REINFORCEMENT LEARNING FINAL PROJECT

Ruhao Xin

New York University Shanghai

ABSTRACT

When the reinforcement learning is becoming more and more powerful, its structure is becoming more and more complex as well. One typical example is the huge number of hyper parameters. In this paper we do the experiments on SAC model. We analyze the hyper parameters γ, the discount rate, α, the exploration-exploitation trade-off, and a hyper parameter that controls whether this agent does deterministic or stochastic evaluation. Moreover, we also analyze the influence from neural structure to the model, like the layers and nodes of a neural network, replay buffer size and batch size. We not only successfully reproduce the similar results from other relevant papers, but also get more information about how the SAC model works base on these hyper parameters.

1 INTRODUCTION

Reinforcement learning is the study of how an agent can interact with its environment to learn a policy that maximize the rewards for a task. Since its birth, the reinforcement learning algorithm is becoming more and more powerful. Initially, in the tabular case, there are already a lot of reinforcement algorithms. One is the dynamic programming, which uses the concept of bootstrapping to estimate each state-action pair's Q-value. However, it requires huge amount of information of the environment, which can be unrealistic when being applied in our real lives. Another algorithm is called Monte Carlo Method, which doesn't require any kind of information of the environment, and uses actual rewards to calculate Q-value instead of bootstrapping. Because of these factors, this algorithm must wait until one complete episode is simulated, which actually slows down its update. Then a breakthrough made by Watkins & Hellaby (1989) introduced one of the most important reinforcement algorithm, called Q-learning, by applying the off-policy TD control to enable a faster convergence, which becomes the key concept of nearly all later reinforcement learning algorithms which requires the update of Q-value.

The power of reinforcement learning started to grow rapidly when it combines with neural network. For example, deep Q-network (DQN) by Mnih et al. (2013) can be used to play a series of Atari games and achieve pretty good rewards compared to human players. More recently, one of the biggest improvement is the appearance of Alpha-Go Zero from David et al. (2017), which is based solely on reinforcement learning and is able to achieve super-human performance.

However, as the development of reinforcement learning, the structure of its model is becoming more and more complicated. One typical point is the huge increase of the number of hyper parameters. In tabular Q-learning, there are only 3 hyper parameters we need to consider: ϵ (used in ϵ-greedy), α(used as step-size parameter) and γ(used as discounted rate). But in DQN, the number of hyper parameters can be very overwhelmed. The hyper parameters come not only from the neural network itself, but also from some extra functions which can make the model perform better. The huge number of hyper parameters in deep reinforcement learning algorithms leads to a fact that the models depend more and more on the hyper parameters. This dependence brings two things: 1. The performance of reinforcement learning algorithms can be heavily influenced by the choice of hyper parameters. 2. By analyzing some hyper parameters, it is possible for us to get more understanding about how things work in some models. As a result, the hyper parameters can be a very potential research field.

So in this paper, I will focus on this kind of dependency to analyze the role of hyper parameters in a specific reinforcement learning model SAC. I will firstly investigate how different hyper parameters

can influence the performance of SAC, and then do more in-depth analysis to get more understanding of how things work in SAC based on some specific hyper parameters.

2 Related Work

2.1 What is SAC?

To demonstrate what SAC looks like, we need to go through how the deep reinforcement learning develops and take a look at how the hyper parameters change through the process.

One of the earliest deep reinforcement learning is called DQN, which was used by Mnih et al. (2013) to play a series of Atari games successfully. This model follows the tabular Q-learning algorithm to do the policy improvement without the step size parameter α, but to make it work well in neural network, there are two new features: 1. Replay buffer, which is used to store previous experiences of the agent. 2. Mini batch, which is very necessary in neural network to do batch gradient descent. As a result, here we have two new hyper parameters, the length of replay buffer and the size of mini batch.

One restriction of DQN is that this model can only work in discrete action spaces. However in our real lives, the action spaces are more likely to be continuous. As a result, a new version of DQN, called DDPG from Lillicrap et al. (2019), appeared to solve environments that have continuous action spaces. Based on DQN, it implemented a new neural network which outputs action value according to the input state value. Besides, it introduced a new hyper parameter ρ to do polyak averaging of target network, which is done by the following formula,

$$\phi_{target} \leftarrow \rho\phi_{target} + (1-\rho)\phi$$

where ϕ_{target} is the target network that is used to calculate the estimate, and ϕ is the network that is required to be updated by gradient descent.

However, neither DQN nor DDPG can solve the problem of overestimation of Q-value. That's the reason why Fujimoto et al. (2018) occurs. The improvement version is called TD3. Inherited from core algorithm and hyper parameters of DDPG, TD3 improves the performance of deep reinforcement learning by implementing two Q-network. When calculating targets, it uses the minimum value from two Q-networks.

Then Haarnoja et al. (2018) comes which is named as SAC. Based on TD3, it introduces a new way to do the balance between exploration and exploitation, called entropy regularization. its value is controlled by a new hyper parameter α. According to the paper, by increasing α, it can result in more exploration, which can accelerate learning later on. Compared to ϵ-greedy, this kind of control can give more flexible to balance exploration and exploitation, which is more likely to prevent the policy to converge to a bad local optimum. It can be accomplished by the following formula:

$$-\alpha log \pi_\theta(a'|s')$$

In conclusion, SAC is an integration and improvement of previous deep reinforcement learning algorithms. As a result, by doing this analysis, we find out many hyper parameters that can be very important in SAC model: size of replay buffer, size of batch size, polyak parameter ρ and entropy control parameter α. In later sections, we will focus on these hyper parameters to investigate their influence to SAC model.

2.2 Relative Analysis

In this section we will discuss how other relative papers analyze the performance of deep reinforcement learning models, which is very likely to give some insights for us to do more things on SAC model.

In Haarnoja et al. (2018), we realize that it has already discussed how the hyper parameters, ρ and α, can influence the performance of SAC model. From the paper, it concludes that the performance of SAC is sensitive to the choice of α so that its value should be tuned very carefully. Comparatively, the range of suitable ρ is relatively wider, but it still matters. Low value of ρ can lead to instability while high values will make the training slower. Besides this, the paper also compares the

performance between deterministic and stochastic evaluation. The conclusion is that deterministic evaluation perform better than stochastic evaluation.

Actually, the performance of a model is not just limited by hyper parameters. There are many other factors that can have influences. For example, according to Henderson et al. (2017), a paper talking about some general problems about deep reinforcement learning, the structure of neural network, environments, the implementation of codes and the use of random seeds can also have big influence. Therefore, these factors can bring a lot of instability to the model. As a result, this paper implies that only analyzing the performance of the model is not adequate enough. We need to think about more methods to analyze a model.

Fortunately, Chen et al. (2021), talking about a new deep reinforcement learning called REDQ, provides some new sights on how other methods can be. In this paper, it not only discusses the performance, but also analyzes the average q-bias and the std of q-bias to get more understanding of this model. According to the paper, the average bias can tell us how much the Q-network is overestimating or underestimating the model. The std tells us how uniform the bias is across different state-action pairs.

Therefore, to have better understandings of the influence of hyper parameters on SAC, we will extend the comparisons of Haarnoja et al. (2018). We will not only try to reproduce the results from this paper, but also compare the performance, average q-bias and std of q-bias of other hyper parameters that we think matter as well in the SAC.

3 METHODS

Although we plan to do the research on SAC model, we will actually use the source code from Chen et al. (2021). The reason is that: Firstly, REDQ is based on SAC so it is very convenient for us to transform a REDQ model into a SAC model. What we have to do is just changing some hyper parameters of REDQ model. Specifically, we set utd ratio as 1 and the number of Q networks as 2. Also, we don't use the adaptive SAC entropy. Besides these, all the other hyper parameters stick to the default value of REDQ. Secondly, the REDQ source code has a very complete function of calculating the average q-bias and std of q-bias. As a result, it can save us a lot of time from writing new codes for them.

We will assign two seeds to each hyper parameter to run on two mujoco environments 'Ant-v2' and 'Hopper-v2'. For each seed, we will run 1000 episodes, and each episode has 1000 interactions with the environment. Because the laptop cannot handle this kind of long time of the training process, we will use the hpc from NYUSH to help us. The results will be drawn by matplotlib library.

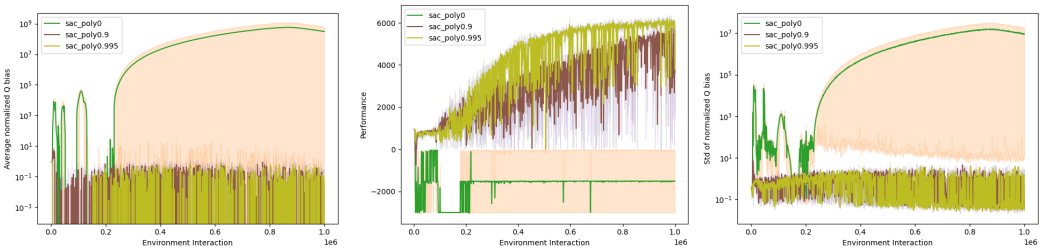
4 RESULTS OF COMPARISON

4.1 POLYAK PARAMETER ρ

From the Figure 1 and 2, we can get the results of the influence of polyak parameter on SAC. There are some very noticeable points. Firstly, when the polyak value is 0, the performance of the model is the lowest. The reason can be seen in the other two figures: in this case, the average bias and the std of bias is much higher than ρ = 0.9 and 0.995, which makes the model very unstable. This result perfectly fits with Haarnoja et al. (2018) that low polyak value can lead to the instability of the model. Secondly, in the Ant environment, when ρ is 0.995, at first its performance is lower than ρ = 0.99. However, in the future its performance starts to surpass ρ = 0.99. This also fits with Haarnoja et al. (2018) that high polyak value can make the training slower but can finally get a better policy. However, in the Hopper environment, this observation is not very obvious. Although finally ρ=0.995 is better then ρ=0.99, we don't see a significant faster speed of ρ=0.9 at the very beginning compared to ρ=0.995.

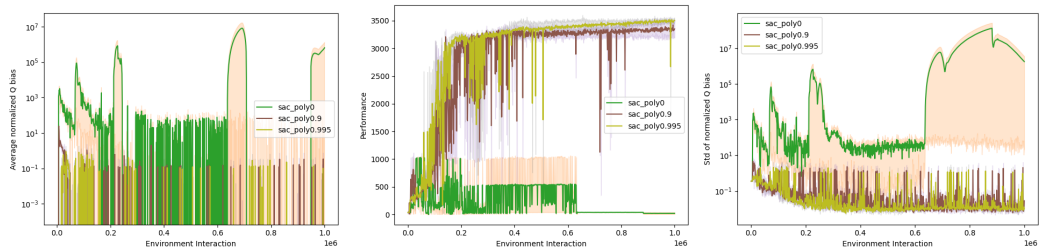
4.2 REWARD SCALE α

From the Figure 3 and 4, we can get the results of the influence of α on SAC. We can see that the role of α is much more complicated than ρ. One complication is that for different environments,



(a) Average Bias, Ant-v2 (b) Performance, Ant-v2 (c) Std of bias, Ant-v2

Figure 1: Performance, mean and std of normalized Q bias for SAC in Ant environment based on different values of polyak parameter ρ=0,0.99 and 0.995.



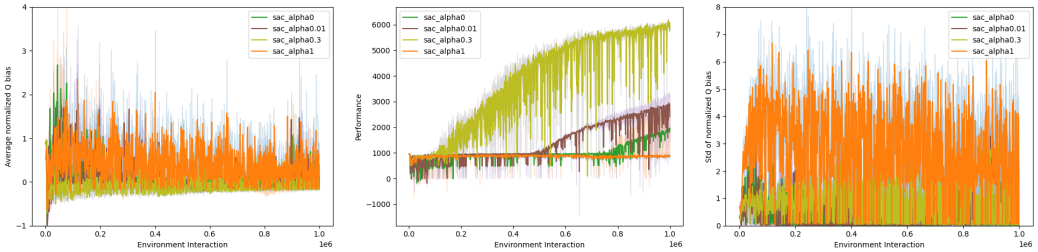
(a) Average Bias, Hopper-v2 (b) Performance, Hopper-v2 (c) Std of bias, Hopper-v2

Figure 2: Performance, mean and std of normalized Q bias for SAC in Hopper environment based on different values of polyak parameter ρ=0,0.99 and 0.995.

the optimal α value is different. In Ant-v2, the optimal choice is 0.3 and its performance is much better than other choices. However, in Hopper-v2, the optimal choice becomes 0 instead of 0.3, and the difference of performance between different α is much smaller than Ant-v2, which is kind of surprising because we don't expect the α=0, meaning totally exploitation policy, can work better than other α values. In my opinion, the explanations of this can be the following: 1. The Hopper environment is suitable for greedy algorithm, and 2. The episode is not long enough because I notice that the gap between α=0 and α=0.3 is becoming smaller and smaller as the episode goes on. No matter what the reason is, it illustrates that this model is very sensitive to α. We can see it more clearly by looking at the average bias of q-values in different models. In Ant, the good α value tends to underestimate the q-values while in Hopper, the good α value tends to overestimate the q-values more. Overall, the result actually fits with Haarnoja et al. (2018) that we require careful tuning for α value in different environments.

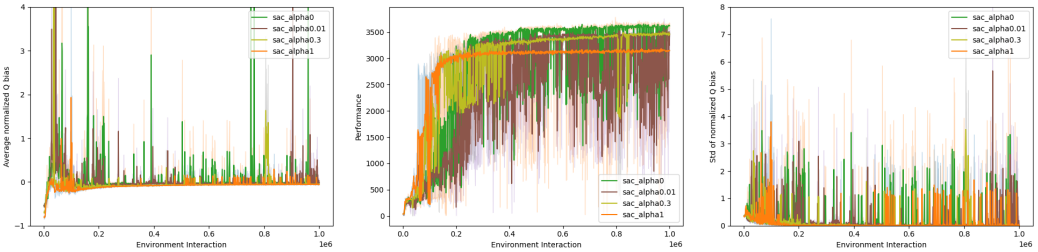
Besides the difference, we can also find similarities between two environments. One common point is that in different models, the relationship between average bias of q-values and the performance seems to be similar. For Ant, we can see that when α=1, its average bias is very unstable and in most time surpasses the value of 0, which means that in this case, compared to other models, it often overestimates the Q-values more. As a result, we can see that the performance of α=1 is the lowest. For Hopper, we can also see this, where the average bias of Q-value for α=1 is the lowest, which means that it underestimates the q-values most among all models. Therefore, its performance is the lowest as well.

In my opinion, the reason is that if the α value is too high, the model will take too much time to do the explorations, which can lead to the result that even if in one state, there is already a dominant action, the model will still tend to choose other options. This behaviour can negatively influence the performance of this mode. This result also fits with Haarnoja et al. (2018), where it says that high value of α can make the model fail to exploit the reward signal.



(a) Average Bias, Ant-v2 (b) Performance, Ant-v2 (c) Std of bias, Ant-v2

Figure 3: Performance, mean and std of normalized Q bias for SAC in Ant environment based on different values of α=0,0.01, 0.3, 1

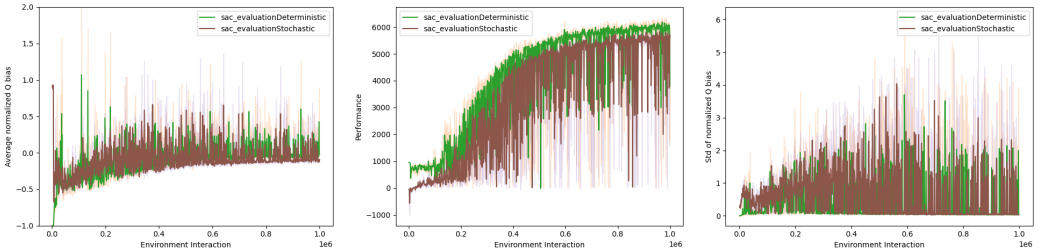


(a) Average Bias, Hopper-v2 (b) Performance, Hopper-v2 (c) Std of bias, Hopper-v2

Figure 4: Performance, mean and std of normalized Q bias for SAC in Hopper environment based on different values of α=0,0.01, 0.3, 1

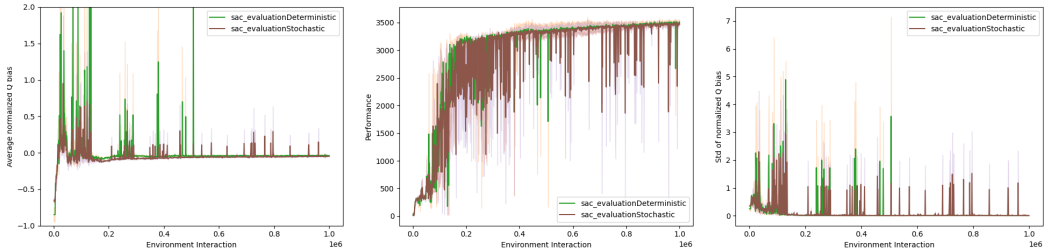
4.3 Deterministic VS Stochastic Evaluation

From Figures 5 and 6, we can get the results of the influence of evaluation method on SAC. We do it under the category of hyper parameters because according to the source code of Chen et al. (2021), the evaluation method can also be determined by a hyper parameter. There are some intriguing points from the figures. Firstly, it shows that deterministic evaluation has better performance than stochastic evaluation, which gives us the same result as Haarnoja et al. (2018). Secondly, although both evaluation tend to underestimate the q-values for both environments, the stochastic evaluation underestimates them more than deterministic evaluation, which actually reinforces the point made by Haarnoja et al. (2018), making it more solid. In my opinion, the reason is that the SAC algorithm is used to optimize a policy network, which is a normal distribution determined by its mean and std. After the training, the mean and the std both reaches to their optimal values. As a result, because in deterministic evaluation we use the mean of the normal distribution as the next action, the performance of deterministic evaluation is better than stochastic one.



(a) Average Bias, Ant-v2 (b) Performance, Ant-v2 (c) Std of bias, Ant-v2

Figure 5: Performance, mean and std of normalized Q bias for SAC in Ant environment based on different evaluation methods



(a) Average Bias, Hopper-v2 (b) Performance, Hopper-v2 (c) Std of bias, Hopper-v2

Figure 6: Performance, mean and std of normalized Q bias for SAC in Hopper environment based on different evaluation methods

4.4 Structure of Neural Network

According to Henderson et al. (2017), the structure of a neural network can also influence the performance of a model. Therefore in this subsection, we will analyze the influence of different **1.** nodes and layers of neural network, **2.** replay buffer sizes and **3.** batch sizes. Moreover, we will also analyze the time the model spends under different structure of neural network above. Because we use 2 nodes to run the experiments, the running time is the average running time of two nodes.

4.4.1 Wide VS deep networks

From Figure 7 and 8, we can get the results of the influence of different numbers of layers and nodes on the model. Firstly we notice that the (128, 128, 128, 128) neural network is more conservative than others because in both environments this model underestimate the q-values more than others, which negatively influence its performance in first 1 million interactions. However, this is not the end. Intuitively, we think the deeper a neural network is, the better performance it will get in the future. In this experiment, although its performance is lower than other two models, the gap between them is becoming smaller and smaller, which makes it possible that if we continue the interactions, finally the performance of it can be higher than other two models.

Secondly, we notice that in both environments, although (512,512) structure learns slower than (256,256) at the beginning, finally it can have better performance than (256,256) structure.

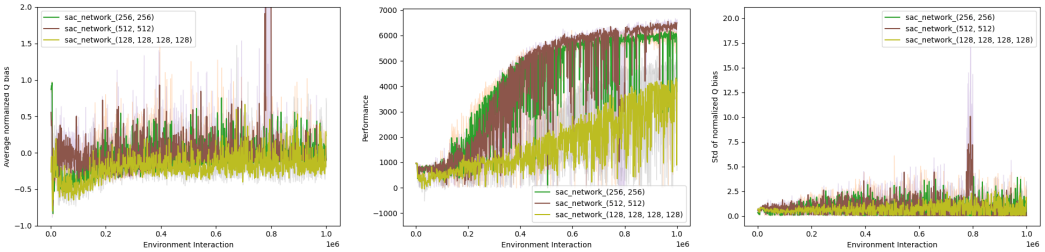
Hidden layers	Average Ant running time(h)	Average Hopper running time(h)
(256,256)	11.39	9.40
(512,512)	24.63	25.79
(128,128,128,128)	10.74	9.03

The running time of different neural structures also gives us some interesting points. We can see clearly that the more nodes a neural network has, the more time a neural network requires to spend. This consequence makes sense. As a result, it is necessary to make a balance between the performance and the running time. Too many nodes can lead to better results in the future while requires a lot more time.

4.4.2 Replay buffer size

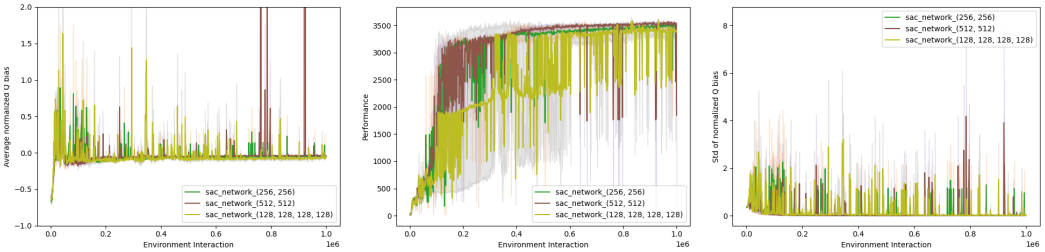
From Figure 9 and 10, we can get the results of the influence of different replay buffer sizes on the model. Firstly, we notice that if the replay buffer size is too small, the performance of it becomes very terrible. The reason is that the small replay buffer sizes means that the model is very forgetful so that it only cares the most recent data. As a result, the model will stick to a very bad local optimal and cannot improve its performance any more.

Secondly, we notice that when the replay buffer size is 50000, its std of bias is bigger than 1000000, which makes it more unstable so that its final performance is smaller. Another interesting point is that although smaller replay buffer size can learn faster in the beginning, finally its performance is surpassed by larger replay buffer sizes.



(a) Average Bias, Ant-v2 (b) Performance, Ant-v2 (c) Std of bias, Ant-v2

Figure 7: Performance, mean and std of normalized Q bias for SAC in Ant environment based on different numbers of layers and nodes

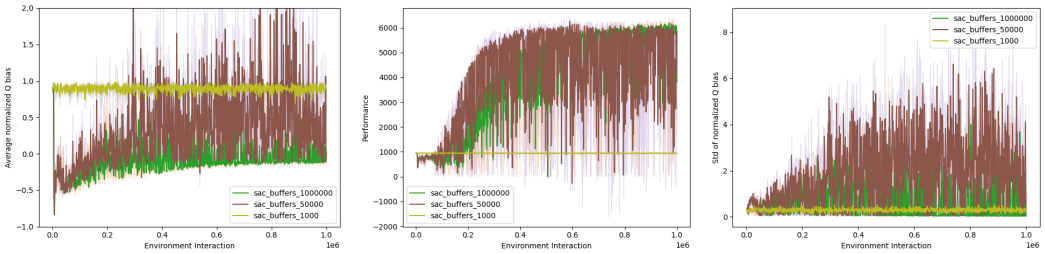


(a) Average Bias, Hopper-v2 (b) Performance, Hopper-v2 (c) Std of bias, Hopper-v2

Figure 8: Performance, mean and std of normalized Q bias for SAC in Hopper environment based on different numbers of layers and nodes

Replay buffer size	Average Ant running time(h)	Average Hopper running time(h)
1000	1.60	0.53
50000	11.13	10.04
1000000	10.83	9.06

The running time table illustrates that if the replay buffer size is too small, its running time can be significantly small. However, when the replay buffer size is huge, there is no big difference of running time between different replay buffer sizes. As a result, I guess the function between replay buffer sizes and running time is a log function. This also tells us that bigger replay buffer is always greater. However, because the storage of computers are limited, we also need to balance the replay buffer sizes of a model.

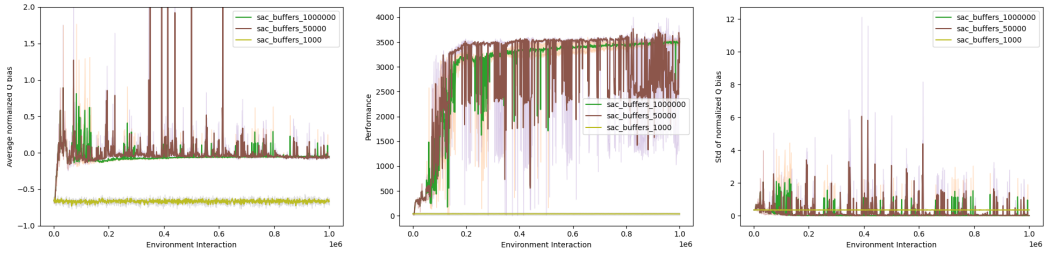


(a) Average Bias, Ant-v2 (b) Performance, Ant-v2 (c) Std of bias, Ant-v2

Figure 9: Performance, mean and std of normalized Q bias for SAC in Ant environment based on different replay buffer sizes

4.4.3 BATCH SIZE

From Figure 11 and 12, we can get the results of the influence of different batch sizes on the model. We notice that when batch size is 256, its average bias of q-values is closer to 0 compared to other



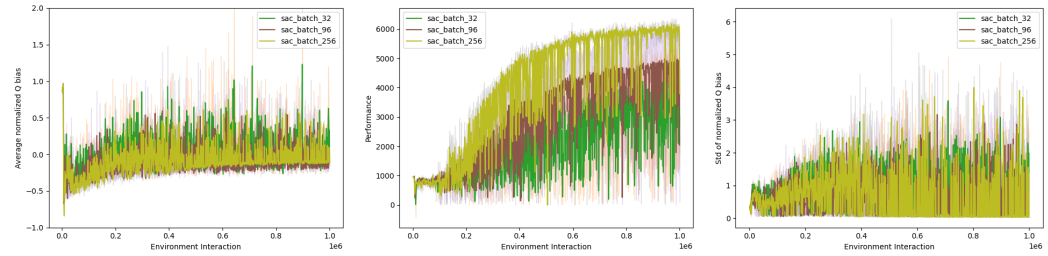
(a) Average Bias, Hopper-v2 (b) Performance, Hopper-v2 (c) Std of bias, Hopper-v2

Figure 10: Performance, mean and std of normalized Q bias for SAC in Hopper environment based on different replay buffer sizes

batch sizes, which means that it is more stable than other batch sizes. As a result, this kind of stability can lead to the better performance for batch size is 256. The reason is that bigger batch sizes means the computer consider more cases when updating, which can intuitively increase its accuracy and performance.

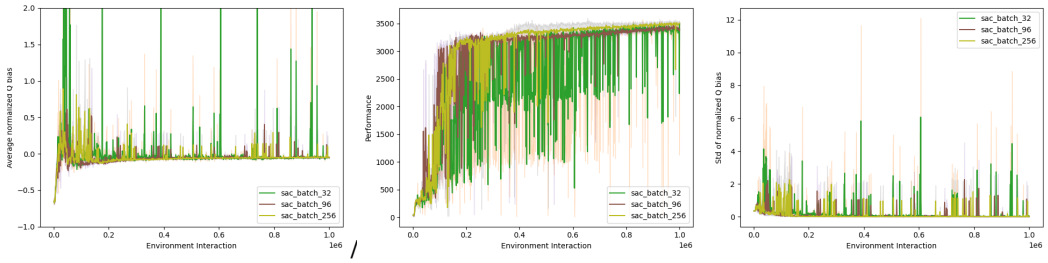
Batch size	Average Ant running time(h)	Average Hopper running time(h)
32	6.94	5.93
96	7.76	6.72
256	10.75	9.06

From this running table, we can see that the bigger a batch size is, the more time the model requires to do the training. As a result, we need to balance between the performance and running time. Too big batch size can lead to better performance while requires more time to run.



(a) Average Bias, Ant-v2 (b) Performance, Ant-v2 (c) Std of bias, Ant-v2

Figure 11: Performance, mean and std of normalized Q bias for SAC in Ant environment based on different batch sizes



(a) Average Bias, Hopper-v2 (b) Performance, Hopper-v2 (c) Std of bias, Hopper-v2

Figure 12: Performance, mean and std of normalized Q bias for SAC in Hopper environment based on different batch sizes

5 MORE ON HYPER PARAMETERS

Now we would like to try to do a more in-depth analysis of the mujoco environments to get a better understanding of how things work based on some hyper parameters. In this section, I will train a SAC agent on the Hopper environment and get the distributions of Q-value and entropy, which are two of the most important elements in SAC algorithm. To do this, after finishing the training, I output the whole SAC agent class into a file by using the 'pickle' library. By doing this, when I need the Q-values or entropy in the replay buffer, I just need to reload the pickle file and get them.

5.1 Q-VALUES

Figure 13 demonstrates the distribution of Q-values of SAC model. To analyze it, we need to focus on a hyper parameter γ, which is the discount rate of the model. In this case, we assign γ as 0.99.

Firstly, we notice that the range of q-values is from -75 to 375. I think this range makes sense because we notice that in current code, the total numbers of interactions with the environment is equivalent to the replay size buffer, which means that the replay buffer stores all interactions and doesn't abandon anything. As a result, it is likely that the very negative q-values represent the exploration behaviour of the agent.

Secondly, we notice that most q-values are concentrated near 350. To analyze this, we generate 100 episodes to test the agent. In each episode, we run 1000 interactions for the agent. As a result we can get the average performance for 100 episodes, which is 3469. Because this performance doesn't use discount rate, we can then get the average reward for taking actions is $3469/1000 = 3.469$. Based on this, we can calculate the average q-value of this agent is $Q(s,a) = G = r + \gamma r + \gamma^2 r + ... = \frac{r}{1-\gamma} = \frac{3.469}{0.01} = 346.9$, which fits with this distribution.

5.2 ENTROPY

Figure 14 demonstrates the distribution of entropy value of SAC model. To analyze it, we need to focus on a hyper parameter α, which is assigned as 0.2 in this experiment.

Firstly, we notice that the range of entropy is from -31 to 10. The existence of very negative entropy demonstrates that this state is very horrible, which occurs when the agent does the exploration. This result is also similar with q-values, where the low values also occur in exploration behaviours. This close relationship can be partly explained by the structure of SAC model: when calculating target q-values, it requires the actual q-value plus the entropy value. When doing update, we not only need to update q-network, but also need to update policy network. In this way, the bad states can make both q-values and the policy entropy be very negative, which reinforces the badness of this state.

Secondly, we notice that most entropy values are concentrated near the value 0. If we multiply the α value with these entropy, the discounted entropy will be even closer to 0, which, compared to the Q-value we get, is negligible. It means that in most cases, when calculating the target value, we just use the q-value. This result demonstrates that after the training of SAC, this model can do exploitation for most states, which fits with the structure of SAC model that it learns both the mean and the std for a Normal distribution. Because of this, during the training process, the std can be smaller and smaller, which makes the selection of actions be more deterministic to selecting the mean value.

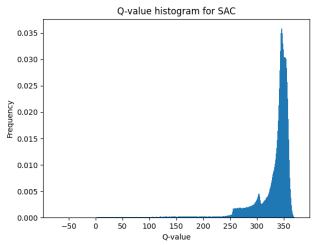
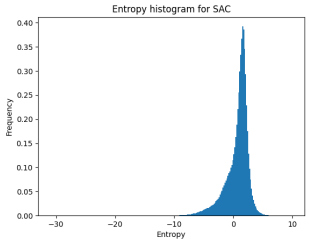


Figure 13: Histogram of Q-values of SAC agent



/

Figure 14: Histogram of Entropy of SAC agent

6 Conclusion

The contributions of this paper are as follows. (1) We not only reproduce the results from Haarnoja et al. (2018), but also do some deeper analysis on the model's average q-bias and std of q-bias, like Chen et al. (2021) do, which allows us to get more understandings of the influence of different hyper parameters. (2). We investigate the influence of different structures of neural network, like layers and nodes of neural network, replay size buffer and batch size, and get similar results with Henderson et al. (2017) where structures of neural network also matters when considering the performance of a model. (3). By analyzing q-values and entropy of an agent, we get more understandings of how SAC model works. Overall, this paper provides a new method to analyze a reinforcement learning algorithm. As a result, we think other people can do similar way to analyze some other reinforcement learning algorithms, like REDQ in Chen et al. (2021). Moreover, these experiments are not very perfect because we can see the plots are very rough. In my opinion, it is because we just use two nodes to train the model. This result implies that nodes also matters in the performance of a model, which also fits with Henderson et al. (2017). Therefore, if this method can be trained on more nodes, like 4 or 5, the plots we draw are very likely to be much smoother. In this case, we may get some more information about the model.

References

Xinyue Chen, Che Wang, Zijian Zhou, and Keith W. Ross. Randomized ensembled double q-learning: Learning fast without a model. *CoRR*, abs/2101.05982, 2021. URL https://arxiv.org/abs/2101.05982.

Silver David, Schrittwieser Julian, and Simonyan Karen. et al. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017. doi: 10.1038/nature24270. URL https://doi.org/10.1038/nature24270.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018. URL http://arxiv.org/abs/1802.09477.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *CoRR*, abs/1709.06560, 2017. URL http://arxiv.org/abs/1709.06560.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Watkins and Christopher John Cornish Hellaby. Learning from delayed rewards. 1989.