RL Algorithms

Kaige Yang
University College London
Kaige.yang.11@ucl.ac.uk

Contents

1 RL Problem Setting **3**

2 TD-Learning **5**

3 SARSA **6**

4 LSTD **7**

5 LSPI **7**

6 Q-Learning **7**

7 REINFORCE **8**

8 Actor-Critic **9**

9 DQN **10**

10 {n-step, Double, Noisy, Dueling, Categorical}-DQN **10**

11 Rainbow **12**

12 Agent 57 **12**

13 A3C **12**

14 A2C **14**

15 ACER **14**

16 PPO **15**

17 DPG **16**

Report Draft.

18 DDPG **17**

19 D4PG **18**

20 TD3 **19**

21 Soft Q-Learning **19**

22 SAC **21**

23 State-of-the-art Algorithms **23**

24 Model-Based Algorithms **24**

25 AlphaGo **24**

26 AlphaZero **24**

27 Papers **24**

Abstract

This report contains the detail of RL algorithms including TD-Learning, SARSA, LSTD, LSPI, Q-Learning, REINFORCE, Actor-Critic, DQN, Rainbow, A3C, A2C, PPO, DDPG, D4PG, TD3, Soft-Q and SAC. The goal is to provide a holistic view on the recent evolution of algorithms. The report starts from describing the RL problem setting, and then moves to algorithms' details.

1 RL Problem Setting

Markov Decision Processes (MDPs) are a tool for modeling sequential decision-making problems where an agent interacts with an environment in a sequential fashion. An MDP, denoted by $\mathcal{M}$, is defined as $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma)$. The state space $\mathcal{S}$, the action space $\mathcal{A}$, the transition probability kernel $\mathcal{P}$ where $\mathcal{P}(s'|s,a)$ gives the probability that the next state s' given the current state s and taken action a. The reward function $r(s,a)$ gives the immediate reward upon taking action a in state s. $\gamma \in [0,1)$ is the discounted factor.

Given an MDP $\mathcal{M}$, the interaction between the agent and environment happens as follows: Let $t \in \mathbb{N}$ denote the current time, let $s_t \in \mathcal{S}$ be the current state and a_t be the action chosen by the agent at time t. Upon the action is chosen, the state of the environment moves to s' following $\mathcal{P}(s'|, s, a)$. The agent receives a reward $r(s,a)$. Then, the agent observes the next state s' and chooses next action a' and the process repeated. A trajectory $\{(s_0, a_0, r_1), ..., (s_{t-1}, a_{t-1}, r_t)\}$ summaries the interaction history up to time t.

The **goal** of the agent is to find a policy π so as to maximize the expected total discounted reward (expected discounted cumulative reward). At each time t, $\pi(a|s_t)$ gives the probability of action a being selecting given the current state s_t. The expected discounted cumulative reward is defined as

$$R = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \tag{1}$$

The goal is to find optimal policy which maximizes R. Namely,

$$\pi^* = \arg\max_{\pi} \sum_{t=0}^{\infty} \gamma^t r_{t+1} \tag{2}$$

The state-value function, $V : \mathcal{S} \to \mathbb{R}$, is defined by

$$V(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} | s_0 = s\right], s \in \mathcal{S}. \tag{3}$$

The action-value function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.

$$Q(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} | s_0 = s, a_0 = a\right], s \in \mathcal{S}, a \in \mathcal{A}. \tag{4}$$

It is known that both $V(s)$ and $Q(s,a)$ satisfy **Bellman consistency equation**:

$$V(s) = \mathbb{E}\Big[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) V(s')\Big], \;\; s, s' \in \mathcal{S} \tag{5}$$

$$Q(s,a) = \mathbb{E}\Big[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) Q(s', a')\Big], \;\; (s,a) \in (\mathcal{S}, \mathcal{A}) \tag{6}$$

The optimal state value $V^*(s)$ is the maximum of the expected return under the constraints that the process starts at state s. Similarly, the optimal action value $Q^*(s,a)$ is the maximum expected return under the constraints that the process starts at state s and the first action chosen is a.

The relationship between V^* and Q^* is

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s,a), \quad s \in \mathcal{S} \tag{7}$$

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) V^*(s'), \quad s \in \mathcal{S}, a \in \mathcal{A}. \tag{8}$$

$V^*(s)$ and $Q^*(s,a)$ satisfy **Bellman optimality equations**:

$$V^*(s) = \max_{a \in \mathcal{A}} \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) V^*(s') \right], \quad s \in \mathcal{S} \tag{9}$$

$$Q^*(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \left[r(s,a) + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a') \right], \quad (s,a) \in (\mathcal{S}, \mathcal{A}) \tag{10}$$

There are three ways to find the optimal policy π^*.

1. The knowledge of $\mathbf{Q}^*$ alone is sufficient for finding π^*.
2. Knowing $\mathbf{V}^*$, $\mathbf{r}$ and $\mathcal{P}$ also suffices to find π^*.
3. Find π^* directly without relying on $\mathbf{Q}^*$ and $\mathbf{V}^*$.

Algorithms follows (1), (2) are called value-based, while algorithm follows (3) is called policy-based.

Basically all algorithms in RL try to answer the following questions:

1. How to learn V^*?
2. How to learn Q^*?
3. How to learn π^*?

The fundamental mechanism of value-based algorithms is **Policy Iteration**: Policy evaluation and policy improvement

- Policy evaluation
  Based on the Bellman equation, the state-value under a specific policy π can be estimated.

  $$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \left[r + \gamma V_\pi(s') \right] \tag{11}$$

  Using the Bellman equation as an update rule:

  $$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \left[r + \gamma V_k(s') \right] \tag{12}$$

  In the limit, $k \to \infty$, $V_k(s) \to V_\pi(s), \forall s \in \mathcal{S}$.
- Policy improvement
  An improved policy π' can be obtained by acting greedy with respect to the current policy π.

  $$\pi'(s) = \arg\max_a \sum_{s'} p(s'|s,a) \left[r + \gamma V_\pi(s') \right] \tag{13}$$

Value Iteration:
The value iteration update is obtained simply by turning the Bellman optimality equation into an update rule. Also note how the value iteration update is identical to the policy evaluation update except that it requires the maximum to be taken over all actions. In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one update of each state).

$$\begin{aligned} V_{k+1}(s) &= \max_a \mathbb{E} \left[r_{t+1} + \gamma V_k(s') | S_t = s, A_t = a \right] \\ &= \max_a \sum_{s',r} p(s'|s,a) \left[r + \gamma V_k(s') \right] \end{aligned} \tag{14}$$

For arbitrary V_0, the sequence $\{V_k\}$ can be shown to converge to V^* under the same conditions that guarantee the existence of V^*.

Generalized Policy Iteration refers to the general idea of letting policy evaluation and policy improvement processes interact independent of the granularity and other details of the two processes. All most all reinforcement learning methods are well described as GPI. That is, all have identifiable polices and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward to the value function for the policy.

The fundamental basis of policy-based algorithm is **Policy Gradient Theorem**.

The policy $\pi(a|s, \boldsymbol{\theta}) = Pr(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$ gives the probability that action a is taken at time t given that the environment is in state s at time t the parameter $\boldsymbol{\theta}$. Policy-based algorithms consider learning the policy parameter $\boldsymbol{\theta}$ base don the gradient of some performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter. These methods seek to maximize performance, so their update approximate gradient ascent in J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \bigtriangledown \hat{J}(\boldsymbol{\theta}) \tag{15}$$

where $\bigtriangledown \hat{J}(\boldsymbol{\theta})$ is a stochastic estimate whose expectation approximation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$.

In episodic case, the performance measure J is the value of the start state of the episode.

$$J(\boldsymbol{\theta}) = V_{\pi_{\boldsymbol{\theta}}}(s_0) \tag{16}$$

where $V_{\pi_{\boldsymbol{\theta}}}(s_0)$ is the true value function for $\pi_{\boldsymbol{\theta}}$.

With function approximation, it may seem challenging to change the policy parameter in a way that ensure improvement. The problem is that performance depends on both the action selections and the distribution of states in which these selections are made, and that both these are affected by the policy parameter. Given a state, the effect of the policy parameter on the action, and thus on reward, can be computed in a relatively straightforward way from knowledge of the parameterization. But the effect of the policy on the state distribution is a function of the environment ans is typically unknown.

How can we estimate the performance gradient with respect to the policy parameter when the gradient depends on the unknown effect of policy changes on the state distribution.

The **Policy Gradient Theorem** answers this question, which provides us an analytic expression for the gradient of performance with respect to the policy parameter that does not involve the derivative of the state distribution. The Theorem establishes that

$$\bigtriangledown J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \bigtriangledown_{\boldsymbol{\theta}} \pi(a|s, \boldsymbol{\theta}) \tag{17}$$

In continuing case, the performance measure is defined as the average reward rate.

$$J(\boldsymbol{\theta}) = r(\pi_{\boldsymbol{\theta}}) = \sum_a \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r \tag{18}$$

Another objective for continuous case is average value

$$J(\boldsymbol{\theta}) = \sum_s d_\pi(s) V_\pi(s) \tag{19}$$

where $d_\pi(s)$ is the stationary distribution of Markov chain for policy π.

2 TD-Learning

`TD-Learning` is a policy evaluation algorithm, which is designed based on **Bellman equation**.

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s, a) \Big[r + \gamma V_\pi(s') \Big] \tag{20}$$

It uses the Bellman equation as an update rule. It updates $V(s_t)$ toward estimated return $r_{t+1} + \gamma V(s_{t+1})$.

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{21}$$

where $r_{t+1} + \gamma \hat{V}(s_{t+1})$ is called TD-target, $\delta_t = r_{t+1} + \gamma V(s_{t+1})$ is the TD-error and α is the learning step.

To update the policy based on $V(s)$, the model of MDP is required. i.e, the reward function $\mathbf{r}$ and transition matrix $\mathbf{P}$.

$$\pi^{'}(a|s_t) = \arg\max_{a \in \mathcal{A}} r(s,a) + P(s_{t+1}|s_t, a)V(s_{t+1}) \tag{22}$$

However, typically the model of MDP is unknown.

With function approximation, suppose the state-value function $\hat{V}(s, \mathbf{w})$ is parameter by $\mathbf{w}$. The goal is to find parameter $\mathbf{w}$ minimizing mean-sqared error between approximate value $\hat{V}(s, \mathbf{w})$ and true value $V_\pi(s)$.

$$J(\mathbf{w} = \mathbb{E}_\pi[(V_\pi(s) - \mathbf{x}(s)^T w)^2] \tag{23}$$

Gradient descent finds a local minimum

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \bigtriangledown_{\mathbf{w}} J(\mathbf{w}) = \alpha \mathbb{E}_\pi[(V_\pi(s) - \hat{V}(s, \mathbf{w})) \bigtriangledown_{\mathbf{w}} \hat{V}(s, \mathbf{w}] \tag{24}$$

Stochastic gradient descent samples the gradient

$$\Delta \mathbf{w} = \alpha(V_\pi(s) - \hat{V}(s, \mathbf{w})) \bigtriangledown_{\mathbf{w}} \hat{V}(s, \mathbf{w} \tag{25}$$

The expected update is equal to full gradient update.

In case of linear function approximator, suppose state-value function can be represented as a linear combination of state feature $\mathbf{x}(s)$ and coefficient vector $\mathbf{w}$

$$\hat{v}(s) = \mathbf{x}(s)^T \mathbf{w} \tag{26}$$

Following stochastic gradient descent, $J(\mathbf{w})$ converges to global optimum. The update rule is particularly simple

$$\bigtriangledown_{\mathbf{w}} \hat{V}(s) = \mathbf{x}(s) \tag{27}$$

$$\Delta \mathbf{w} = \alpha(V_\pi(s) - \hat{V}(s))\mathbf{x}(s) \tag{28}$$

where the true value $V_\pi(s)$ is unknown and be replaced by TD-target

$$V_\pi(s_t) = r_{t+1} + \gamma \hat{V}_\pi(s_{t+1}) \tag{29}$$

The update step is

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha(r_{t+1} + \gamma \hat{V}_\pi(s_{t+1}) - \hat{V}_\pi(s_t))\mathbf{x}(s) \tag{30}$$

3 SARSA

SARSA is also a policy evaluation algorithm based on the Bellman equation of action value function $Q(s, a)$.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{31}$$

With function approximation, the action-value of approximated as

$$\hat{Q}(s, a, \mathbf{w} \approx Q_\pi(s, a) \tag{32}$$

The objective is to minimize mean-square error between approximate value and true value

$$J(\mathbf{w}) = \mathbb{E}_\pi[(Q_\pi(s, a) - \hat{Q})_\pi(s, a))^2] \tag{33}$$

Use stochastic gradient descent to find a local minimum.

$$\Delta \mathbf{w} = -\frac{1}{2} \bigtriangledown_{\mathbf{w}} J(\mathbf{w}) = \alpha(Q_\pi(s, a) - \hat{Q}_\pi(s, a)) \bigtriangledown_{\mathbf{w}} \hat{Q}_\pi(s, a) \tag{34}$$

In case of linear function approximation,

$$\hat{Q}(s, a) = \mathbf{x}(s, a)^T \mathbf{w} \tag{35}$$

where $\mathbf{x}(s,a)$ is the state-action pair feature. KG: Note that $\mathbf{x}(s,a)$ is typically unknown in MDP, as the observation at time is s_t which is the state feature. The state-action feature needs to be constructed somehow. See later discussion about coding.

The stochastic gradient descent update is

$$\Delta \mathbf{w} = \alpha(Q_\pi(s,a) - \hat{Q}_\pi(s,a))\mathbf{x}(s,a) \tag{36}$$

The update rule is

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha(r_{t+1} + \gamma \hat{Q}(s_{t+1}, a_{t+1}) - \hat{Q}(s_t, a_t))\mathbf{x}(s_t, a_t) \tag{37}$$

SARSA can be used for control (policy improvement), thanks to the state-action value $Q(s,a)$.

$$\pi'(a|s_t) = \arg\max_{a \in \mathcal{A}} \hat{Q}_\pi(s,a) \tag{38}$$

KG: This is on-policy control. The only difference with Q-Learning is that Q-Learning is off-policy control. See detail in Q-Learning later.

4 LSTD

The update rule of linear TD-learning is essential the incremental update of least-square estimation. Thus, $\mathbf{w}$ can be calculated by least-square estimator based on experience samples. The resulting algorithm is called LSTD.

LSTD finds parameter vector $\mathbf{w}$ minimising sum-squared error between $\hat{V}(s_t, \mathbf{W}$ and target values $V_\pi(s)$.

$$J(\mathbf{w}) = \sum_{t=1}^{T} (V_\pi(s_t) - \hat{V}(s_t, \mathbf{w}))^2 = \mathbb{E}_{\mathcal{D}}[(V_\pi(s) - \hat{V}(s, \mathbf{w})^2] \tag{39}$$

The solution is

$$\hat{\mathbf{w}} = (\sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^T)^{-1} \sum_{t=1}^{T} \mathbf{x}(s_t) V_\pi(s_t) \tag{40}$$

Plug $V_\pi(s_t) = r_{t+1} + \gamma \hat{V}(s_{t+1}, \mathbf{w})$ yields

$$\hat{\mathbf{w}} = \left(\sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t))^T - \gamma \mathbf{x}(s_{t+1}) \right)^{-1} \sum_{t=1}^{T} \mathbf{x}(s_t) r_{t+1} \tag{41}$$

5 LSPI

LSPI is an off-policy control algorithm following policy iteration principle which is the combination of LSTD-Q and policy improvement. In previous section, LSTD is policy evaluation based on state value function $V(s)$. This same process can be applied to action value $Q(s,a)$, the resulted algorithm is called LSTD-Q. The action-value of policy π is

$$\hat{\mathbf{w}} = \left(\sum_{t=1}^{T} \mathbf{x}(s_t, a_t)\mathbf{x}(s_t, a_t))^T - \gamma \mathbf{x}(s_{t+1}, \pi(s_{t+1})) \right)^{-1} \sum_{t=1}^{T} \mathbf{x}(s_t, a_t) r_{t+1} \tag{42}$$

To control, alternate LSTD-Q and policy improvement yields LSPI.

$$\pi'(s) = \arg\max_{a} \hat{Q}_\pi(s,a) \tag{43}$$

6 Q-Learning

Q-Learning is an off-policy control algorithm. Off-policy means that evaluate a target policy $\pi(a|s)$ to compute $V_\pi(s)$ or $Q_\pi(s,a)$, while following behaviour policy $\mu(a|s)$

$$\{s_1, a_1, r_2, ..., s_t\} \sim \mu \tag{44}$$

Why this important ?

- Learning from observing humans or other agents.
- Reuse experience generated from old polices $\pi_1, \pi_2, ..., \pi_{t-1}$.
- Learn about optimal policy while following exploratory policy.
- Learn about multiple policies while following on policy.

TD-learning can be used as off-policy with important sampling.

$$V(s_t) \leftarrow V(s_t) + \alpha\left(\frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}\big(r_{t+1} + \gamma V(s_{t+1})\big) - V(s_t)\right) \tag{45}$$

Off-policy learning (evaluation) of action-value $Q(s, a)$.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a') - Q(s_t, a_t) \tag{46}$$

where the behaviour policy gives the next action $a_{t+1} \sim \mu(a|s_t)$, the target policy gives an alternative action $a' \sim \pi(a|s)$ where $a' \neq a_{t+1}$.

KG: This is off-policy action value evaluation.

Combine the above with policy improvement step yields `Q-Learning`. The target policy is greedy w.r.t $Q(s, a)$

$$\pi(s_t) = \arg\max_a Q(s_t, a) \tag{47}$$

The behaviour policy μ is ϵ-greedy w.r.t $Q(s, a)$.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right) \tag{48}$$

KG: This update rule mergers the policy evaluation and policy improvement in one place.

7 REINFORCE

`REINFORCE` is a Monte-Carlo Policy gradient algorithm. Before show the detail of `REINFOCE`, we first lay more foundations of policy gradient algorithms.

As discussed in section 1. The policy $\pi(s, \boldsymbol{\theta})$ is parameterized by $\boldsymbol{\theta}$. Policy-based algorithm is an optimization problem: find $\boldsymbol{\theta}$ maximizing objective $J(\boldsymbol{\theta})$.

In episodic case, the performance measure J is the value of the start state of the episode.

$$J(\boldsymbol{\theta}) = V_{\pi_{\boldsymbol{\theta}}}(s_0) \tag{49}$$

In continuing case, the performance measure is defined as the average reward rate.

$$J(\boldsymbol{\theta}) = r(\pi_{\boldsymbol{\theta}}) = \sum_a \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)r \tag{50}$$

Another objective for continuous case is average value

$$J(\boldsymbol{\theta}) = \sum_s d_\pi(s) V_\pi(s) \tag{51}$$

where $d_\pi(s)$ is the stationary distribution of Markov chain for policy π.

The optimization problem can be solved by gradient descent to search local optimum. What we need is the gradient of $\boldsymbol{\theta}$ w.r.t the objective function.

$$\Delta\boldsymbol{\theta} = \alpha \bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{52}$$

where $\bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is the policy gradient.

KG: How to compute the policy gradient analytically?
Assume the policy $\pi_{\boldsymbol{\theta}}$ is differentiable whenever it is non-zero and we know the gradient $\bigtriangledown_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s, a)$. The likelihood ratios is defined as

$$\bigtriangledown_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s, a) = \pi_{\boldsymbol{\theta}}(s, a)\frac{\bigtriangledown_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s, a)}{\pi_{\boldsymbol{\theta}}(s, a)} = \pi_{\boldsymbol{\theta}} \bigtriangledown_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s, a) \tag{53}$$

The score function is defined as $\bigtriangledown_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s, a)$.

Remember that what we want is policy gradient $\bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. **Policy Gradient Theorem** links policy gradient and score function.

Theorem 1. ***(Policy Gradient Theorem)*** *For any differentiable policy $\pi_{\boldsymbol{\theta}}(s,a)$, for any of the policy objective functions defined above, the policy gradient is*

$$\bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\Big[\bigtriangledown_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a) Q_{\pi_{\boldsymbol{\theta}}}(s,a)\Big] \tag{54}$$

Now, we turn our attention to REINFORCE exploits Theorem 1 directly. However, there is an issue that $Q_{\pi_\theta}(s,a)$ is unknown. In REINFORCE, $Q_{\pi_{\boldsymbol{\theta}}}(s,a)$ is estimated at the end of each episode where

$$\hat{Q}(s_t,a_t) = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{T-1} r_T \tag{55}$$

At the end of each eposide, the update rule is applied to each state-actio pair along the episode

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \bigtriangledown_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_t,a_t) \hat{Q}(s_t,a_t) \tag{56}$$

8 Actor-Critic

In REINFORCE, $Q(s,a)$ is estimated in Monte-Carlo way. This results in high variance. However, $Q(s,a)$ can be approximated by another function $Q_{\mathbf{w}}(s,a)$. The rest are the same as REINFORCE, this leads to Actor-Critic where $\pi_{\boldsymbol{\theta}}(s,a)$ is called Actor and $Q_{\mathbf{w}}(s,a)$ is called Critic. In such way, $Q(s,a)$ can be estimated at any step instead of waiting until the end of episode. Specifically, After each transition step $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, the following update rules apply

$$\delta = r_{t+1} + \gamma \hat{Q}_{\mathbf{w}}(s_{t+1}, a_{t+1}) - \hat{Q}_{\mathbf{w}}(s_t, a_t) \tag{57}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \bigtriangledown_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_t,a_t) \hat{Q}(s_t,a_t,\mathbf{w}) \tag{58}$$

$$\mathbf{w} \leftarrow \mathbf{w} + \beta \delta \bigtriangledown_{\mathbf{w}} \hat{Q}_{\mathbf{w}}(s_t,a_t) \tag{59}$$

where α, β are learning step size. In linear case, $\bigtriangledown_{\mathbf{w}} \hat{Q}_{\mathbf{w}}(s_t,a_t) = \mathbf{x}(s_t,a_t)$.

High variance can be reduced using a baseline $B(s)$. It requires that $B(s)$ reduces variance without changing expectation. A good baseline is the state value function $B(s) = V(s)$. Thus the policy gradient can be written using advantage function

$$\bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\bigtriangledown_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a)(Q_{\pi_{\boldsymbol{\theta}}}(s,a) - V_{\pi_{\boldsymbol{\theta}}}(s)] = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\bigtriangledown_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a) A_{\pi_{\boldsymbol{\theta}}}(s,a)] \tag{60}$$

where $A_{\pi_{\boldsymbol{\theta}}}(s,a) = Q_{\pi_{\boldsymbol{\theta}}}(s,a) - V_{\pi_{\boldsymbol{\theta}}}(s)$.
Note that TD-error is an unbiased estimate of advantage.

$$\delta = r + \gamma V(s_{t+1}) - V(s_t) \tag{61}$$

$$\pi[\delta] = \pi[r + \gamma V(s_{t+1}) - V(s_t)] = Q(s_t,a_t) - V(s_t) \tag{62}$$

Thus, in practice, Advantage Actor-Critic consists two function approximators: $\pi_{\boldsymbol{\theta}}$ and $V_{\mathbf{w}}$.

KG: Why does baseline reduce variance? any proof?

An potential issue of Actor-Critic is the bias introduced when approximating the policy gradient.
KG: Why it is biased?.
The policy gradient is approximated as

$$\bigtriangledown_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \mathbf{E}_{\pi_{\boldsymbol{\theta}}}[\bigtriangledown_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a) Q_{\mathbf{w}}(s,a)] \tag{63}$$

This is approximation, as $Q_{\mathbf{w}}(s,a) \approx Q(s,a)$.

KG: Does REINFORCE suffer the same issue? No, as Monte-Carlo estimation $\hat{Q}(s,a)$ is an unbiased estimator.

Luckily, if we choose value function approximation carefully, we can avoid introducing any bias. The following gives some conditions.

Theorem 2. *(Compatible function approximation Theorem) If the following two conditions are satisfies:*

- *Value function is compatible to the policy:*

$$\bigtriangledown_{\mathbf{w}} Q_{\mathbf{w}}(s,a) = \bigtriangledown_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a) \tag{64}$$

- *Value function parameters* $\mathbf{w}$ *minimize the mean-squared error:*

$$\mathbf{E}_{\pi_{\boldsymbol{\theta}}}[(Q_{\pi_{\boldsymbol{\theta}}}(s,a) - Q_{\mathbf{w}}(s,a))^2] \tag{65}$$

Then, the policy gradient is exact.

KG: AC for continuous action space

9 DQN

DQN is basically Q-Learning with function approximator in the form of neural network. In addition, to deal with highly correlated sample, replay buffer and delay-updated target net are introduced.

As the neural network is updated by SGD, but the fundamental requirements for SGD optimization is that the training data is i.i.d. In our case, the data samples are not independent and they will be close to each other as they belong to the same episode. In addition, the distribution of our training data won't be identical to samples provided by the optimal policy that we want to learn.

To deal with this issue,we usually need to use a large buffer of our past experience and sample training data from it, instead of using our latest experience. This is called replay buffer.

Another issue is the correlation between steps. To make the training more stable, there is a trick called target network, when we keep a copy of our network and use it for $Q(s', a')$ is the bellman equation, This network is synchronized with out main network only periodically.

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the Q function .This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values $r + \gamma \max_a Q(s,a)$*). We address these instabilities with a novel variant of Q-learning, which uses two key ideas. First, we used a biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution (see below for details). Second, we used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target.*[12]

KG: Update rule of DQN

10 {n-step, Double, Noisy, Dueling, Categorical}-DQN

In this section, we discuss variants of DQN.
N-step DQN: The bellman equation can be unrolled into many steps in the fashion as

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a) = r_t + \gamma r_{t+1} + \gamma^2 \max_a Q(s_{t+2}, a) \tag{66}$$

In the case that reward only received at the end of an episode, unroll the update rule many steps can improve the propagation speed of values which improves the convergence. KG: This is similar to the idea of eligibility trace. However, how many steps to unroll is a matter of hyper-parameter tuning.

If too many steps are unrolled, DQN will fail to converge as in the intermediate steps, the max operations are omitted which means that we assume the corresponding selected actions are optimal. However, we have no guarantee to assume this is true. Indeed, at the beginning of the training, the agent acts randomly. In this case, the calculated (Q_s, a_t) maybe smaller than the optimal value of the state-action. Therefore, more unrolling steps leads to more inaccurate estimate. In addition, the large replay buffer makes it even worse, as it increases the chance of getting transitions obtained from old bad policies. This will lead to a wrong update of the current Q-value, as it can easily break our

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D

    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Figure 1: DQN Algorithm

training progress. Nevertheless, the convergence speeding up makes it a worth doing, though the number of unrolling steps is a matter of hyper-parameter tuning.

KG: Why off-policy works? Off-policy methods do not need freshness of data. For example, a simple DQN is off-policy, which means that we can use very old data sampled from the environment several million steps old and this data will still be useful fro learning. That is because we are just updating the value of the action value $Q(s_t, a_t)$ with immediate reward r_t, plus discounted current approximation of the best action's value. Even if the action was sampled randomly, it does not matter because for this particular action a_t, in the state s_t, our update will be correct. That is why in off-policy methods, we can use a very large experience reply buffer to make our data closer to being i.i.d.

On the other hand, on-policy methods heavily depend on the training data to be sampled according to the current policy we are updating. That happens because on-policy methods are trying to improve the current policy indirectly or directly.

Double DQN: It is noticed that DQN tends to overestimate Q-value which may be harmful to training performance and sometimes can lead to sub-optimal policies. The reason is that the max operation in the Bellman equation. In DQN, the upate rule is

$$Q(s_t, a_t) = r_t + \gamma \max_a Q'(s_{t+1}, a) \tag{67}$$

$Q'(s_{t+1}, a)$ is calculated by target net. The author proposed choosing actions for the next state using the trained network instead of the target net, but calculate the Q-value using the target net.

$$Q(s_t, a_t) = r_t + \gamma Q'(s_t, \arg\max_a Q(s_{t+1}, a)) \tag{68}$$

The authors proved that this simple tweak fixed overestimation completely and they called this new architecture Double DQN.

KG: Why DQN overestimates the Q-value, how to proof?

Noisy networks: To balance the trade-off of exploration and exploitation, DQN employs ϵ-greedy strategy. The author proposed to add noise to the weight of fully-connected layer and adjust the

parameters of this noise during training using back-propagation. The authors proposed two ways of adding noises.

- Independent Gaussian noise: For every weight in the fully-connected layer, we have a random value that we draw from the normal distribution.
- Factorized Gaussian noise: To minimize the amount of random values to be samples, the author proposed keeping only two random vectors, one with the size of input and another with the size of the output of the layer. Then, a random matrix for the layer is created by calculating the outer product of the vectors.

Prioritized replay buffer: This method tries to improve the efficiency of samples in the replay buffer by prioritized those samples according to the training loss. The author of the paper questioned this uniform random sample policy and proved that by assigning priorities to buffer samples, according to training loss and sampling the buffer proportional to those priorities, we can significantly improve convergence and the policy quality of the DQN. The method can be seen as train on data that surprises you. The tricky point here is to keep the balance of training on an unusual sample and training on the rest of the buffer.

Dueling DQN: The paper noticed that Q-value can be divided into state value and advantage $Q(s,a) = V(s) + A(s,a)$. The advantage represents how much extra reward some particular action than in average in the state.

The paper learns the state value and advantage value separately. The networks has two heads one outputs state value the other output advantage value. The combination gives the Q-value. However, we need to make sure the average advantage is zero. This is achieved by $Q(s,a) = V(s) + A(s,a) - 1/N \sum_k A(s,k)$.

Categorical DQN: `DQN` output the Q-value of actions. This paper challenges that the distribution of Q-values of actions are more informative in particular in complex environment.
KG: Why distribution of Q-value of each action could help?

11 Rainbow

`Rainbow` is an algorithm employing all the tricks described in the previous section.

12 Agent 57

We propose `Agent57`, the first deep RL agent that outperforms the standard human benchmark on all 57 Atari games. To achieve this result, we train a policies ranging from very exploratory to purely exploitative. We propose an adaptive mechanism to choose which policy to prioritize throughout the training process. Additionally, we utilize a novel parameterization of the architecture that allows for more consistent and stable learning.

13 A3C

`A3C`: Asynchronous Advantage Actor-Critic, an on-policy policy-gradient algorithm, gathers transitions (samples) from several parallel environments, all of them exploiting the current policy. The motivation of this setting is to improve the stability of policy gradient method.

The reason behind is the correlation between samples, which breaks the i.i.d assumption of SGD. The negative consequence of correlated samples is very high variance in gradients, which means our training batch contains very similar examples, all of them pushing our network in the same direction. However, this direction may be totally the wrong direction in the global sense, all all those examples could be from sub-optimal policies.

KG: Can we use replay buffer as in `DQN`? No, as `Actor-critic` is an on-policy algorithm. Samples from other polices are not useful.
KG: Why on-policy algorithm can only use samples from the current policy?

In A3C, samples are collected from several parallel environment while following the same current policy. This breaks the correlation within one single episode, as we now train on several episodes

obtained from different environments. As the same time, we are still using the same current policy. A disadvantage is sample inefficiency, as we throw away all experiences that we have just got after one single training. KG: This issue will be dealt with by DPG, which transform policy-gradient method to be off-policy.

In terms of implementation, KG: How samples are gathered together from different environments? KG: How the actor net and critic net are updated?

We propose a conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controller. We present asynchronous variants of four standard reinforcement learning algorithms and show that parallel actor-learners have a stabilizing effect on training allowing all four methods to successfully training neural network controller. The best performing methods, A3C, surpasses the current state-of-the-art on the Atari domain.

KG: The Asynchronous RL Framework

We now present multi-threaded asynchronous variants of on-step SARSA, one-step Q-learning, n-step Q-learning and advantage actor-critic. The aim in designing these methods was to find RL algorithms that can train deep neural network policies reliably and without large resource requirements. While the underlying RL methods are quite different, with actor-critic being an on-policy method and Q-learning being an off-policy value-based method, we use two main ideas to make all four algorithms practical given our design goal.

First, we use asynchronous actor-critics, where we use multiple CPU threads on a single machine. Keeping the learners on a single machine removes the communication costs of sending gradients and parameters and enables us to use Hogwild! style updates for training. KG: What is Hogwild!?.

Second, we make the observation that multiple actors learners running in parallel are likely to be exploring different parts of the environment. Moreover, one can explicitly use different exploration policies in each actor-learner to maximize this diversity. By running different exploration policies in different threads, the overall changes being made to the parameters by multiple actor-learners applying online updates in parallel are likely to be less correlated in time than a single agent applying online updates. Hence, we do not user a replay buffer and rely on parallel actors employing different exploration policies to perform the stabilizing role undertaken by experience replay in the DQN training algorithm.

KG: On-policy needs data from the current policy, what happen if the actor learners follows different exploration policies?

In addition to stabilizing learning, using multiple parallel actor learners has multiple practical benefits. First, we obtain a reduction in training time that is roughly linear in the number of parallel actor-learners. Second, since we no linger rely on experience replay, we are able to use on-policy reinforcement learning methods such as actor-critic to train neural network in a stable way.

KG: Asynchronous advantage actor-critic A3C

The algorithm maintains a policy $\pi(a_t|s_t, \boldsymbol{\theta})$ and an estimate of the value function $V(s_t, \boldsymbol{\theta}_v)$. This algorithm operates in the forward view and uses the same mix of n-step returns to update both the policy and the value-function. The policy and the value function are updated after every t_{max} actions or when a terminal state is reached. The update performed by the algorithm ca be seen as

$$\bigtriangledown_{\boldsymbol{\theta}'} \log \pi(a_t|s_t, \boldsymbol{\theta}')A(s_t, a_t, \boldsymbol{\theta}, \boldsymbol{\theta}'_v) \tag{69}$$

where $A(s_t, a_t, \boldsymbol{\theta}, \boldsymbol{\theta}'_v)$ is an estimate of the advantage function given by

$$\sum_{i=0}^{k-1} \gamma^i r_{t+1} + \gamma^k V(s_{t+k}, \boldsymbol{\theta}_v) - V(s_t, \boldsymbol{\theta}_v) \tag{70}$$

where k can vary from state to state and is upper-bounded by t_{max}.

As with the value-based methods we rely on parallel actor-learners and accumulated updates to improving training stability. Note that while the parameters $\boldsymbol{\theta}$ of the policy and $\boldsymbol{\theta}_v$ of the value function are shown as being separate for generality, we always share some of the parameters in practice. We typically use a convolutional neural network that has one softmax output for the policy $\pi(a_t|s, \boldsymbol{\theta})$ and one linear output for the value function $V(s_t, \boldsymbol{\theta}_v)$ with all non-output layers shared.

We also found that adding the entropy of the policy to the objective function improved exploration by discouraging premature convergence to sub-optimal deterministic policies.The gradient of the full objective function including the entropy regularization term with respect to the policy parameters take the form

$$\bigtriangledown_{\boldsymbol{\theta}'} \log \pi(s_t|s_t, \boldsymbol{\theta}')(R_t - V(s_t, \boldsymbol{\theta}_v) + \beta \bigtriangledown_{\boldsymbol{\theta}'} H(\pi(s_t; \boldsymbol{\theta}')) \tag{71}$$

where $H(\cdot)$ is the entropy. The hyper-parameter β controls the strength of the entropy regularization term.

KG: Other Tricks

- We use a shared and slowing changing target network in computing the loss.
- We also accumulate gradients over multiple time-steps before they are applied, which is similar to using mini-batches. This reduces the chance of multiple actor learners overwriting each other's updates. Accumulating updates over several steps also provides some ability to trade-off computational efficiency for data efficiency.
- Finally, we found that giving each thread a different exploration policy helps improve the robustness. While there are many possible ways of making the exploration policies differ we experiment with using ϵ-greedy exploration with ϵ periodically sampled from some distribution by each thread.
- We use the same mix n-step returns to update the policy and the value-function.
- We typically use a convolutional neural network that has one soft-max output for the policy and one linear output for the value function, with all non-linear layers shared.
- We also found that adding entropy of the policy to the objective function improved exploration by discouraging pre-mature convergence to sub-optimal deterministic policies.

Algorithm S2 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t; \theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i; \theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 2: A3C Algorithm

14 A2C

A2C a synchronous, deterministic variant of Asynchronous Advantage Actor Critic (A3C) which OpenAI found gives equal performance. This algorithm is shown in the [Link].

15 ACER

ACER is the off-policy counterpart of A3C.

16 PPO

PPO: Proximal Policy Optimiaztion is an on-policy policy gradient algorithm like Actor-Critic , A3C and A2C. PPO is an improvement over A3C, which changes the expression used to estimate the policy gradient. Instead of the gradient of logarithm probability of the action taken, the PPO method uses a different objective: the ratio between the new and old policy policy scales by the advantage.

To understand PPO, we start from policy gradient method, and then move to TROP. Next, we discuss the improvement of PPO over TROP.

Policy gradient methods:
Policy gradient methods compute the the gradient of policy and uses stochastic gradient ascent to update the policy. The most commonly used gradient estimator is

$$\hat{g} = \hat{\mathbb{E}}_t[\bigtriangledown_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\hat{A}_t] \tag{72}$$

where $\pi_{\boldsymbol{\theta}}$ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at timestep t. Here, the expectation indicates the empirical average over a finite batch of samples, in an algorithm that alternates between sampling and optimization.

The estimator $\hat{g}$ is obtained by differentiating the objective

$$L^{PG}(\boldsymbol{\theta} = \hat{\mathbb{E}}_t[\log \pi_{\boldsymbol{\theta}}(a_t|s_t)\hat{A}_t] \tag{73}$$

While it is appealing to perform multiple steps of optimization on this loss using the same trajectory, doing so is not well-justified, and empirically it often leads to destructively large policy updated.

In **TRPO** an objective is maximized subject to a constraint on the size of policy update.

$$\max_{\boldsymbol{\theta}} \hat{\mathbb{E}}_t \left[\frac{\pi_{\boldsymbol{\theta}}(a_t|s_t}{\pi_{\boldsymbol{\theta}_{old}}(a_t|s_t)} \hat{A}_t \right] \tag{74}$$

$$subject\ to\ \hat{\mathbb{E}}_t[KL[\pi_{\boldsymbol{\theta}_{old}}(\cdot|s_t), \pi_{\boldsymbol{\theta}}(\cdot|s_t)]] \leq \delta$$

Here $\boldsymbol{\theta}_{old}$ is the vector of policy parameters before the update.

The main objective proposed by PPO is

$$L^{CLIP}(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[\min \left(\frac{\pi_{\boldsymbol{\theta}}(a_t|s_t}{\pi_{\boldsymbol{\theta}_{old}}(a_t|s_t)} \hat{A}_t, clip(\frac{\pi_{\boldsymbol{\theta}}(a_t|s_t}{\pi_{\boldsymbol{\theta}_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{75}$$

where ϵ is a hyper-paramter. The motivation of this objective is as follows:

- The first term is the same as TRPO.
- The second term modifies the surrogate objective by clipping the probability ratio, which removes the incentive for moving $r_t = \frac{\pi_{\boldsymbol{\theta}}(a_t|s_t}{\pi_{\boldsymbol{\theta}_{old}}(a_t|s_t)}$ outside of the interval $[1-\epsilon, 1+\epsilon]$.
- Finally, we take the minimum of the clipped and unclipped objective, so the final objective is a lower bound on the unclipped objective.

KG: What does the objective function try to achieve?
With this scheme, we only ignore the change in the ratio when it would make the objective improve and we include it makes the objective worse. Note that the ratio is clipped at $1-\epsilon$ or $1+\epsilon$ depending on whether the advantage is positive or negative.

KG: How to compute the advantage?
Regarding to computing the advantage, most techniques for computing variance-reduced advantage-function estimation make use a learned stat-value function $V(s)$. We use a truncated version of generalised advantage estimation, which reduce to n-step unrolling advantage when $\lambda = 1$

$$\hat{A}_t = \delta_t + \gamma\lambda\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{76}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$.

In contrast, A3C uses n-step unrolling advantage.

$$A_t = r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-1} V(s_T) - V(s_t) \tag{77}$$

KG: The overall objective function if actor and critic share the same neural network architecture
If using a neural network architecture that shares parameters between the policy and value function, we must use a loss function that combines the policy surrogate and a value function error term. This objective can further be augmented by adding an entropy bonus to ensure sufficient exploration.

$$L(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[L_t^{clip} - c_1 L_t^{VF} + c2S[\pi_{\boldsymbol{\theta}}](s_t) \right] \tag{78}$$

S denotes an entropy bonus, and L_t^{VF} is a squared-error loss $(\hat{V}(s_t) - V_t^{target})^2$.

KG: How to updated via SGD?
PPO algorithm that uses fixed-length trajectory segments is shown below. Each iteration, each of N actors collect T time-steps of data. Then we construct the surrogate loss on these NT time-steps of data, and optimize it with mini-batch SGD for K epoches. KG: The length of trajectory is NT, which is then divided in to K segments, each segment is used to calculate the loss ad gradient to update the model (mini-batch SGD).

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 3: PPO Algorithm

17 DPG

DPG: Deterministic policy gradients which is a variation of Actor-Critic, but being off-policy. It means we can have a huge replay buffer which improve the sample efficiency greatly.

This paper proposed a off-policy deterministic policy gradient algorithm for RL problems with continuous action space. The algorithm is an Actor-Critic typed algorithm which has an actor provides the action (output) to take under an state (input) and a critic while estimate the action value via Q-learning. The exploration is ensured by off-policy learning following behaviour polices (more exploration).

The key difference between DPG and Actor-Critic that the actor of Actor-Critic estimate the stochastic policy which returns the distribution of actions while actor of DPG estimate the deterministic policy which returns the action to take.

KG: Why important sampling can be avoided?
We note that stochastic off-policy actor-critic algorithms typically use important sampling for both actor and critic. However, because the deterministic policy gradient removes the integral over actions, we can avoid importance sampling in the actor, and by using Q-learning, we can avoid importance sampling in the critic.

KG: How to update actor via SGD?
Let $a = u(s)$ denote the actor which returns the action a to take under state s and $\hat{Q}(s, a)$ be the critic which estimates the action-value. We can substitute the actor into the critic as $\hat{Q}(s, a) = \hat{Q}(s, u(s))$. Note the objective function of policy gradient algorithm is exact the Q-value, therefore, we can update the parameter of actor $u_{\boldsymbol{\theta}}(s)$ to maximize Q-value. The gradient is

$$\bigtriangledown_{\boldsymbol{\theta}_Q} Q(s, \mu(s)) \bigtriangledown_{\boldsymbol{\theta}} \mu(s) \tag{79}$$

KG: The difference with A2C
Note that despite both A2C and DPG belonging to policy-gradient family, the way that critic is used

is different. In A2C, we used the critic as a baseline for the reward, so the critic is a optinal piece (without it, we get REINFORCE) and is used to improve the stability.

In DPG, as out policy is deterministic, we can now calculate the gradient from Q, obtained from the critic up to the actor weights, so the whole system is differentiable and could be optimized end-to-end with SGD. To update the critic network, we can use the Bellman equation to find the approximation of $Q(s, a)$ and minimize the MSE objective.

In summary, the critic is update in the same way as in A2C and the actor (policy) is updated in a way to maximize the critic's output. The essence is that the method is off-policy, so we can have a replay buffer for training.

KG: How to ensure exploration?
However, note that unlike in A2C the policy is stochastic, the exploration can be encourage by cross-entropy loss, the policy of DPG is deterministic, we need a mechanism to take care exploration. This can be done by adding noise to the action returned by the actor (policy) or following more exploratory behaviour polices

18 DDPG

DDPG is an extension of DPG, which combines DPG with neural network, replay buffer and delayed updated target network and batch normalization. It is an off-policy, policy gradient deterministic policy algorithm.

KG: The objective function
Q-learning, a commonly used off-policy algorithm, uses the greedy policy $\mu(s) = \arg\max_a Q(s, a)$. We consider function approximation parameterized $\boldsymbol{\theta}^Q$, which we optimize by minimizing the loss

$$L(\boldsymbol{\theta}^Q) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim \mathbb{E}}[(Q(s_t, a_t|\boldsymbol{\theta}^Q - y_t)^2] \tag{80}$$

where

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\boldsymbol{\theta}^Q) \tag{81}$$

while y_t is also dependent on $\boldsymbol{\theta}^Q$, this is typically ignored.

The use of large, non-linear function approximation for learning value or action-value functions has often been avoided in the past since theoretical performance guarantees are impossible, a practically learning tend so be unstable. Recently DQN adapted the Q-learning algorithm in order to make effective use of large neural networks as function approximators. In order to scale Q-learning they introduce two major changes: the use of a replay buffer, and a separate target network for calculating y_t. We employ these in the context of DDPG and explain their implementation in the next section.

DPG algorithm maintains a parameterized actor function $\mu(s|\boldsymbol{\theta}^\mu)$ which specifies the current policy by deterministically mapping states to a specific action. The critic $Q(s, a)$ is learned using the Bellman equation as in Q-learning. The actor is updated by following the applying the chain rule to the expected return from the start distribution J with respect to the actor parameters:

$$\begin{aligned} \bigtriangledown_{\boldsymbol{\theta}^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta}[\bigtriangledown_{\boldsymbol{\theta}} Q(s, a|\boldsymbol{\theta}^Q|s = s_t, a = \mu(s_t|\boldsymbol{\theta}^\mu)] \\ &= \mathbb{E}_{s_t \sim \rho^\beta}[\bigtriangledown_a Q(s, a|\boldsymbol{\theta}^Q) s = s_t, a = \mu(s_t) \bigtriangledown_{\boldsymbol{\theta}_\mu} \mu(s|\boldsymbol{\theta}^\mu)|s = s_t] \end{aligned} \tag{82}$$

KG: The update rule
Update the actor policy using the sampled policy gradient

$$\bigtriangledown_{\boldsymbol{\theta}^\mu} J \approx \frac{1}{N} \sum_i \bigtriangledown_a Q(s, a|\boldsymbol{\theta}^Q) \bigtriangledown_{\boldsymbol{\theta}^\mu} \mu(s|\boldsymbol{\theta}^\mu) \tag{83}$$

Update the target network

$$\boldsymbol{\theta}^{Q'} \leftarrow \tau \boldsymbol{\theta}^Q + (1 - \tau)\boldsymbol{\theta}^{Q'} \tag{84}$$

$$\boldsymbol{\theta}^{\mu'} \leftarrow \tau \boldsymbol{\theta}^\mu + (1 - \tau)\boldsymbol{\theta}^{\mu'} \tag{85}$$

KG: Replay Buffer
As in DQN, we used an replay buffer to address the issue of high correlated samples. Transitions are

sampled from the environment according to the exploration policy and stored in the replay buffer. At each time step the actor and critic are updated by sampling a minibatch uniformly from the buffer.
KG: Why target net?
We also used the target net in DQN. We create a copy of the actor and critic networks respectively, that are used for calculating the target values. The weights of these target networks are then updated by having them slowly track the learned networks:

$$\boldsymbol{\theta}' \leftarrow \tau\boldsymbol{\theta} + (1-\tau)\boldsymbol{\theta}' \tag{86}$$

where $\tau << 1$. This means that the target values are constrained to change slowly, greatly learning the action-value function closer to the case of supervised learning, a problem for which robust solutions exist.

KG: Why batch normalization?
When learning from low dimensional feature vector observations, the different components of the observation may have different physical units and the range may vary across environment. We address this issue by adapting batch normalization. Batch Normalization normalizes each dimension across the samples in a mini-batch to have unit mean and variance. In addition, it maintains a running average of the mean and the variance to use for normalization during testing. In deep network, it is used to minimize co-variance shift during training by ensuring that each layer receives whitened input. In the low-dimensional case, we used batch normalization on the state input and all layers of actor network and all layers of critic network prior to the action input. With batch normalization, we were able to learn effectively across many different tasks with differing types of units, without needing to manually ensure the units were within a set range.

KG: Exploration
A major challenge of learning in continuous actions space is exploration. We construct an exploration policy μ' by adding noise sampled from a noise process $\mathcal{N}$ to the actor policy

$$\mu'(s_t) = \mu(s_t) + \mathcal{N} \tag{87}$$

The noise process is Ornstein-Uhlenbeck process.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 4: DDPG Algorithm [7]

19 D4PG

D4PG: Distributed Distributional Deep Deterministic Policy Gradient is in the same sense of Rainbow, which mergers various tricks into DDPG.

- It adapts the distributional representation of Q-value.
- It uses n-step Bellman equation.
- It uses the prioritized replay buffer.

20 TD3

TD3: Twin Delayed DDPG is an improvement over DDPG which focus on the overestimation of action-value in policy-based algorithms. The issue is only exploited in value-based algorithms like Double DQN before.

KG: The theoretical insight
In the value-based RL methods such as deep Q-learning, function approximation errors are known to lead to overestimated value estimates and sub-optimal policies. We show that this problem persists in an actor-critic setting and propose novel mechanisms to minimize its effects on both actor and critic.

KG: The motivation of TD3
While DDPG can achieve great performance sometimes, it is frequently brittle with respect to hyper-parameters and other kinds of tuning. A common failure mode for DDPG is that the learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function.

KG: The ket components of TD3
Twin Delayed DDPG (TD3) is an algorithm that addresses this issue by introducing three critical tricks:

- Clipped Double-Q Learning. TD3 learns two Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.
- "Delayed" Policy Updates. TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.
- Target Policy Smoothing. TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

KG: Update Rule of TD3

21 Soft Q-Learning

We show that there is a precise equivalence between Q-learning and policy gradient methods in the setting of entropy-regularized reinforcement learning, that soft Q-learning is exactly equivalent to a policy gradient method. We also point out a connection between Q-learning methods and natural policy gradient methods.

Entropy-Regularized RL Let us define the entropy-augmented return

$$R(s_0) = \sum_{t=0}^{\infty} \gamma^t (r_t - \tau D_{KL}[\pi_t(s_t)||\bar{\pi}(s_t)])) \tag{88}$$

where r_t is the reward, $\gamma \in [0, 1]$ is the discount factor, τ is a scalar coefficient and $KL_t = D_{KL}[\pi_t(s_t)||\bar{\pi}(s_t)]$ is the KL divergence between the current policy π_t and a reference policy $\bar{\pi}$. KG: What is the motivation of entropy-regularized RL setting?. KG: Why the reference policy is needed?.

Maximum entropy reinforcement learning alters the RL objective, though the original objective can be recovered using a temperature parameter. More importantly, the maximum entropy formulation provides a substantial improvement in exploration and robustness. Maximum entropy policies are robust in the face of model and estimation errors, and improve exploration by acquiring diverse behaviors.

The state-value function

$$V_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t (r_t - \tau D_{KL}[\pi_t(s_t)||\bar{\pi}(s_t)]))|s_0 = s\right] \tag{89}$$

Algorithm 1 Twin Delayed DDPG

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute target actions

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      if j mod `policy_delay` $= 0$ **then**
        Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

        Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

      end if
    end for
  end if
until convergence

Figure 5: TD3 Algorithm

The Q-function as

$$Q_\pi(s, a) = \mathbb{E}\left[r_0 + \sum_{t=0}^{\infty} \gamma^t (r_t - \tau D_{KL}[\pi_t(s_t)||\bar{\pi}(s_t)]))|s_0 = s, a_0 = a\right] \tag{90}$$

The relationship between Q_π and V_π

$$V_\pi(s) = \mathbb{E}_{a \sim \pi}[Q_\pi(s, a)] - \tau KL(s) \tag{91}$$

In the entropy-regularized setting, we also need to define the greedy policy, as the optimal policy is stochastic.

The Boltzmann policy

$$\pi_Q^{\mathcal{B}}(\cdot|s) = \arg\max_\pi \{\mathbb{E}_{a \sim \pi}[Q_\pi(s, a)] - \tau KL(s)\} = \frac{\bar{\pi}(a|s) \exp(Q(s, a)/\tau)}{\mathbb{E}_{a' \sim \pi}[\exp(Q(s, a')/\tau]} \tag{92}$$

Bellman equation
We can generalize the $\mathcal{T}_\pi$ operators in standard reinforcement learning correspond to computing the expected return with a non-step look-ahead: they take the expectation over one step of dynamics, and them fall back on the value function at the next time-step. We can easily generalize these operators to the entropy-regularized setting. We define

$$\mathcal{T}_\pi V(s) = \mathbb{E}_{a \sim \pi, (r, s') \sim P(r, s'|s, a)}[r - \tau KL(s) + \gamma V(s')] \tag{93}$$

$$\mathcal{T}_\pi Q(s,a) = \mathbb{E}_{(r,s')\sim P(r,s'|s,a)}[r + \gamma(\mathbb{E}_{a'\sim\pi})[Q(s',a')] - \tau KL(s')] \tag{94}$$

The 'Bellman error' is defined as

$$\delta_t = (r_t - \tau KL_t) + \gamma V(s_{t+1}) - V(s_t) \tag{95}$$

then we have

$$\mathcal{T}_\pi^n V(s) = \mathbb{E}[\sum_{t=0}^{n-1} \gamma^t \delta_t + \gamma^n V(s_n)|s_o = s] \tag{96}$$

Policy Gradient Algorithm

22 SAC

In this paper, we propose soft-actor-critic, an off-policy actor-critic deep RL algorithm based on the maximum entropy RL framework. In this framework, the actors aims to maximize expected reward while also maximizing entropy. That is, to. succeed at the task while acting as randomly as possible. Prior deep RL methods based on this framework have been formulated as Q-learning methods. By combining off-policy updates with a stable stochastic actor-critic formulation, our method achieves state-of-the-art performance on a range of continuous control benchmark tasks.

We explore how to design an efficient and stable model-free deep RL algorithm for continuous state and action spaces. To that end, we draw on the maximum reward framework, which augments the standard maximum reward RL objectives with an entropy maximization term.

In this paper, we demonstrate that we can devise an off-policy maximum entropy actor-critic algorithm, which we call SAC, which provides for both sample-efficient learning and stability. The algorithm consists of 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.

KG: Maximum Entropy Reinforcement Learning
Standard RL maximizes the expected sum of rewards

$$\sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t)] \tag{97}$$

We consider a more general maximum entropy objective which favors stochastic policies by augmenting the objective with the expected entropy of the policy over $\rho_\pi(s_t)$.

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))] \tag{98}$$

The objective has a number of conceptual and practical advantages:

- First, the policy is incentivized to explore more widely, while giving up on clearly unpromising avenues.
- Secon, the policy can capture multiple modes of near-optimal behavior. In problem settings where multiple actions seems equally attractive, the policy will commit equal probability mass to those actions.
- Lastly, prior work has observed improved exploration with this objective, and in our experiments we observe that it considerably improves learning speed.

KG: Soft policy iteration
In the policy evaluation step of soft policy iteration, we compute the value of a policy π according to the maximum entropy objective. The soft Q-value can be compuated iteratively.

$$\mathcal{T}^\pi Q(s_t,a_t) = r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim\rho}[V(s_{t+1})] \tag{99}$$

where

$$V(s_t) = \mathbb{E}_{a_t\sim\pi}[Q(s_t,a_t) - \log\pi(a_t|s_t)] \tag{100}$$

is the soft state value function.

In the policy improvement step, we update the policy towards the exponential of the new Q-function. We update the policy according to

$$\pi_{new} \arg\min_{\pi' \in \Pi} D_{KL}\left(\pi'(\cdot|s_t)||\frac{exp(Q^{\pi_{old}}(s_t,\cdot)}{Z^{\pi_{old}}(s_t)}\right) \tag{101}$$

Since in practice we prefer policies that are tractable, we will additionally restrict the policy to some set of policies Π, which can correspond to a parameterized family of distributions such as Gaussians.

The full soft policy iteration algorithm alternates between the soft policy evaluation and the soft policy improvement step, and it will provably converge to the optimal maximum entropy policy among the policies in Π.

KG: Soft Actor-Critic SAC

We use function approximators for bot the Q-function and policy. We consider a parameterized state value function $V_\psi(s_t)$, soft Q-function $Q_\theta(s_t, a_t)$ and a policy $\pi_\phi(a_t|s_t)$.

The soft value function $V_\psi(s_t)$ is trained to minimized the squared residual error

$$J_V(\psi) = \mathbb{E}_{s_t \sim \mathcal{D}}[\frac{1}{2}(V_\psi(s_t) - \mathbb{E}_{a_t \sim \pi_\phi}[Q_\theta(s_t, a_t) - \log \pi_\phi(a_t|s_t)])^2] \tag{102}$$

The gradient is

$$\bigtriangledown_\psi J_V(\psi) = \bigtriangledown_\psi V_\psi(s_t)(V_\psi(s_t) - Q_\theta(s_t, a_t) + \log \pi_\phi(a_t|s_t)) \tag{103}$$

The soft Q-function is trained to minimize the soft Bellman residual.

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim\mathcal{D}}[\frac{1}{2}(Q_\theta(s_t, a_t) - \hat{Q}(s_t, a_t))^2] \tag{104}$$

with

$$\hat{Q}(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1}\sim\rho}[V_{\bar{\psi}}(s_{t+})] \tag{105}$$

The gradient is

$$\bigtriangledown_\theta J_Q(\theta) = \bigtriangledown_\theta Q_\theta(s_t, a_t)(Q_\theta(s_t, a_t) - r(s_t, a_t) - \gamma V_{\bar{\psi}}(s_{t+1})) \tag{106}$$

The policy is learned by minimizing the expected KL-divergence

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D}}[D_{KL}(\pi_\phi(\cdot|s_t)||\frac{exp(Q_\theta(s_t,\cdot))}{Z_\theta(s_t)})] \tag{107}$$

We use a reparameterization trick resulting in a lower variance estimator. To that end, we reparameterize the olicy using a neural network transformation

$$a_t = f_\phi(\epsilon_t, s_t) \tag{108}$$

where ϵ_t is an input noise vector, sampled from some fixed distribution, such as spherical Gaussian. The objective is

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D},\epsilon_t\sim\mathcal{N}}[\log \pi_\phi(f_\phi(\epsilon_t; s_t)|s_t) - Q_\theta(s_t, f_\phi(\epsilon_t, s_t)] \tag{109}$$

where π_ϕ is defined implicitly in terms of f_ϕ. The gradient is

$$\bigtriangledown_\phi J_\pi(\phi) = \bigtriangledown_\phi \log \pi_\phi(a_t|s_t) + (\bigtriangledown_{a_t} \log \pi_\phi(a_t|s_t) - \bigtriangledown_{a_t} Q(s_t, a_t)) \bigtriangledown_\phi f_\phi(\epsilon_t; s_t) \tag{110}$$

Where a_t is evaluated at $f_\phi(\epsilon_t, s_t)$.

This algorithm also makes user of two Q-functions to mitigate positive bias (overestimate) in the policy improvement step that is known to degrade the performance of value based methods.

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right) \Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Figure 6: SAC Algorithm

23 State-of-the-art Algorithms

On-Policy Algorithms:
Vanilla Policy Gradient is the most basic, entry-level algorithm in the deep RL space because it completely predates the advent of deep RL altogether. The core elements of VPG go all the way back to the late 80s / early 90s. It started a trail of research which ultimately led to stronger algorithms such as TRPO and then PPO soon after.

A key feature of this line of work is that all of these algorithms are on-policy: that is, they don't use old data, which makes them weaker on sample efficiency. But this is for a good reason: these algorithms directly optimize the objective you care about—policy performance—and it works out mathematically that you need on-policy data to calculate the updates. So, this family of algorithms trades off sample efficiency in favor of stability—but you can see the progression of techniques (from VPG to TRPO to PPO) working to make up the deficit on sample efficiency.

Off-Policy Algorithms:
DDPG is a similarly foundational algorithm to VPG, although much younger—the theory of deterministic policy gradients, which led to DDPG, wasn't published until 2014. DDPG is closely connected to Q-learning algorithms, and it concurrently learns a Q-function and a policy which are updated to improve each other.

Algorithms like DDPG and Q-Learning are off-policy, so they are able to reuse old data very efficiently. They gain this benefit by exploiting Bellman's equations for optimality, which a Q-function can be trained to satisfy using any environment interaction data (as long as there's enough experience from the high-reward areas in the environment).

But problematically, there are no guarantees that doing a good job of satisfying Bellman's equations leads to having great policy performance. Empirically one can get great performance—and when it happens, the sample efficiency is wonderful—but the absence of guarantees makes algorithms in this class potentially brittle and unstable. TD3 and SAC are descendants of DDPG which make use of a variety of insights to mitigate these issues.

24 Model-Based Algorithms

25 AlphaGo

26 AlphaZero

27 Papers

[Sutton]Ref: Reinforcement learning: An introduction [17]

[DQN]Ref: Human-level control through deep reinforcement learning[12]

[Double DQN]Ref: Deep reinforcement learning with double q-learning [18]

[Noisy Network]Ref: Noisy networks for exploration[5]

[Prioritized Replay Buffer]Ref: Prioritized experience replay [13]

[Dueling DQN]Ref: Dueling network architectures for deep reinforcement learning[19]

[Categorical DQN]Ref: A distributional perspective on reinforcement learning [4]

[RainBow]Ref: Rainbow: Combining improvements in deep reinforcement learning [8]

[DPG]Ref: Deterministic policy gradient algorithms[16]

[DDPG]Ref: Continuous control with deep reinforcement learning[10]

[D4PG]Ref: Distributed distributional deterministic policy gradients[3]

[TD3]Ref: Addressing function approximation error in actor-critic methods[6]

[A3C]Ref: Asynchronous methods for deep reinforcement learning [11]

[A2C][Code | Link]

[PPO]Ref: Proximal policy optimization algorithms[15]

[Soft Q-learning] Ref: Equivalence between policy gradients and soft q-learning [14]

[SAC]Ref: Soft Actor-Critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor [7]

[NGU] Ref: Never Give Up: Learning Directed Exploration Strategies[2]

[R2D2] Ref: Recurrent experience replay in distributed reinforcement learning[9]

[Agent57] Ref: Agent57: Outperforming the atari human benchmark[1]

References

[1] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. *arXiv preprint arXiv:2003.13350*, 2020.

[2] Adrià Puigdomènech Badia, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, Bilal Piot, Steven Kapturowski, Olivier Tieleman, Martín Arjovsky, Alexander Pritzel, Andew Bolt, et al. Never give up: Learning directed exploration strategies. *arXiv preprint arXiv:2002.06038*, 2020.

[3] Gabriel Barth-Maron, Matthew W Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva Tb, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. *arXiv preprint arXiv:1804.08617*, 2018.

[4] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887*, 2017.

[5] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[6] Scott Fujimoto, Herke Van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[7] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[8] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.

[9] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

[10] 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. *arXiv preprint arXiv:1509.02971*, 2015.

[11] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[13] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[14] John Schulman, Xi Chen, and Pieter Abbeel. Equivalence between policy gradients and soft q-learning. *arXiv preprint arXiv:1704.06440*, 2017.

[15] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[16] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.

[17] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[18] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

[19] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003, 2016.