in network weighted more on random experiences. However, trained model is more stable using higher N_A as previously examined. From plots (right), optimal learning rate is achieved at median **α** = 0.0015. This is because, at low **α**, agent does not efficiently learn from new experiences so that more visits are required at similar states. In contrast, if **α** is too high (>=0.0020), training overfits as new experiences are overly weighted.

In addition, lower update steps (C=2) outperforms that of higher. This agrees with Mnih's work as a sufficiently high update step should increase underestimation in reward by lagging to update new experiences. Lastly, batch size K=25 solves the problem ~14% quicker than at K=50. This is because higher K causes model to train more on prior random experiences during annealing steps.

8. Initial Experimental Pitfall

One of the challenges in this project was tuning hyper-parameters. Many initial experiments failed to solve Lunar Lander. For example, DQNs were initially much more complex, using 4 layers, 128 nodes each, and with a low **α** = 0.00010. Computation and learning were sufficient slow to converge that scores did not improve for over 15 hours. Reducing learning rate to 0.0015 improved performance significantly, allowing convergence to ~7 hours. However, it was not until suitable batch size, annealing size and update steps used that Lunar Lander was solved within an hour, demonstrating that it is critical to tune hyper-parameters.

9. Conclusion

DDQN is a power algorithm that can efficiently solve challenging AI problems such as Lunar Lander. However, selecting correct model/network and range of hyper-parameter significantly affects its performance. It is critical to tune learning rate, discount factor, annealing size, and batch size so that model does not overfit but still provide sufficient training experiences to solve problem in suitable time.

10. Video Presentation

https://youtu.be/x8SppvHqq80

11. Reference

[1] V. Mnih. Human-level control through deep reinforcement learning. Nature 518, 529-533, 2015.