James Hundertmark
CSPB 3202 – Artificial Intelligence

https://youtu.be/8rcxioNrKHg
https://github.com/jahu6304/lunar_lander_dqn.git

Overview

The algorithm I built is a Deep Q-Network (DQN) for Gymnasium's LunarLander-v3 environment. The goal of this environment is to develop an autonomous agent (the lunar lander) that could touch down on a landing pad using reinforcement learning techniques. The lunar lander is rewarded based on its position (x/y coordinates) and velocity, angle, and angular velocity, and the number of lander legs that made contact with the ground. I trained the model over the course of more than ten thousand episodes.

Approach

In the code is defined two classes, "dqn" and "replayBuffer". The "dqn" class is the initialization of the neural networks, the layers of weight matrices and bias vectors that Q-values are passed through adjust them for proper action choice. Initially, these values are random, but as the model is developed, these neural networks are optimized to evaluate random state action pairs. Inside this class is the "forward function" that uses the ReLU activation function, which zeroes any negative values in the hidden layer matrices and vectors of the neural networks. This is to enable the model to process nonlinear inputs. As the neural networks are built via the "linear" function, the matrices and vectors it produces are initially linear. By zeroing any negative values, matrices can produce non-linear outputs, allowing the networks to process more complex, nonlinear inputs. The negative values of the final matrix and vector of the neural network is not zeroed to allow the raw values, both positive and negative, to be applied to the q-values.

The "replayBuffer" class is simply the class where state actions are saved to an array for future reference. It removes states if the replayBuffer is at capacity to add new states, and randomly samples a certain batch size of these states for realearning.

The code also has two functions, the "chooseAction" and "trainStep" functions. The "chooseAction" function takes two arguments, the current state and the exploration coefficient, "epsilon". Epsilon's value dictates the chance that the lunar lander will make a random action from its current state, to evaluate otherwise unexplored actions from that state. Epsilon's value will slowly degrade over the course of the training episodes, as in the beginning the model needs a better idea of the value of different actions from states, while towards the end of the training, it can start to apply what it has learned, and develop more positive rewards. Without this, the model may find a sequence of actions that works, but is not necessarily the best set of actions to take, and get stuck in the groove of simply "actions that work". The "chooseAction" function generates a random number, and if that number is less than epsilon, the model will take a random action. Otherwise, the model chooses the action with the highest q-value.

The "trainStep" function is the function that actually trains the neural network. It retrieves a batch of states, actions, rewards, next states, and done indicators from the replay buffer. It finds the policy network's predicted q-value for selected actions from the current states. From the target network, it gets the maximum Q-value for the next states, and calculates the target Q-values as the immediate reward plus the discounted future Q-value, taking into account whether the episode has ended (using the done indicators). Finally, it minimizes the mean squared error between the predicted Q-values and the target Q-values to update the policy network with.

The training loop of the code runs a certain number of lunar lander episodes. Starting from a random state, the "chooseAction" function is run for the start state and each consecutive state until done = True breaks the while loop. Each state is stored in the replay buffer to revisit in the neural

network optimization. Once a action and next state have been selected, the reward/penalty for that action is applied to the total reward. (The "reward" parameter is predetermined from the gymnasium environment via the ".step(action)" method.) When an entire episode has run, the exploration coefficient, epsilon, is reduced, and if the specified period of episodes has elapsed, the target network is updated with the policy network. Once all the episodes have run, the model is saved to a .pth file for future use.

Q-learning is an algorithm that develops the optimal action-value function, Q*(s, a) given state action inputs. The values of taking certain actions in different states is adjusted as the process is run through repeatedly, adjusted via a learning rate and discount factor (that throttle the amount reward/penalty affect). Given that the lunar lander is a continuous, two dimensional state space, there are an infinite number of possible states. DQN is applicable to this situation as a neural network, as it is able to generalize it's training, navigating not previously seen states due to their similarity to previously seen states.

DQN also replays previously seen states during training to break up any patterns that might develop in its learning from correlating similar states. It does this by saving previously seen states to whats called a replay buffer. This buffer has a capacity of states that is saves, and once that limit is reached, it starts overwriting already replayed states.

Another feature DQN implements is called target network. DQN maintains two neural networks. The policy network is updated after every training episode. The target network is another neural network that copies the policy network saves to periodically. The policy network is optimized toward this target network instead of itself, to provided stability and prevent oscillations in weights and biases applied by the network.

Troubleshooting

Previous troubleshooting I had done was ineffective. I had a lack of understanding of how much training a model needed to show improvement. I was only running the model through about one thousand episodes, expecting to see improvement. When the model didn't, I adjusted the reward/penalty values to be more significant and increased the learning rate coefficient, hoping to more significantly affect the training. Now understanding it takes thousands of episodes to train a model, I reset those values, and saw improvement (although the model is now oscillating).

Something I had also previously implemented (that I will reimplement), is adding in a ninth input, "frameCount". Although the prebuilt gymnasium reward system already penalizes the model for using the main thruster, the model still has a significant problem with constant hovering. I would like to more directly penalize the model for taking too long to land, by penalizing it for frame count (the number of action states it takes in an episode). This also has the affect of reducing training times.

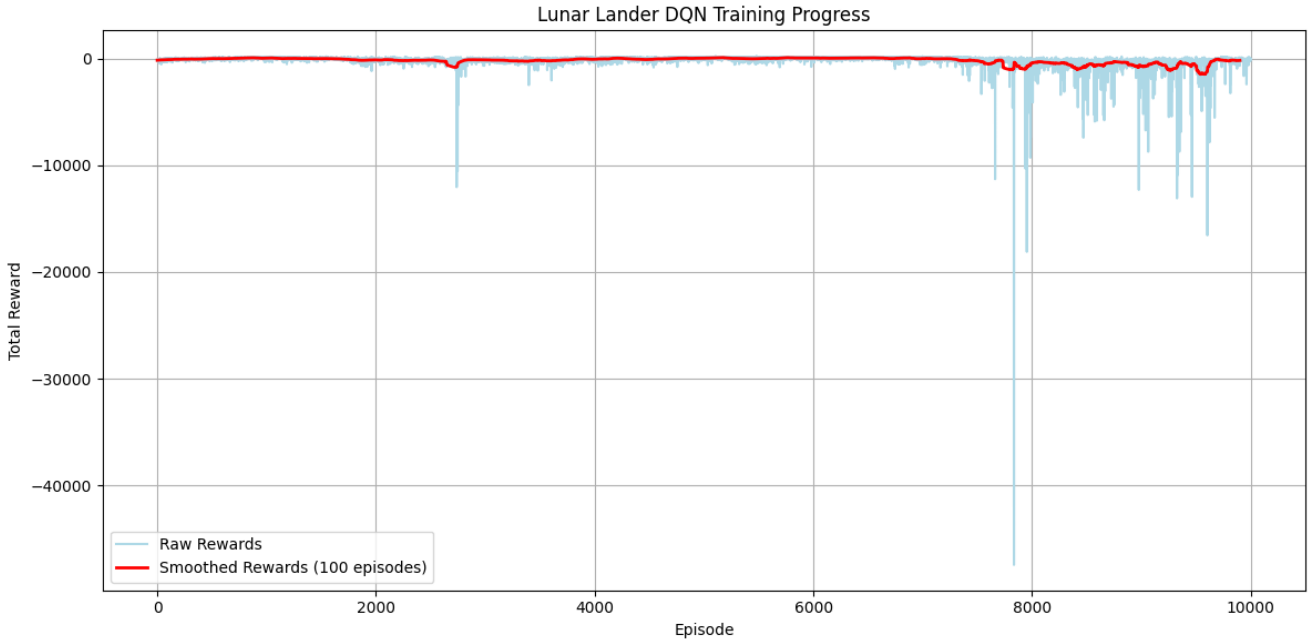
I also added a line that breaks an episodes loop by setting "done" to true if both lander legs have touched down, regardless of where. This prevents the lander from ruining reward for a good landing by continuing to side thrust because the episode hasn't ended.

Results & Reflections

My algorithm runs successfully and shows definitive learning from being trained through a series of ten thousand episodes. However, the results are oscillating between long streaks of success and failures. For the first four thousand episodes, the total reward would vary around negative one hundred. From episodes four thousand to about eight thousand, the model was consistently producing positive total rewards. From episode eight thousand until it finished the model got progressively worse and worse, almost always producing negative total rewards, sometimes in thousands (e.g. the lowest reached -47000).

There are several factors I believe to be causing this. Firstly, I believe my target network update period was FAR too frequent for the number of episodes being run through, and thereby not providing the stabilization needed to the optimization of the policy network (I had the target network update period set to ten and have changed it to five hundred for a series of ten thousand episodes.)

I also believe that the exploration decay coefficient needs to be adjusted, although I'm unsure to which direction. With a higher coefficient, the model will explore more, providing a more informed model. With a lower coefficient, that model will hone in on certain patterns of actions that it discovers earlier. I'm unsure which will stabilize the oscillations, as I'm not sure whether too much exploration is confusing the model, or it is becoming tunnel visioned with poor patterning.



Above is a graph of the rewards from each episode. The blue values are the total rewards from individual episodes, while the red line an the average. It is difficult to see the oscillation, as the negative rewards in the latter part of training are so dramatic.

References

Volodymyr Mnih

https://paperswithcode.com/method/dqn

Chanseok Kang
https://goodboychan.github.io/python/reinforcement_learning/pytorch/udacity/2021/05/07/DQN-LunarLander.html