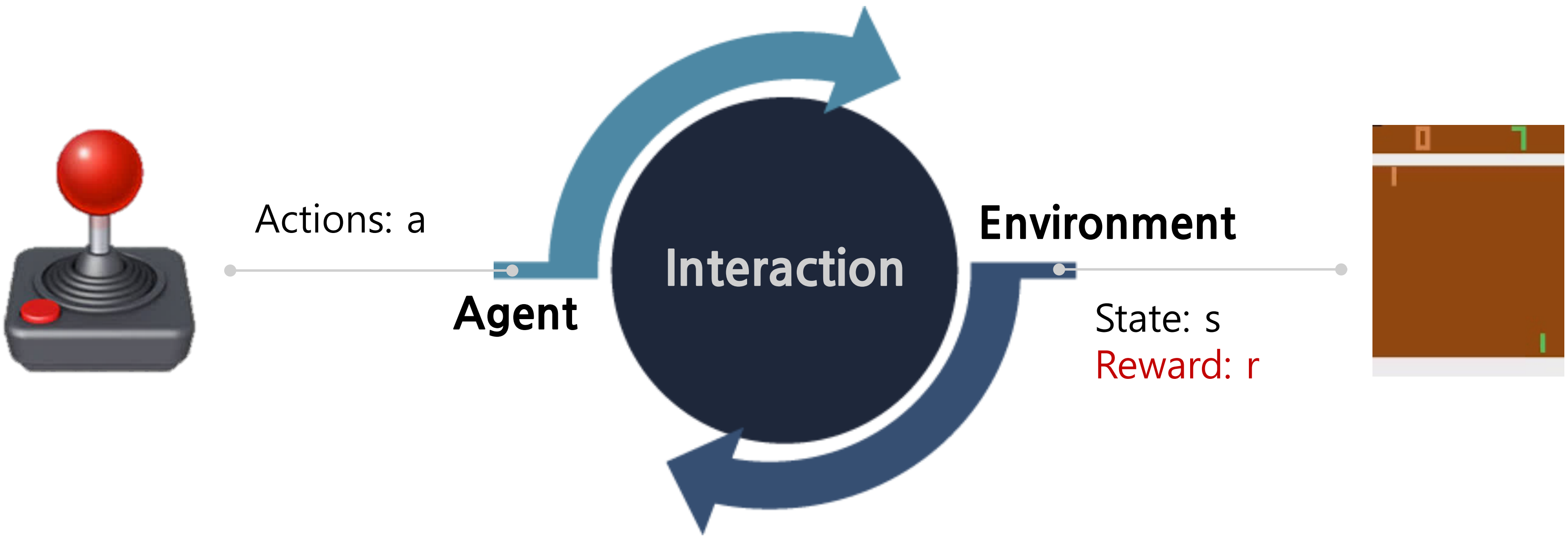
Actor-Critic Reinforcement Learning Algorithms

고려대학교
박영준

Contents

Reinforcement Learning (RL)

- Find policy $\pi(a|s)$ while maximize cumulative rewards
- It is called *approximate dynamic programming*



Policy

- In Deep RL, policy is parametrized by neural networks (θ)

- Deterministic policy

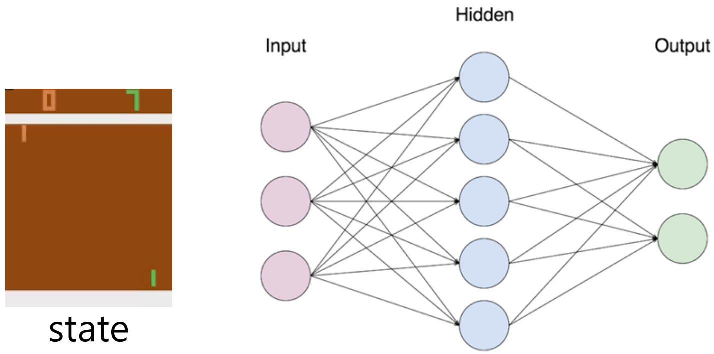
$$\boldsymbol{a} = \underset{a}{\operatorname{argmax}}\, \boldsymbol{\pi}(\boldsymbol{S})$$

- Stochastic Policy

$$\boldsymbol{a} \sim \boldsymbol{\pi}(\boldsymbol{S})$$

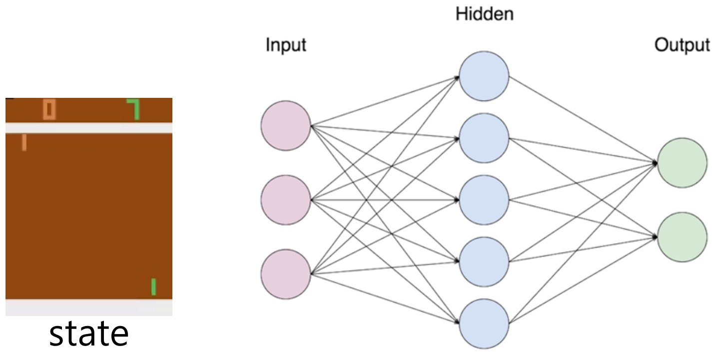
Policy Gradient & Q-Learning

- Policy gradient



$$P(a = a_1 | S = s)$$
$$P(a = a_2 | S = s)$$

- Q-learning



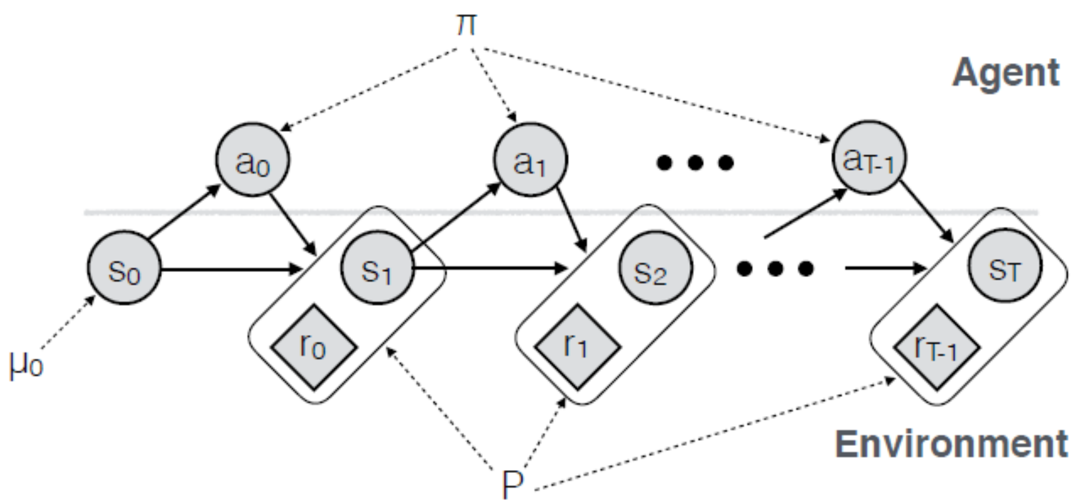
$$Q(a = a_1 | S = s)$$
$$Q(a = a_2 | S = s)$$

$Q(s, a) = E[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots | s_0 = s, a_0 = a]$: Q-function

Training Policy Gradient

- Episodic update: REINFORCE

$$\max E[G|\pi_\theta] \qquad where\ G = \sum_{t=0}^{T-1} r_t$$



Gradient Estimator (optional)

- $E_{x \sim p(x|\theta)}[f(x)]$

$$\nabla_\theta E_x[f(x)] = \nabla_\theta \int p(x|\theta) f(x) dx$$

$$= \int \nabla_\theta p(x|\theta) f(x) dx$$

$$= \int \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)} p(x|\theta) f(x) dx$$

$$= \int \nabla_\theta \log p(x|\theta)\, p(x|\theta) f(x) dx$$

$$= E_x[f(x) \nabla_\theta \log p(x|\theta)]$$

Sample $x_i \sim p(x|\theta)$, and compute $\hat{g}_i = f(x_i) \nabla_\theta \log p(x_i|\theta)$

Policy Gradient Estimator

- Now random variable x is a whole trajectory

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$$

$$\nabla_\theta E_\tau[G(\tau)] = E_\tau[G(\tau)\nabla_\theta \log p(\tau|\theta)]$$

- Just need to write out $p(\tau|\theta)$

$$p(\tau|\theta) = \mu(s_o)\prod_{t=0}^{T-1}[\pi(a_t|s_t,\theta)P(s_{t+1},r_t|s_t,a_t)]$$

$$\log p(\tau|\theta) = \log \mu(s_o) + \sum_{t=0}^{T-1}[\log \pi(a_t|s_t,\theta) + \log P(s_{t+1},r_t|s_t,a_t)]$$

$$\nabla_\theta \log p(\tau|\theta) = \nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t|s_t,\theta)$$

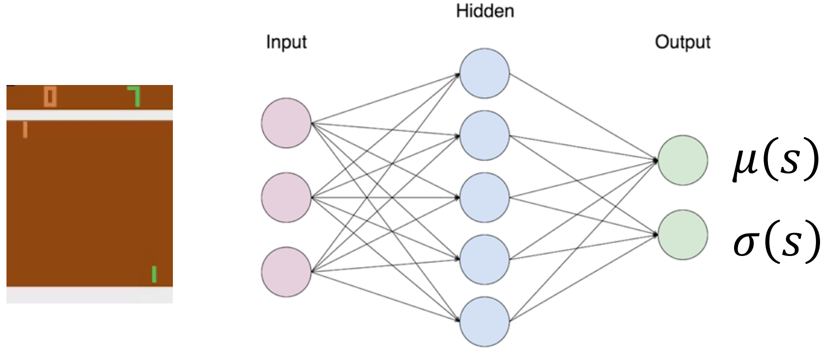
$$\nabla_\theta E_\tau[G] = E_\tau\left[G\nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t|s_t,\theta)\right]$$

Policy Gradient for Continuous Action

- DQN cannot solve continuous action tasks
- Policy gradient can simply solve the tasks (Gaussian policy parameterization)

$$\pi(s) = N\big(\mu(s), \sigma(s)\big)$$

$$a \sim N\big(\mu(s), \sigma(s)\big)$$



Sutton, R. S., et al., (2018). Reinforcement Learning: An Introduction, 338p

Policy Gradient Algorithm (REINFORCE)

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\quad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

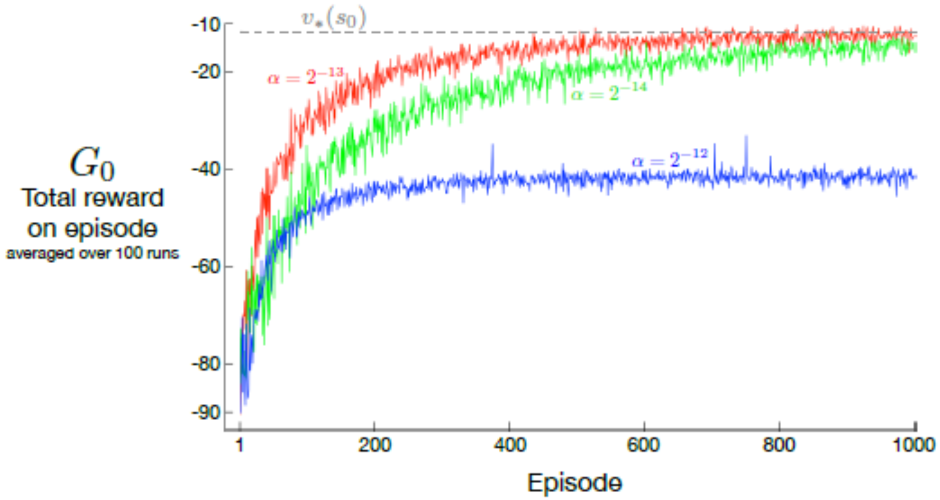


Figure 13.1: REINFORCE on the short-corridor gridworld (Example 13.1). With a good step size, the total reward per episode approaches the optimal value of the start state.

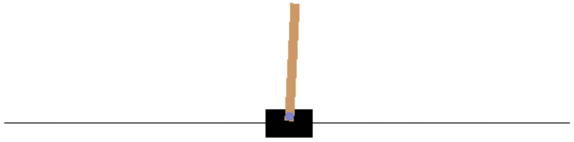
REINFORCE: Cartpole Example

- Policy $\pi(a|s)$

```
class Policy(nn.Module):
    def __init__(self, s_size=4, h_size=16, a_size=2):
        super(Policy, self).__init__()
        self.fc1 = nn.Linear(s_size, h_size)
        self.fc2 = nn.Linear(h_size, a_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.softmax(x, dim=1)

    def act(self, state):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        probs = self.forward(state).cpu()
        m = Categorical(probs)
        action = m.sample()
        return action.item(), m.log_prob(action)
```



https://github.com/udacity/deep-reinforcement-learning/blob/master/reinforce/REINFORCE.ipynb

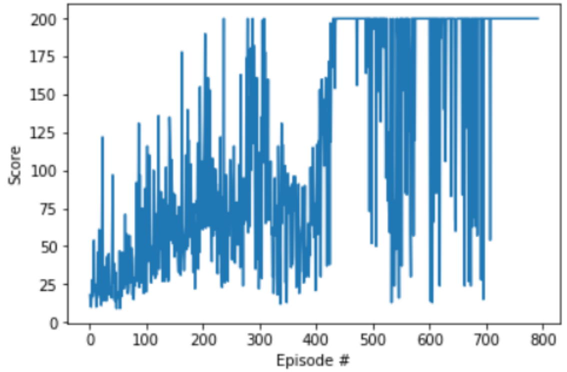
REINFORCE: Cartpole Example

```
for i_episode in range(1, n_episodes+1):
    saved_log_probs = []
    rewards = []
    state = env.reset()
    for t in range(max_t):
        action, log_prob = policy.act(state)
        saved_log_probs.append(log_prob)
        state, reward, done, _ = env.step(action)
        rewards.append(reward)
        if done:
            break
    scores_deque.append(sum(rewards))
    scores.append(sum(rewards))

    discounts = [gamma**i for i in range(len(rewards)+1)]
    R = sum([a*b for a,b in zip(discounts, rewards)])

    policy_loss = []
    for log_prob in saved_log_probs:
        policy_loss.append(-log_prob * R)
    policy_loss = torch.cat(policy_loss).sum()

    optimizer.zero_grad()
    policy_loss.backward()
    optimizer.step()
```

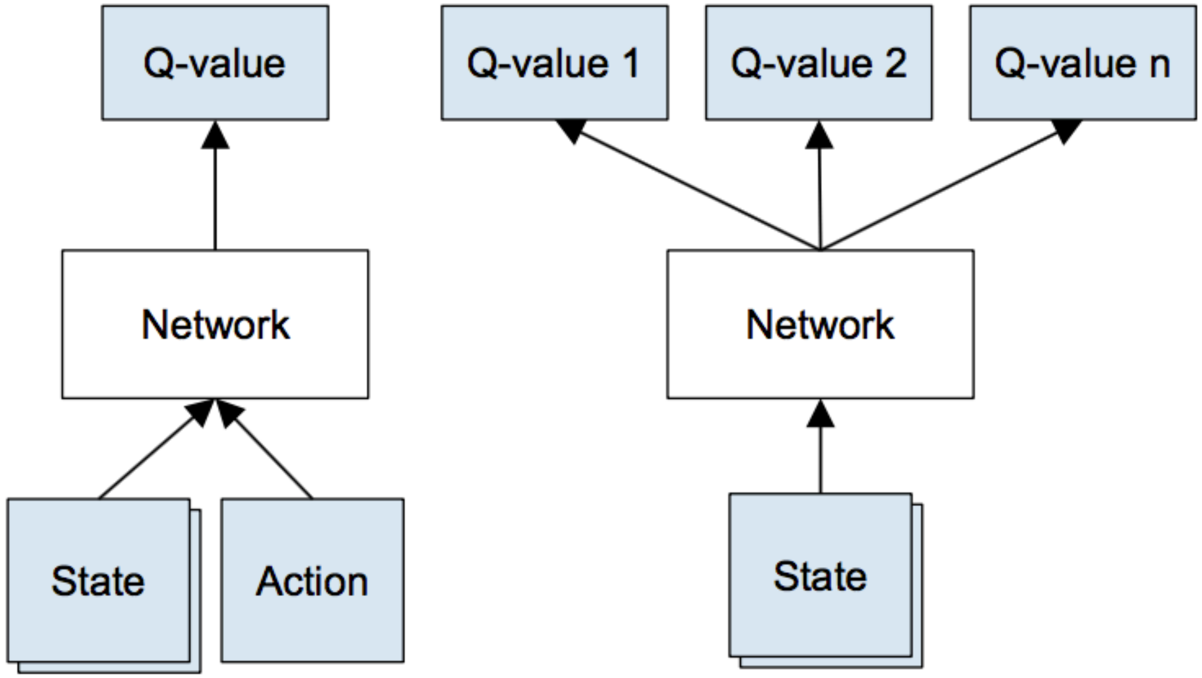


https://github.com/udacity/deep-reinforcement-learning/blob/master/reinforce/REINFORCE.ipynb

Q-Learning

- Q-function

$$Q(s,a) = E[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots | s_0 = s, a_0 = a]$$



DQN Algorithm

- Temporal difference learning
- Off-policy
- Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for

REINFORCE vs DQN

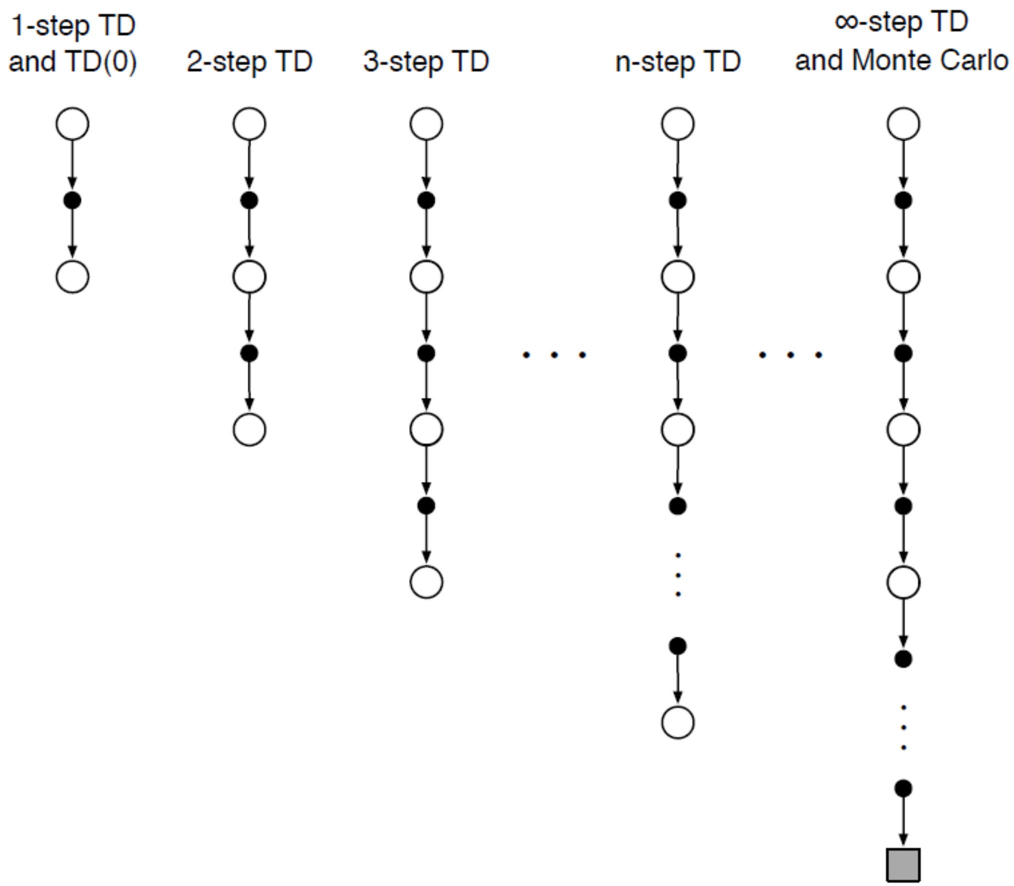
- Comparison between policy gradient and DQN

	REINFORCE	DQN
Estimation	Monte Carlo method	Bootstrapping (temporal difference)
Pros.	• Continuous action • Stochastic policies	• Low variance • Long episode
Cons.	• High variance • Local optimal	• No continuous action • Only deterministic policy

Estimation Method: n-step TD

- Bias-variance tradeoff

Low variance
High bias

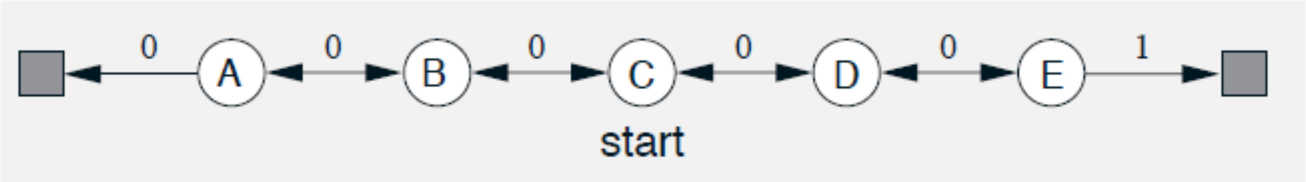


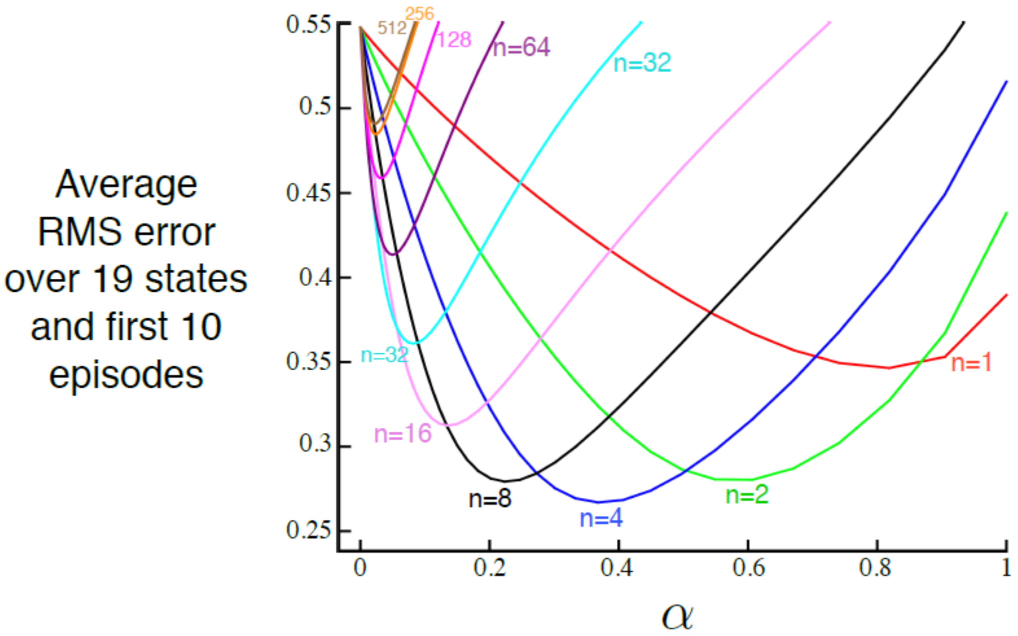
High variance
Low bias

https://www.quora.com/Intuitively-why-is-there-a-bias-variance-tradeoff-between-TD-k-0-and-TD-k-%E2%88%9E

Estimation Method: n-step TD

- n-step TD example

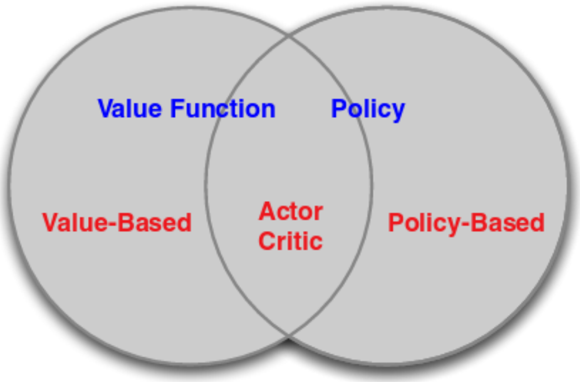


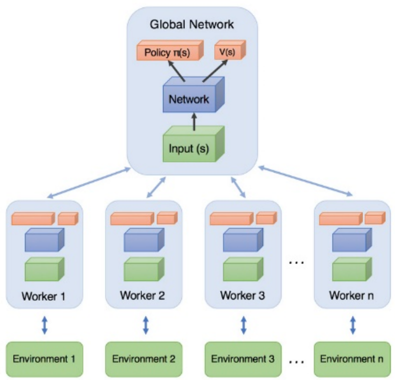


Contents

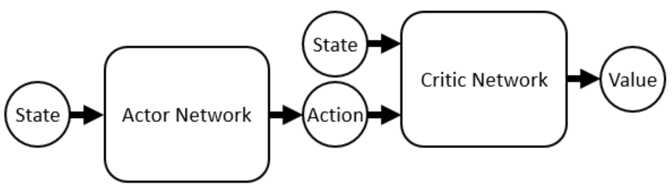
Actor-Critic Algorithms

- Actor-critic is hybrid of DQN & policy gradient
- Variants: A3C, DDPG, TRPO…





A3C

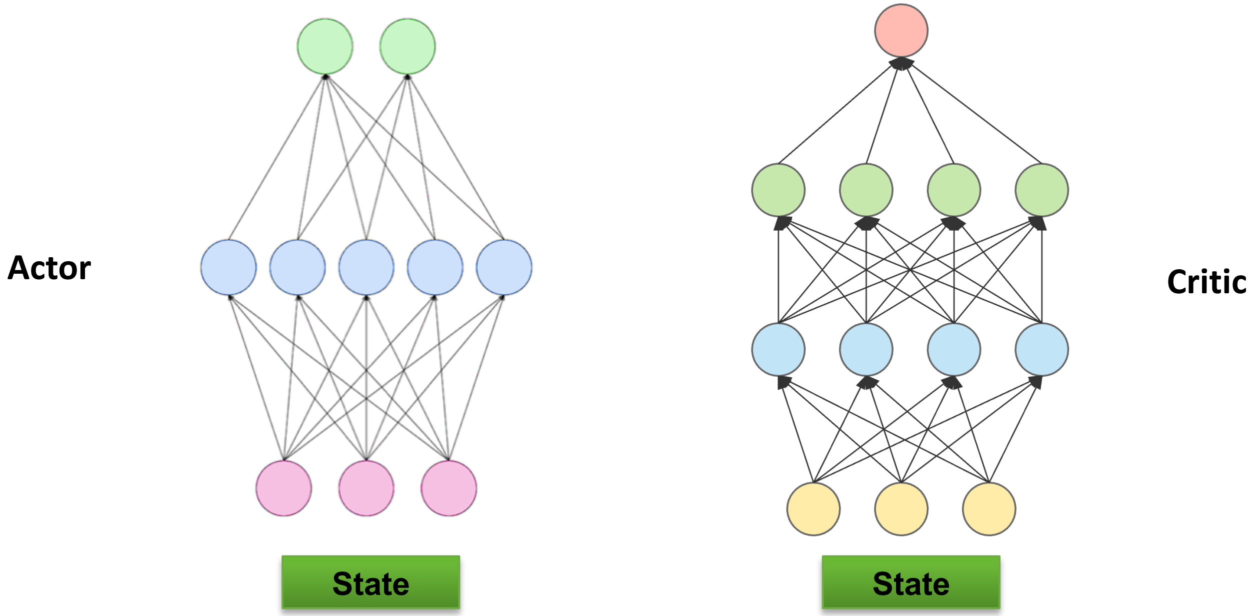


DDPG

Actor-Critic Method

- Network architectures
- Actor: policy network / Critic: value network

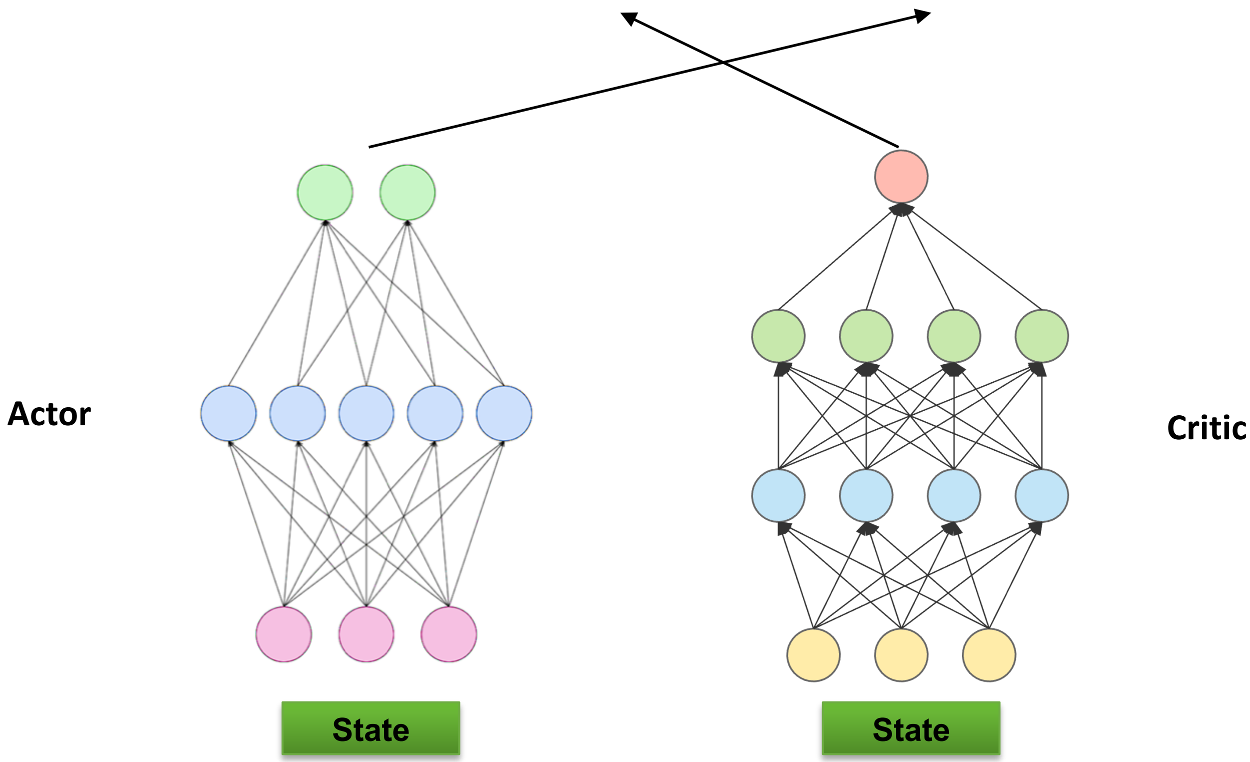
$$\theta \leftarrow \theta + \alpha \boldsymbol{G} \nabla_\theta \log \pi(a_t | s_t, \theta)$$



Actor-Critic Method

- Network architectures
- Actor: policy network / Critic: value network

$$\theta_a \leftarrow \theta_a + \alpha \boldsymbol{V(s|\theta_c)} \nabla_{\theta_a} \log \boldsymbol{\pi(a_t|s_t, \theta_a)}$$

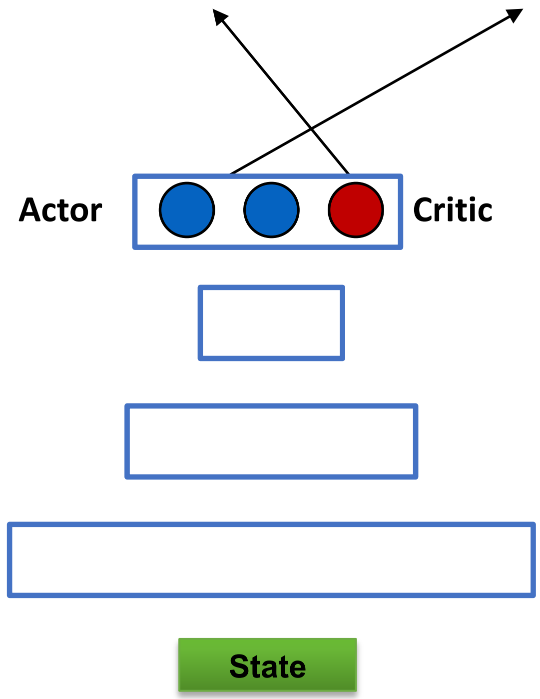


$V(s) = E[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots | s_0 = s\ ]$: V-function

Actor-Critic Method

- Network architectures
- Actor: policy network / Critic: value network

$$\theta_a \leftarrow \theta_a + \alpha \boldsymbol{V(s|\theta_c)} \nabla_{\theta_a} \log \boldsymbol{\pi(a_t|s_t, \theta_a)}$$



$V(s) = E[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots | s_0 = s]$: V-function

Actor-Critic Method

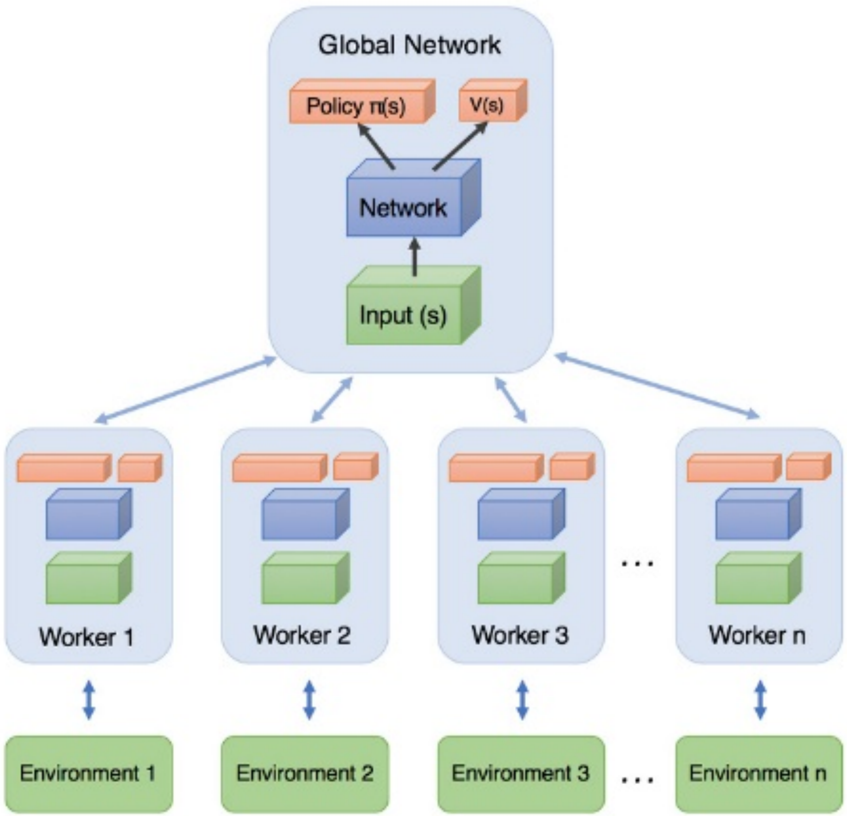
- Network architectures
- Actor: policy network / Critic: value network
- Update actor

$$\theta_a \leftarrow \theta_a + \alpha \boldsymbol{Score} \nabla_{\theta_a} \log \boldsymbol{\pi}(\boldsymbol{a_t}|\boldsymbol{s_t}, \boldsymbol{\theta_a})$$

- ***Score***
  - G *REINFORCE*
  - $Q(s, a)$ *Q Actor-Critic*
  - $A(s, a) = Q(s, a) - V(s)$ *Advantage Actor-Critic*
  - $\delta = r + \gamma V(s_{t+1}) - V(s_{t+1})$ *TD Actor-Critic*

A3C

- Asynchronous Advantage Actor-Critic (A3C)
  - Use CPU thread
  - Multiple Actor-Critic
  - Advantage
  - Episodic update



A3C

- No replay memory, On policy

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: $d\theta_v \leftarrow d\theta_v + \partial\left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
    end for
    Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Actor-Critic Update

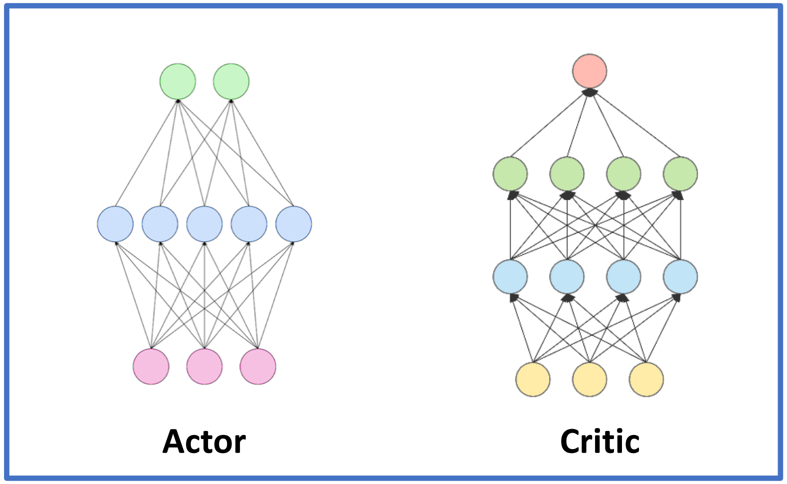
- For a single episode,

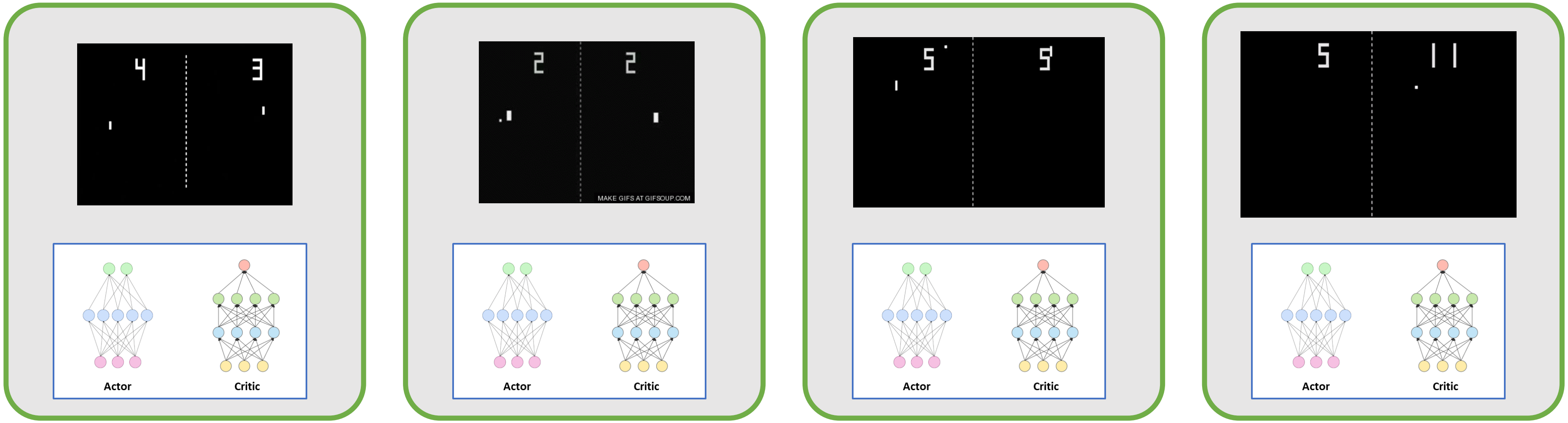
Episode Step	r	$R \leftarrow r_i + \gamma R$	$V(s)$	$\pi(a_t \mid s_t)$	$R - V(s)$
6	10	10+0=10	11		-1
5	0	0+9=9	8		1
4	0	0+8.1=8.1	7		1.1
3	0	0+7.3=7.3	6		1.3
2	0	0+6.6=6.6	5		1.6
1	0	0+5.9=5.9	4		1.9

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: $d\theta_v \leftarrow d\theta_v + \partial\left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
end for

Asynchronous Update

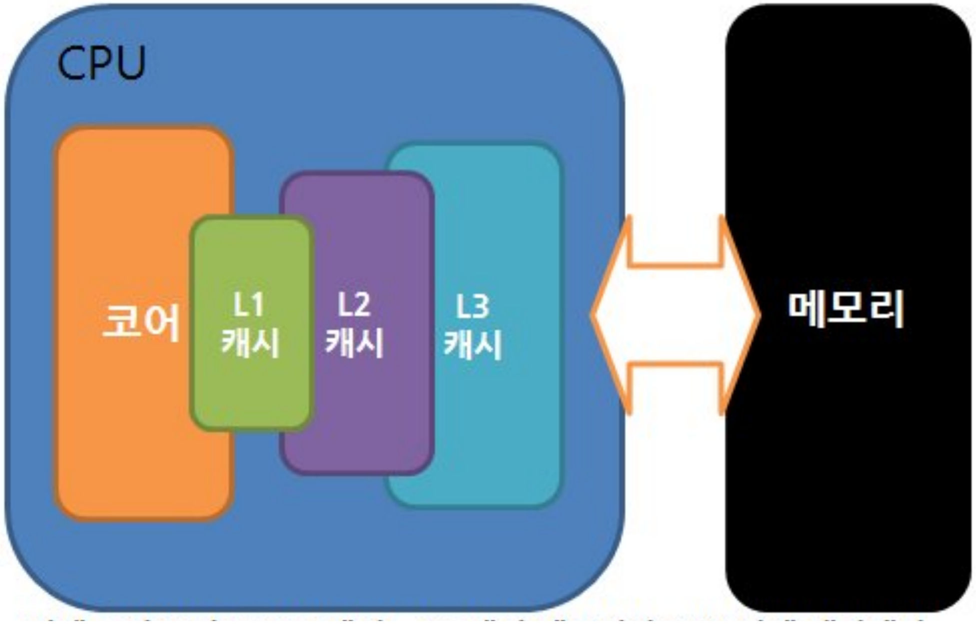
- Uncorrelated experience





Asynchronous Update

- Fast communication speed between CPU & memory



A3C Performance

- 5 Atari games

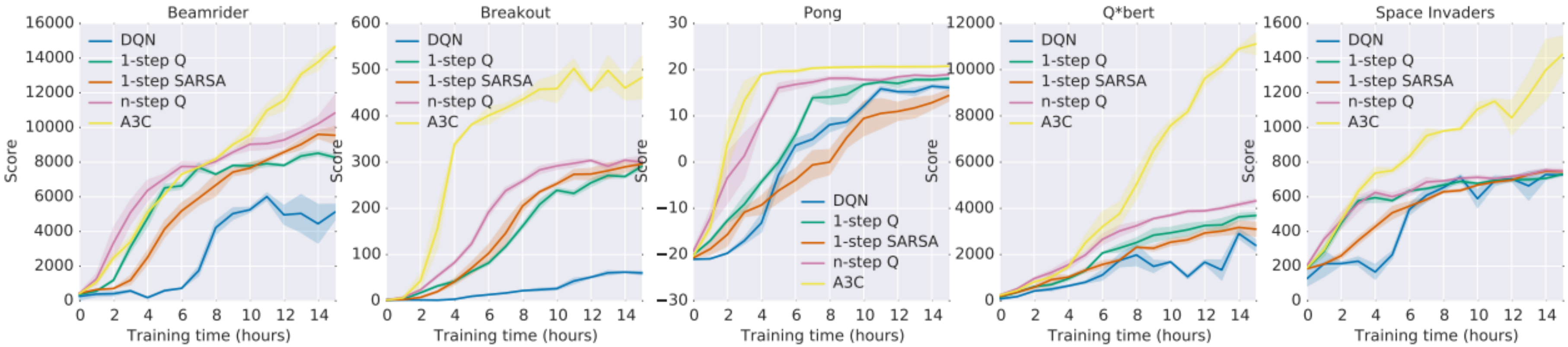


Figure 1. Learning speed comparison for DQN and the new asynchronous algorithms on five Atari 2600 games. DQN was trained on a single Nvidia K40 GPU while the asynchronous methods were trained using 16 CPU cores. The plots are averaged over 5 runs. In the case of DQN the runs were for different seeds with fixed hyperparameters. For asynchronous methods we average over the best 5 models from 50 experiments with learning rates sampled from $LogUniform(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.

A3C Performance

- 57 Atari games

Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorila	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%

Table 1. Mean and median human-normalized scores on 57 Atari games using the human starts evaluation metric. Supplementary Table SS3 shows the raw scores for all games.

DDPG

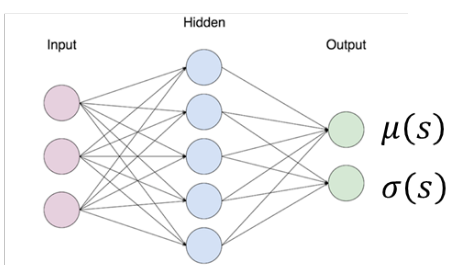
- Deep deterministic policy gradient (DDPG)

  Deterministic policy gradient† + DQN (replay memory)

- Continuous action (categorical action)
- Continuous update (not episodic update)

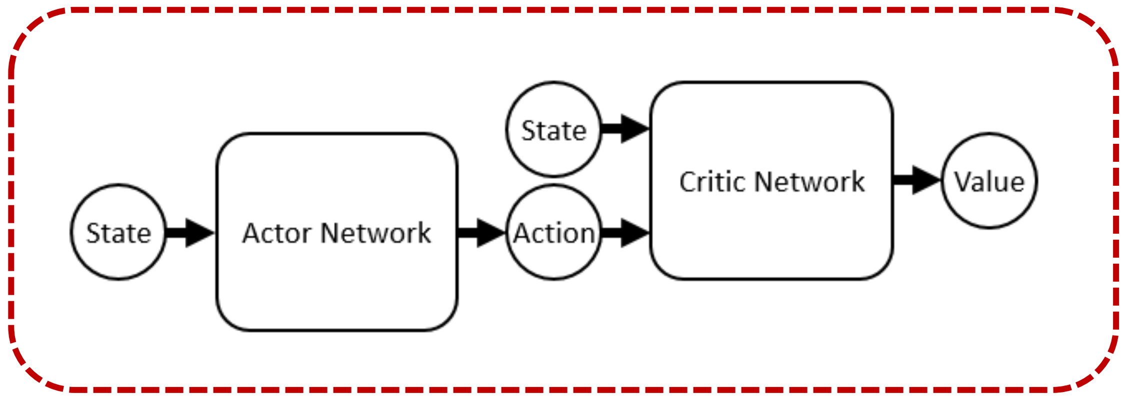
$$\pi(s) = N\big(\mu(s), \sigma(s)\big)$$

$$a \sim N\big(\mu(s), \sigma(s)\big)$$



Gaussian policy

$$\boldsymbol{a} = \boldsymbol{\pi}(\boldsymbol{S})$$



DDPG

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

† Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014, June). Deterministic policy gradient algorithms. In *ICML*.

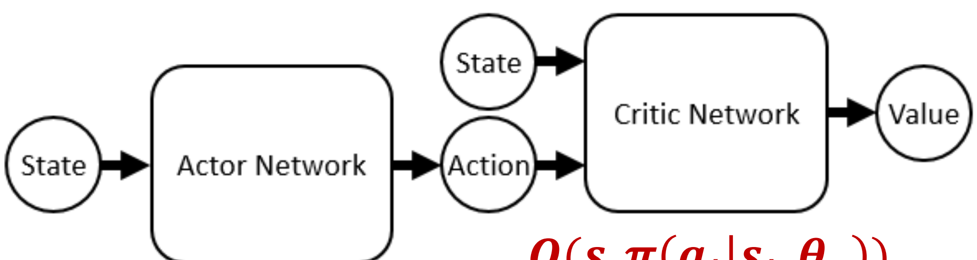
Deterministic Policy Gradient

- REINFORCE

$$\nabla_\theta E_\tau[G] = E_\tau\left[G\nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t|s_t,\theta)\right]$$

- Deterministic policy gradient

$$\nabla_{\theta_a} E_\tau[Q(s,a)] = E_\tau\left[\nabla_{\theta_a} Q(s,\pi(s_t|\theta_a)|\theta_c)\right]$$
$$= \frac{\partial Q}{\partial \pi}\frac{\partial \pi}{\partial \theta_a}$$



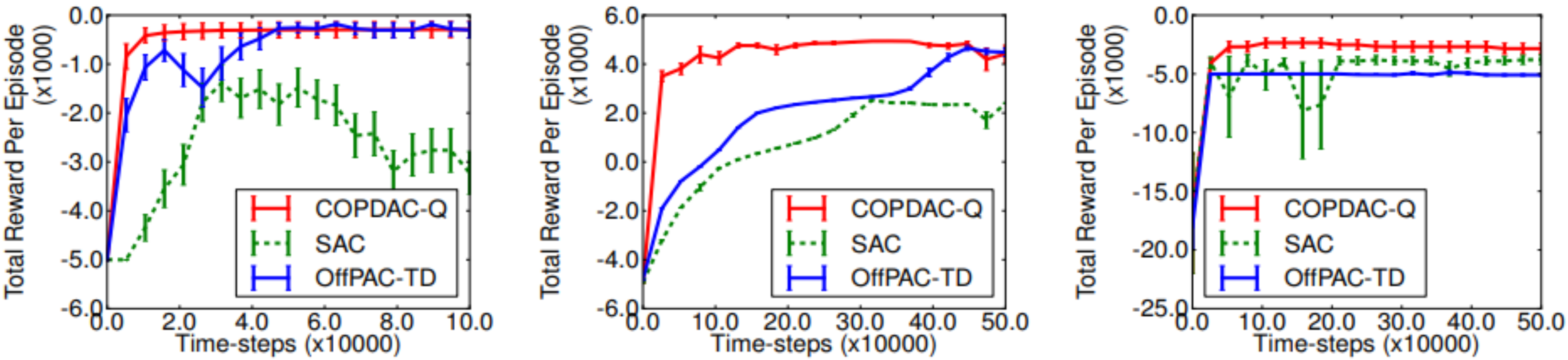
$\boldsymbol{Q(s,\pi(a_t|s_t,\theta_a))}$

$\boldsymbol{\pi(s_t|\theta_a)}$

Deterministic Policy Gradient

- Continuous action environment

Off-policy + deterministic policy > Off-policy + stochastic policy > On-policy + stochastic policy



(a) Mountain Car (b) Pendulum (c) 2D Puddle World

Figure 2. Comparison of stochastic on-policy actor-critic (SAC), stochastic off-policy actor-critic (OffPAC), and deterministic off-policy actor-critic (COPDAC) on continuous-action reinforcement learning. Each point is the average test performance of the mean policy.

DDPG

- Incorporate replay memory and target network ideas from DQN for increased stability

Algorithm 1 DDPG algorithm

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

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

    Update the target networks:

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

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

  end for
end for

DDPG Performance

- Continuous control examples

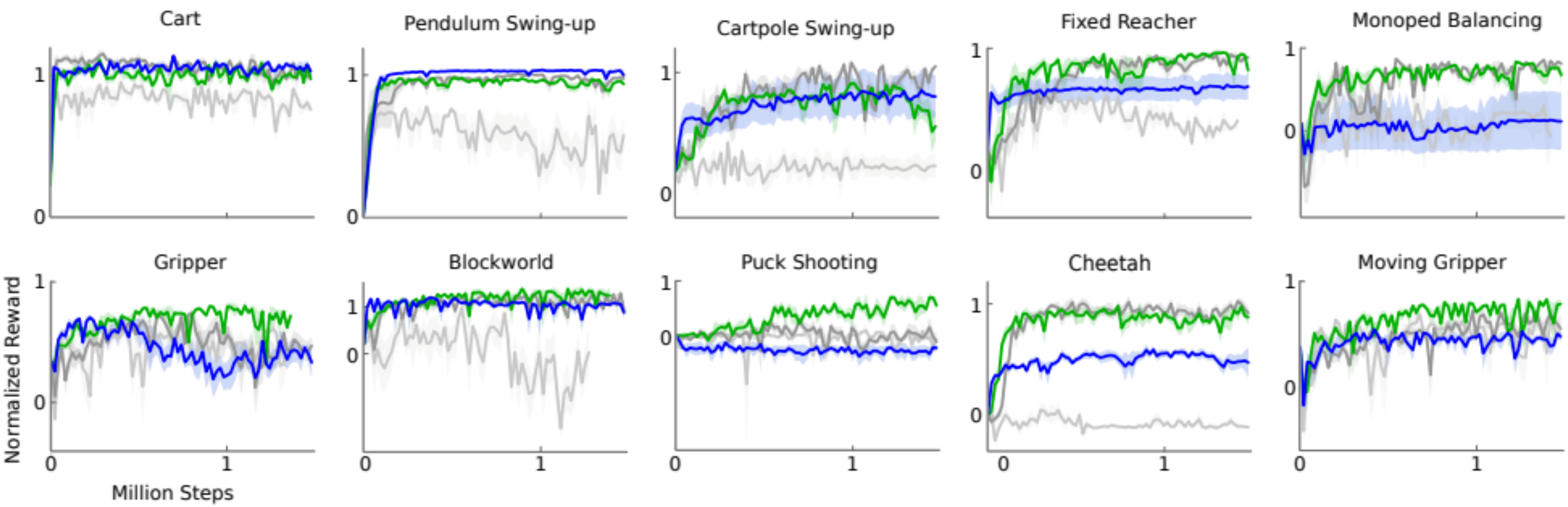


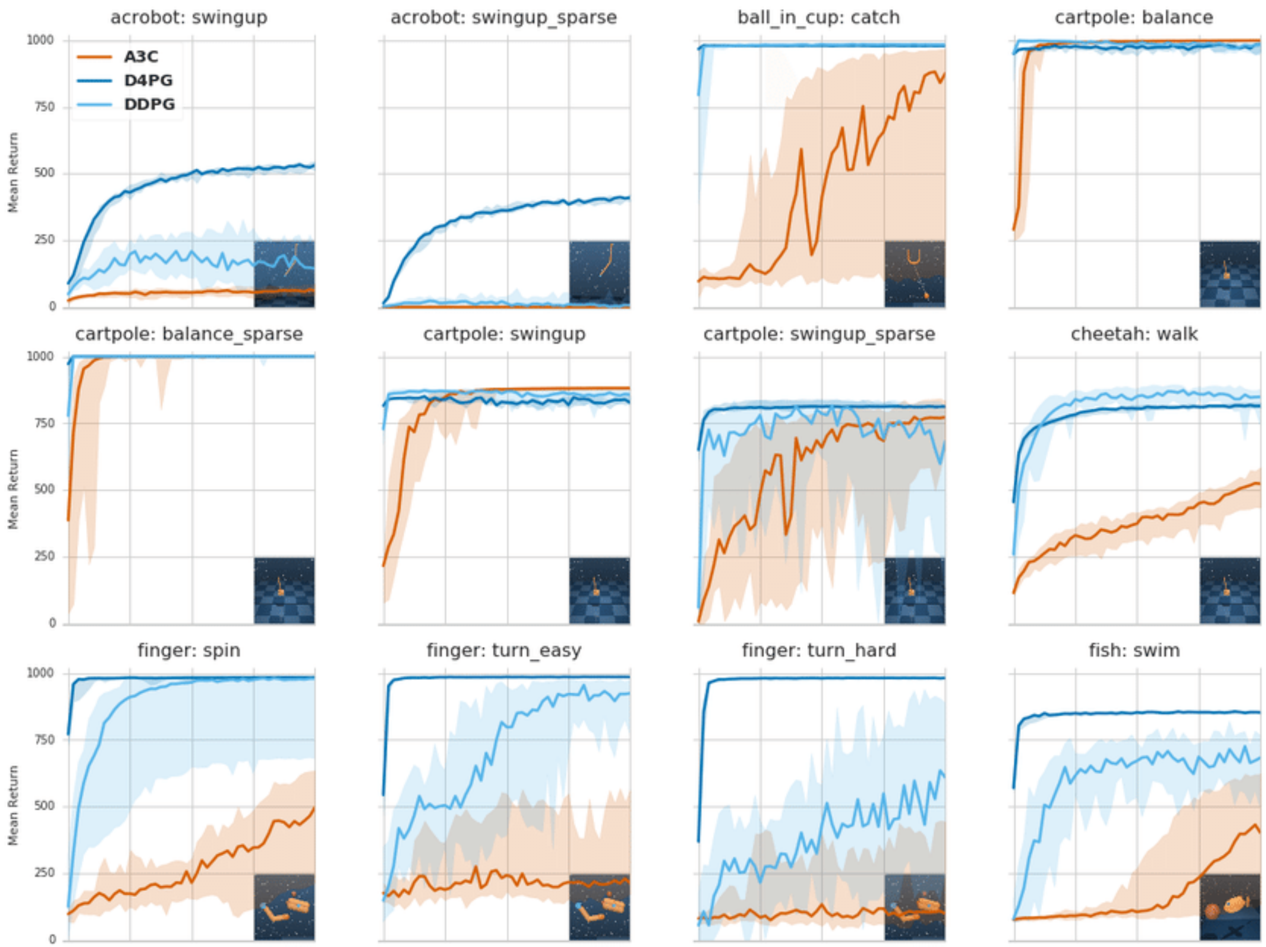
Figure 2: Performance curves for a selection of domains using variants of DPG: original DPG algorithm (minibatch NFQCA) with batch normalization (light grey), with target network (dark grey), with target networks and batch normalization (green), with target networks from pixel-only inputs (blue). Target networks are crucial.

DDPG Performance

- Continuous control examples

DDPG vs A3C

- Continuous control examples



Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. D. L., ... & Lillicrap, T. (2018). DeepMind Control Suite. *arXiv preprint arXiv:1801.00690*.

Contents

Future of RL Algorithms

- Efficient Learning
- Robust Training
- Complex problem
  - Hierarchical RL
  - Multi-agent RL