Project 2

Bing Yang

Abstract

Building reinforcement learning (RL) agent to play simple computer games is a widely used approach to evaluate the performance of algorithms. In this report, I implement a very simple Policy Gradient (PG) RL algorithm to train an agent to play the Lunar Lander (LunarLander-v2) game provided in Gym environment. The performance over successive episodes is evaluated, as well as the effects of varying hyper-parameters on the final result. I found that 1). Learning rate and discounted factor have the biggest impact on the final result; 2). The size of the network does not have a strong impact; 3). The size of the training batch could affect the stability of the performance.

Keywords

Reinforcement learning, Policy gradient, OpenAI gym environment, hyper-parameter tuning

Introduction

Reinforcement learning (RL) is widely used to build agent to perform different tasks without human supervision. Among these tasks, training agents for playing computer games is a fast-developing field. Examples include *AlphaGO, OpenAI Five (dota2)* and *AlphaStar (StarCraft)*. One of the goals of those complex RL infrastructures is to evaluate efficiency of different algorithms. OpenAI also provides many simple games for the same purpose. In this project, I use the Lunar Lander (LL) environment to monitor the behavior of a simple Policy Gradient (PG) based RL algorithm.

Problem description. In LL environment, the goal is to train an agent to precisely control a lunar lander to land successfully on a landing pad on the moon. The position of the lander is represented by the (x, y) coordinates in a 2D space, and the landing pad is always at (0, 0).

The state of the lander is represented by an 8-dimension vector as below,

$$(x, y, v_x, v_y, \theta, v_\theta, left, right) \quad (1)$$

in which x, y are the coordinates of the lander, v_x, v_y are the velocities the specific directions, θ is the angle of the lander, v_θ is the angular velocity, and the last two variables are boolean variables indicating whether the two arms have contact with the ground.

To be able to land successfully, the lander has four discrete actions in each time step: do nothing, fire the left engine, fire the right engine and fire the main engine. A single episode is considered finished if either the lander comes to rest or crashes.

The lander gains 100 to 140 points if it moves from the top of the screen to the landing pad, with the actual reward depending on the final speed. The lander gains +100 points when lands to rest, while loses 100 points if crashes. It gains an additional +10 points if each arm contacts the ground. Each firing of the main engine costs 0.3 points. In summary, the lander can gain around 260 points upon a 'successful' landing. The problem is considered solved if the agent can end the game with 200 points or more. Details of the implementation of the environment can be found at Gym's documentation section[1].

Algorithm description. In this project, I use Policy Gradient (PG) to build the agent. PG does not estimate state and state-action values to improve policies and select future actions. Instead, PG uses parametric forms to represent policy and directly improves a policy by the gradient of the policy relative to the performance measure.

Suppose that a policy is represented as $\pi_\theta = \pi(a|s, \theta)$, in which θ is the parameter that one needs to optimize to get the best policy. Define

$$J(\pi_\theta) = E_{\tau\sim\pi_\theta}[R(\tau)] \quad (2)$$

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t \quad (3)$$

as the expected return when following the policy, in which τ represents a state-action-reward trajectory $(\dots S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1} \dots)$ and $R(\tau)$ is the regular discounted sum of rewards from time 0. The PG algorithm updates the parameter based on the following rule:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_t} \quad (4)$$

, which is a regular gradient descent problem. The main question is then how to efficiently calculate the gradient. It is not feasible to directly calculate the gradient because it involves derivatives of an expected value. However, in Sutton (2000)[2], the authors have essentially shown the equivalence between the gradient and the values below

$$\nabla_\theta J(\pi_\theta) = E[\sum_{t=0}^{T} \nabla_\theta log\pi(a_t|s_t,\theta)\, R(\tau)] \qquad (5)$$

From inspection, we can see that the gradient can be replaced by the expectation of the weighted sum of the gradients of logarithm transformed policy function over the entire trajectory, with the weights being the total expected return under the policy for that trajectory. This means that one can first sample many trajectories following the same policy and calculate the sample mean of the weighted sums defined as

$$\frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T} log\pi(a_t|s_t,\theta)\, R(\tau) \qquad (6)$$

, in which n is the number of trajectories sampled. The gradient of equation (6) relative to θ is equivalent to the policy gradient defined in (5). The gradient calculated in this way can be used to update the parameters based on (4) to search for a better policy.

The basic policy gradient algorithm that is used in this report is described in Box 1

Initiate:

1. *An initial policy (the parameters)*
2. *Hyper-parameters: number of trajectories for each parameter update (n), learning rate (α), discounted factor (γ)*

While True, do:

- *Sample n trajectories based on the current policy*
- *For each trajectory, calculate the weighted sum of logarithm transformed policy function values defined in (6)*
- *Calculate the sample mean*
- *Calculate the gradient of the sample mean relative to the parameters*
- *Update the parameters*
- *If some predefined conditions met, break.*

Box 1. A basic policy gradient algorithm

In the actual implementation, two changes have been made to the weight value in equation (6) in order to make the algorithm more efficient. First, the total discounted return from timepoint zero has been replaced by the reward-to-go defined as

$$R(t) = \sum_{s=t}^{T} \gamma^t\, r_t \qquad (7)$$

It can be shown that this replacement does not change the result[3]. Second, a baseline value is subtracted from equation (7) for each timepoint. The baseline value is just the average reward up to timepoint t. This change can make the variance of the sample mean smaller, thus make the overall performance more stable.

In this project, I use a two-layer feed-forward fully connected neural network to parameterize the policy. The dimension of the input is 8 and that of output is 4. The output is essentially the logits value for each action under the current state, and the probability of selecting action i is then calculated as

$$p_i = \frac{e^{y_i}}{\sum e^{y_j}} \qquad (8)$$

, in which y_i is the output for action i from the neural network.

In this case, the parameter updates in equation (4) can be replaced by the backpropagation algorithm in the neural network. To be able to do that, a score (or loss) function has to be defined for the neural network as the target to optimize. The loss function used is defined as

$$-\frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T} log\pi(a_t|s_t,\theta)\, R(t) - entropy \qquad (9)$$

The $entropy$ term is defined as

$$\sum p_i \log(p_i) \qquad (10)$$

, and is used as a regularization term to encourage more exploration of the parameter space to avoid early trap in a local optimum[4].

The policy gradient algorithm described above is a very basic and simple realization of the prototype of a very large family of policy optimization algorithms. Algorithms in this family include A3C, PPO and DDPG[5].

Implementation details

Python (version > 3.6) is used to implement the algorithm. The lunar lander environment is from Gym (version 0.14.0) 'LunarLander-v2'. PyTorch (version 1.2.0) is used to train the neural network. Adam is used as the optimization algorithm during neural network training.

Methods

For each experiment, 200 epochs of trajectory sampling are used, with each epoch consisting of 64 random trajectories generated by the current policy. During each trajectory generation, logits values for each of the 4 actions are generated by the neural network and are used for selecting actions stochastically for each time step. Parameters are updated once per epoch, in a batch update fashion. The rewards and average rewards in the last 100 episodes are recorded for each episode. Thus, for each agent, there are in total 12800 episodes. The

rewards and average rewards over the last 100 episodes are plotted below. The trained parameters with the best performance for each experiment are saved and used to play the game in 100 test environments, the rewards of which are all recorded for later comparisons.

To check the effect of varying hyperparameters on the final result, four hyperparameters are tuned: the learning rate (α), the discounted parameter (γ), the number of hidden units in the neural network and the size of the update batch. Tuning is done independently for each hyperparameter with other hyperparameters fixed at some prior values, due to the limitation in computation resources. This approach gives weaker conclusions than a grid search method but might still capture the general trend.

Because it is generally very slow to train neural network without GPU support, I used a heuristic iterative approach to select the prior values (which are also the best values) for each hyperparameter mentioned above. First, I searched online for general guidelines on how to select initial values for the lunar lander problem. Based on these suggestions, I picked my initial hyperparameters. Second, I varied each hyperparameters with others fixed, and replace the initial hyperparameters if the new values perform better as measured by the average scores on a predefined test dataset. Then I run everything again until performance did not change too much. In this way I got my prior values for each hyperparameter. Each run in this process has much less episodes than the actual run so might be affected by stochastic errors. However, this could be a descent approach considering the limited amount of computation resources. The prior values for each hyperparameter can be found in table 1.

Hyperparameters	
α	0.01
γ	0.99
network size	64
batch size	64

Table 1. Prior values for hyperparameters.

None of the above experiments has replicates. This is the biggest pitfalls of my design. The generalizability of the final model should be assessed in replication to avoid any bias caused by stochastic errors. Still, the dynamics of the rewards over episodes might reflect some properties of the algorithm with varying parameters.

Results

A general overview of how scores and average scores change over episodes. First, the dynamics of rewards and average rewards over last 100 episodes against episodes using the hyperparameter values in Table 1 are shown in Figure 1 below.

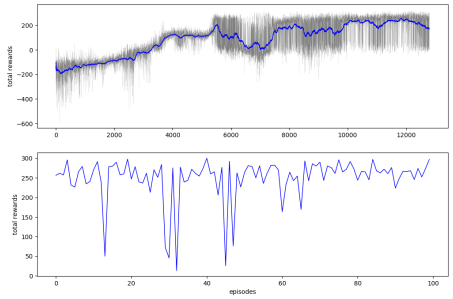

Figure 1. Rewards and average rewards over last 100 episodes for the best parameter combinations. Upper panel: performance during training. Gray line is the reward per episode and blue line is the average reward in the last 100 episodes per episode. Lower panel: test results on 100 random environments.

As we can see from Figure 1, the performance of the agent improves over episodes. The performance in each episode varies even in late stages, as can be seen from the fluctuations of the gray lines. However, the frequency of low rewards decreases with number of episodes increases, which is consistent with the trend of improvement on average performance.

One observation from upper panel in Figure 1 is that, even at late stages the performance can decrease. In fact, the average performance of the agent is not at the highest score throughout the trajectory. There are at least two approaches that we can use to prevent this. One could stop the training process if a predefined performance threshold has been reached. In this case 200 might be a good choice for the threshold. Also, we can design a schedule of decreasing the learning parameter to prevent the exploration process in the late stages. Both approaches suffer from the risk of trapping the agent in a local optimum. On the other hand, it might not be meaningful to achieve a very high performance with limited samples. As can be seen from the lower panel in Figure 1, the best agent during the training did a decent job in generalizing to random environments, with only 8 out of 100 test performance falls below 200 and scores for ~10 plays even reach ~300.

The large fluctuations in the performance of each episode throughout the training reflects the limitations of the simple policy gradient model. The baseline used in the current project is too simple to control the variance. Researchers have developed many new approaches to reduce the variance in policy optimization[6]. I could implement one of these techniques to improve the performance of my RL agent.

Parameter tuning for learning rate. To check the effect of varying hyperparameters on the final performance of the RL agent, I first vary the learning rate in the Adam optimizer. The result is shown in Figure 2. As a remark, the rewards per episode will not be shown in all downstream analysis in order to make the figures look less crowded.

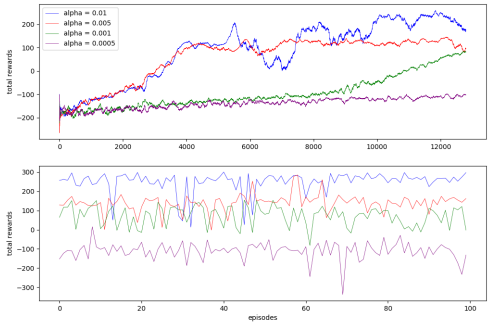

Figure 2. Performance with different learning rates. Upper panel, average rewards per episode. Lower panel, rewards on 100 random test environments.

The result in Figure 2 is quite interesting. With large learning rate ($\alpha = 0.01$), the agent quickly reaches 200 points (considered as solved). However, the improvement is not smooth. For example, the performance has a big fall in ~7500 episodes (~110 epochs). When $\alpha = 0.005$ the agent cannot even reach 200 points and stuck at ~100 points. On the contrary, even though the agent with small learning rate ($\alpha = 0.001$) does not reach 200 points, the trajectory is smooth. In addition, the performance of this agent keeps increasing at the end of the 200 epochs.

This comparison suggests that it is very important to select a good learning rate for training the agent. A large learning rate encourages exploration throughout the training process. Thus, the reward curve is usually not smooth due to the relatively large updates between episodes. For simple problem like the lunar lander, agent with a large learning rate could reach a satisfying solution quickly, as the blue line suggests. However, when the problem becomes more complicated, such as a state space with high dimensions, the agent with large learning rate might miss the best solution. On the other hand, even though agent with a small learning rate can explore the space more thoroughly, it is very slow. This is problematic if the computation resources are limited, like my situation. This is the reason why I picked the largest learning rate as the prior values.

A better strategy, as discussed above, is to schedule a decreasing plan for the learning rate, similar to the approach used in simulated annealing. This balances strengths of both exploration and exploitation.

Performance on the 100 random test environments is consistent with the highest rewards reached by different agents. In this case, the agent with the largest learning rate wins over others.

Parameter tuning for gamma. The discount factor is used to make the sum of rewards finite in non-episodic problems. In lunar lander, the episode is well defined. Nevertheless, we can still use the discount factor to reduce the influence on the weight by rewards in the far future.

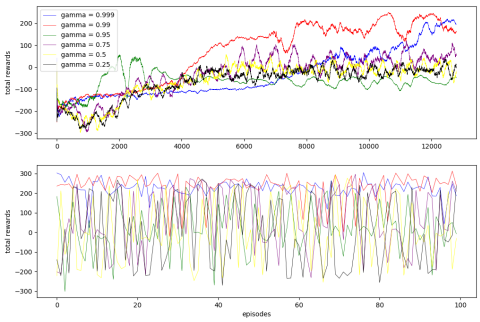

Figure 3. Performance with different discount factor. Upper panel, average rewards per episode. Lower panel, rewards on 100 random test environments.

Figure 3 shows the result for varying the discount factor. A small discount factor does not work in this scenario. This is consistent with my assumption, because the most important positive gains come from the landing moment, which is at the end of the episode. Discounting too much on future rewards thus has a detrimental effect on the performance of the agent.

However, it is less clear why agents with 0.999 and 0.99 have a big difference in performance. This

might be due to the stochastic nature of the training process.

Parameter tuning for network size. When tune the network size, the two layers share the same network size to decrease the sample space. The result is shown in Figure 4.

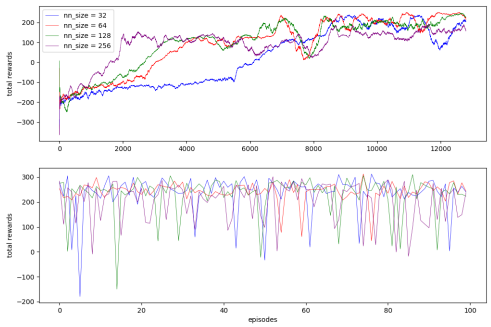

Figure 4. Performance with different network size. Upper panel, average rewards per episode. Lower panel, performance over 100 random test environments.

To my surprise, the size of the network does not have a strong impact on the final performance. The size does have an effect on the rate of reaching a satisfying solution. It takes longer for a smaller network to reach 200 points in this case. This reflects the fact that larger neural network is more powerful in fitting. The equivalence of performance might be due to the simple structure of the problem at hand.

Even though larger networks can reach a satisfying solution faster, they seem to overfit to the training data as compared to the smaller networks. When assess performances on the 100 random test environments, the network with 64 hidden units in both layers seems to have a more table output. This result suggests that it is very important to select the size of the neural network to balance the tradeoff between rate of learning and the ability of generalization.

Parameter tuning of batch size. When batch size changes, the total number of episodes also change if the number of epochs keeps constant. To accommodate this, I changed the number of epochs so that each experiment has the same number of episodes. Thus, varying batch size is equivalent to varying frequency of updating parameters. Figure 5 shows the result

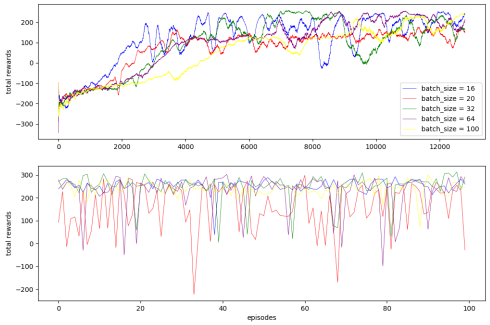

Figure 5. Performance with different batch size.

Overall, batch size does not have a strong impact on the performance. A subtle difference among different agents is that with larger batch size, the trajectory is smoother. However, the differences are too small and could be due to stochasticity of the experiments.

Surprisingly, when the batch size is 20, there is a huge amount of instability when apply the agent to 100 random test environments. The exact reason for this poor performance is not clear to me at the current stage.

Future directions

There are several changes I would like to make if I have enough time. First, the simple policy gradient based algorithm could be improved to a large extent by advanced policy optimization algorithms, such as A3C and DDPG. Second, in the current project, the best hyperparameters are picked based on the highest total rewards during training in 2000 episodes. By running more episodes, parameters with more stable training curves but lower top rewards might have better results (like the green curve in Figure 2) and I can have a better way to pick parameters.

Reference

1.https://gym.openai.com/envs/LunarLander-v2/
2.R.S.Sutton et al. *Policy Gradient Methods for Reinforcement Learning with Function Approximation*. 2000
3.https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html
4.Zafarali Ahmed et al. *Understanding the impact of entropy on policy optimization*. 2019
5.https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html
6.Pan Xu et al. *Sample efficient policy gradient methods with recursive variance reduction*. 2020