Policy Based Methods

Deep Reinforcement Learning

Mohammad Hossein Rohban, Ph.D.

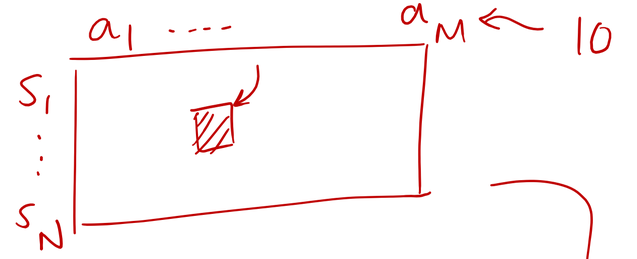
Courtesy: Some slides are adopted from CS 285 Berkeley, and CS 234 Stanford, and Pieter Abbeel's compact series on RL.

RIML Lab, CE Department, Sharif University of Technology

Spring 2026

Function Approximation

Function approximation and deep RL



- The policy, value function, model, and agent state update are all functions
- We want to learn these from experience (data).
- If there are too many states, we need to approximate.
- This is often called deep reinforcement learning, when using neural networks to represent these functions.

function approximation

$\rightsquigarrow Q(s,a)$

$\rightsquigarrow (S, a, S', r)$

→ Q-learning (off-policy)

$Q(s,a) \leftarrow Q(s,a) + \alpha [\text{target} - Q(s,a)]$ (lr)

$\text{target} = r(s,a) + \gamma \max_{a'} Q(s',a')$

SARSA: $a' \sim \pi \rightsquigarrow \epsilon$-greedy Q

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

Large-Scale Reinforcement Learning

- Reinforcement learning can be used to solve large problems, e.g.
  - Backgammon: 10^{20} states
  - Go: 10^{170} states
  - Helicopter: continuous state space
  - Robots: real world
- How can we apply our methods for prediction and control?

Value Function Approximation

Value Function Approximation

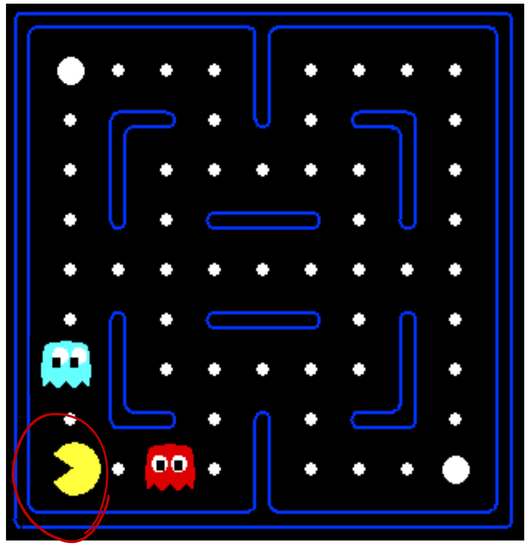
- So far we mostly considered lookup tables
  - Every state s has an entry $v(s)$
  - Or every state-action pair s, a has an entry $q(s, a)$

- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
  - Individual environment states are often not fully observable

Value Function Approximation

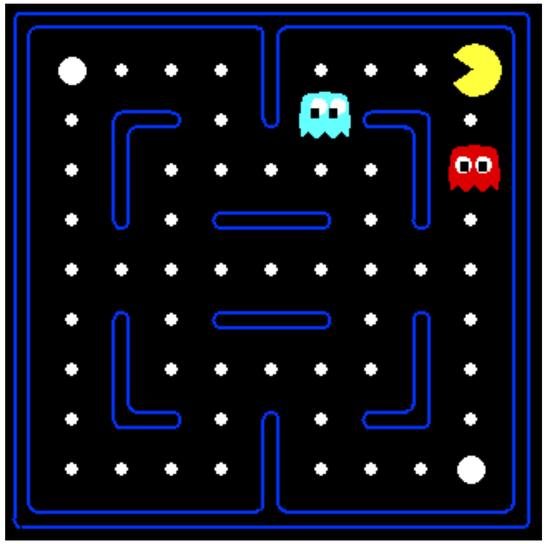
$Q(s_1, a)$ ⇒ $Q(s_2, a)$ — generalize — $Q(s_3, a)$

Let's say we discover through experience that this state is bad:



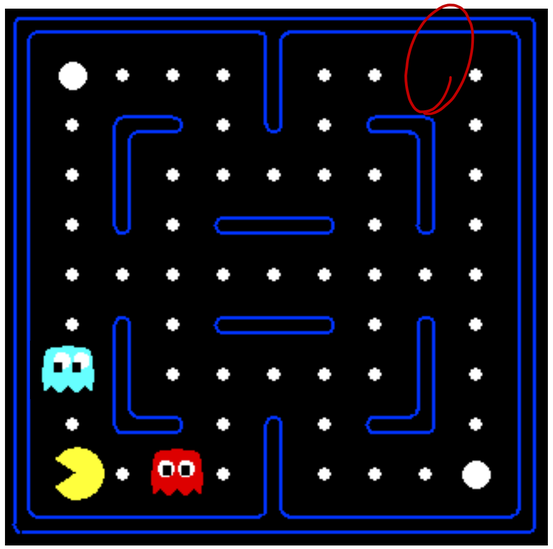
s_1

In naïve Q-learning, we know nothing about this state:



s_2

Or even this one!



s_3

Value Function Approximation

- Solution for large MDPs:
  - Estimate value function with function approximation

$$v_w(s) \approx v_\pi(s) \qquad (or \;\; v_*(s))$$

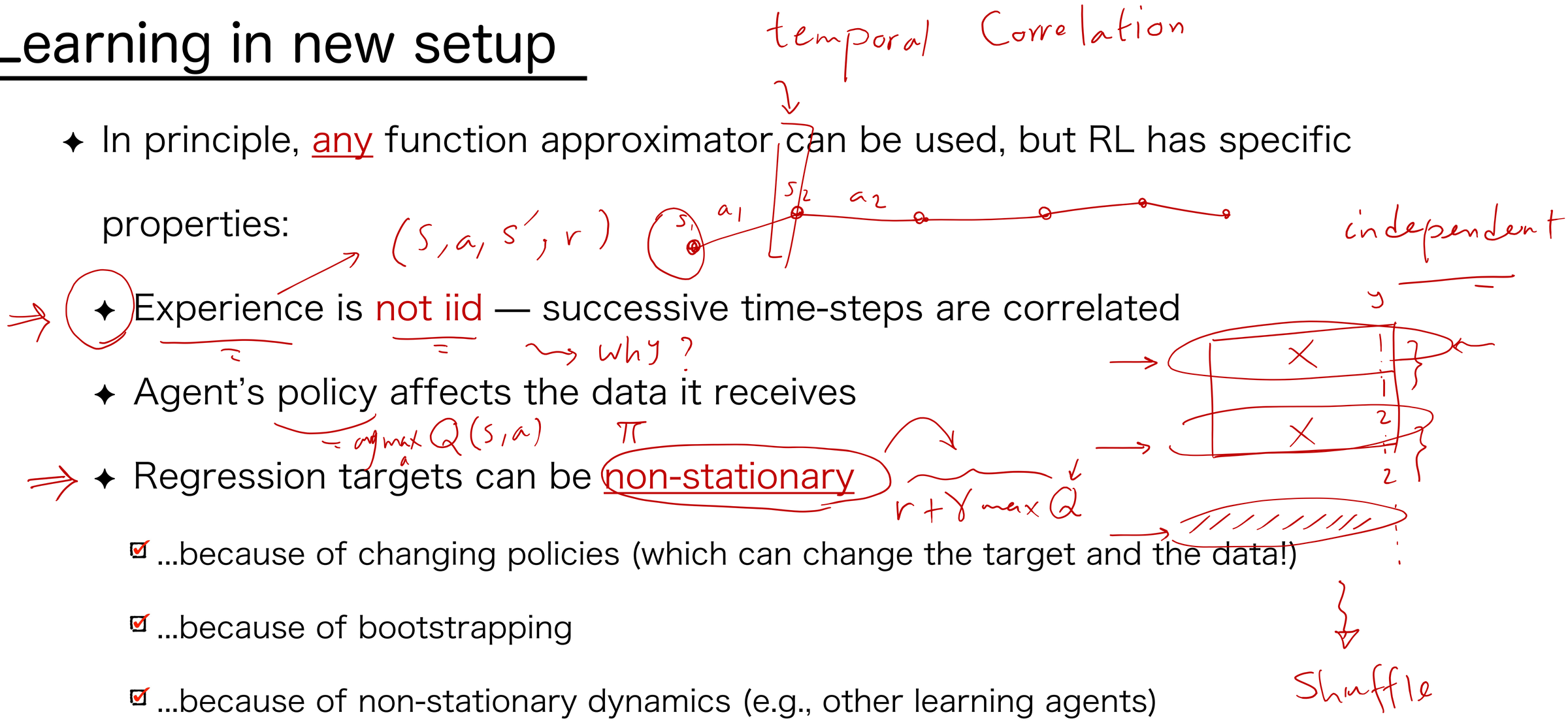
$$q_w(s,a) \approx q_\pi(s,a) \qquad (or \;\; q_*(s,a))$$

  - Update parameter w (e.g., using MC or TD learning)
  - Generalize to unseen states

Key Requirements

Learning in new setup

- In principle, any function approximator can be used, but RL has specific properties:
  - Experience is not iid — successive time-steps are correlated
  - Agent's policy affects the data it receives
  - Regression targets can be non-stationary
    - ☑ ...because of changing policies (which can change the target and the data!)
    - ☑ ...because of bootstrapping
    - ☑ ...because of non-stationary dynamics (e.g., other learning agents)
    - ☑ ...because the world is large (never quite in the same state)

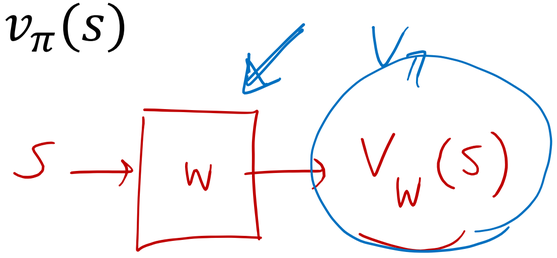


Gradient-based Algorithms

Approximate Values By Stochastic Gradient Descent

- Goal: find w that minimize the difference between $v_w(s)$ and $v_\pi(s)$

$$J(\mathbf{w}) = \mathbb{E}_{S\sim d}\left[\left(v_\pi(S) - v_{\mathbf{w}}(S)\right)^2\right]$$



- Gradient descent:

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w}) = \alpha \mathbb{E}_d\left[\left(v_\pi(S) - v_{\mathbf{w}}(S)\right)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S)\right]$$

- Stochastic gradient descent (SGD), sample the gradient:

$$\Delta \mathbf{w} = \alpha\left(G_t - v_{\mathbf{w}}(S_t)\right)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

- Note: Monte Carlo return G_t is a sample for $v_\pi(s_t)$

Training Loss

- We can't update towards the true value function $v_\pi(s)$
- We substitute a target for $v_\pi(s)$
  - For MC, the target is the return G_t

$$\Delta \mathbf{w}_t = \alpha(G_t - v_{\mathbf{w}}(s))\nabla_{\mathbf{w}} v_{\mathbf{w}}(s)$$

  - For TD, the target is the TD target $R_{t+1} + \gamma v_w(S_{t+1})$

$$\Delta \mathbf{w}_t = \alpha(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

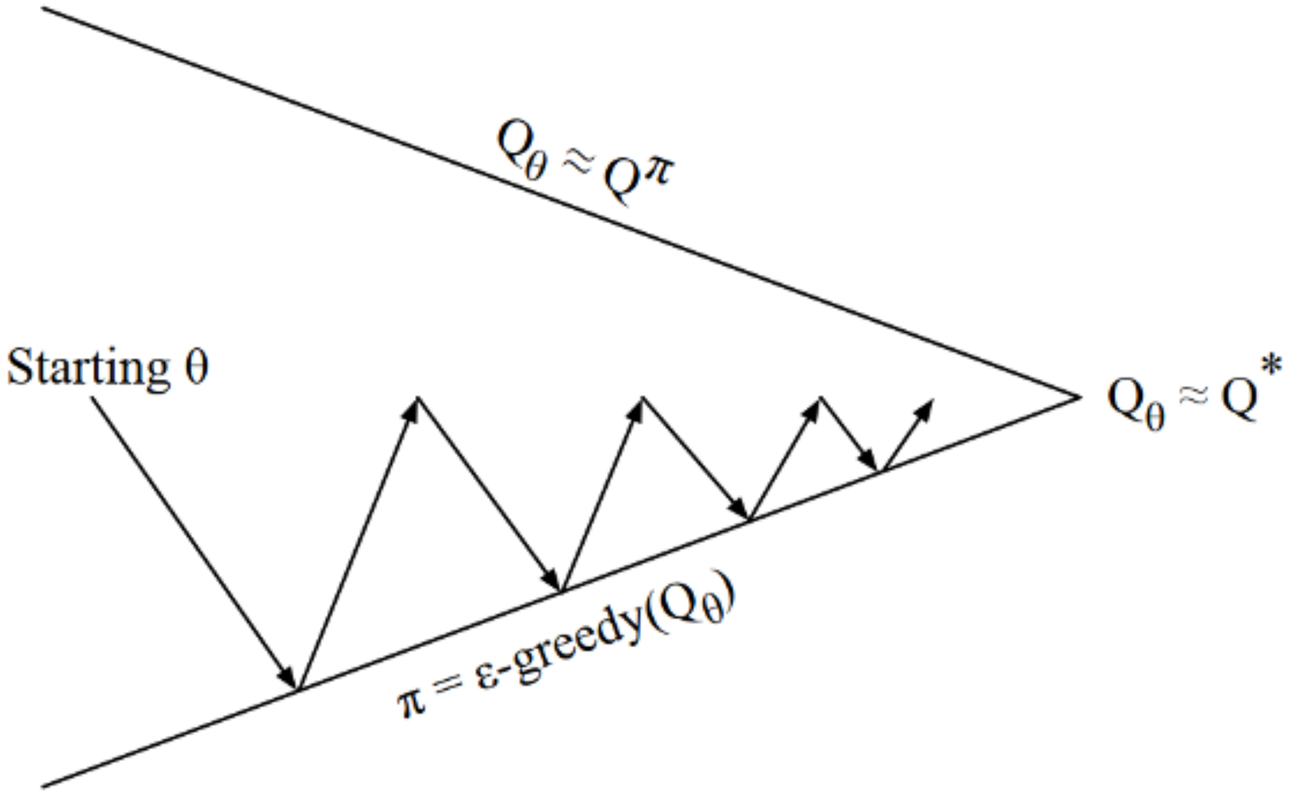
Monte-Carlo with Value Function Approximation

- The return G_t is an unbiased sample of $v_\pi(s)$
- Can therefore apply "supervised learning" to (online) "training data":

$$\{(S_0, G_0), \ldots, (S_t, G_t)\}$$ dataset

Control with value-function approximation

Control with Value Function Approximation



Policy Evaluation Approximate policy Evaluation, $q_w \approx q_\pi$

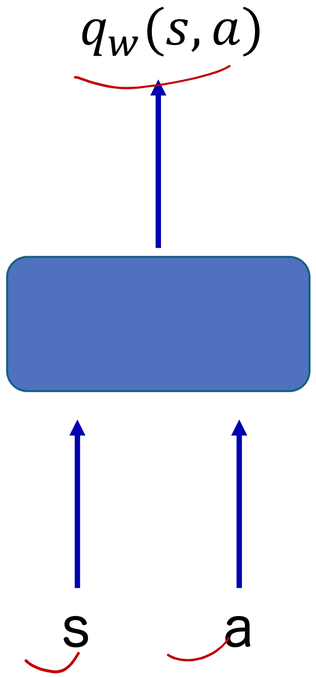
Policy improvement E.g. $\epsilon - greedy$ policy improvement

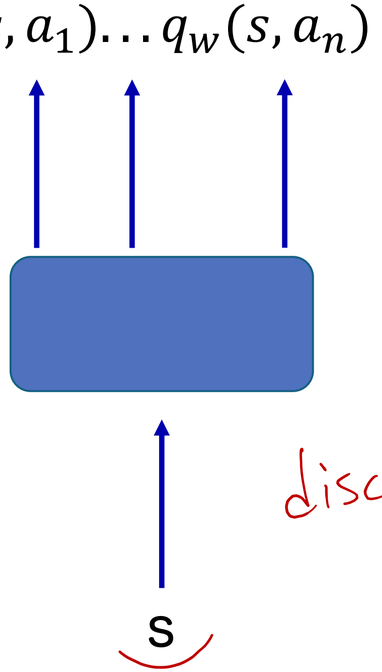
Action-Value Function Approximation

Q-learning

Q^*

- Approximate the action-value function $q_w(s,a) \approx q_\pi(s,a)$





action is continuous

discrete & limited # of actions

Action-Value Function Approximation

- Should we use action-in, or action-out?
  - Action in: $q_w(s,a) = w^T x(s,a)$
  - Action out: $q_w(s) = Wx(s)$ such that $q_w(s,a) = q_w(s)[a]$

- One incorporates a in feature learning, the other uses the same features for all a's
- Unclear which is better in general
- If we want to use continuous actions, action-in is easier
- For (small) discrete action spaces, action-out is common (e.g., DQN)

Deep Reinforcement Learning

Recall: Model free control

- Similar to policy evaluation, true state-action value function for a state is unknown and so substitute a target value
- In Monte Carlo methods, use a return G_t as a substitute target

$$\Delta \boldsymbol{w} = \alpha \left(G_t - \hat{Q}(s_t, a_t; \boldsymbol{w}) \right) \nabla_{\boldsymbol{w}} \hat{Q}(s_t, a_t; \boldsymbol{w})$$

- For SARSA instead use a TD target

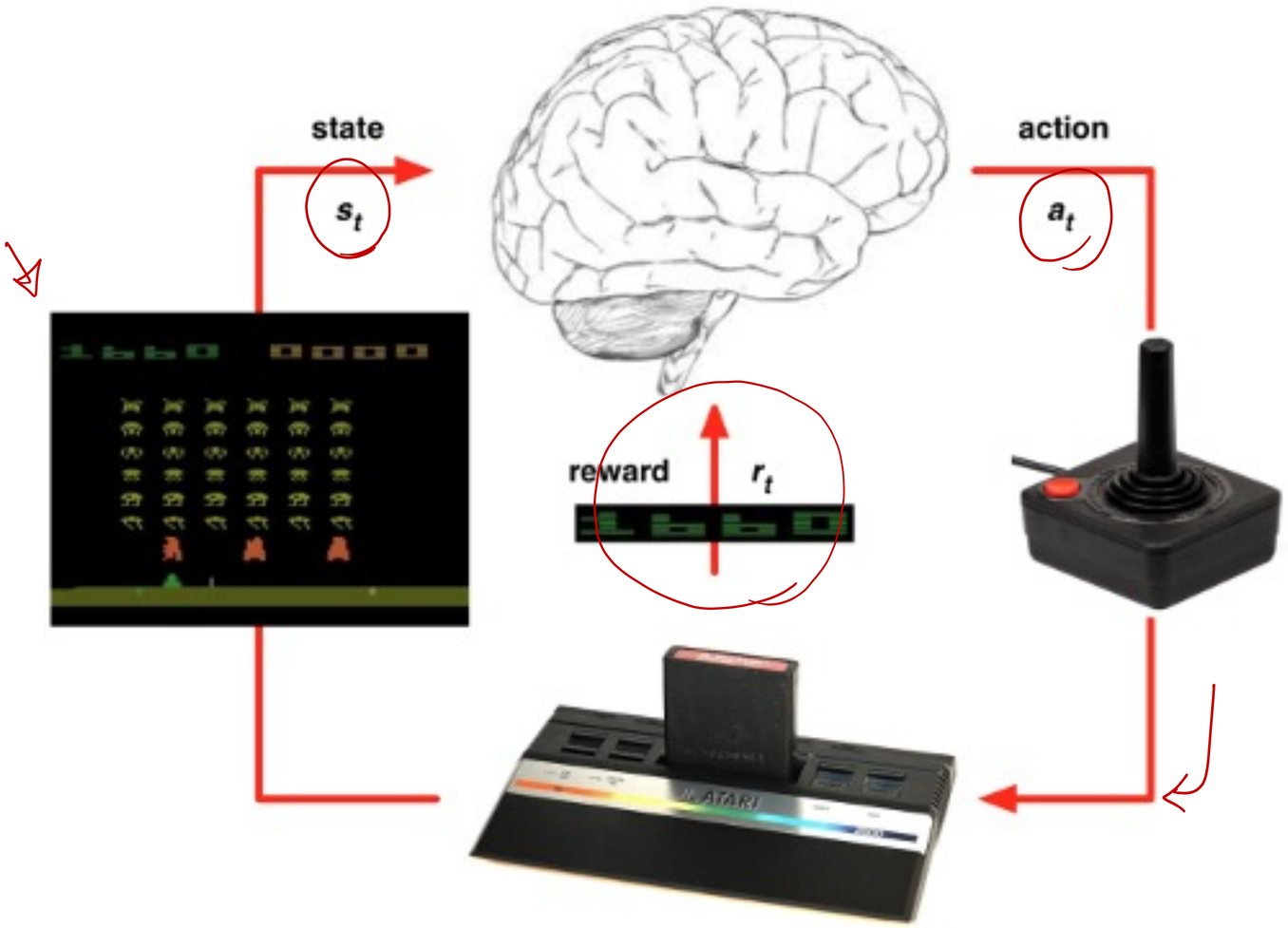
$$\Delta \boldsymbol{w} = \alpha \left(r + \gamma \hat{Q}(s_{t+1}, a_{t+1}; \boldsymbol{w}) - \hat{Q}(s_t, a_t; \boldsymbol{w}) \right) \nabla_{\boldsymbol{w}} \hat{Q}(s_t, a_t; \boldsymbol{w})$$

- For Q-learning

$$\Delta \boldsymbol{w} = \alpha \left(r + \gamma \max_a \hat{Q}(s_{t+1}, a; \boldsymbol{w}) - \hat{Q}(s_t, a_t; \boldsymbol{w}) \right) \nabla_{\boldsymbol{w}} \hat{Q}(s_t, a_t; \boldsymbol{w})$$

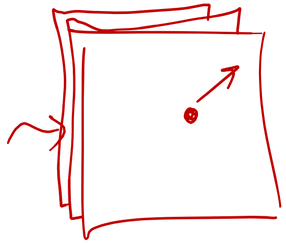
DQN

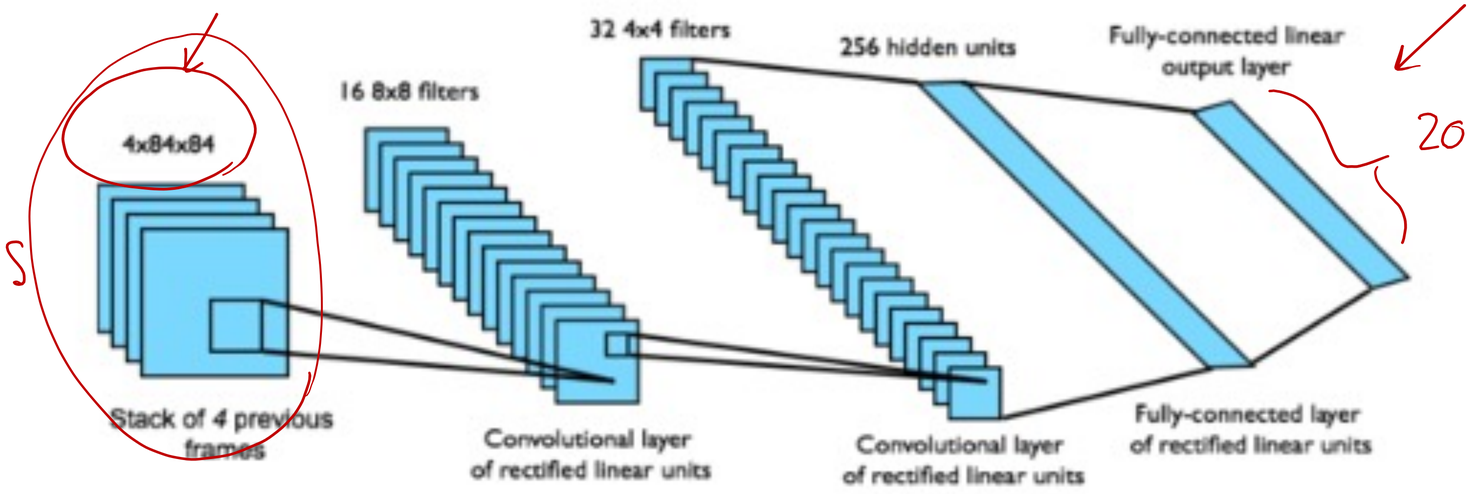
Doing deep RL in Atari



DQNs in Atari

- End-to-end learning of values Q(s, a) from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is Q(s, a) for 18 joystick/button positions
- Reward is change in score for that step
- Network architecture and hyperparameters fixed across all games



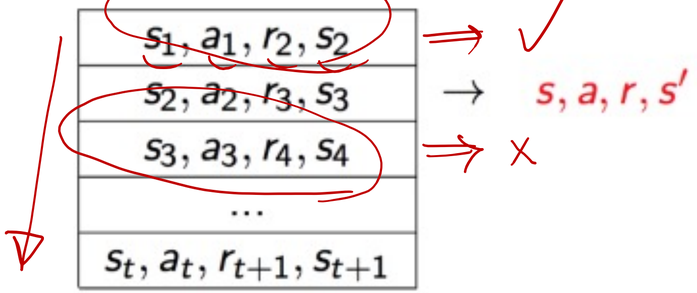


DQNs in Atari

- Q-learning converges to the optimal $Q^*(s,a)$ using table lookup representation.
- But Q-learning with Value Function Approximation can diverge
- Two of the issues causing problems:
  - Correlations between samples (non-iid training)
  - Non-stationary targets
- Deep Q-learning (DQN) addresses these challenges by
  - Experience replay
  - Fixed Q-targets

DQNs: Experience Replay

- To help remove correlations, store dataset (called a replay buffer) D from prior experience



Replay Buffer

FIFO

priority queue

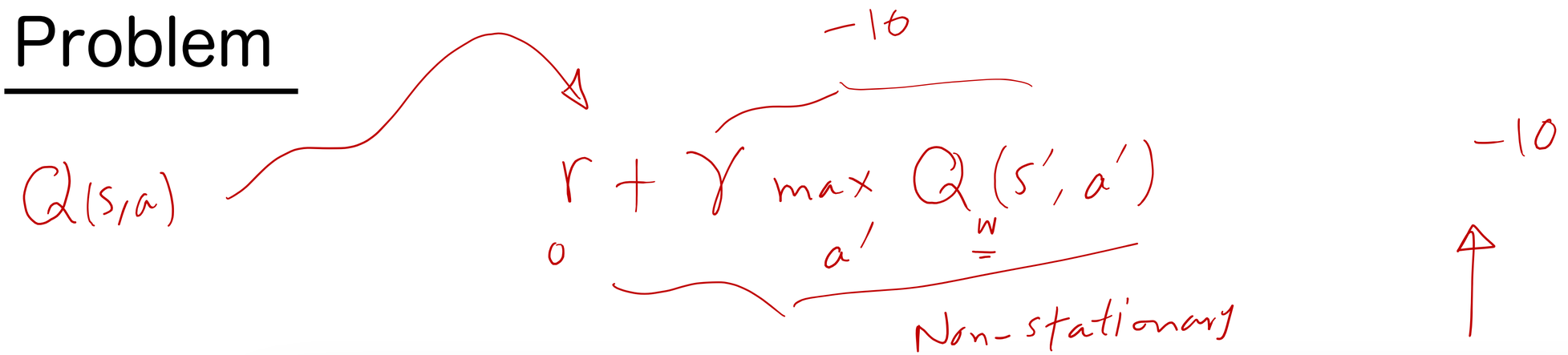
- To perform experience replay, repeat the following:
  - $(s, a, r, s') \sim D$: sample an experience tuple from the dataset
  - Compute the target value for the sampled s: $r + \gamma \max_{a'} \hat{Q}(s', a'; w)$
  - Use stochastic gradient descent to update the network weights

Temporal Diff. Error

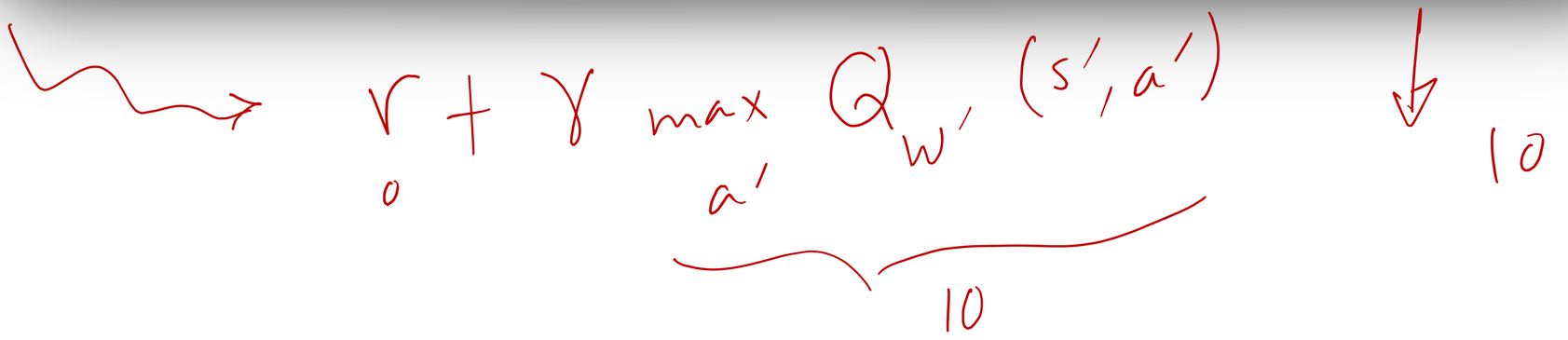
TD error

$$\Delta \boldsymbol{w} = \alpha \left(r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}) - \hat{Q}(s, a; \boldsymbol{w}) \right) \nabla_{\boldsymbol{w}} \hat{Q}(s, a; \boldsymbol{w})$$

Problem



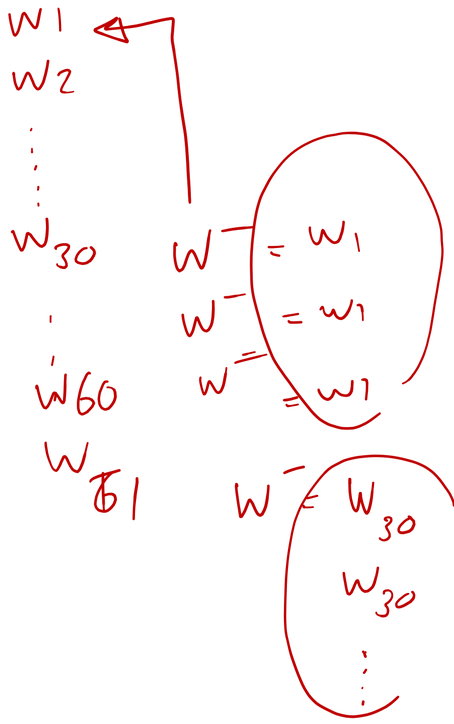
Can treat the target as a constant scalar, but the weights will get updated on the next round, changing the target value



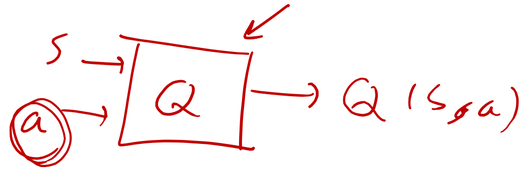
DQNs: Fixed Q-Targets

- To help improve stability, fix the target weights used in the target calculation for multiple updates
- Target network uses a different set of weights than the weights being updated
- Let parameters w^- be the set of weights used in the target, and w be the weights that are being updated
- Slight change to computation of target value:
  - $(s, a, r, s') \sim D$: sample an experience tuple from the dataset
  - Compute the target value for the sampled s: $r + \gamma \max_{a'} \hat{Q}(s', a'; w^-)$
  - Use stochastic gradient descent to update the network weights

$$\Delta \boldsymbol{w} = \alpha \left(r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}^-) - \hat{Q}(s, a; \boldsymbol{w}) \right) \nabla_{\boldsymbol{w}} \hat{Q}(s, a; \boldsymbol{w})$$



DQN Algorithm

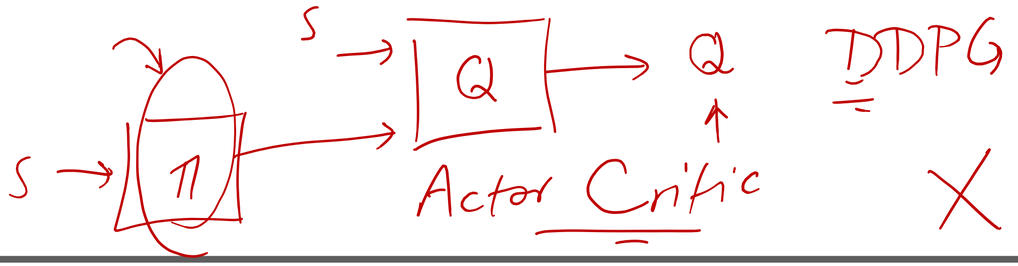


$\max_a Q(s,a)$

Algorithm:

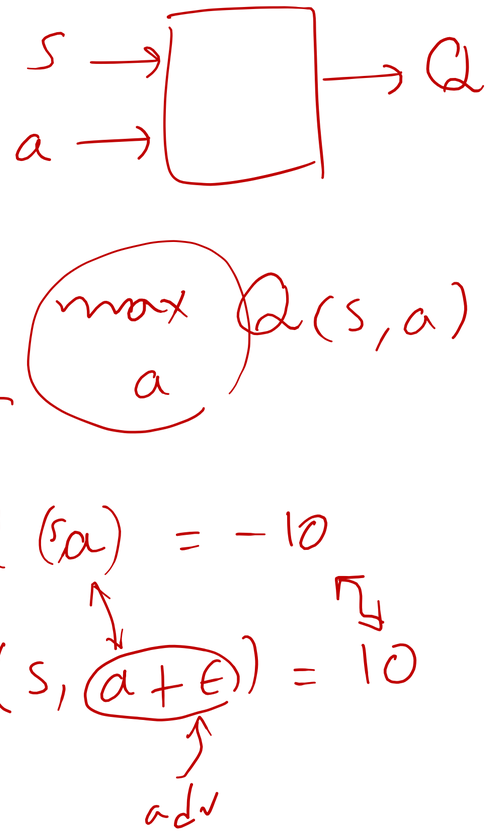
Input C, α, $D = \{\}$, Initialize $\boldsymbol{w}$, $\boldsymbol{w}^- = \boldsymbol{w}$, $t = 0$ Get initial state s_0
loop
  Sample action a_t given $\epsilon - greedy$ policy for current $\hat{Q}(s_t, a; \boldsymbol{w})$
  Observe reward r_t and next state s_{t+1}
  Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
  Sample random minibatch of tuples (s_i, a_i, r_i, s_{i+1}) from D
  for j in minibatch do
    if episode terminated at step $i+1$ then
      $y_i = r_i$
    else
      $y_i = r_i + \gamma max_{a'} \hat{Q}(s_{i+1}, a'; \boldsymbol{w}^-)$
    end if
    Do gradient descent step on $(y_i - \hat{Q}(s_i, a_i; \boldsymbol{w}))^2$ for parameters $\boldsymbol{w}$: $\Delta \boldsymbol{w} = \alpha \left(y_i - \hat{Q}(s_i, a_i; \boldsymbol{w}) \right) \nabla_{\boldsymbol{w}} \hat{Q}(s_i, a_i; \boldsymbol{w})$
  end for
  $t = t + 1$
  if $mod(t, C) == 0$ then
    $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$
  end if
end loop

Tuples that are used to update w, may be selected in future again.

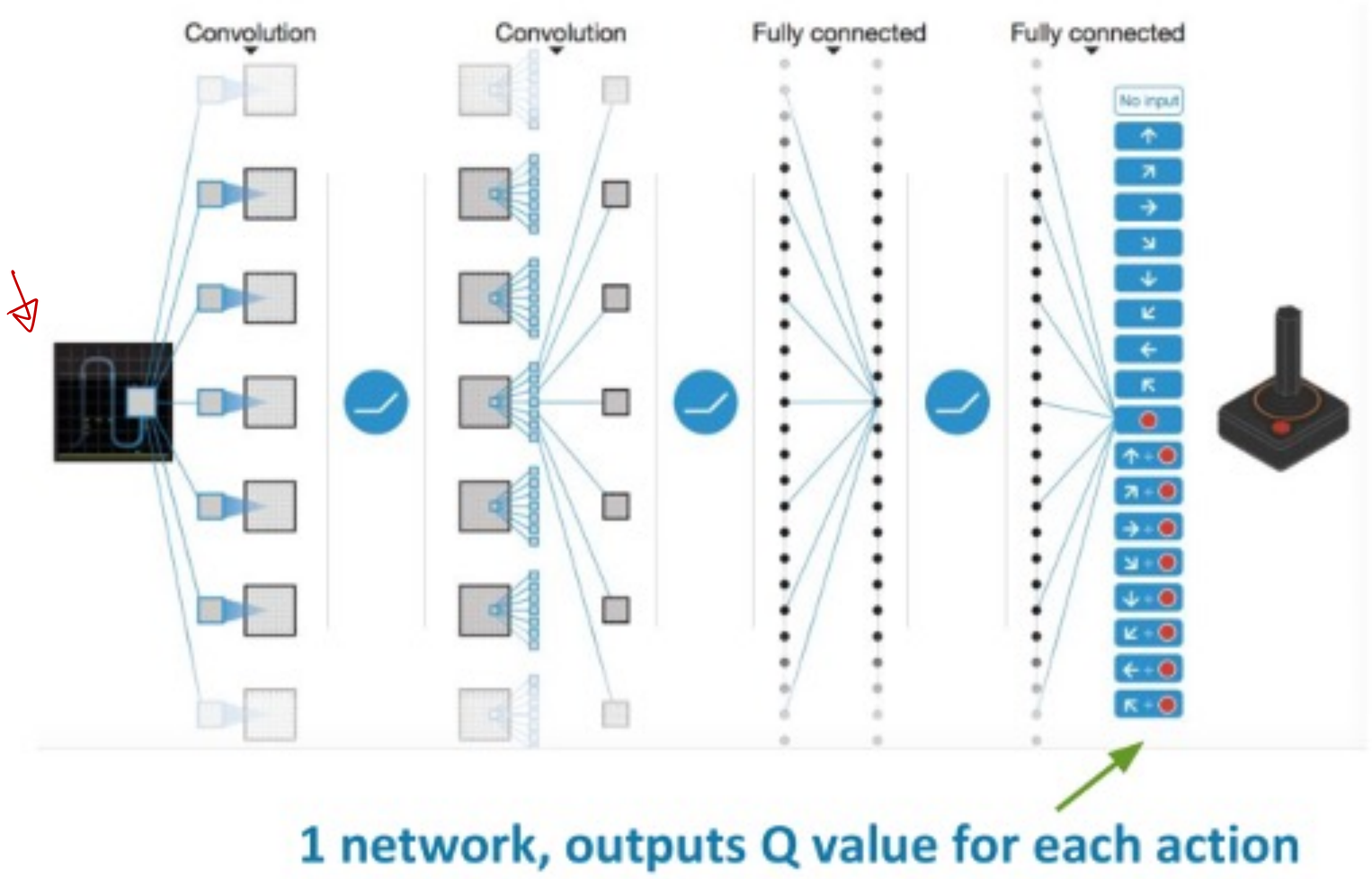


DQN Summary

- DQN uses experience replay and fixed Q-targets
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D
- Sample random mini-batch of transitions (s, a, r, s') from D
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimizes MSE between Q-network and Q-learning targets
- Uses stochastic gradient descent



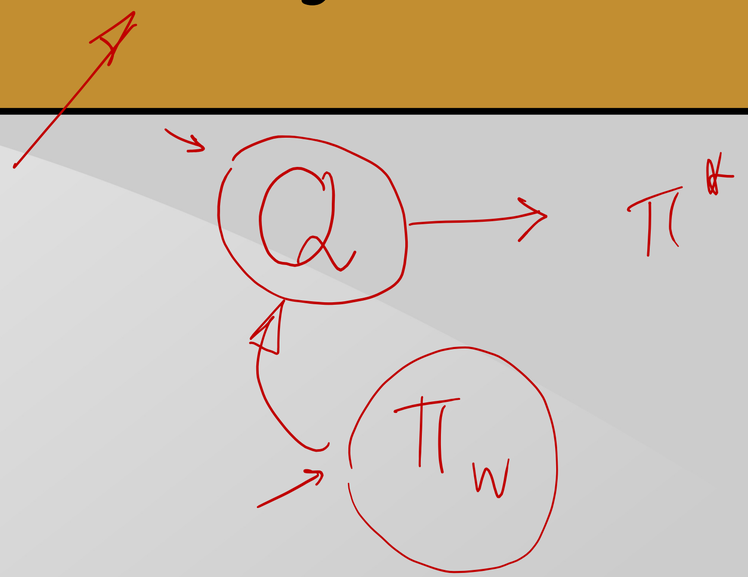
DQN


Convolution
Convolution
Fully connected
Fully connected
No input
1 network, outputs Q value for each action

Results

Game	Linear	Deep Network	DQN w/fixed Q	DQN w/replay	DQN w/replay and fixed Q
Breakout	3	3	10	241	317
Enduro	62	29	141	831	1006
River Raid	2345	1453	2868	4102	7447
Seaquest	656	275	1003	823	2894
Space Invaders	301	302	373	826	1089

Policy Gradient



The Goal of Reinforcement Learning

Obj. function

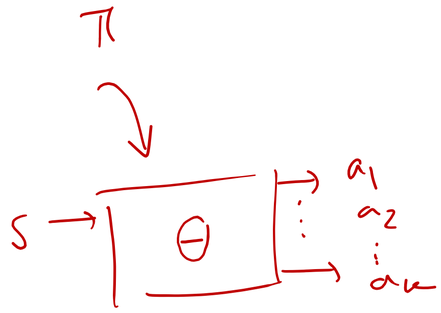
return

$$J(\theta) = E[\Sigma_t r(s_t, a_t)]$$

$$\rightarrow s_1 \sim p(s_1)$$

$$\rightarrow a_t \sim \pi(.|s_t) \rightsquigarrow \pi_\theta$$

$$\rightarrow s_{t+1} \sim p(.|s_t, a_t)$$

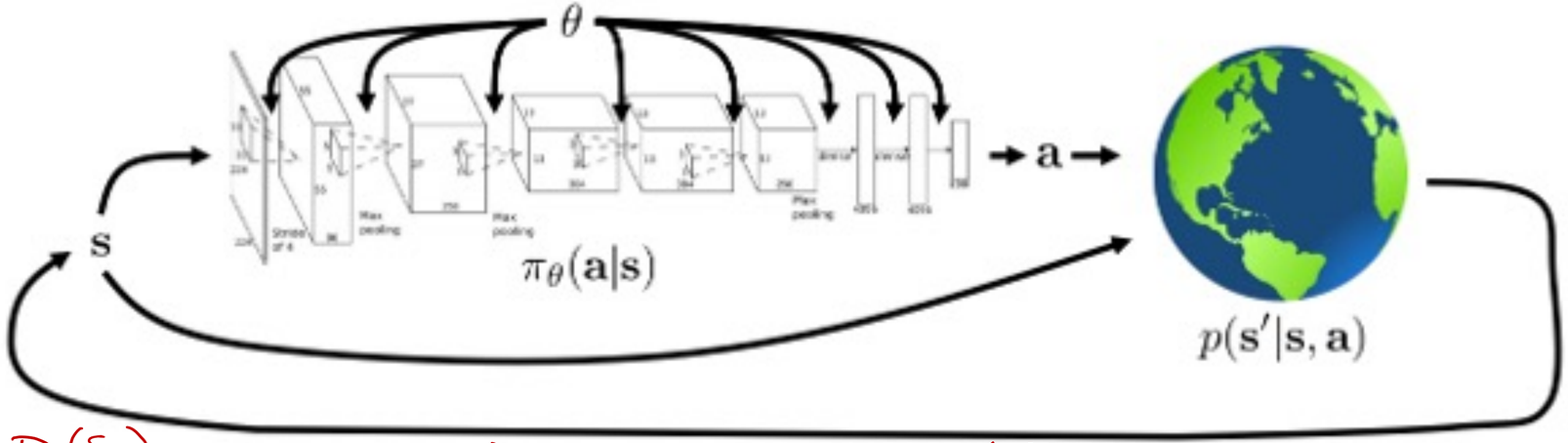


$\theta \leftarrow \theta + \alpha \nabla_\theta J$ gradient ascent

Challenge $\mathbb{E}$ cannot be calculated!

Trajectory Probability

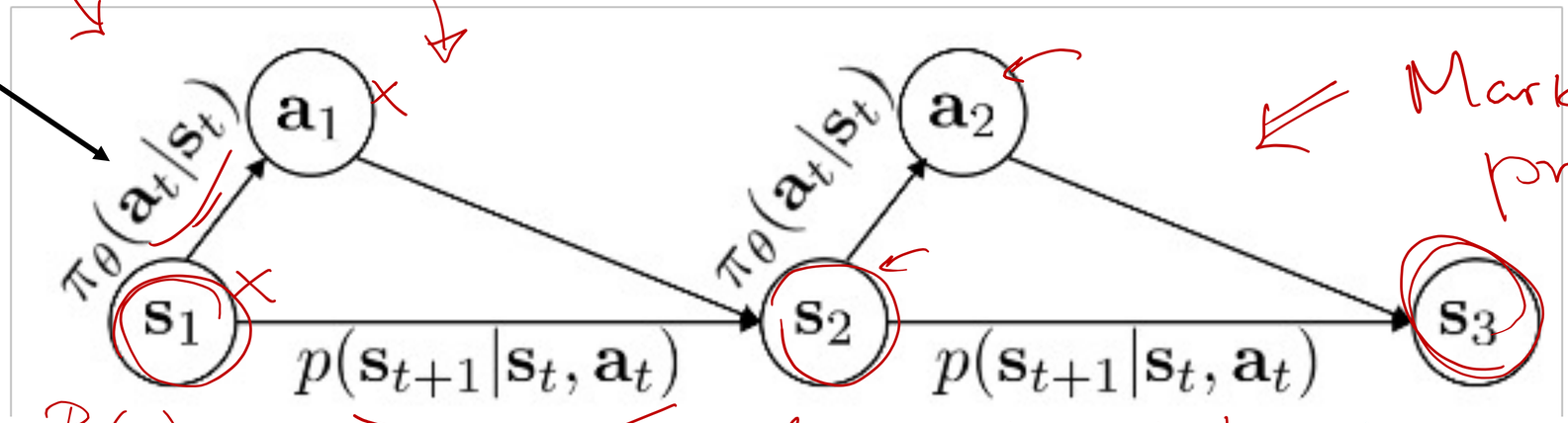
$\nabla_\theta \log P_\theta(\tau)$



$P_\theta(\tau) = P(s_1) \cdot \pi(a_1|s_1) \cdot P(s_2|s_1,a_1) \cdot \pi(a_2|s_2) \cdot P(s_3|s_2,a_2) \cdots$

stochastic policy

log



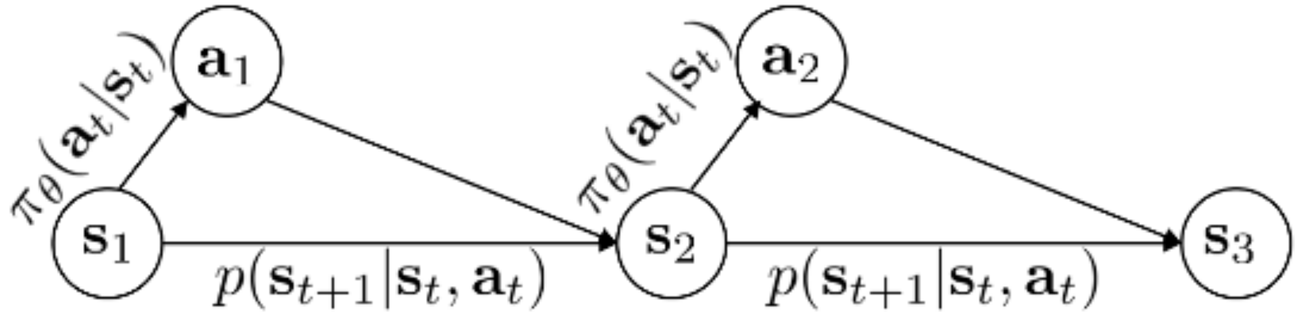
Markov prop.

∇_θ ($\log P(\tau) = \log P(s_1) + \log \pi_\theta(a_1|s_1) + \log P(s_2|s_1,a_1) + \cdots$

$= 0 + \nabla \log \pi_\theta(a_1|s_1) + 0 + \cdots = \sum_{i=1}^{n} \nabla_\theta \log \pi(a_i|s_i)$

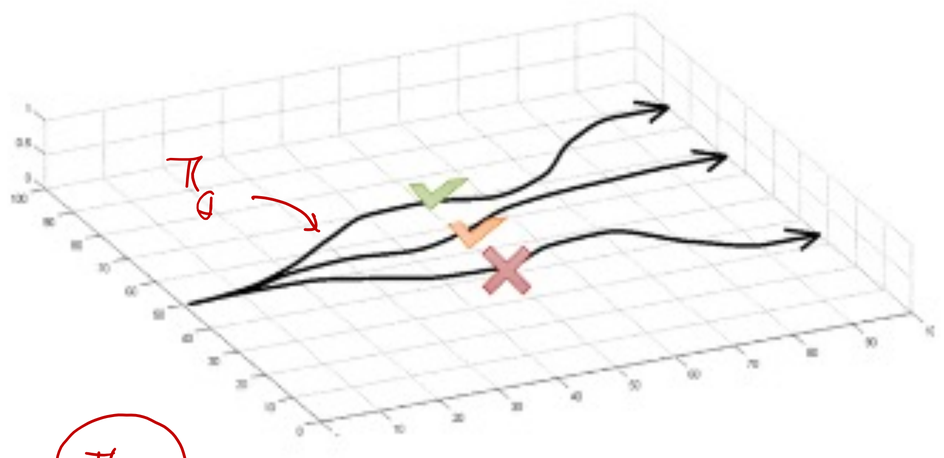
Trajectory Probability

$$\underbrace{p_\theta(\mathbf{s_1}, \mathbf{a_1}, \ldots, \mathbf{s_T}, \mathbf{a_T})}_{p_\theta(\tau)} = p(\mathbf{s_1}) \prod_{t=1}^{T} \pi_\theta(\mathbf{a_t}|\mathbf{s_t}) p(\mathbf{s_{t+1}}|\mathbf{s_t}, \mathbf{a_t})$$



Evaluating the Objective

$$\theta^\star = \arg\max_\theta \underbrace{E_{\tau\sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s_t}, \mathbf{a_t})\right]}_{J(\theta)}$$



rep. trick ← VAE

Monte Carlo estimation for objective function:

$$J(\theta) = E_{\tau\sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s_t}, \mathbf{a_t})\right] \approx \frac{1}{N}\sum_i \sum_t r(\mathbf{s_{i,t}}, \mathbf{a_{i,t}})$$

π_θ

* this doesn't work. Why?

* So how to go about this?

$\nabla_\theta J$ ....

Some over samples from π_θ

Direct Policy Differentiation

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} \left[\underbrace{r(\tau)}_{\sum_{t=1}^{T} r(\mathbf{s_t}, \mathbf{a_t})} \right] = \int p_\theta(\tau) r(\tau) d\tau$$

return

density func. for the traj

a convenient identity

$$p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \nabla_\theta p_\theta(\tau)$$

$$= \arg\max_\theta \underbrace{E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\mathbf{s_t}, \mathbf{a_t}) \right]}_{J(\theta)}$$

$$\Rightarrow \int g(\tau) \cdot p_\theta(\tau) d\tau = \mathbb{E}(g(\tau)) \approx \sum_{i=1}^{N} g(\tau_i)$$

it doesn't have the form of an $\mathbb{E}$ through Sampling

$\Rightarrow$ Couldn't be estimated

estimate

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau = \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) r(\tau) d\tau = E_{\tau \sim p_\theta(\tau)} [\nabla_\theta \log p_\theta(\tau) r(\tau)]$$

$g(\tau)$

Q-Learning for Off-Policy Control

$$\theta^\star = \arg\max_\theta J(\theta)$$

$$J(\theta) = E_{\tau\sim p_\theta(\tau)}[r(\tau)]$$

$$\nabla_\theta J(\theta) = E_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) r(\tau)]$$

Log of both sides

$$\text{Log}\, p_\theta(\tau) = \log p(\mathbf{s}_1) + \sum_{t=1}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \log p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$$

$$\underbrace{p_\theta(\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T)}_{p_\theta(\tau)} = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$$

$$\nabla_\theta \left[\log p(\mathbf{s}_1) + \sum_{t=1}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \log p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \right]$$

$$\nabla_\theta J(\theta) = E_{\tau\sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \right) \left(\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t) \right) \right]$$

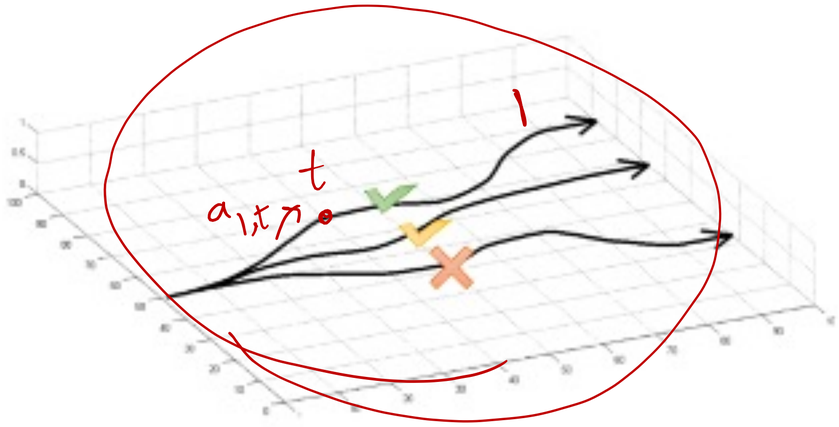
Evaluating the Policy Gradient

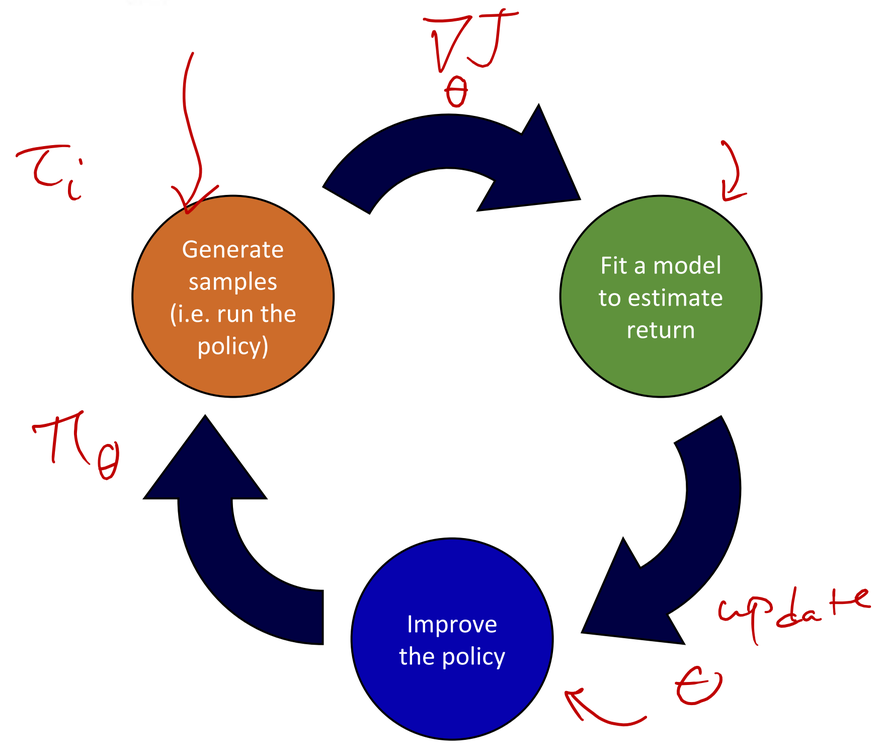
$$\text{recall: } J(\theta) = E_{\tau \sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s_t}, \mathbf{a_t})\right] \approx \frac{1}{N}\sum_i \sum_t r(\mathbf{s_{i,t}}, \mathbf{a_{i,t}})$$

$$\nabla_\theta J(\theta) = E_{\tau \sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a_t}|\mathbf{s_t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s_t}, \mathbf{a_t})\right)\right]$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a_{i,t}}|\mathbf{s_{i,t}})\right)\left(\sum_{t=1}^{T} r(\mathbf{s_{i,t}}, \mathbf{a_{i,t}})\right)$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$





REINFORCE Algorithm

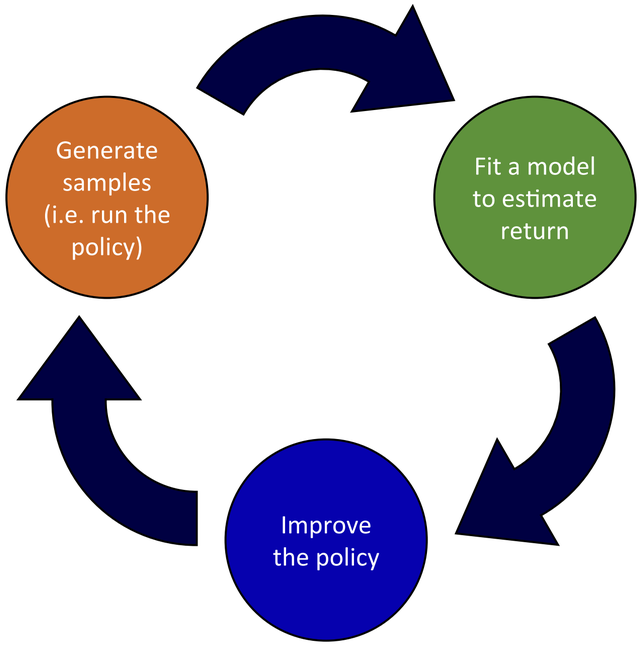
Algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)

2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$

3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Actor Critic



Imitation Learning: Behavioral Cloning

Directly learns a policy by using supervised learning on observation-action pairs from expert demonstrations.

maximum likelihood: $\nabla_\theta J_{\mathrm{ML}}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(\mathbf{a}_{\mathbf{i,t}}|\mathbf{s}_{\mathbf{i,t}}\right)\right)$

Policy Gradient Vs Maximum Likelihood

policy gradient: $$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(\mathbf{a_{i,t}}|\mathbf{s_{i,t}}\right)\right)\left(\sum_{t=1}^{T} r\left(\mathbf{s_{i,t}}, \mathbf{a_{i,t}}\right)\right)$$

maximum likelihood: $$\nabla_\theta J_{\mathrm{ML}}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(\mathbf{a_{i,t}}|\mathbf{s_{i,t}}\right)\right)$$