Solving the Lunar Lander Problem with Deep Q-learning with Experience Replay in PyTorch

Ansel Lim / ansel@gatech.edu

Github commit hash: —

I. Introduction to the Lunar Landing problem

The lunar lander environment in OpenAI Gym is a classic reinforcement learning problem framed as a classic rocket trajectory optimization problem.

The lander starts at the top center of a 2D plane with a random initial force. The lunar rocket has two legs (left and right), and three engines (left, main, and right). The task is to land tje lunar rocket on the landing pad, which is fixed at coordinates $(0, 0)$. The description of the actions, states, and rewards in the following paragraphs is directly taken from the official OpenAI Gym documentation [1].

There are four discrete actions in the action space: do nothing, fire the left engine, fire the main engine, and fire the right engine. The state vector is an 8-vector consisting of the two coordinates of the lander, its velocities in the x and y dimensions, its angle, its angular velocity, and two booleans which represent whether the left and right leg are in contact with the ground. The state vector may be written as an 8-tuple:

$$(x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, \text{left\_leg}, \text{right\_leg}) \tag{1}$$

The possible rewards that the lunar lander may gain are as follows:

1) Moving away from the landing pad produces negative reward. That is, the agent is penalized the amount of reward that would be gained by moving toward the pad.
2) If the lander crashes, it receives -100 points.
3) If the lander comes to rest, it receives 100 points.
4) Each leg with ground contact gains 10 points.
5) Firing the main engine costs -0.3 points.
6) Firing a side engine costs -0.03 points.
7) Solving the problem yields 200 points. The problem is considered solved if an average score of 200 points or higher is achieved over 100 consecutive games.

The episode terminates if any of the following occurs.

1) The lander crashes
2) The lander exceeds the bounds of the viewport/2D plane.
3) The lander comes to rest, that is, it doesn't move and doesn't collide with any other body.

II. Introduction to "Classical" (Non-Deep) Q-Learning

As described in [2], Q-learning is an off-policy temporal difference control algorithm which learns an action-value function Q for each state and action pair. The action-value function Q is a mapping between state-action pairs and rewards. Note that the learned action-value function directly approximates the optimal action-value function without depending on a policy, therefore Q-learning is an off-policy learning procedure. The Q-learning procedure is defined by the Bellman equation

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2}$$

The discount factor γ in this equation characterizes the tradeoff between prioritizing early payoff versus late payoff of rewards.

The procedure aims to build a Q-table. The size of this Q-table is the product of the number of distinct states and the number of actions available in the action space. At the start of the procedure, the Q-table is initialized to random values for all its values except the terminal states, where the action-values are zero. The procedure updates the Q-table over multiple episodes. In each episode, after initialization of the state S to the starting state, the following steps are performed sequentially until a terminal state is reached.

1) Use an epsilon-greedy policy to choose A from S.
2) Take action A, and observe the new state and reward.
3) Update $Q(S, A)$ from its previous value, using the increment
$$\left[R_{t+1} + \gamma \max_{A\prime} Q(S', A') - Q(S, A) \right] \tag{3}$$
multiplied by the learning rate α.
4) Proceed to the next state; that is, the variable S now points to the new state.

While this method is attractive because of its simplicity, it is likely to do very poorly on the lunar lander problem. This is because we would have to construct an explicit Q-table for all the possible state and action pairs. Notice that the state vector 1 is not a discrete vector, since the coordinates, velocities, angle, and angular velocity are all real-valued numbers (floating point). Clearly, for a non-discrete case, a Q-table is not computationally feasible for two reasons:

1) Since there are an extremely large number of (s, a) tuples, depending on the precision of the floating point representation of real-numbered values, the amount of memory required to create and save this table is intractable.
2) Furthermore, it is likely that even with a long training time, the agent would not have seen ALL the possible (s, a) pairs that it may encounter at test time. That is, it is likely that some states are not going to be explored appropriately.

One naive approach to solve this issue is to employ binning. That is, the problem may be discretized; continuous values may be binned into discrete categories. For example, we may choose to fire the left rocket if the x and y coordinates fall in a certain range. However, this method is not satisfying and it is susceptible to design flaws, since the choice of binning is manual and assumess substantial prior knowledge of the environment.

III. Introduction to Deep Q-learning

Deep Q-learning, as the name suggests, is nothing more than using a deep neural network as a function approximator to estimate the Q-values. That is, instead of building an explicit Q-table, we build a neural network which takes a state vector S, as in 1, as input, and returns the action values $Q(S, a)$ for each action a executable at state S. This approach aims to benefit from the universal approximation power of deep neural networks.

A paper published by DeepMind researchers [3] used deep Q-learning to solve Atari. Specifically, the authors described a technique which they called *deep Q-learning with experience replay*. The pseudocode is described in detail as "Algorithm 1" in the paper and it will not be reproduced here. Here, I describe the high-level intuition. This technique was used to update the neural network which approximated action-values for state-action pairs. Every time the agent makes a step and observes the environment, it collects information about its experience (in particular, including the next state observed and the reward obtained) which it stores in its memory. The technique of experience replay is characterized by the agent periodically drawing samples from its memory to perform updates to the weights in the network by comparing the Q-value predictions of the network with target Q-values, which are derived from applying the Bellman equations to successor states.

A. Applying experience replay to the Lunar Lander problem

I applied the technique of experience replay directly to the Lunar Lander problem. I will describe the technique in more detail here. At each time step t, the agent performs an action from its current state s_t (an 8-vector), based on an epsilon-greedy approach. An epsilon-greedy approach means that with probability $(1 - \varepsilon)$, the agent takes a greedy action, and with probability ε the agent takes a random action in its action space; any action is okay as long as it is a legal action. Say that the action chosen is a_t. The agent executes the action a_t from the state s_t and observes a reward as well as the next state. Define the agent's "experience" to be a tuple $e_t = (s_t, a_t, r_t, s_{t+1}, e_{t+1})$, consisting of the current state, s_t, the action taken, a_t, the reward obtained, r_t, the next state, r_t, and a boolean e_{t+1} which indicates whether the next state is a terminal state. This experience is stored in the memory of the agent in a buffer of finite and fixed length N.

Every k time steps (the exact value of k chosen is a hyperparameter designed by the programmer), learning, or updating of the agent's Q networks, is performed. The agent draws a minibatch sample of m experiences from its memory buffer. Note that learning is done only if the number of experiences actually in the buffer is more than or equal to the minibatch size m; if there are fewer than the minimum, then we continue accumulating experiences by exploring the environment until there are sufficient experiences to construct a minibatch from.

The state-action pairs in these m experiences are fed into the existing Q-network; forward propagation produces predicted action values of state-action pairs. These predictions are then compared with target action-values which are obtained by applying the Bellman equation (equation 2) on the state-action pairs and their associated next states. The loss criterion applied to the predicted and target values is the mean squared error loss. Notice that the discrepancy between the predicted action-values (calculated by forward-propagating on s_t and a_t) and the target action-values (calculated from applying Bellman equation on the experiences) drives the loss, which the learning procedure attempts to minimize. The weights of the network may then be updated by means of a gradient-based approach based on backpropagation of the loss function, just like any other neural network. For example, stochastic gradient descent or the Adam optimizer may be used.

B. Advantages and disadvantages of experience replay

According to the paper [3], the method of experience replay, which is characterized by random minibatch selection of experiences in each optimization step, has several advantages:

1) The experiences selected for each optimization step are randomly chosen. Therefore, they are not consecutive samples like in classical Q-learning. Consecutive samples are strongly correlated and may lead to noisy, high-variance updates.
2) Random selection of experiences in the replay buffer results in averaging of the behavior distribution over several states, so that there is less noise, oscillations, or divergence. In contrast, with an on-policy method, the training samples could be biased very strongly by maximizing actions. For example, the training samples could by chance consist of states where the lunar lander is on the left side of the plane, in a case where "moving left" is a strong maximizing action that biases the behaviors.

A disadvantage of this simple approach is that we are working with a finite memory buffer, and furthermore in the mini-batch construction, all experiences/transitions are given equal weight. This uniform sampling of transitions from the memory buffer deprioritizes important transitions, potentially leading to suboptimal convergence speed.

C. Primary and secondary networks

It is important to note that this is a deep reinforcement learning algorithm, and so it suffers from a common problem of a "moving target". Unlike supervised learning tasks, the target values (for the same inputs) are a nonstationary target. This follows directly from the Bellman equation (equation 2), where the target Q-values for a state S and action A has a

dependence on the Q-values of the successor S' and its actions. Intuitively, with an increasing number of episodes, we find out more about the environment, and we have a better idea of the action-values of state-action pairs.

Notice that in the currently described paradigm, we are using a single neural network to calculate the predicted value of $Q(S, A)$ (for all actions A executable at S). This predicted value is then compared with the target value which is obtained by applying the Bellman equation (equation 2) on the experience tuple. However, the problem is that to apply the Bellman equation, we need to use the value of $\max_{A'} Q(S', A')$ which is, of course, calculated by forward propagation of the successor state S'. Therein lies the problem. If we use the same network to calculate predicted and target action-values, we are basically using the same network to guide itself toward the global optimum. This method is prone to divergence and ill-conditioning.

A better approach is to use one network (the "primary" network) to predict $Q(S, A)$ and use another network (the "secondary" network) to calculate the prediction $\max_{A'} Q(S', A')$, which is then plugged into the target $R_{t+1} + \gamma \max_{A'} Q(S', A')$. The difference between the prediction and the target recovers exactly increment seen in equation 3.

The two networks, of course, have to be "synced" with one another, since it is our ultimate goal to build a close enough approximator of Q-values for all possible state inputs into the deep Q-learning network. Two approaches may be used to accomplish this:

1) Interpolating the two networks at the end of each learning step, where the secondary network's parameters are set to a linear combination of its parameters as well as the primary network's parameters (which had been updated during the learning step)
2) Copying over the weights of the main network to the secondary network without any interpolation parameter. This is performed periodically every x number of iterations (up to the user) and with a good value of x, the Q-values may be more stable and less oscillatory.

IV. Implementation of Deep Q-learning

Deep Q-learning with experience replay and primary and secondary networks was implemented in PyTorch. The implementation closely parallels the theoretical and conceptual discussions in the previous section. For the neural networks (deep Q networks), multilayer perceptrons were used. Although the game has a visual representation, the state at any point in time is represented (summarized) as a simple 8-vector, making a multilayer perceptron suitable for this task. Some points about my implementation are described here. This will be referred to as my "baseline" implementation upon which modifications will be made.

1) Primary and secondary networks are used. There are three hidden layers of 128, 128, and 32 neurons each. The ReLU activation function is used.
2) The replay buffer contains up to 10,000 experiences.
3) In each episode, a time step involves selection of an action via the epsilon-greedy algorithm. With probability ε, a random action is selected. With probability $(1 - \varepsilon)$, a greedy action is selected based on the action-values calculated using the primary network. The value of ε starts at 1 at the beginning of training, and it is decayed by a multiplicative decay factor of 0.9 with each time step. The minimum value of ε is 0.01.
4) A learning step occurs once every 4 time steps of an episode.
5) The maximum number of time steps in each episode is 1000.
6) In each learning step of the agent, a batch of 64 experiences (transitions) is randomly selected. The starting states in these experiences are forward propagated through the primary network to obtain predictions of the Q-values of the starting states. The next states observed in these experiences are forward propagated through the secondary network, and then the target Q-values of the starting states are computed by means of the Bellman equation. The discount factor is $\gamma = 0.99$. The mean squared error loss between the target values and the predicted values drives the backpropagation and optimization process for the primary network. The Adam optimizer with learning rate of 0.001 is used for the optimization process.
7) Interpolation between the primary and secondary networks occurs with each learning step of the agent. The secondary network's parameters are set to a linear combination of its own parameters and the primary network's parameters. This is governed by the interpolation parameter τ.
8) The training process keeps track of the cumulative score in each episode, averaged over 100 episodes.
9) The maximum number of episodes is 4000. Early stopping is implemented. If the average score (over 100 episodes) exceeds 200, then the algorithm terminates.

V. Results of Deep Q-Learning Procedure

Figures 1 show the cumulative reward per episode during the training process for different experimental configurations. The following hyperparameters were explored. Note that in each experiment, only ONE hyperparameter was changed from the "baseline" configuration described in the previous section, that is, "other things being equal". The captions in each figure provide some numerical details of the results.

1) Learning rate (α): 0.001 (Figs. 1, 2), 0.01 (Fig. 3), and 0.1 (Fig. 4)
2) ε decay factor in the epsilon-greedy algorithm: 0.5 (Fig. 5) and 0.1 (Fig. 6)
3) Update interval: 10 (Fig. 7) and 1 (Fig. 8)

Fig. 9 shows the performance of the trained baseline model on 100 episodes. This is the model whose training curve was shown in Fig. 1. Note that early stopping was instituted during training. It is plausible to think that the test results may be better if the model is overfit to the environment with a larger

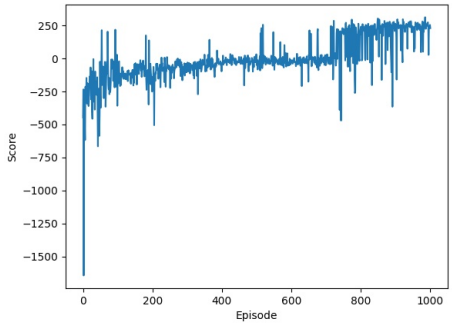

Fig. 1. Training curve of baseline model. Configuration of the experiment is unchanged from the description in section IV. Achieved average score of 223.23 after 1,000 episodes (early stopping). Running time was 981.10 seconds.

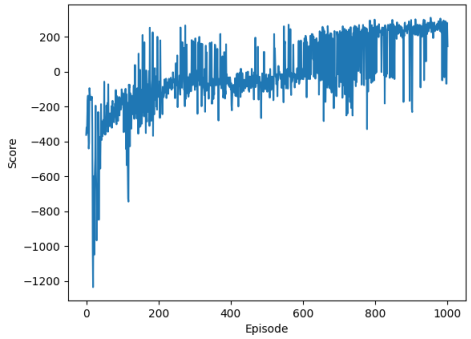

Fig. 2. Training curve of another run of the baseline model. Final achieved score is smaller than the previous run in Fig. 1. The running time was similar. The oscillations appear larger and noisier here than in Fig. 1.

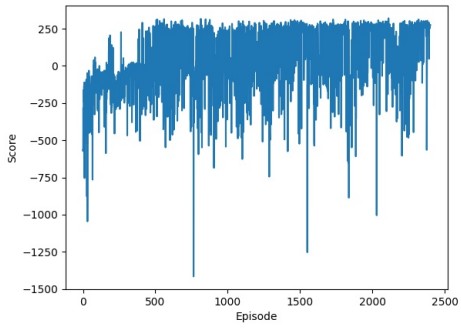

Fig. 3. Learning rate of 0.01 instead of 0.001. The configuration is otherwise the same. After 2,400 episodes (early stopping), final mean score was 230.34. Running time was 1004.9 seconds.

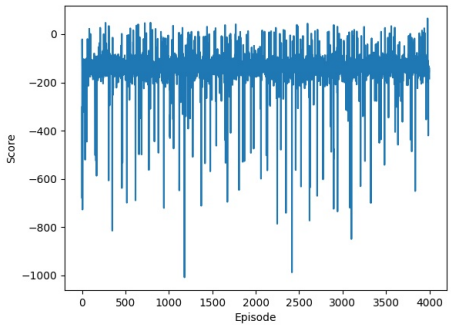

Fig. 4. Learning rate of 0.1 instead of 0.001. After 4,000 episodes (the maximum allowable), the agent failed to solve the problem. Final mean score was -129.80. Running time was 180.69 seconds.

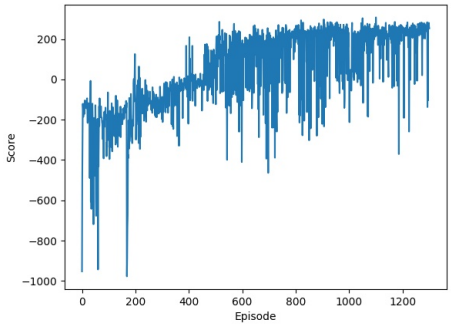

Fig. 5. Epsilon decay factor changed from 0.9 to 0.5. After 1,300 episodes (early stopping), the final mean score was 216.66. Running time was 775.3 seconds.

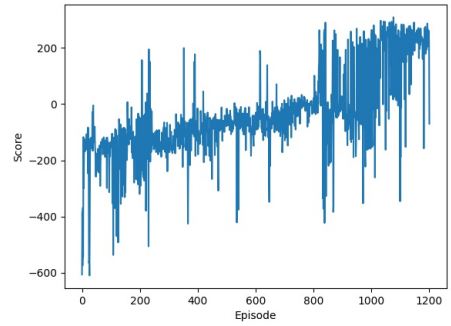

Fig. 6. Epsilon decay factor changed from 0.9 to 0.1. After 1,200 episodes (early stopping), the final mean score was 216.5. Running time was 991.2 seconds.

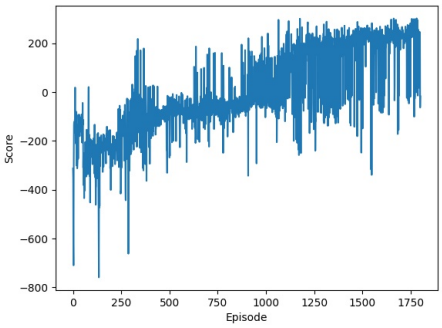

Fig. 7. Update interval changed from 4 to 10. This means that the agent updates every 10 time steps, instead of every 4 time steps. After 1,800 episodes (early stopping), the final mean score was 208.14. Running time was 1122.9 seconds.

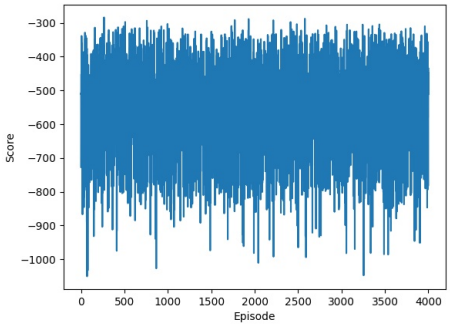

Fig. 8. Update interval changed from 4 to 1. This means that the agent updates every single time step. Ironically, with such frequent updates, the agent failed to learn the environment. After 4000 episodes (maximal), the final mean score was -540.9. Running time was 68 seconds.

number of epochs. However, on my initial experiments, it seems that this may not be the case Fig. 10 shows how the same model, trained without early stopping until 2000 epochs, performs similarly to the model with early stopping.

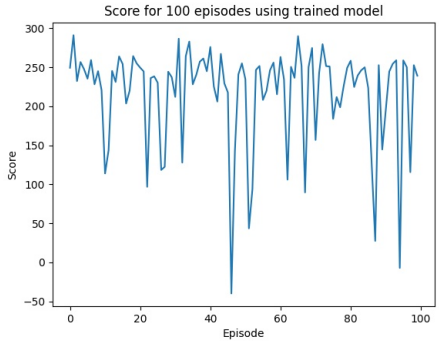

Fig. 9. Baseline model, evaluated on 100 episodes. The average score was 216.68. Notice that although the average score exceeds 200, on several episodes it is painfully obvious that the agent is failing at landing.

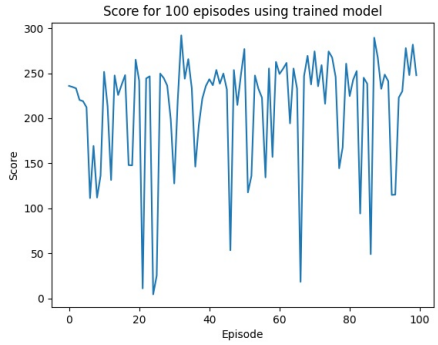

Fig. 10. Baseline model without early stopping, evaluated on 100 episodes. Due to computational constraints, I trained until 2,000 epochs instead of allowing the model to train until 4,000 epochs. The average score was 210.80.

VI. Discussion

The algorithm worked well. With the baseline configuration, the agent could learn to solve the problem after 1,300 episodes. It makes sense that the score improves with increasing episode count. As the agent sees more and more transitions, it makes periodic updates of the deep Q learning networks, and with sufficient steps of the optimization procedure, the network eventually becomes a good approximator of the optimal Q-value function, which is likely high-dimensional and very difficult to approximate without the power of nonlinearities which deep nets do so well in. Since the Q-value function implicitly defines a policy at each state, a better approximation of the Q-value function translates into a better policy which is likely to maximize the cumulative discounted rewards, i.e. the score.

I noticed that the choice of hyperparameters had variable impact on the problem.

1) *Learning rate:* With a large learning rate of 0.1, the agent could not solve the problem, whereas the agent with the "baseline" default learning rate of 0.001 did well. This is likely due to the large learning rate creating overly large changes in the Q values, so that the local minimum is never achieved; rather, the agent skips over local optima by making very large strides.
2) *ε decay factor in epsilon-greedy selection algorithm:* Minimal difference is seen.
3) *Update interval:* With over-frequent updates, the learner fails to learn the environment.

VII. Reflections and Further work

If I had more time, I would perform more hyperparameter tuning and try a systematic neural network architecture search. It would also be interesting to try a convolutional neural network.

I would also like to explore how to make the algorithm more robust and less noisy, since I notice that the oscillations between episodes can be quite substantial. Perhaps one approach is to use dropout layers, or to try a larger batch size in an attempt to reduce the variance.

REFERENCES

[1] OpenAI Gym documentation and source code. Published by OpenAI. Accessed at https://gym.openai.com/envs/LunarLander-v2/ & https://github.com/openai/gym/tree/master/gym on 20 Mar 2022.

[2] Sutton, Richard and Barto, Andrew. "Reinforcement Learning: An Introduction", Second Edition. MIT Press (2013), Cambridge, MA. Accessed at http://www.incompleteideas.net/book/the-book.html on 20 Mar 2022.

[3] Mnih, Volodymyr, Kavukcuoglu, Koray, et al. "Playing Atari with Deep Reinforcement Learning" NIPS Deep Learning Workshop 2013. Accessed at https://arxiv.org/abs/1312.5602 on 20 Mar 2022.