Lunar Lander with Deep-Q-Learning

Travis Latzke

1 Introduction

OpenAI was introduced in 2015 by Elon Musk and Sam Altman. Its mission is to create a public institution to advance AI research in a way that priorities the wellbeing of the human race. To accomplish its mission, OpenAI created OpenAI Gym, which is an open source library that serves to standardize environments for reinforcement learning.

An environment can be thought of as an instance of a reinforcement learning problem. It consists of an agent, states that an agent can be in, actions that an agent can take, and rewards for taking actions in certain states. Each action allows the agent to transition to a new state with an associated reward. The problem is considered to be solved when the agent learns how to interact with its environment in a way that maximizes the total reward the agent obtains while interacting within the environment.

Some environments, like taxi-v2, contain a relatively small state space, so simple Q-learning algorithms can be used to find an optimal policy for the agent to follow within the environment. Other environments, like Lunar-Lander-v2, contain large and complex state spaces that make it impossible to compute the correct action for every state due to the number of combinations a state can be in. Therefore, approximation methods are used to reduce the computation complexity involved in calculating the optimal policy for problems with large state spaces. The intent of this paper is to address a useful technique that solves reinforcement problems with large state spaces, using OpenAI's Lunar-Lander-V2 environment as an example.

2 Lunar Lander

The Lunar-Lander-V2 environment contains a lander agent whose goal is to land in a designated area without crashing. The state that the agent can be in at any single point of time consists of eight features (x-coordinate, y-coordinate, change in x-coordinate, change in y-coordinate, orientation of agent, change in orientation, left leg has landed, right leg has landed). The agent can transition to a new state through four different actions (do nothing, fire the left engine, fire the right engine, and fire the main engine). The agent always starts at position (), but with random gradients. The agent explores the environment by taking action and then receiving feedback from the environment. The feedback includes the next state that the agent is in, the reward for taking the action, whether the agent is done, and additional metadata for the step transition. The agent is considered to be done if the agent has crashed or if the agent has successfully landed. The problem is considered to be solved when the agent achieves an average reward of 200 over 1000 trial runs.

2.1 Approach

The difficulty in solving problems like Lunar Lander stem from the size of their state space. To address this issue, researchers developed a method called functional approximation to approximate what action an agent should take in a given state. The primary reason that functional approximation is so powerful is because it uses an updatable weight vector to approximate the best action, when it's given a state as input. Its formal definition follows: (definition of functional approximation) The definition stated above is general enough to include several approximation methods, but this paper focuses on leveraging neural networks to approximate what action an agent should take. A Neural Network is a sophisticated form of functional approximation due to its inherent ability to approximate non-linear functions.

Neural Networks are typically trained in a supervised learning fashion, where the model is given examples of input data and the expected outcome of the data. The goal of the model is to generalize well enough to be able to predict the correct outcome given a piece of input data that is different from the example data that the model was trained on. This type of training can be used in conjunction with Q-learning algorithms to form an algorithm that is known as Deep Q-learning (DQR).

In DQR, the agent must create its own dataset to train the neural network that is responsible for predicting the best action for a given state. The agent creates the dataset by following the policy dictated by the neural network, which is initially set to random weights and therefore, behaves like an initial random policy. If the agent takes random actions in the Lunar Lander environment, there is a high probability that the agent will spin out of control and crash. A crash landing is the least desirable outcome and the agent is rewarded with a value of negative one hundred. Crash landings can typically be avoided if the agent were to choose better actions in previous states. Therefore, the neural network model should be penalized for choosing the sequence of actions that led to the agent to crash.

One way to penalize the model for choosing a sequence of actions that led it to crash involves propagating the end reward backwards through the action sequence by assigning each state action pair a new discounted reward. The discount factor (gamma) is value from 0 – 1 and is used to

control how far the end reward is propagated through the state-action sequence. Discounted rewards are propagated further along the sequence for gamma values closer to one and less further along the sequence for gamma values closer to zero. Thus, gamma is an important tuning parameter for discovering a policy that leads to a solution. The importance of the value of this parameter is illustrated in figure 123 in the results section. Once every state-action pair is assigned a discounted reward, the tuple (state, action, discounted reward) sequence can be used to train the neural network.

Deep-Q-Learning is powerful because the algorithm uses two neural networks (Q_{target}, Q_{actual}) to stabilize the learning process. Section 3.2 elaborates more on how each network is updated and the following section outlines the general algorithm.

2.2 Algorithm

Initialize the discount factor γ
Initialize the update rate τ
Initialize the learning rate α
Initialize experience array $E_{samples}$ to size E_{max}
Initialize target network Q_{target}
Initialize actual network Q_{actual}
Initialize number of experiences $E_n \leftarrow 0$
for $episode = 1, N$ do:
    Initialize current state s_c
    for $j = 1, M$ or $done = true$ do:
        $E_p \leftarrow$ decayed exploration probability
        $r \leftarrow$ random number from $[0 - 1]$
        if $r < E_p$ then $a \leftarrow$ random action
        else $a \leftarrow predictAction(Q_{actual}, s_c)$
        $E_{samples}[E_n \% E_{max}] \leftarrow environment.step(a)$
        Increment E_n by 1
        $b \leftarrow$ random mini batch from $E_{samples}$
        $y_t \leftarrow discountedPredictions(m, Q_{target})$
        $y_a \leftarrow actualPredictions(m, Q_{actual})$
        $minimizeMeanSquaredErrorLoss(Q_a, y_a, y_t)$
        $updateTargetModel(Q_a, Q_t, \tau)$
    end for loop
end for loop

3 Results

The Deep-Q-Learning algorithm in the preceding section is general enough to be applied to a variety of RL problems but contains several subtleties when being applied to the lunar lander environment. The subtleties were in large part a consequence of what values were chosen for the hyperparameters used to train the model responsible for choosing actions to maximize the agent's episodic reward. Therefore, hyperparameters play a pivotal role in getting the model to converge to an optimal policy.

There are numerous hyperparameters involved in the Deep-Q-Learning process. Thus, several models were trained to figure out an estimate of the best hyperparameters to use. The following subsections describe each hyperparameter along with a discussion regarding how the hyperparameter affects the model's performance.

3.1 Discount Factor

The discount factor determines how the agent weights the tradeoff between short-term rewards and long-term rewards that an action might lead to. The discount factor may consist of any real valued number in the interval [0-1]. An agent will favor long term rewards more for larger discount factor values and will favor short term rewards more for smaller discount factor values.

Long term rewards are important in the lunar lander environment because there are essentially only two outcomes for the agent. One, the agent successfully lands in the correct landing zone. Two, the agent does not successfully land in the correct landing zone. In order for the agent to land in the landing zone successfully, it must approach an optimal landing state where the agent must be stabilized (meaning that its orientation is perpendicular to the landing zone and is not significantly changing) and its velocity must be close to zero. To get to the optimal landing state, the correct actions need to be taken far in advance. Thus, a large discount factor is appropriate for training the model. However, stabilizing the agent can be viewed as more of a short-term goal in this environment. Since stabilization is conducive to landing successfully, there needs to be an incentive for the agent to focus on stabilization in certain state scenarios. A discount factor of one could potentially keep the model from learning how to stabilize in a general way.

To find an appropriate discount-factor for the lunar lander agent, I ran the Deep-Q-Learning algorithm for a range of different values for the discount factor, while holding all other hyperparameters constant. The results are shown in figure 1 below.

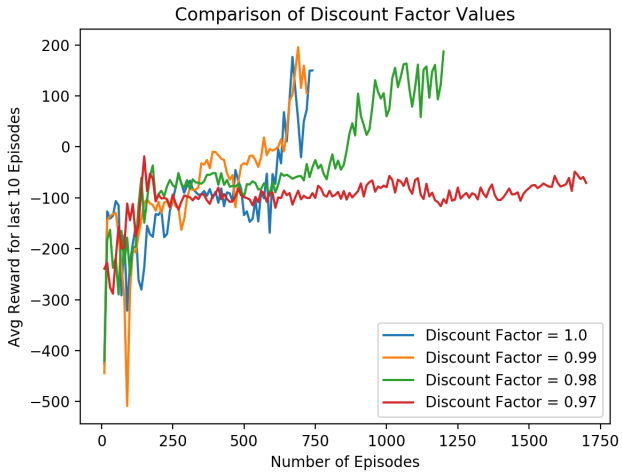

Fig. 1: Generated by keeping all hyperparameters constant except for γ. Other hyperparameters values are defined as:
$\alpha = 0.001$
$\tau = 0.001$
Initial Exploration = 0.25
Exploration decay rate = 0.001
Hidden Layers = 50

Figure 1 shows a that the model was not able to converge with a γ value of 0.97. It is also clear that the model trained with a γ value of 0.98 took more episodes to converge than the models with γ values of 0.99 and 1.0. This shows how

important the agent views long term rewards. I speculate that smaller gamma values are causing the agent to stabilize itself successfully, but the agent hovers to random states without landing because the gamma values are too small to provide an incentive for the agent to land.

3.2 Learning Rate

The learning rate, update rate, and number of hidden layers are hyperparameters that are directly involved in the neural network models use to approximate an optimal policy. In Deep-Q-Learning, two models (target, actual) are used to stabilize that training process. The learning rate is used by the actual model to determine how much error is used to train the model during the backpropagation process. The actual model is trained by first providing it with a random batch of samples that was generated by the agent. Then, the actual model and the target model make predictions on each of the training samples. The error is determined by how far off the two model's prediction are from each other. Therefore, it is important that the parameters of both of the networks converge to the same value. The update rate parameter allows the networks to converge smoother because it updates the target network in small steps. This prevents an update that overshoots or causes the target network to diverge from the approximation of an optimal policy. To reduce the complexity in figuring out the optimal relationship of these hyperparameters, I fixed the number of hidden layers to be at fifty and fixed the update rate (τ) to be 0.001. This allowed me to analyze how the learning rate affected the model's performance. Figure 2 compares different learning rates for the model.

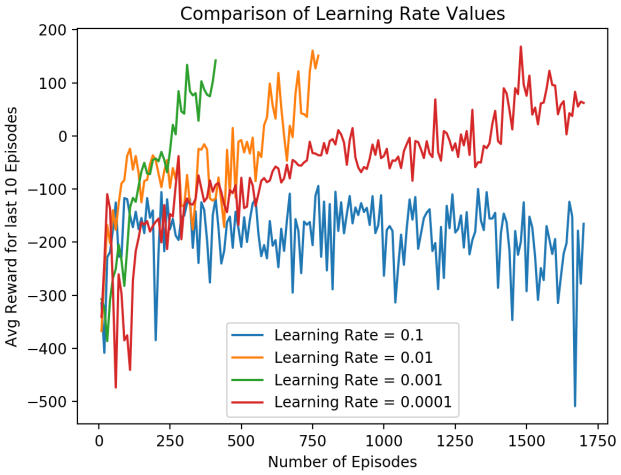

Fig. 2: Generated by keeping all hyperparameters constant except for γ. Other hyperparameters values are defined as:
$\gamma = 0.99$
$\tau = 0.001$
Initial Exploration = 0.25
Exploration decay rate = 0.001
Hidden Layers = 50

Figure 2 shows how different learning rates affect the way a model converges to a policy. A learning rate value of 0.1 was large enough to cause the model to diverge when being trained, which can be observed by looking at how the curve for 0.1 oscillates between a wide range of rewards over a sequence of episodes. It is also clear that the learning rate curve for 0.1 has no upward trend, thus the value is too large to allow the model to converge properly. The next learning rate curve of 0.01 also shows large reward oscillations, but the general pattern of the oscillations has an upward trend, allowing the model to converge for a learning rate value of 0.01. The next learning rate value of 0.001 shows an even better convergence pattern because of the reduction in oscillations and the number of episodes need for the model to converge. However, if the learning rate value is too low, the model requires a significant increase in the number of episodes to successfully train the model. This is evidenced by the learning rate curve of 0.001, which shows a slow and steady increase in the average reward obtained per ten episodes.

3.3 Exploration Rate

The exploration rate E_p defines a threshold for how frequently an agent chooses a random action. Choosing random actions is important because sometimes policies get stuck at a local maximum instead of progressing towards a global maximum. One example of a policy stuck at a local maximum that the lunar lander agent followed caused the agent to always select the 'do nothing' action. This usually allowed the agent to crash right into the landing zone instead of crashing outside of the landing zone. Since the reward was higher for crashing in the landing zone, the agent continued to crash there without learning any new actions. I observed a policy similar to the policy described in the example above whenever I set E_p to be less than 0.01.

Once I increased E_P to 0.25, I noticed that the policy improved itself from the local maximum example described above and evolved into a new policy that led the agent to hover. Eventually, the agent was able to successfully land, but was not consistent in doing so. I speculate that the inconsistent landing was due to the agent making random actions when it was close to landing, causing it to lose control. To reduce the number of random actions an agent takes in later training stages, I introduced a decay hyperparameter that caused E_p to decay to 0.01 as the number of episodes increased. The decay parameter was the last ingredient for approximating a policy that led to the solution of the problem in the least number of episodes. Figure 5 shows how different decay rates affect the convergence of a policy.

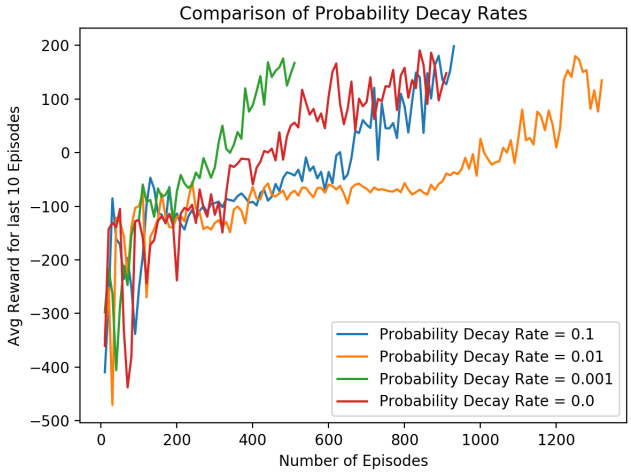

Fig. 3: Generated by keeping all hyperparameters constant except for the exploration decay rate. Other hyperparameters values are defined as:
$\gamma = 0.99$
$\alpha = 0.001$
$\tau = 0.001$
Initial Exploration = 0.25
Hidden Layers = 50

Figure 3 shows how exploration decay rates affect the number of episodes needed to train a successful model. One surprising result that I noticed was that if the Deep-Q-Learning algorithm used a decay rate of 0.0, the algorithm was still able to yield a model that achieved a score to solve the problem. This means that the agent learned how to quickly stabilize itself into an optimal state even though the agent would take a random action with a probability of 0.25.

Even though figure 3 shows how different decay rates affect how quickly a successful model is trained, it is possible that there is a high degree of variance with the results obtained. To reduce the variance, the same experiment could be ran several times to collect an average for the values described above. However, an experiment to calculate average values could take days to calculate because each single experiment may take several hours to complete.

4 Conclusion

The previous figures show how the relationship between hyperparameters can affect the performance of a Deep-Q-Learning algorithm. In fact, there are many more hyperparameter relationships that could be analyzed to improve the algorithm even further. For example, one could construct a table that illustrates how the reward changes when the discount factor and learning rate values are changed simultaneously. However, the relationship is fairly complex to study because it would involve running the experiment hundreds of times. A single experiment takes nearly two hours to run on a CPU based MacBook pro, so the experiment time could be reduced if modern GPU methods are implemented. Then it would be more feasible to find the optimal values for each of the hyperparameters.

5 Git Hash

bae2cc816701a9fbd6f95ee943f3b05cc6ded43f

References

[1] Richard Sutton, Andrew Barto. Reinforcement Learning: An Introduction. arXiv preprint arXiv:1711.05225, 2017.
[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. arXiv preprint arXiv:1705.02315, 2017.