强化学习原理及应用
Reinforcement Learning (RL): Theories & Applications

DCS6289 Spring 2022

Yucong Zhang （张宇聪）

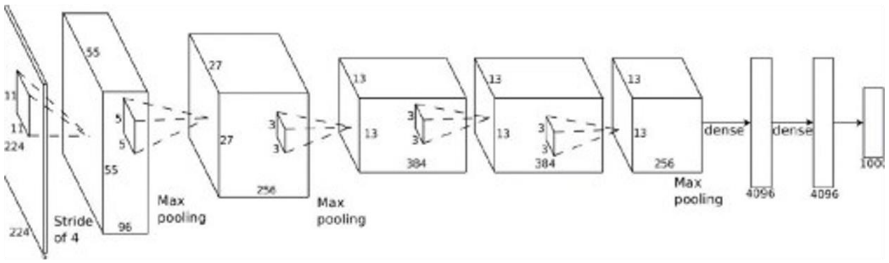
School of Computer Science and Engineering
Sun Yat-Sen University

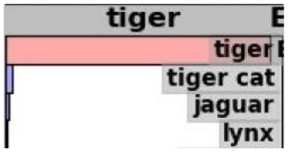
Lecture 7： Deep Reinforcement Learning

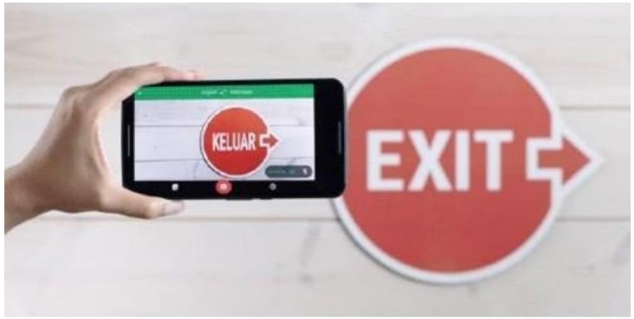
24th April. 2022

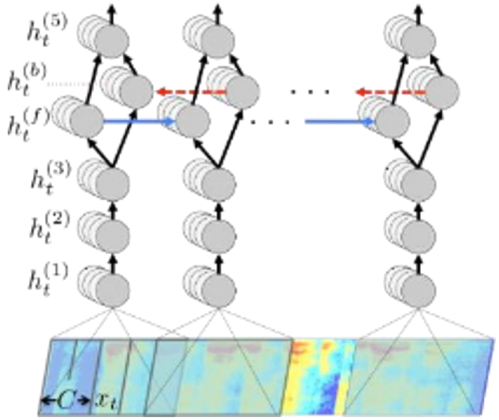
Deep Learning

- Deep Learning is part of a broader family of machine learning methods based on artificial neural networks with representation learning



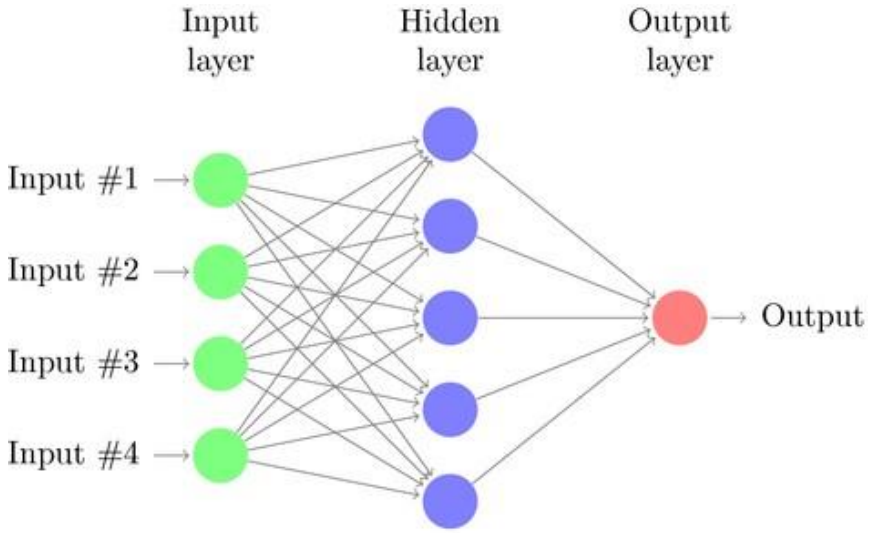






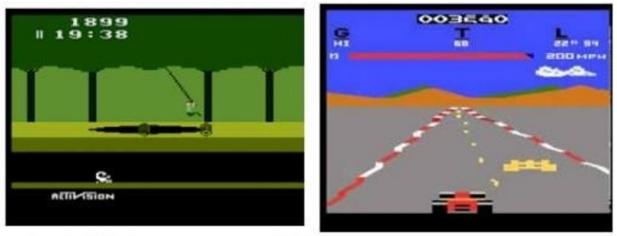
Deep Neural Networks(DNN)

- Composition of multiple functions
- Can use the chain rule to backpropagate the gradient
- Generally combines both linear and non-linear transformations
- To fit the parameters, require a loss function(MSE, log likelihood, etc.)
- Major innovation: tools to automatically compute gradients for a DNN
- Deep Learning helps us handle unstructured environments



Deep Reinforcement Learning

- *What is deep RL, and why should we care?*
  - Deep models are what allow reinforcement learning algorithms to solve complex problems
    - Deep = can process complex sensory input
    - RL = can choose complex actions
  - Use deep neural networks to represent Value, Q function, Policy, Model



Atari games:

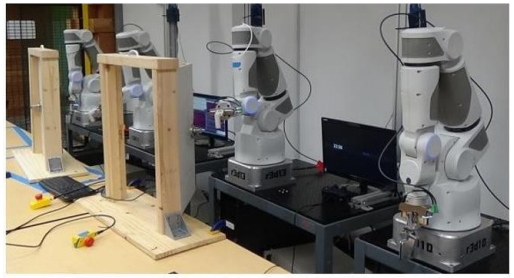
Q-learning:

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, et al. "Playing Atari with Deep Reinforcement Learning". (2013).

Policy gradients:

J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. "Trust Region Policy Optimization". (2015).

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, et al. "Asynchronous methods for deep reinforcement learning". (2016).



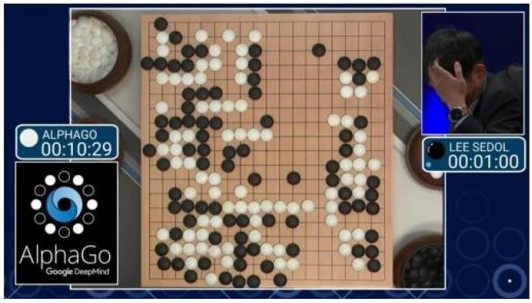
Real-world robots:

Guided policy search:

S. Levine*, C. Finn*, T. Darrell, P. Abbeel. "End-to-end training of deep visuomotor policies". (2015).

Q-learning:

D. Kalashnikov et al. "QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation". (2018).



Beating Go champions:

Supervised learning + policy gradients + value functions + Monte Carlo tree search:

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, et al. "Mastering the game of Go with deep neural networks and tree search". Nature (2016).

Deep RL with Q-Functions

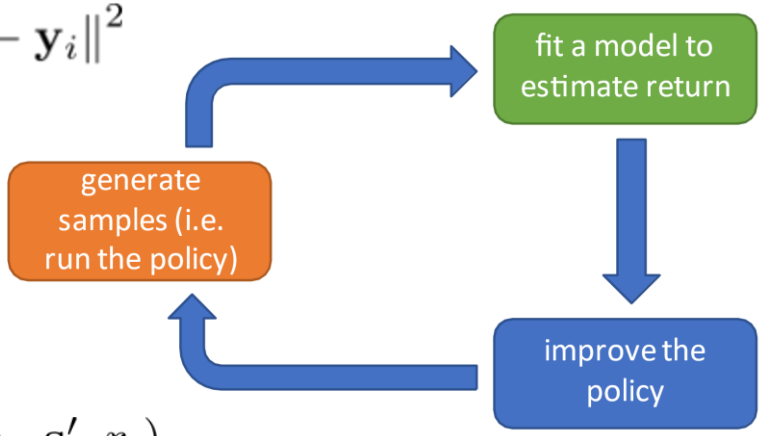
- Naïve deep Q-learning
  - Represent state-action value function by Q-network

full fitted Q-iteration algorithm:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

($K\times$)

$Q_\phi(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}', \mathbf{a}')$



$\mathbf{a} = \arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})$

online Q iteration algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

Deep RL with Q-Functions

- Two of the issues:
  - Correlations between samples - sequential states are strongly correlated
  - Non-stationary targets - target value is always changing

online Q iteration algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

these are correlated!

isn't this just gradient descent? that converges, right?

Q-learning is *not* gradient descent!

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$$

no gradient through target value

Deep RL with Q-Functions

- Solution: replay buffers

online Q iteration algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

special case with K = 1, and one gradient step

full fitted Q-iteration algorithm:

~~1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy~~

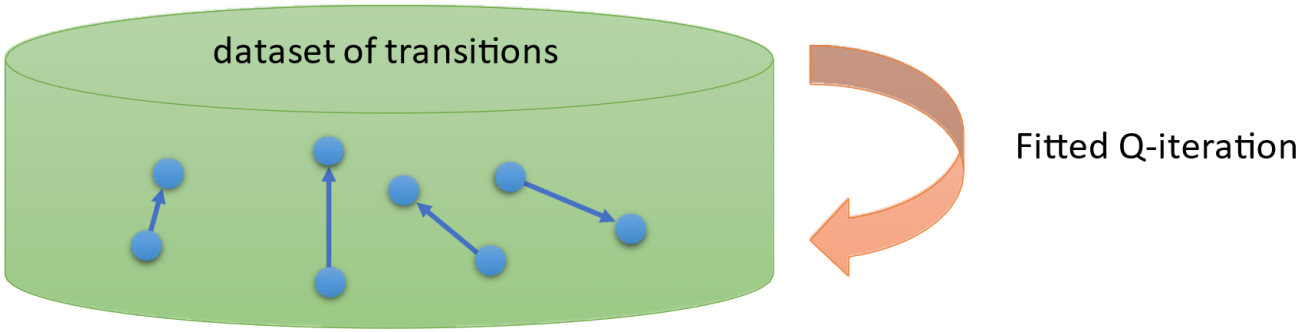
$K\times$

2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

any policy will work! (with broad support)

just load data from a buffer here

still use one gradient step



Deep RL with Q-Functions

Solution: replay buffers

full Q-learning with replay buffer:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$

$K\times$

2. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
3. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

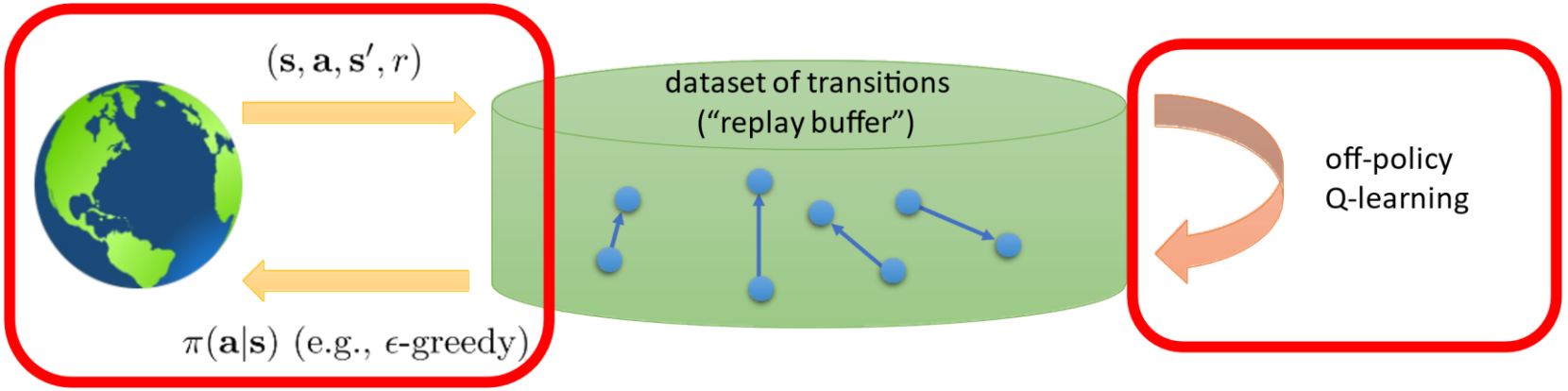
+ samples are no longer correlated

+ multiple samples in the batch (low-variance gradient)

but where does the data come from?

need to periodically feed the replay buffer…

K = 1 is common, though larger K more efficient



Deep RL with Q-Functions

- Solution: Target Networks

online Q iteration algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

~~these are correlated!~~

use replay buffer

Q-learning is *not* gradient descent!

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$$

no gradient through target value

This is still a problem!

Deep RL with Q-Functions

- Solution: Target Networks

Q-learning with replay buffer and target network:

1. save target network parameters: $\phi' \leftarrow \phi$

$N\times$

2. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$

$K\times$

3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$

4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)])$

targets don't change in inner loop!

supervised regression

Deep RL with Q-Functions

Deep Q-Network(DQN)

Q-learning with replay buffer and target network:

1. save target network parameters: $\phi' \leftarrow \phi$
2. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)])$

steps 3–4 repeated $K\times$; steps 2–4 repeated $N\times$

"classic" deep Q-learning algorithm:

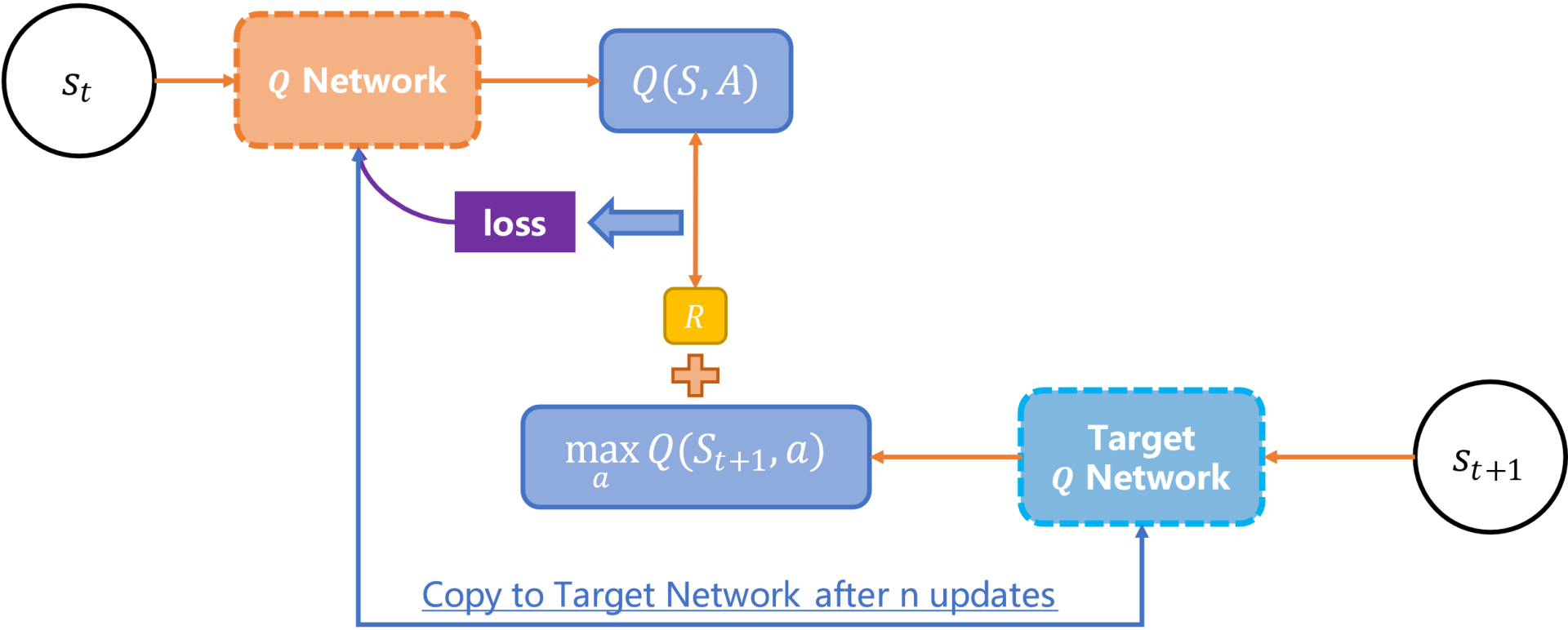
1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$

steps 2–4 repeated $K = 1$ times

5. update ϕ': copy ϕ every N steps

Deep RL with Q-Functions

- Deep Q-Network(DQN)

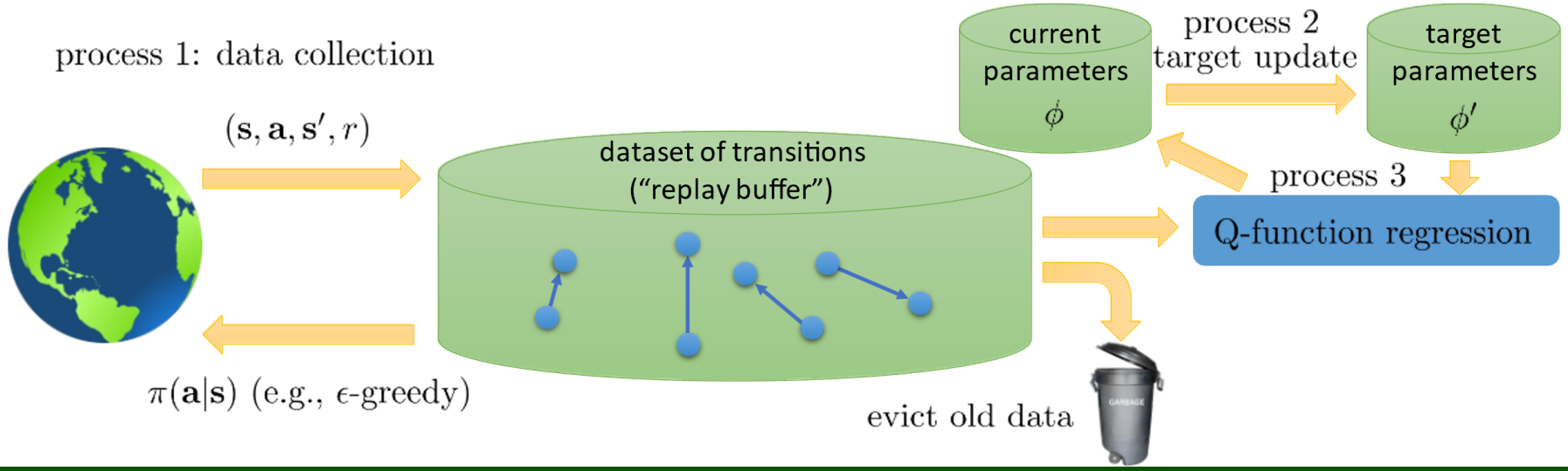


Deep RL with Q-Functions

- Deep Q-Network(DQN) Summary
  - Use experience replay and target network
  - The target network is time-delayed
  - Sample random mini-batch from replay buffer
  - Use stochastic gradient descent

Q-learning with replay buffer and target network:

1. save target network parameters: $\phi' \leftarrow \phi$
   - $N\times$ 2. collect M datapoints $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add them to $\mathcal{B}$
     - $K\times$ 3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
     - 4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)])$



Deep RL with Q-Functions

```
def main():
    env = gym.make(args.env)
    o_dim = env.observation_space.shape[0]
    a_dim = env.action_space.n
    agent = DQN(env, o_dim, args.hidden, a_dim)
    for i_episode in range(args.n_episodes):
        obs = env.reset()
        episode_reward = 0
        done = False
        while not done:
            action = agent.choose_action(obs)
            next_obs, reward, done, info = env.step(action)
            agent.store_transition(obs, action, reward, next_obs, done)
            episode_reward += reward
            obs = next_obs
            if agent.buffer.len() >= args.capacity:
                agent.learn()
```

Deep RL with Q-Functions

```
class DQN:
    def __init__(self, env, input_size, hidden_size, output_size):
        self.env = env
        self.eval_net = QNet(input_size, hidden_size, output_size)
        self.target_net = QNet(input_size, hidden_size, output_size)
        self.optim = optim.Adam(self.eval_net.parameters(), lr=args.lr)
        self.eps = args.eps
        self.buffer = ReplayBuffer(args.capacity)
        self.loss_fn = nn.MSELoss()
        self.learn_step = 0

    def choose_action(self, obs):
        if np.random.uniform() <= self.eps:
            action = np.random.randint(0, self.env.action_space.n)
        else:
            action_value = self.eval_net(obs)
            action = torch.max(action_value, dim=-1)[1].numpy()
        return int(action)

    def store_transition(self, *transition):
        self.buffer.push(*transition)
```

Deep RL with Q-Functions

```
def learn(self):
    if self.eps > args.eps_min:
        self.eps *= args.eps_decay

    if self.learn_step % args.update_target == 0:
        self.target_net.load_state_dict(self.eval_net.state_dict())
    self.learn_step += 1

    obs, actions, rewards, next_obs, dones = self.buffer.sample(args.batch_size)
    actions = torch.LongTensor(actions)  # LongTensor to use gather latter
    dones = torch.IntTensor(dones)
    rewards = torch.FloatTensor(rewards)

    q_eval = self.eval_net(obs).gather(-1, actions.unsqueeze(-1)).squeeze(-1)
    q_next = self.target_net(next_obs).detach()
    q_target = rewards + args.gamma * (1 - dones) * torch.max(q_next, dim=-1)[0]
    loss = self.loss_fn(q_eval, q_target)
    self.optim.zero_grad()
    loss.backward()
    self.optim.step()
```

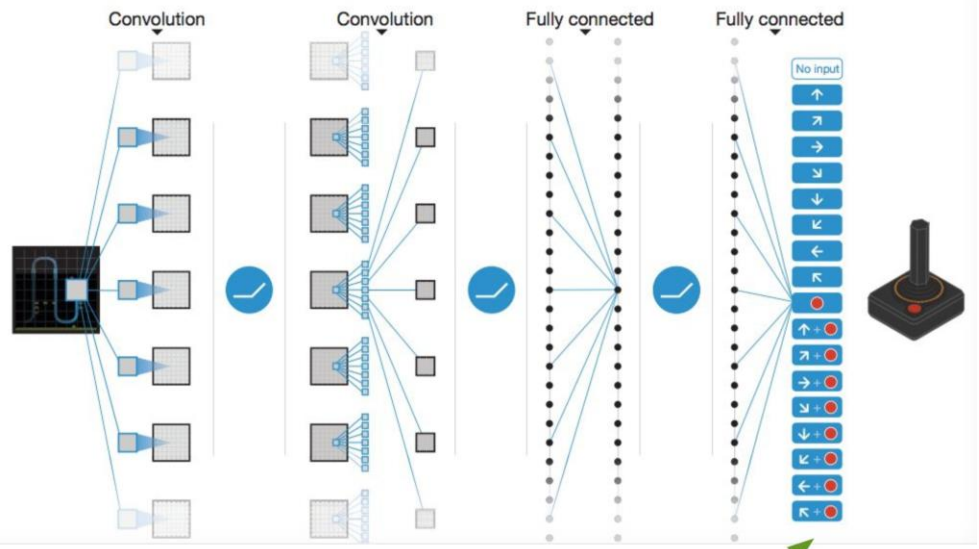
Deep RL with Q-Functions

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = []
        self.capacity = capacity

    def len(self):
        return len(self.buffer)

    def push(self, *transition):
        if len(self.buffer) == self.capacity:
            self.buffer.pop(0)
        self.buffer.append(transition)

    def sample(self, n):
        index = np.random.choice(len(self.buffer), n)
        batch = [self.buffer[i] for i in index]
        return zip(*batch)

    def clean(self):
        self.buffer.clear()
```

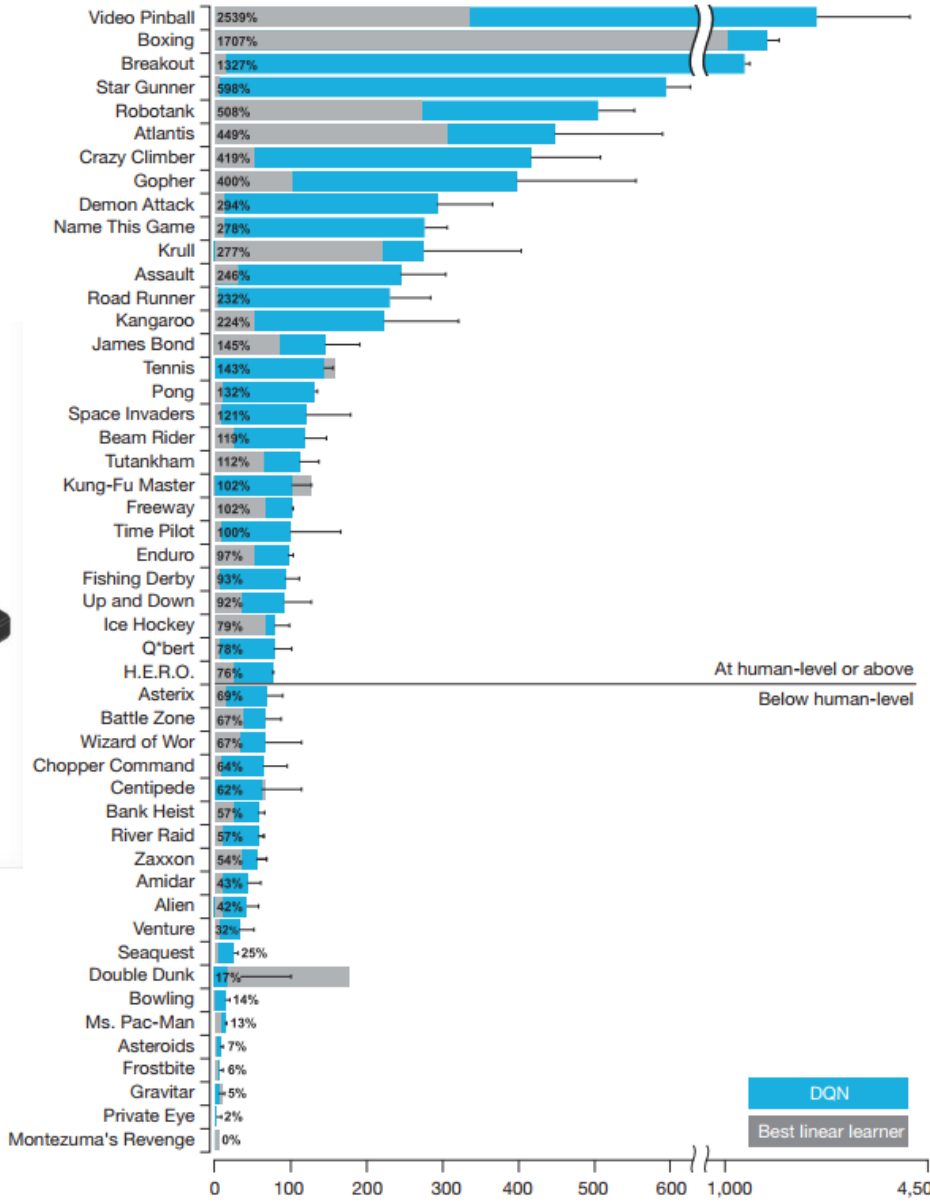
```python
class QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(QNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.Tensor(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Deep RL with Q-Functions

- Network and Performance



1 network, outputs Q value for each action



Deep RL with Q-Functions

- Variant
  - Double DQN：solving overestimation in DQN

target value $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$

this last term is the problem

imagine we have two random variables: X_1 and X_2

$$E[\max(X_1, X_2)] \geq \max(E[X_1], E[X_2])$$

$Q_{\phi'}(\mathbf{s}', \mathbf{a}')$ is not perfect – it looks "noisy"

hence $\max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}', \mathbf{a}')$ *overestimates* the next value!

idea: don't use the same network to choose the action and evaluate value!

"double" Q-learning: use two networks:

$$Q_{\phi_A}(\mathbf{s}, \mathbf{a}) \leftarrow r + \gamma Q_{\phi_B}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi_A}(\mathbf{s}', \mathbf{a}'))$$

$$Q_{\phi_B}(\mathbf{s}, \mathbf{a}) \leftarrow r + \gamma Q_{\phi_A}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi_B}(\mathbf{s}', \mathbf{a}'))$$

if the two Q's are noisy in *different* ways, there is no problem

Deep RL with Q-Functions

- Variant
  - Double DQN：solving overestimation in DQN

where to get two Q-functions?

just use the current and target networks!

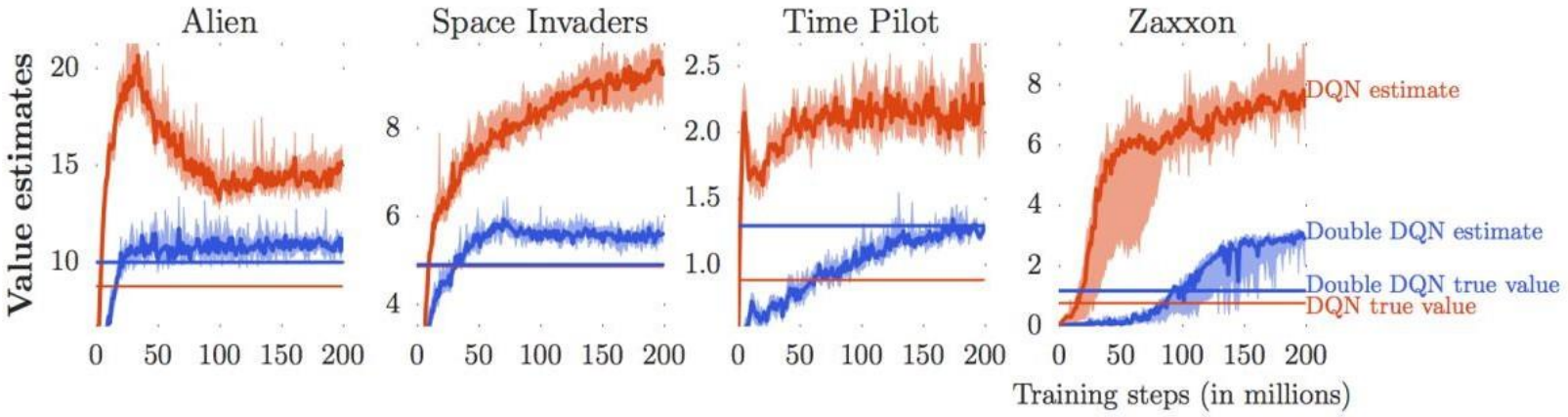
standard Q-learning: $y = r + \gamma Q_{\phi'}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}', \mathbf{a}'))$

double Q-learning: $y = r + \gamma Q_{\phi'}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi}(\mathbf{s}', \mathbf{a}'))$

just use current network (not target network) to evaluate action

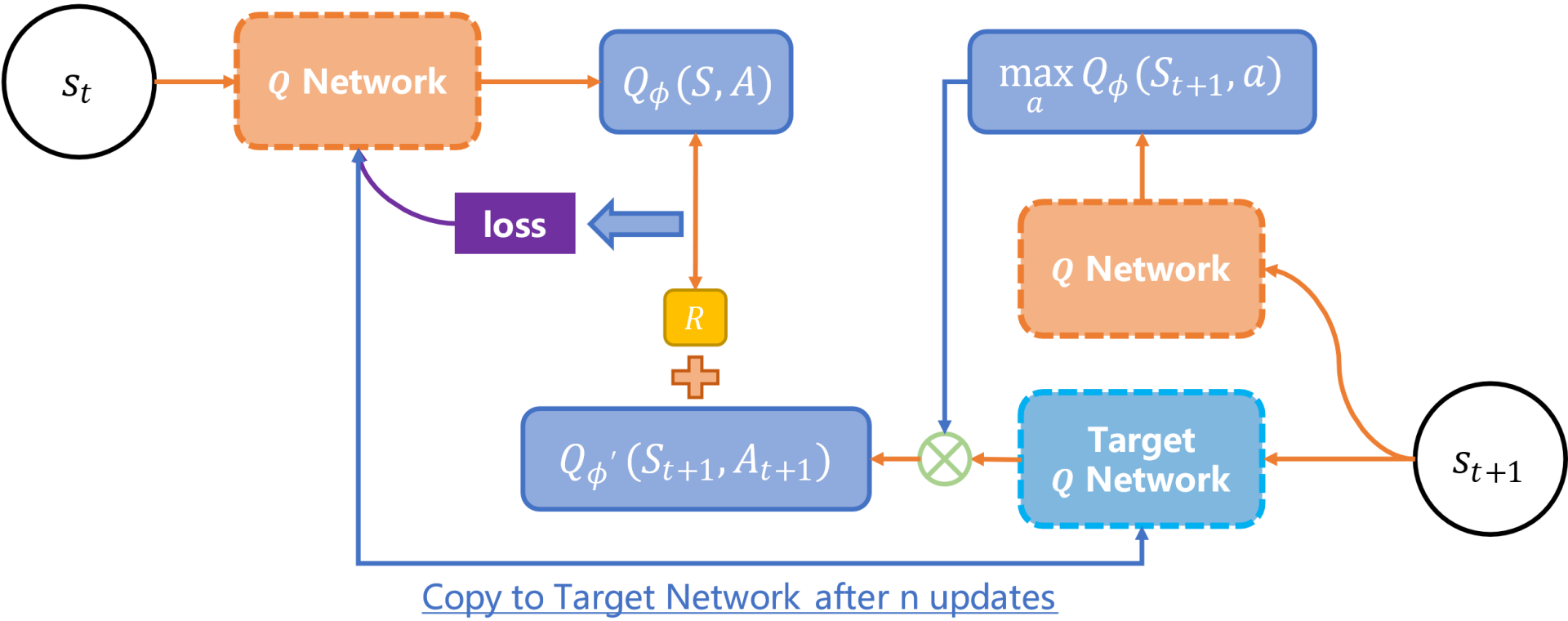
still use target network to evaluate value!

  - Value estimation in Atari



Deep RL with Q-Functions

- Variant
  - Double DQN： solving overestimation in DQN



Deep RL with Q-Functions

- Performance of Double DQN in Atari

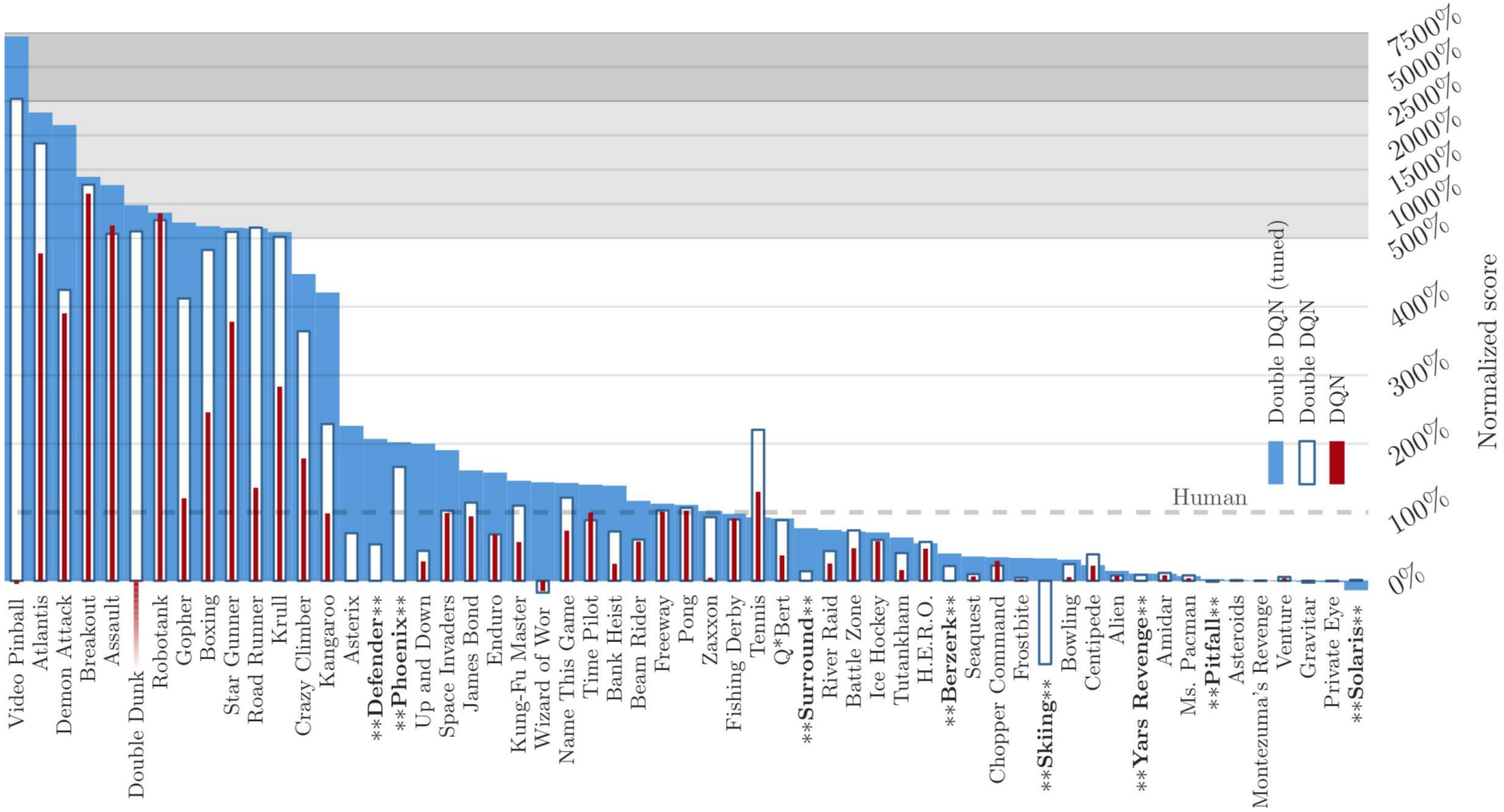


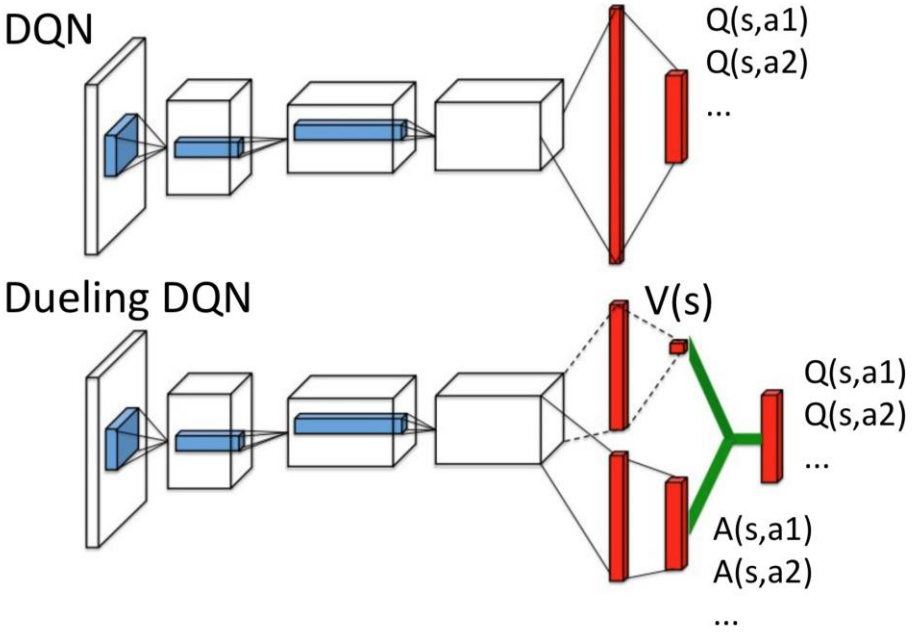
Figure: van Hasselt, Guez, Silver, 2015

Deep RL with Q-Functions

- Variant
  - Dueling DQN
    - Sometimes it is unnecessary to know the exact value of each action
    - Split the Q-values in two different parts, the value function V(s) and the advantage function A(s, a), Q(s, a) = V(s) + A(s, a)
    - Value function V(s) : how much reward we will collect from the state s
    - Advantage function A(s, a) : how much better one action is compared to the other actions.
  - Prioritized experience replay
    - Weigh the samples so that “important” ones are drawn more frequently for training
  - Rainbow
    - Combining improvements : Double DQN、Dueling DQN、Prioritized Replay Buffer、Multi-Step Learning、Distributional DQN（Categorical DQN）、NoisyNet

Deep RL with Q-Functions

- Network and performance of Dueling DQN



Wang et.al., ICML, 2016

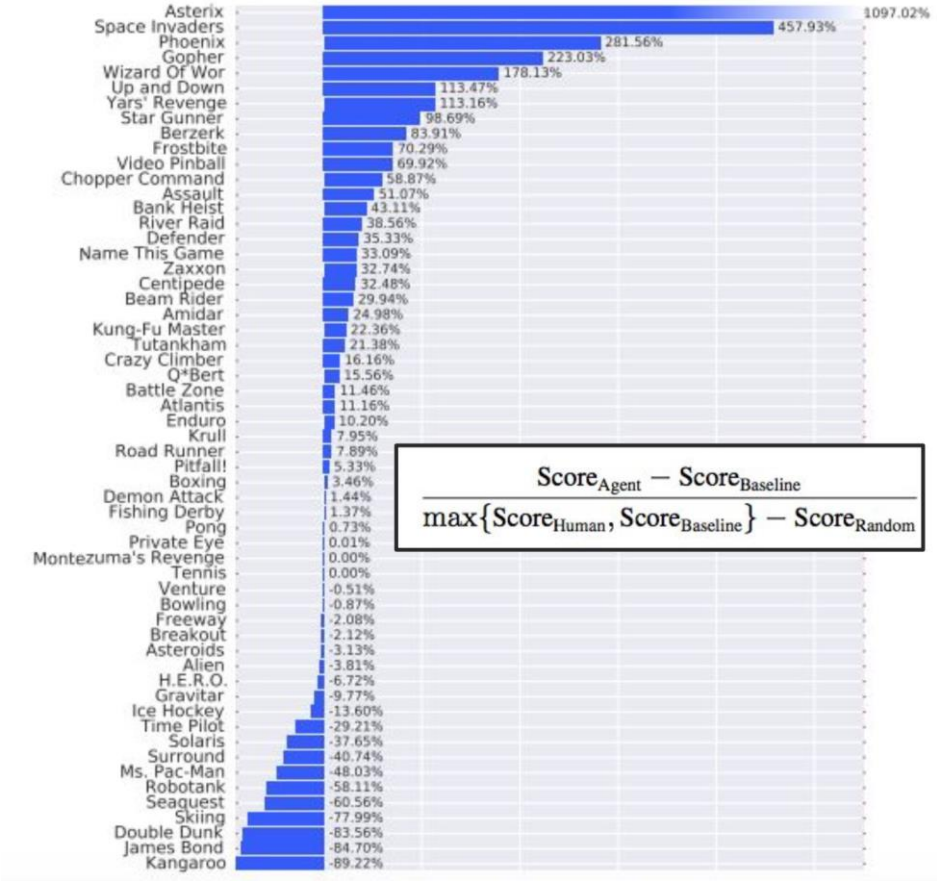


Figure: Wang et al, ICML 2016

Deep RL with Q-Functions

- Performance of Prioritized Experience Replay in Atari

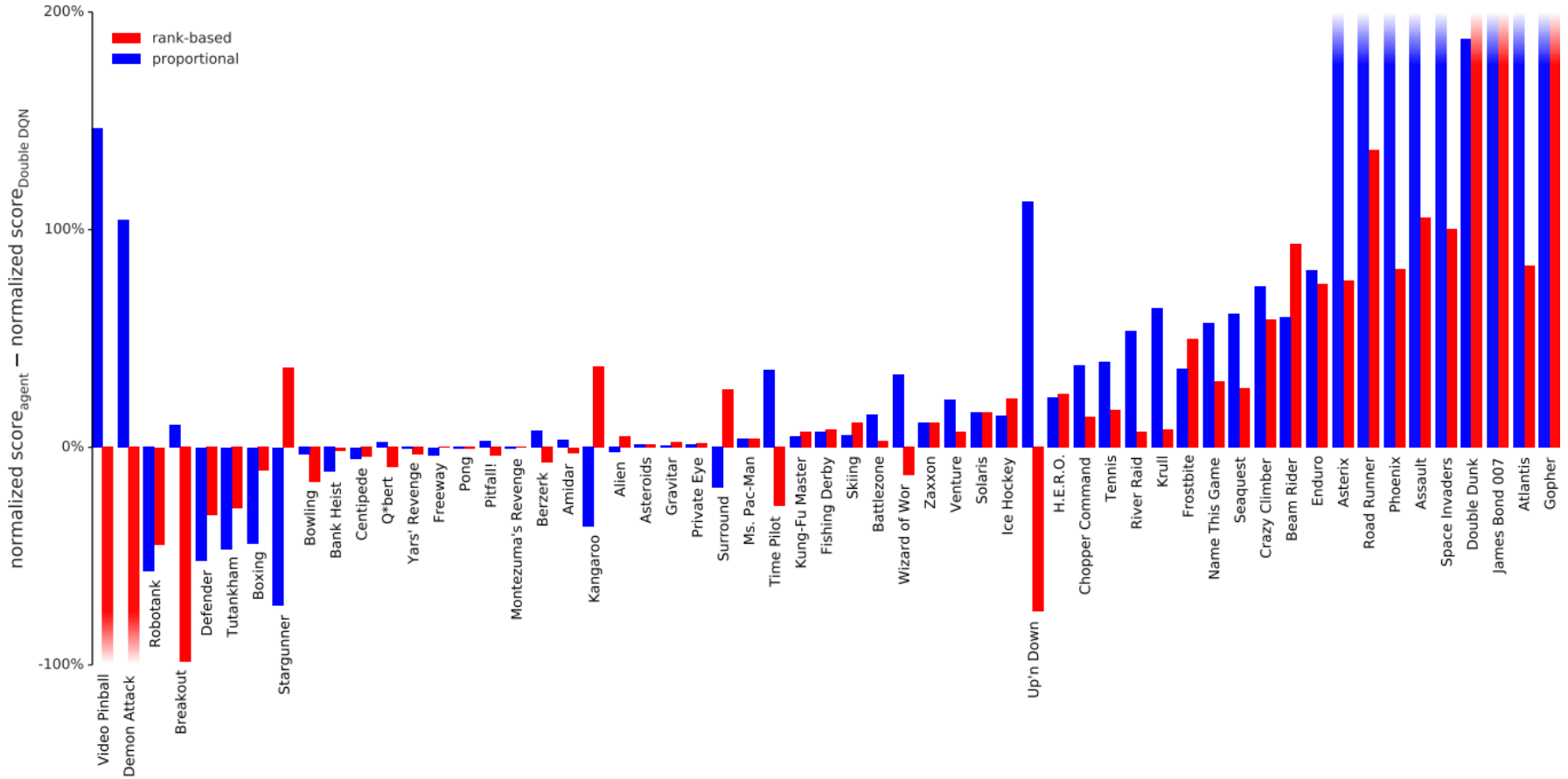


Figure: Schaul, Quan, Antonoglou, Silver ICLR 2016

Deep RL with Q-Functions

- Performance of Rainbow

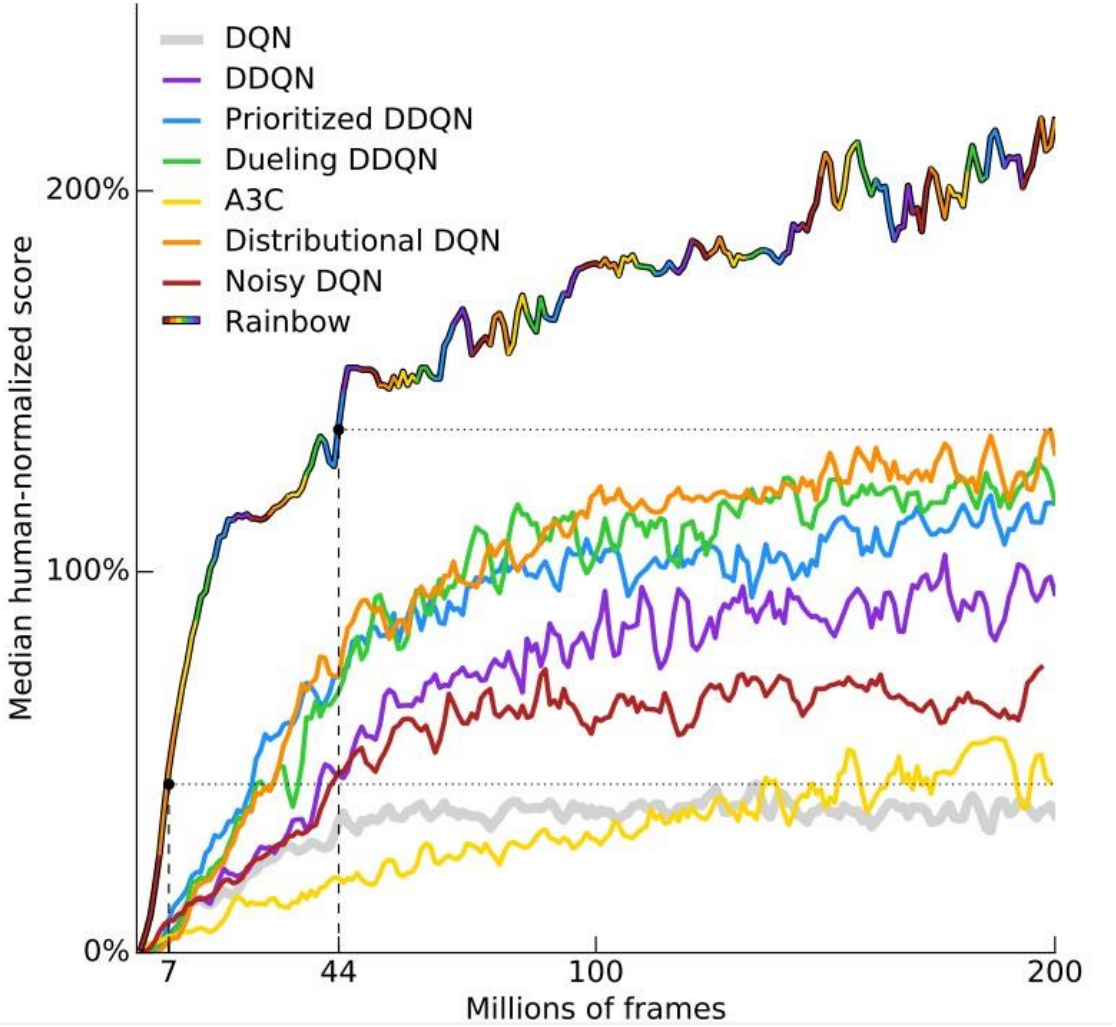


Figure: Hessel, Matteo, et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning."

Deep RL with Q-Functions

- Q-learning with continuous actions
  - Problem

$$\pi(\mathbf{a}_t|\mathbf{s}_t) = \begin{cases} 1 \text{ if } \mathbf{a}_t = \arg\max_{\mathbf{a}_t} Q_\phi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 \text{ otherwise} \end{cases}$$

$$\text{target value } y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$$

  - Solution
    - $\max_a Q(s,a) \approx \max\{Q(s,a_1), \dots, Q(s,a_N)\}, (a_1, \dots, a_N)$sampled from some distribution (e.g., uniform, Gaussian), but not very accurate.
    - Learn an approximate maximizer, Policy Gradient algorithm or DDPG (“deterministic” actor-critic, Lillicrap et al., ICLR 2016)

Deep RL with Policy Gradient

- Recap: Policy Gradient
  - Goal: given a policy $\pi_\theta(s,a)$ with parameters θ, find best θ that maximize $V(s,\theta)$
  - Can use gradient free optimization
    - Hill climbing、Cross-Entropy method etc.
  - Assume policy π_θ is differentiable and we can calculate gradient $\nabla_\theta \pi_\theta(s,a)$ analytically
    - Differentiable policy classes including: Softmax、Gaussian、Neural Networks
    - REINFORCE algorithm
    - A2C(Advantage Actor-Critic) algorithm
    - TRPO(Trust Region Policy Optimization) algorithm

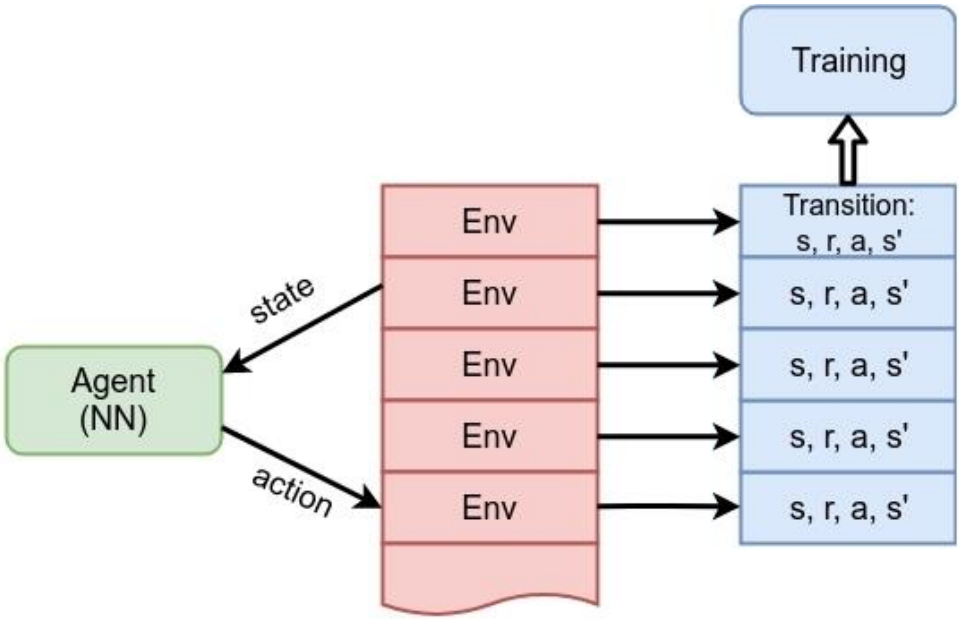
Deep RL with policy gradient

A3C

- Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy. The gradient of $J(\pi_\theta)$ is

$$\nabla_\theta J(\pi_\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)A^{\pi_\theta}(s_t,a_t)\right]$$

- Asynchronous A2C
  - Agents interact with their respective environments asynchronously, learning with each interaction. Each agent is controlled by a global network.

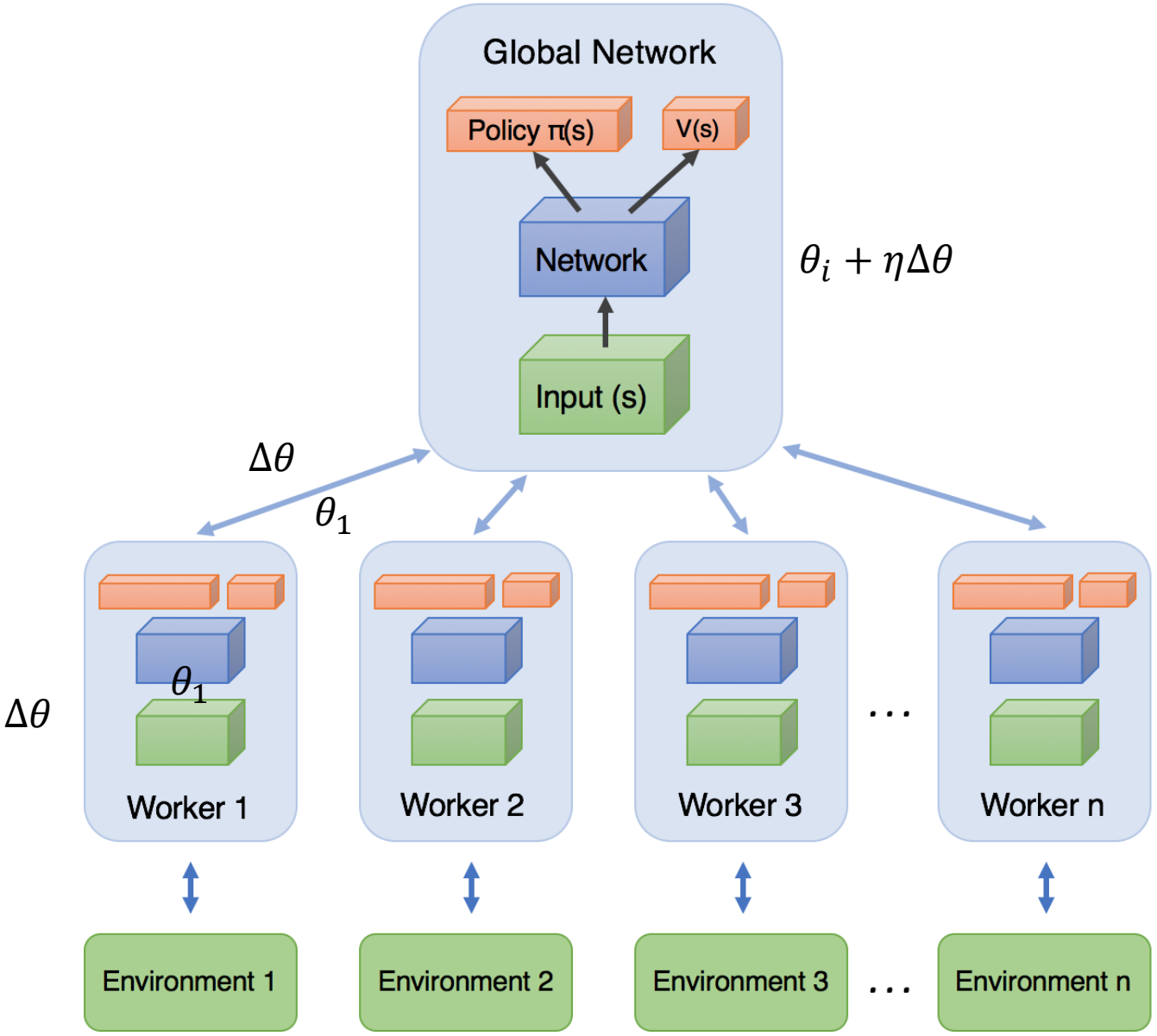


Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// 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 $T > T_{max}$

Deep RL with policy gradient

- A3C



Deep RL with policy gradient

DDPG(Deep Deterministic Policy Gradient)

- Idea: train actor network $\mu_\theta(s) \approx argmax_a Q_\phi(s,a)$
- Use four neural networks: a Q network, a deterministic policy network , a target q network, a target policy network
- The Q network and policy network is similar to actor-critic algorithm. But the Actor directly maps states to actions instead of outputting the probability distribution across a action space.
- Actor network:

$$\theta \leftarrow argmax_\theta Q_\phi\big(s, \mu_\theta(s)\big), \frac{dQ_\phi}{d\theta} = \frac{da}{d\theta}\frac{dQ_\phi}{da}$$

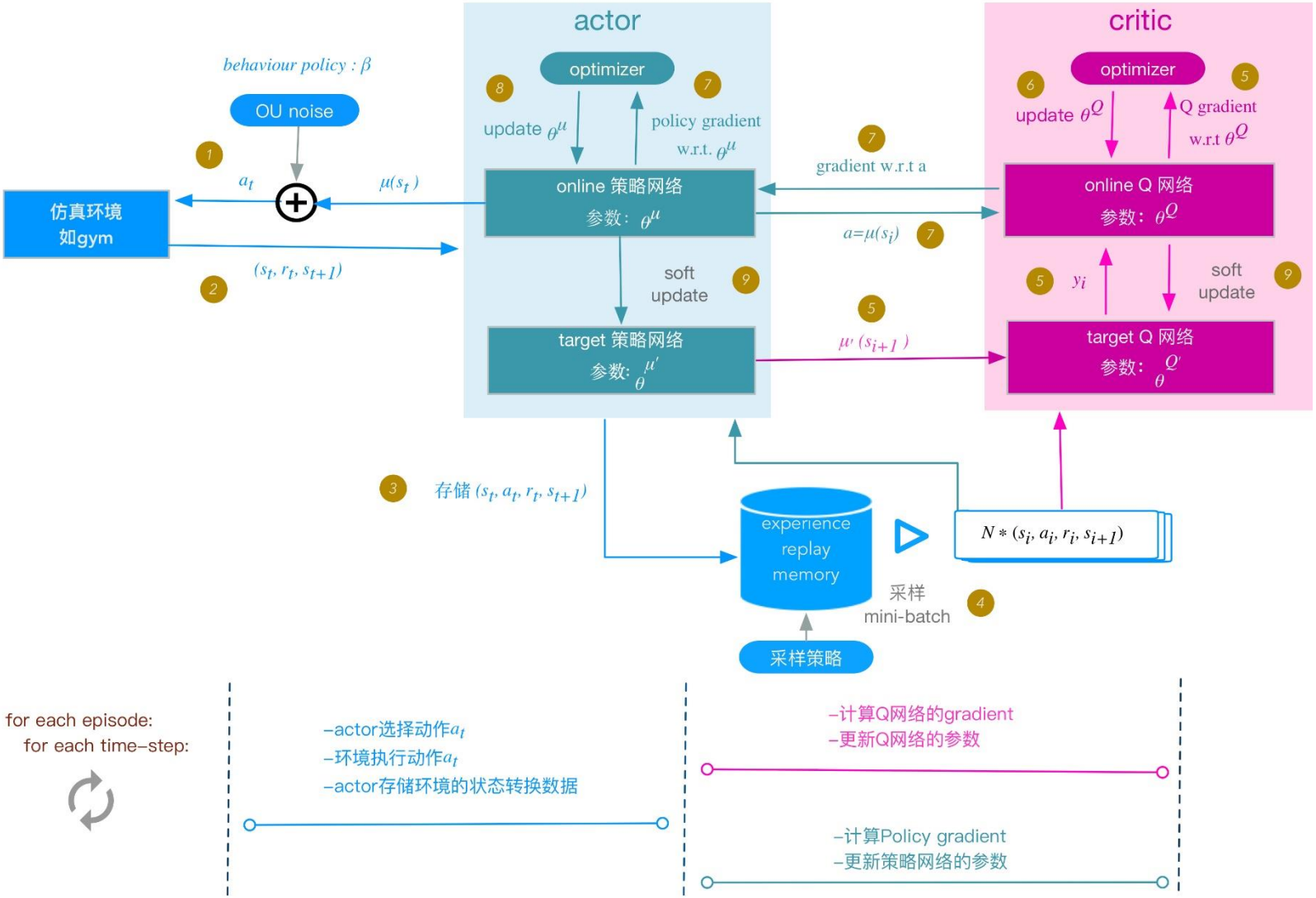
- Critic network: $y_j = r_j + \gamma Q_{\phi'}\left(s_j', \mu_{\theta'}(s_j')\right)$

$$\approx r_j + \gamma Q_{\phi'}\left(s_j', argmax_{a'} Q_{\phi'}(s_j', a_j')\right)$$

Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." *arXiv preprint arXiv:1509.02971* (2015).

Deep RL with policy gradient

Network of DDPG



Deep RL with policy gradient

- DDPG
  - **Pseudo Code**

- DDPG
  - Pseudo Code

- Soft Updates(different with DQN)
  - Slowly track those of the learned networks via "soft updates"

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$
$$\phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

- Soft Updates(different with DQN)
  - Slowly track those of the learned networks via "soft updates"

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$
$$\phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

Deep RL with policy gradient

TRPO

- TRPO updates policies by taking the largest step possible to improve performance, while satisfying a special constraint on how close the new and old policies are allowed to be. The constraint is expressed in terms of KL-Divergence.
  - let π_θ denote a policy with parameter θ. The theoretical TRPO update is

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

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

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

Schulman, John, et al. "Trust region policy optimization." *International conference on machine learning*. PMLR, 2015.

Deep RL with policy gradient

TRPO

$$\eta(\tilde{\pi}) = \eta(\pi) + E_{s_0, a_0, \cdots \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t, a_t) \right]$$

$$E_{\tau|\tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t, a_t) \right]$$

$$= E_{\tau|\dot{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t \left(r(s) + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t) \right) \right]$$

$$= E_{\tau|\dot{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t \left(r(s_t) \right) + \sum_{t=0}^{\infty} \gamma^t \left(\gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t) \right) \right]$$

$$= E_{\tau|\dot{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t \left(r(s_t) \right) \right] + E_{s_0} \left[-V^{\pi}(s_0) \right]$$

$$= \eta(\tilde{\pi}) - \eta(\pi)$$

Deep RL with policy gradient

TRPO

$$\eta(\tilde{\pi}) = \eta(\pi) + E_{s_0, a_0, \cdots \sim \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi}(s_t, a_t) \right]$$

$$\eta(\tilde{\pi}) = \eta(\pi) + \sum_{t=0}^{\infty} \sum_{s} P(s_t = s \mid \tilde{\pi}) \sum_{a} \tilde{\pi}(a \mid s) \gamma^t A_{\pi}(s, a)$$

Assume: $\rho_{\pi}(s) = P(s_0 = s) + \gamma P(s_1 = s) + \gamma^2 P(s_2 = s) + \cdots$

$$\eta(\tilde{\pi}) = \eta(\pi) + \sum_{s} \rho_{\tilde{\pi}}(s) \sum_{a} \tilde{\pi}(a \mid s) A^{\pi}(s, a)$$

Trick:

$$L_{\pi}(\tilde{\pi}) = \eta(\pi) + \sum_{s} \rho_{\pi}(s) \sum_{a} \tilde{\pi}(a \mid s) A^{\pi}(s, a)$$

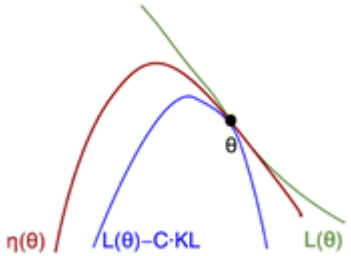
$$\sum_{a} \tilde{\pi}_{\theta}(a \mid s_n) A_{\theta_{\text{old}}}(s_n, a) = E_{a \sim q} \left[\frac{\tilde{\pi}_{\theta}(a \mid s_n)}{q(a \mid s_n)} A_{\theta_{\text{old}}}(s_n, a) \right]$$

$$L_{\pi}(\tilde{\pi}) = \eta(\pi) + E_{s \sim \rho_{\theta_{\text{old}}}, a \sim \pi_{\theta_{\text{old}}}} \left[\frac{\tilde{\pi}_{\theta}(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)} A_{\theta_{\text{old}}}(s, a) \right]$$

Deep RL with policy gradient

TRPO

$$L_{\pi_{\theta_{\text{old}}}}\left(\pi_{\theta_{\text{old}}}\right)=\eta\left(\pi_{\theta_{\text{old}}}\right) \quad \nabla_\theta L_{\pi_{\theta_{\text{old}}}}\left(\pi_\theta\right)\Big|_{\theta=\theta_{\text{old}}}=\nabla_\theta \eta\left(\pi_\theta\right)\Big|_{\theta=\theta_{\text{old}}}$$



$$\eta(\tilde{\pi}) \geqslant L_\pi(\tilde{\pi}) - CD_{KL}^{\max}(\pi,\tilde{\pi}) \quad \text{where } C=\frac{2\varepsilon\gamma}{(1-\gamma)^2}$$

C is the penalty coefficient that determines the step size

$$\text{maximize}_\theta\left[L_{\theta_{\text{old}}}(\theta)-CD_{KL}^{\max}\left(\theta_{\text{old}},\theta\right)\right]$$

Optimization objective

Rewrite the above equation

$$\text{maximize}_\theta\, E_{s\rho_{\theta_{\text{old}}},a\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a\mid s)}{\pi_{\theta_{\text{old}}}(a\mid s)}A_{\theta_{\text{old}}}(s,a)\right] \quad \text{subject to } D_{KL}^{\max}\left(\theta_{\text{old}},\theta\right)\leq\delta$$

Deep RL with policy gradient

- PPO(Proximal Policy Optimization)
  - 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.
  - 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

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

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a\mid s)}{\pi_{\theta_k}(a\mid s)} A^{\pi_{\theta_k}}(s,a),\quad \mathrm{clip}\left(\frac{\pi_\theta(a\mid s)}{\pi_{\theta_k}(a\mid s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s,a)\right)$$

Schulman, John, et al. "Proximal policy optimization algorithms." *arXiv preprint arXiv:1707.06347* (2017).

Deep RL with policy gradient

PPO

Pseudo Code

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

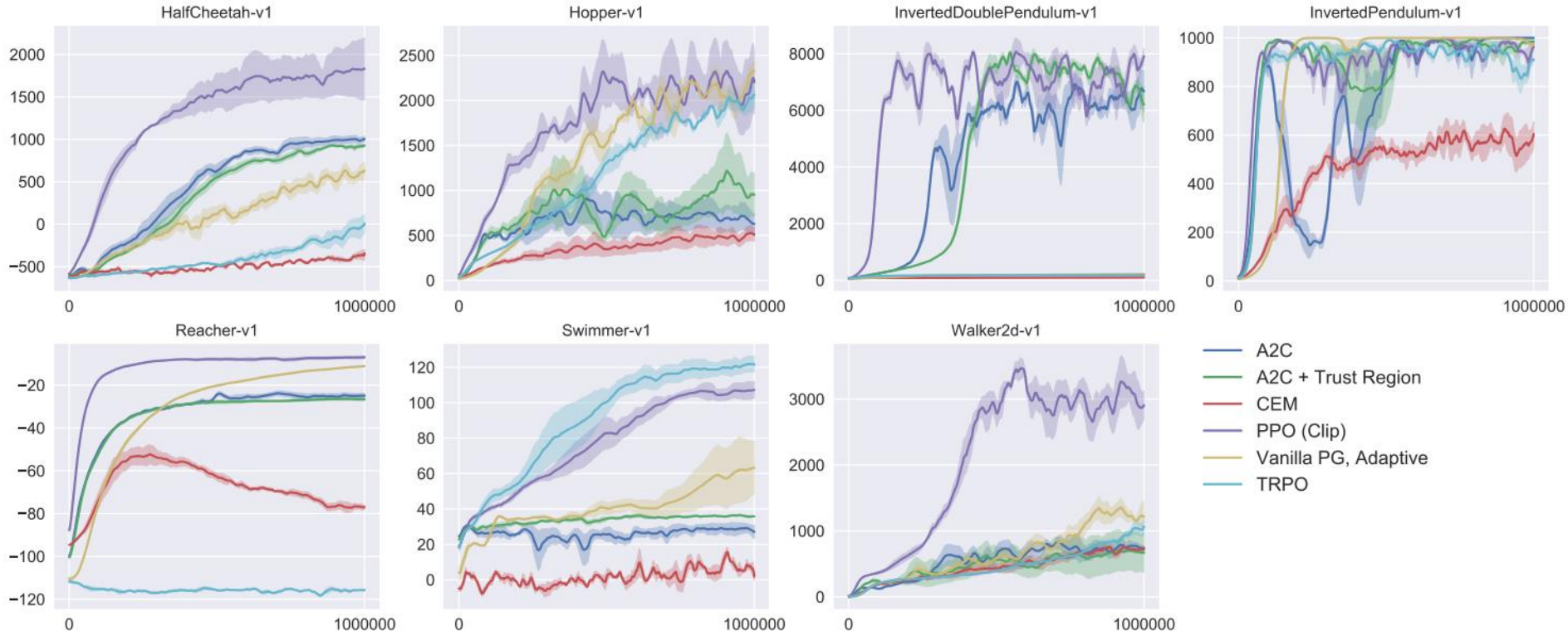
typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Deep RL with policy gradient

- PPO
  - **Performance**



Schulman, John, et al. "Proximal policy optimization algorithms." *arXiv preprint arXiv:1707.06347* (2017).

Deep RL with policy gradient

- SAC(Soft Actor Critic)
  - Entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy
  - Entropy is a quantity which says how random a random variable is
  - The RL problem:

$$\pi^* = \arg\max_\pi \mathop{\mathrm{E}}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_t,a_t,s_{t+1})+\alpha H\left(\pi(\cdot\,|\,s_t)\right)\right)\right]$$

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

$$Q^\pi(s,a)\approx r+\gamma\left(Q^\pi(s',\tilde{a}')-\alpha\log\pi(\tilde{a}'\,|\,s')\right),\quad \tilde{a}'\sim\pi(\cdot\,|\,s')$$

Haarnoja, Tuomas, et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor." *International conference on machine learning*. PMLR, 2018.

Deep RL with policy gradient

- SAC and TD3
  - **Pseudo Code**

Algorithm 1 Soft Actor-Critic

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

TD3 →

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

13: Update Q-functions by one step of gradient descent using

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

14: 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 θ via the reparametrization trick.
15: Update target networks with

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

16: **end for**
17: **end if**
18: **until** convergence

Deep RL with policy gradient

TD3(Twin Delayed Deep Deterministic)

- Solve overestimation in DDPG
- Clipped Double-Q learning
- "Delayed" Policy Updates
- Target Policy Smoothing

Algorithm 1 TD3

Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ

Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$ double q network

Initialize replay buffer $\mathcal{B}$

for $t = 1$ **to** T **do**

Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'

Store transition tuple (s, a, r, s') in $\mathcal{B}$

Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$

$\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$ 正则化

$y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$ clipped double q

Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$

if t mod d **then** Delaying policy update

Update ϕ by the deterministic policy gradient:

$\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$

Update target networks:

$\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$

$\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$

end if

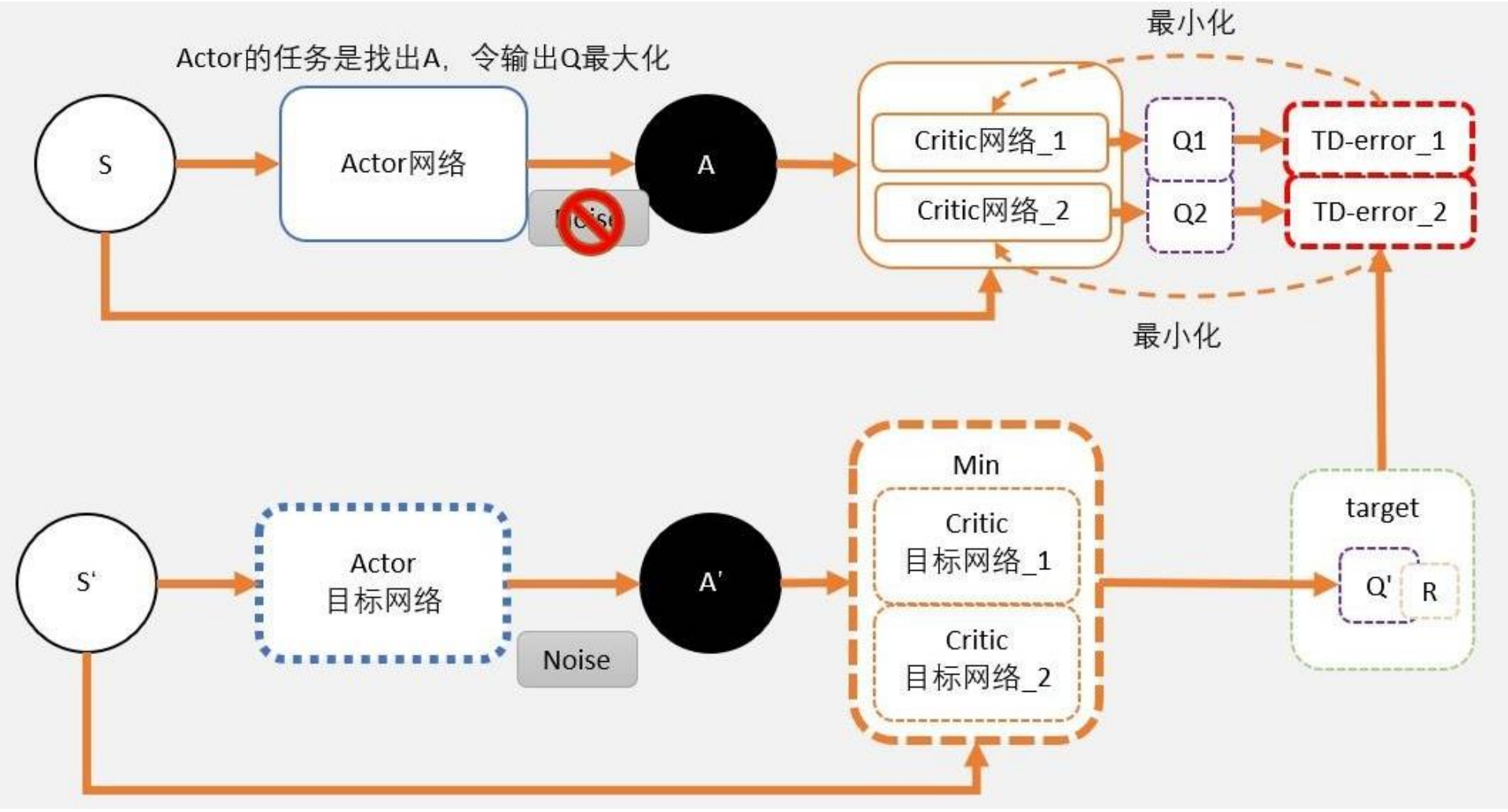
end for

Fujimoto, Scott, Herke Hoof, and David Meger. "Addressing function approximation error in actor-critic methods." *International conference on machine learning*. PMLR, 2018.

Deep RL with policy gradient

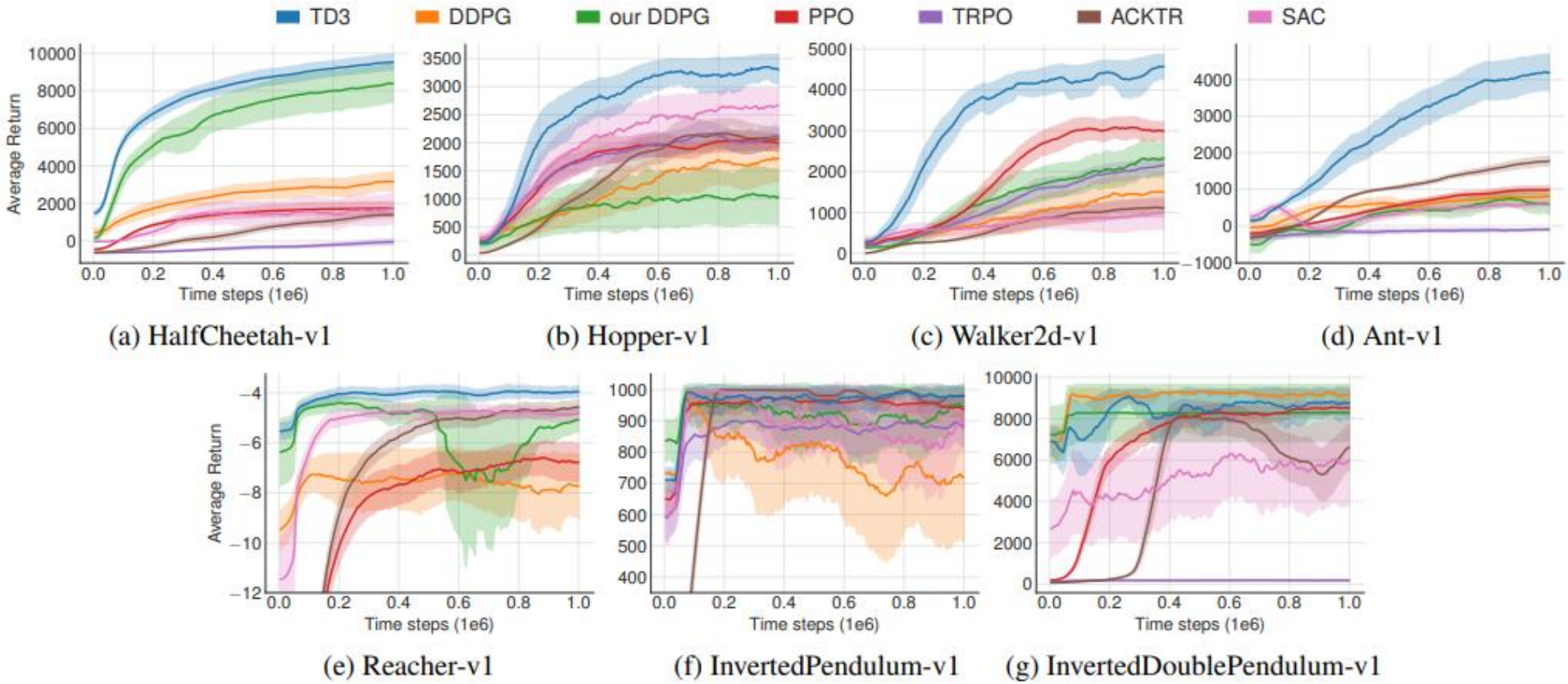
TD3(Twin Delayed Deep Deterministic)

Solve overestimation in DDPG



Deep RL with policy gradient

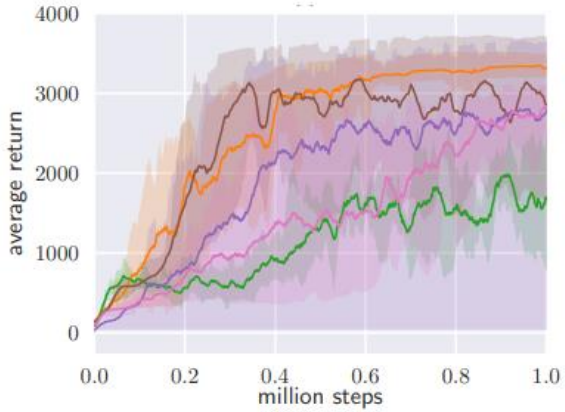
☐ TD3(Twin Delayed Deep Deterministic)



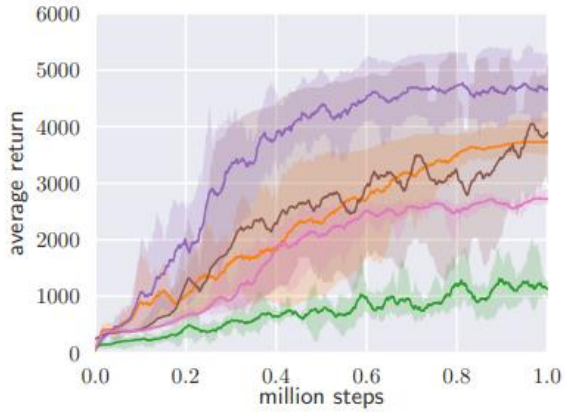
Deep RL with policy gradient

SAC

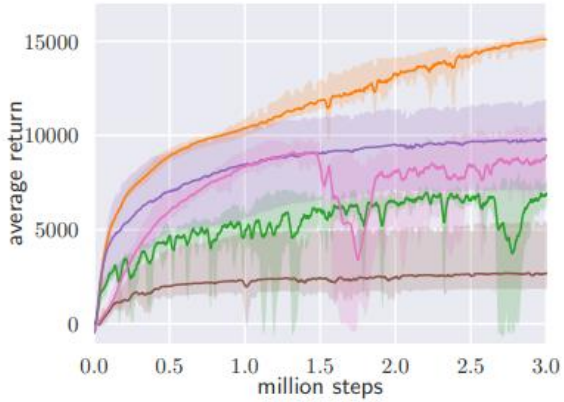
Performance



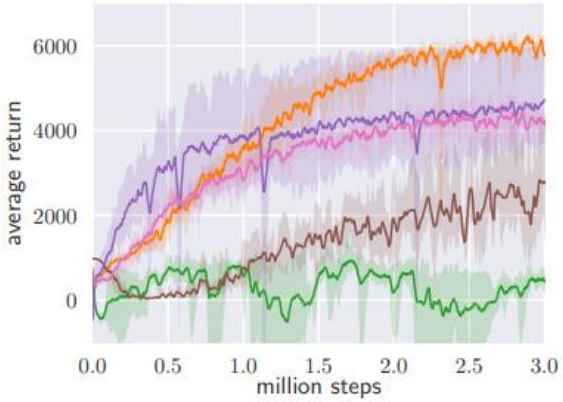
(a) Hopper-v1



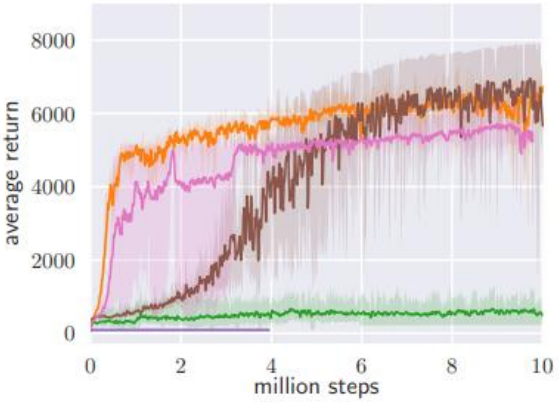
(b) Walker2d-v1



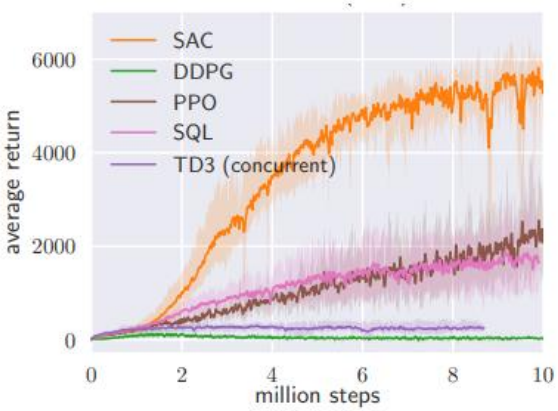
(c) HalfCheetah-v1



(d) Ant-v1

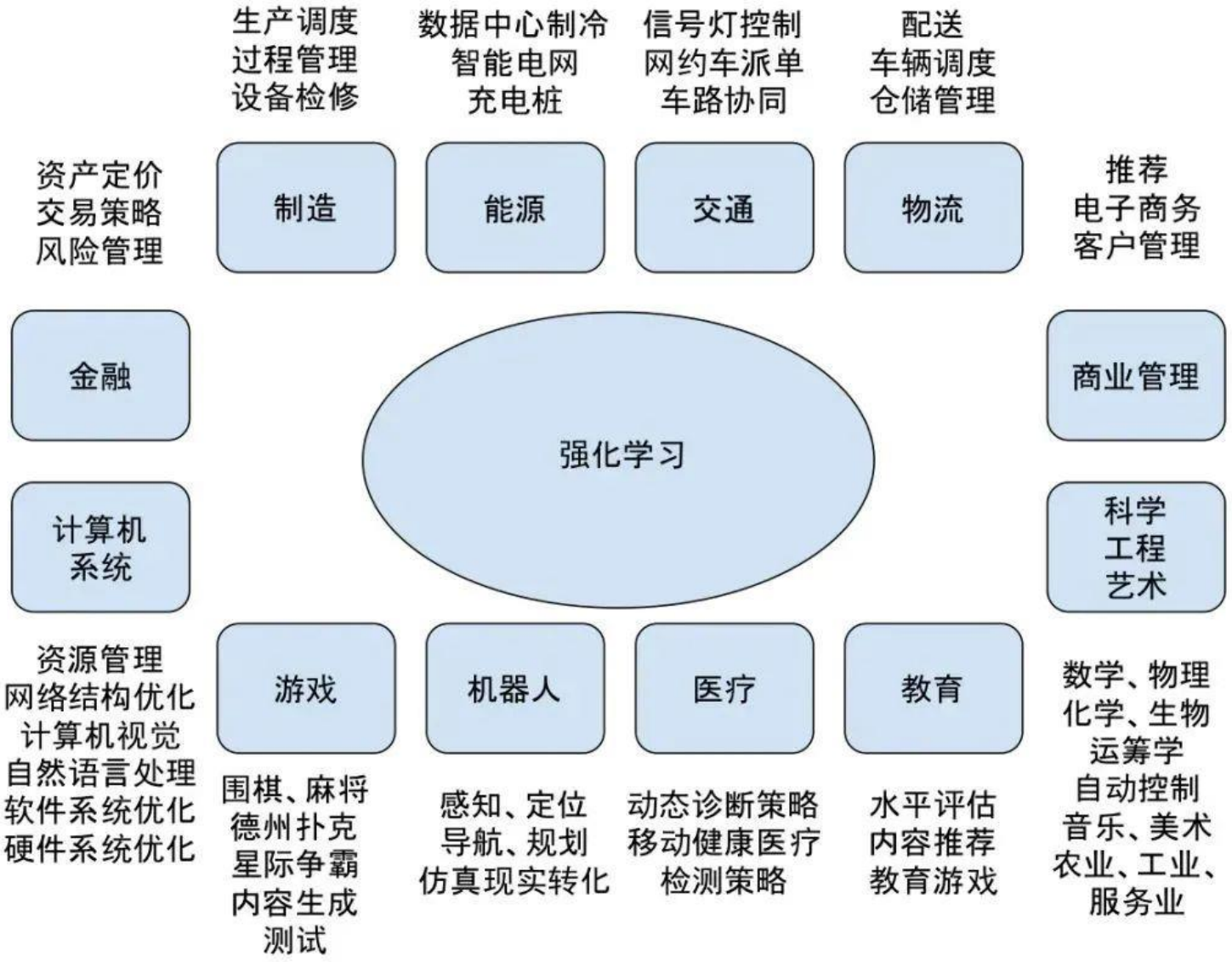


(e) Humanoid-v1



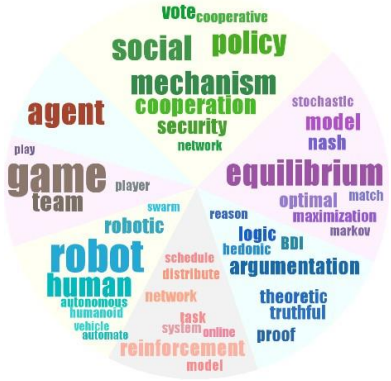
(f) Humanoid (rllab)

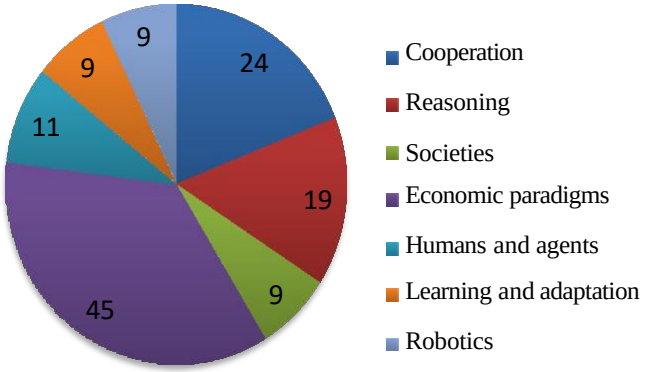
Application of Deep RL

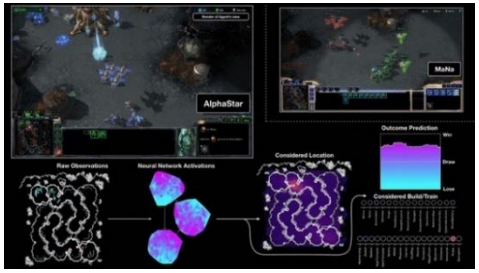


Application of Deep RL

- Game: Go、Texas Hold'em Poker、Honor of Kings、Dota、StarCraft、Atari、Football, etc.
- UAV(Unmanned Aerial Vehicle)
- Autonomous Driving
- Traffic flow control
- Finance
- Medicine
- Robotics

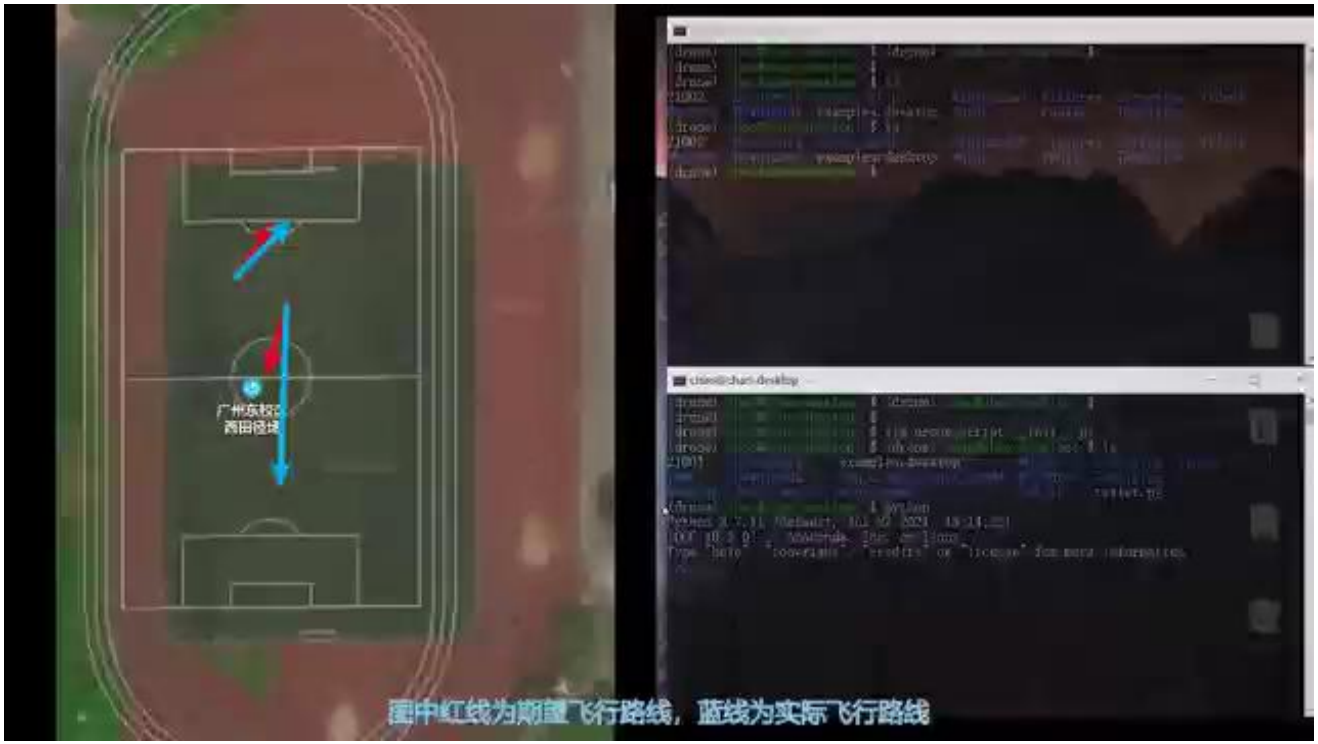






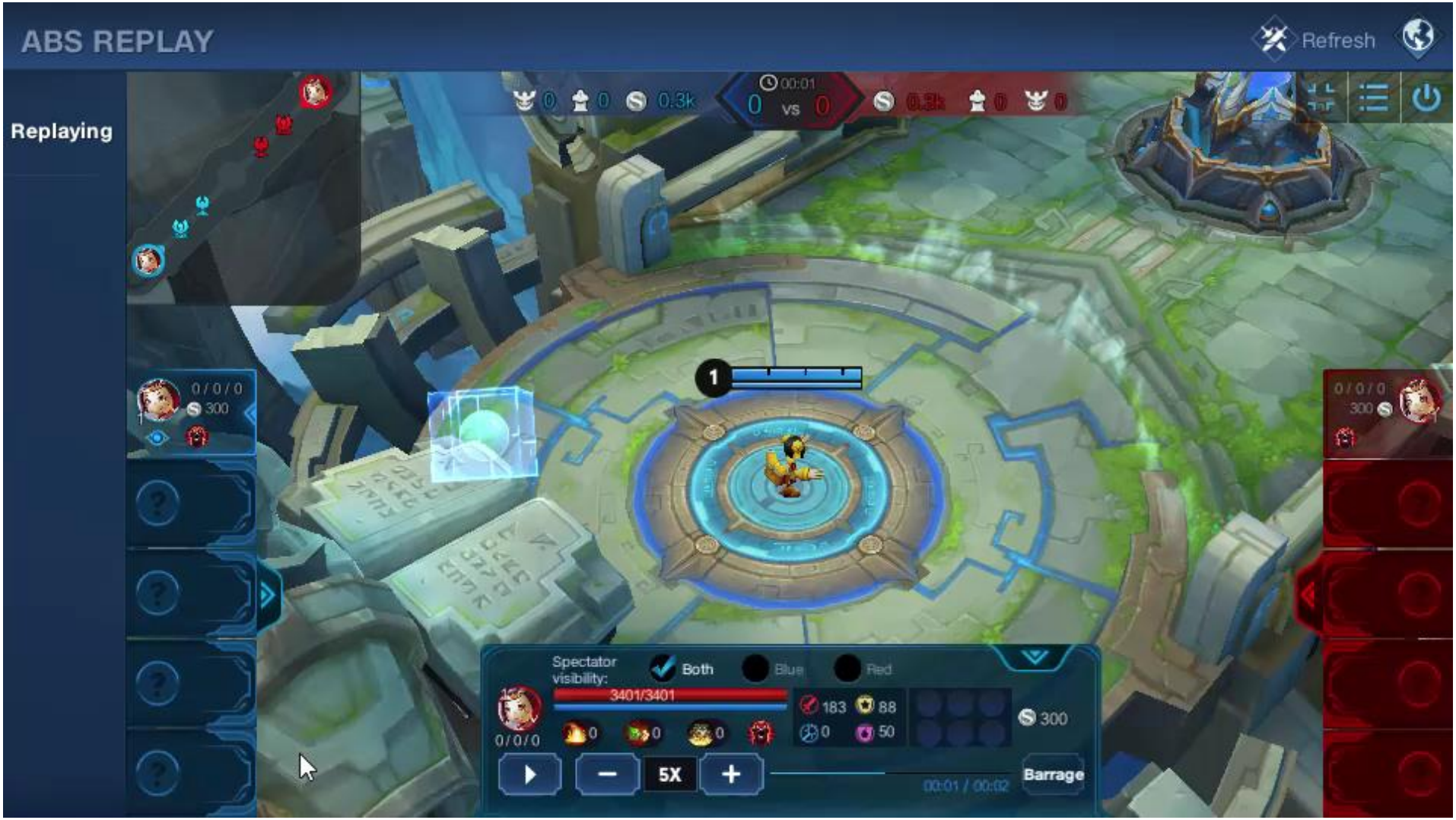
Application of Deep RL

- UAV



Application of Deep RL

- Honor of Kings



Tricks in Deep RL

- Simplify the problem by using a low-dimensional state space or action space
- Simplify the reward function
- Scaling observation and reward: normalization, clipping, etc.
- GAE, λ-return, etc.
- Exploration and Exploitation: entropy, Epsilon annealing, etc.
- Parallelized environment
- Test your algorithm on a known baseline environment
- Mini-batch update
- Parameter sharing
- Activation function: relu and tanh
- Orthogonal initialization and layer scaling
- Optimizer: Adam or RMSprop
- Global Gradient Clipping
- Value Function Loss Clipping
- Try different random seeds
- Look at episode return closely

Deep Reinforcement Learning

谢 谢!

汇报人：张宇聪

中山大学计算机学院