JOHNS HOPKINS UNIVERSITY

Reinforcement Learning II

Thanh Nguyen-Tang*

*Many of the slides borrowed from Doina Precup at MLSS'20, Rich Sutton's book, Katerina Fragkiadaki & Tom Mitchell (Fall 2018, CMU 10703); some slides are written by ChatGPT.

Drawback of Dynamic Programming

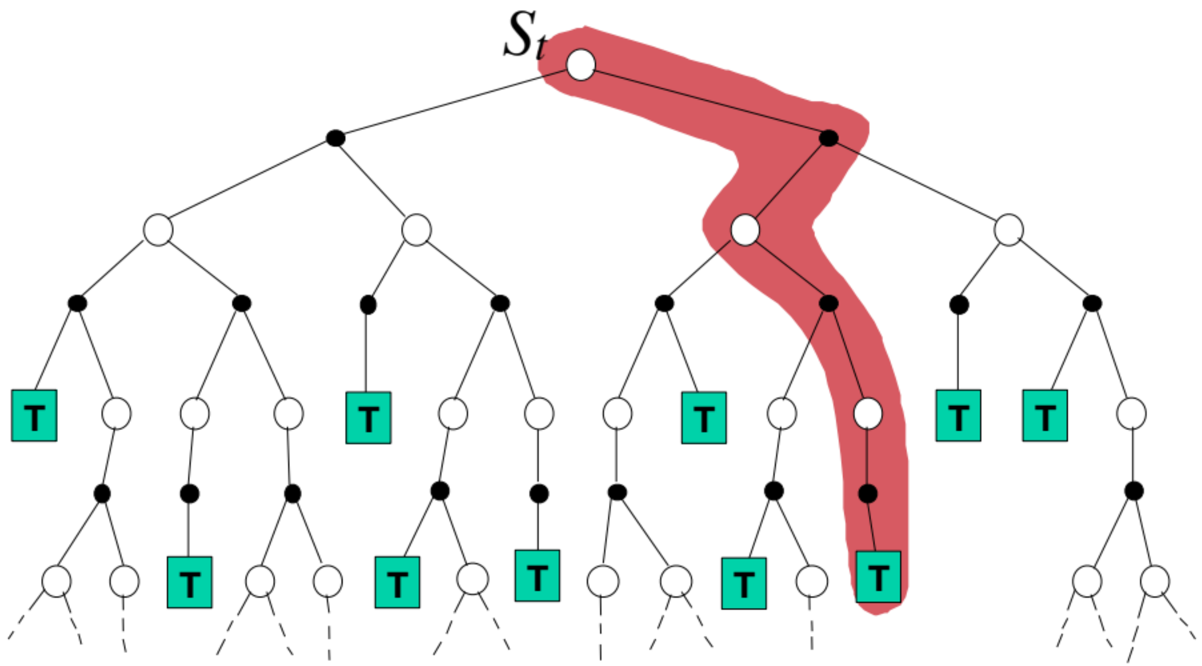
- DP require the full knowledge about the MDP

- In learning setting, we don’t know MDP and must learn from experience
  - Monte Carlo methods
  - TD (temporal-difference) learning

Simple Monte Carlo for policy evaluation

- We learn from experience without knowing the environment

$$V(s_t) \leftarrow V(s_t) + \alpha\big(G_t - V(s_t)\big)$$

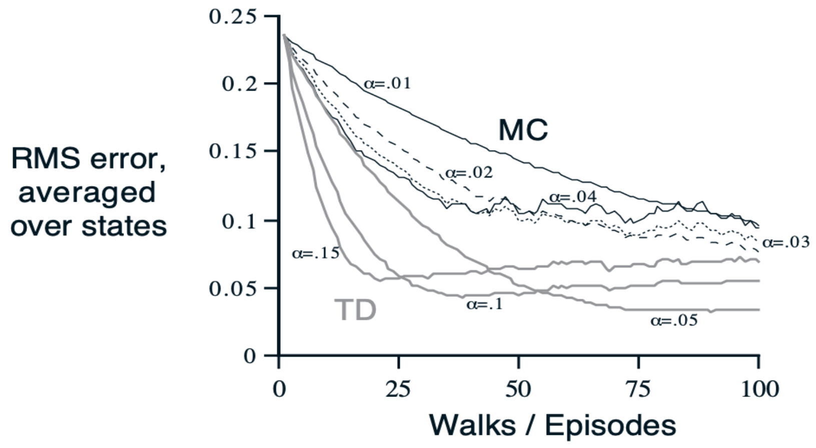


TD (temporal-difference) learning for policy evaluation

- One of the most important ideas in RL
- A bootstrapping method that does not wait until the end of an episode to make an update: Given an experience (s_t, a_t, r_t, s_{t+1})

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

- This is critical since episode can be long and continual



How about for control tasks?

- Learning for control tasks: Learn an optimal policy (only) from experience
- Learning from experience need exploration

- If the agent is taking the currently greedy action, we say it is exploiting its knowledge

- But agent also needs to try actions that are currently not optimal (exploration)

Types of exploration

- **Randomization**: add noise to the greedy policy (e.g. ϵ-greedy)

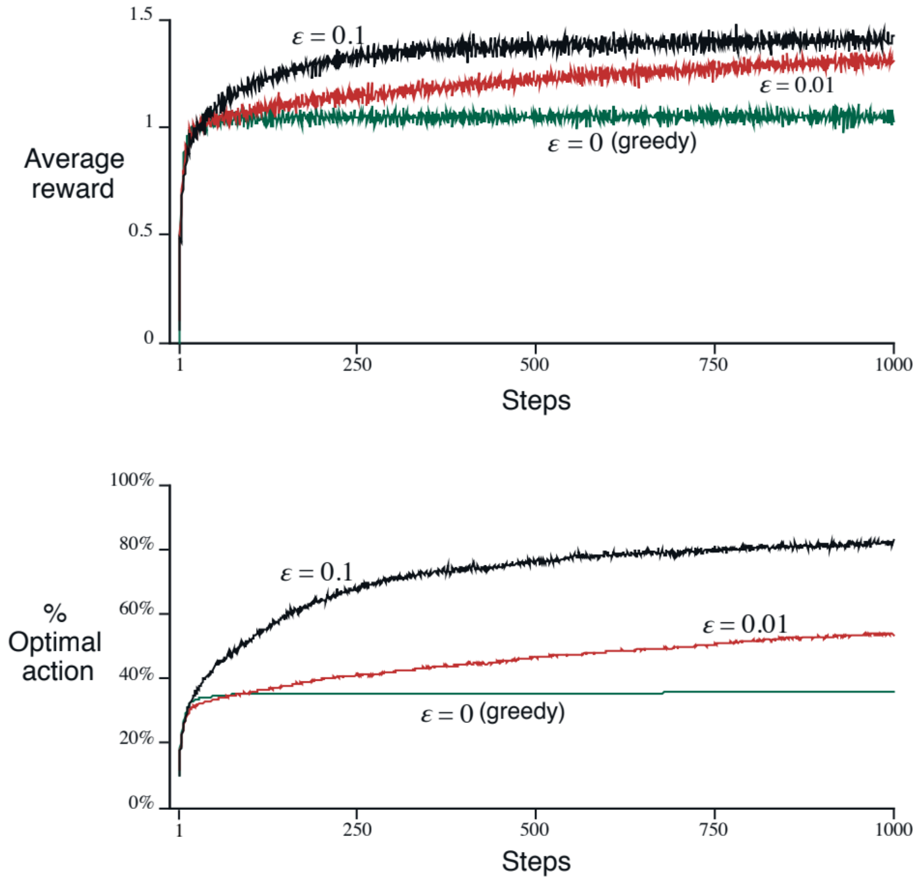
- **Optimism in face of uncertainty principle**: prefer actions with high estimated value and high uncertainty (e.g., UCB)

- **Probability matching**: select actions based according to their probability of being optimal (e.g., Thompson sampling)

ϵ-Greedy action selection

- In greedy action selection, agent always exploits its knowledge
- In ϵ-greedy action selection, agent exploits its knowledge most of the time but explore random actions with probability ϵ
- This is perhaps the simplest way to balance exploration and exploitation

Illustration of ϵ-greedy methods



SARSA: On-policy TD control

- Given some current action-value function estimate $Q(s,a)$
- Choose a from s using current Q (e.g., ϵ-greedy with respect to Q)
- Take a and observe r, s'
- Choose a' from s' using current Q (e.g., ϵ-greedy with respect to Q)
- TD update:

$$\mathrm{Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a)\ )}$$

SARSA in python (written by ChatGPT)

python Copy code

```python
import numpy as np

def sarsa(env, num_episodes, alpha, gamma, epsilon):
    q = np.zeros((env.observation_space.n, env.action_space.n))
    for i in range(num_episodes):
        s = env.reset()
        a = e_greedy(q, s, epsilon)
        while True:
            s_, r, d, _ = env.step(a)
            a_ = e_greedy(q, s_, epsilon)
            q[s,a] += alpha * (r + gamma*q[s_,a_] - q[s,a])
            s, a = s_, a_
            if d: break
    return q

def e_greedy(q, s, eps):
    if np.random.rand() < eps:
        return np.random.randint(q.shape[1])
    else:
        return np.argmax(q[s,:])
```

SARSA

- (Policy evaluation) Always learn the the action-value function of the current policy

- (Policy Improvement) Always act near-greedily wrt the current action-value function estimates

Q-learning (Watkins, 1989): Off-policy TD control

- One of the most important breakthroughs in RL
- Given some current action-value function estimate $Q(s, a)$
- Choose a from s using current Q (e.g., ϵ-greedy with respect to Q)
- Take a and observe r, s'

$$\mathrm{Q}(\mathrm{s}, \mathrm{a}) \leftarrow \mathrm{Q}(\mathrm{s}, \mathrm{a}) + \alpha\left(\mathrm{r} + \gamma \max_{\mathrm{a}'} \mathrm{Q}(\mathrm{s}', \mathrm{a}') - \mathrm{Q}(\mathrm{s}, \mathrm{a})\right)$$

Q-learning in Python (written by ChatGPT)

Sure, here's a more condensed version of the Q-learning algorithm in Python that should fit within a single page width:

python Copy code

```python
import numpy as np

def q_learning(env, num_episodes, alpha, gamma, epsilon):
    q = np.zeros((env.observation_space.n, env.action_space.n))
    for i in range(num_episodes):
        s = env.reset()
        while True:
            a = e_greedy(q, s, epsilon)
            s_, r, d, _ = env.step(a)
            q[s,a] += alpha * (r + gamma*np.max(q[s_,:]) - q[s,a])
            s = s_
            if d: break
    return q

def e_greedy(q, s, eps):
    if np.random.rand() < eps:
        return np.random.randint(q.shape[1])
    else:
        return np.argmax(q[s,:])
```

Q-learning

- In tabular case, converges to Q_*

- Theoretical convergence with function approximation require more assumptions

SARSA vs Q-learning

- **Q-learning**: Off-policy TD update

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\ )$$

- **SARSA**: On-policy TD update

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a)\ )$$

(a' from s' using current Q (e.g., ϵ-greedy with respect to Q))

SARSA vs Q-learning

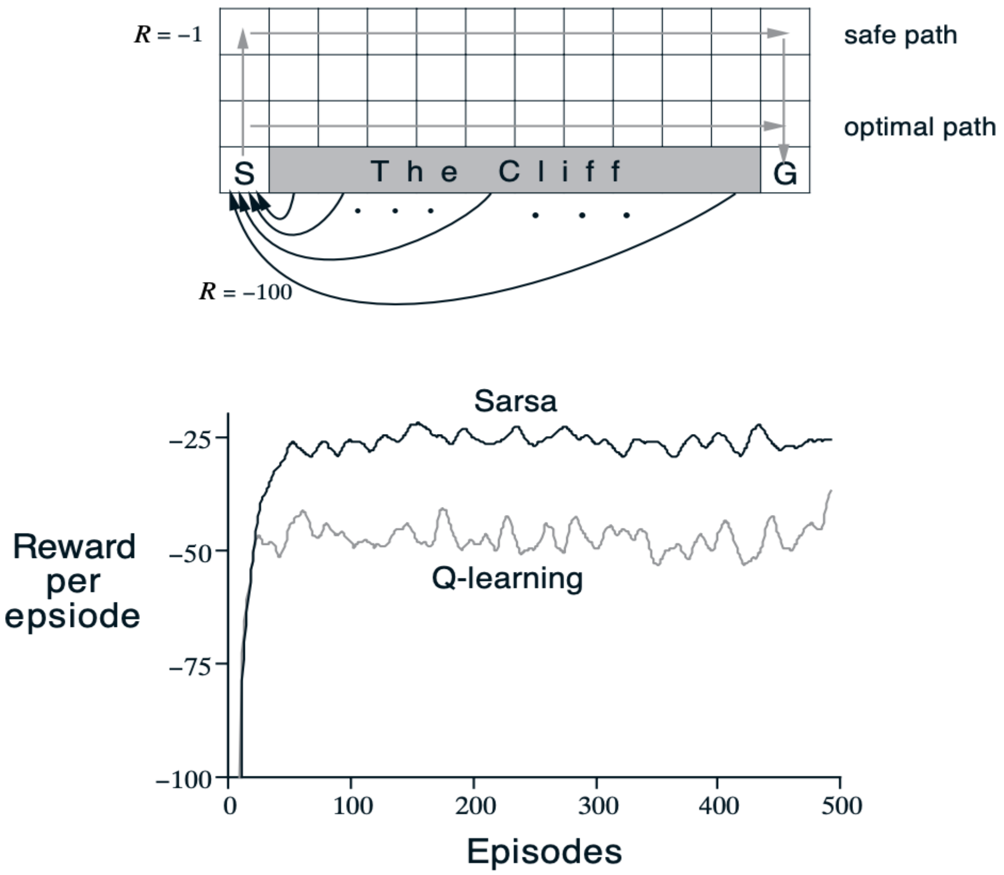


Figure 6.13: The cliff-walking task. The results are from a single run, but smoothed.

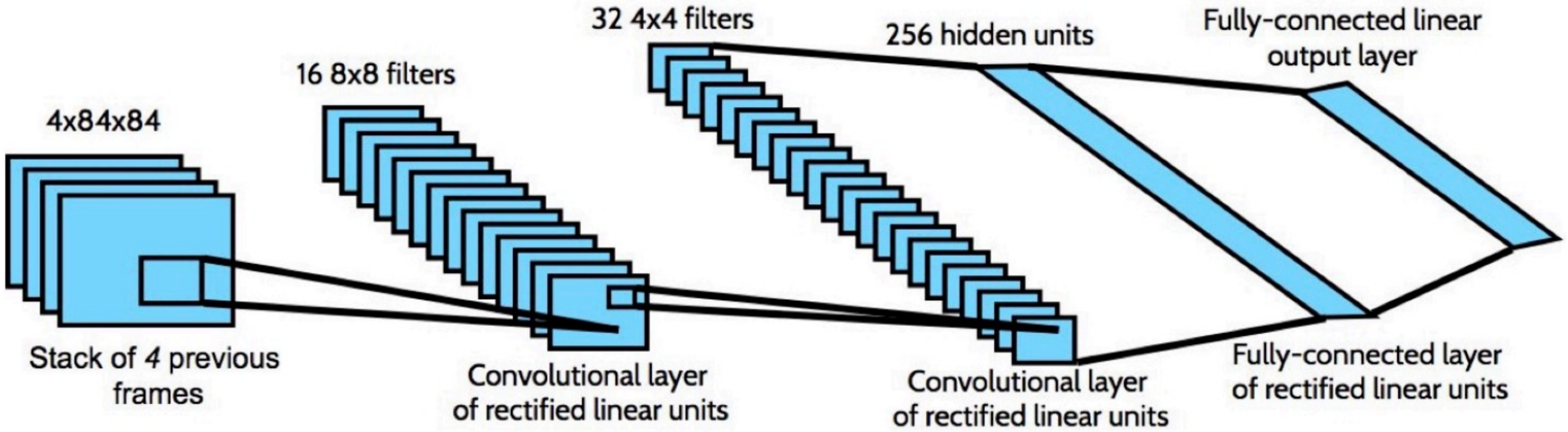
Value Iteration with function approximation

What if the state and action spaces are large?

- Approximate $\mathrm{Q}(\mathrm{s},\mathrm{a};\theta) \approx \mathrm{Q}^*(\mathrm{s},\mathrm{a})$, e.g., neural network
- For each experience (s, a, r, s'),
  - Compute semi-gradient:
    $$\Delta\theta = \left(\mathrm{Q}(\mathrm{s},\mathrm{a};\theta) - \mathrm{r} - \gamma \max_{\mathrm{a}'} \mathrm{Q}(\mathrm{s}',\mathrm{a}';\theta)\right) \nabla \mathrm{Q}(\mathrm{s},\mathrm{a};\theta)$$
  - Update gradient descent: $\theta \leftarrow \theta - \Delta\theta$
- Can extend this to large-scale problem

Deep Q-Network (DQN) (Mnih et al, Nature 2015)

- Learning to play video games simply by looking at video pixels
- Use neural network to approximate the value functions

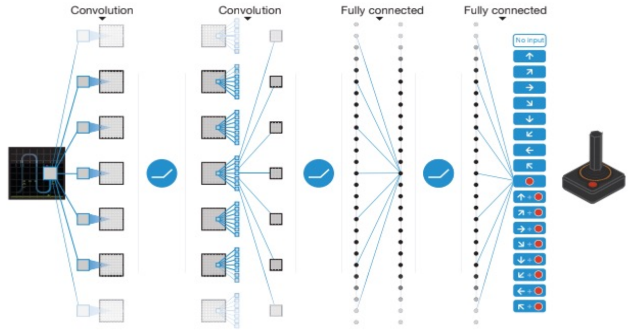


DQN applied to Classic Atari Games



- Learn to play 49 games for the Atari 2600 game console without label or human output

mapping raw screen pixels



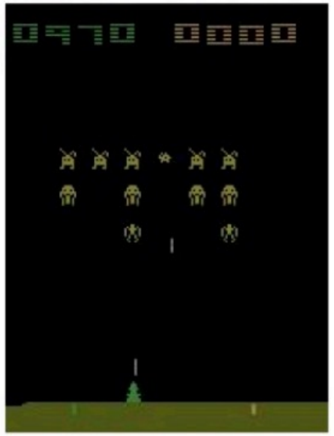
to predictions of final score for each of 18 joystick actions

- Result: Play better than all previous algorithms and at human level for more than half of the games

Core components of DQN

s

s'



- **Target network (Mnih et al., 2015):**

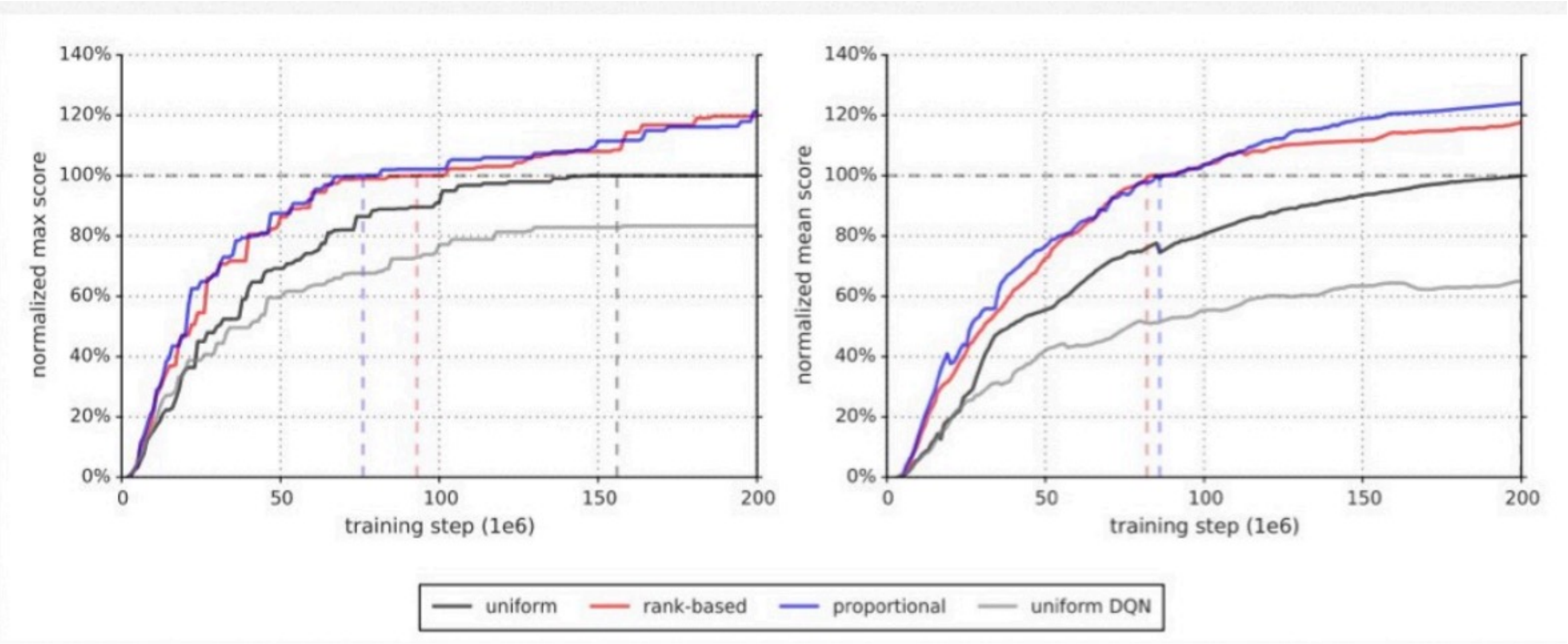
$$\Delta\theta = \left(Q(s,a;\theta) - r \;-\; \gamma \max_{a'} Q(s',a';\theta^{-})\right) \nabla Q(s,a;\theta)$$

- **Intuition**: Changing the value of one action will change the value of other actions and similar states → Network can end up chasing its own tail because of bootstrapping
- **Experience replay (Lin 1992):** replay previous experiences (s, a, r, s')

Prioritized experience replay (Schau et al. 2016)

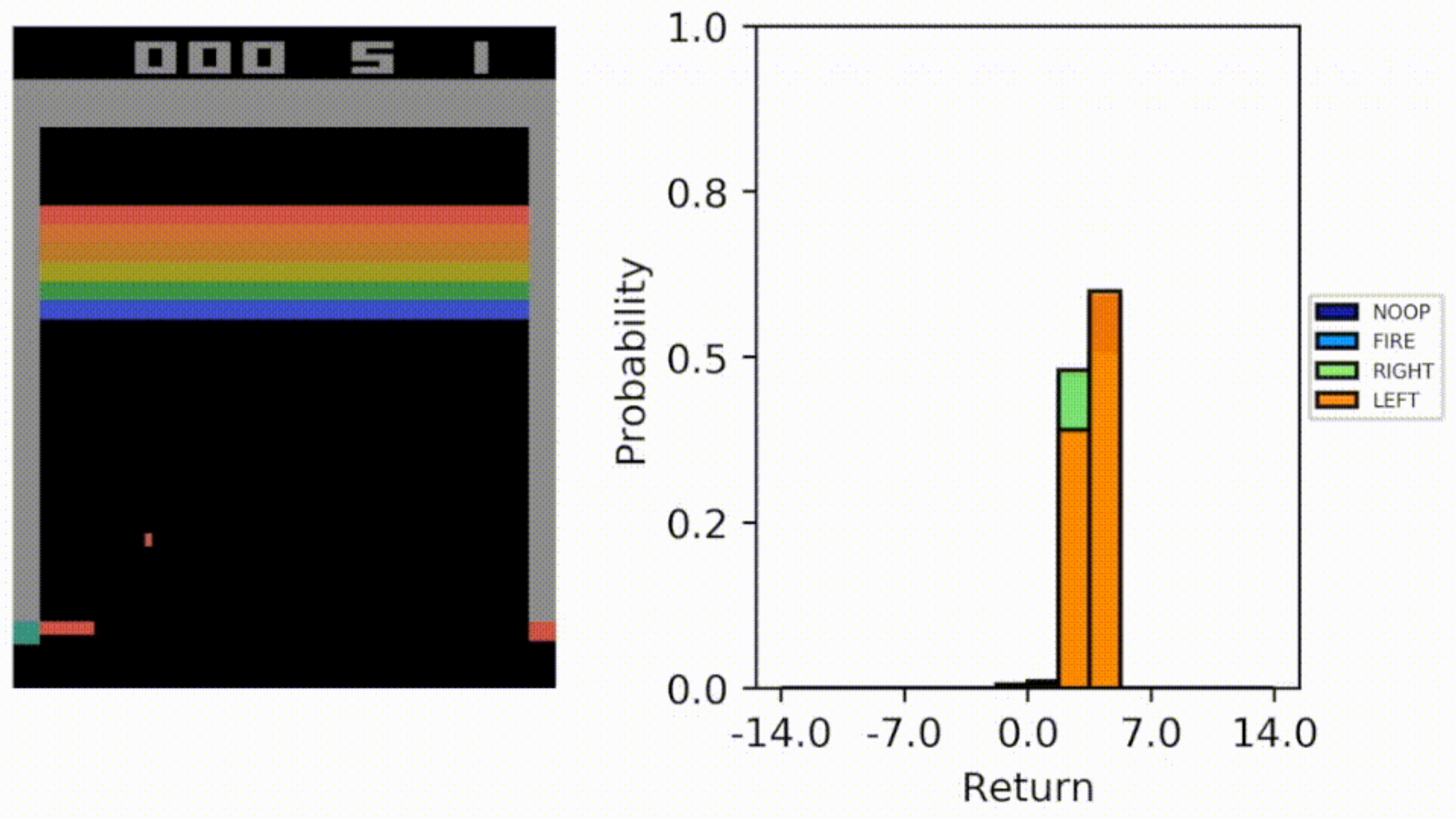
- **Idea**: Replay transitions with probability in proportion to its TD error

$$\left| Q(s, a; \theta) - r - \gamma \max_{a'} Q(s', a'; \theta^{-}) \right|$$

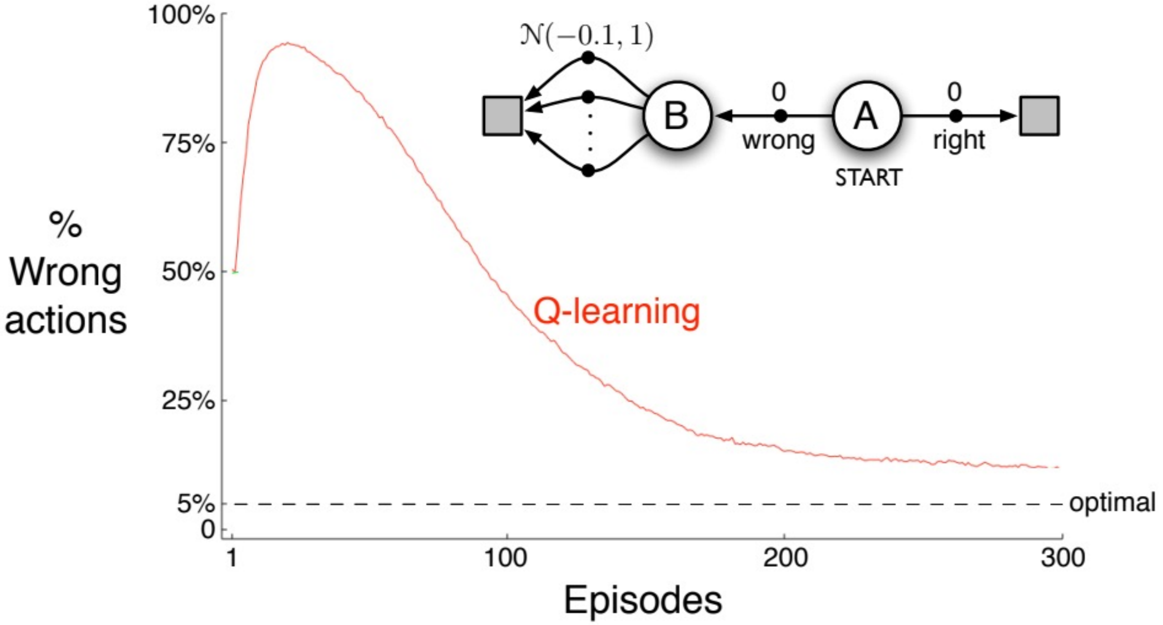


Distributional RL (Bellemare et al., ‘17)

- **Idea**: Learn the **entire distribution**, instead of the **expected value**, of the **return**



Maximization bias



Tabular Q-learning: $$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]$$

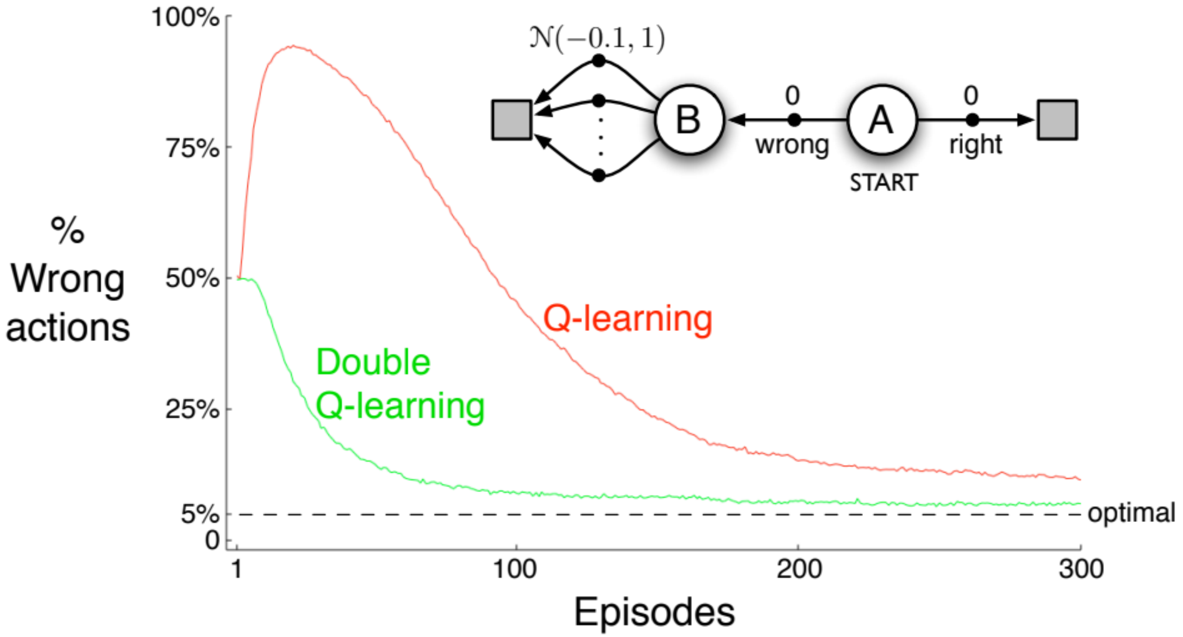
Double Q-learning (Hado van Hasselt 2010)

- Train two action-value functions Q_1 and Q_2 independently
- At each time step, randomly pick Q_1 and Q_2 and do Q-learning in it
- If updating Q_1, use Q_2 for the value of the next state and vice versus

$$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha\left(r_t + \gamma Q_2\left(s_{t+1}, \mathrm{argmax}_a Q_1(s_{t+1}, a)\right) - Q_1(s_t, a_t)\right)$$

- Action selection is ϵ-greedy wrt the sum of Q_1 and Q_2

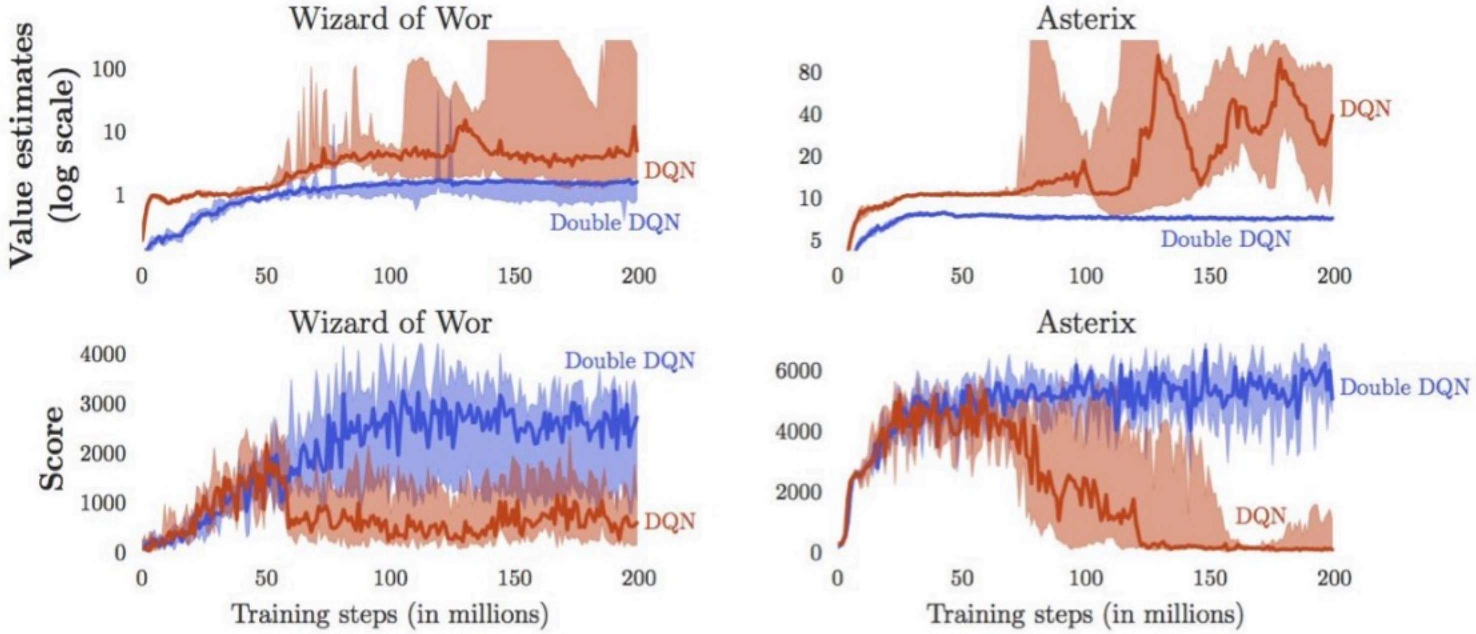
Maximization bias removed by double Q-learning



Double Q-learning:

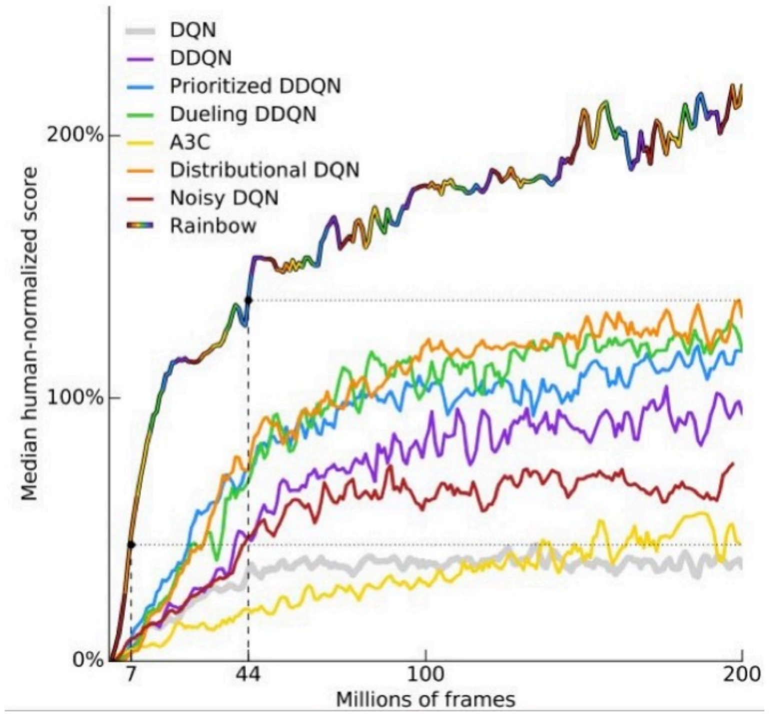
$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma Q_2\big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t) \Big]$$

Double DQN



cf. van Hasselt et al, 2015)

DQN Improvement



Rainbow model, (Hessel et al, 2017)

Policy gradient methods

Approaches to control

- Previous approach: **Action-value methods**
  - Learn the value of each action
  - Pick the max (often)
- New approach: **policy-gradient methods**
  - Learn the parameters of a stochastic policy
  - Update by gradient ascent in performance
  - Includes actor-critic methods, that learn both value and policy parameters

Why approximate policies rather than value functions?

- In many problems, the policy is <u>simpler</u> to approximate than value functions
- In many problem, the optimal policy is <u>stochastic</u>
- To enable <u>smoother</u> change in policy
- To avoid a search on every step (i.e., max over action-value functions)

General policy-gradient setup

- Directly parameterize policy $\pi_\theta(a|s)$
- Objective function:

$$J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t | \pi_\theta\right]$$

- Gradient ascent: $\theta \leftarrow \theta + \eta \nabla_\theta J(\theta)$
- How can we estimate the gradient $\nabla_\theta J(\theta)$ given we don't know the MDP?

Policy gradient theorem

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\left(\sum_{t=0}^{T} \gamma^t r_t\right)\left(\nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)\right)\right]$$

Proof: Key idea $\nabla_\theta\, \pi_\theta(a|s) = \pi_\theta(a|s)\, \nabla_\theta \log \pi_\theta(a|s)$

$$\begin{aligned}
\nabla_\theta \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t\right] &= \nabla_\theta \int \left(\sum_{t=0}^{T} \gamma^t r_t\right) p(s_{0:T}, a_{0:T})\, ds_{0:T}\, da_{0:T} && (11)\\
&= \int \left(\sum_{t=0}^{T} \gamma^t r_t\right)\left(p(s_0) \prod_{t=1}^{T} p(s_t|s_{t-1}, a_{t-1})\right)\left(\nabla_\theta \prod_{t=0}^{T} \pi_\theta(a_t|s_t)\right) ds_{0:T}\, da_{0:T} && (12)\\
&= \int \left(\sum_{t=0}^{T} \gamma^t r_t\right)\left(\nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)\right) p(s_{0:T}, a_{0:T})\, ds_{0:T}\, da_{0:T} && (13)\\
&= \mathbb{E}\left[\left(\sum_{t=0}^{T} \gamma^t r_t\right)\left(\nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)\right)\right]. && (14)
\end{aligned}$$

https://www.tuananhle.co.uk/notes/dqn-pg-a2c.html

Algorithm outline for policy gradient

- Initialize a policy network θ
- For some number of episodes
  - Sample a trajectory $(s_0, a_0, r_0, \ldots, s_T, a_T, r_T)$
  - Compute an unbiased estimate of policy gradient:

$$\widehat{\nabla_\theta J(\theta)} = \left(\sum_{t=0}^{T} \gamma^t r_t\right)\left(\nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)\right)$$

  - Perform (stochastic) gradient ascent: $\theta \leftarrow \theta + \eta\, \widehat{\nabla_\theta J(\theta)}$

Actor-Critic

- Monte-Carlo policy gradient has high variance.
- We can use a critic to estimate the action-value function:

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t | (s_0, a_0) = (s,a)\right]$$

- Actor-critic algorithm maintains two sets of parameters
  - Critic updates action-value function parameter w
  - Actor updates policy parameters θ, in a direction suggested by the critic
- Actor-critic algorithms follow an approximate policy gradient

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ Q_w(s,a)\right]$$

$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s,a)\ Q_w(s,a)$$

Reducing Variance using a Baseline

- We can subtract a baseline function $B(s)$ from the policy gradient
- This can reduce variance, without changing the expectation

$$\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) B(s)\right] = 0$$

- A good baseline is the state value function $B(s) = V^{\pi_\theta}(s)$
- We can re-write the policy gradient using the **advantage function**

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ A^{\pi_\theta}(s,a)\right]$$

Estimate the advantage function

- For the true value function $V^{\pi_\theta}(s)$, the TD error: $\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$

is an **unbiased estimate** of the advantage function

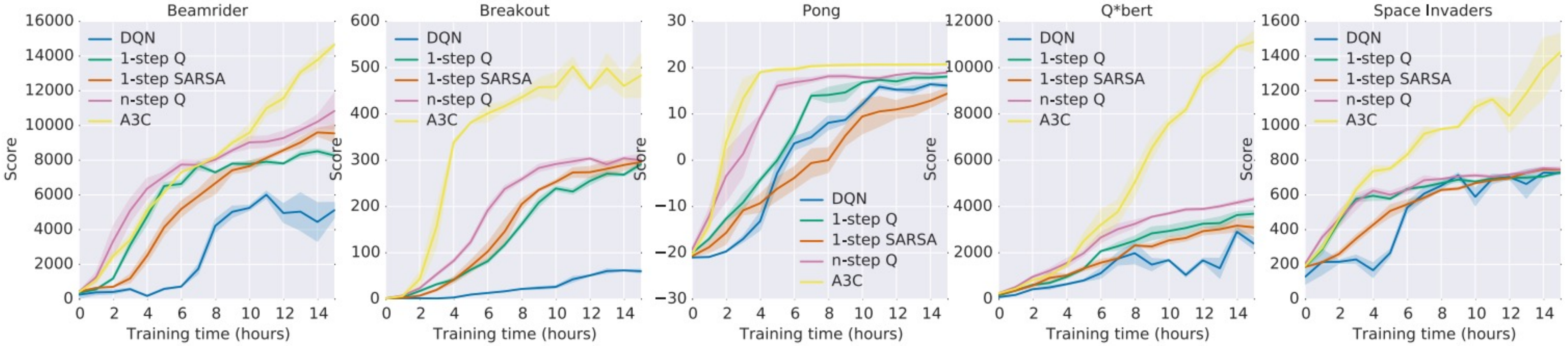
$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\delta^{\pi_\theta} | s, a\right] &= \mathbb{E}_{\pi_\theta}\left[r + \gamma V^{\pi_\theta}(s') | s, a\right] - V^{\pi_\theta}(s) \\ &= Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s, a)\end{aligned}$$

- So, we can use the TD error to compute the policy gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a)\ \delta^{\pi_\theta}\right]$$

- In practice, we can use **an approximate TD error:** $\delta_v = r + \gamma V_v(s') - V_v(s)$

(Asynchronous) Advantage Actor Critic (Mnih et al. '16)



Trust Region Policy Optimization (TRPO)

- Constrained optimization

$$\max_{\pi} L(\pi), \text{ subject to } \overline{\mathsf{KL}}[\pi_{\text{old}}, \pi] \leq \delta$$

$$\text{where } L(\pi) = \mathbb{E}_{\pi_{\text{old}}}\left[\frac{\pi(a \mid s)}{\pi_{\text{old}}(a \mid s)} A^{\pi_{\text{old}}}(s, a)\right]$$

- Construct loss from empirical data

$$\hat{L}(\pi) = \sum_{n=1}^{N} \frac{\pi(a_n \mid s_n)}{\pi_{\text{old}}(a_n \mid s_n)} \hat{A}_n$$

- Make quadratic approximation and solve with conjugate gradient algorithm

Proximal Policy Gradient (PPO)

- Use penalty instead of constraint

$$\underset{\theta}{\text{minimize}} \sum_{n=1}^{N} \frac{\pi_\theta(a_n \mid s_n)}{\pi_{\theta_{\text{old}}}(a_n \mid s_n)} \hat{A}_n - \beta \overline{\text{KL}}[\pi_{\theta_{\text{old}}}, \pi_\theta]$$

- Algorithm

for iteration$=1, 2, \ldots$ **do**
    Run policy for T timesteps or N trajectories
    Estimate advantage function at all timesteps
    Do SGD on above objective for some number of epochs
    If KL too high, increase β. If KL too low, decrease β.
end for

- Same performance as TRPO, but only first-order approximation

PPO performance

- Works very well in many non-linear problems in practice
- Actually used to fine-tune ChatGPT

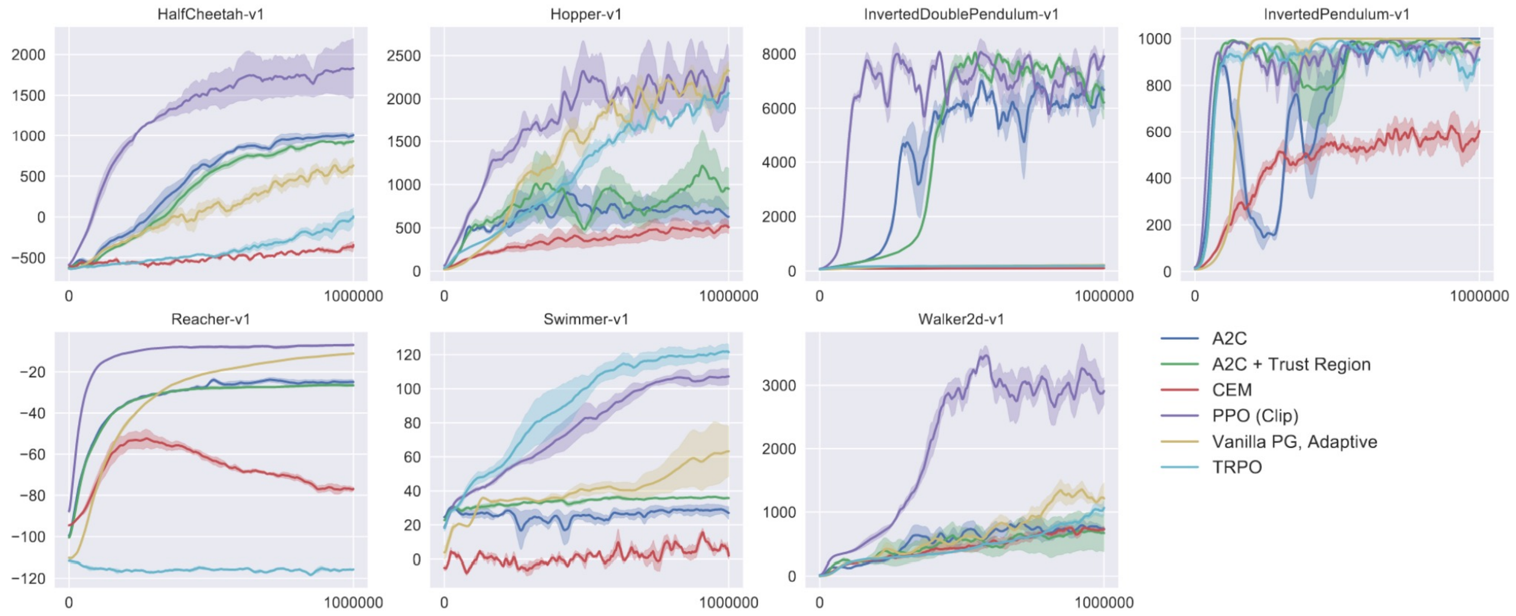


Figure: Performance comparison between PPO with clipped objective and various other deep RL methods on a slate of MuJoCo tasks. [10]