CS 7642: Reinforcement Learning and Decision Making
Project 2: Lunar Lander

Thomas Kim
tkim338
git hash: da801f80188e1578076634f2b54470f44e85c490

Problem

The purpose of this project is to implement and train an agent to successfully land a LunarLander-v2 model from the OpenAI gym library. Limitations to this project are implementing a reinforcement learning algorithm from scratch (but allows for using off-the-shelf neural networks to support the RL model). The agent is considered to have ”solved” the problem when it could score an average of 200 points over a 100 episode testing run. Several methods were attempted to reach a solution to this problem before finding one that worked: value-based learning, a SARSA model, and a deep Q-network.

Value-based Learning

The first attempt at developing an agent to successfully land the Lunar Lander was based on a previous homework implementation of value-based learning. This attempt quickly proved futile, as it relies on the capability to predict the next state based on the current state and action. As this gym environment has states with high dimensionality, it's difficult to predict the next state without first developing a physics model of the system, which is tangential to the focus of this project.

SARSA Model

The next attempt at developing an agent was built off of a previously developed SARSA agent from Homework 3. Running the model as-is made apparent some inherent issues with applying this type of agent to a continuous state-space model. The SARSA agent worked well for the Frozen Lake gym environment since this environment has a discrete number of states. With a limited number of possible states, a SARSA agent is generally capable of computing state-action values for every possible combination and converging to optimum values for each, within a reasonable number of episodes and run-time on a personal computer.

The Lunar Lander environment has a state-space that is continuous in six dimensions (horizontal coordinate, vertical coordinate, horizontal velocity, vertical velocity, angle, angular velocity) and discrete in two dimensions (leg 1, leg 2). In theory, this results in an infinite number of possible states, which means there is an infinite number of possible state-action pairs, even with just four discrete actions. In practice, the number of states cannot be truly infinite, due to the limited number of digits able to be stored on a computer. However, this still leaves a huge number of state-action possibilities to compute.

As a first attempt, a SARSA agent was implemented while limiting states to coordinate values rounded to the nearest 0.1, reducing the number of possible state-action values by many orders of magnitude. This resulted in requiring still too many episodes required for any meaningful optimization, so a reduction in the computation time of each episode was attempted. Because a completed agent should ideally fly straight down and quickly to the lunar surface, artificial cut-off points were implemented. If the lander flew too far left or right or if the lander took too long to complete the episode, the episode was cut short, right after assigning an additional negative reward to avoid teaching the agent to work towards the local optimum of ending every episode as quickly as possible, since reward is constantly decreasing until the end of the episode (and only if a successful landing is reached).

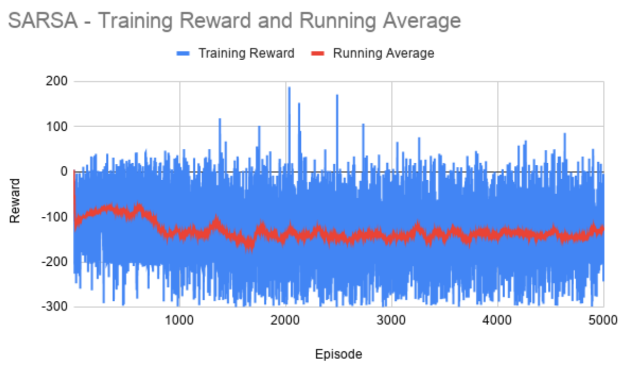

Figure 1: Training results from final iteration of SARSA agent.

The final iteration of the SARSA model was tested and its performance is shown in Figure 1. As shown, despite running 5000 episodes, the SARSA agent failed to make any real learning progress. There is notably better performance than a purely random agent (Figure 2), but this is likely due to the artificial episode limits and additional negative rewards applied to the SARSA agent that guide this behavior.

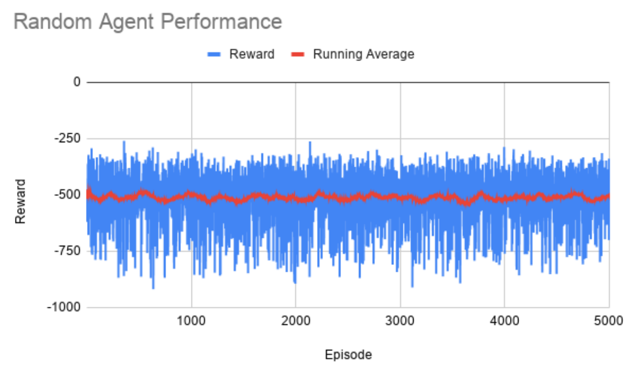

Figure 2: Reward history from a purely random agent.

Deep Q-Network

Following the failed attempt at implementing a SARSA model capable of successfully landing the Lunar Lander, a deep Q-network (DQN) was attempted, the algorithm for which was presented by Mnih et al in 2015. This algorithm was developed from the algorithm ”Deep Q-learning with Experience Replay” first presented by Minh et al in a previous paper in 2013. This original algorithm was developed from standard Q-learning and adds an ”experience replay” function to the original Q-learning algorithm. Minh et al claims three advantages that this function brings to standard Q-learning: using each experience multiple times to improve data efficiency, learning from non-consecutive samples to reduce correlation between learning experiences, and learning from data samples dominated by transitions close to the agent’s current state. This leads to an agent that learns more quickly and is less likely to diverge or get stuck in a poor local optimum, as learning is more gradual with less oscillations.

Minh et al also writes about different methods of storing and sampling experiences in the replay memory, which is finite, restricted by hardware. In this implementation of DQN, experiences were appended to replay memory until it reached a given capacity, after which new experiences were still added, but in place of existing experiences in replay memory that had a lower associated reward. The idea here being that a fully trained Lunar Lander agent would follow a relatively small sequence of states (fly straight down to the landing zone with little to no horizontal deviation). Because of this, it logically follows that this agent would glean the most learning from other experiences that end with successful landings, as it would likely not reach state-action pairs that lead far from the optimal path to the landing site.

Replay memory sampling is done by simply randomly choosing from memory a fixed set of samples. And of course, an artificial neural network assists learning by effectively correlating a complex state-space to a small set of actions.

PyTorch Implementation

The first attempt at implementing a DQN was done in PyTorch. After spending some time trying to figure out how to implement a basic neural network in PyTorch, this library was abandoned, due to initial issues and a lack of helpful documentation to troubleshoot these issues.

Keras Implementation

Keras was tried next, as it appeared to be more popular and thus more developed and refined than PyTorch.

A simple, run-of-the-mill neural network was set up, with input dimensions equal to 8, for the 8 dimensions of the environment’s state-space, and output dimensions of 4, for the 4 actions of the Lunar Lander. Intermediate layers of the neural network contained 500, 200, and 50 neurons respectively, from input (state-space) to output (actions). These values for numbers of neurons and the number of layers were chosen empirically based on a handful of trial training runs, but not a lot of time was spent tuning these values, as training differences between these were relatively minor and also obscured by noise from other sources. These values are likely not optimal for this neural network, but succeeded in training an agent in a reasonable number of episodes. The decisions to use ReLU activation for each layer and the Adam optimizer for gradient descent were based on what seemed to be common for contemporary neural networks of this scale.

Additional hyperparameters learning rate (alpha), discount rate (gamma), and replay memory sample size, were not initially fixed and instead tested across a range of values to determine rough optimums for each hyperparameter. Figures 3, 4, and 5 show the results of these tests. Based on these results, $\alpha = 0.001$, $\gamma = 0.99$, and $sample\_size = 100$ were chosen. These hyperparameters were chosen for analysis because previous implementations of RL agents showed that varying learning rate could cause either slow learning or catastrophic divergence, varying gamma could change the local/absolute optimum the agent reaches, and replay memory is a new concept not previously seen in this course so its effects were unknown.

It should be noted that some parameters led to drastic changes in reward. A learning rate of 0.01 somehow led to much worse rewards than both learning rates of 0.001 and 0.1. It seems possible that this deviation is due to chance and may be corrected with multiple training sessions. The same can be said for the reward history produced using a discount rate of 1 and the rewards produced using a sample size of 25, although in these two cases, the values producing extreme results are also on the extreme ends of the values tested, so it cannot be known if these deviations are due to only chance. More testing sessions with wider ranges of discount rate and sample size would be needed to more confidently determine these trends.

Implementation Pitfalls

During implementation of the DQN algorithm, the previous horizontal coordinate threshold for early episode cut-off first used in the SARSA model, where a Lander that ventured too far to the left or right was penalized and its episode ended

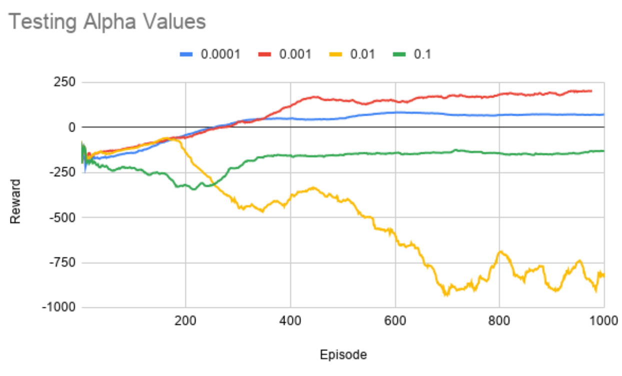

Figure 3: Rewards from varying values for α.

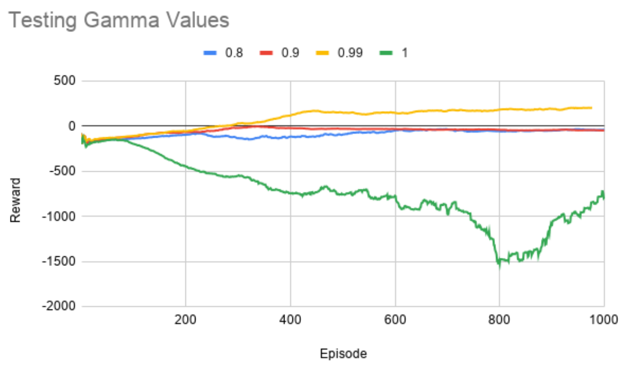

Figure 4: Rewards from varying values for γ.

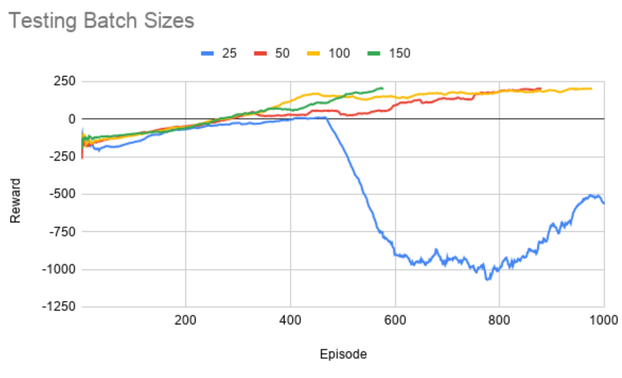

Figure 5: Rewards from varying values for replay memory sample size.

early, proved to be detrimental to learning. Through empirical observation of the heuristic solution provided by OpenAI, it was observed that there were some initial states that caused the Lander to veer far left or right without any actions taken. The optimal solution in these cases were to allow the Lander to drift off course horizontally and make corrections only at the end of the episode in order to minimize the number of thruster burns used. So, this early episode cut-off was removed in the DQN implementation.

At the beginning of testing various hyperparameters to achieve a "solved" state by the end of training, rewards were shown to continue increasing over time, rather than decreasing. It was discovered that the agent was learning to rapidly spin the Lander clockwise throughout the entire flight. This was not a one time fluke, but occurred consistently. Various hyperparameters were adjusted to attempt to manually "tune" out this behavior, but the only effective change was increasing the maximum episode length from 300 steps to 500. This change somehow caused the agent to instead spin counterclockwise. The root cause of this bug was eventually traced to a misuse of numpy.amax() in which the maximum of each state was used in place of what should have been the maximum action value.

Another implementation error caused the agent to learn normally for the first few dozen episodes but quickly stop learning and even start to reduce reward. This bug was traced down to a simple mis-assignment of total replay memory size to memory sample size.

Some training sessions were also observed to increase average reward to over 200 points, but then as training continued, average reward would drop off rapidly. This failure was attributed to over-training of the agent and corrected by imposing a limit to the number of training episodes, ending training early once the running average of reward reached 200.

Evaluation of Results

The episode reward history and 100-episode running average reward is shown in Figure 6. The maximum number of episodes to train on was set to 1000, with an early termination condition of running average reward greater than 200. This condition was met shortly before the 1000 episode limit was reached.

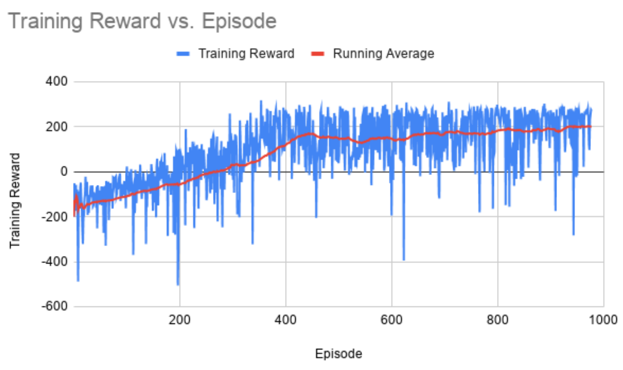

Figure 6: Reward history for training session.

A final testing session of 100 episodes using the DQN agent trained earlier showed an average reward of just over 215 for the entire session. An average reward of 200 is considered "solved" in the scope of this project. By this metric, this agent is successful. The rewards for this test session are shown in Figure 7. However, there are a handful of episodes in which the agent scored an episode reward of close to or below zero.

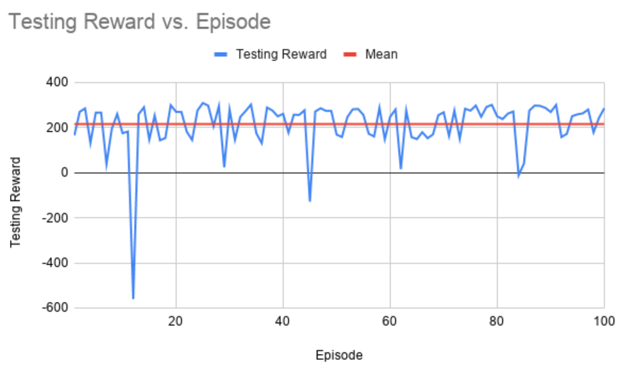

Figure 7: Reward history for testing session.

The trained agent was observed to occasionally get stuck on a slope right next to the landing site and continually fire a side thruster to attempt to fully reach the landing site, although the side thruster did not provide enough force to do so. There were also episodes in which the agent successfully landed in the landing site but then proceeded to continually fire both side thrusters, alternating between the two and remaining in place, but not completely still and thus not ending the episode. In both of these cases, reward would drop correspondingly, as each thruster burn event reduces reward. Eliminating these cases from the test session would bring the average test reward up to 233. A better trained agent may be able to avoid these situations entirely.

In future work, some way of teaching the agent if a certain action is and likely will not help (such as using the side thruster to go up a steep incline) and instead try other actions could help to avoid these situations. More training could possibly result in an agent that recognizes these cases, but it appears to be unlikely from empirical testing through many training sessions.

Some other improvements could be made in this agent. The gradient descent step of the algorithm was performed only every 10 steps of the environment to reduce runtime at the cost of losing some data to learn on. This cost was seen as minor, as the data that was passed over for learning were those transitions directly adjacent to those that were learned on and thus, likely to be very similar. However, learning on these steps, especially if they have high rewards, could produce a better trained agent.

Another possible improvement is in how transitions are saved in replay memory. In this current implementation, when a new transition is sent to be saved in replay memory, the existing replay memory list is scanned to find a transition with a lower reward than the one being added. This search is started from the top of the list and stopped as soon as a lower reward is found. A more optimal solution would be to search the whole list and find the transition with lowest reward, but this would come at the cost of increased runtime.

A test session was conducted using the heuristic agent provided by OpenAI as a benchmark. This agent outperformed the DQN agent by a significant margin, scoring an average reward of 235 over the course of 1000 episodes. However, there are cases where the heuristic agent scores well below 0, indicating that it still sometimes crashes the Lander. It's also interesting to note that if the handful of failures in the test session of the DQN agent were removed, its average reward would closely match that of the heuristic agent. A DQN agent with more finely tuned hyperparameters or neural network in general may outperform the heuristic agent here.

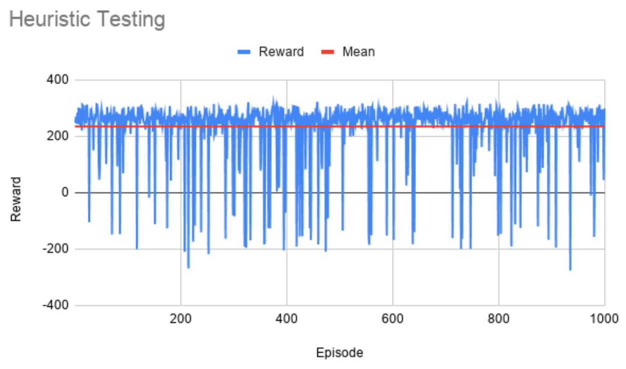

Figure 8: Reward history for heuristic agent.

Conclusions

The reinforcement learning agent developed here and trained to play Lunar Lander succeeded in scoring above a set reward threshold over a set of 100 testing episodes. However, this agent score slightly worse than the heuristic agent provided by OpenAI. A more refined DQN agent could possibly outperform this heuristic agent, but this heuristic agent could also be greatly improved. As-is, it implements just a proportional control schema to guide the Lander, while a PID-controller could be implemented without too much difficulty and likely land successfully a vast majority of the time.

For future work (if more time was available), a much better performing agent could likely be developed if the DQN agent used the heuristic agent for at least some training episodes. Even replacing the exploration step in the agent with the heuristic action finder could produce interesting results.

References

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* 2nd Ed. MIT press, 2020. url: http://incompleteideas.net/book/the-book-2nd.html.

Volodymyr Minh, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. ”Playing Atari with Deep Reinforcement Learning”. 19 (Dec. 2013).

Volodymyr Minh, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Belemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, Demis Hassabis. ”Human-level control through deep reinforcement learning”. In: *Nature* 26 (Feb. 2015), pp. 529-541.