

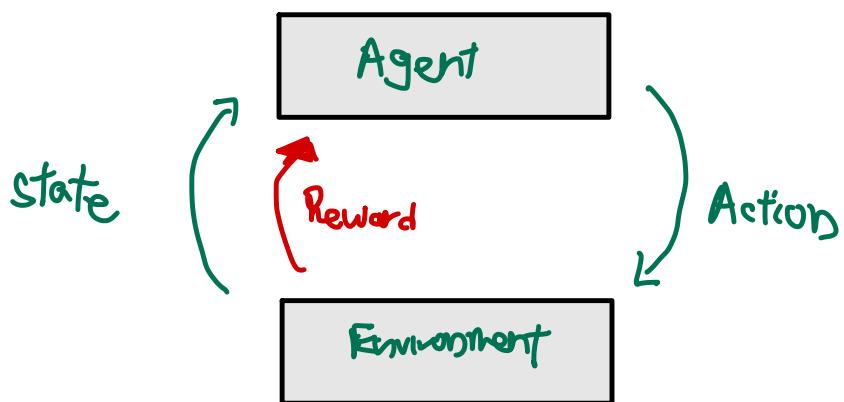


Chapter 1 :

Terminology in  
Markov Decision Process

Agent : Something that can be directly controlled

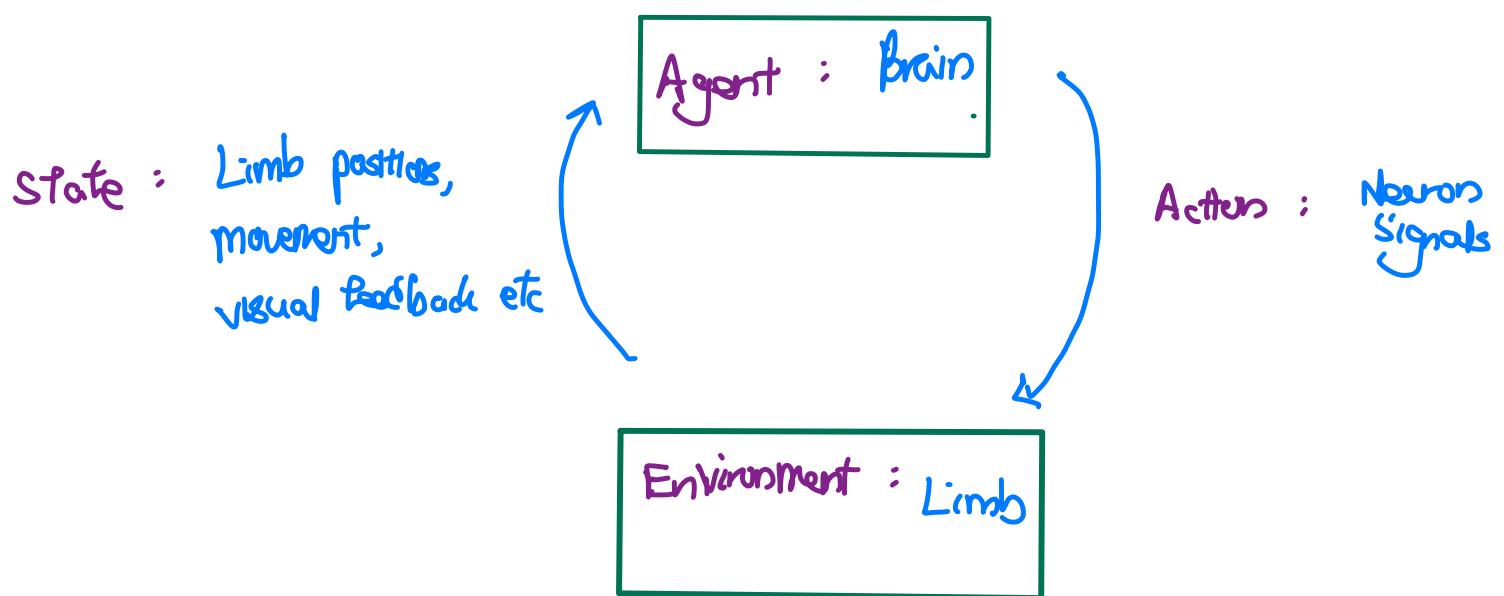
Environment : Something that cannot be controlled,  
but can be interacted with through agent



How to determine agent, environment, action & state  
can be quite arbitrary

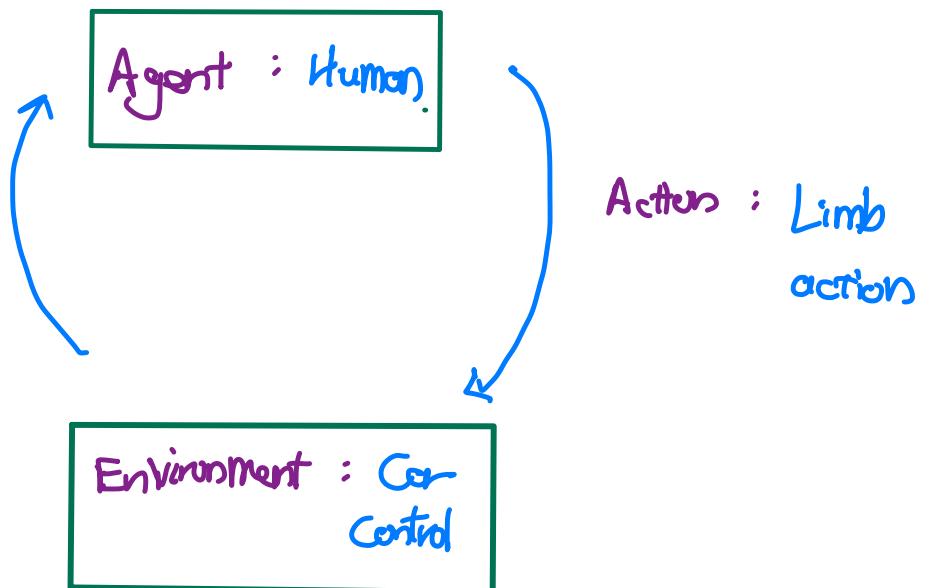
For instance, in achieving a task of controlling a  
human to drive a car, it can be separated into  
a few programs

Program A : Control limb movement



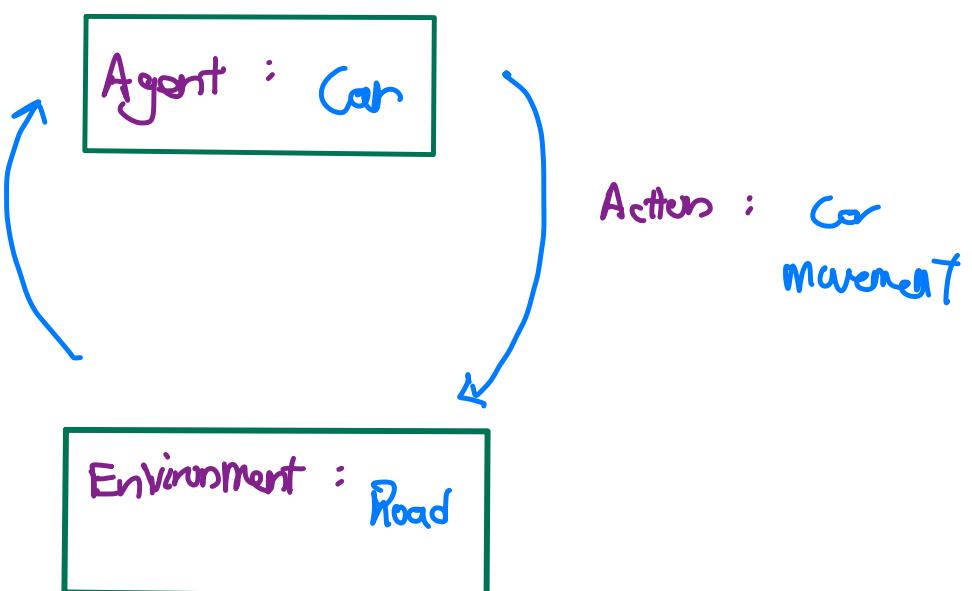
Program B : Control interaction with the vehicle

State : Element  
of  
car



Program C : Control the dung

State : Road  
situation /  
feedback



## Markov Decision Process :

- Made up of a series of

$S_0, a_0, r_0, S_1, a_1, r_1, S_2, a_2, r_2, \dots$

$\downarrow$        $\downarrow$   
 $0^{\text{th}}$  state       $0^{\text{th}}$  reward  
 $0^{\text{th}}$  action

Note: In other sources, this is defined as

$S_0, a_0, \underline{R_1}, S_1, a_1, \underline{R_2}, S_2, a_2, \dots$

- The end of this series is known as an episode

- Follows Markov Property

each state is only dependent on its immediate previous state

(The state here can refer to either state, action, reward)

- Intuition behind MDP is that

- optimize influence of state onto action, that can lead to maximum reward

$$S_0 \xrightarrow{} a_0 \xrightarrow{} r_0 \uparrow$$

- This is done through :

a) Policy,  $\pi$

$$\pi(a|s) = \text{Probability}$$

- which can be thought of as the probability of taking a certain action given a certain state.
- The involvement of randomness or probability ensures that the agent explores different actions given the same state.

b) Return,  $G_t$

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} \dots$$

where

$\gamma$  is a discount factor,  $0 \leq \gamma \leq 1$ ,  
discounting future reward given it is harder to predict



The goal of Markov Decision process is to :  
find the policy that can maximize the return !

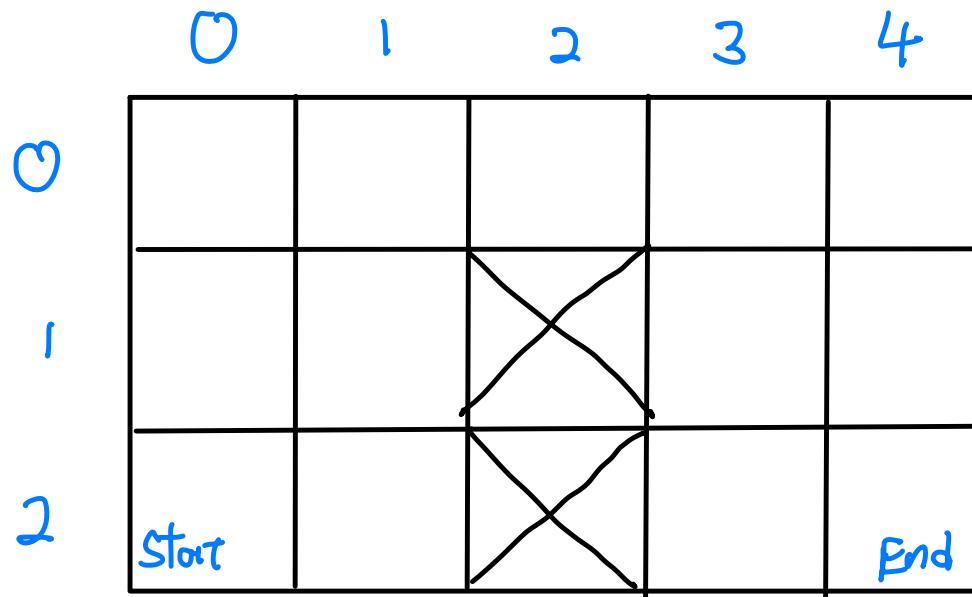
Chapter 2 :

Grid Example

+

Monte Carlo

## Grid Problem



Step 1 : Define State, Action, Reward & Episode,

Episode terminates when

a) step = 20

b) Reach End

State : 2 numbers  $\rightarrow$  (x, y) coordinates

Action : 1 number  $\rightarrow$  0 / 1 / 2 / 3  
( up / down / left / right )

Reward : 0 if in target cell .  
-1 otherwise

Step 2 : Define a world model,  $p$

In this case

if the agent is at  $(1, 1)$  and takes an action of 3 (right), it should stay in place as it is hitting the obstacle.

So,

the world model is as such :

$$p(s', r | s, a)$$

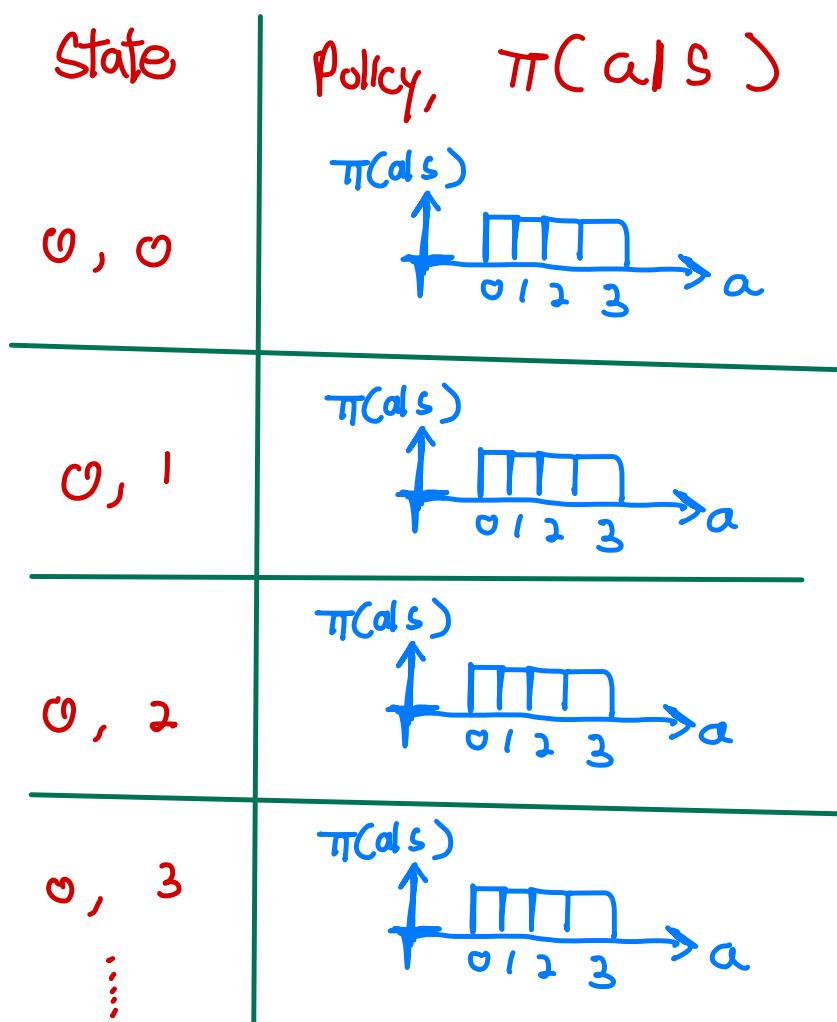
→ the subsequent state & reward given the previous state & actions

When the agent has access to the world model, it can learn significantly faster & better.

In this case, we are modelling the problem as a model free problem, so this step will be skipped ... ...

\* which also means the state & action numbers mean nothing to the agent at the start of training

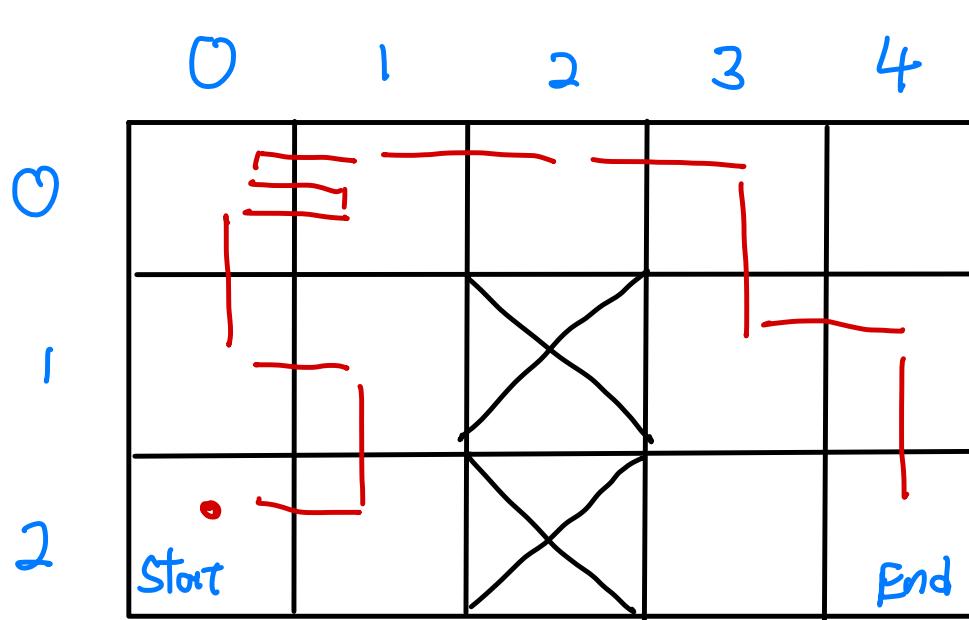
Because of the lack of world model access



otherwise, if we have access to the model,  
 $\pi(a|s)$  will not be uniform distribution

Step 3 : Sample a trajectory (an episode)

$t$	State	Action	Next State	Reward	Return
0	0, 2	3	1, 2	-1	-4.33
1	1, 2	0	1, 1	-1	-4.16
2	1, 1	2	0, 1	-1	-3.95
3	0, 1	0	0, 0	-1	-3.69
4	0, 0	3	1, 0	-1	-3.36
5	1, 0	3	2, 0	-1	-2.952
6	2, 0	3	3, 0	-1	-2.44
7	3, 0	1	3, 1	-1	-1.8
8	3, 1	3	4, 1	-1	-1
9	4, 1	1	4, 2	0	0



Easier to calculate bracketed

$$\text{Return, } g_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \dots$$

Given  $r = 0.8$

Why is it easier to calculate return from back?

$$g_9 \approx r_9 = 0$$

$$\begin{aligned}g_8 &\approx r_8 + \gamma r_9 = -1 + 0.8(0) \\&\approx -1\end{aligned}$$

$$\begin{aligned}g_7 &= r_7 + \gamma r_8 + \gamma^2 r_9 = -1 + \gamma(r_8 + \gamma r_9) \\&\approx -1 + \gamma g_8 \\&= -1 + (0.8)(-1) \\&= -1.8\end{aligned}$$

$$\begin{aligned}g_6 &= r_6 + \gamma r_7 + \gamma^2 r_8 + \gamma^3 r_9 \\&= r_6 + \gamma(r_7 + \gamma r_8 + \gamma^2 r_9) \\&= r_6 + \gamma(g_7) \\&= -1 + (0.8)(-1.8) \\&= -2.44\end{aligned}$$

In short,

$$g_t \approx r_t + \gamma g_{t+1}$$

Each time a trajectory or episode is sampled, the policy would have to be updated for the agent to perform better in the subsequent trajectory or episode.

This can be done in numerous ways:

- a) Policy gradient method
- b) Policy gradient method, with value function
- c) Only using action-value function

Before going deeper, let's review what are action-value function

- State-value function,  $V_{\pi}(s)$   
→ Average return ( $G_t$ ) expected when following a certain policy ( $\pi$ ) in a certain state, ( $s$ )
- Action-value function,  $Q_{\pi}(s, a)$   
→ Average return ( $G_t$ ) expected when following a certain policy ( $\pi$ ) in a certain state, ( $s$ ) + action, ( $a$ )

An example on how  $Q_{\pi}(s, a)$  is calculated  
Let's use back this sampled trajectory / episode

$t$	State	Action	Next State	Reward	Return
0	0, 2	0	0, 1	-1	-4.94
1	0, 1	0	0, 0	-1	-4.93
2	0, 0	3	1, 0	-1	-4.91
3	1, 0	0	1, 0	-1	-4.88
4	1, 0	3	2, 0	-1	-4.86
5	2, 0	2	1, 0	-1	-4.82
6	1, 0	0	1, 0	-1	-4.78
7	1, 0	3	2, 0	-1	-4.73
8	2, 0	2	1, 0	-1	-4.65
9	1, 0	3	2, 0	-1	-4.57

To calculate  $Q_{\pi}(s, a)$ , a table can be created for all possible combination of state & actions

(Not because that the model is aware of the world model)

→ Initialised at 0

	0	1	2	3
0	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : 0	3 : 0	3 : 0
0	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : 0	3 : 0	3 : 0
0	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : 0	3 : 0	3 : 0

From every step in the sampled trajectory / episode :

$$Q(S, a) = Q(S, a) + (g_t - Q(S, a)) \alpha$$

$$\text{new value} = \text{old value} + (\text{New return} - \text{old value}) \times \Delta\%$$

Let set  $\Delta\%$  at  $0.1$ .

 Take note :  
This is the  
method of  
calculating average

t	State	Action	Next State	Reward	Return
0	0, 2	0	0, 1	-1	-4.94
1	0, 1	0	0, 0	-1	-4.93
2	0, 0	3	1, 0	-1	-4.91
3	1, 0	0	1, 0	-1	-4.88
4	1, 0	3	2, 0	-1	-4.86
5	2, 0	2	1, 0	-1	-4.82
6	1, 0	0	1, 0	-1	-4.78
7	1, 0	3	2, 0	-1	-4.73
8	2, 0	2	1, 0	-1	-4.65
9	1, 0	3	2, 0	-1	-4.57



Let focus on  
those 2  
steps given  
they have  
same  $(S, a)$

At time step 7 :

$$\begin{aligned}\text{new value} &= 0 + (-4.57 - 0) (0.1) \\ &= -0.457\end{aligned}$$

At time step 4 :

$$\begin{aligned}\text{new value} &= -0.457 + (-4.73 - (-0.457)) (0.1) \\ &= -0.8843\end{aligned}$$

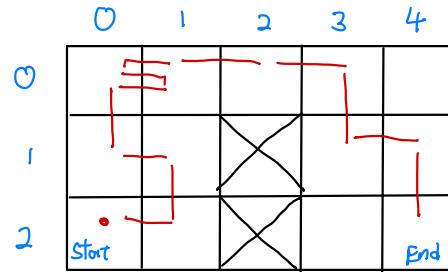
	0	1	2	3
0	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : -0.8843	3 : 0	3 : 0
0	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : 0	3 : 0	3 : 0
0	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : 0	3 : 0	3 : 0

Back to updating the policy,

This can be done in numerous ways :

- a) Policy gradient method
- b) Policy gradient method, with value function
- c) Only using action-value function

- Only using action-value function
    - No need to design policy
    - Agent makes decision by using the highest action-value



- Sample a trajectory / episode
  - Decision is done based on the highest value in action-value

	0	1	2	3
0	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : 0	3 : 0	3 : 0
	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : 0	3 : 0	3 : 0
	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : 0	3 : 0	3 : 0
	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : 0	3 : 0	3 : 0

◦ Update action-value,  $Q_{\pi}(s, a)$

✓ via

Monte Carlo  
(shows above)

t	State	Action	Next State	Reward	Return
0	0, 2	3	1, 2	-1	-4.33
1	1, 2	0	1, 1	-1	-4.16
2	1, 1	2	0, 1	-1	-3.95
3	0, 1	0	0, 0	-1	-3.69
4	0, 0	3	1, 0	-1	-3.36
5	1, 0	3	2, 0	-1	-2.952
6	2, 0	3	3, 0	-1	-2.44
7	3, 0	1	3, 1	-1	-1.8
8	3, 1	3	4, 1	-1	-1
9	4, 1	1	4, 2	0	0

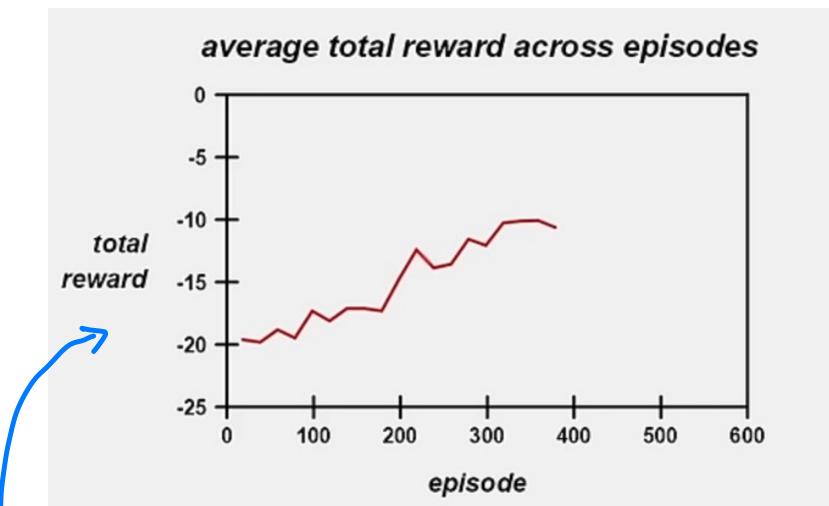
- Compute return (discounted)

↑ This cycle is known as generalized policy i

Approximating  $Q_{\pi}$   $\rightarrow$  Approximating  $Q_*$   
(optimal action value)

Policy,  $\pi$ , based on  $Q$ , also approaches  $\pi^*$   
(optimal policy)

p/s : A good way to visualize the learning of the agent is as following :



Total reward used here is undiscounted

$$g_t = r_t + r_{t+1} + r_{t+2} \dots$$

- which is why start at -20 because maximum step allowed is 20 steps.
- When the agent improves, it means that it takes less than 20 steps to reach end goal, hence has less steps with "-1" as reward.

A problem of using c) using only action-value is that it is 100% exploitation & no exploration in the end

However, a good RL agent should have a certain balance between exploration & exploitation

To introduce this balance,

- Epsilon,  $\epsilon$  is introduced
- where  $\epsilon$  :
  - i) proportion of time that actions are picked randomly
  - ii) decreases over time as policy improves and less exploration is needed
  - iii) i.e. start off at 0.9 and gradually decrease to 0.1

Note : This c) method is considered as a Monte Carlo process, which is defined as

→ a computational algorithm that rely on repeated random sampling to obtain numerical results

Chapter 3 :

Temporal

Difference

However, there's a few problems using Monte Carlo's approach

- 1) Has to wait until the entire trajectory / episode to be completed before action value function can be updated, hence is a slow & inefficient process.
- 2) Not even feasible on a continuous task where the episode has no termination point
- 3) Within the same episode, there is no way to evaluate each individual action separately.  
It relies on the large amount of sampled trajectories to average out the evaluation of each individual action
  - ↳ which is also the credit assignment problem
    - Figuring out the impact of an individual action within a sequence of many actions

The alternative approach to MONTE CARLO is known as TEMPORAL DIFFERENCE

Where

Monte Carlo

- Wait for the episode to be done
- Calculate the returns (as discounted reward) which would only be calculable when the entire episode is done

while

Temporal difference

- does not have to wait for the entire episode to be over
- Instead, it comes up with an estimate for each state-action pair relative to the state-action pair at the immediate after time step
- Therefore, only require the reward in between the 2 pairs
- Because of this, also does not need the episode to necessarily must be terminated

As previously discussed

$$g_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3}$$

$$g_t = r_t + \gamma (r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3})$$

$$g_t = r_t + \gamma g_{t+1}$$

As showcased above in Monte Carlo, Q will be updated as such:

\*  $\alpha$  is learning rate

$$\begin{aligned} Q(s_t, a_t) &= Q(s_t, a_t) \\ &\quad + \alpha (g_t - Q(s_t, a_t)) \\ &= Q(s_t, a_t) \\ &\quad + \alpha ([r_t + \gamma g_{t+1}] - Q(s_t, a_t)) \end{aligned}$$

which can be written as

$$Q(s_t, a_t) \rightarrow r_t + \gamma g_{t+1}$$

The action value,  $Q(s_t, a_t)$  is approaching  $g_t = r_t + \gamma g_{t+1}$

\* This method is inefficient

because computation of  $g_t$  needs to wait for the end of episode

In Monte Carlo :

$$Q(S_t, a_t) \rightarrow r_t + \gamma g_{t+1}$$

In Temporal Difference :

Approach 1 : SARSA

$$\text{Estimation} : g_{t+1} \approx Q(S_{t+1}, a_{t+1})$$

Hence,

$$Q(S_t, a_t) \xrightarrow{\text{Approach}} r_t + \gamma Q(S_{t+1}, a_{t+1})$$

$$Q(S_t, a_t) \xleftarrow{\text{Assign}} Q(S_t, a_t) \\ + \alpha(r_t + \gamma Q(S_{t+1}, a_{t+1}) - Q(S_t, a_t))$$

Approach 2 : Expected SARSA

$$\text{Estimation} : g_{t+1} \approx \sum_a \pi(a|s_{t+1}) Q(S_{t+1}, a)$$

\* Estimated as the sum of probability weighted actions that would be taken

Hence,

$$Q(S_t, a_t) \xrightarrow{\text{Approach}} r_t + \gamma \sum_a \pi(a|s_{t+1}) Q(S_{t+1}, a)$$

$$Q(S_t, a_t) \xleftarrow{\text{Assign to}} Q(S_t, a_t)$$

$$+ \alpha(r_t + \gamma \sum_a \pi(a|s_{t+1}) Q(S_{t+1}, a) - Q(S_t, a_t))$$

### Approach 3 : Q-Learning

$$\text{Estimation} : q_{t+1} \approx \max_a Q(S_{t+1}, a)$$

Hence,

$$Q(S_t, a_t) \xrightarrow{\text{Approach}} r_t + \gamma \max_a Q(S_{t+1}, a)$$

$$Q(S_t, a_t) \xleftarrow[\text{Assign}]{\leftrightarrow} Q(S_t, a_t)$$

$$+ \alpha (r_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, a_t))$$

On-Policy / Off-Policy

SARSA & Expected SARSA

- On-Policy

because

- a) they learn from the actions they already took
- b) behaviour policy = target policy

Q-Learning

- off-policy

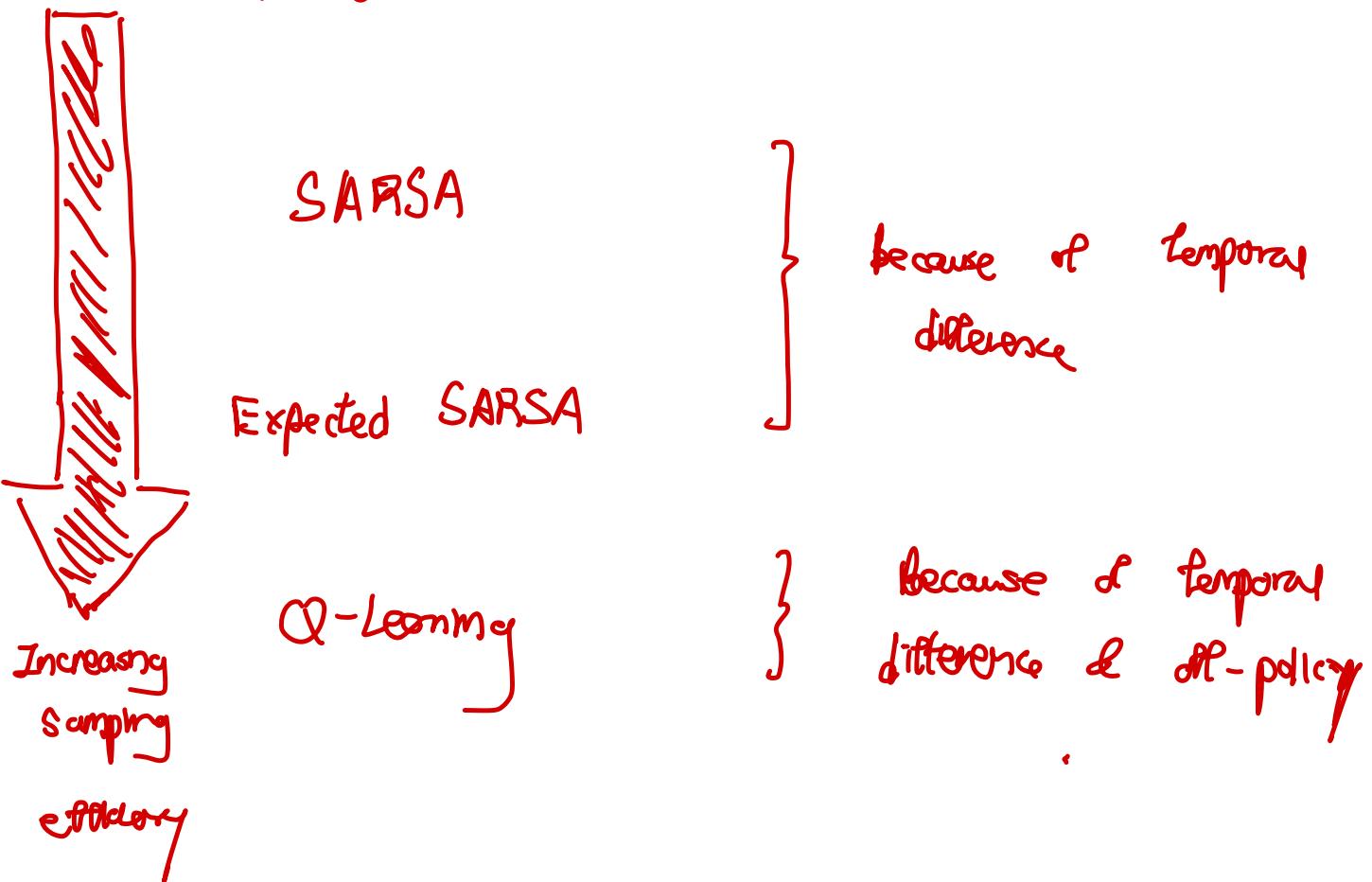
because

- a) it does not necessarily learn from the action it takes (due to randomness caused by  $\epsilon$ )

- b) behaviour policy  $\neq$  target policy

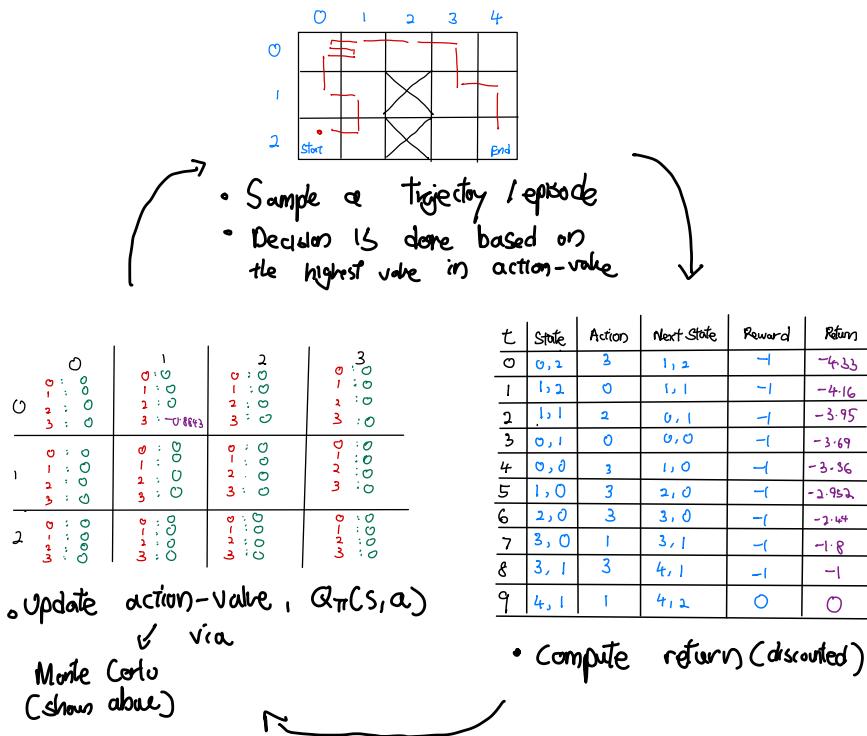
Sample Efficiency - Number of episodes required to get good at a task

## Monte Carlo



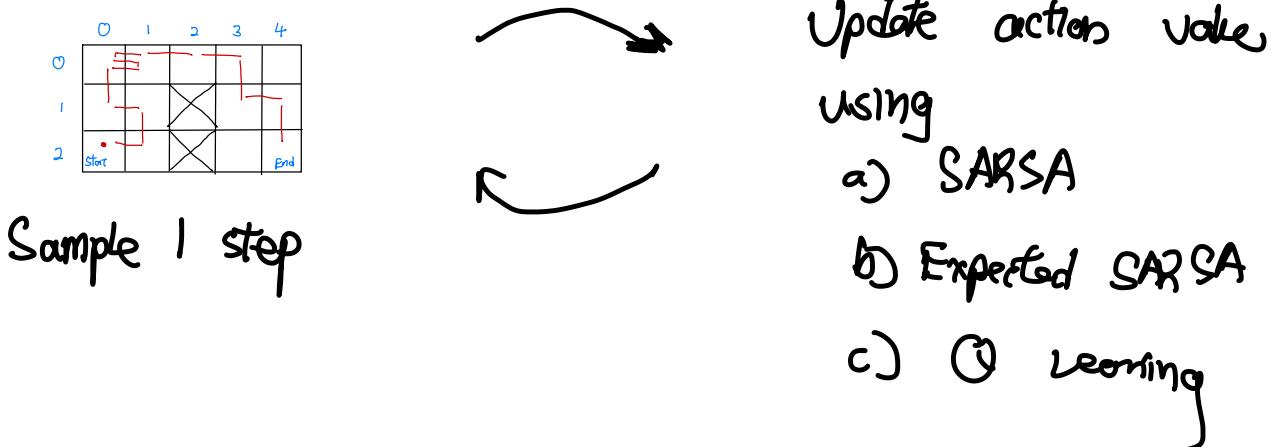
In Monte Carlo :

→ update of action value done once every end of episode



In Temporal difference

→ update of action value is done once every step is completed



Chapter 4 :

Deep Q Learning

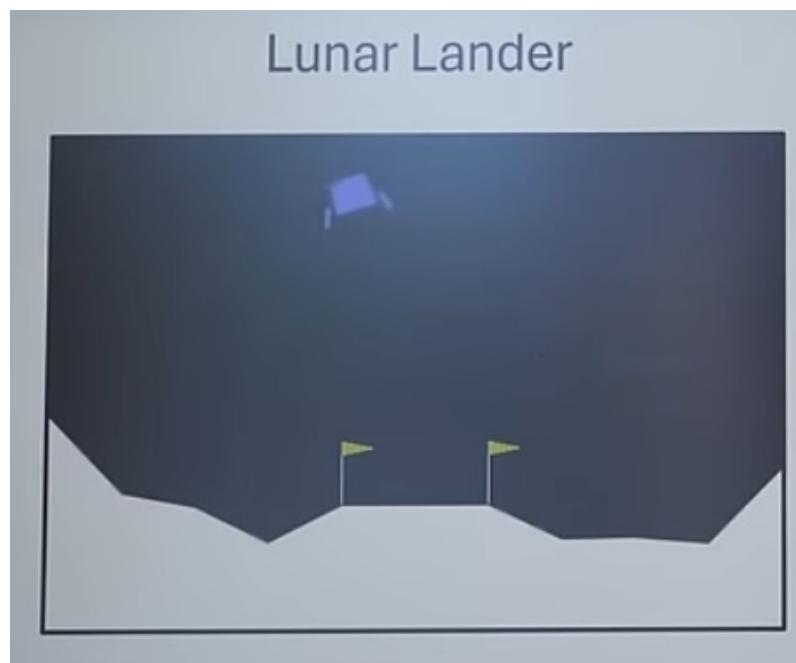
Deep Q Learning basically replaces the action value table used in Monte Carlo / Temporal Difference with a Deep Learning Network.

The advantage of this model is that it enables

input of continuous state  
output of discrete actions

(output of continuous action is still not available here, only can be enabled using policy-based method instead of action value based)

Let's explain Deep Q Learning with a case study



Agent : A rocket

Environment : Outer space

State :

- Coordinate in  $x$ ,  $x$
- Coordinate in  $y$ ,  $y$
- Linear Velocity in  $x$ ,  $v_x$
- Linear Velocity in  $y$ ,  $v_y$
- Angle ,  $\theta$
- Angular Velocity ,  $w$

Left Leg in contact with Ground , L

Right Leg in contact with Ground , R

Continuous

Discrete / Boolean

## Action (Discrete)

- No action
- Right Engine
- Left Engine
- Down Engine

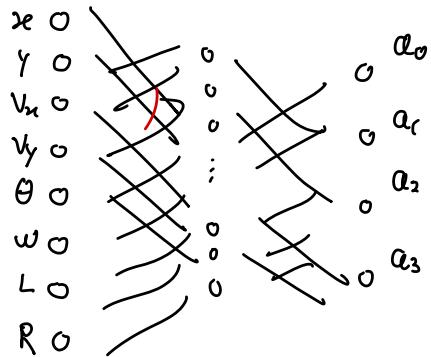
## Reward :

- a) The closer the lander to the landing pad, the higher the reward
- b) The slower the lander is moving, the higher the reward
- c) The less the lander is tilted, the higher the reward
- d) Increase by 10 for each leg in contact with ground
- e) Decrease by 0.03 each step a side engine is firing
- f) Decrease by 0.3 each step the main engine is firing
- g) +100 if land safely or -100 for crashing

An episode is considered as solved if scores at least 200.

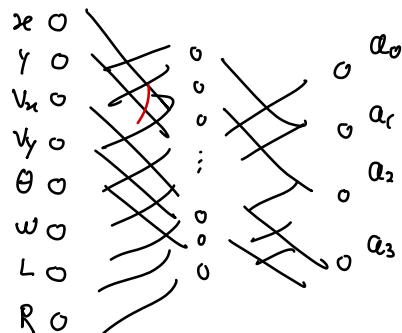
How is Deep Q Learning applied here?

## Initialization



$\theta$   
Main Network

→ use to decide  
actions to be  
taken during sampling

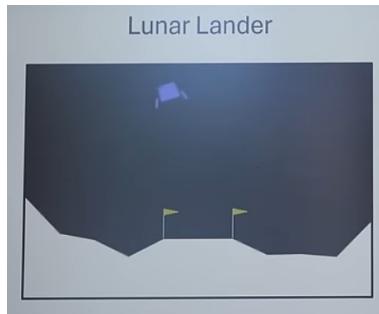


$\theta^-$   
Target Network

→ use to calculate  
target action, value  $Q$   
for loss computation

- Initialize  $\gamma$  (discount)
- Initialize  $\alpha$  (learning rate)
- Initialize  $\epsilon$  (exploration rate)

## Sampling Process



Collect state information ( $s$ )

$$s = s'$$

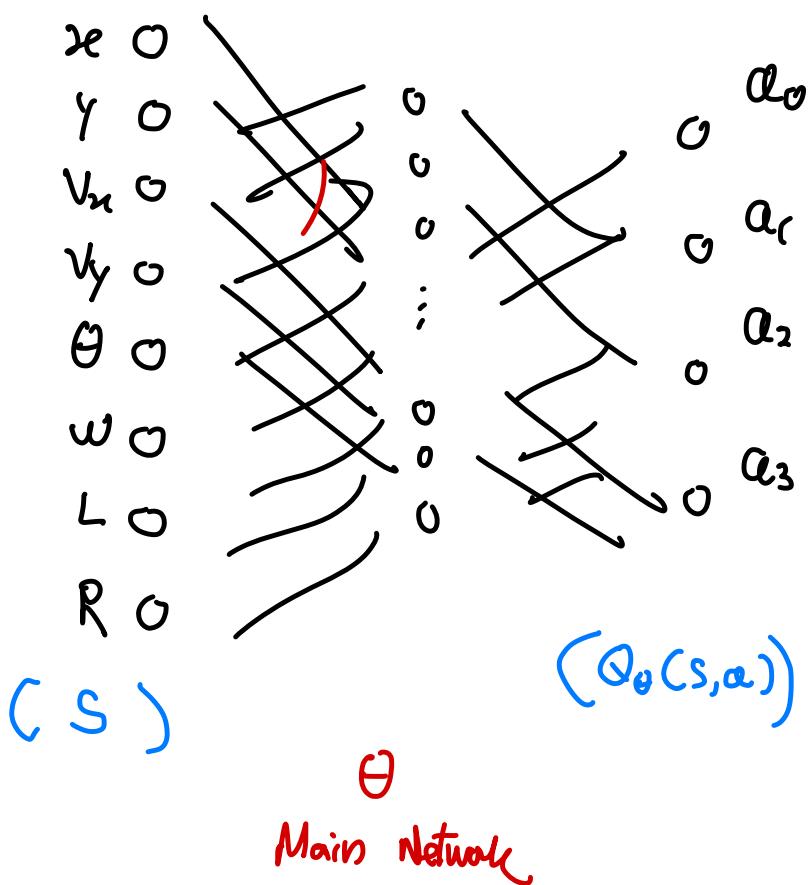


Store [ $s, a, r, s'$ ] in replay buffer



- Receive Reward ( $r$ )

- Rotate next state ( $s'$ )



Sample a probability ( $p$ )

if  $p > \epsilon$  :

Select  $a = \text{Argmax} [Q_\theta(s, a)]$

else :

$a = \text{Randomly selected}$

→ Perform selected action

Once enough samples are stored in replay buffer,  
the parameter update / Gradient descent process will  
be conducted !

### Training update

① Calculate Target action value,  $y$

$$y = r + \gamma \max_{a'} Q_{\theta^{-1}}(s', a')$$

This part is similar to Q learning

where

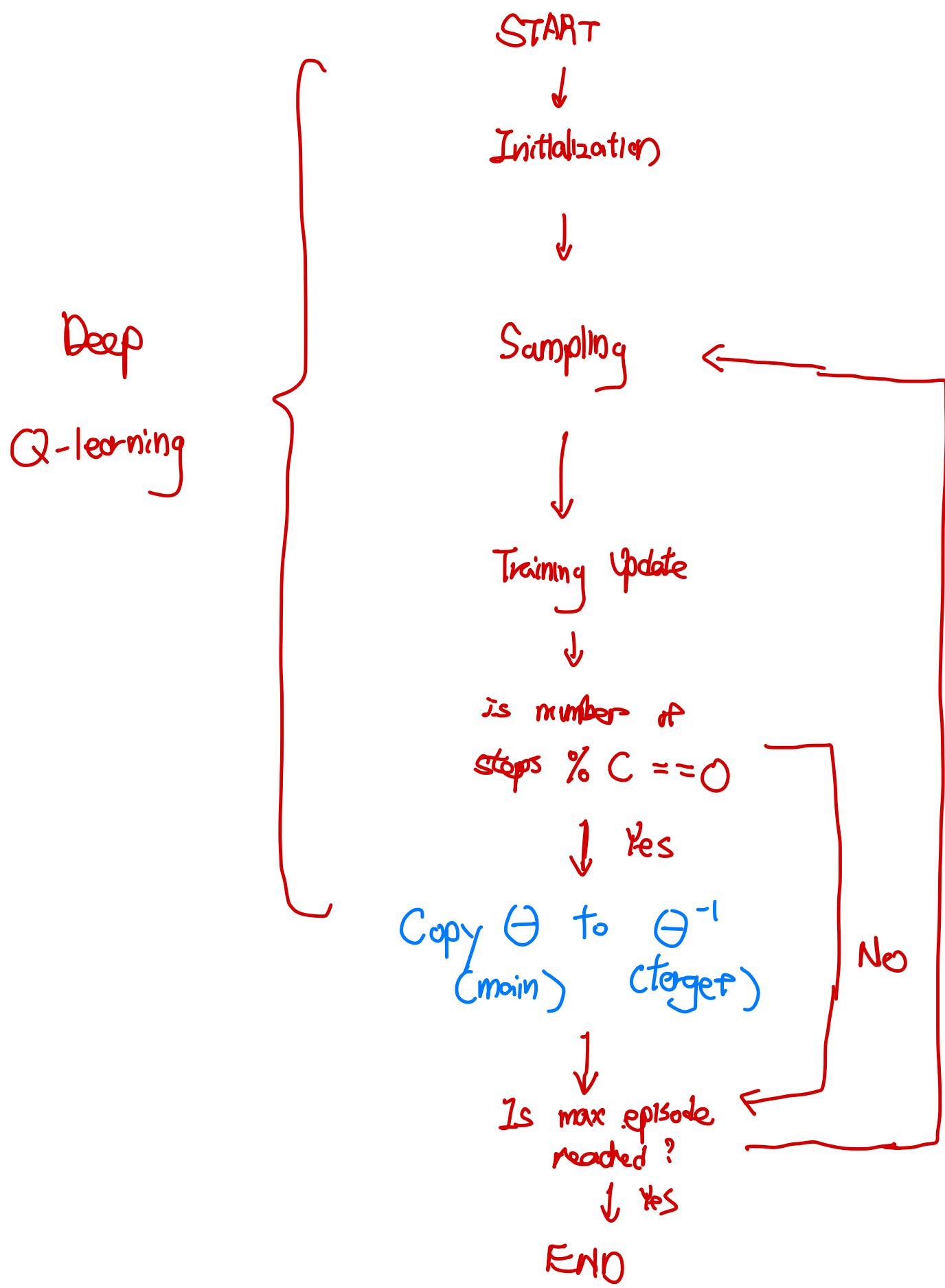
- 1) Pass next state,  $s'$  to the target network,  $\theta^{-1}$
- 2) To get a list of action value  $Q_{\theta^{-1}}(s', a')$
- 3) From these action values, pick the one that is maximum ( $\max_{a'}$ )

② Compare Target action value,  $y$  with the value output by main network  $\theta$ , using current state ( $s$ ), not next state ( $s'$ ) using a Loss function

$$L = E(y - Q_\theta(s, a))^2$$

\*  $E \rightarrow$  expected value  $\rightarrow$  average

③ Perform Backpropagation & Gradient Descent to update the parameter in  $\Theta$



# Comparison between Q-Learning & Deep Q Learning

In Q-Learning :

$$Q(S_t, A_t) \xrightarrow{\text{Approach}} r_t + \gamma \max_a Q(S_{t+1}, a)$$

$$Q(S_t, A_t) \xleftarrow[\text{Assign}]{\leftrightarrow} Q(S_t, A_t)$$

$$+ \alpha (r_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

In Deep Q-Learning :

$$L = E((y - Q_\theta(s, a))^2)$$

$$= E((r + \gamma \max_{a'} Q_{\theta^{-1}}(s', a') - Q_\theta(s, a))^2)$$

Based on the maximum action value in the next state.

Slowly move the current action value towards that estimated / target value

Why do you need 2 networks here?

- ① Main network not always the same as target network
  - Notice the target network did not get replaced by main network every update
  - So, it allows the main network to be trained for a while and become a little better before it is set as the target
  - If you use the same network to calculate both target & current behaviour, your target will keep changing every step & the model will become unstable.
- ② By using a separate network & only update it occasionally, it ensures that the target remains constant for a while (a few steps) before it is changed!

# Some of the tricks used in Deep Q Learning

## 1) Experience replay

- Store  $[S_t \ A_t \ R_t \ S_{t+1}]$  in a buffer
- Efficient use of experience during training
  - allow same experience to be reused
  - Avoid forgetting previous experience

## 2) Random sampling from replay buffer

- Remove correlation in the observation sequence

## 2) Fixed Q Target

- Use a different deep learning network to estimate Q Target and only update at an interval of few steps
- Avoid constant changing of Q Target

# Full pseudocode for Deep Q Learning (DQN)

For each step,  $t$

① Update epsilon,

$$\epsilon_t = \max\left(\epsilon_{start} + \frac{\epsilon_{end} - \epsilon_{start}}{\text{Total number of exploration episode}} \times t, \epsilon_{end}\right)$$

②

Sample a random number

- If more than epsilon, Sample an action with highest probability by feeding current state to the Q network,  $Q_\theta$
- Else, sample a random action

③

Take a step in that action & store the following information in a replay buffer

- Check if truncated
  - if it is, the agent will be in meaningless next state,  $S_{t+1}$  and need to be replaced from infos.
- Store  $S_{t+1}$ ,  $S_t$ ,  $a_t$ ,  $r_t$ , terminal\_flag, infos in buffer

④

Pass the next state to the next step.

⑤ If step > learning\_start\_step :

If step % learning\_interval == 0 :

- sample a batch of experience from buffer  
 $S_t, S_{t+1}, a_t, r_t, \text{Termination\_flag}, \text{infos}$

- Compute Target

$$\text{Target} = r_t + \gamma \left[ \max_{a_{t+1}} Q_\theta^-(S_{t+1}, a_{t+1}) \right] [1 - \text{Termination\_Flag}]$$

- Compute current value

$$\text{Current value} = Q_\theta(S_t, a_t)$$

- Compute Loss

$$\text{Loss} = \text{MSE}(\text{Current value}, \text{Target})$$

- Backpropagation & Gradient Descent

If step % target\_network\_update\_f == 0 :

$$Q_\theta^- \leftarrow \tau(Q_\theta) + (1-\tau)(Q_\theta^-)$$

(1) When  $t = \text{Number of episode to explore}$

$$\epsilon_{\text{start}} + \frac{\epsilon_{\text{end}} - \epsilon_{\text{start}}}{\text{Total number of exploration episode}} \times t < \epsilon_{\text{end}}$$

Therefore,  $\epsilon = \epsilon_{\text{end}}$

(5)

$$\text{Target} = r_t + \gamma \left[ \max_{a_{t+1}} Q_{\theta^-}(s_{t+1}, a_{t+1}) \right] \left[ \begin{array}{l} \text{- Termination\_Flag} \\ \hline \end{array} \right]$$

- Pass the  $s_{t+1}$  that was stored in buffer to the target network
- Based on the output, take the value of the node that has highest value

If termination flag = 1,  
that means the agent ends in  $s_{t+1}$  triggers termination,  
no need for a next step  
again

$$\text{Current value} = Q_{\theta}(s_t, a_t)$$

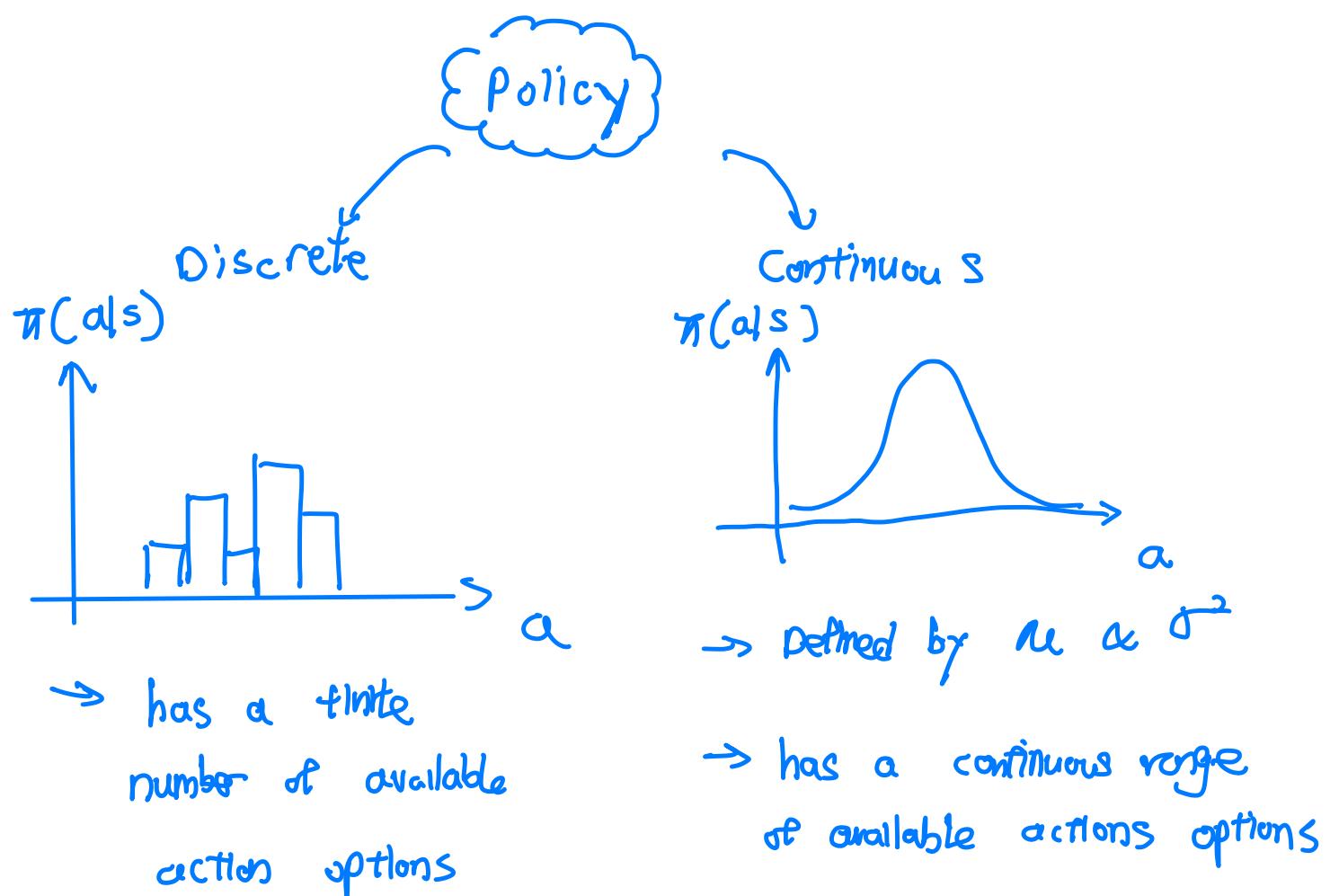
- Pass  $s_t$  to the  $Q$  network
- From the output, find the value of the node corresponds to  $a_t$
- Both  $s_t$  &  $a_t$  are from the buffer

Chapter 5 :

Policy  
Gradient

Up until this point, all the discussions on how the agent decides an action to take is purely action-value based.

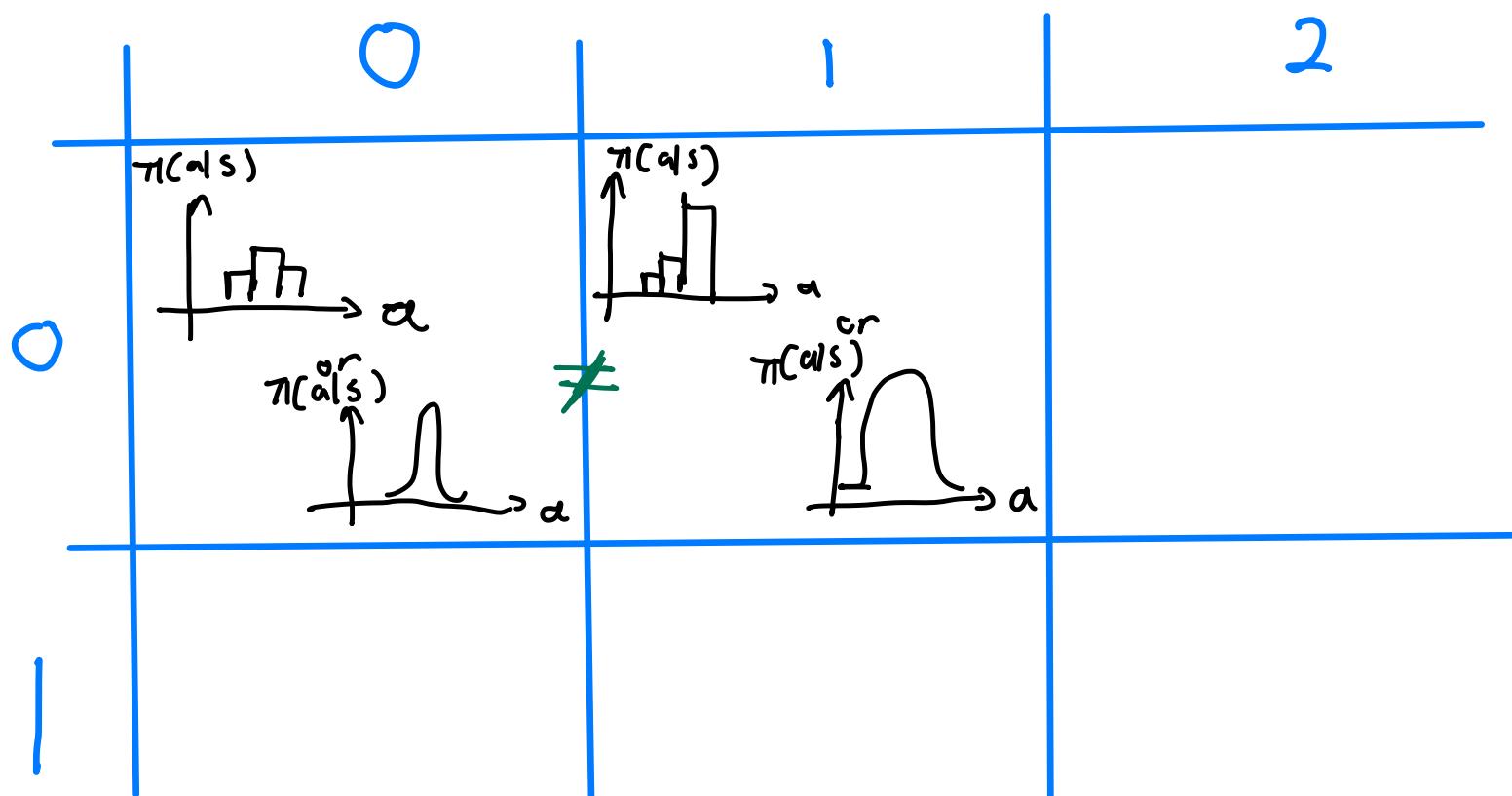
Now, let's talk about policy based, which offer some randomness naturally for exploration  
(Unlike action-value based that depends on  $\epsilon$  for exploration)



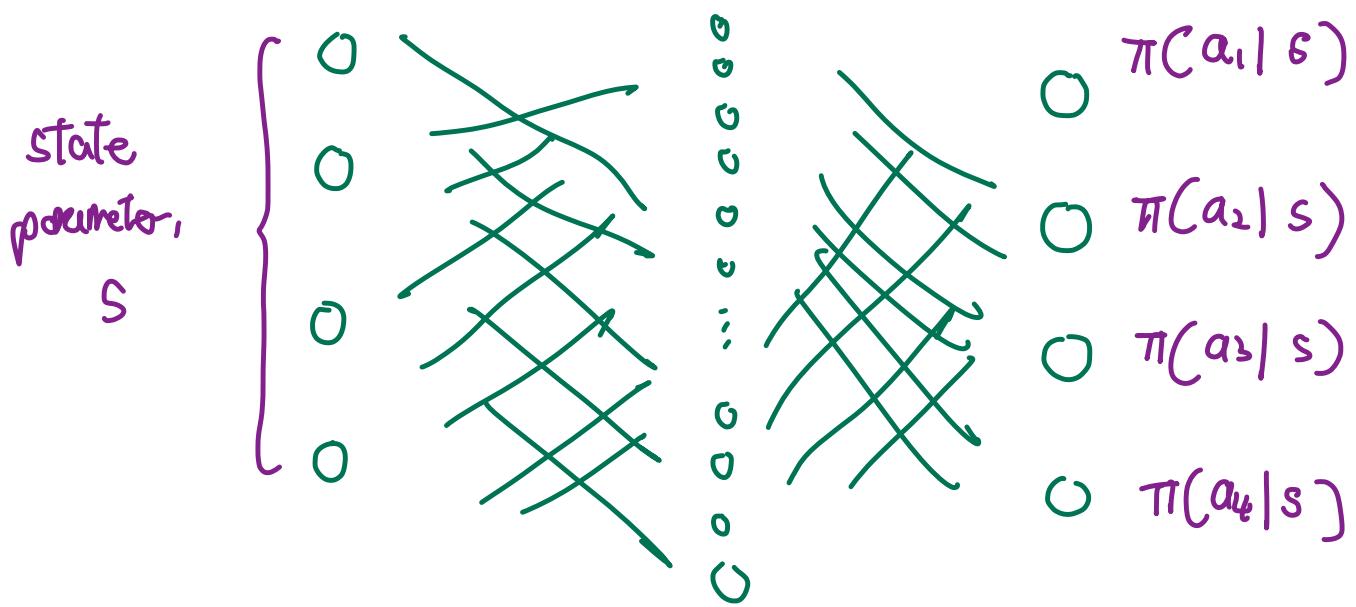
## Some key concept :

- a) When a policy is involved, when decision making, a random action will be sampled from the distribution.
- \* The higher the  $\pi(a|s)$ , the more likely a will be selected

- b) As we introduce earlier, different state would have different policy distribution



However, when a neural network is introduced to govern the policy, only 1 neural network would ever be needed

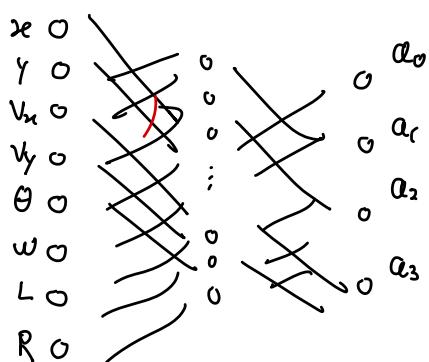


Policy network,  $\pi(a|s)$



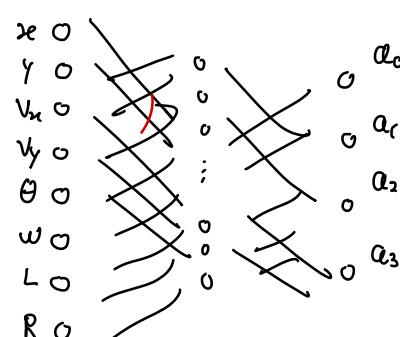
But how do you train this policy network?

For a action-value network or state-value network, as previously showcased in Deep Q Learning,



$\theta$   
Main Network

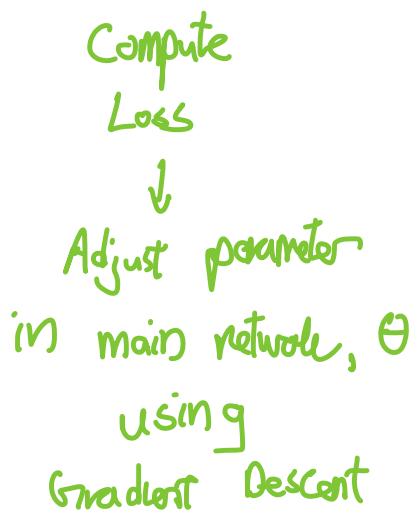
$$Q(s, a)$$



$\theta^*$   
Target Network

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

[TARGET]



- ① But there is no Ground Truth nor Target
- in the training of the policy network,  $\pi$

- Given there's no ground truth or target
- There will be no loss
- If there's no loss that needs to be minimized
- Then, we need to find other ways to quantify the performance of the policy network in always leading to good decision with high reward or return
- And maximize that performance

That performance quantity is defined as

## PERFORMANCE OBJECTIVE

$J(\pi_\theta)$

Performance objective  $\leftarrow$   $J(\pi_\theta)$  ↳ of a policy network,  $\pi$  with  $\theta$  parameter

$J(\pi_\theta)$  can be mathematically quantified in many ways :

$$\textcircled{1} \quad J(\pi_\theta) = E_{J \sim \pi_\theta} [R(J)]$$

- Expected value of the reward at each step in all possible trajectories,  $J$ , by following the  $\pi_\theta$  policy

$$\sum_{\text{All possible traj}} \left[ \begin{array}{c} \text{Probability of a trajectory} \\ \times \\ (\text{Probability of taking actions that leads to this trajectory}) \end{array} \right] \times \text{Total Reward in that trajectory}$$

$\pi(a_1 | s_0) \xrightarrow{\text{multiply}}$

$\xrightarrow{\text{where } a_1 \text{ leads to } s_1} \pi(a_3 | s_1) \xrightarrow{\text{multiply}}$

$\xrightarrow{\text{where } a_3 \text{ leads to } s_3} \pi(a_5 | s_3) \dots$

$$\textcircled{2} \quad J(\pi_0) = V_{\pi}(S_0)$$

- The expected value of cumulative reward starting from initial state,  $S_0$  following policy,  $\pi$

Can be calculated using

- Monte Carlo  
→ after each trajectory,

$$V(S_t) = V(S_t) + \alpha ([r_t + \gamma g_{t+1}] - V(S_t))$$

- Note :
- This method is called running average
  - = average = expected value
  - This formula update for all state
  - $V_{\pi}(S_0)$  only look at the  $S_0$  state

- Temporal Difference - using SARSA

Estimation :  $g_{t+1} \approx V(S_{t+1})$

Hence,

$$V(S_t) \xrightarrow{\text{Approach}} r_t + \gamma V(S_{t+1})$$

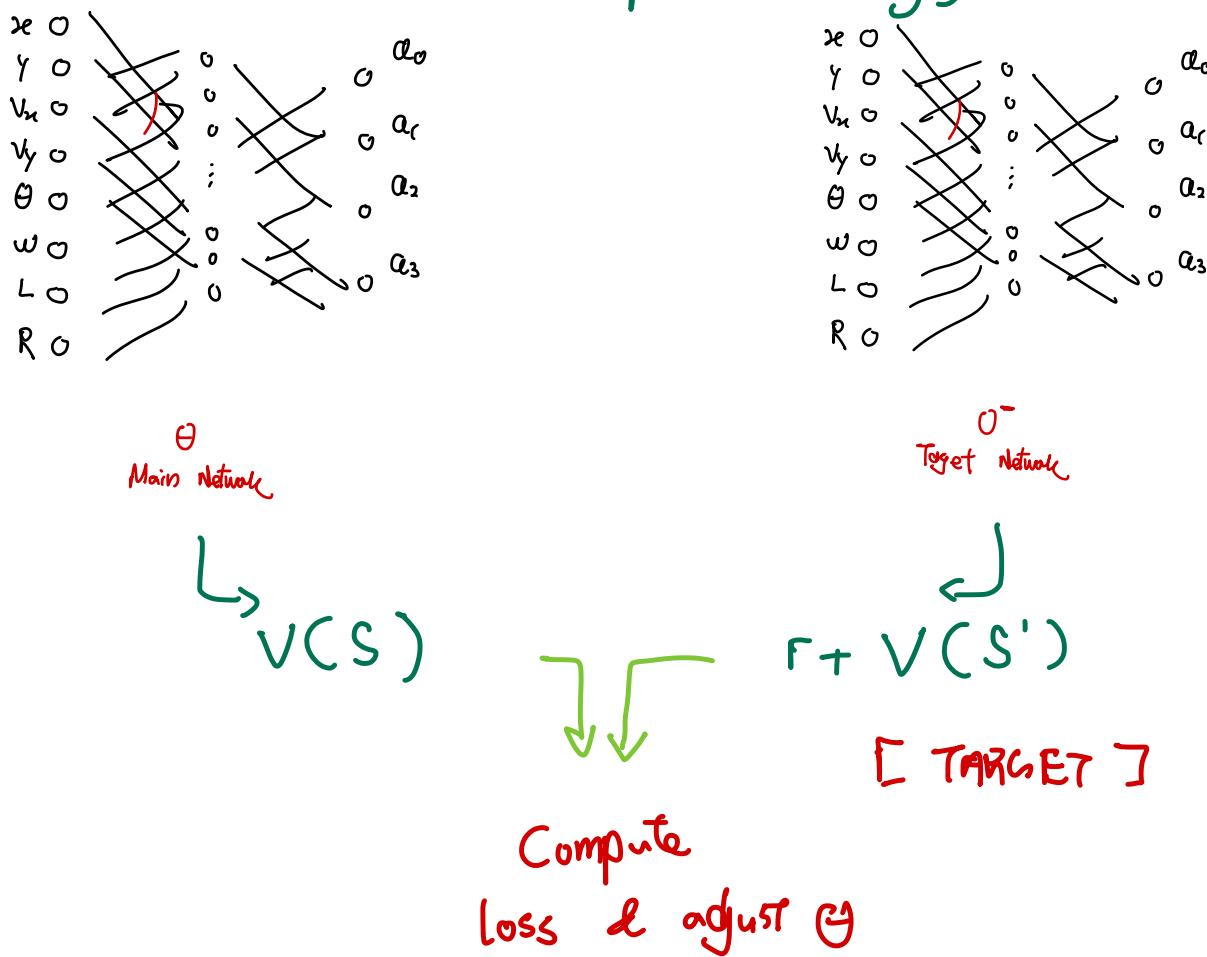
$$V(S_t) \xleftarrow{\text{Assign}} V(S_t)$$

$$+ \alpha (r_t + \gamma V(S_{t+1}) - V(S_t))$$

Note:  
 This method is still running average  
 $\Rightarrow$  average = expected value, because  
 derived from Monte Carlo

- This formula update for all state
- $V_{\pi}(S_0)$  only look at the  $S_0$  state
- It is updated after each step in Monte Carlo

c) Use a state value network  
 (similar to deep Q learning)



Note:  
 Find  $S_0$  to the model to get  $V_{\pi}(S_0)$

$$\textcircled{3} \quad J(\pi_\theta) = \sum_s d_\pi(s) V_\pi(s)$$

- The expected value of cumulative reward starting from each state,  $s_0$  following policy,  $\pi$ , weighted by the frequency of the state being visited  $V_\pi(s)$

$$\textcircled{4} \quad J(\pi_\theta) = \sum_s d_\pi(s) \sum_a \pi_\theta(a|s) r_{a,s}$$

- The expected value of reward, weighted by the probability of the action & the frequency in the state being visited.

But generally, no matter which definition is used,

Performance objective,  $J(\pi_\theta)$  measures the average reward returned by following policy  $\pi$

## Performance Objective, $J(\pi_\theta)$

↓ through policy gradient theorem

$$\nabla_\theta J(\pi_\theta) = E \left[ \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t | s_t) \cdot \Phi_t \right]$$

$\nabla_\theta J(\pi_\theta)$  : Derivative of the policy performance in respect to the policy network parameter

$E \left[ \sum_{t=0}^{\infty}$  : Expected value / Mean  
up sum of across all time steps in the trajectory

$\nabla_\theta \log \pi_\theta(a_t | s_t)$  : Derivative of log of the probability of that a being taken at state  $s$ , in a step  $t$  of a trajectory with respect to the policy network parameter

$\bar{V}_t$

: Advantage, measure how good or  
how bad the action is

Can be quantified by

a)  $g_t$  : Total discounted return after that step

- x bad because some state is naturally closer to the goal, hence has higher  $g_t$
- x Not a good way to evaluate each action
- x high variance

b)  $g_t - V\pi(S_t)$

✓ Make it fair by considering how good that state is

- x but  $g_t$  needs to wait for the whole episode to end

c)  $Q_\pi(S_t, A_t)$  [using deep Q learning]

- ✓ Make use of temporal difference to evaluate the action

✓ So no need to wait for end  
of trajectory

✗ Does not consider state difference  
as a)

d)  $Q_{\pi}(s_t, a_t) - V_{\pi}(s_t)$

✓ Consider state difference

✗ Heavy computational, as requires  
3 networks  $\xrightarrow{\text{policy}}$   
 $\xrightarrow{Q}$   
 $\xrightarrow{V}$

e)  $r_t + \gamma V_{\pi}(s_{t+1}) - V_{\pi}(s_t)$

$\underbrace{\phantom{r_t + \gamma V_{\pi}(s_{t+1})}}$        $\underbrace{\phantom{V_{\pi}(s_t)}}$   
Target  $\checkmark$                   Current  $\checkmark$

✓ Measures how small their difference is

Note : e) is known as Temporal difference  
error (TD error). which also appears  
in action-value update using SARSA  
or Q learning

TD Error  $\xrightarrow{\text{update}}$  policy network through  
advantage (critic)  
 $\xrightarrow{\text{update}}$  value network (actor)

Once  $\nabla_{\theta} J(\pi_{\theta})$  is obtained, GRADIENT ASCEND

can then be conducted:

$$\Theta \leftarrow \Theta + \alpha \Delta_{\theta} J(\pi_{\theta})$$

$\alpha$  learning rate

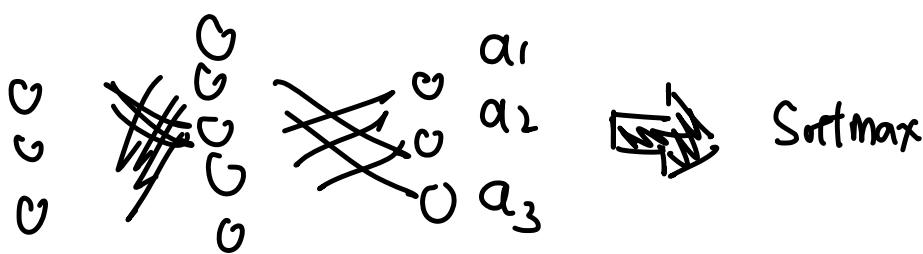
If positive



If negative

← - more left

When policy network is to account for discrete actions



$\circ \pi(a_1|s)$

$\circ \pi(a_2|s)$

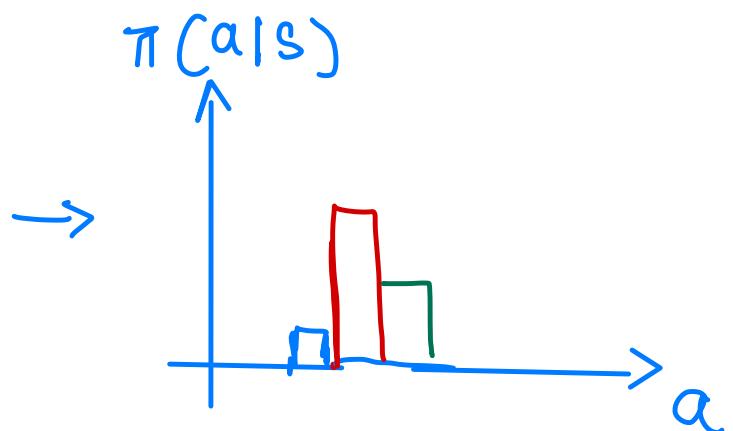
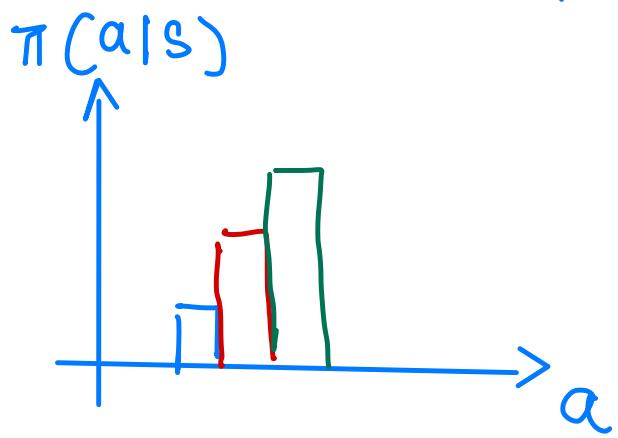
$\circ \pi(a_3|s)$



- Compute  $\Delta_{\theta} J(\pi_{\theta})$
- Update policy network via Gradient Descent



## Visualization of policy update in discrete actions



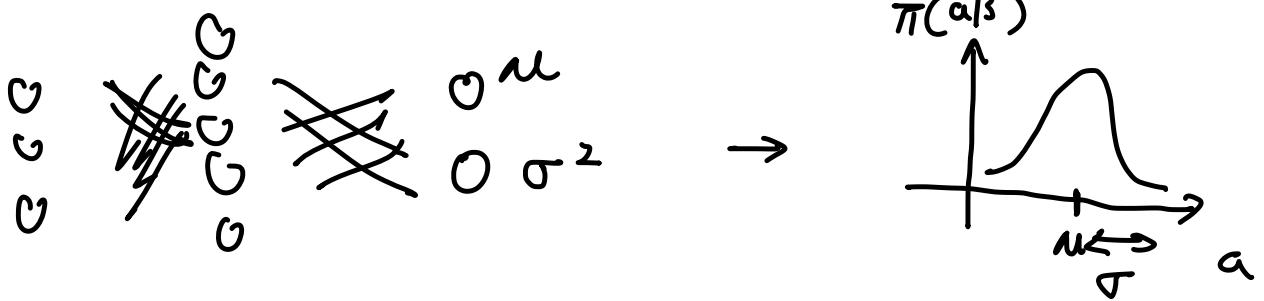
- Due to the evaluation made by the advantage,  $\Psi$ ,  $a_2$  is deemed a good action, receive a high  $\Delta \mathcal{J}(\pi_\theta)$  where the parameter in the policy network are tuned in such a way that leads to higher  $\pi(a_2|s)$ .

- Because the policy neural network's output will go through softmax, when  $\pi(a_2|s)$ ,  $\pi(a_1|s)$  &  $\pi(a_3|s)$  decrease.

\*  $a_2 \rightarrow$  get positive advantage

\*  $a_1, a_3 \rightarrow$  get negative advantage

When policy network is to account for continuous actions



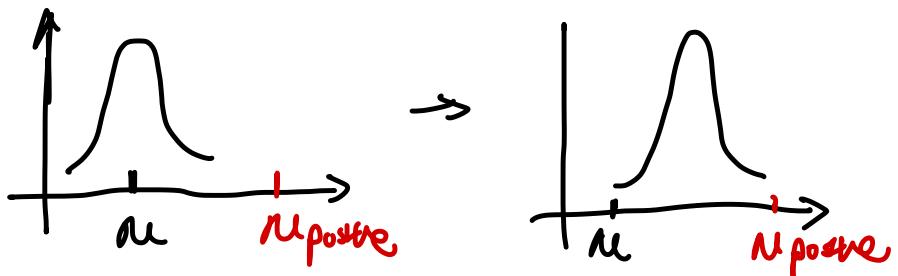
- Compute  $\Delta_\theta J(\pi_\theta)$
- Update policy network via Gradient Descent



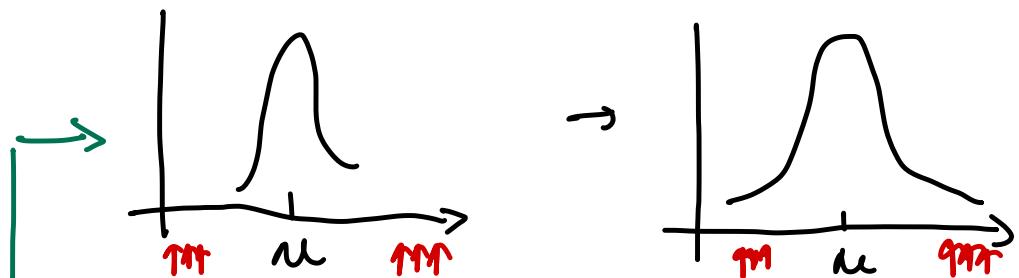
Visualization of policy update in continuous actions

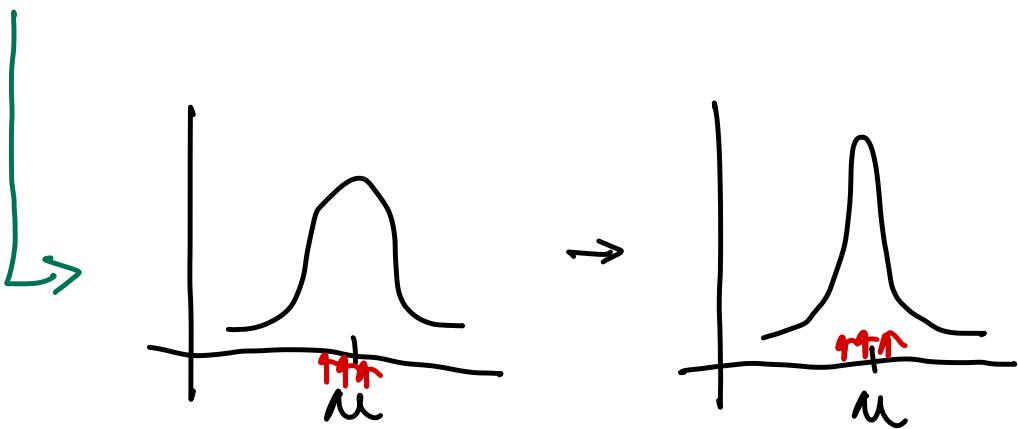
when get positive advantage

Adjust mean  
to be closer



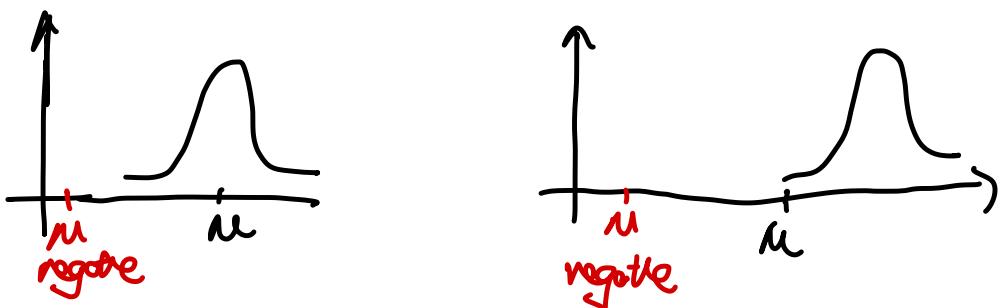
Adjust  $\sigma^2$   
to be closer



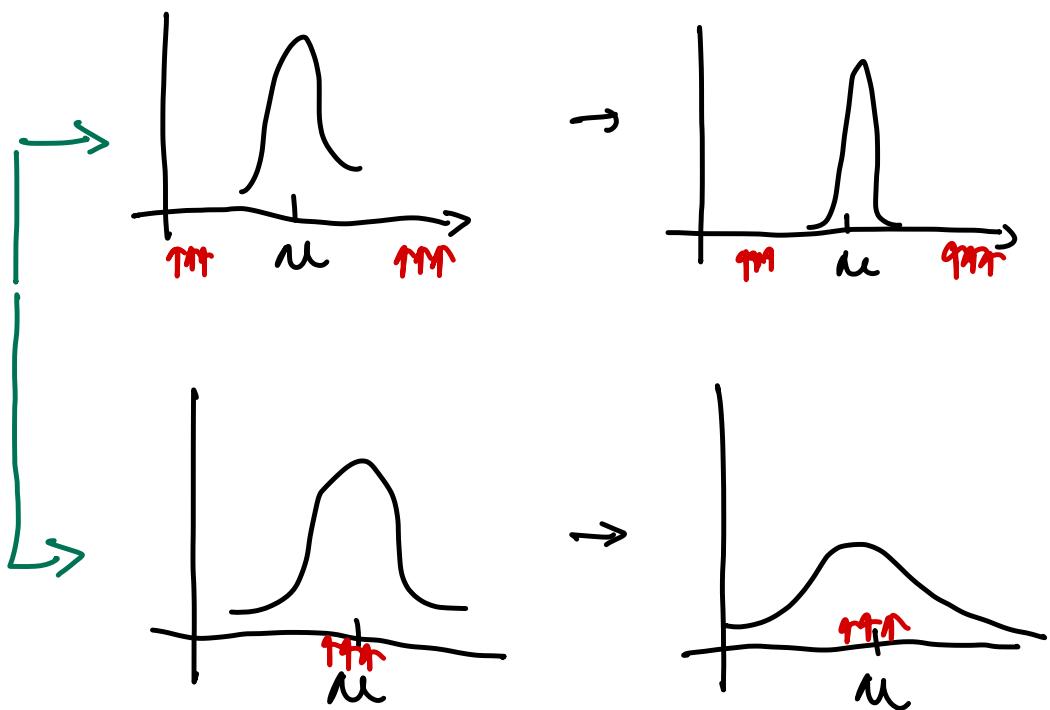


When get negative advantage

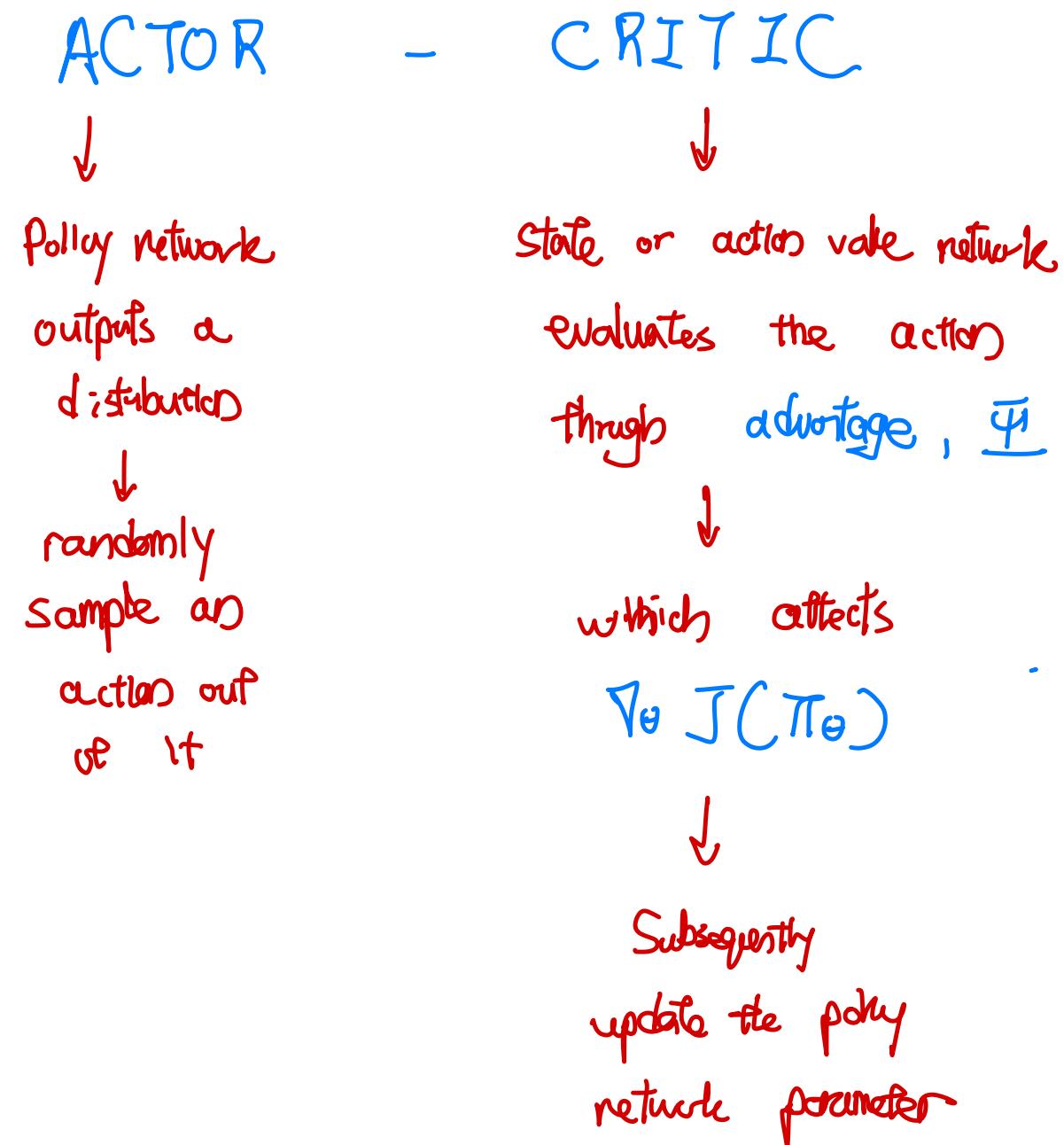
adjust mean  
to be further



adjust  $\sigma^2$   
to be further



This policy gradient update is based on a principle known as



# The flow of using policy gradient

- ① Collect the current state ( $s$ )
  - ② Feed the state information ( $s$ ) into the policy network to get a action distribution
  - ③ Sample an action ( $a$ ) from that distribution
  - ④ Store  $[s, a, r, s']$  in  
↓      ↓  
reward      next state,  
replay buffer. Once enough samples are  
in the buffer, the action or state  
value network will be updated
  - ⑤ Calculate performance objective,  $J(\pi_\theta)$  as  
well as its derivative with respect to the  
policy network parameter,  $\nabla_\theta J(\pi_\theta)$ .  
Then perform gradient ascent to adjust  
the policy network
  - ⑥ Repeat 1 - 5 until maximum number  
of episode is reached.
- Critic update
- Actor update

Frequency of critic update & frequency of actor update may differ based on different algorithms

→ but the flow is similar !

## DYNAMIC PROGRAMMING

$$Q(s, a) = Q(s, a) + (\hat{q}_t - Q(s, a)) \alpha$$

Note:

$$V(s) = V(s) + (\hat{q}_t - V(s)) \alpha$$



Markov Reward Process (only has state value fn)

$$V = R + \gamma P V$$

$$\begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} + \gamma \underbrace{\begin{bmatrix} p_{11} & \dots & p_{1n} \\ \vdots & \ddots & \vdots \\ p_{n1} & \dots & p_{nn} \end{bmatrix}}_{\text{probability of changing to another state}} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

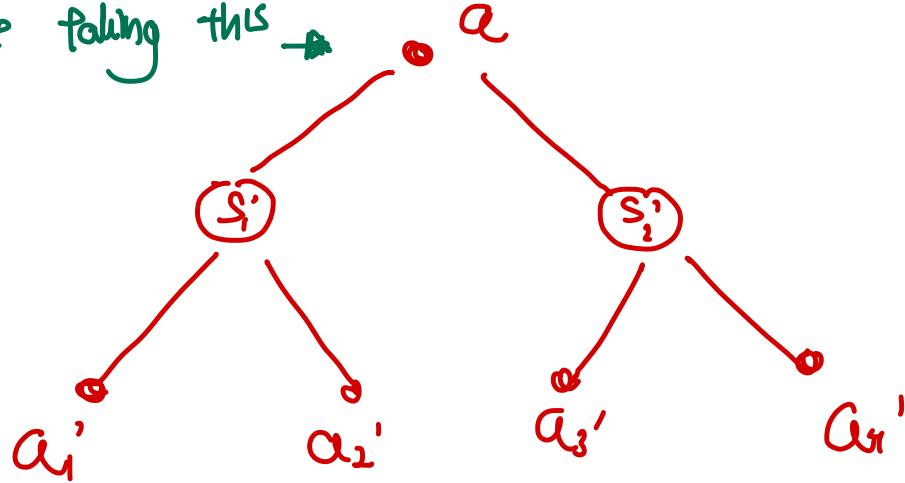
$\xrightarrow{\text{current / expected value mean}}$

Markov Decision Process (Has both action & value)

$$Q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} \sum_{a' \in A} \pi(a'|s') Q_\pi(s', a')$$

The value of taking this action

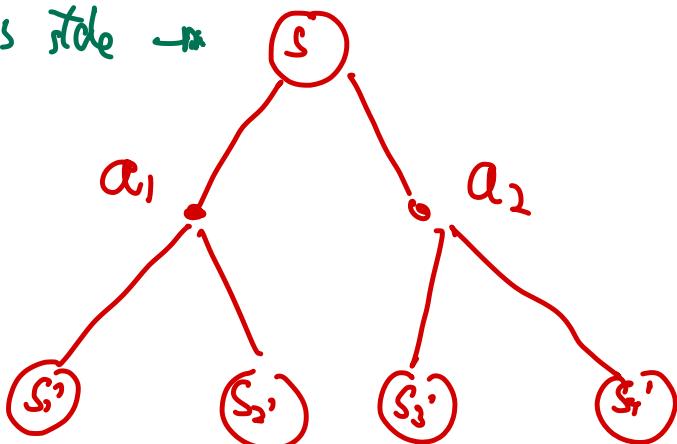
in relevant to these weighted actions



$$V_{\pi}(s) = \sum_{a \in A} \pi(a|s) \left( R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{\pi}(s') \right)$$

The value of this state

in relevant to these states



proof that mean can be calculated incrementally

$$\mu_k = \frac{1}{k} \sum_{j=1}^k x_j$$

$$= \frac{1}{k} (x_k + \sum_{j=1}^{k-1} x_j) \Rightarrow \mu_{k-1} = \frac{1}{k-1} \sum_{j=1}^{k-1} x_j$$

$$= \frac{1}{k} (x_k + (k-1) \mu_{k-1}) \quad (k-1) \mu_{k-1} = \sum_{j=1}^{k-1} x_j$$

$$= \frac{1}{k} x_k + \frac{1}{k} (k-1) (\mu_{k-1}) - \frac{1}{k} \mu_{k-1}$$

$$= \frac{1}{k} x_k + \mu_{k-1} - \frac{1}{k} \mu_{k-1}$$

$$= \mu_{k-1} + \frac{1}{k} (x_k - \mu_{k-1})$$



In the previously showcased example,

- $\frac{1}{k}$  is set to 0.1
- Ideally,  $\frac{1}{k}$  should depends number of times the state ( $x$ ) has been visited,  $\alpha_x$
- but, by setting  $\frac{1}{k}$  to 0.1 ( $\alpha$ ), a learning rate, it helps to forget some older episodes

Chapter 6 :

Problems

2

Future Research Directions

# Problems with RL

- Unreliable
  - with exactly same settings (except different initialization seed), the performance will vary greatly
- Sample Inefficiency
  - Take high number of steps and episodes to reach good performance

Some research direction to address these issues

- a) Model-based reinforcement learning
  - with the help of a world model, it significantly reduces the number of trials & eras the agent need to go through to learn basic ways of how the environment around it works.
- b) Imitation learning / Inverse RL
  - Learn policy from an expert's trajectories, without reward signals
  - The expert's trajectories can be obtained by

- behavior cloning
- having a human to perform the task.
  - Can be useful when a good reward is hard to be defined

→ Bad when a random action causes the agent to deviate from the expert trajectories, causing error accumulation

- Can be fixed by dataset aggregation (DAGGER) where noise is added to the expert's trajectories, and have humans annotate the right action to take under such noise

→ Bad because tedious annotation