An Implementation Guide and Empirical Analysis on the Soft Actor-Critic Algorithm

Che Wang

December 20, 2018

1 Introduction

Soft Actor-Critic (SAC) algorithm is one of the most recent and powerful deep reinforcement learning (DRL) method, published this year (2018) by the Berkeley AI group [1]. SAC is a model-free, off-policy method with high sample efficiency and robustness over random seeds in high-dimensional continuous control problems.

In this project, we present a refined Pytorch implementation designed for easy learning, along with a list of implementation tips, and a set of empirical analysis on the SAC algorithm, aiming for the following three objectives:

- Implementation guide: discuss some critical parts of the SAC implementation and how they might alter the algorithm.
- Additional empirical analysis: experiment and analyze key components of SAC that are not emphasized in the SAC paper but can significantly impact performance.
- Enhancing the algorithm: experiment and analyze how recent advances in DRL can be combined with the SAC algorithm to boost its performance.

Note that originally the project was proposed to focus on giving a tutorial of SAC, but some good tutorials are already available online and it's quite redundant and boring to simply reproduce a tutorial. So we decided to instead focus more on the empirical analysis and enhancement sections. For the implementation part, we give a densely-commented Pytorch implementation, as well as a list of small tips that complement the existing resources in order tos help interested reader implement the algorithm with less struggle.

2 Background

In the past several years the field of reinforcement learning has been growing fast, since the successes of deep-Q-learning (DQN) [2] which achieved human-level performance in the Atari games in 2013, many new algorithms have been developed and applied to various tasks including high-dimensional continuous action space tasks and the state-of-the-art performance keeps being updated every year.

The SAC algorithm is related to many of the older algorithms in various ways: just like in DQN, SAC uses Q-networks to estimate the value of state-action pairs. DQN can only be used for discrete action space tasks, since it requires the selection of an action that has the highest Q-value among all actions, this is very difficult when the action space is large, since it's necessary to iterate through all actions' Q-values in order to find the action with max Q-value. In continuous action space tasks, selecting such an action is even harder. SAC is designed to tackle continuous action space tasks and manages to circumvent that problem by using a separate policy network that learns to output actions that would give the highest Q-network values. This design essentially allows SAC to use related works on DQN easily in the continuous action space tasks. And in fact SAC does use a modified version of DQN, Double DQN (DDQN) [3] to improve its robustness. Note that although using a policy network to generate actions sounds like a kind of policy-gradient[4], SAC's policy update depends heavily on the Q-networks, and doesn't involve an advantage value. In most policy gradient methods, such advantage value is calculated using a combination of rewards and value predictions from a critic value network that takes in states and output value estimation. In SAC there is a value network exist but its purpose is more on helping the Q-networks learn, instead of directly working together with policy learning.

This close relation with DQN makes SAC an off-policy algorithm, allowing it to use past data freely and not worry about a trust-region. Compared to popular on-policy methods such as PPO[5] and TRPO[6], that have to constrain the policy update to stay within a trust region and can only use the most recent data for updates, SAC thus has a huge advantage. At time of update, with a default replay buffer of size 1 million, SAC can continue to extract information from past data long after their generation, while a method like PPO only uses the tiny amount of 2000 data points (default) from the last training iteration.

Another key component in the SAC algorithm is the entropy term that is mainly used to balance exploration and exploitation. From the analysis in the SAC paper we know the entropy's parameter α is a hyper-parameter that is equivalent to inverse of the reward scale. A large α leads to encouraged entropy, thus more exploration, a very large α thus makes the policy nearly uniform and stops learning. A very small α, on the other hand will make the policy greedy and deterministic, makes it easy to stuck in local optimal. Exploration with entropy fall in the category of action space noise in [7], entropy is not the only way to achieve adequate exploration, and it's possible to use other exploration methods to achieve similar results, for instance, a perturbed policy[7].

3 Implementation Guide

In this section we discuss some interesting and important things in the SAC implementation. Since there are already good explanations and tutorials on how SAC works [8], here we do not intend to reproduce an explanation of all the elements in SAC, but aim to focus on several details that might cause trouble for interested reader who are learning the algorithm. This guide serves as a complement material, instead of the main material, to the SAC paper and the OpenAI Spinup documentation.

3.1 Recommended Way of Learning

It is recommended to first have a glance at the paper [1] to get a basic idea of the algorithm, the paper not only introduce all the core elements of SAC, but also gives some theoretical analysis as well as nice ablation studies on the algorithm. Interested reader should then carefully look at the Spinup documentation [8] before implementing the algorithm. The documentation includes explanation on how the networks interact with each other and has a concise pseudocode section that is very helpful.
The reader then can try implement the algorithm, and read our implementation (see last subsection of this section for links) for a line-by-line reference on network updates that is consistent with the OpenAI Spinup doc and the pseudocode, as well as consult this section of the report for problems to pay attention to and how to avoid them.

All equations in this section are refering to the OpenAI Spinup SAC doc and pseudocode.

3.2 Hidden Sizes

Hidden layer size has a huge impact on the performance of SAC, if you previously learned PPO or TRPO, then you should notice that SAC uses larger network, which is by default 2 hidden layers with 256 neurons each. Consult the empirical analysis section for details.

3.3 Network Initializations

Hidden layers in the networks are initialized with a bound that is inverse proportional to the square root of number of input. And last layers in the networks are initialized with a small weight, this might not be a critical thing, but can have some effect on performance.

3.4 Log Std

SAC uses state-dependent log std for action sampling. This means that your policy network, given a state, should output an action mean, as well as a log std value that decides how much action noise you want for taking action in this state. The log std has been limited to between -2 and 20 to increase stability. We can do a rescale or simply clip the log std to achieve this. In our implementation we used rescale to be consistent with OpenAI tensorflow SAC code, but the official SAC code used clipping, both works. Not implementing this constraint might lead to environment instability.

3.5 Entropy Alpha and Reward Scale

As mentioned in the background section, the α hyper-parameter in the entropy term is very important. It directly affect the objective of the agent:

$\pi^* = argmax_{\pi} E_{\tau \sim \pi}[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot|s_t)))]$

Note that the best α reported in the SAC paper is different for each Mujoco environment, so when we run experiments, we might need different hyper-parameters for each of the environment to reproduce the results in the paper.

3.6 Sampling New Actions and Update Policy

A crucial part of SAC is sampling new actions and update networks based on these new actions, see line 12 of pseudocode:

$y_v(s) = min_{i=1,2} Q_{\phi,i}(s, \tilde{a}) - \alpha log_{\pi_\theta}(\tilde{a}|s)$

Here although the state comes from the data collected (sampled from replay buffer), the action $\tilde{a}$ is newly sampled from the policy. So during the udpate, after sampling a batch of data from the replay buffer, we should first put these data into our policy and get the new actions $\tilde{a}$ from it. Also note that since we are using the reparameterization trick, the actions have to be part of the computation graph so that we can backpropagate gradient through it. See pseudocode line 15:

$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} (Q_{\phi,1}(s, \tilde{a}_\theta(s)) - \alpha log_{\pi_\theta}(\tilde{a}_\theta(s)|s))$

When we generate the actions $\tilde{a}$, make sure computation graph doesn't break when going from the parameters of the policy network to the new actions.

Another potential problem here is that we are doing gradient ascent, so we should pay attention to how we make the loss function when we use optimizers.

3.7 Reparameterization Trick

Can be very confusing if you get to implement this the first time, but essentially what we need to do is implement the following line:

$\tilde{a}_\theta(s, \xi) = tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \xi), \xi \sim N(0, I)$

There are potentially many ways of doing this, in our implementation, when given state s, the policy network does the following:

1. feed through the network to generate 2 outputs, the mean and log std of your action. The mean here hasn't gone through a tanh function yet, so its range is arbitrary. Note that to use the log std in a distribution, you need to change it to std, by taking its exponential.

2. Now we generate a sample action, from a normal distribution that uses the mean and std from the policy network as its mean and std. Make sure the gradient graph is intact.

3. Now we put this sample action through tanh function. Again make sure the gradient is intact. And now you have the action.

If you want deterministic action, simply take the mean output from step 1, put it through a tanh and use that as your action.

Note: Not implementing this step correctly can lead to all kinds of strange errors.

3.8 Action Limit

Different Mujoco environments have different action limits. For HalfCheetah it's $(-1, 1)$, but for other environments this can be something different. Since our policy network outputs a tanh-range value, you will need to rescale the action so that it fits the environment's limits. The easy way is to simply rescale the action before putting it into environment. This can save us some trouble, compared to adding rescaling to the network which makes it part of the computation graph.

3.9 Pytorch Implementation

We have made a Pytorch implementation of the SAC algorithm, with focus to the following key points:

1. Consistent with OpenAI Spinup documentation: OpenAI has a list of well-written tutorials on popular RL algorithms, but they don't have Pytorch version code, but only Tensorflow. Making the code consistent with the OpenAI documentation makes help our readers to easily get additional references for their study as well as use their fully-tested utilities for experiemnts.

2. A concise and minimal implementation. To make the code friendly to DRL beginners, the code should be as simple as possible, without redundant structures.

3. Extensive documentation: write comments whenever needed, to make sure every line of the code is easy to understand. In critical parts of the algorithm there are line-by-line reference to the Spinup doc.

4. Bug-free and Comparable performance: although the code should be minimal, we still want to have all the necessary components correct and working so that the code has equivalent performance compared to the results in the SAC paper.

The official code for SAC is relatively complicated, and OpenAI's code is very nice but only has Tensorflow version. For the above mentioned features, interested students who use Pytorch will find my code very easy to read and to work with.

Our implementation is available at: `https://github.com/watchernyu/spinningup`

See our repo readme page for where exactly the SAC code is located, as well as guide on setting up environment and running experiments.

The main loop of the algorithm (data collection and network update) is in sac_pytorch.py file, and helping functionalities, including replay buffer, tanh normal distribution, network modules, etc. are in the core.py file. Except for logging utilities, all the SAC components can be found in these two files.

4 Additional Analysis on SAC performance

There is a set of hyperparameter analysis and ablation analysis in the SAC paper [1]. But there are also some aspects of the algorithm are not discussed adequately by the paper. So here we perform 2 sets of additional analysis on the SAC algorithm:

1. experiment on how a relatively large hidden layer size contribute to the success of SAC.
2. experiment on how replay buffer can be modified to affect performance of SAC.

We use the Mujoco robotics control benchmarks as test environments for the analysis [9]. We test on HalfCheetah and Ant environments, Ant is a more difficult task than HalfCheetah and the agent can terminate early in Ant if performed bad actions. The results and analysis show that both components are critical to the SAC algorithm, and reveal potential direction of research for further improvement.

4.1 Hidden Layer Size

Compared to some other popular algorithms in DRL, SAC uses relatively wider networks. In Proximal Policy Optimization (PPO) algorithm, a very popular and robust algorithm published in 2017, the policy network for Mujoco tasks has 2 fully connected layers with size of 64 [5], and another early work Trust-Region Policy Optimization (TRPO) reported a single hidden layer with size of only 30 for its continuous action space policies [6]. Another family of methods, Deep Deterministic Policy Gradient (DDPG) [10] and Twin-Delay DDPG (TD3) [11] used relatively larger networks of 2 layers with 300, 400 neurons each. In terms of methods that are not gradient-based, Evolution Strategy for DRL[12] reported 2 hidden layers with 64 neurons each, Genetic Algorithm in [13] used 2 hidden layers with 256 neurons each. These architectures all have different network sizes, note that smaller networks tend to make training computation faster, but not necessarily give better sample efficiency. Now if we look at SAC's official implementation, it has 2 hidden layers with 256 hidden neurons each. Is this network size critical for SAC's learning process? This analysis is not given in the original SAC paper, and in fact, such analysis is rarely seen in previous DRL papers. So we experiment with it and present our analysis here:

Figure 1 shows total average return of evaluation rollouts during training for SAC baseline, which uses hidden size of 256, compared to SAC with hidden size of 128 and 64. Number of hidden layers is 2 for all 3 variants, for all networks. Solid curve indicates mean across 3 random seeds, and shaded area show maximum and minimum values. Each experiment is run for 1 million timesteps (datapoints). Although the data usage is the same, total computation time is different: for 3 seeds, SAC-256 takes about 18 hours, SAC-128 takes about 11 hours, SAC-64 takes about 8 hours.

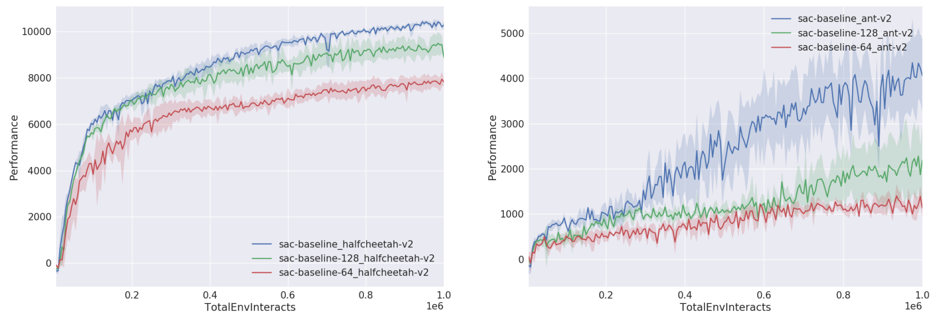

Figure 1: SAC performance comparison on HalfCheetah and Ant Mujoco environments, on different hidden sizes.

From the results we can see that in both environments, larger networks tend to give better performance, especially in the more difficult Ant environment, where SAC-256 achieves about 2 times the performance compared to SAC-128, and 4 times compared to SAC-64, at 1M data. This shows that network size is critical for SAC to achieve high data efficiency, although SAC will have to run slower since it requires relatively large networks. But overall, the benefit of a larger network is very significant.

The above results also pose a question: is it fair to compare DRL algorithms tested using different architectures? And for some of the early methods that used small networks, such as PPO and TRPO, will their performance on Mujoco be significantly improved by simply using a larger network? It will be an interesting research direction to find out how much performance can we gain in DRL by simply focus on enhancing the network structure, especially when now the community is getting more interested in the data efficiency problem.

4.2 Replay Buffer Size

SAC is an off-policy algorithm, which means it can use data generated early in the training process without disturbing its learning. In the SAC paper, a replay buffer size of 1e6 is used, consistent with many previous off-policy methods, such as Deep Q-Learning (DQN) [2] and Double DQN [3]. Storing up to 1 million data allows the algorithm to use much more old data compared to on-policy methods, but it uses these data in a uniform fashion. This poses the question: will very old data have a negative impact on the performance since they are generated by the early stage policy which performs very badly? Treating different data in the replay buffer differently has been explored in [14], where a prioritized buffer is designed for DQN and it prioritize over data points that are less expected by the agent. Here we perform an simple analysis by reducing the buffer size of SAC, and see if if smaller replay buffer can impact the performance in a positive way, the reasoning being that a smaller buffer forces the algorithm to focus on the more recent data points, which are collected by an improved agent and are more relevant to the most up-to-date learning process. Note this analysis is also not present in the original SAC paper.

Figure 2 shows total average return of evaluation rollouts during training for SAC baseline, which uses replay buffer size of 1e6, compared to SAC with buffer size of 1e5 and 1e4. Solid curve indicates mean across 3 random seeds, and shaded area show maximum and minimum values. Each experiment is run for 1 million timesteps (datapoints).

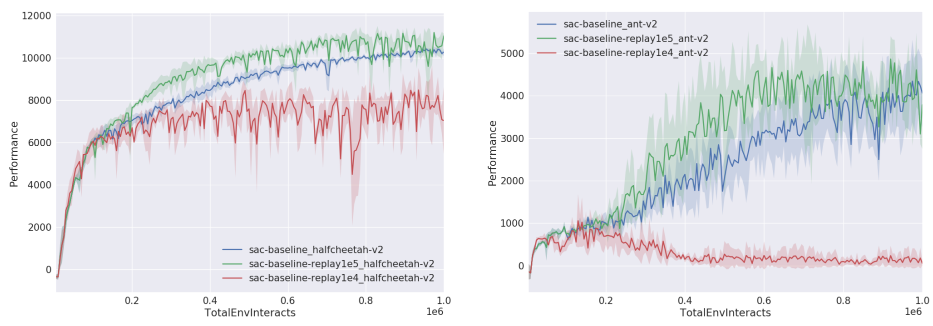

Figure 2: SAC performance comparison on HalfCheetah and Ant Mujoco environments, on different replay buffer sizes.

The result shows that a smaller buffer of size 1e5 can indeed boost performance, and as expected, a very small buffer of size 1e4 hurts performance, and even completely destroys learning in the more difficult Ant environment. Clearly using a lot of old data is crucial to the success of SAC, but it can be beneficial to use these data in a more sophisticated fashion so that performance is further improved. It's possible to use something similar to a prioritized buffer[14], or use a design that's more tailored towards the structure of SAC.

5 Enhancing SAC

In addition to modifying existing structure, we want to try further improve the performance of SAC by incorporating some of the most recent advances in general techniques for deep learning. We experimented with combining SAC with the following 3 new techniques:

1. Adding remaining time to agent observation, discussed in Time Limits in Reinforcement Learning [15]

2. Selu activation unit, introduced in Self-Normalizing Neural Networks [16]

3. AMSGrad optimizer, proposed in On the Convergence of Adam and Beyond [17]

Note that AMSGrad and Selu are general deep learning methods that are proven to work well for supervised learning tasks, but their performance in reinforcement learning is not tested and not guaranteed. In each of the subsections to follow, we will give a brief introduction on what these methods are, discuss why we might use them in reinforcement learning, and then present the result of the experiments and give analysis on the result.

We again use the Mujoco robotics control benchmarks as test environments for the analysis [9]. The Ant environment is more difficult than the HalfCheetah environment, and the difficulty can affect the results. We consider the method to be more useful if it can give us more performance gain for the more difficult task.

5.1 SAC with AMSGRAD Optimizer

The original SAC paper uses the very popular and powerful Adam optimizer [18]. Adam uses the idea of momentum to make sure the gradients used to update the networks are neither too large nor not too small, and is maintained in a proper magnitude in each direction. A recent paper "On the convergence of Adam and beyond"[17] proved that there is a problem with Adam's convergence guarantee and proposed a new optimization method called AMSGrad, which has been tested on a set of supervised learning tasks including MNIST and CIFAR, and shown to outperform Adam in terms of faster learning and better convergence. However, it is unclear whether it also works well with DRL tasks, which are completely different from supervised learning ones.

Figure 3 shows total average return of evaluation rollouts during training for SAC baseline which uses Adam, and the modified SAC, which uses AMSGrad. Solid curve indicates mean across 3 random seeds, and shaded area show maximum and minimum values. Each experiment is run for 1 million timesteps (datapoints).

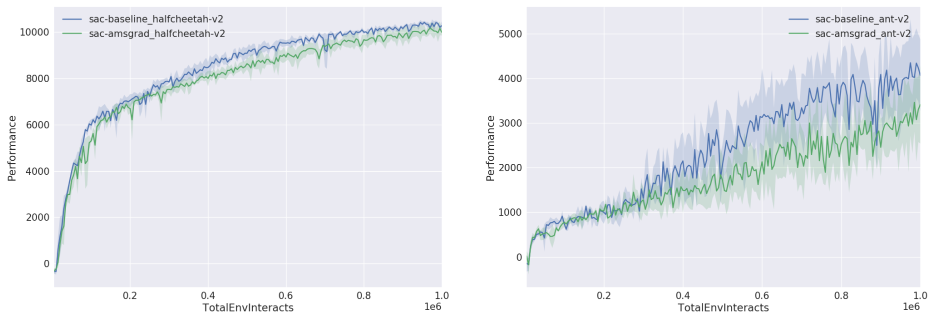

Figure 3: SAC performance comparison on HalfCheetah and Ant Mujoco environments, with Adam and AMSGrad optimizer.

The result is very interesting: although AMSGrad is proven to have better convergence property, it seems to perform slightly worse than original SAC, which uses Adam. This might be caused by the difference between supervised learning and reinforcement learning tasks, but it's also possible that AMSGrad should be modified to better fit the reinforcement learning scenario, instead of applied naively. Further research is required to see how DRL methods can benefit from the better convergence guarantees of AMSGrad.

5.2 SAC with Self-Normalizing Activation

The original SAC paper uses relu activation, an extremely popular and effective activation function. Self-Normalizing Neural Networks [16] is a recent paper that proposed a new type of activation called "scaled exponential linear units" (selu), these selu activations are what makes a self-normalizing neural net (SNN). Selu activation have a similar effect compared to batch normalization [19]. Which is, using the words of the authors: "neuron activations of SNNs automatically converge towards zero mean and unit variance". SNNs are comprehensively tested on an array of supervised learning tasks and outperform all competing fully-connected network methods. Similar to AMSGrad, this is a general DL method that is proven to work well with supervised learning tasks, and we want to test if it also works well in DRL tasks. SNNs can work well with deep network structures and improve robustness of learning, it's possible that it can also help DRL methods become more stable in training. Although very few, there already exist papers exploring the power of selu in DRL, for example, [20] tested selu in an ensemble Deep Deterministic Policy Gradient DRL Algorithm [10] showing selu is better than relu in the learn-to-run task they worked on. Since batch normalization methods have also proven to work well in some DRL methods, such as virtual batch norm in Evolutionary Strategy [12], it's very likely that selu will work well with SAC and further improve its performance.

We test a variant of SAC where we modify all the SAC networks to be SNNs, by changing the activation and also initialize the networks in the way described in the SNN paper [16].

Figure 4 shows total average return of evaluation rollouts during training for SAC baseline which uses relu, and the modified SAC, which uses selu. Same activation applies to all networks in SAC. Solid curve indicates mean across 3 random seeds, and shaded area show maximum and minimum values. Each experiment is run for 1 million timesteps (datapoints).

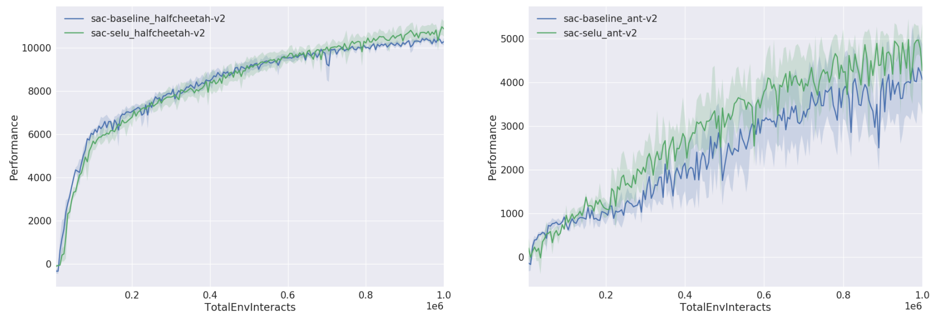

Figure 4: SAC performance comparison on HalfCheetah and Ant Mujoco environments, with relu activation and selu activation, selu activation also comes with a specific type of network weight initialization, details see [16]

Results indicate that selu activation doesn't affect much for the easier HalfCheetah environment, but can improve performance significantly on the more difficult Ant environment. This shows that selu activation can not only help in supervised learning scenario, but also in the reinforcement learning case. However, using the selu activation brings more computation time. Compared to relu activation, experiments using selu activation takes about 50% more wall clock time to run (from about 20 hours to about 30 hours for 1M data on 3 seeds), this is likely caused by the fact that selu uses exponential when the input is negative (details see [16]) while relu doesn't have this extra computation. But in terms of sample efficiency, as we can observe from the Ant environment result, it does bring substantial improvement.

5.3 SAC with Time-Aware Agent

NOTE: we have fixed a bug and re-run the experiments for this part, and have updated the figures and our analysis since Tuesday's presentation.

In Time Limits in Reinforcement Learning [15], Pardo et al. proposed to add remaining environment time (let's call it the time feature) to agent's observation to improve performance. This can be important for the Mujoco tasks, since many of the Mujoco tasks have a max timestep limit for each episode, which is typically 1000. That means for each episode, the agent gets the initial state, and then can interact with the environment for at most 1000 timesteps before the environment resets. Although this limit can be set manually to a longer period, in the literature it is often left as the default 1000. Some of the environments can terminate early, for instance, in the Ant environment, a very bad action can cause the environment to simply terminate and the agent won't get any more rewards from that episode. The authors found that getting cut off at time limit of the environment when the agent is not really in a bad state can confuse the agent, making its state value estimations inaccurate. The solution they proposed is adding the time feature to the agent's observation and their experiment on PPO [5] show that this leads to the learning of time-dependent actions that significantly improve performance.

Figure 5 shows total average return of evaluation rollouts during training for SAC baseline which doesn't have time feature, and the modified SAC, which has the time feature. Solid curve indicates mean across 3 random seeds, and shaded area show maximum and minimum values. Each experiment is run for 1 million timesteps (datapoints).

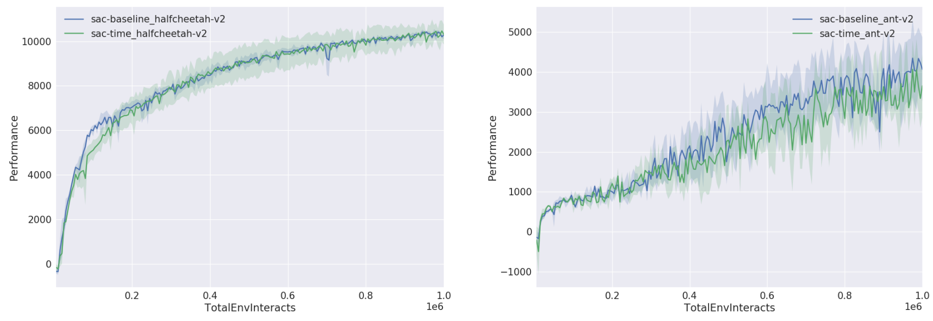

Figure 5: SAC performance comparison on HalfCheetah and Ant Mujoco environments, with baseline and time-aware enhancement.

The results show that the difference between performance is small. Adding the time feature doesn't seem to affect too much on the agent's learning, in the Ant environment performance even gets slightly worse, although the agent now has more information about the environment. This result could be caused by the fact SAC already somewhat count for early termination with a "done" flag that can affect the network update. It could also be because that the technique needs to be modified to fit the SAC algorithm.

6 Conclusions

To summarize the project:

We first presented an implementation guide: we have created a well-documented Pytorch implementation and discussed a list of critical implementation problems in making SAC work. This guide doesn't serve as a standalone tutorial, but as a support material that complement the existing online resources. Our code is also incorporated with OpenAI Spinup, making it easy to use all Spinup utilities and do algorithm comparison with all other algorithms provided on Spinup. This guide can help students who want to get an in-depth understanding of SAC to learn more effectively and run experiments easily. We also provide a NYU hpc setup guide on github.

We then did additional empirical analysis that are not done in the SAC paper: we experimented and analyzed how hidden layer sizes and replay buffer max size can be modified to affect SAC's performance. We show that SAC needs a relatively large network to learn well, thus runs slower in terms of computation, but achieves high data efficiency. We also showed that it's possible to optimize usage of data in the replay buffer to boost performance at no extra cost.

In the end we explore possible enhancements to the SAC algorithm: we experimented with 3 different recently developed DL techniques, experimented and analyzed how they can be combined with SAC to make it work better. The results show that although Adam is wrong in theory, it still achieves best performance in the tasks we tested. For time feature, it doesn't add much to the performance, might due to that SAC already counted for the early cut off problem. And for Selu, we see a significant improvement over data efficiency, at the cost of higher computation time.

7 Acknowledgment

Great thanks to Josh Achiam, for creating the amazing OpenAI Spinning Up site, and helping me with some issues on using Spinup utilities.

References

[1] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv preprint arXiv:1801.01290*, 2018.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[3] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning.," in *AAAI*, vol. 2, p. 5, Phoenix, AZ, 2016.

[4] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, pp. 1057–1063, 2000.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[6] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning*, pp. 1889–1897, 2015.

[7] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz, "Parameter space noise for exploration," *arXiv preprint arXiv:1706.01905*, 2017.

[8] J. Achiam, "Openai spinning up documentation." `https://spinningup.openai.com/en/latest/index.html`. Accessed: 2018-12-20.

[9] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 5026–5033, IEEE, 2012.

[10] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[11] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *arXiv preprint arXiv:1802.09477*, 2018.

[12] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[13] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint arXiv:1712.06567*, 2017.

[14] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[15] F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev, "Time limits in reinforcement learning," *CoRR*, vol. abs/1712.00378, 2017.

[16] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in *Advances in Neural Information Processing Systems*, pp. 971–980, 2017.

[17] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," in *International Conference on Learning Representations*, 2018.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[19] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[20] Z. Huang, S. Zhou, B. Zhuang, and X. Zhou, "Learning to run with actor-critic ensemble," *arXiv preprint arXiv:1712.08987*, 2017.