Policy Gradient Agent

Min Wang
st168632@stud.uni-stuttgart.de

Abstract

After implementing the policy gradient algorithm and its variants, including variance reduction tricks such as reward-to-go and neural network baselines, I run several experiments to test the validity of my code and analyze the results of all experiments with data visualization.

1 introduction

1.1 Policy gradient

The goal of reinforcement learning is to maximizes the following objective function $J(\theta)$, which is the expected return (cumulative rewards along trajectories τ).

$$J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[r(\tau)] \tag{1}$$

where each trajectory τ is state-action pair sequences $\tau = (s_0, a_0, ..., s_{T-1}, a_{T-1})$ with length T and each action a_t is sampled from stochastic parameterized policy π_θ.

return $r(\tau)$ is cumulative rewards along trajectories τ :

$$\begin{aligned} r(\tau) &= r(s_0, a_0, ..., s_{T-1}, a_{T-1}) \\ &= \sum_{t=0}^{T-1} r(s_t, a_t) \end{aligned}$$

Assuming Markov property, the probability of a trajectory $\pi_\theta(\tau)$ can be written as multiplying all conditional probability along the trajectory :

$$\begin{aligned} &\pi_\theta(\tau) \\ =&p(s_0, a_0, ..., s_{T-1}, a_{T-1}) \\ =&p(s_0)\pi_\theta(a_0|s_0) \prod_{t=1}^{T-1} p(s_t|s_{t-1}, a_{t-1})\pi_\theta(a_t|s_t) \end{aligned}$$

The policy gradient method in (Williams, 1992) is to take the gradient of objective function directly $\nabla_\theta J(\theta)$ and then apply gradient accent on it.

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)] \tag{2}$$

In practice, the expectation over trajectories τ in formula above can be approximated from a batch of N sampled trajectories, so the approximated gradient can be written as follows:

$$\begin{aligned} &\nabla_\theta J(\theta) \\ &\approx \frac{1}{N}\sum_{i=1}^{N}[\nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i)] \\ &= \frac{1}{N}\sum_{i=1}^{N}(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}))(\sum_{t=0}^{T-1} r(s_{it}, a_{it})) \end{aligned} \tag{3}$$

1.2 Variance Reduction

In Formula 3 the policy gradient estimator highly depends on random samples of trajectory. Due to finite sample size N, the main problem of policy gradient is high variance. The commonly used method to reduce variance are reward-to-go, discounting and baseline.

1.2.1 Reward-to-go

Considering causality property of a process: the current action of policy cannot affect rewards in the past (**?**). This yields the following modified policy gradient, where the sum of rewards here does not include the rewards achieved prior to the time step t at which the policy is being queried. This sum of rewards is a sample estimate of the Q function, and

is referred to as the "reward-to-go."

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_{it}|s_{it}))(\sum_{t'=t}^{T-1} r(s_{it'}, a_{it'})) \tag{4}$$

1.2.2 Discounting

Multiplying a discount factor γ to the rewards can be interpreted as encouraging the agent to focus more on the rewards that are closer in time, and less on the rewards that are further in the future. This can also be thought of as a means for reducing variance (because there is more variance possible when considering futures that are further into the future). We can apply discount for rewards on either full trajectory or reward-to-go.

The discounted reward-to-go is as follows:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_{it}|s_{it})(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'})) \tag{5}$$

1.2.3 baseline

Another method is to adding or subtracting a baseline b from the sum of rewards:

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau)(r(\tau)-b)] \tag{6}$$

Any function $b(s_t)$ which only depends on state is called a baseline. That is a constant with respect to τ, which is deterministic part in expectation expression so that subtracting baseline leaves the policy gradient unbiased (Xu et al., 2019).

$$E_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) b(s_t)] = 0 \tag{7}$$

The most common choice of baseline is the on-policy value function $V^\pi_\phi(s_t)$ (spinning Up, 2017)

$V^\pi_\phi(s_t)$ cannot be computed exactly, so it has to be approximated. This is usually done with a neural network, which is updated concurrently with the policy (so that the value network always approximates the value function of the most recent policy). This value function will be trained to approximate the sum of future rewards starting from a particular state.The simplest method for learning this baseline is to minimize a mean-squared-error objective.

After applying all the variance reduction tricks including reward-to-go, discounting and baseline, the approximate policy gradient now is as follows:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_{it}|s_{it}) A^\pi(s_t, a_t) \tag{8}$$

where $A^\pi(s_t, a_t)$ is called advantage function, which describes how much better or worse it is than other ac!ons on average (rela!ve to the current policy).

$$\begin{aligned} A^\pi(s_t, a_t) &= Q^\pi(s_t, a_t) - V^\pi_\phi(s_t) \\ &= \sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'}) - V^\pi_\phi(s_{it}) \end{aligned}$$

2 Small-Scale Experiments

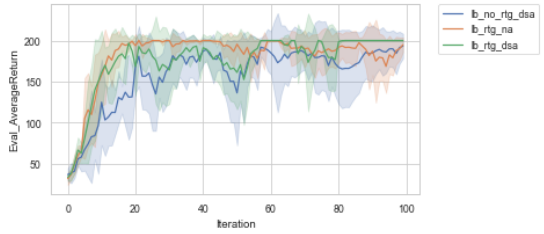
In this section, I run two small-scale experiments to test the basic implementation of policy gradient,i.e., without baseline, and get a feel of how different settings impact the performance of policy gradient methods.

2.1 Experiment 1

At first, I run multiple experiments with the PG algorithm on the simple discrete CartPole-v0 environment. To get the general intuition about the influence of hyper parameter batch size, the variance reduction trick reward-to-go and standardized advantages, I run experiments with different combination of these parameter and flags as the following:

- two different batch sizes: the small one is 1000, large one is 5000
- use reward-to-go or not
- standardize advantage or not

Figure 1b and Figure 1a shows the learning curves (average return at each iteration) for the experiments prefixed with q1_sb_(i.e., the small batch experiments) and q1_lb_(i.e., the large batch experiments) respectively. All the experiments are run on five random seeds and the curves are the mean of experiments over random seeds. Besides the mean of return, I add standard deviation around the mean donated by shaded part. The best configuration of CartPole in both the large and small batch cases converge to a maximum score of 200, which indicates that my implementation is correct.



(a) large batch size

(b) small batch size

Figure 1: Experiment 1 setting impact on CartPole-v0

The configuration in figures and command line arguments are the same in these experiments. I list all of them as follows:

- b: batch size (number of state-action pairs sampled while acting according to the current policy at each iteration). sb: small batch size set as 1000, lb: large batch size set as 5000.
- rtg: use reward-to-go; $no \quad rtg$: don't use reward-to-go
- dsa: do not standardize the advantage values; na: normalize advantages

reward-to-go: First, without advantage-standardization (dsa) I compare the value estimator: the trajectory-centric one ($no \quad rtg$) and the one using reward-to-go (rtg). Using reward-to-go the learning curves converges much faster and can achieve higher score in the end and is more stable. Furthermore, reward-to-go can reduce the standard deviation introduced by random seeds significantly.

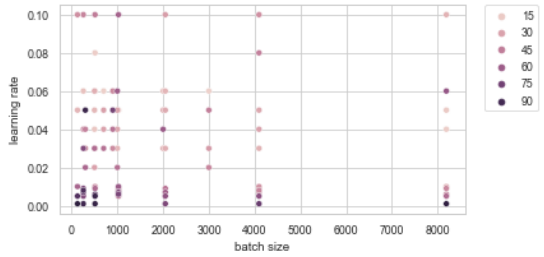
Figure 1a shows that in large batch size, the learning curve with reward-to-go converge much faster than the one without reward-to-go. Furthermore, the one with reward-to-go converge to a maximum score of 200; while the one without reward-to-go doesn't converge to a value and doesn't reach the maximum score either. Figure 1b shows that in small batch size, the learning curve with reward-to-go not only converge much faster than the one without reward-to-go but also always achieve higher score than the one without reward-to-go. Although the one with reward-to-go doesn't converge to a maximum score of 200, it fluctuates around 200 while one without reward-to-go only reach 150 after 100 times iteration. Note that both of them doesn't converge to the maximum. It is because the batch size is too small.

advantage standardization: Second, when using reward-to-go (rtg), I compare the experiments with advantage-standardization (na) and without advantage-standardization (dsa). Advantage standardization helps a lot for small batch size but it doesn't seem to help for large batch size. I think I need to do more experiments over more random seeds to test the influence of advantage standardization in the future.

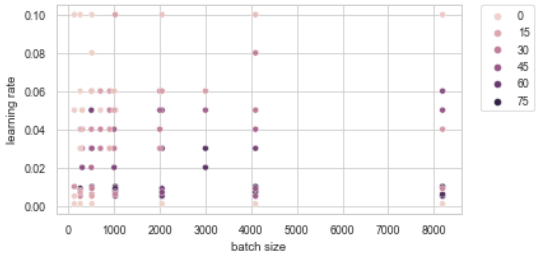
Figure 1a shows that in large batch size, the learning curve with advantage-standardization converge slightly faster than the one without advantage-standardization after 15th iteration, and achieve higher return. However, it is not stable in the end, while the one without advantage-standardization converge to maximum return stably. Figure 1b shows that in small batch size, the learning curve with advantage-standardization converge much faster than the one without advantage-standardization after 40th iteration, and achieve higher return. Furthermore the one with advantage-standardization is more stable than the one without advantage-standardization.

batch size: Third, fixing other flags, I compare the experiments with large batch size in Figure 1a and small batch size in Figure 1a respectively. The comparison order in the following is based on the order of experiments in figure 1a. In general, large batch size helps to speed up the convergence and achieve higher return, increase the stability and reduce the variance. Note that sometimes the effect is not really obvious because of other configurations.

Without reward-to-go and without advantage-standardization, the learning curve with large batch size converges much faster than the one with small batch size. Furthermore, the one with large batch size achieve much higher score than the one with small batch. With reward-to-go and with advantage-standardization, the learning curve with large batch size converges much faster than the one with small batch size. In this case, large batch size doesn't help too much about the higher average return in the end. With reward-to-go and without advantage-

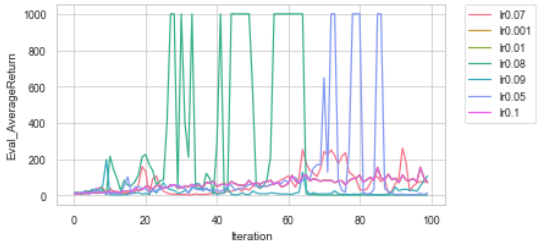


(a) convergence speed: the color indicates the number of iteration needed to reach its highest return at the first time

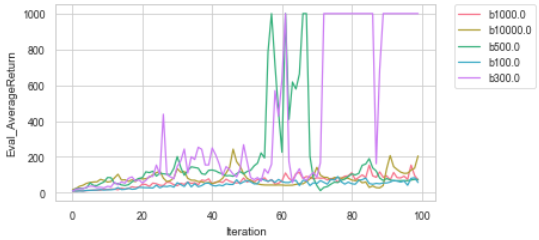


(b) convergence stability: the color indicates the number of iteration in the iteration range [70,100] where the return is larger than threshold 900

Figure 2: Experiment 2 hyper-parameter search on InvertedPendulum-v2



(a) fix batch size 1000



(b) fix learning rate 0.01

Figure 3: Experiment 2 single hyper-parameter effect on InvertedPendulum-v2

standardization, large batch size doesn't help too much about the speed of convergence before 20th iteration. However, the one with large batch size achieve much higher score than the one with small batch. Furthermore, the curve with large batch size is much more stable compared to the one with small batch.

2.2 Experiment 2

Second, I run experiments on the InvertedPendulum-v2 continuous control environment to find the smallest batch size b* and largest learning rate r* that gets to optimum (maximum score of 1000) in less than 100 iterations. I use Grid Search method to test the combination of different batch sizes and learning rates and then narrow the search space. To avoid the the curse of dimensionality of grid search, I only search the subset of predefined hyper parameter space by sampling the combinations. I also reduce the number of seeds for the parameters search to 3 to get some speedup for this scenario.

Figure 2 shows the result of grid search.

convergence speed: Figure 2a shows the speed of convergence. The size of the points indicates how many times iteration needed to reach the highest return of the learning curve at the first time.

The hyper-parameter impact is subtle and it is hard to find some general regulations.

For an arbitrary fixed batch size, when I increase the learning rate from 0.001 to 0.1, there are lots of points in the middle of learning rate range make it hard to find general rules. For instance, fix batch size as 1000, when I increase learning rate to 0.05, the number of iteration needed decreases (i.e., convergence speed increases) , however, when I increases learning rate further to 0.06, the number of iteration needed increases (i.e., convergence speed decreases).

For an arbitrary fixed learning rate, when I increase the batch size from 256 to 8192, the general trend of convergence speed maybe increasing, however, there are some points don't obey this rule at all. For instance, fix learning rate as 0.06, the speed of convergence to its maximum decrease, increase and then decreases again.

Note that Figure 3 shows that when I choose an incorrect hyper-parameter, the return of experiments is too small to reach our goal. In this scenario it's useless to analyze the speed of convergence to its own maximum return. Therefore, when I try to find the optimal hyper-parameter combination, I won't use the metric of speed to reach its own maximum at the first time.

To get the optimal hyper-parameter smallest batch size, largest learning rate, I focus on points located in top left corner. In this case, I want higher speed i.e., the number of iteration is small (in light color). Therefore the approximate range of hyper-

parameter are the following:

- batch size < 4000
- learning rate in $[0.02, 0.08]$

convergence stability: Figure 2b shows the stability of convergence. The size of the points indicates how many times iteration where the return is above manually defined threshold (here I set it as 900).

The hyper-parameter impact is also subtle and it is hard to find some general regulations just like speed.

For an arbitrary fixed batch size, when I increase the learning rate from 0.001 to 0.1, there are lots of points in the middle of learning rate range make it hard to find general rules. For instance, fix batch size as 1000, when I increase learning rate to 0.01, the number of iteration over threshold increases (i.e., convergence stability increases) , however, when I increases learning rate further to 0.03, the number of iteration over threshold decreases (i.e., convergence stability decreases). Figure 3a compares the learning curve of different learning rate when I fix batch size. In this case, the better learning rate is in the range [0.05, 0.08]. The learning rate should not be either too small or too large, otherwise return of the corresponding learning curve won't exceed 300. The comparison of learning rate 0.08 and 0.09 shows the subtlety of choosing correct hyper-parameter.

For an arbitrary fixed learning rate, when I increase the batch size from 256 to 8192, different learning rate leads to different result. For instance, fix learning rate as 0.05, the number of iteration over threshold increases (i.e., convergence stability increases) , however, when I increases batch size further to 1000, the number of iteration over threshold decreases (i.e., convergence stability decreases). Figure 3b compares the learning curve of different batch size when I fix learning rate. In this case, the better batch size is in the range [300, 500]. The batch size also should not be either too small or too large, otherwise return of the corresponding learning curve won't exceed 200.

To get the optimal hyper-parameter smallest batch size, largest learning rate, I focus on points located in top left corner. In this case, I want higher speed i.e., the number of iteration is small (in light color). Therefore the approximate range of hyper-parameter are the following:

- batch size $[500, 4000]$
- learning rate in $[0.02, 0.08]$

conclusion: Figure 2 shows the difficulty of hyper-parameter search and I can conclude that different combination of hyper-parameters are independent of each other. Correct hyper-parameters such as batch size and learning rate are also task-dependent. The performance of policy is affected by batch size and learning rate at the same time. If one of hyper-parameter is not tuned properly, it will limit the overall performance even though the other parameters are tuned properly. Therefore, I need to automate the hyper-parameter search using systematic hyper-parameter tuning method (This part will be my future work).

return_threshold	batch size	learning rate
77	4096.0	0.010
75	8192.0	0.010
73	256.0	0.009
71	4096.0	0.008
69	1024.0	0.010
69	1024.0	0.008
68	8192.0	0.005
66	1024.0	0.009
66	8192.0	0.006
64	3000.0	0.030
63	3000.0	0.020
63	4096.0	0.007
59	2048.0	0.030
55	500.0	0.050
55	2048.0	0.005
55	4096.0	0.030
54	512.0	0.010
54	2048.0	0.009
53	1000.0	0.020
53	2048.0	0.050
51	2048.0	0.007

Table 1: ranking experiment by return over threshold

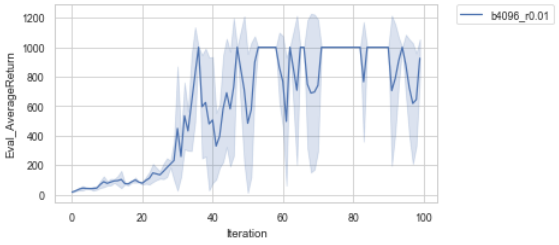


Figure 4: Experiment 2 optimal hyper-parameter

In Table 1, I rank the experiments by re-

turn_over_threshold. The result shows the optimal combination of hyper-parameter setting is using batch size 4096 and learning rate 0.01. Note that it is only the best among my hyper-parameter search space and it doesn't mean the optimal is exactly $bs = 4096$ and $lr = 0.01$, because of inefficiency and proximity of grid search. However, it is a good approximate. Here I remove the experiments whose performance are bad because the complete table is too long.

Figure 4 shows the learning curve of optimal hyper-parameter. Compared to all curves in Figure 3, the return of optimal hyper-parameter is more stable. There is no sharp oscillation in optimal setting. The shaded part of the curve indicates the standard deviation around the mean. Note that the policy performance fluctuate around 1000 slightly and the standard deviation is not small sometimes. It may be because other settings are not optimal). Also note that when I add nn baseline, the curve doesn't change much.

3 Complex Experiments

In this section, I test the implementation of state-dependent neural network baseline, which is trained along the policy gradient update. For all the remaining experiments, I use reward-to-go estimator.

3.1 Experiment 3

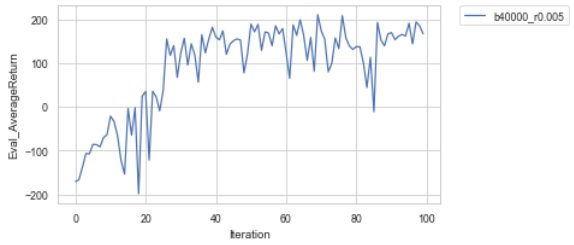
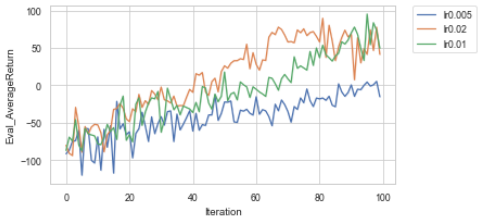


Figure 5: Experiment 3 on LunarLander

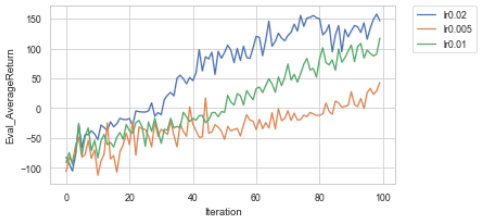
First, I run a experiment to learn a controller for LunarLanderContinuous-v2 environment to test and debug the baseline implementation. Due to the long training time of this experiment on my laptop, I only run this experiment once and don't use multiple random seeds. In Figure 5 I achieve an average return of around 180 by the end of training, which indicates the validity of my implementation.

3.2 Experiment 4

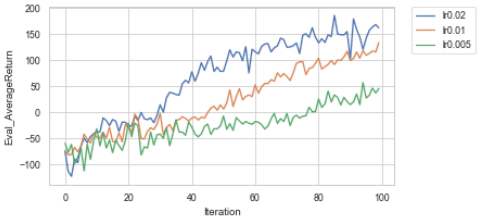
Then I run multiple experiments on to learn a controller for the HalfCheetah-v2 benchmark en-



(a) fix batch size 10000

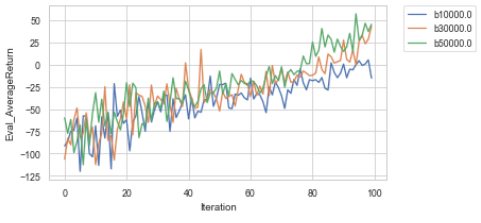


(b) fix batch size 30000

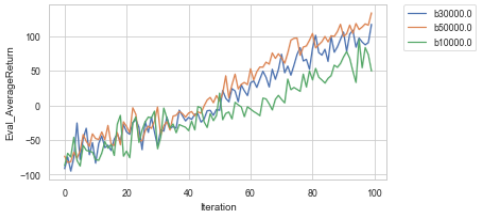


(c) fix batch size 50000

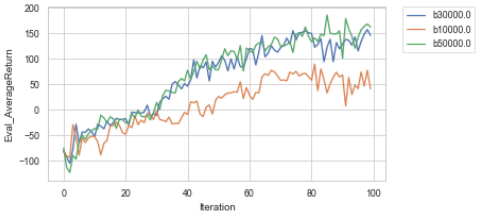
Figure 6: Experiment 4 hyper parameter search on InvertedPendulum-v2



(a) fix learning rate 0.005



(b) fix learning rate 0.01



(c) fix learning rate 0.02

Figure 7: Experiment 4 hyper parameter search on InvertedPendulum-v2

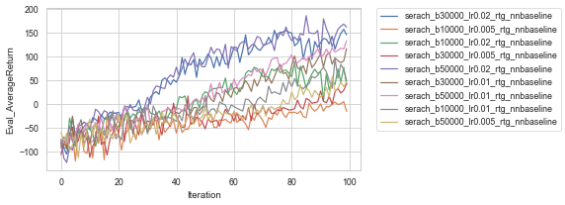


Figure 8: Experiment 4 search on HalfCheetah-v2

vironment with an episode length of 150. This is shorter than the default episode length (1000), which speeds up training significantly. I search over batch sizes $b \in [10000, 30000, 50000]$ and learning rates $r \in [0.005, 0.01, 0.02]$. Once I have found optimal values b* and r*, I use optimal hyper parameter to find the influence of reward-to-go and neural network baseline. These experiments are average over five different random seeds.

To figure out how the batch size and learning rate affected task performance, I use control variable method i.e., fix one hyper parameter and compare different settings of another one. Figure 6 shows the result when I fix batch size. In general, larger learning rate leads to faster convergence but less stability. The return score depends on only proper learning rate instead of higher one. In particular, Figure 6a shows that for small batch size, too large learning rate may not achieve higher return.

Figure 6 shows the result when I fix learning rate. In general, large batch size leads to higher return. Figure 7a and Figure 7b shows that if the learning rate is too small, large batch size doesn't contribute to speeding up convergence; while in Figure 7c if learning rate is large enough, larger batch size can speed up convergence. Figure 7c also shows that if learning rate is large, the difference of increasing batch size from 30000 to 50000 is small, which means the impact of batch size is reduced so that I don't need to continue to increase batch size in this scenario.

To find the optimal hyper-parameter I plot all the curves in Figure 8, which shows the optimal combination is $b = 50000$ and $r = 0.02$

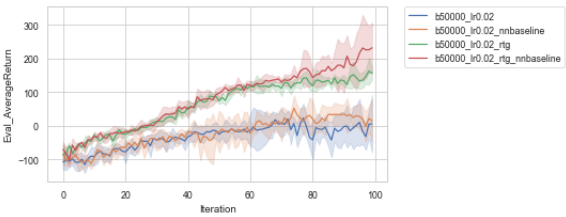


Figure 9: Experiment 4 optimal on HalfCheetah-v2

reward-to-go: In Figure 9, the orange and blue curve donate experiments without reward-to-go and red and green curves donate the experiments with reward-to-go. It is obvious that reward-to-go can accelerates the convergence and increase the final return. Note that here reward-to-go decrease the variance without baseline however, it doesn't decrease variance with baseline (in fact it increase the variance), which is strange situation. I may need to do more experiments about it in the future.

baseline: In Figure 9, the blue curve donates experiment without reward-to-go and baseline, when I add baseline on it, the final return increases a little. Similarly, you can also compare the green and red curves to get the same conclusion. The baseline doesn't help to reduce the standard deviation.

4 Conclusion

Although I finished implementation, there are still some problems need to be solved in the future, for example, check and debug the correctness of implementation of advantages standardization and reward-to-go.

Possible Extensions are as follows:

- automate hyper-parameter search
- Implement paralleling of sample collection across multiple threads and compare the difference in training time.
- Implement GAE-λ for advantage estimation to speed up training
- Compare single-step PG and multi-step PG
- Implement more complex algorithm i.e., SRVR-PG-PE
- Integrate policy gradient into the ADVISER framework

References

OpenAI spinning Up. 2017. Intro to policy optimization.

Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

Pan Xu, Felicia Gao, and Quanquan Gu. 2019. Sample efficient policy gradient methods with recursive variance reduction. *arXiv preprint arXiv:1909.08610*.