Deep Reinforcement Learning

Key Concepts and Summary

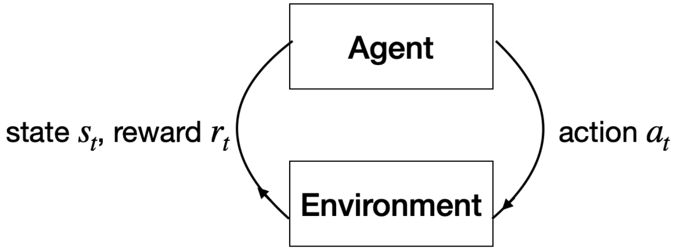
Ashkan Jasour

- Deep Reinforcement Algorithms
- Reinforcement Learning from Human Feedback (RLHF)
- Imitation Learning

- Deep Reinforcement Algorithms

- Achiam, J., & OpenAI. *Spinning Up in Deep Reinforcement Learning*. https://spinningup.openai.com/en/latest/

- **RL:** a type of machine learning where an agent learns to make decisions by interacting with an environment.
  - The agent takes an action in a given state.
  - The environment responds with a new state and a reward.
  - The agent's goal is to maximize cumulative reward over time.

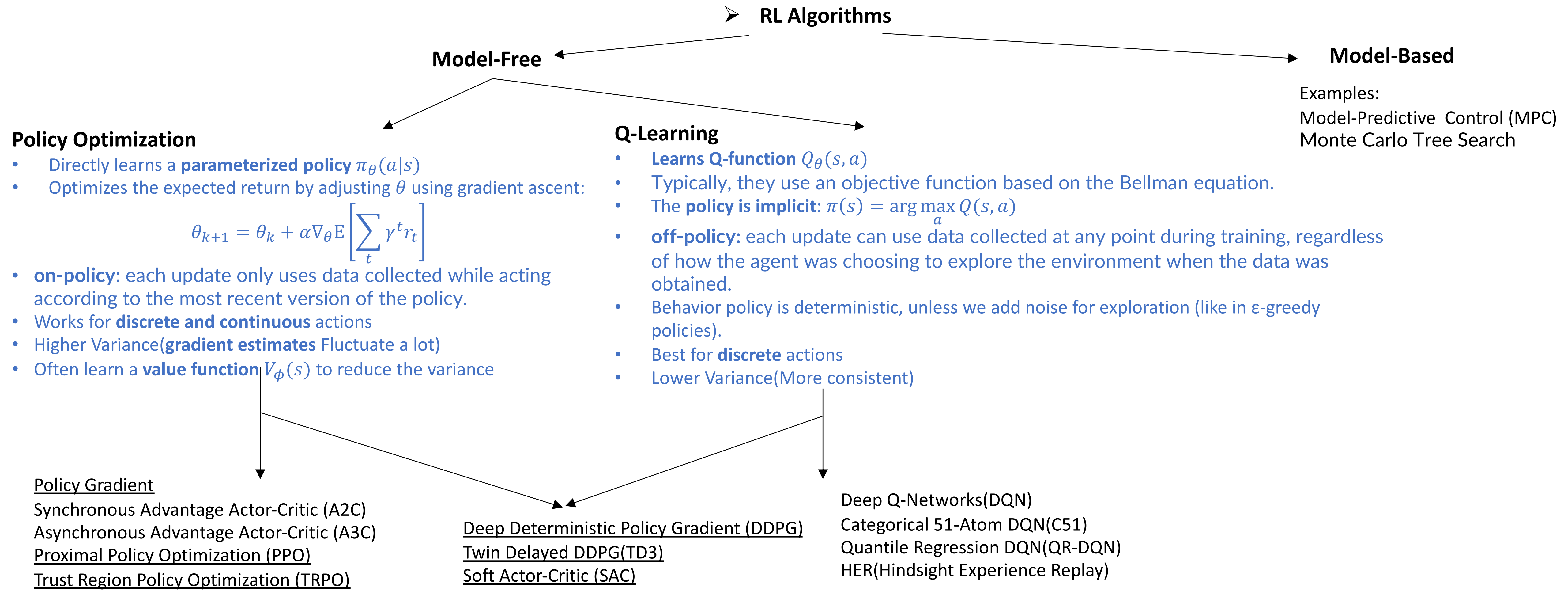


Agent	The learner or decision maker.
Environment	Everything the agent interacts with.
State (s_t)	The current situation of the agent.
Action (a_t)	The move the agent makes.
Trajectory (τ) **(episodes** or **rollouts)**	Sequence of states and actions $\tau = (s_0, a_0, s_1, a_1, ..)$, $s_0 \sim p_0(.)$ is randomly sampled from the initial state distribution **State transition**: Deterministic $s_{t+1} = f(s_t, a_t)$ or Stochastic: $s_{t+1} \sim P(.\|s_t, a_t)$ **Probability of a trajectory:** $\tau \sim \pi$, $P(\tau\|\pi) = p_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}\|s_t, a_t)\pi(a_t\|s_t)$
Reward (r_t)	Feedback from the environment after an action e.g., $r_t = R(s_t, a_t, s_{t+1})$
Policy (π)	Strategy that the agent uses to choose actions. Mapping from state s_t to action a_t. **Deterministic policy** $a_t = \mu(s_t)$, **Stochastic policy** $a_t \sim \pi(.\|s_t)$: sampled from a probability distribution over actions, given the state $s_t, \text{e.g.}, \pi(.\|s_t) = N(\mu_\theta(s_t), \Sigma_\theta(s_t))$, e.g., Softmax over discrete action logits
Value Function ($V^\pi(s)$)	Estimates how good a state (or state-action pair) is in terms of expected future rewards, e.g.,"If I'm in state s_t, how much total reward can I expect to collect in the future if I keep acting well (according to the optimal policy)?" There's also a **Q-value function** $Q^\pi(s, a)$, which estimates the expected return of taking action a in state s and then following policy π **Bellman Equation:** The value of your current state is equal to the reward you get now, plus the value of wherever you end up next (It's a recursive way of bootstrapping value estimates) $V^\pi(s_t) = \mathrm{E}_{a_t \sim \pi, s_{t+1} \sim P}[r_t + \gamma V^\pi(s_{t+1})]$, Optimal value function: $V^*(s_t) = \max_a \mathrm{E}_{s_{t+1} \sim P}[r_t + \gamma V^*(s_{t+1})]$ $Q^\pi(s_t, a_t) = \mathrm{E}_{a_{t+1} \sim \pi, s_{t+1} \sim P}[r_t + \gamma Q^\pi(s_{t+1}, a_{t+1})]$, Optimal Q function: $Q^*(s_t, a_t) = \mathrm{E}_{s_{t+1} \sim P}[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})]$

- **RL Optimization:** *Find a policy* π^* to **maximize the total discounted reward**

$$\pi^* = \arg\max_\pi \mathrm{E}\left[\underbrace{\sum_{t=0}^{\infty} \gamma^t r_t}_{R(\tau)\text{: cumulative reward over the trajectory } \tau}\right] = \int_\tau P(\tau|\pi)R(\tau)$$

  - γ: Discount factor (how much future rewards are worth)



RL Algorithms

Model-Free

Policy Optimization

- Directly learns a **parameterized policy** $\pi_\theta(a|s)$
- Optimizes the expected return by adjusting θ using gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta \mathrm{E}\left[\sum_t \gamma^t r_t\right]$$

- **on-policy**: each update only uses data collected while acting according to the most recent version of the policy.
- Works for **discrete and continuous** actions
- Higher Variance(**gradient estimates** Fluctuate a lot)
- Often learn a **value function** $V_\phi(s)$ to reduce the variance

Q-Learning

- **Learns Q-function** $Q_\theta(s,a)$
- Typically, they use an objective function based on the Bellman equation.
- The **policy is implicit**: $\pi(s) = \arg\max_a Q(s,a)$
- **off-policy:** 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.
- Behavior policy is deterministic, unless we add noise for exploration (like in ε-greedy policies).
- Best for **discrete** actions
- Lower Variance(More consistent)

Policy Gradient
Synchronous Advantage Actor-Critic (A2C)
Asynchronous Advantage Actor-Critic (A3C)
Proximal Policy Optimization (PPO)
Trust Region Policy Optimization (TRPO)

Deep Deterministic Policy Gradient (DDPG)
Twin Delayed DDPG(TD3)
Soft Actor-Critic (SAC)

Deep Q-Networks(DQN)
Categorical 51-Atom DQN(C51)
Quantile Regression DQN(QR-DQN)
HER(Hindsight Experience Replay)

Model-Based

Examples:
Model-Predictive Control (MPC)
Monte Carlo Tree Search

Policy Optimization

- We need to describe the gradient of policy performance ($\nabla_\theta J(\pi_\theta)$: **policy gradient**) with respect to policy parameters: $\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)$

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) && \text{Expand expectation} \\
&= \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) && \text{Bring gradient under integral} \\
&= \int_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) && \text{Log-derivative trick} \\
&= \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta) R(\tau)] && \text{Return to expectation form}
\end{aligned}$$

$$\boxed{\therefore\ \nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right] \quad \text{Expression for grad-log-prob}}$$

$\pi_\theta(a\|s)$	Scalar probability	∈ [0,1]
$\log \pi_\theta(a\|s)$	Scalar	$\in \mathbb{R}$
$\nabla\, \pi_\theta(a\|s)$	**Vector**	Same size as parameters θ

Collect a set of trajectories $D = \{\tau_i\}_{\{i=1,\ldots,N\}}$ where each trajectory is obtained by letting the agent act in the environment using the policy π_θ, the policy gradient can be estimated with

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau),$$

where |D| is the number of trajectories.

For each trajectory, i) loop over t = 0 to T, ii) At each time step, compute $\nabla_\theta log\pi_\theta(a_t \mid s_t)$, ii) Multiply each of those by **the same** scalar $R(\tau)$.

- Discrete action space: The log likelihood for an action a can then be obtained by indexing into the vector $\log \pi_\theta(a|s) = \log\left[P_\theta(s)\right]_a$
- Log-Likelihood. The log-likelihood of a k -dimensional action a, for a diagonal Gaussian with mean $\mu = \mu_\theta(s)$ and standard deviation $\sigma = \sigma_\theta(s)$ is given by

$$\log \pi_\theta(a|s) = -\frac{1}{2}\left(\sum_{i=1}^{k}\left(\frac{(a_i-\mu_i)^2}{\sigma_i^2} + 2\log\sigma_i\right) + k\log 2\pi\right)$$

- **Reward-to-go based Policy Gradient:** Actions are only reinforced based on rewards obtained after they are taken.

(Reward-to-go): $\hat{R}_t \doteq \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})$

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})\right]$$

- **Baseline based Policy Gradient:** For any function b which only depends on state $\mathop{\mathrm{E}}_{a_t\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)] = 0$

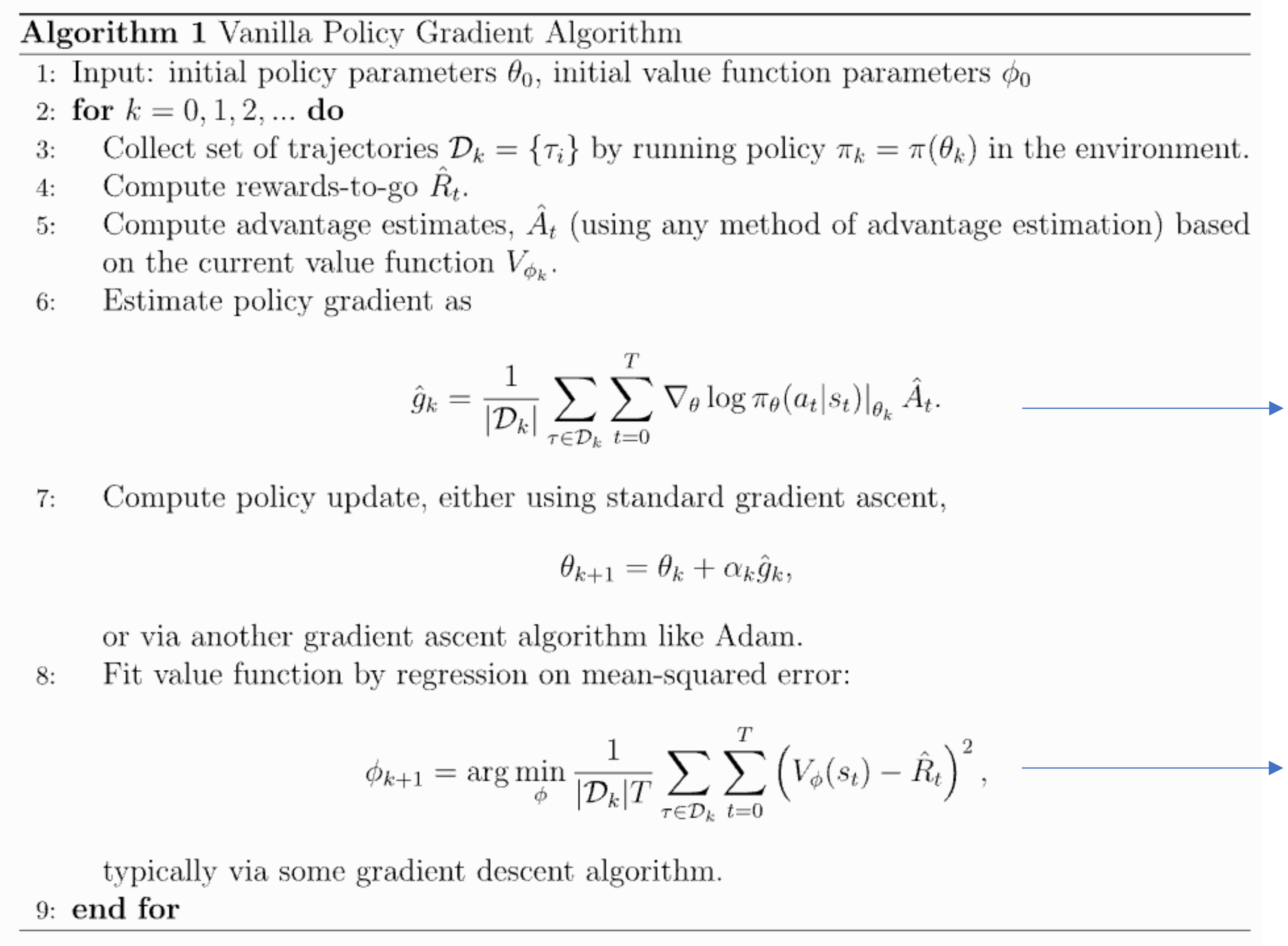
This allows us to add or subtract any number of terms to policy gradient, without changing its expectation

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)\right)\right]$$

common choice of baseline: $\boldsymbol{b(s_t) = V_\phi^\pi(s_t)}$ **(=Expected reward-to-go from state s_t)** to reduce the variance/ faster and more stable policy learning.

$$\phi_k = \arg\min_\phi \mathop{\mathrm{E}}_{s_t,\hat{R}_t\sim\pi_k}\left[\left(V_\phi(s_t) - \hat{R}_t\right)^2\right]$$

Policy Optimization: Vanilla Policy Gradient (VPG)



```
Algorithm 1 Vanilla Policy Gradient Algorithm
Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
for $k = 0, 1, 2, ...$ do
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
    on the current value function $V_{\phi_k}$.
    Estimate policy gradient as
        $$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$
    Compute policy update, either using standard gradient ascent,
        $$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$
    or via another gradient ascent algorithm like Adam.
    Fit value function by regression on mean-squared error:
        $$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$$
    typically via some gradient descent algorithm.
end for
```

- Advantage function: $A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$
- Estimated Advantage function : $\hat{A}_t = \hat{R}_t - V_{\phi_k}(s_t)$
  Hence, Vanilla Policy Gradient : **baseline-based policy gradient** method — where the **baseline** is the learned **state-value function** $V_{\phi_k}(s_t)$

- Value function estimation

Policy Gradient Methods for Reinforcement Learning with Function Approximation, Sutton et al. 2000

Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs, Schulman 2016(a)

Benchmarking Deep Reinforcement Learning for Continuous Control, Duan et al. 2016

High Dimensional Continuous Control Using Generalized Advantage Estimation, Schulman et al. 2016(b)

Policy Optimization: Trust Region Policy Optimization(TRPO)

- **Goal:** how can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so far that we accidentally cause performance collapse
- TRPO updates policies by taking the largest step possible to improve performance, while satisfying a constraint on how close the new and old policies are allowed to be (expressed in terms of KL-Divergence).
- This is different from normal policy gradient, which keeps new and old policies close in parameter space. But even seemingly small differences in parameter space can have very large differences in performance—so a single bad step can collapse the policy performance. This makes it dangerous to use large step sizes with vanilla policy gradients, thus hurting its sample efficiency. TRPO nicely avoids this kind of collapse and tends to quickly and monotonically improve perform.

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}(\theta_k, \theta) \quad \text{s.t. } \bar{D}_{KL}(\theta||\theta_k) \le \delta$$

$\mathcal{L}(\theta_k, \theta)$:surrogate advantage, a measure of how policy π_θ performs relative to the old policy π_{θ_k} using data from the old policy

$$\mathcal{L}(\theta_k, \theta) = \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a)\right]$$

$\bar{D}_{KL}(\theta||\theta_k)$ average KL-divergence between policies across states visited by the old policy:

$$\bar{D}_{KL}(\theta||\theta_k) = \underset{s\sim\pi_{\theta_k}}{\mathrm{E}}\left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)\right]$$

Taylor expand the objective and constraint

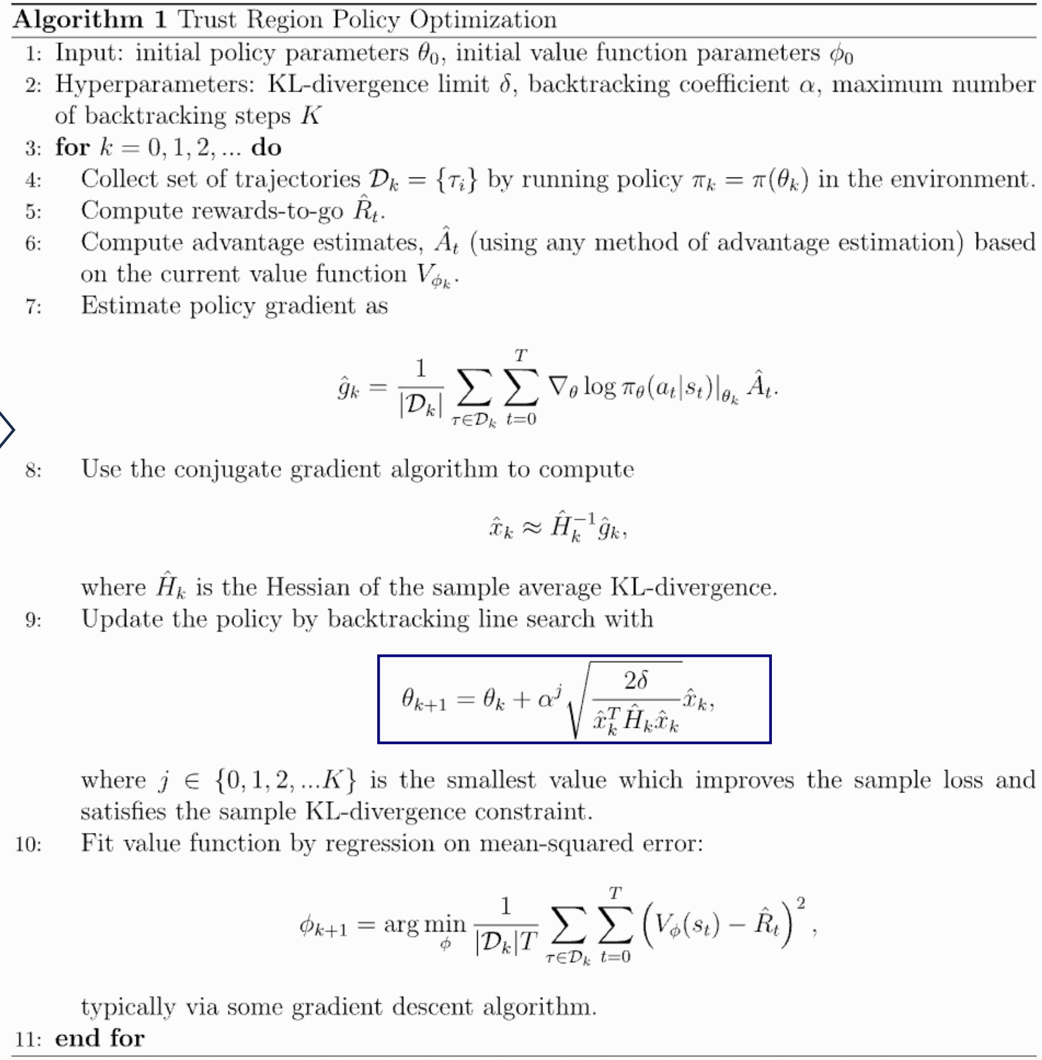
$$\theta_{k+1} = \arg\max_{\theta} g^T(\theta - \theta_k) \quad \text{s.t. } \frac{1}{2}(\theta-\theta_k)^T H(\theta - \theta_k) \le \delta.$$

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

Due to the approximation errors introduced by the Taylor expansion, this may not satisfy the KL constraint or actually improve the surrogate advantage. TRPO adds a modification to this update rule: a backtracking line search

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

where α ∈(0,1) is the backtracking coefficient, and j is the smallest nonnegative integer such that π_{θ_k} satisfies the KL constraint and produces a positive surrogate advantage.



Algorithm 1 Trust Region Policy Optimization

```
Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
Hyperparameters: KL-divergence limit $\delta$, backtracking coefficient $\alpha$, maximum number of backtracking steps $K$
for $k = 0, 1, 2, ...$ do
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
    Estimate policy gradient as
        $\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$
    Use the conjugate gradient algorithm to compute
        $\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$
    where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
    Update the policy by backtracking line search with
        $\boxed{\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,}$
    where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.
    Fit value function by regression on mean-squared error:
        $\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$
    typically via some gradient descent algorithm.
end for
```

Policy Optimization: Proximal Policy Optimization(PPO)

- **Goal:** same as TRPO
- PPO is a family of first-order methods (instead of TRPO's second-order method); Hence, PPO methods are significantly simpler to implement.

PPO-Penalty approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately.

PPO-Clip doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.

PPO-clip updates policies via

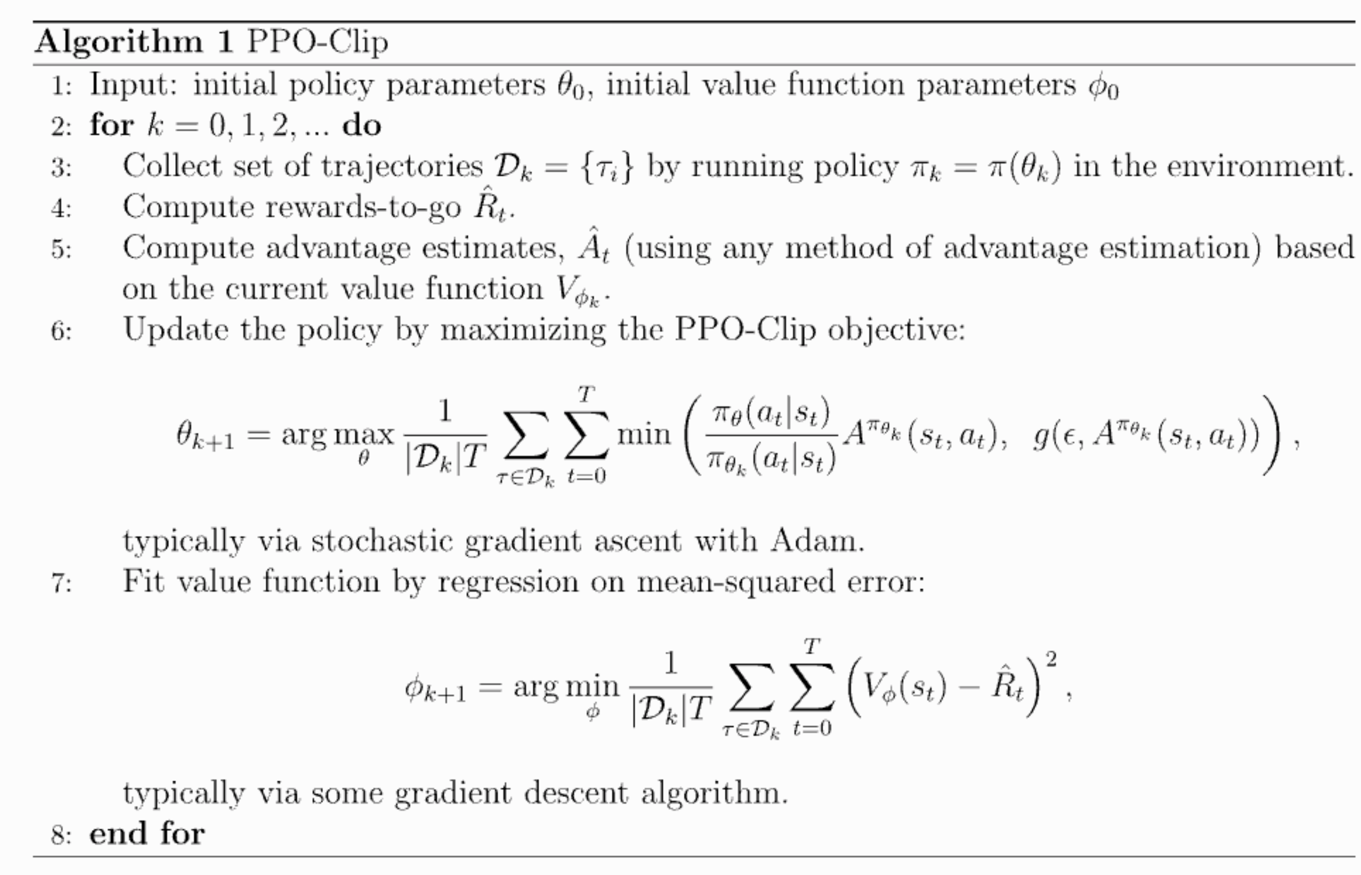
$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[L(s,a,\theta_k,\theta)\right]$$

where

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s,a))\right)$$

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

- If $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ deviates beyond $1 \pm \epsilon$, it stops the update from going further.



Algorithm 1 PPO-Clip

```
Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
for $k = 0, 1, 2, \ldots$ do
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
    Update the policy by maximizing the PPO-Clip objective:
        $\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t,a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t,a_t))\right),$
    typically via stochastic gradient ascent with Adam.
    Fit value function by regression on mean-squared error:
        $\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$
    typically via some gradient descent algorithm.
end for
```

Policy Optimization & Q-Learning : Deep Deterministic Policy Gradient (DDPG)

- Concurrently learns a Q-function and a policy. It uses off-policy data and the **Bellman equation** to learn the **Q-function**, and uses the Q-function to learn the policy.
- DDPG is an off-policy algorithm.
- For **continuous control tasks**. DDPG can be thought of as being deep Q-learning for continuous action spaces.

➢ **Components: i) Actor Network:** $\mu_\theta(s)$ deterministic policy, **ii) Critic Network:** $Q_\phi(s,a)$ estimates the value of taking action a in state s, **iii) Target Networks**: $\mu_{\theta_{targ}}(s)$ and $Q_{\phi_{targ}}(s,a)$: smoothed copies of the actor and critic for stable training

➢ **Goal:** The **critic** learns how good actions are (Q-leaning). The **actor** learns to choose actions that the **critic says are good** (Deterministic policy gradients).

➢ **Q-Learning:**

- Bellman equation for optimal Q-function: $Q^*(s_t,a_t) = \mathrm{E}_{s_{t+1}\sim P}[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1},a_{t+1})]$
- **Mean-Squared Bellman Error (MSBE)**:

Collected a set D of transitions (s_t,a_t,r_t,s_{t+1},d) (where $d \in \{0,1\}$ indicates whether state s_{t+1} is terminal)

$$\mathrm{L} = \mathrm{E}_{(s_t,a_t,r_t,s_{t+1},d)\sim D}\left[\left(Q_\phi(s_t,a_t) - \underbrace{(r_t+\gamma(1-d)\max_{a_{t+1}} Q_\phi(s_{t+1},a_{t+1}))}_{\text{Target Network}}\right)^2\right]$$

d=1 for terminal state, we **ignore the next Q-value** — because there is no next state to plan for.

- Target Network depends on the same parameters we are trying to train: ϕ. This makes MSBE minimization unstable. Instead, we use target networks $Q_{\phi_{targ}}(s_{t+1},\mu_{\theta_{targ}}(s_{t+1}))$

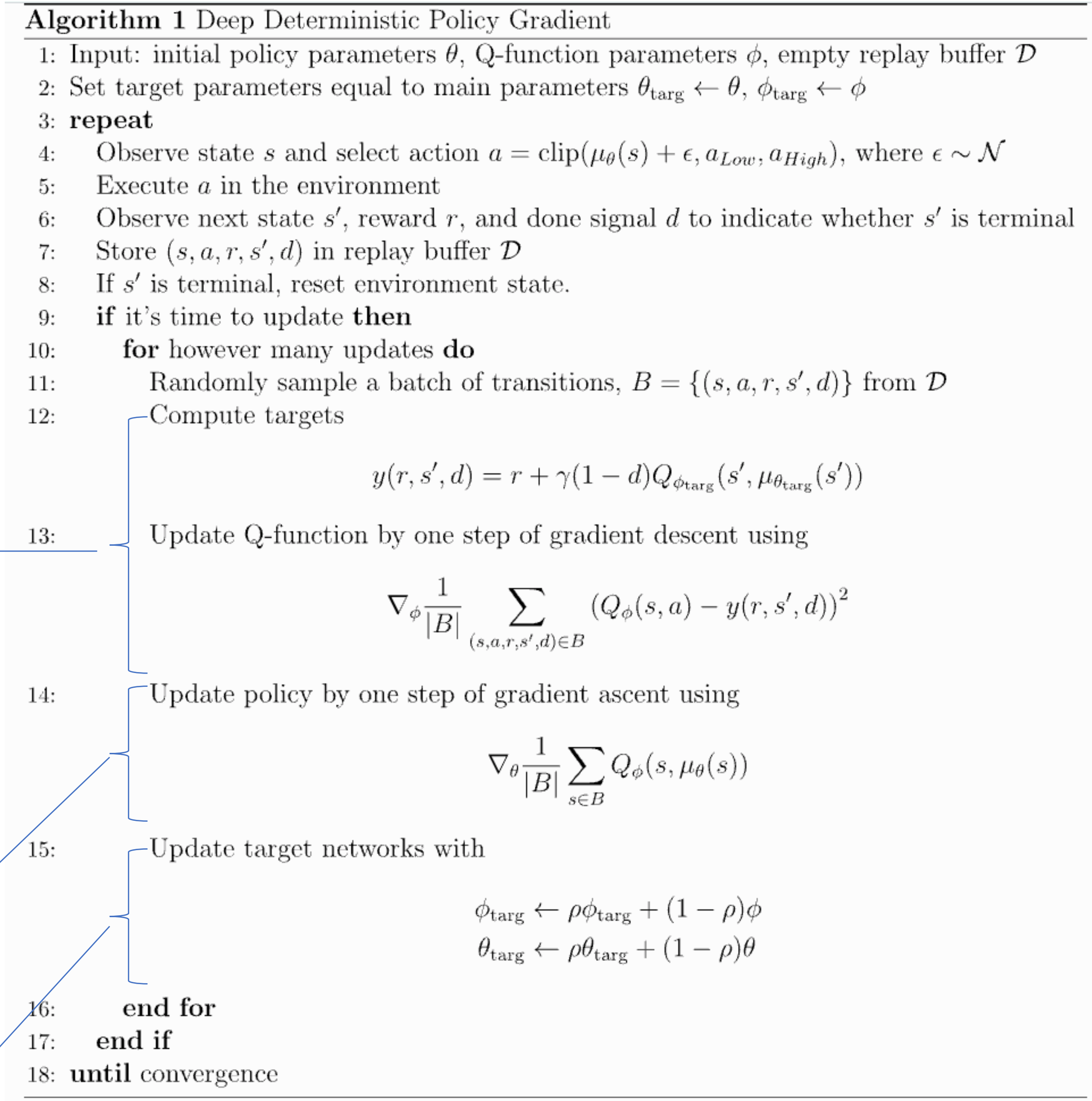
$$\mathrm{L} = \mathrm{E}_{(s_t,a_t,r_t,s_{t+1},d)\sim D}\left[\left(Q_\phi(s_t,a_t) - (r_t+\gamma(1-d)\,Q_{\phi_{targ}}(s_{t+1},\mu_{\theta_{targ}}(s_{t+1})))\right)^2\right]$$

➢ **Deterministic policy gradients:**

- Given optimal Q function: $a^*(s_t) = \arg\max_a Q^*(s_t,a_t)$
- Hence: $\max_\theta E_{s\sim D}[Q_\phi(s,\mu_\theta(s))]$

➢ **Targett Network Updates:** The target network parameters in DDPG can be viewed as a **time-delayed** (or exponentially smoothed) version of the original network parameters.

- Slowly update target networks



Algorithm 1 Deep Deterministic Policy Gradient

```
Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
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 however many updates do
      Randomly sample a batch of transitions, $B = \{(s,a,r,s',d)\}$ from $\mathcal{D}$
      Compute targets
          $y(r,s',d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s',\mu_{\theta_{\text{targ}}}(s'))$
      Update Q-function by one step of gradient descent using
          $\nabla_\phi \frac{1}{|B|}\sum_{(s,a,r,s',d)\in B}(Q_\phi(s,a)-y(r,s',d))^2$
      Update policy by one step of gradient ascent using
          $\nabla_\theta \frac{1}{|B|}\sum_{s\in B} Q_\phi(s,\mu_\theta(s))$
      Update target networks with
          $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$
          $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$
    end for
  end if
until convergence
```

Policy Optimization & Q-Learning: Twin Delayed DDPG (TD3)

- Improves the DDPG by:

- **Clipped Double-Q Learning:** TD3 learns two Q-functions (Q_{ϕ_1} and Q_{ϕ_2})instead of one (hence “twin”), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

$$\mathrm{L}_1 = \mathrm{E}_{(s_t,a_t,r_t,s_{t+1},d)\sim D}\left[\left(Q_{\phi_1}(s_t,a_t) - (r_t + \gamma(1-d)\, min_{i=1,2} Q_{\phi_{targ_i}}(s_{t+1}, a(s_{t+1})))\right)^2\right]$$

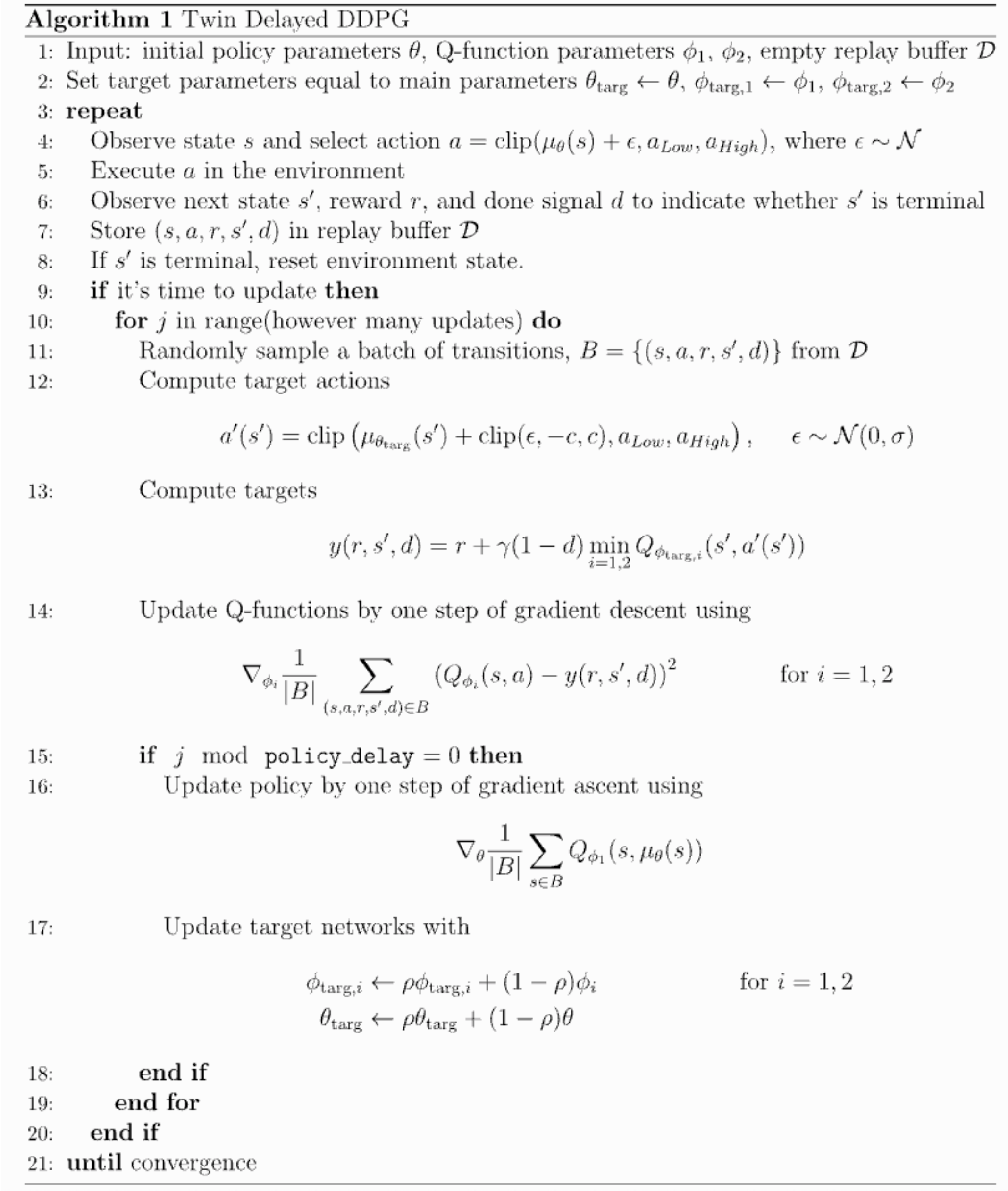
$$\mathrm{L}_2 = \mathrm{E}_{(s_t,a_t,r_t,s_{t+1},d)\sim D}\left[\left(Q_{\phi_2}(s_t,a_t) - (r_t + \gamma(1-d)\, min_{i=1,2} Q_{\phi_{targ_i}}(s_{t+1}, a(s_{t+1})))\right)^2\right]$$

- **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.

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

$$\downarrow -c \le \epsilon \le c$$

- **Delayed Policy Updates.** TD3 updates the policy (and target networks) less frequently than the Q-function, e.g., one policy update for every two Q-function updates.
  The policy is learned just by maximizing Q_{ϕ_1} : $\max_{\theta} E_{s\sim D}[Q_{\phi_1}(s, \mu_\theta(s))]$



```
Algorithm 1 Twin Delayed DDPG
Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, 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
          $a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma)$
      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
          $\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_{\phi_i}(s,a) - y(r,s',d))^2 \qquad \text{for } i = 1, 2$
      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
```

Policy Optimization & Q-Learning: Soft Actor-Critic (SAC)

- SAC is an off-policy algorithm.
- Uses a stochastic policy
- Adds an entropy bonus to the objective
- Uses two Q-functions (like TD3) for stability.

➢ **Entropy-regularized reinforcement learning:** the agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep

$$\pi^* = \arg\max_{\pi} \underset{\tau\sim\pi}{\mathrm{E}}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t,s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\Big)\right] \quad \alpha > 0 \qquad H(P) = \underset{x\sim P}{\mathrm{E}}\left[-\log P(x)\right]$$

Value function: $$V^{\pi}(s) = \underset{\tau\sim\pi}{\mathrm{E}}\left[\left.\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t,s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\Big)\right| s_0 = s\right] \Rightarrow V^{\pi}(s) = \underset{a\sim\pi}{\mathrm{E}}\left[Q^{\pi}(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right)$$

Q-function: $$Q^{\pi}(s,a) = \underset{\tau\sim\pi}{\mathrm{E}}\left[\left.\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^t H\left(\pi(\cdot|s_t)\right)\right| s_0 = s, a_0 = a\right]$$

Bellman Equation: $$\begin{aligned} Q^{\pi}(s,a) &= \underset{\substack{s'\sim P\\ a'\sim\pi}}{\mathrm{E}}\left[R(s,a,s') + \gamma\left(Q^{\pi}(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)\right] \Rightarrow \underset{\substack{s'\sim P\\ a'\sim\pi}}{\mathrm{E}}\left[R(s,a,s') + \gamma\left(Q^{\pi}(s',a') - \alpha\log\pi(a'|s')\right)\right] \\ &= \underset{s'\sim P}{\mathrm{E}}\left[R(s,a,s') + \gamma V^{\pi}(s')\right]. \end{aligned}$$

➢ **Q-Learning:** The Q-functions (Q_{ϕ_1}, Q_{ϕ_2}) are learned in a similar way to TD3

- Like in TD3, both Q-functions are learned with MSBE minimization, by regressing to a single shared target.
- Like in TD3, the shared target is computed using target Q-networks, and the target Q-networks are obtained by polyak averaging the Q-network parameters over the course of training.
- Like in TD3, the shared target makes use of the clipped double-Q trick.
- Unlike in TD3, the target also includes a term that comes from SAC's use of entropy regularization.
- Unlike in TD3, the next-state actions used in the target come from the **current policy** instead of a target policy.
- Unlike in TD3, there is no explicit target policy smoothing. TD3 trains a deterministic policy, and so it accomplishes smoothing by adding random noise to the next-state actions. SAC trains a stochastic policy, and so the noise from that stochasticity is sufficient to get a similar effect.

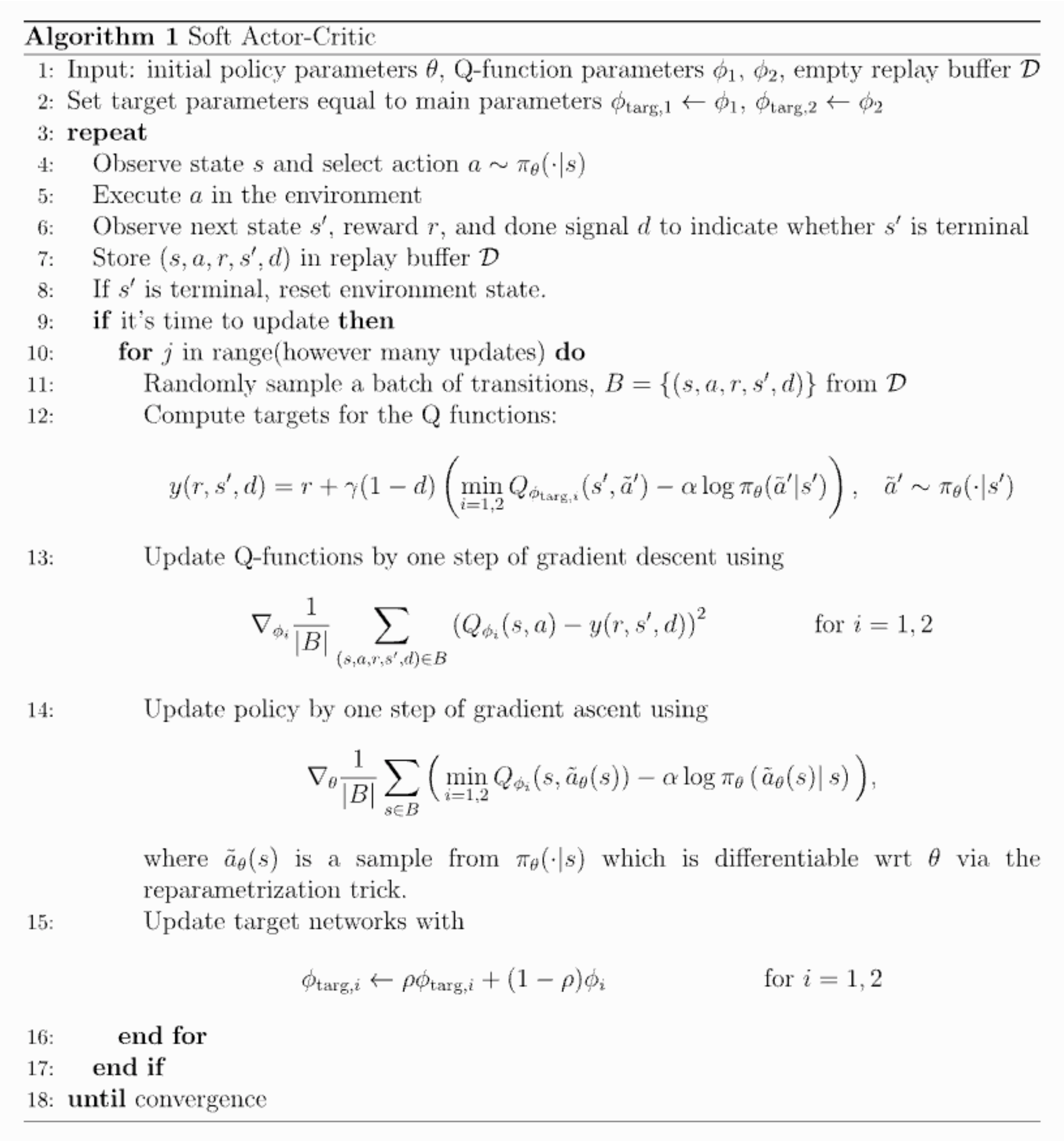
➢ **Policy Learning:** The policy should, in each state, act to maximize the expected future return plus expected future entropy. That is, it should maximize

$$\begin{aligned} V^{\pi}(s) &= \underset{a\sim\pi}{\mathrm{E}}\left[Q^{\pi}(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right) \\ &= \underset{a\sim\pi}{\mathrm{E}}\left[Q^{\pi}(s,a) - \alpha\log\pi(a|s)\right] \end{aligned}$$

Unlike in TD3, which uses Q_{ϕ_1} (just the first Q approximator), SAC uses $\min(Q_{\phi_1}, Q_{\phi_2})$. The policy is thus optimized according to

$$\max_{\theta} \underset{\substack{s\sim\mathcal{D}\\ \xi\sim\mathcal{N}}}{\mathrm{E}}\left[\min_{j=1,2} Q_{\phi_j}(s,\tilde{a}_\theta(s,\xi)) - \alpha\log\pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right]$$

- Gaussian policy: $\tilde{a}_\theta(s,\xi) = \tanh\left(\mu_\theta(s) + \sigma_\theta(s)\odot\xi\right), \quad \xi\sim\mathcal{N}(0,I)$



Algorithm 1 Soft Actor-Critic

```
Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, 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
          $\nabla_\theta\frac{1}{|B|}\sum_{s\in B}\left(\min_{i=1,2} Q_{\phi_i}(s,\tilde{a}_\theta(s)) - \alpha\log\pi_\theta\left(\tilde{a}_\theta(s)|s\right)\right),$
      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt $\theta$ 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
```

Method	Main Idea	Policy Update Equation	Drawbacks	Improvement in Successor(s)
REINFORCE (Basic PG)	Learn directly from sampled returns by computing Monte Carlo estimates of the policy gradient.	$\theta_{k+1} = \theta_k + \alpha\,\hat{g}$ $\hat{g} = \frac{1}{\lvert D \rvert} \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \lvert s_t) R(\tau)$	• High variance in gradient estimates • No baseline → credits all actions equally • Poor sample efficiency • On-policy only (can't reuse past data) • Full trajectory return $R(\tau)$ used at every step, causing delayed credit assignment	• Add a **baseline** (e.g. value function $V(s)$) to reduce variance → **Vanilla PG (with baseline)** • Use **reward-to-go** R_t instead of full return $R(\tau)$ to better assign credit → **Reward-to-go PG** • Add **KL constraint** or **clipping** to stabilize updates → **TRPO**, **PPO** • Use **bootstrapped critics** and off-policy sampling for efficiency → **Actor-Critic methods (DDPG, TD3, SAC)**
VPG (with baseline)	Use reward-to-go R_t and a value function baseline $V_\phi(s_t)$ to reduce variance without biasing the gradient.	Same as above, but replace $R(\tau)$ with advantage estimate: A_t = R_t - $V_\phi(s_t)$	Still on-policy; small updates can lead to slow learning.	**TRPO** introduces constrained optimization for larger but safe updates.
TRPO	Maximize a surrogate objective while **constraining KL divergence** from the old policy.	$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$	Complex implementation (2nd-order method); expensive due to Hessian.	**PPO** simplifies with first-order clipping instead of KL constraint.
PPO (Clip)	Simpler version of TRPO using **clipped surrogate objective** to restrict updates.	$\theta_{k+1} = \arg\max_{\theta} \mathop{\mathrm{E}}_{s,a \sim \pi_{\theta_k}} \left[L(s,a,\theta_k,\theta) \right]$ $L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a \lvert s)}{\pi_{\theta_k}(a \lvert s)} A^{\pi_{\theta_k}}(s,a),\ \ \mathrm{clip}\left(\frac{\pi_\theta(a \lvert s)}{\pi_{\theta_k}(a \lvert s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right)$	Still on-policy; less sample-efficient; sensitive to clip range.	Move to **off-policy** methods for better sample reuse → **DDPG/TD3/SAC.**
DDPG	Actor-critic method for **continuous actions** using deterministic policies.	Critic: Mean-Squared Bellman Error (MSBE) loss with respect to the target network described in terms of target Q function and targe policy Actor: Q-function maximization	Overestimation bias in Q-learning; unstable training.	**TD3** fixes with clipped double Q-learning and delayed policy update.
TD3	Improves DDPG with: 1) **2 critics**, 2) **target smoothing**, 3) **delayed updates**.	Same as DDPG, but uses minimum of 2 target Q-functions in MSBE	Still deterministic policy; no explicit exploration via policy.	**SAC** uses stochastic policy and adds entropy regularization.
SAC	Stochastic actor-critic with **entropy bonus** for exploration + stability.	Critic: like TD3 but with entropy Actor: maximizes value function with entropy	Requires tuning or learning the entropy coefficient α; slightly higher computational cost due to two Q-networks and sampling from stochastic policy	

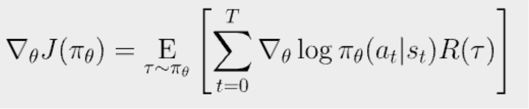
Concept	**Used In**	**Purpose**
Value Function Baseline V(s)	VPG, TRPO, PPO	Reduce gradient variance
Advantage Estimation A = R - V	All PG methods	Credit assignment
KL Constraint	TRPO	Prevent destructive updates
Clipping	PPO	Ensure stable updates
Entropy Bonus $\alpha\ log\ \pi$	SAC	Encourage exploration
Target Networks	DDPG, TD3, SAC	Stabilize critic training

➢Example:

Policy MLP Network:

```
pi_net = nn.Sequential(
                    nn.Linear(obs_dim, 64),
                    nn.Tanh(),
                    nn.Linear(64, 64),
                    nn.Tanh(),
                    nn.Linear(64, act_dim)
                    )
```

Loss:



$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right]$$

$\Rightarrow$

s_t a_t $R(\tau)$

def **compute_loss**(obs, act, weights):

    logp = get_policy(obs).log_prob(act) $\log \pi_\theta(a_t|s_t)$

    return -(logp * weights).mean()

Architecture:

1) **mlp**(sizes, activation=nn.Tanh, output_activation=nn.Identity)

Gymnasium env

2) **train**(env_name='CartPole-v0', hidden_sizes=[32], lr=1e-2, epochs=50, batch_size=5000, render=False)

- logits_net = mlp(sizes=[obs_dim]+hidden_sizes+[n_acts]) policy network
- get_policy(obs) action distribution (policy)
- get_action(obs) action sampled from policy
- compute_loss(obs, act, weights)
- **train_one_epoch**()
    - batch_loss = compute_loss(obs=torch.as_tensor(np.array(batch_obs), dtype=torch.float32),
      act=torch.as_tensor(batch_acts, dtype=torch.int32),
      weights=torch.as_tensor(batch_weights, dtype=torch.float32))
    - batch_loss.backward()
    - optimizer.step()

3) for i in range(epochs): train loop

batch_loss, batch_rets, batch_lens = **train_one_epoch()**

torch.save(logits_net.state_dict(), 'trained_policy.pt')

- Reinforcement Learning from Human Feedback (RLHF)

➢ Reinforcement Learning from Human Feedback (RLHF)

- **Reinforcement Learning (RL)** for **fine-tuning AI models**
  - Treat the **pretrained model** as a policy $\pi_\theta(a|s)$ where:

    s : input context (prompt, observation), a : model output (e.g., text, image, action), θ : model parameters

Then apply **policy gradient optimization** to improve the model via interaction or evaluation-based feedback.

Example: At each step:

- The model observes a **context** (previous tokens)
- Then **chooses the next token** from a fixed **vocabulary set**: $a_t \in \{token_1, token_2, \dots, token_v\}$
- The policy $\pi_\theta(a|s)$ is a **categorical distribution** over discrete actions.

➢ **Reinforcement Learning (RL)** for **fine-tuning AI models**

- Treat the **pretrained model** as a policy $\pi_\theta(a|s)$ where:

  s : input context (prompt, observation), a : model output (e.g., text, image, action), θ : model parameters

Then apply **policy gradient optimization** to improve the model via interaction or evaluation-based feedback.

1. Trajectory Collection

Instead of an environment simulator, we **generate samples** (e.g., model responses) from the model: D = $\{(s_i, a_i, R_i)\}$

s_i : user prompt or input, $a_i \sim \pi_\theta(.|s_i)$: model-generated output, R_i : scalar reward (from a human, a reward model, or a metric)

2. Policy Gradient Estimation

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right]$$

Since most AI fine-tuning tasks don't have multi-step episodes, reduce to **one-step trajectory**: $\nabla_\theta \mathrm{J}(\pi_\theta) = \mathrm{E}_{s,a}[\nabla_\theta \log \pi_\theta(a|s)\, R(s,a)]$

3. Add Reward-to-Go or Advantage

- If we have multi-step outputs (e.g., token-by-token generation), we can compute reward-to-go instead of reward: $\hat{R}_t \doteq \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})$
- We can use **advantage estimation** with a baseline $b(s)$:

$$\nabla_\theta \mathrm{J}(\pi_\theta) = \mathrm{E}_{s,a}[\nabla_\theta \log \pi_\theta(a|s)\, (R(s,a) - b(s))]$$

In RLHF, $b(s)$ is typically learned from value function $V_\phi(s) \approx \mathrm{E}_a[R(s,a)]$

4. Apply PPO or Clipping

To ensure **stable fine-tuning**, we can use **Proximal Policy Optimization (PPO)**

- Imitation Learning

➢ Imitation Learning (IL)

- Agent learns to perform tasks by **observing and mimicking expert behavior**, rather than learning purely from trial and error like in traditional reinforcement learning (RL).
- Instead of learning from rewards, the agent is trained using **demonstrations** provided by a human or an expert policy.

Feature	**Imitation Learning**	**Reinforcement Learning**
Learning Signal	Expert demonstrations (no reward needed)	Reward feedback from environment
Exploration	No exploration (just mimic expert)	Actively explores actions
Sample Efficiency	High (if expert data is good)	Often low (needs many trials)
Generalization	May overfit to expert data	Can generalize if trained properly

Common Imitation Learning Methods:

1.Behavior Cloning (BC) :

Supervised learning from state-action pairs (like training a classifier).

The agent learns a policy π(s) ≈ a from demonstration data (s, a).

Simple but suffers from **compounding errors** (distribution shift).

2. Inverse Reinforcement Learning (IRL) :

Learns the **reward function** that the expert is assumed to optimize.

Then uses RL to derive the policy under this reward.

3. DAgger (Dataset Aggregation) :

An interactive approach: the agent tries to imitate, gets corrected by the expert, and the dataset is updated.

Reduces compounding errors better than Behavior Cloning.

1. Behavior Cloning (BC)

A **supervised learning** approach where the agent learns to imitate an expert's behavior by mapping observations to actions.

- Collect a dataset of **expert trajectories**: : $D = \{(s_1, a_1), \dots, (s_N, a_N)\}$
- Train a policy $\pi_\theta(a|s)$ to **minimize prediction error**: $\min_\theta \sum_{i=1}^{N} \|\pi_\theta(s_i) - a_i\|^2$

Essentially: "Watch and copy the expert."

Pros: 1) Simple and fast to implement. 2) Works well when expert demonstrations cover the state space well.
Cons: 1)Fails if the agent visits unseen states not in the demo, 2) Compounding errors: small mistakes lead to drifting into unknown territory.

2. DAGGER (Dataset Aggregation)

- An iterative imitation learning algorithm that **fixes the distribution shift problem** in behavior cloning by aggregating more data from the agent's own policy.
- Let the agent **explore**, but always ask the expert what they would do, and use that to **correct** the model.

1) Initialize policy with Behavior Cloning, 2) Roll out the current policy in the environment, 3) For each visited state, query the expert for the correct action, 4) Add these new (state, expert action) pairs to the training set, 5) Retrain the policy (like in BC)on the updated dataset, 6) Repeat.

Pros: 1) Reduces **compounding error** and distribution mismatch, 2) Handles unseen states by learning from them directly.
Cons: 1) Requires **online access to the expert** (e.g., human or simulator), 2) More expensive to run than pure behavior cloning.

- **Behavior Cloning**: One-shot supervised imitation.
- **DAGGER**: Iterative imitation with expert corrections during agent exploration.

➢ Reinforcement Learning from Human Feedback (RLHF) VS Imitation Learning (IL)

	Imitation Learning (IL)	**Reinforcement Learning from Human Feedback (RLHF)**
Supervision Type	Supervised learning (behavior cloning)	Reinforcement learning (reward-based)
Human Data Type	Human demonstrations (actions or outputs)	Human **preferences** or **rankings** over model outputs
Learning Signal	Imitate human actions/output exactly	Learn to **maximize rewards** from a reward model
Goal	Mimic expert behavior	Align with human preferences / values
Training Objective	Minimize prediction error on human-provided actions	Maximize expected reward via policy gradient (e.g., PPO)
Exploration	No exploration (passive learning)	Includes exploration (active learning via policy updates)
Example Use	Self-driving cars, robotics demonstrations	ChatGPT, InstructGPT, LLM alignment
Algorithm Examples	Behavior Cloning, DAGGER	RLHF pipeline (Reward model + PPO)