1. Introduction

Model-free deep reinforcement learning (RL) algorithms have been applied in a range of challenging domains, from games to robotic control. In Model-free RL, we ignore the model and depend on sampling and simulation to estimate rewards so we don't need to know the inner working of the system. While model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune. Model-free RL can be roughly divide into value-based and policy based methods. In this project, I implement this two kinds of model-free RL methods to solve several benchmark environments.

Value-based methods usually learn an approximation function $Q_\theta(s,a)$ for the optimal action-value function, $Q^*(s,a)$. Typically they use an objective function based on the Bellman equation and this optimization is almost always performed off-policy, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. The corresponding policy is obtained via the connection between Q^* and $\pi*$: the actions taken by the Q-learning agent are given by $a(s) = \arg\max_a Q_\theta(s,a)$. Most typical methods of value-based RL is DQN and its variants.

The many recent successes in value-based reinforcement learning (RL) to complex sequential decision-making problems were kick-started by the Deep Q-Networks algorithm(DQN). Its combination of Q-learning and convolutional neural networks and experience replay enabled it to learn to play Atari games at human-level performance. Since then, many extensions have been proposed that enhance its speed or stability, such as Double DQN, Dueling DQN, Noisy DQN, Prioritized experience replay, and Distributional DQN. In this project, I do experiments on DQN and its six variants and their combination in Atari environments Pong and Boxing.

Policy-based methods usually represent a policy explicitly as $\pi_\theta(a|s)$ and the policy is approximated by neural network with parameters θ. Then they optimize the parameters θ either directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. In policy-based RL, there are both off-policy methods, such as PPO and DDPG, and on-policy methods, which means that each update only uses data collected while acting according to the most recent version of the policy, such as A3C and OOP.

Value-based RL and Policy-based RL has their own strengths and weaknesses, but they are not incompatible. Some policy-based methods which concurrently learn Q-function and policy, such as DDPG and SAC, are able to carefully trade-off between the strengths and weaknesses of either side. SAC incorporates three key ingredients: an actor-critic architecture with separate policy and value function networks, an off-policy formulation that enables reuse of previously collected data for efficiency, and entropy maximization to enable stability and exploration. In this project, I do experiments on SAC with DDPG as baseline on three MuJoCo environments Hopper-v2, Ant-v2, and HalfCheetah-v2.

2. Value-Based Reinforcement Learning - DQN

2.1 Notation

The interaction between agents and environments is formalized as a Markov Decision Process(MDP), $\langle \mathcal{S}, \mathcal{A}, T, r, \gamma \rangle$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $T(s,a,s') = P[S_{t+1} = s' | S_t = s, A_t = a]$ is the transition probability, r is reward function, and γ is a discount factor. The discounted return $G_t = \sum_{k=0}^{\infty} \gamma_t^{(k)} R_{t+k+1}$ is the discounted sum of future rewards. Agents learns an estimate pf the expected discounted reward, that is value function $V^\pi(s) = E_\pi[G_t | S_T = s]$ or state action function $Q^\pi(s,a) = E_\pi[G_t | S_t = s, A_t = a]$.

2.2 Nature DQN

DQN combine reinforcement learning and deep learning. It uses neural network to approximate state action values $Q(s, a)$ with two methods, target network and experience replay, to reach stable performance. There are two networks in DQN, θ used to interact with environment and updated, $\bar{\theta}$ used to fix the Q-value targets temporarily. As moving target is unstable, we don't train target network $\bar{\theta}$ and use parameters of θ to update it at predefined intervals, which behaves more like supervised training. Experience replay means we put the last million transitions into a buffer and randomly sample a mini-batch of samples from this buffer to train the deep network. As we randomly sample from the replay buffer, the data is more independent of each other.

At each step, the agent selects an action $\epsilon-$greedily based on the current state, adds a transition $(S_t, A_t, R_{t+1}, S_{t+1})$ to replay buffer, and then randomly samples a mini-batch from replay buffer to compute the loss. The parameters of the neural network are optimized by minimizing the loss

$$\left(R_{t+1}+\gamma_{t+1} \max_{a'} Q_{\bar{\theta}}\left(S_{t+1}, a'\right)-Q_\theta\left(S_t, A_t\right)\right)^2$$

2.3 Extensions to DQN

2.3.1 Double DQN

In Nature DQN, Q value is usually overestimated because we use greedy algorithm to choose target Q value, which leads to tendency to select the action that is over-estimated. So instead of target network $\bar{\theta}$, θ is used to select the action. If Q_θ overestimate an action and select it, $Q_{\bar{\theta}}$ would give it a proper value. Thus, the loss becomes

$$\left(R_{t+1}+\gamma_{t+1} Q_{\bar{\theta}}\left(S_{t+1}, \underset{a'}{\operatorname{argmax}} Q_\theta\left(S_{t+1}, a'\right)\right)-Q_\theta\left(S_t, A_t\right)\right)^2$$

2.3.2 Dueling DQN

The dueling network is a neural network architecture which features two streams, the value and action advantage streams, sharing a convolutional encoder and owing different linear layers, and merged by a special aggregator. If input is state s, value streams output a single value $V(S)$ and action advantage streams output a vector $A(s, a)$. Then aggregator aggregates the two parts and output a vector $Q(s, a)=V(s)+A(s, a)-\frac{\sum_{a'} A(s, a')}{N_{action}}$.

2.3.3 Prioritized Replay

Nature DQN randomly samples a mini-batch from replay buffer. If we assign higher sample probability to those transitions with larger TD error, agent would learn faster. Accurately, we sample transitions with probability p_t relative to the last encountered absolute TD error:

$$p_t \propto \left|R_{t+1}+\gamma_{t+1} \max_{a'} Q_{\bar{\theta}}\left(S_{t+1}, a'\right)-Q_\theta\left(S_t, A_t\right)\right|^\omega$$

Here ω is a hyperparameter that determines the shape of the distribution and I set it to 2, so this term becomes loss. New transitions are inserted into the replay buffer with maximum priority, providing a bias towards recent transitions. Then, when a transition is sampled, its priority will be updated by p_t which is calculated with loss.

2.3.4 Noisy DQN

Formerly, we use $\epsilon-$greedy to add noise on action choosing, Noisy DQN adds noise on parameters of neural network at the beginning of each episode. It replaces standard linear layer $\boldsymbol{y}=(\boldsymbol{b}+\mathbf{W}\boldsymbol{x})$ with noisy linear layer:

$$\boldsymbol{y}=(\boldsymbol{b}+\mathbf{W}\boldsymbol{x})+\left(\boldsymbol{b}_{noisy} \odot \epsilon^b+\left(\mathbf{W}_{noisy} \odot \epsilon^w\right) \boldsymbol{x}\right)$$

Here ϵ^b and ϵ^w are random variables and $\odot$ donates the element-wise product. As time goes on, network can learn ignore most noisy and explore in a consistent way, allowing state-conditional exploration with a from of self-annealing.

2.3.5 Distributional DQN

The idea behind Distributional DQN is to learn approximate the distribution of returns instead of the expected return. $\mathcal{T}$ is defined as Bellman backup operator and we have $\mathcal{T}^\pi Q(\mathrm{s}, \mathrm{a}) \leftarrow r(\mathrm{s}, \mathrm{a}) + \gamma \mathbb{E}_{\mathrm{s}' \sim p, \mathrm{a}' \sim \pi}\left[Q\left(\mathrm{s}', \mathrm{a}'\right)\right]$. The distribution we want to get is $Z(s,a)$ and $Q(s,a) = \mathbb{E}Z(s,a)$, so we have $\mathcal{T}^\pi Z(\mathrm{s}, \mathrm{a}) := R(\mathrm{s}, \mathrm{a}) + \gamma Z\left(\mathrm{s}', \mathrm{a}'\right), \mathrm{s}' \sim p, \mathrm{a}' \sim \pi$. What's more , we can define Bellman optimality operator $\mathcal{T}^*$ and get

$$\mathcal{T}^* Z(\mathrm{s}, \mathrm{a}) := R(\mathrm{s}, \mathrm{a}) + \gamma Z\left(\mathrm{s}', \mathrm{a}'\right), \mathrm{s}' \sim p, \mathrm{a}' = \arg\max_{\mathrm{a}'} \mathbb{E}Z\left(\mathrm{s}', \mathrm{a}'\right)$$

Looking back the objective of Q-learning :$\left(R_{t+1} + \gamma_{t+1} \max_{a'} Q_{\bar{\theta}}\left(S_{t+1}, a'\right) - Q_\theta\left(S_t, A_t\right)\right)^2$, Q-learning wants to lessen the gap between $R + \gamma \max_{a'} Q_{\bar{\theta}}\left(s, a'\right)$ and $Q_\theta\left(s, a\right)$. Similarly, distributional DQN want to lessen the gap between $Z_\theta(s,a)$ and $R + \gamma Z_{\bar{\theta}}\left(\mathrm{s}', \mathrm{a}'\right), \mathrm{s}' \sim p, \mathrm{a}' = \arg\max_{\mathrm{a}'} \mathbb{E}Z\left(\mathrm{s}', \mathrm{a}'\right)$.

A practical method is to discrete distribution $Z(s,a)$ and use KL divergence to measure the distance between two distributions. So we can model the distributions with probability masses placed on a discrete support z, where z is a vector with $N_{actoms} \in \mathbb{N}^+ atoms$, defined by $z^i = v_{min} + (i-1)\frac{v_{max} - v_{min}}{N_{atoms} - 1}$ for $i \in \{1, \ldots, N_{atoms}\}$. On each atoms i, the probability mass is $p_\theta^i(S_t, A_t)$ and the approximating distribution $d_t = (z, p_\theta(S_t, A_t))$. The goal is to update θ such that d_t closely matches target distribution $d_t' \equiv \left(R_{t+1} + \gamma_{t+1}\boldsymbol{z}, \boldsymbol{p}_{\bar{\theta}}\left(S_{t+1}, \bar{a}_{t+1}^*\right)\right)$, where $\bar{a}_{t+1}^* = \mathrm{argmax}_a\, q_{\bar{\theta}}\left(S_{t+1}, a\right)$ is the greedy action with respect to the mean action values $q_{\bar{\theta}}\left(S_{t+1}, a\right) = \boldsymbol{z}^\top \boldsymbol{p}_\theta\left(S_{t+1}, a\right)$ in state S_{t+1} in state S_{t+1}.

Then, after a L2-projection of target distribution onto the fixed support z, KL-divergence is used to calculate the loss $D_{\mathrm{KL}}\left(\Phi_z d_t' \| d_t\right)$. The parametrized distribution d_t and d_t' can be represented by a neural network, as in DQN, but with $N_{atoms} \times N_{action}$ outputs. A softmax is applied independently for each action dimension of the output to ensure that the distribution for each action is appropriately normalized. As in the nature DQN , a frozen copy of the parameters θ is used to construct the target distribution.

2.3.6 Multiply Step

Instead of direct return , n-step return $R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}$ can be used to calculate targets, that is forward-view multi-step targets, $R_t^{(n)} + \gamma_t^{(n)} \max_{a'} Q_{\bar{\theta}}\left(S_{t+n}, a'\right)$. So now the loss we need to minimize becomes

$$\left(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} Q_{\bar{\theta}}\left(S_{t+n}, a'\right) - Q_\theta\left(S_t, A_t\right)\right)^2$$

If we choose appropriate hyperparameter n, agent will learn faster.

2.4 Environment Wrapper

The wrappers applied to the environment are very important for both speed and convergence. Here are the list of the environment wrapper I use and most of them are borrowed from OpenAI baseline project:

- **EpisodicLifeEnv**: ends episode at every life lost which helps to converge faster.
- **TimeLimit**: sets the max number of steps of an episode.
- **NoopResetEnv**: performs random amount of NOOP actions on the reset.
- **MaxAndSkipEnv**: repeats chosen action for 4 Atai environment frames to speed up training.
- **FireResetEnv**: presses fire in the beginning. Some environment requires this to start the game.
- **ProcessFrame84**: Frame converted to gray scale and scaled down to 84×84 pixels, which helps us save a lot of memory and increase computation speed.
- **FrameStack**: passes the last 4 frames as observation to speed up training
- **ClippedRewardWrapper**: clips reward to -1, 0, and +1, which make model more stable.

3 Policy-Based Reinforcement Learning - SAC

3.1 Notation

We consider the problem as a Markov decision process(MDP), (S, A, p, r, γ), where S and A are continuous state and action space, p is state transition probability which represents the probability of the next state $s_{t+1} \in S$ given the current state $s_t \in S$ and action $a_t \in A$, and $r : S \times A \to [r_{min}, r_{max}]$ is bounded environment reward. The state and state-action marginals of trajectory distribution induced by policy $\pi(a_t|s_t)$ is $\rho_\pi(s_t)$ and $\rho_\pi(s_t, a_t)$.

3.2 Maximum Entropy RL

The entropy of the event x is the sum over all possible outcomes i of x, of the product of the probability of outcome i times the log of the inverse of the probability of i, that is $H(x) = -\sum_{i=1}^{n} p(i) \log_2 p(i)$.

Standard RL wants to find the policy that maximizes the expected sum of rewards, that is

$\pi^*_{std} = argmax_\pi \sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_{pi}} [r(s_t, a_t)]$

SAC consider a more general maximum entropy objective by adding the expected entropy of the policy over $\rho_\pi(s_t)$, so the policy is

$\pi^*_{maxEnt} = argmax_\pi \sum_t \mathbb{E}_{(s_t,a_T\sim\rho_\pi)} [r(s_t, a_t) + \alpha H(\pi(\cdot|s_t))]$

The temperature parameter α determines the relative importance of the entropy term against the reward, and thus controls the stochasticity of the optimal policy. Here, $H(\pi(\cdot|s')) = -\mathbb{E}_a \log \pi(a'|s')$.

3.3 Soft Policy Iteration

Actually, we can take $H(\pi(\cdot|s_t))$ as a part of reward:

$r_{soft}(s_t, a_T) = r(s_t, a_t) + \gamma\alpha\mathbb{E}_{s_{t+1}\sim\rho} H(\pi(\cdot|s_{t+1}))$

Then, we can get soft Q function and soft V function:

$Q_{soft}(s_t, a_t) = r(s_t, a_t) + \gamma\mathbb{E}_{s_{t+1},a_{t+1}} (Q_{soft}(s_{t+1}, a_{t+1}) - \alpha \log \pi(a_{t+1}|s_{t+1}))$

$V_{soft}(s_t, a_t) = \mathbb{E}_{a_t\sim\pi}[Q_{soft}(s_t, a_t) - \alpha \log \pi(a_t|s_t)]$

The definition of energy-based policy is $\pi(a_t|s_t) = e^{-\varepsilon(s_t,a_t)} / \int e^{-\varepsilon(s_t,a_t)}\, dx$, where ε is energy function. To connect soft Q function, we set energy function to $-\frac{1}{\alpha} Q_{soft}(s_t, a_t)$. So we get

$\pi^*(a_t|s_t) = exp(\frac{1}{\alpha} Q^*_{soft}(s_t, a_t) - V^*_{soft}(s_t))$

Policy iteration consists of policy evaluation and policy improvement. In policy evaluation, we fix policy and update Q value to convergence via Bellman equation. In policy improvement, we update policy via $\pi'(s) = \arg\max_a Q_\pi(s, a)$. Similarly, we can get soft policy iteration.

In soft policy evaluation, we update Q via soft Bellman equation:

$Q_{soft}(s_t, a_t) = r(s_t, a_t) + \gamma\mathbb{E}_{s_{t+1},a_{t+1}} (Q_{soft}(s_{t+1}, a_{t+1}) - \alpha \log \pi(a_{t+1}|s_{t+1}))$

In soft policy improvement, we use KL divergence to approximate $exp(Q^\pi_{soft}(s_t, \cdot))$ and update policy:

$\pi' = \arg\min_{\pi_k\in\Pi} D_{KL}(\pi_k(\cdot|s_t) || \dfrac{exp(\frac{1}{\alpha} Q^\pi_{soft}(s_t, \cdot))}{Z^\pi_{soft}(s_t)})$

3.4 Soft Actor-Critic

We use two Q network $Q_\theta(s_t, a_t)$ and a policy network $\pi_\phi(a_t|s_t)$.

Q network is quite simple, with 6 linear layers and outputs a single value. $Q_\theta(s_t, a_t)$ is network we train and $Q_{\bar{\theta}}(s_t, a_t)$ is target Q network. The objective of Q network is

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t,s_{t+1})\sim D, a_{t+1}\sim\pi_\phi}[\tfrac{1}{2}(Q_\theta(s_t,a_t) - (r(s_t,a_t) + \gamma(Q_{\bar{\theta}}(s_{t+1},a_{t+1}) - \alpha\log(\pi_\phi(a_{t+1}|s_{t+1})))))^2]$$

Policy network have 3 linear layers and output a distribution, generally the mean and covariance of Gaussian distribution. The objective of policy network is to KL divergence:

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D}, a_t\sim\pi_\phi}\left[\log\pi_\phi(a_t|s_t) - \frac{1}{\alpha}Q_\theta(s_t,a_t) + \log Z(s_t)\right]$$

We use re-parameterization trick to get action:

$$a_t = f_\phi(\varepsilon_t; s_t) = f_\phi^\mu(s_t) + \varepsilon_t \odot f_\phi^\sigma(s_t)$$

Here, function f is the policy network that outputs mean and variance and ε is noise term. Then, we sample action from this distribution. This process can be differentiated. Finally, we remove the constant item $\log Z(s_t)$ and get

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D}, \varepsilon\sim\mathcal{N}}[\alpha\log\pi_\phi(f_\phi(\varepsilon_t; s_t)|s_t) - Q_\theta(s_t, f_\phi(\varepsilon_t; s_t))]$$

3.5 Modification

3.5.1 Learned Temperature

In the previous section, we consider temperature α as a hyperparameter, but choosing the optimal temperature is non-trivial, and the temperature needs to be tuned for each task. Besides, given that the reward is ever changing, it's unreasonable to use a fixed temperature. Therefore, it's reasonable to make it self adjusting. When we explore a new area, the optimal action is uncertain and we need larger temperature to explore more. As the exploration goes on, temperature reduces slowly.

Our aim is to find a stochastic policy with maximal expected return that satisfies a minimum expected entropy constraint. Formally, we want to solve the constrained optimization problem

$$\max_{\pi_{0:T}} \mathbb{E}_{p_\pi}\left[\sum_{t=0}^{T} r(\mathbf{s}_t, \mathbf{a}_t)\right] \text{ s.t. } \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi}[-\log(\pi_t(\mathbf{a}_t|\mathbf{s}_t))] \geq \mathcal{H}, \forall t$$

We can solve its dual problem via the following equation.

$$\alpha_t^* = \arg\min_{\alpha_t} \mathbb{E}_{\mathbf{a}_t\sim\pi_t^*}\left[-\alpha_t\log\pi_t^*(\mathbf{a}_t|\mathbf{s}_t;\alpha_t) - \alpha_t\overline{\mathcal{H}}\right]$$

We can learn α by minimizing the above dual objective, but in practice, we compute gradients for α with the following objective:

$$J(\alpha) = \mathbb{E}_{\mathbf{a}_t\sim\pi_t}\left[-\alpha\log\pi_t(\mathbf{a}_t|\mathbf{s}_t) - \alpha\overline{\mathcal{H}}\right]$$

Now the whole algorithm is listed as follows.

Algorithm 1 Soft Actor-Critic

```
Input: $\theta_1, \theta_2, \phi$                                        ▷ Initial parameters
  $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$                         ▷ Initialize target network weights
  $\mathcal{D} \leftarrow \emptyset$                                         ▷ Initialize an empty replay pool
  for each iteration do
    for each environment step do
      $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$                         ▷ Sample action from the policy
      $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$                 ▷ Sample transition from the environment
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$     ▷ Store the transition in the replay pool
    end for
    for each gradient step do
      $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$    ▷ Update the Q-function parameters
      $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$                    ▷ Update policy weights
      $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$                      ▷ Adjust temperature
      $\bar{\theta}_i \leftarrow \tau \theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$      ▷ Update target network weights
    end for
  end for
Output: $\theta_1, \theta_2, \phi$                                       ▷ Optimized parameters
```

The algorithm uses two soft Q networks and each Q network has its own target Q network. It's said that two soft Q network can mitigate positive bias in the policy improvement step that is known to degrade performance of value based methods and significantly speed up training, especially on harder tasks.

3.5.2 Soft Target Update and Hard Target Update

As we can't find a oracle target network at first, we update target network from time to time. This changing target network leads to the non-stationarity of the regression objective, but it isn't bound to lead to instability. If we keep target network and eval network the same, it does cause instability. There are some method to avoid this, including soft target update and hard target update.

Original algorithm uses soft target update $\bar{\theta} \leftarrow \tau\theta + (1 - \tau)\bar{\theta}$ and update target network parameters every gradient step rather than hard target update $\bar{\theta} \leftarrow \theta$ and updating target network parameters at predefined intervals. I will compare the performance of soft target update and hard target update in both SAC and DQN in the following part.

3.5.3 Noise & Multi-Step

Given that SAC uses Q network, I think some methods in variants of DQN may be useful for SAC. In fact the Q network in DQN and SAC have some differences. Q network in SAC inputs state and action and output Q value for this state and action while Q network in DQN inputs state and outputs Q value of all the possible actions in this state. Therefore, double and dueling method seems infeasible because of network structure.

I think noise, multi-step, and prioritized replay buffer seem feasible and easy. For noise and multi-step method, it's easy to follow the example of DQN. For prioritized replay buffer, maybe we need to reconsider how to choose priority item, but I don't do experiments on this.

4 Experiments

4.1 DQN

4.1.1 Setup

I set network structure and hyperparameters mainly based on Rainbow, but I modify some hyperparameters to speed up and save memory. I do experiments on two Atari games, Pong and Boxing, with the structure of network and most hyperparameters the same.

The basic network has 3 convolutional layers: with 32, 64 and 64 channels. The layers use 8 × 8, 4 × 4, 3 × 3 filters with strides of 4, 2, 1, respectively. The basic network, the value and advantage streams of the dueling architecture, and distributional architecture all have a hidden layer with 512 units. For basic network, the output layer of the network has a number of units equal to the number of actions available in the game. For dueling architecture, value steams output a single value and advantage streams output a vector with a number of units equal to the number of actions. For distributional architecture, the shape of the output is $N_{atoms} \times N_{action}$. A softmax is applied independently for each action dimension of the output to ensure that the distribution for each action is appropriately normalized. Other activation function is ReLU.

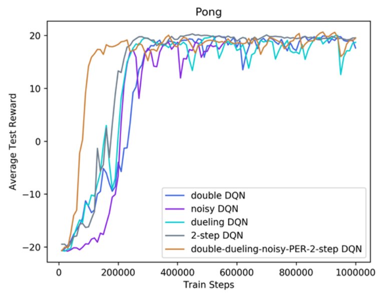
I use a discount factor of 0.99 and a mini-batches of 32 transitions. DQN and its variants do not perform learning updates during the first 10000 frames, to ensure sufficiently uncorrelated updates. I use the Adam optimizer with a learning rate of 0.001. For prioritized replay buffer, we used the recommended proportional variant, with priority exponent ω of 0.5, and linearly increased the importance sampling exponent β from 0.4 to 1 over the course of training. The value of n in multi-step learning is a sensitive hyper-parameter. We compared values of n = 2, 3, and 5. I observed that n=3 performed better. For distributional part, I choose 51 distributional atoms and set the distributional min and max values to -10 and 10. I update the target network every 1000 steps.

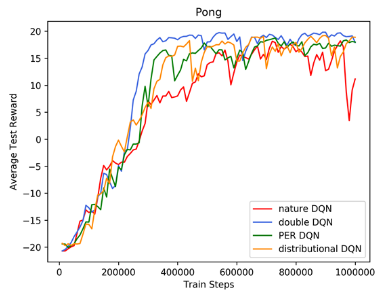
There are some hyperparameters different in Pong and Boxing. As it takes a long time and too much memory to train agent if I use hyperparameters in Rainbow, I modify some hyperparameters. And I find Pong is easier to train than Boxing, so I set different hyperparameters for those two environments. Main differences are listed as follows.

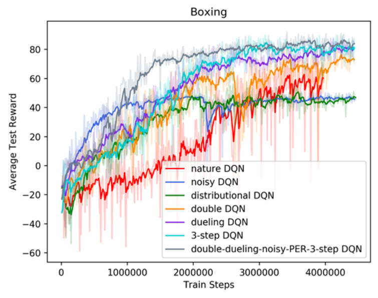
hyperparameter	**Pong**	**boxing**
buffer size	100K transitions	3M transitions
epsilon decay	100K step	1M step
epsilon final	0.02	0.01
train frequency	1 step	4 step
parameters initialization	False	True

4.1.2 Results

I evaluate nature, double, dueling, noisy, multi-step, prioritized replay buffer, distributional DQN and their combination on two environments, Pong and boxing. All of the algorithm use the same network architecture and parameters under the same environment. Every 10000 steps, I test the model I train for 10 episodes and record the average episode rewards, and then I get the figures below.







Usually, DQN with extension performs better than nature DQN and the combination of extensions performs better than single extension. The combination of double, dueling, noisy, prioritized replay buffer, and multi-step DQN performs better than any of others under both environments. The best average test results in Pong and Boxing are 20.69(double-dueling-noisy-PER-2-step DQN) and 88.23(double-dueling-noisy-PER-3-step DQN).

I've tried distributional DQN and it performs better than nature DQN, but I find it's hard to combine distributional DQN and others together and achieve good performance. There may be several reasons. First, there are some bugs in distributional DQN or the combination due to limited time. Second, it's because I use hyperparameters which are different from that in Rainbow, limited to time and computational resource. I don't use hyperparameters in rainbow, because it need too much memory(large replay buffer) and it takes a longer time to converge. Take Pong as example, rainbow uses 1-2 million steps to converge but my setting only uses 0.1-0.2 million.

4.2 SAC

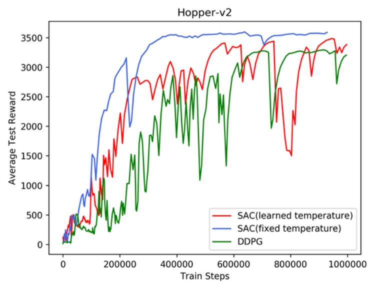
4.2.1 Setup

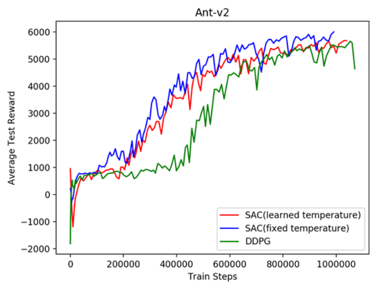
I do experiments on three continuous environments, Hopper-v2, Half Cheetah-v2, and Ant-v2, with same network structure and hyperparameters.

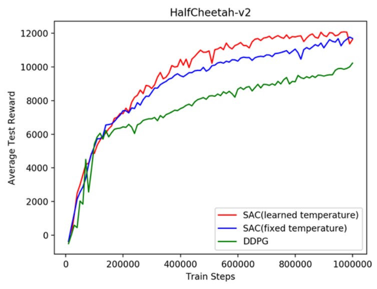
SAC uses two groups of soft Q network and target network and a policy network. All the Q networks are of the same structure. Input of network are state and action, and then they go through a two-layer neural network with 256 hidden units on each layer. Gaussian policy is also modeled by a two-layer neural network. It inputs states and outputs parameters of Gaussian distribution, μ and σ, modeled by a shared 256-units linear layer and then two separate 256-units linear layers. Then the final action is sampled from this distribution. For baseline DDPG, I use the same Q network with SAC but different policy network. Policy network of DDPG inputs state and outputs action, with a two-layer neural network with 256 hidden units on each layer.

I use Adam optimizer with the same learning rate of 0.0003 to train all the networks and temperature parameter α=0.2 for fixed temperature setting. I use a discount factor of 0.99 and a mini-batches of 32 transitions and a replay buffer of 1 million. I simply set the target entropy to be -1 per action dimension and do not perform learning updates during the first 10000 step. For fixed temperature method, I set the temperature to 0.2. For soft update, I set target smoothing coefficient to 0.005. DDPG uses the same hyperparameters as SAC expect for some useless ones like temperature.

4.2.2 Results







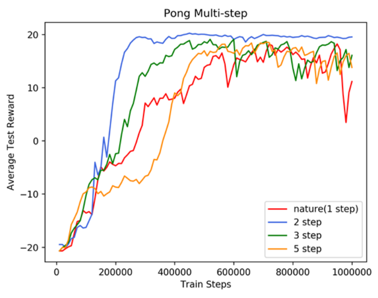
I do experiments on three MuJoCo environments, Hopper-v2, Ant-v2, and HalfCheetah-v2, with Deep Deterministic Policy Gradient(DDPG) and SAC. All of the algorithm use the same network architecture and parameters under the different environments. Every 10 train episodes, I test the model I train for 10 episodes and record the average episode rewards, and then I get the figures above.

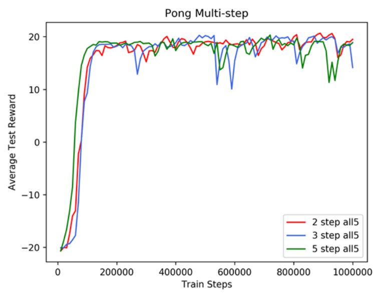
As the figure shows, SAC converges faster and gets higher rewards, so overall it performs better than DDPG in all three environments. The results also indicate that SAC with learned temperature usually achieves practically identical or better performance compared to SAC with fixed temperature, which is well tuned per environment for all environments. In Hopper, it seems SAC with learned temperature is more instable than fixed temperature, but learned temperature reach similar or better performance in Ant and HalfCheetah. The best average test results in Hopper, Ant, and HalfCheetah are 3620(SAC, fixed temperature), 6336(SAC, learned temperature), and 12281(SAC, learned temperature).

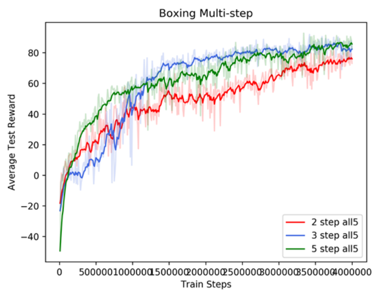
5 Discussion and Analysis

5.1 DQN

5.1.1 Multi-step





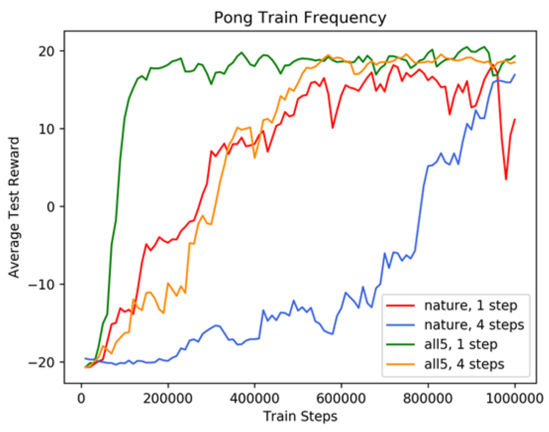


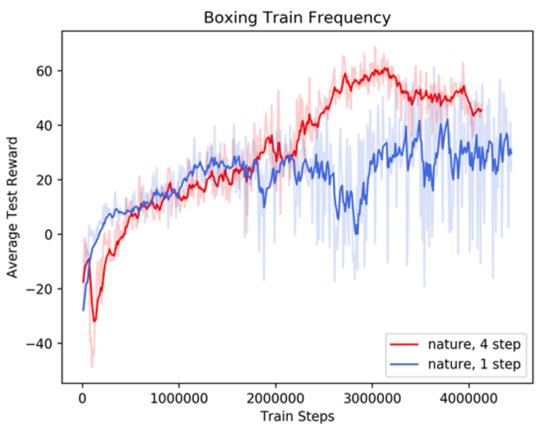
Here, I compare the results of different steps and explain why I finally choose 2 step in Pong. As the figure shows, Multi-step does take an effect and performs better than nature DQN. One-step learning obtains a reward r only directly affects the value of the state action pairs s, a that led to the reward which slows learning since many updates are required to propagate a reward to the relevant preceding states and actions. N-step learning makes a reward directly affect n state action pairs and makes the process of propagating rewards to relevant state-action pairs potentially much more efficient, so it learns faster.

But larger step brings larger variance and that's why 3 step and 5 step performs worse than 2 step in Pong. The middle figure, where all5 means the combination of double, dueling, noisy, prioritized replay buffer, and multi-step DQN, also shows that 2 step is more stable than 3 and 5 step, despite that their converging speed are about the same. However, the trade-off between speed and stability may vary from environment to environment, for example, the right figure shows that 3 step performs slightly better than 2 step in Boxing.

5.1.2 Difference between Pong and Boxing

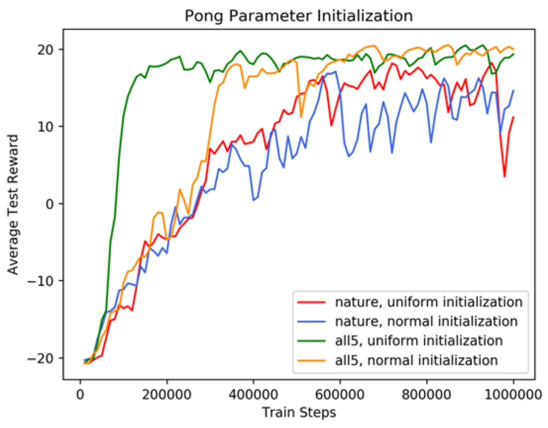
Besides difference in choice of multi-step, Pong and Boxing still have many differences in many aspects. The most obvious one is that Boxing is more hard to train than Pong. Pong usually takes 10 to 20 thousand steps to converge with a memory buffer of 10000 while Boxing takes 1 to 3 million steps to converge with a memory buffer of 3 million. This also results they need different final epsilon value and epsilon decay step. Pong and Boxing also have different preference for parameter initialization and train frequency.

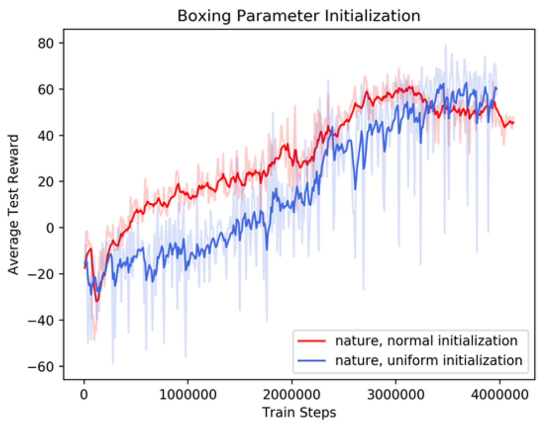




The figure above shows that small train step performs better in Pong while the situation is exactly reversed in Boxing. This may have thing to do with game itself. One of the reasons that Pong is easy to train may be that information is more denser in frames of Pong and training model with frame we get in every step is more efficient. However, in Boxing, the information we can get from several consecutive steps may be similar, so training model every step becomes less efficient.

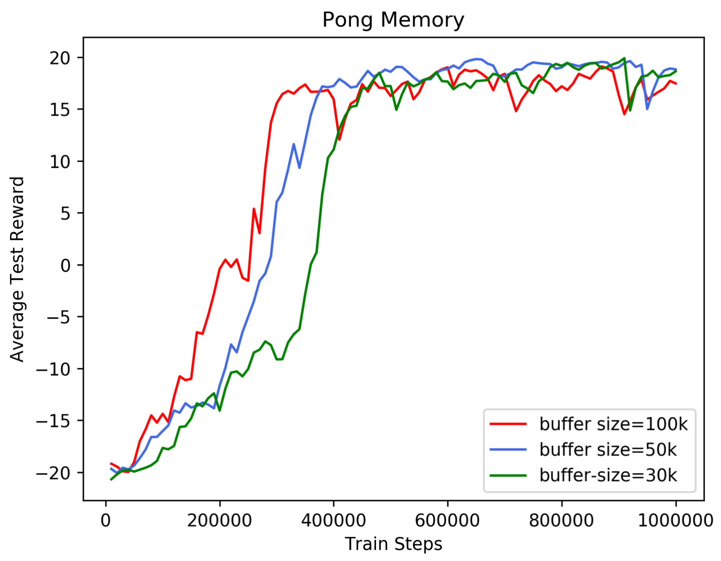
Expect training frequency, Boxing and Pong also show different preference in the way that parameters of convolution layers initialize. The figure below shows that Pong prefers uniform initialization while Boxing prefers normal initialization.





Actually, maybe there are many parameters for which different environments has different preference. Although I clip step reward to same range, it only reduces part of difference of environments and environments still have their own preference for parameters. So it's understandable that my experiments converge faster than Rainbow, since Rainbow use same parameters for different environments and those parameters can't fit every environment perfectly. Tuning parameters specially for some environments does improve performance, but it lose the ability of generation and it is time-consuming if there are many environments. I do this because I only do experiments on two environments and I need it to converge as soon as possible due to the limits of time and resources.

5.1.3 Memory Saving



The experiments on the size of replay buffer shows that the larger the replay buffer is, the fast it converges and it's hard for agent to converges in 1 million steps if the size of replay buffer is smaller than 10000. As we know, regression learning assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states. Replay buffer and randomizing the samples breaks these correlations and therefore reduces the variance of the updates, which smooths out learning and avoids oscillations or divergence in the parameters. Given that correlation between frames we get from atari game is stronger that simple RL problems, usually we need larger replay buffer.

Although larger replay buffer brings better results, it takes more memory, especially for environments that return images as observations. An ordinary practice of memory saving for atari games is to compress and skip frames as some environment wrappers does. Here I list some tips for memory saving:

- Use environment wrapper ProcessFrame84 we mention before to convert frame to gray scale and scaled down to 84×84 pixels, which helps to save a lot of memory and increase computation speed.
- Store frames in uint8 type, which takes less memory than float.
- number frames so that next state can be stored in next transition, but in this way, sampling may become a bit complicated.
- If pytorch is used, we can use torch.from_numpy() to share memory between numpy and torch variable.
- Try more efficient sample method, such as prioritized replay buffer.

5.1.4 What DQN Learns?

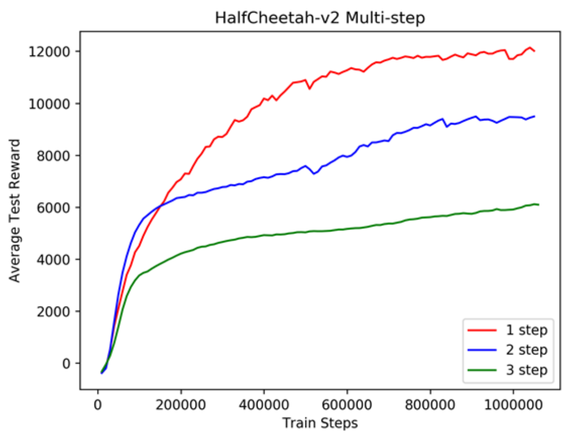
I watch the game video between the agent I trained and its opponent, and I find agent performs better partly because of its fast reaction speed. So does agent really learn good strategy and wisdom?

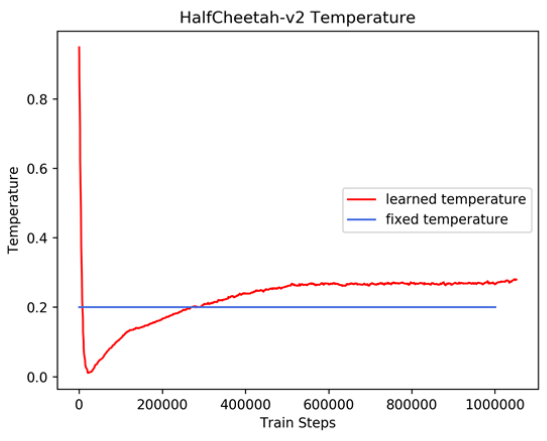
I don't know. It seems that agent shows little wisdom in this sample game, but complex RL agent such as AlphaGo does learn wisdom of human and even beyond human. I think it's because my environment is simple and shortcut like fast reaction speed is available here. Wisdom is complex and hard to learn while things like fast reaction speed are easy for computer. If we limit the reaction speed of agent to human level, it may learn more about good strategy and wisdom, but it may need more time, more resources, and more powerful algorithms. Actually, we have achieve a lot at those things. RL is hard but powerful.

5.2 SAC

5.2.1 Temperature

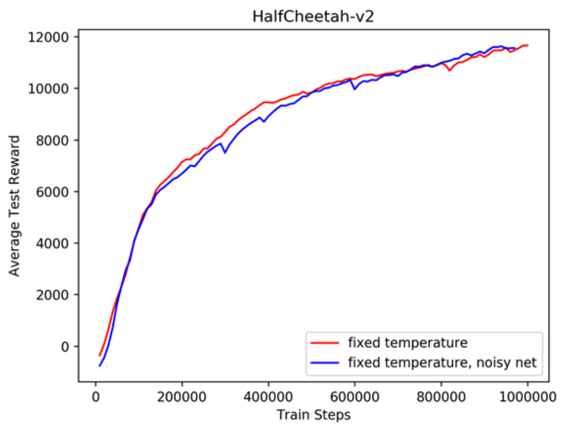
I compare the performance of SAC with learned temperature and with difference fixed temperatures of 0.05, 0.2, and 0.5. The results shows that temperature of 0.2 performs best among fixed temperatures and learned temperature performs better than any of them. SAC with learned temperature usually achieves practically identical or better performance compared to SAC with fixed temperature, which is well tuned. Choosing the optimal temperature is non-trivial, and the temperature needs to be tuned for each task, so learned temperature can help us save a lot of time.

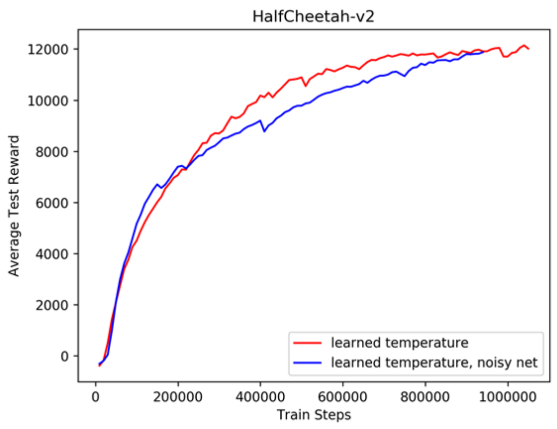




Then, I compare how the temperature evolve during training. The right figure compares the temperature parameter of the two methods. SAC with learned temperature (red) actively adjusts the temperature, particularly in the beginning of training when the Q-values are small and the entropy term dominates in the objective. The temperature is quickly pulled down so as to make the entropy to match the target. For other simulated environments, I observe temperature curves throughout the learning.

5.2.2 Noisy Net

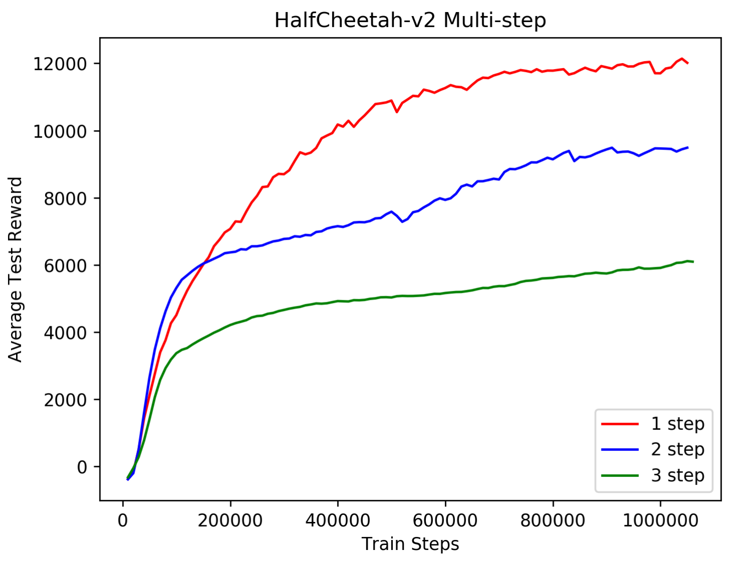




As the figures shows, whatever it's in fixed or learned temperature SAC, noisy net shows little effect. I think a possible cause of this problem is: in SAC, policy network models a Gaussian distribution and the exploration mainly happens when we sample action from this distribution while explorations happens in choosing action from the output of Q network in DQN. Therefore noisy in Q network improve the performance of DQN, but shows little effect on SAC.

5.2.3 Multi-step

It seems multi-step doesn't work in SAC. A possible cause may be I just simply store n-step reward and next state, but actually SAC need multi-step off-policy correction. DQN uses stochastic approximation to solve Bellman Optimality Equation. In one-step Q learning and SAC, estimate Q value has nothing to do with old policy, so it doesn't need off-policy correction. Theoretically, n-step learning need off-policy correction for the estimate Q value is relevant to old policy, but my method and Rainbow don't use this and it still improves our performance in practice. Maybe unlike multi-step DQN, multi-step off-policy correction is necessary for policy-based RL. However, due to the limit of time, I don't do experiments on SAC with multi-step off-policy correction to confirm my guess.



To sum up in few words, Q network in SAC is difference from that in DQN and many extensions that work well in DQN can't be directly used in SAC even if it seems to work on the face of it.

5.2.4 Advantages and Problems

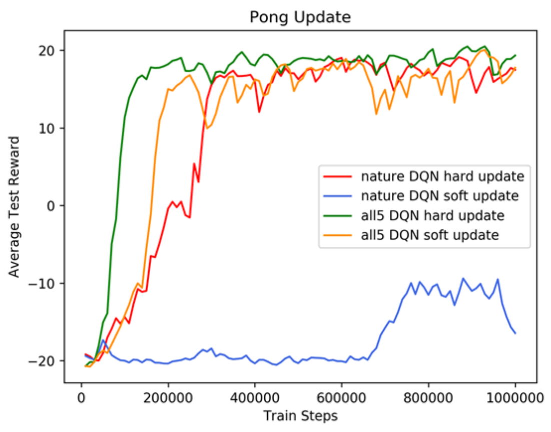
SAC uses off-policy learning and thus improves sample efficiency. it uses maximum entropy reinforcement learning to optimizes policies to maximize both the expected return and the expected entropy of the policy. In deterministic policy-based algorithm, we only need to search for an optimal path. SAC also maximum entropy of policy, which means it need to explore all possible optimal paths. So its ability of exploration and generalization is stronger and agent can learn more near-optimal actions and improve its learning rate.

There also are some problems in SAC. It doesn't reach ideal energy-based policy but use Gaussian distribution to approximate Boltzmann Distribution. So, actually SAC is still a unimodal but not multi-model of soft q-learning.

5.3 Soft Update and Hard Update on SAC and DQN

Given that we update target network from time to time, so we optimizes an objective that is non-stationary in two ways: the target values are updated during training, and the distribution under which the Bellman error is optimized changes, as samples are drawn from different policies. Those problem is inevitable for both DQN and SAC since they both use Q target network.

However, they use different update method. SAC uses soft update, which introduce an additional smoothing parameter τ to Q-iteration, where the target values are now computed as an τ-moving average over previous targets. The update formula is $\bar{\theta} = \tau\theta + (1 - \tau\bar{\theta})$, where θ and $\bar{\theta}$ are the parameters of Q network and target Q network. This means that the target values are constrained to change slowly, greatly improving the stability of learning. This simple change moves the relatively unstable problem of learning the action-value function closer to the case of supervised learning, a problem for which robust solutions exist. DQN use hard update, which assigns θ to $\bar{\theta}$ at a regular interval. The large update interval guarantees a stage that target values and distribution don't move. I wonder whether soft update or hard update performs better, so I do those two experiments on both SAC and DQN



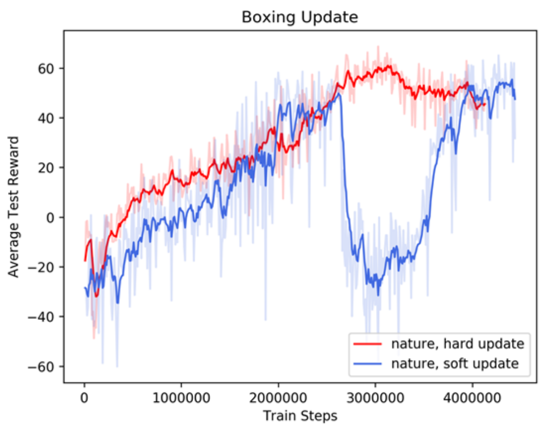
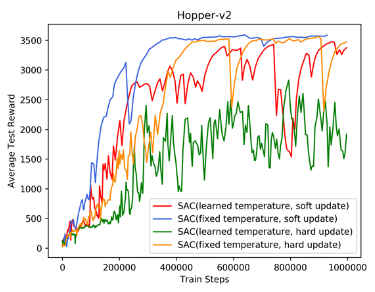
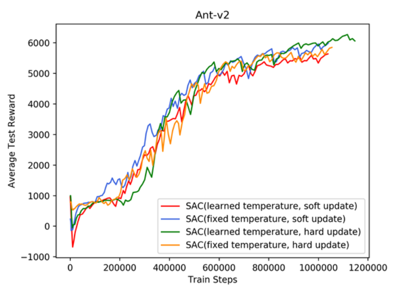
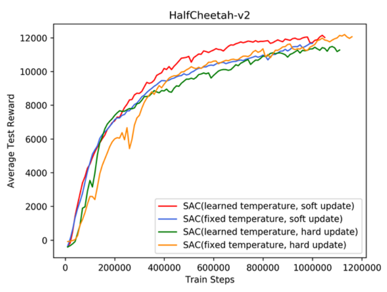


Figure above shows hard update is better on DQN and soft update shows larger fluctuation than hard update in both Pong and Boxing. However, in SAC, things goes in the opposite way. Soft update is more stable than hard update in Hopper and performs better than hard update. Maybe soft update and hard update have their own applicable scope. Soft update better suits SAC or continuous control environment like MuJoCo while hard update better suits DQN or discrete action environments.







6 Reference

[1] M. Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning.", 2017.

[2] Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.

[3] Soft Actor-Critic Algorithms and Applications

[4] Learning to Walk via Deep Reinforcement Learning

[5] Continuous control with deep reinforcement learning