Kevin McCarville
June 18th, 2024
Reinforcement Learning - CS7642
Project 2 - Lunar Lander
GitHub Hash:
506a223cdbeb1a7cd77f4b3ef20973592e9eaec9

Abstract:
In this paper, we will be implementing and training an agent to successfully land the “Lunar Lander" (LunarLander-v2) from OpenAI gym. We will be applying a Deep Q-Network (DQN) utilizing stochastic gradient descent (SGD). We will be analyzing the performance of our agent based on a reward per episode basis and then evaluate how that performance varies as we tune various hyperparameters and what these changes in performance indicate. We will conclude by noting any observations or larger lessons learned from the results of the experiment as well as a discussion of issues we had and further research we would like to conduct.

Introduction:

A. Lunar Lander

The Lunar Lander problem in OpenAI's Gym environment presents a fascinating and complex challenge, ideal for testing the capabilities of reinforcement learning algorithms. In this problem, an agent controls a spacecraft tasked with landing smoothly on the moon's surface. The environment provides continuous state variables such as the lander's position, velocity, angle, and angular velocity, which the agent uses to make decisions. This is in contrast to the discrete state variables we have seen so far, for instance in our Taxi problem. The agent can choose from four discrete actions: do nothing, fire left orientation engine, fire main engine, and fire right orientation engine. The goal is to land the spacecraft between the flags without crashing, balancing fuel consumption with a gentle descent. This problem encapsulates a real-world scenario of autonomous control under constraints, making it a valuable benchmark for advancements in artificial intelligence and machine learning.

The reward structure in the Lunar Lander environment is designed to encourage successful and efficient landings. The agent receives a reward for moving closer to the landing pad and additional rewards for landing successfully between the flags. Conversely, the agent is penalized for crashing or using excessive fuel. Specifically, the agent receives +100 to +140 points for a successful landing, -100 points for crashing, and a small penalty for each frame to discourage prolonged hovering. Additional rewards or penalties are given for the lander's leg contact with the ground and for staying within the designated landing zone. This nuanced reward structure incentivizes the agent to optimize its trajectory and engine use to achieve a controlled and efficient landing.

As far as stochasticity, the initial conditions of the lander, such as its position, angle, and velocity, are randomized at the start of each episode. This introduces variability in each landing attempt, requiring the agent to generalize its learning across different scenarios rather than memorizing a fixed sequence of actions. The rest of the environment is deterministic.

The inputs of the environment are the state observations of the lunar lander, which consist of an 8-dimensional vector. This vector includes the lander's horizontal and vertical positions, horizontal and vertical velocities, angle, angular velocity, and two boolean values indicating whether the left or right leg has made contact with the ground. The 4 outputs are the actions that the lander can take, represented by an integer in the range [0, 3]. These actions correspond to doing nothing, firing the left orientation engine, firing the main engine, or firing the right orientation engine.

B. Q-Learning

We will be utilizing a Deep Q-Learning Network (DQN) to tackle the Lunar Lander problem. Q-Learning is an off-policy RL algorithm as it improves upon one policy while training on a different policy for actions. We’ll briefly discuss the mechanics of On-Policy and Off-Policy learning and then describe the DQN framework.

I. On Policy (SARSA)

SARSA, which stands for State - Action - Reward - State (next) - Action (next), is an on-policy reinforcement learning algorithm used to learn the action-value function. This function estimates the value of taking a certain action in a given state under a particular policy. The SARSA learning process begins with the initialization of the Q-value function, Q(s,a) arbitrarily for all state-action pairs, followed by selecting an initial state s0 from the environment. SARSA is characterized as an on-policy algorithm, meaning it learns the value of the policy that is actively being used to make decisions. Typically, an ϵ-greedy policy is employed, where the agent primarily follows the policy derived from the Q-values but occasionally takes random actions to ensure adequate exploration. However, we will be using Q-Learning as opposed to SARSA, for this experiment.

II. Off Policy (Q-Learning)

Off-policy Q-learning is a reinforcement learning algorithm that differs from on-policy methods like SARSA in that it learns the optimal action-value function independently of the policy being followed by the agent. In Q-learning, the agent updates the Q-values based on the maximum possible future rewards, regardless of the actions actually taken during training. This is known as the "off-policy" nature of Q-learning. The primary distinction between Q-learning and SARSA lies in their approach to updating the Q-values. SARSA, being an on-policy algorithm, updates the Q-values based on the action actually taken by the policy, which incorporates the exploration strategy. In

contrast, Q-learning, as an off-policy algorithm, updates the Q-values based on the maximum potential future rewards, assuming the optimal action is always taken, which can lead to faster learning of the optimal policy but might also result in less stable learning if the exploration strategy significantly deviates from the optimal actions. This fundamental difference allows Q-learning to be more effective in environments where exploring all possible actions is crucial for identifying the optimal policy. Here are the two update rules:

Q-learning Update Rule:

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

SARSA Update Rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

The key difference is the use of $\max_a(Q(s_{t+1},a))$ which represents the maximum predicted value of the next state, over all possible actions, rather than the action chosen by the current policy.

III. Deep Q-Learning (DQN)

Deep Q-Learning Network (DQN) is an advanced off-policy reinforcement learning algorithm that extends traditional Q-learning by incorporating neural networks for function approximation. Instead of maintaining a table of Q-values for each state-action pair, DQN leverages a deep neural network to approximate the Q-value function, enabling it to handle environments with large and continuous state spaces. The DQN architecture typically consists of several hidden layers with nonlinear activation functions, which allow it to capture complex patterns and relationships in the input data. Training the network involves using stochastic gradient descent to minimize the mean squared error between the predicted Q-values and the target Q-values. The target Q-values are computed using a separate target network, which helps stabilize the learning process. By combining the strengths of deep learning and Q-learning, DQN has demonstrated remarkable success in various challenging domains, such as playing Atari games directly from pixel inputs. Below is the pseudo code.

Process:

We will implement a DQN to solve the lunar lander problem. Below are the parameters that will be static throughout the experiment:

- ANN layers - input layer (6 node inputs from environment), 2 64-node hidden layers, output layer (4 node outputs for possible actions)
- epsilon start - 1.0
- epsilon decay - 0.998
- epsilon min - 0.01
- ANN learning rate - 0.001
- episodes - 1500

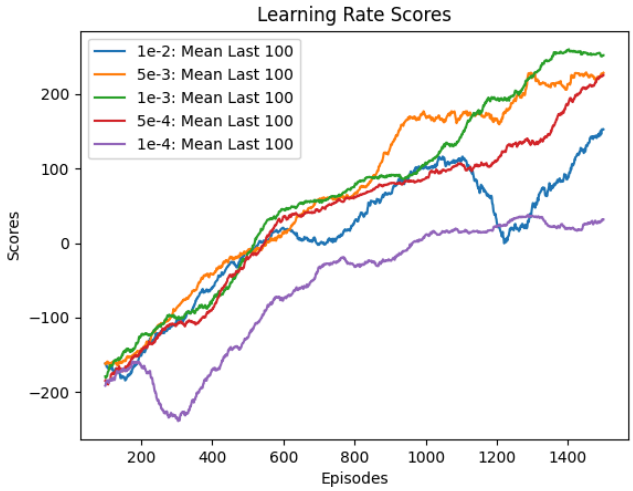
```
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
Initialize target action-value function Q' with same weights as Q

for episode = 1 to M do
    Initialize state s
    for t = 1 to T do
        With probability ε select a random action a
        otherwise select a = argmax_a Q(s, a)
        Execute action a in environment
        Observe reward r and next state s'
        Store transition (s, a, r, s') in replay memory D
        s = s'

        if replay memory D has enough samples then
            Sample random minibatch of transitions (s, a, r, s') from D
            for each transition do
                if s' is terminal state then
                    target = r
                else
                    target = r + γ * max_a' Q'(s', a')
                end if
                Perform gradient descent step on
                (target - Q(s, a))^2 with respect to network parameters θ
            end for
        end if

        Every C steps, update target network: Q' = Q
    end for
end for
```

These decisions were made through a little trial and error to get relatively solid parameter values so we can observe differences in performance adjusting other parameters. The 1500 episodes were used as this is about the point where on certain combinations, the Lunar Lander problem is “solved” by its own definition of obtaining an average of over 200 reward for 100 consecutive episodes.
We will continuously utilize the OpenAI gym to query the Lunar Lander environment to receive our rewards and new states from our actions in the current state. We will now analyze the results.

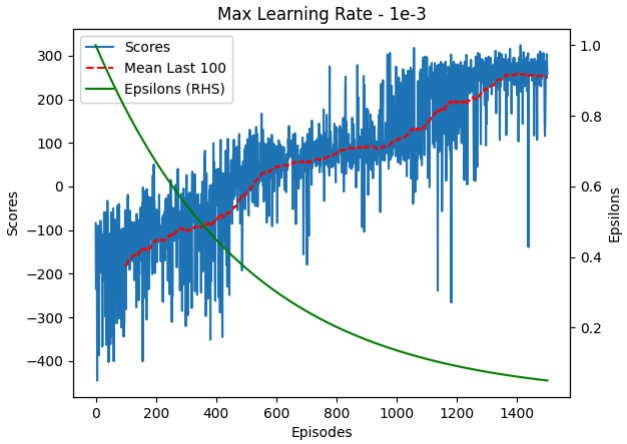
Experiment and Results:

For this experiment we will vary 3 hyperparameters and evaluate the resulting change in performance: Learning rate, gamma, steps per episode. We will then utilize what we learned from those experiments to train an agent and extrapolate what the changes in performance say about the learning process.

The first hyperparameter we tuned was learning rate:

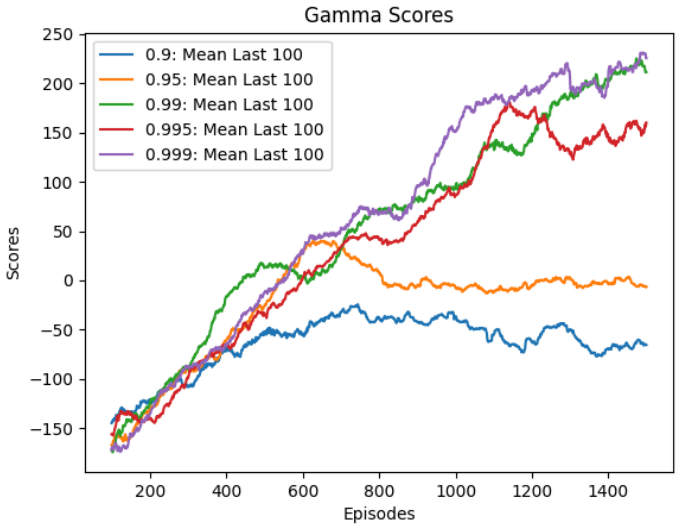


Our best performing learning rate was 0.001. It was closely followed by 0.0005 and 0.005. The two extreme values of 0.0001 and 0.01 had the most interesting behavior. 0.01 had the most volatile behavior, decreasing rapidly from 1000 to 1200 episodes but simply bouncing around more than the other rates. This is somewhat to be expected as the higher learning rate can lead the agent to learn more forcefully which can sometimes lead it astray as its learning isn't as steady as the lower rates. The 0.0001 rate's path is interesting too as it is the worst performer by a wide margin as it simply is too slow to pick up useful data. One is tempted to say it will eventually converge to the others performances given more episodes but the trend above looks like it is slowing down towards a potential plateau. It's also noteworthy that the four lowest rates learn at the same speed for the first ~600 episodes. This is somewhat to be expected as while epsilon is high in the early going, most actions are random so whatever has been learned is not incorporated into any policy decisions yet.
Here are the results when we run with just a learning rate of 1e-3, our best performer:

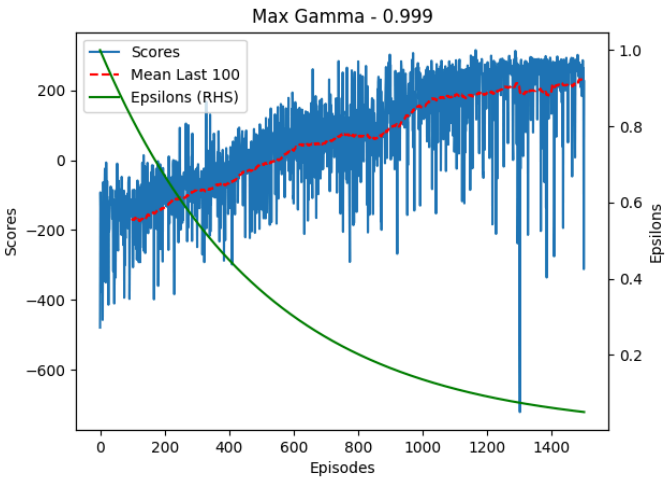


The 100 rolling average shows pretty consistent progress, though it does appear to be plateauing at the end but that might be temporary. It's also noteworthy that there are still some real downside cases (downward spikes) occurring late in training. Epsilon is still close to 10% at the end so this could just be some random exploration gone very awry. For what it's worth, the problem is considered "solved" when the 100-run average is above 200 so by that standard, this agent has indeed solved the Lunar lander problem.

Next we tuned our gamma rate. For reference, gamma was 0.99 for the learning rate tuning. For this exercise, we will utilize our assumed optimal learning rate of 0.001. Here are the results

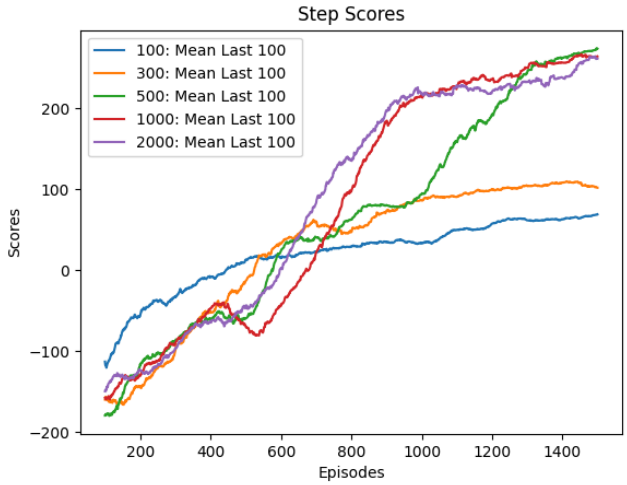


The first thing that jumps out here is the underperformance of the two lowest gammas. They flatline and actually decrease around 800 episodes. The decay in learning that permeates throughout the steps is just too great and not enough learning can be passed back. Thus their performance plateaus. The other three rates perform comparably for most of the experiment. The 0.995 figure, the middle value of the 3 top performers, performed the worst and had the highest volatility of the 3. I think this was probably just the stochastic nature of the run and was natural randomness. The 0.999 was our top performer though it was more or less the same performance as 0.99, but we will stick with the 0.999 value moving forward. These results indicate that a higher gamma is better as the actions should permeate deep through the history of steps and not decay too quickly. The agent should have results passed back from many actions to make a decision on its next best action. Here are the individual results for the 0.999 run:

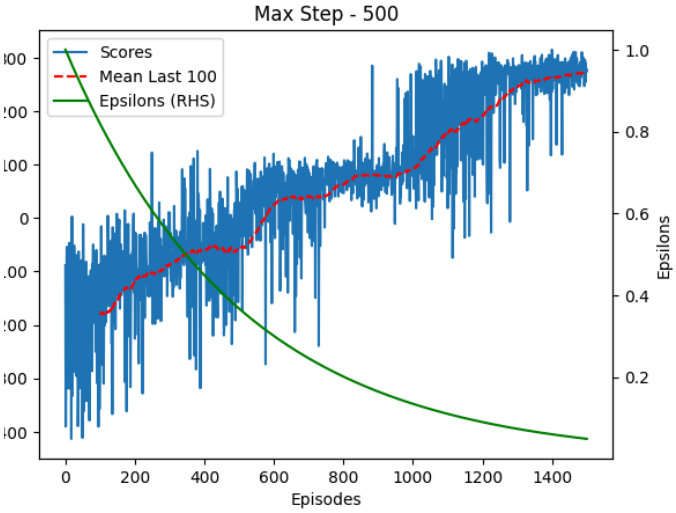


Here we see pretty steady progress in performance as shown by the average of the last 100 trials line. It is noteworthy that again we see the downward spikes late in training. A negative value < -600 is the worst value we've seen yet. Again I would attribute this to the epsilon still being around 12% at this time and maybe some catastrophic random actions are affecting these individual runs. Also noteworthy is the performance here is slightly lower though comparable to the learning rate max run (which used a gamma of 0.99). This is most likely due to the stochastic nature of a run and supports our theory that the upper values of gamma exhibit similar performance.

Lastly, we tune the number of steps hyperparameter. These are the number of steps allowed per each run. Here were the results:



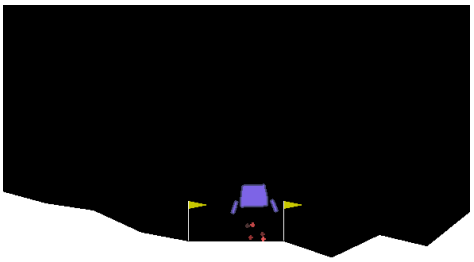
This is the most interesting of our 3 comparison charts. The 100-step results were the best performer at first and then our worst later on with minimal upward slope. The 300-step value exhibits similar performance. The theory here is that the fewer steps minimize how bad a run can be in terms of negative value as the run ends sooner. Later on, they have had less steps to train on so the performance suffers. Our 500, 1000, and 2000 performances finished in similar places. Around 700 episodes, the 1000 and 2000-steps started to pull away and paralleled each other. Also noteworthy, the 1000-step agent had declining performance between episodes 400-600 but quickly recovered. These two higher step values have learned more than the 500 step variable at this stage because they have more steps per episode. As they start learning to complete the landing, the steps variable becomes less important as the episode ends before the max is hit which explains their symmetry in the back half of the chart. As this happens, the 500 step run catches up and surpasses the performance of the other two to just barely be our best performer. One theory here is that the episodes with a lot of steps in the beginning are learning about scenarios that do not come up often and do not generalize well to scenarios the agent needs to know more about. This can lead to the weights of the neural net trying to accommodate these scenarios which will not matter much later on. The 500 step agent ends the run before then and moves on to a new episode, focusing its node weights on solving the scenarios that come up more often. Here are the results for the 500 run agent:



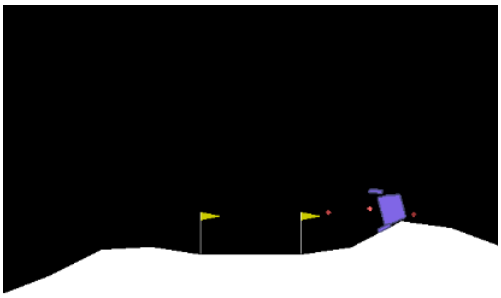
This is more or less our best performer which is good to see that the performance is indeed increasing as we improve our hyperparameters. It's an objectively better performer than our gamma run and slightly better than the learning rate run. The defaults for the learning rate run had a step size of 1000 and a gamma of 0.99 so its parameters were actually not very different from these as we saw those values perform comparably to our best performers. It's thus very reasonable that for a given stochastic run, those values would produce comparable results. With that said, we see here a big elimination of downward spikes towards the end which is great to see as our agent has really learned to avoid catastrophe. We also see the final average approaching 300, the highest we have seen yet. Our little lunar lander has indeed learned how to solve this problem and improve its performance with some hyperparameter tuning.

We now trained an agent on 3000 episodes with the optimal parameters we found above. And then test the

agent for 100 episodes with no exploration and see the results. Here is a screenshot of what our agent achieved:



Compared to our humble beginnings:



(Full gifs included in the github)
And the scores speak for themselves:

Our agent has clearly learned how to solve this problem.

Pitfalls and Problems:
Some issues I ran into was selecting the range of hyperparameters. There were parameter combinations where the learning plateaued very early and even started consistently declining after a certain point. Theoretically interesting to observe and think through but not ideal when trying to train the agent. I also had an issue early on where I had too many steps per episode. I was thinking the more steps the better to give more learning opportunities but the results did not reflect for the reasons discussed above. Epsilon decay was another area that warranted significant trial and error. I found the best results starting with epsilon equal to 1 which is a totally random agent. While counterintuitive this allowed a lot of early exploration that really overcame a nagging problem I was having of performance plateauing later in the project. Also getting the neural network setup successfully took some trial and error. Getting the error function between the target model and the current model took some research to get all the variables and outputs in the right locations. I also ran into computation speed issues as some of the experiments took 2-3 hours to run. Not unworkable but was difficult to quickly incorporate feedback back into the model to improve performance.

Conclusion:
The end result of our training and tuning is an agent that successfully solved the Lunar Lander problem. We were able to set up a Deep-Q network that trained an agent to navigate an environment with a stochastic starting point as well as a continuous state space and somewhat complex reward structure. This was a significant jump from the discrete state spaces we have been navigating so far. We also were able to see the power of deep learning for the first time. Our previous algorithms have used tabular methods such as Q-Learning that maintains a table (or Q-table) that maps state-action pairs to their respective Q-values. The Deep-Q network is a fundamentally different approach and we were able to witness its tremendous capabilities.

Further Research:
The first thing I would love to spend more time on is the shape of the Neural Network. The possibilities here are endless so after a little experimentation, we decided our 2 hidden layers with 64 nodes each was an appropriate compromise between complexity and convergence / training speed. I would also like to experiment with epsilon decay. The dynamic nature of epsilon decay and how it functions with learning and other hyperparameters made it very hard to isolate the effects of the decay speed so we decided it was more straightforward to keep it static at an acceptable speed and observe the tuning of the other parameters. Finally I would like to spend more time just making a super awesome agent. Spend lots of episodes, decay epsilon down to zero, and make a large more precise neural net to really develop an awesome lunar lander. Not really the direct goal of this exercise but always very satisfying in a reinforcement learning project to train your agent to where it's really dominating the task.

References:

- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.
- https://github.com/winter3514/cs7642-RLDM/
- https://github.com/JeremyCraigMartinez/RL-CS7642/
- https://github.com/repogit44/CS7642/blob/master/project%202/Project%202.ipynb
- https://gym.openai.com/envs/LunarLander-v2/