

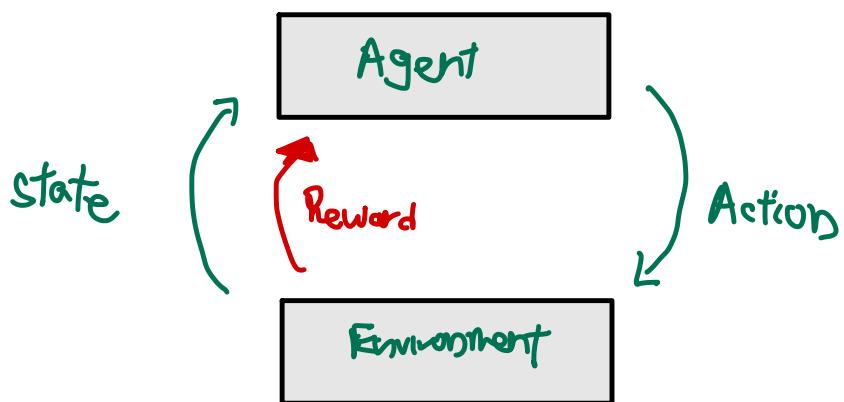


Chapter 1 :

Terminology in  
Markov Decision Process

Agent : Something that can be directly controlled

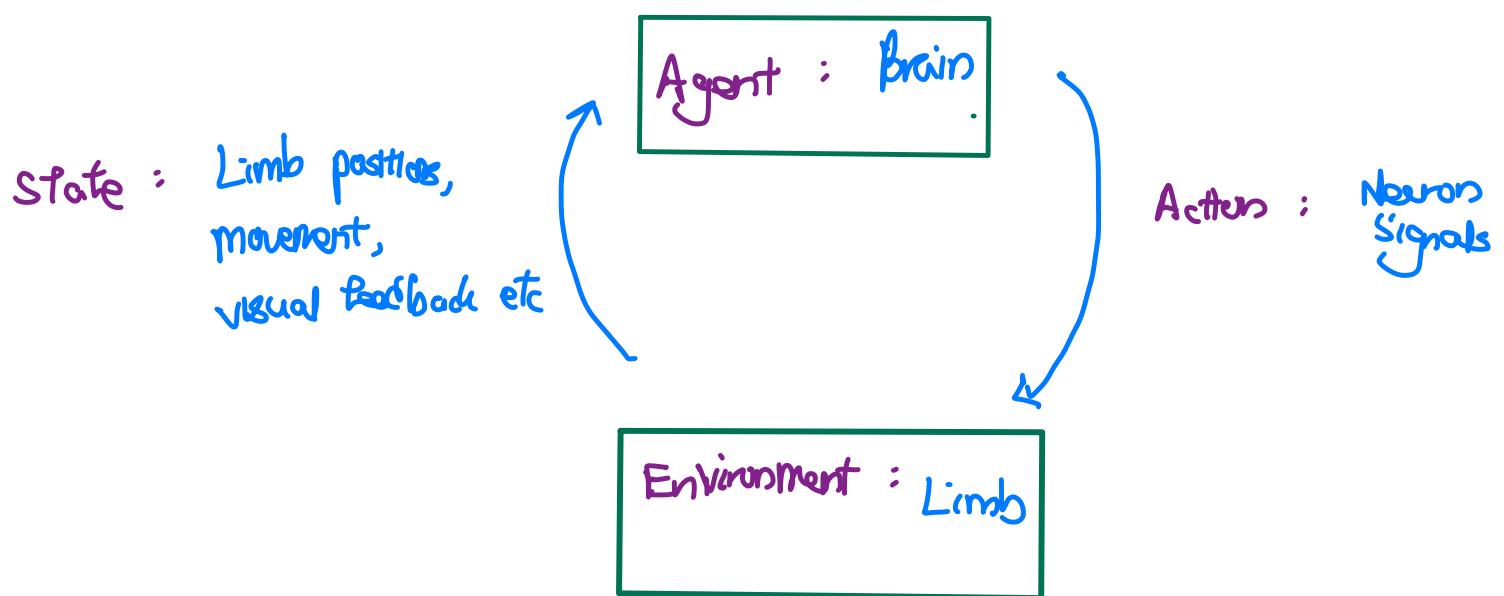
Environment : Something that cannot be controlled,  
but can be interacted with through agent



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

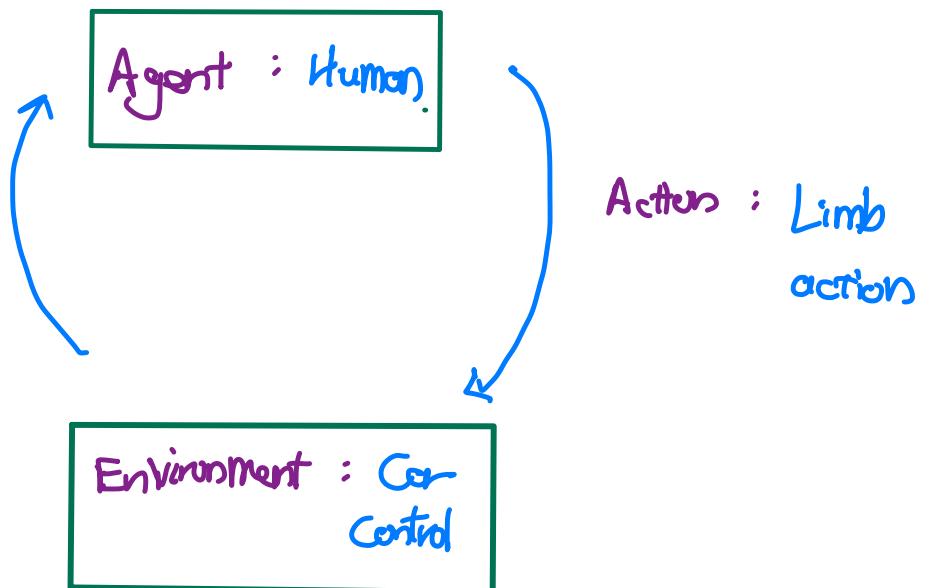
For instance, in achieving a task of controlling a  
human to drive a car, it can be separated into  
a few programs

Program A : Control limb movement



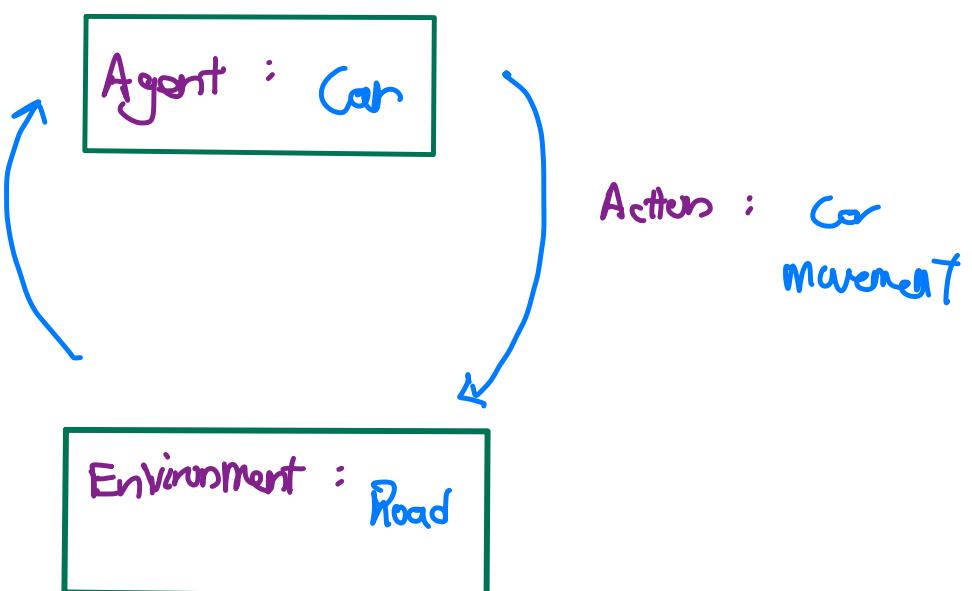
Program B : Control interaction with the vehicle

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 !

Q - Learning

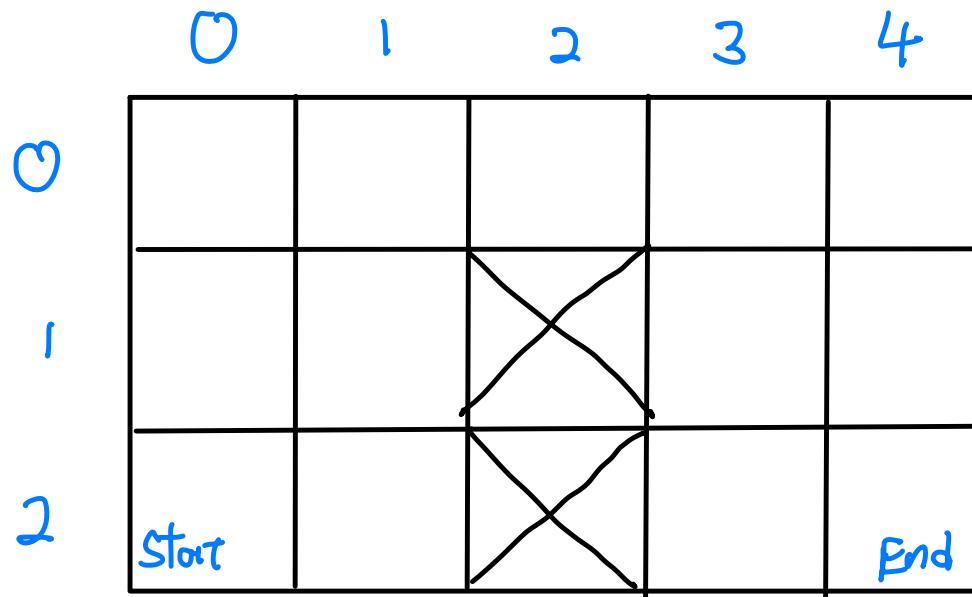
Chapter 2 :

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 :

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

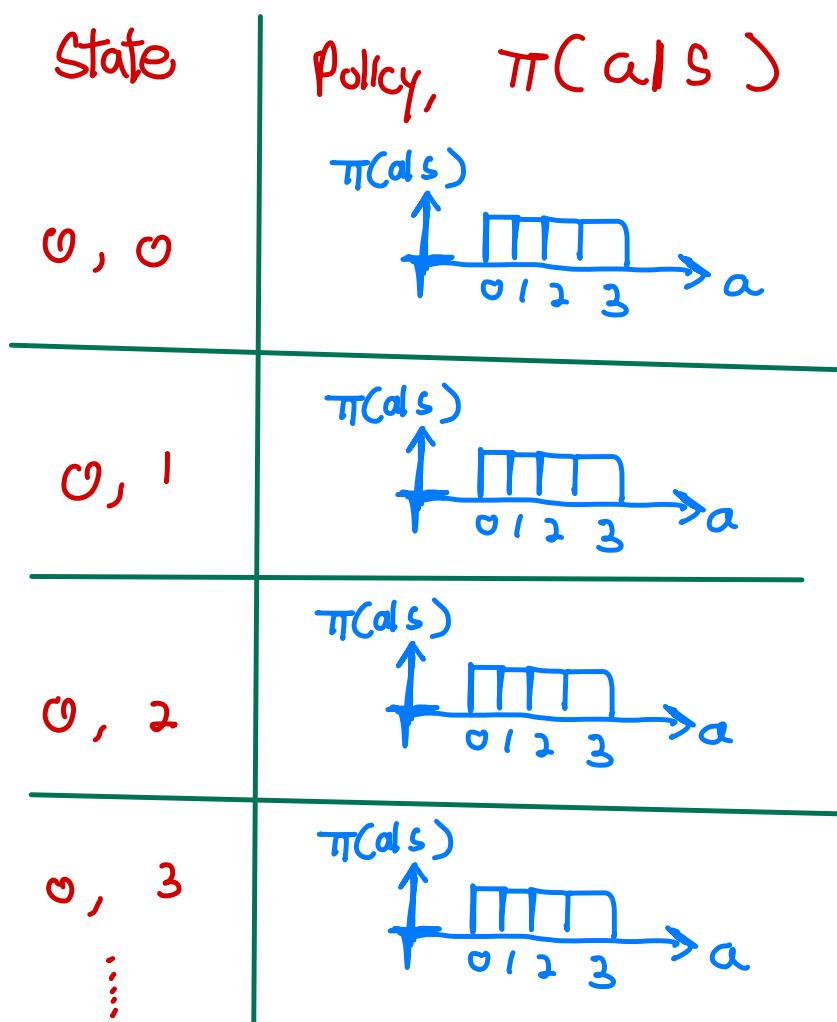
→ the subsequent state & reward given the previous state & actions

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

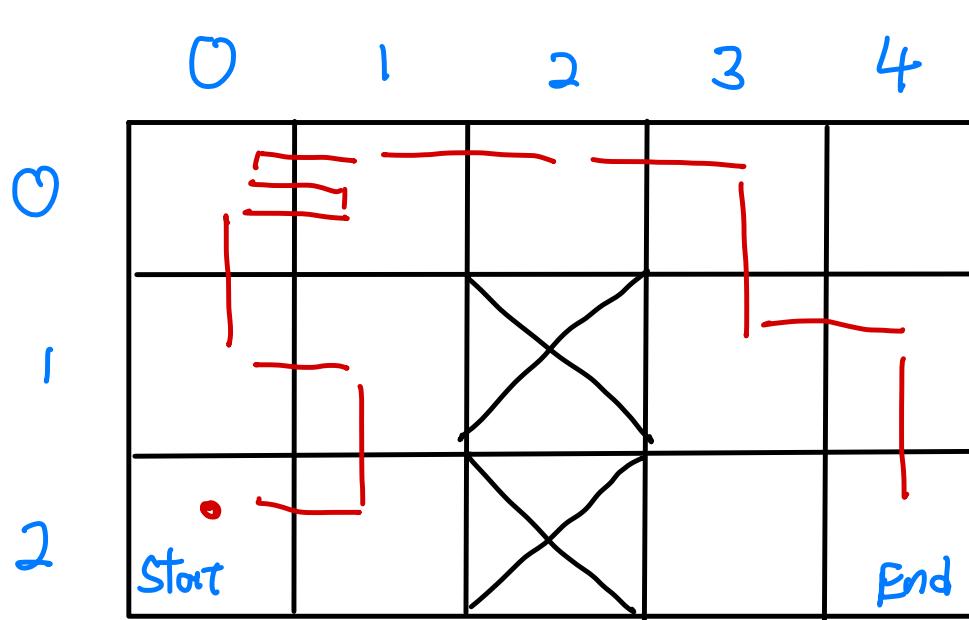
Because of the lack of world model access



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 :

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

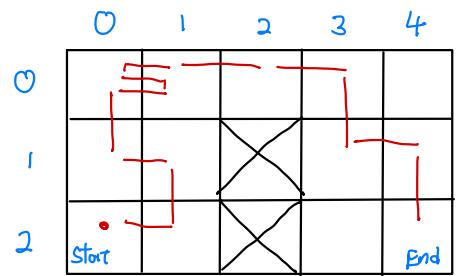
	0	1	2	3
0	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : -0.8843	3 : 0	3 : 0
0	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	3 : 0	3 : 0	3 : 0	3 : 0
0	0 : 0	0 : 0	0 : 0	0 : 0
1	1 : 0	1 : 0	1 : 0	1 : 0
2	2 : 0	2 : 0	2 : 0	2 : 0
3	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
		-0.8843		
	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
1	0 : 0	0 : 0	0 : 0	0 : 0
2	1 : 0	1 : 0	1 : 0	1 : 0
3	2 : 0	2 : 0	2 : 0	2 : 0
4	3 : 0	3 : 0	3 : 0	3 : 0
2	0 : 0	0 : 0	0 : 0	0 : 0
3	1 : 0	1 : 0	1 : 0	1 : 0
4	2 : 0	2 : 0	2 : 0	2 : 0
5	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

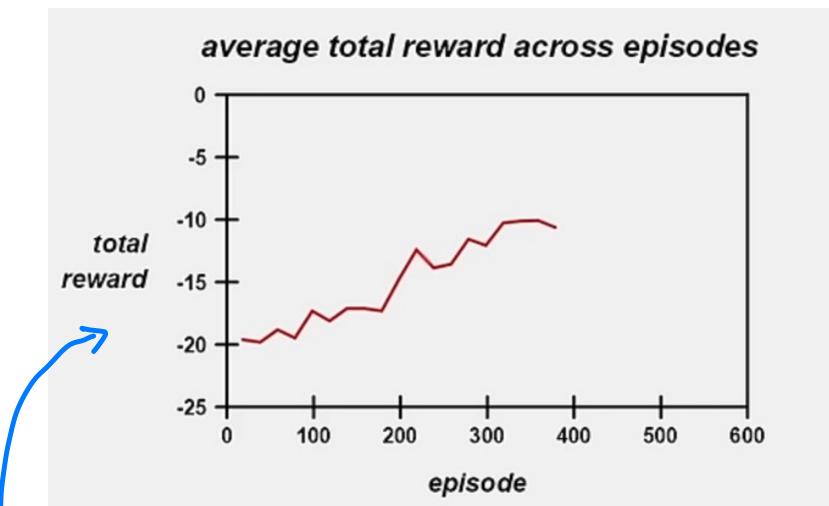
- Compute return (discounted)

→ 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

# Mathematical Proof

proof that mean can be calculated incrementally

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

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

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

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

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

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



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

Q-Learning

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

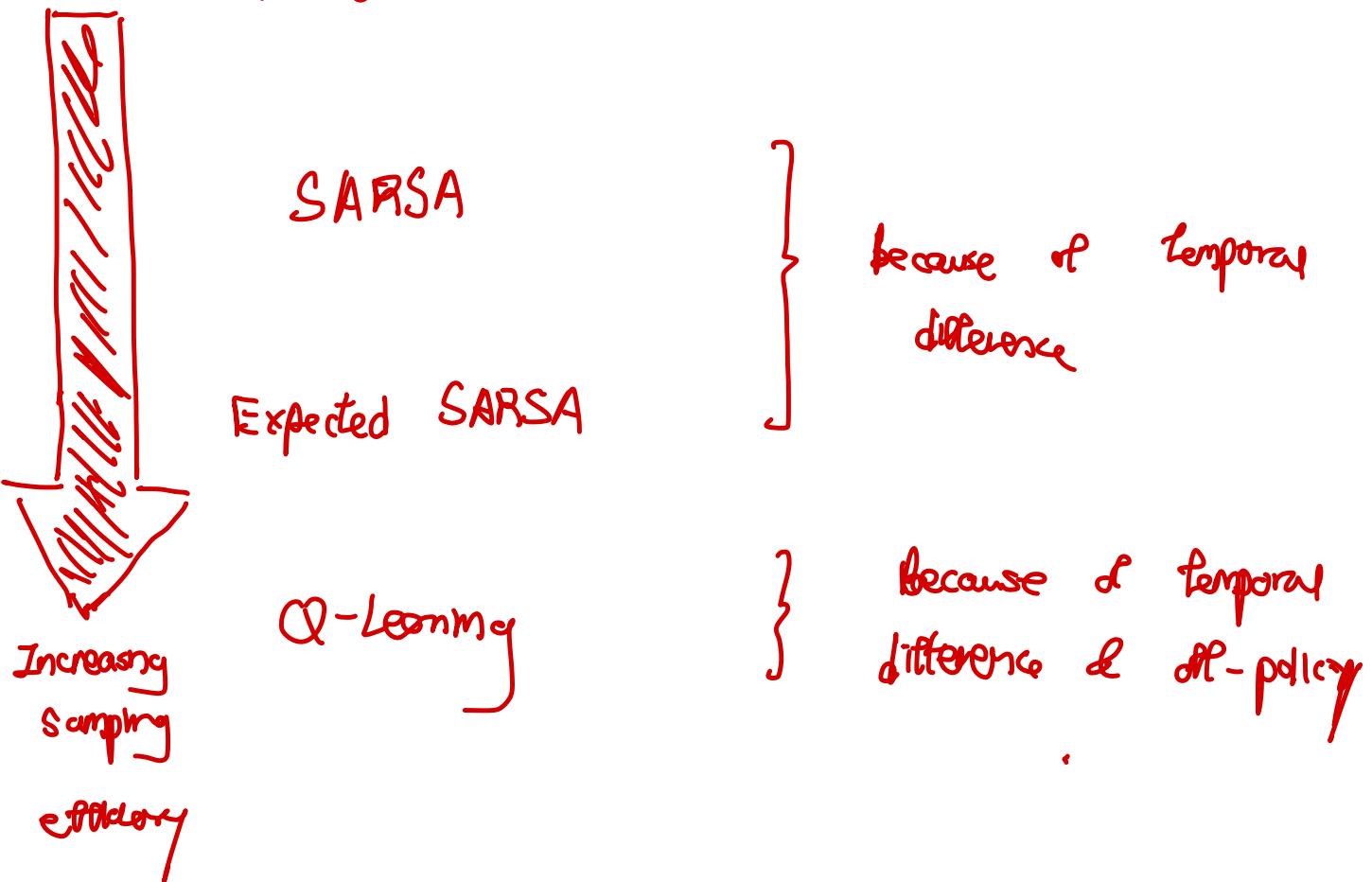
because

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

- b) behaviour policy  $\neq$  target policy

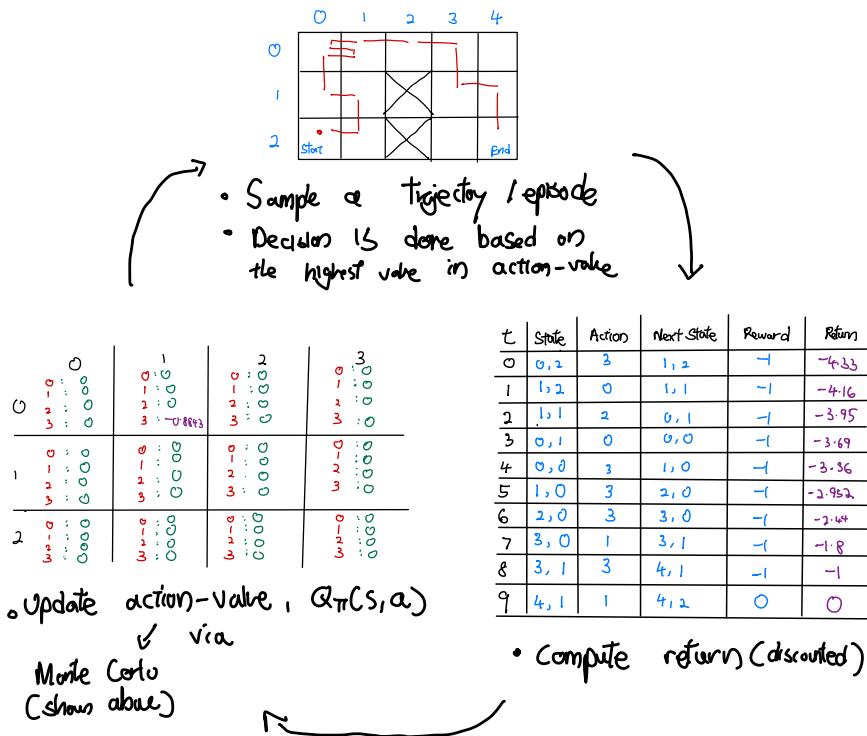
Sample Efficiency - Number of episodes required to get good at a task

## Monte Carlo



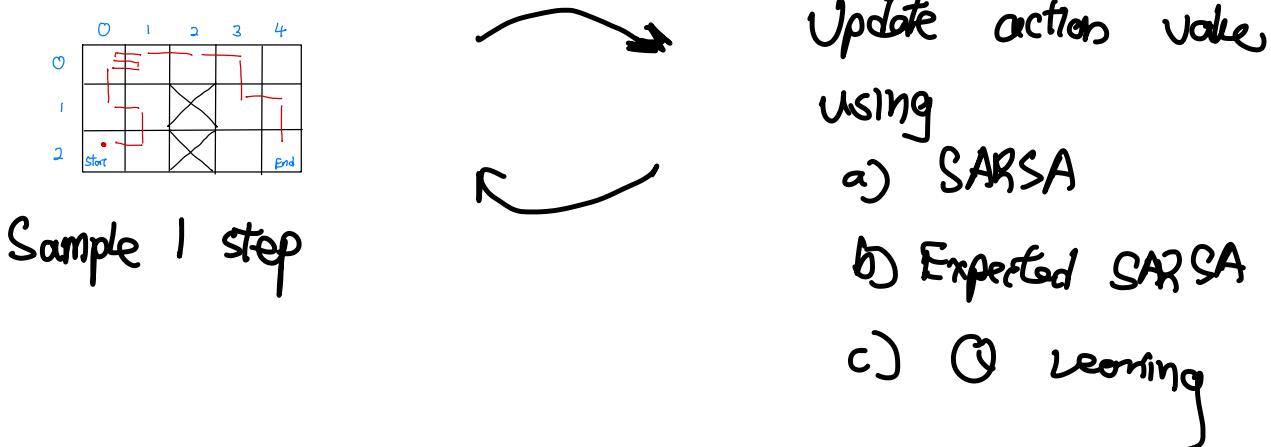
In Monte Carlo :

→ 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

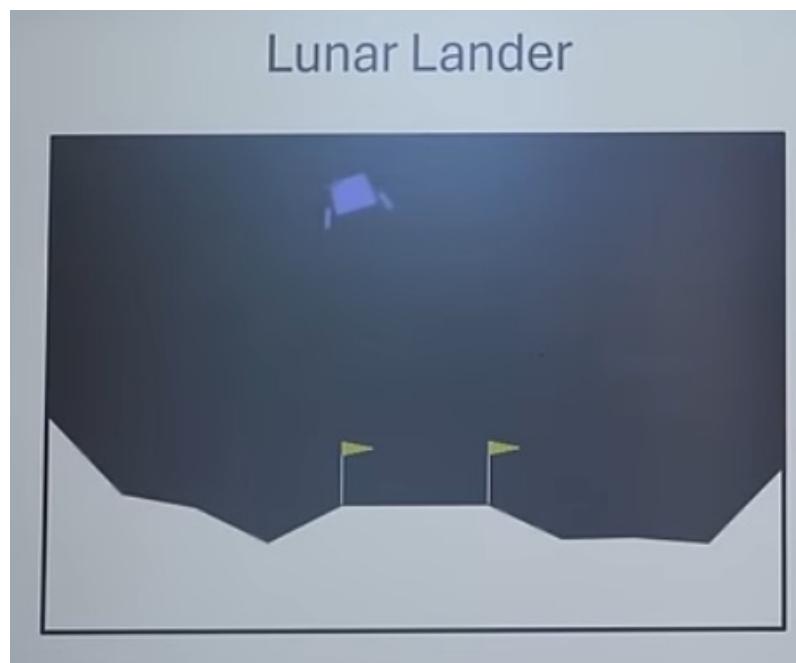
Deep Q Learning basically replaces the action value table used in Monte Carlo / Temporal Difference with a Deep Learning Network.

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 ,  $w$

Continuous

Left Leg in contact with Ground , L

Right Leg in contact with Ground , R

Discrete

/ Buttons

## Action (Discrete)

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

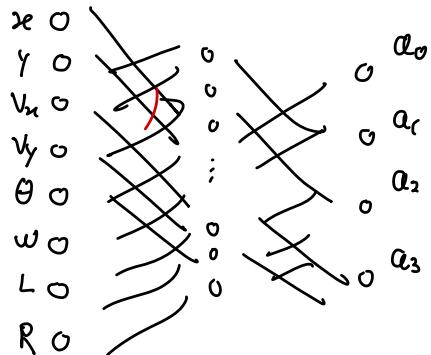
Reward :

- 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.

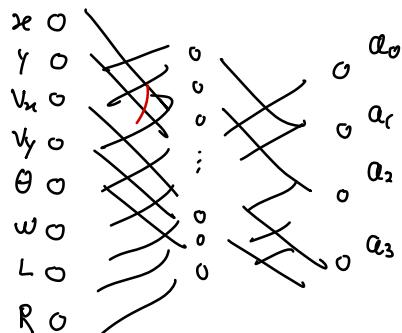
How is Deep Q Learning applied here?

## Initialization



$\theta$   
Main Network

→ use to decide  
actions to be  
taken during sampling

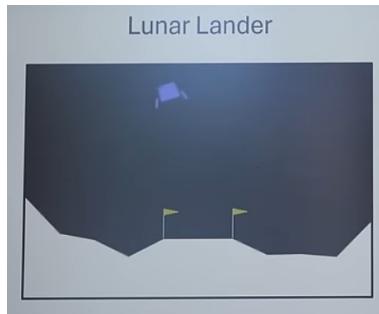


$\theta^-$   
Target Network

→ 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 ( $s$ )

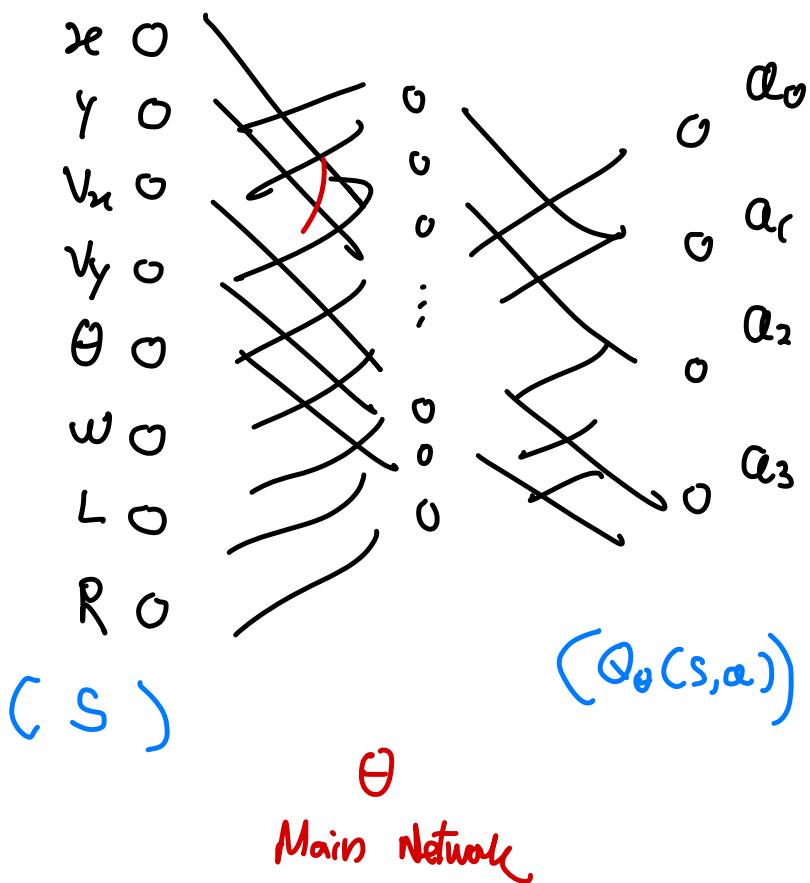
$$s = s'$$



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

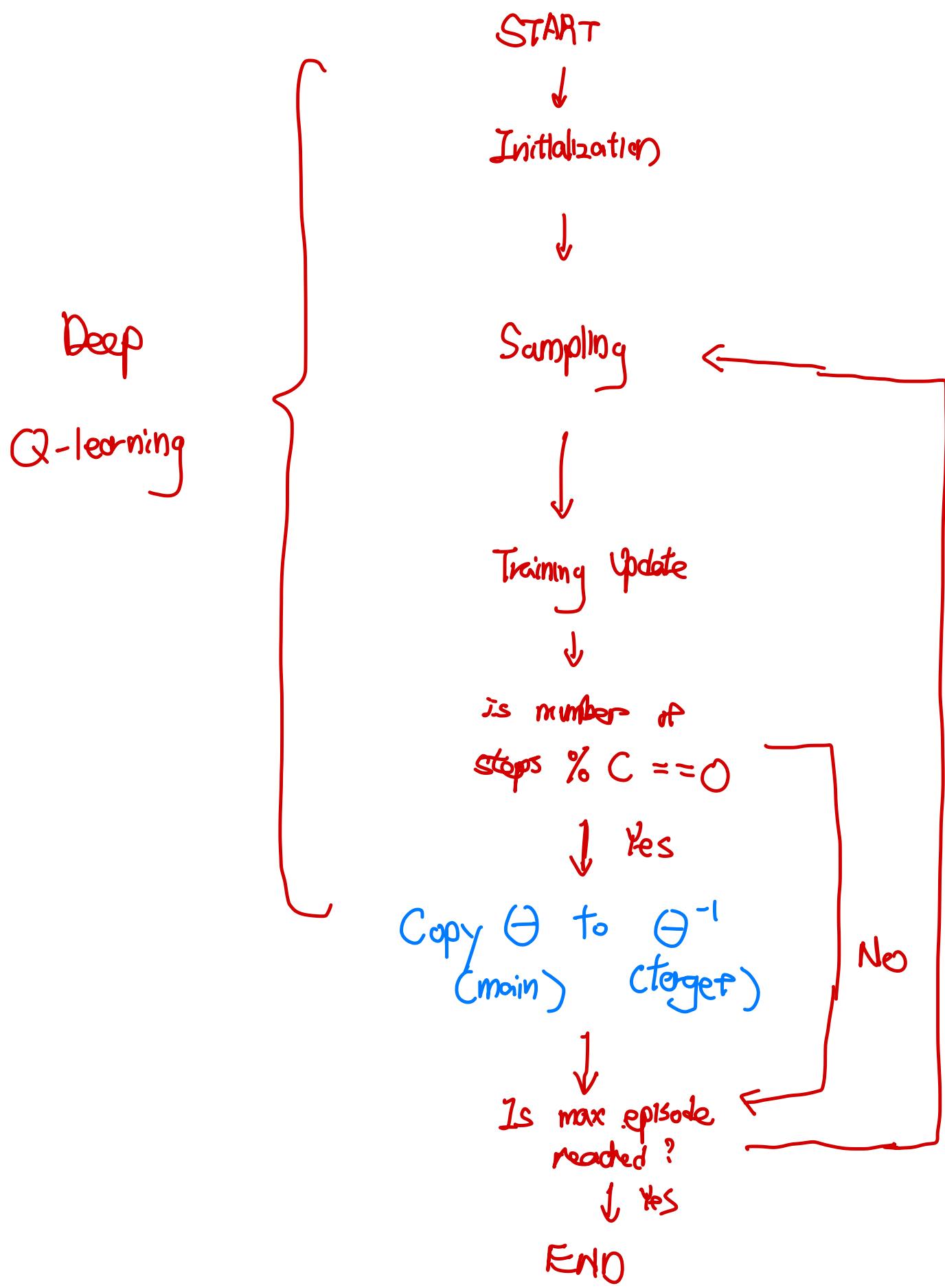
- 1) Pass next state,  $s'$  to the target network,  $\theta^{-1}$
- 2) To get a list of action value  $Q_{\theta^{-1}}(s', a')$
- 3) From these action values, pick the one that is maximum ( $\max_{a'}$ )

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)

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

- 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

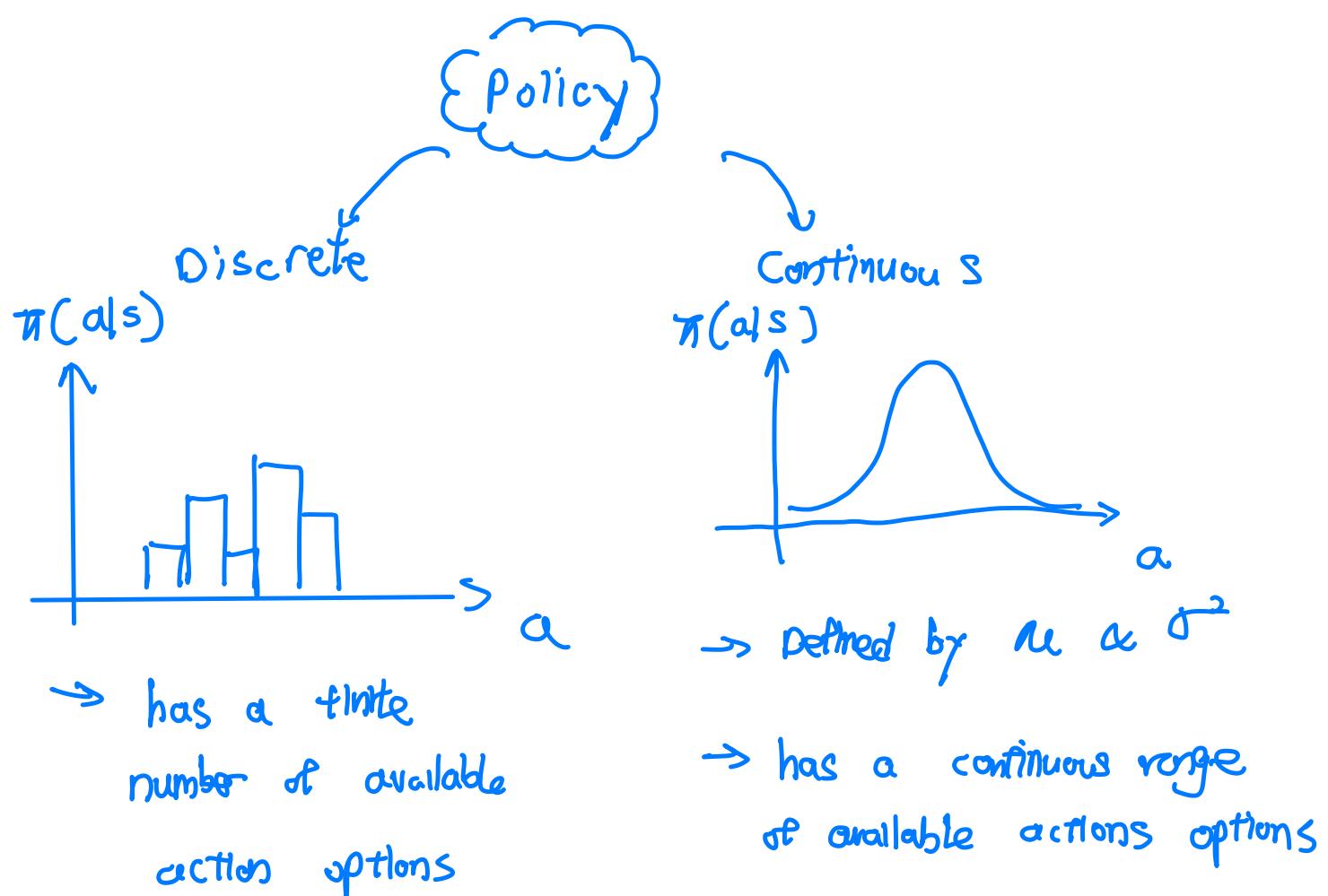
Chapter 5 :

Policy  
Gradient

Reinforce

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

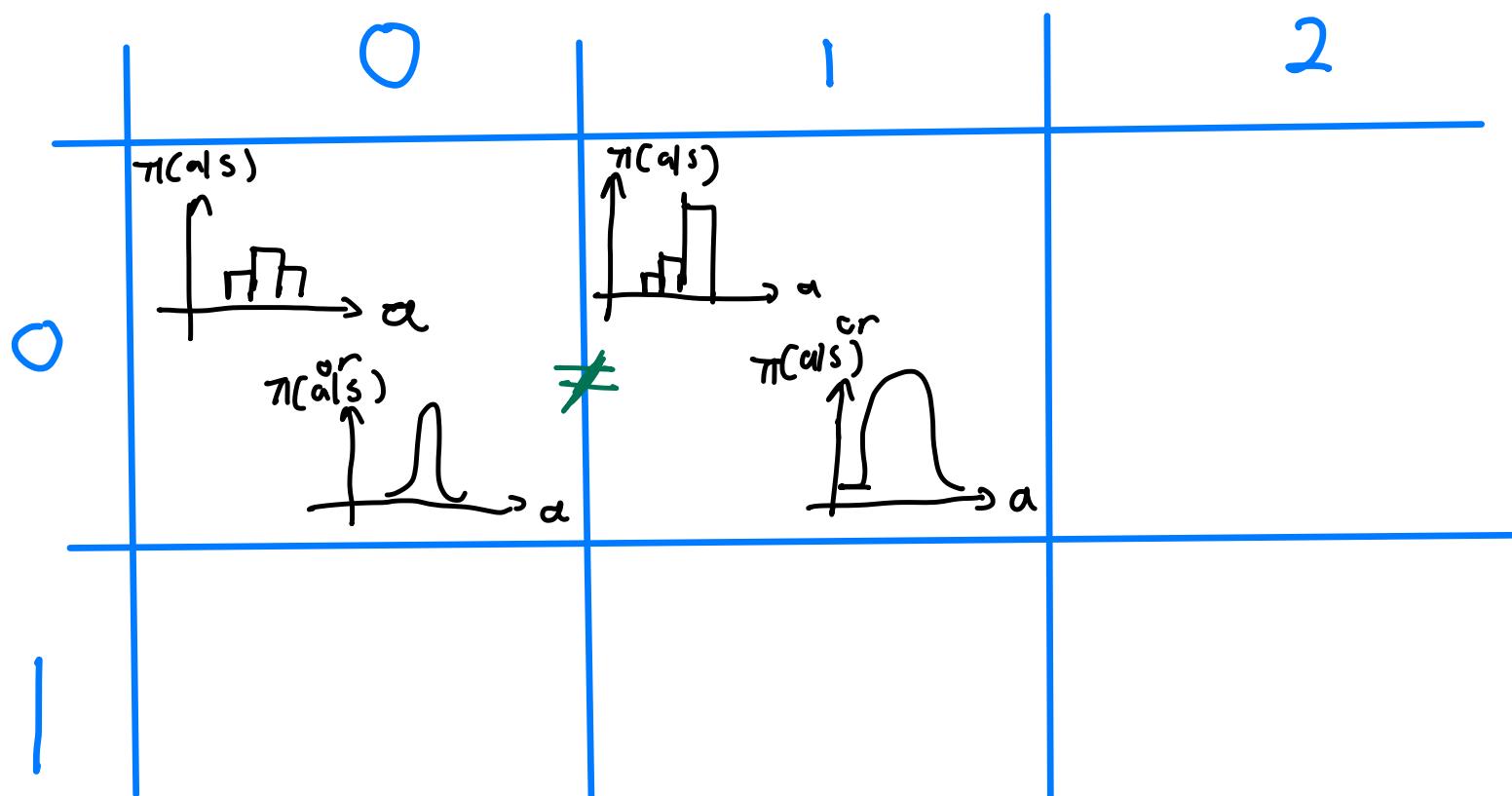
Now, let's talk about policy based, which offer some randomness naturally for exploration  
(Unlike action-value based that depends on  $\epsilon$  for exploration)



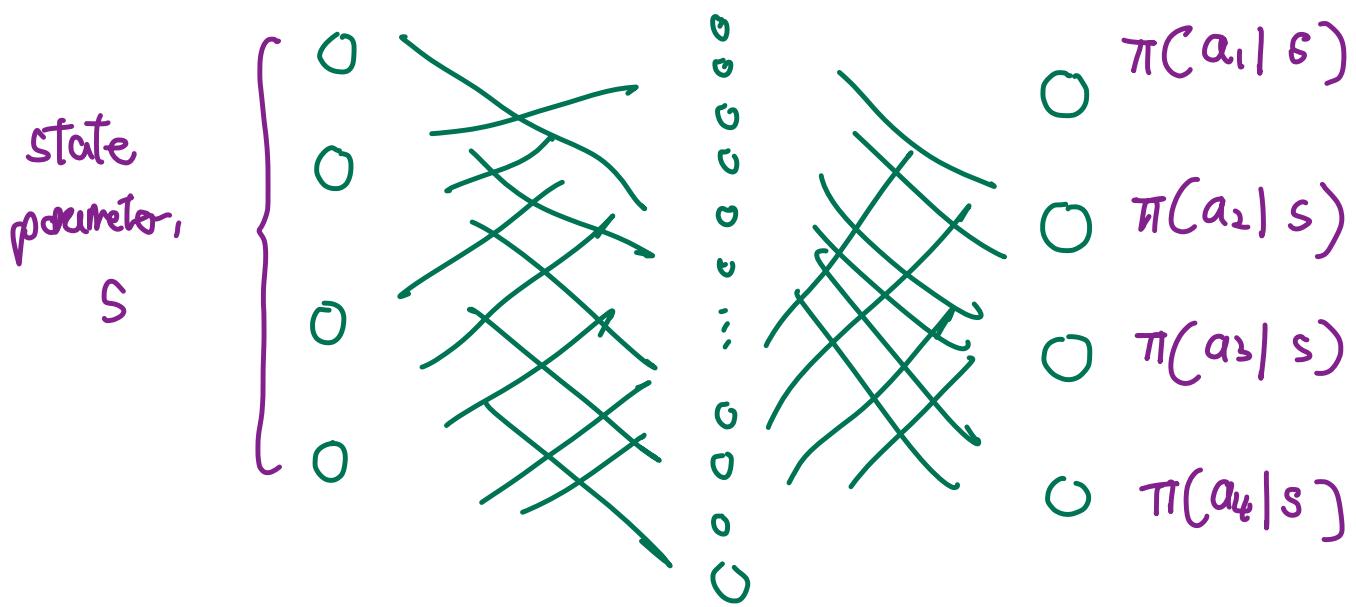
## Some key concept :

- 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

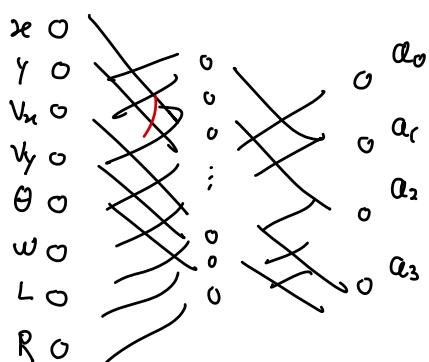


Policy network,  $\pi(a|s)$



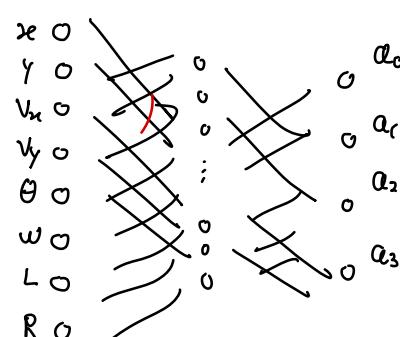
But how do you train this policy network?

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



$\theta$   
Main Network

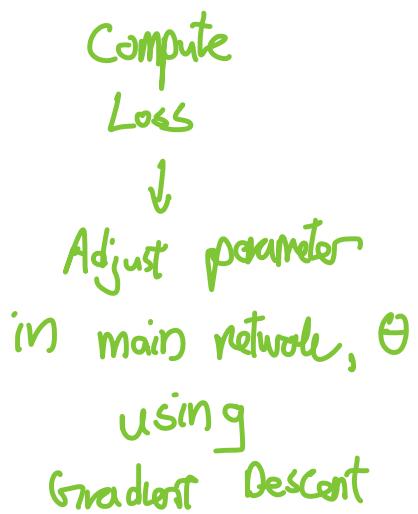
$$Q(s, a)$$



$\theta^*$   
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 ← ↗ 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

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

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

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

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

For a more detailed breakdown,

$$J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{E}[R(\tau)]$$

$J(\pi_\theta)$  : The objective function,  $J$  scored by following a policy network  $\pi$  with a weight of  $\Theta$

$\tau$  : A trajectory / episode

$\tau \sim \pi_\theta$  : All possible trajectories by following the policy, provided by the policy network  $\pi$  with a weight of  $\Theta$

$R(\tau)$  : Cumulative discounted return. by following a trajectory,  $\tau$

$E_{\tau \sim \pi_\theta}[R(\tau)]$  : Expected value of the cumulative discounted return,  $R$  across all possible trajectories,  $\tau$  offered by the policy network,  $\pi$  with a weight of  $\Theta$

$$R(\tau) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \dots \dots$$

$\gamma$  : A discount factor

Further break down,

$$J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} [R(\tau)]$$

$$= \sum_{\tau} P(\tau; \theta) R(\tau)$$

$P(\tau; \theta)$ : probability of this trajectory,  $\tau$  being taken under a policy network with a weight of  $\theta$

$$P(\tau; \theta) = \prod_{t=0} T P(S_{t+1} | S_t, A_t) \pi_\theta(A_t | S_t)$$

$\pi_\theta(A_t | S_t)$ : probability of the policy network,  $\pi$  with a weight of  $\theta$  in asking the agent to take the action  $A$  at time step  $t$  provided an input of state  $S$  at the step  $t$

$P(S_{t+1} | S_t, A_t)$ : Environment dynamic. The probability of the agent ending in a state  $S$ , in the subsequent time step,  $t+1$  given that the agent took action  $A$  at the step  $t$  when being in state  $S$ , at the step  $t$ .

In full breakdown

$$\begin{aligned} J(\pi_\theta) &= \underset{\tau \sim \pi_\theta}{E[R(\tau)]} \\ &= \sum_{\tau} P(\tau; \theta) R(\tau) \\ &= \sum_{\tau} \left[ \pi \left( p(s_{t+1} | s_t, a_t) \pi_\theta(a_t | s_t) \right) \cdot R(\tau) \right] \end{aligned}$$

The ultimate goal :

$$\max_{\theta} J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{E[R(\tau)]}$$

⚠ However, there are a few issues with this objective function

① Computationally expensive  
- to find all possible trajectories

②  $p(s_{t+1} | s_t, a_t)$  is state dynamics that is attached to the environment. It may not be differentiable given that we may not have access to it.

💡 Solution - Refer to POLICY-GRADIENT THEOREM

Before that, let's first introduce some other variations ...

②  $J(\pi_0) = V_{\pi}(s_0)$

- 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,

$$V(s_t) \xrightarrow{\text{Approach}} r_t + \gamma V(s_{t+1})$$

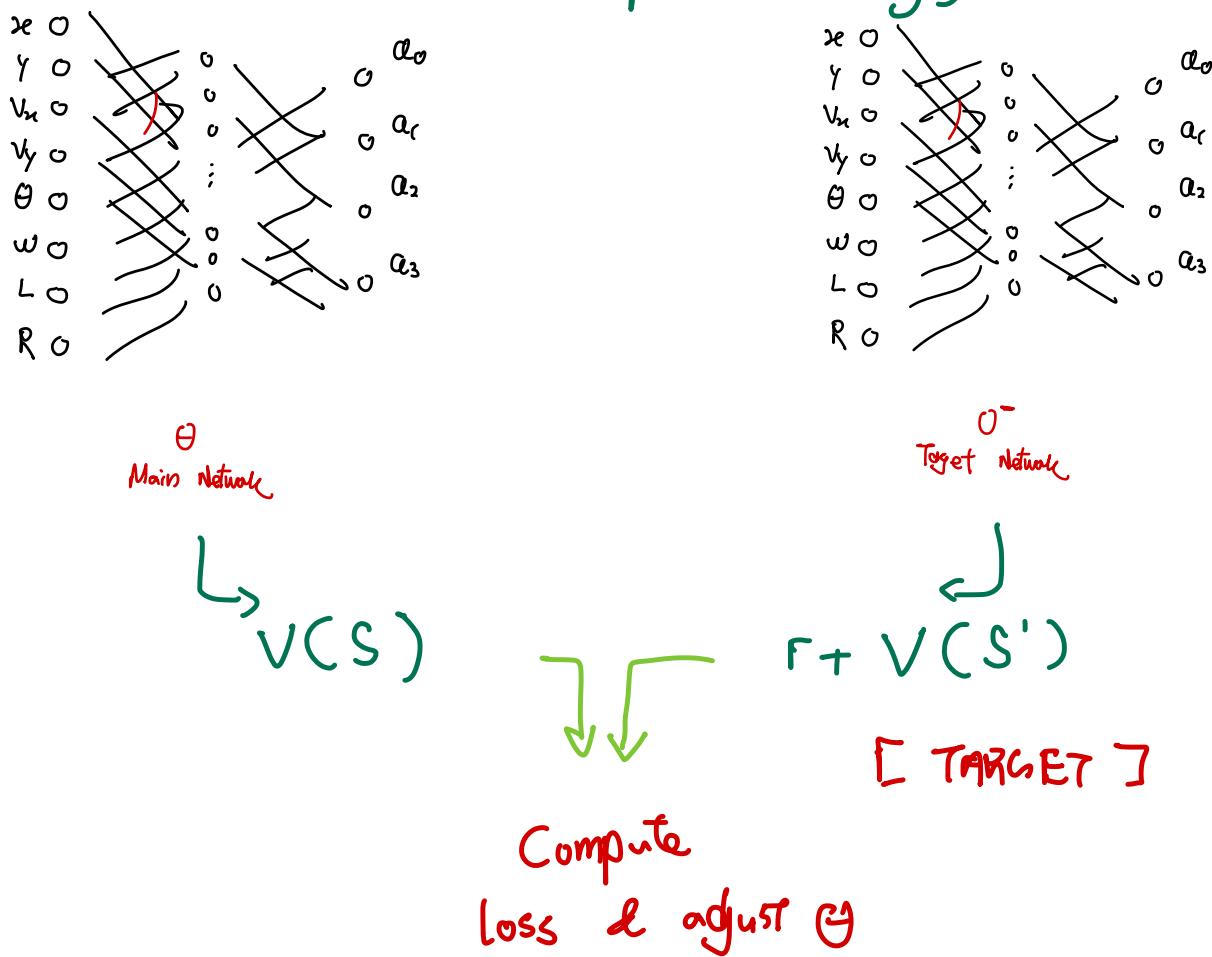
$$V(s_t) \xleftarrow{\text{Assign}} V(s_t)$$

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

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

For a single trajectory,

$$\nabla_\theta J(\pi_\theta) = \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t | s_t) \cdot \bar{\Psi}_t$$

For multiple trajectories,

$$\nabla_\theta J(\pi_\theta) = \frac{1}{m} \sum_{i=1}^m \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) \cdot \bar{\Psi}_t^{(i)}$$

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

$E\left[\sum_{t=0}^{\infty}\right]$  : Expected value / Mean  
of 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

$\bar{A}_t$

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



Can be quantified by

p/s : a) & c) are not advantage as no baseline  
is introduced

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 (discussed further in

Chapter 11 : Bias vs 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)  $Q_\pi(S_t, A_t)$

✓ 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  $\begin{array}{l} \xrightarrow{\text{policy}} \\ \xrightarrow{Q} \\ \xrightarrow{V} \end{array}$

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

$\underbrace{r_t + \gamma V_{\pi}(s_{t+1})}_{\text{Target } \checkmark} - \underbrace{V_{\pi}(s_t)}_{\text{Current } \checkmark} \quad \left. \right\} \quad \begin{array}{l} \text{Conventionally,} \\ \text{this is} \\ \text{usually referred} \\ \text{to as} \\ \text{advantage} \end{array}$

✓ 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  $\begin{array}{l} \xrightarrow{\text{update}} \text{policy network through} \\ \text{advantage (critic)} \\ \xrightarrow{\text{update}} \text{value network (actor)} \end{array}$

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

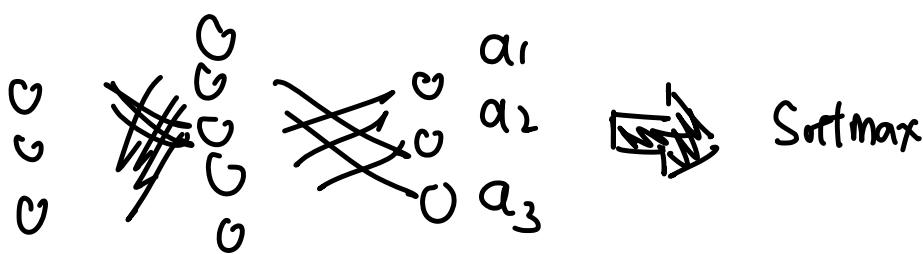
If positive



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



# Derivation of Derivative of PERFORMANCE OBJECTIVE

$$\begin{aligned} J(\pi_\theta) &= \underset{\tau \sim \pi_\theta}{E}[R(\tau)] \\ &= \sum_{\tau} P(\tau; \theta) R(\tau) \\ &= \sum_{\tau} \left[ \left( \prod_{t=0}^{\infty} P(S_{t+1} | S_t, a_t) \pi_\theta(a_t | S_t) \right) (R(\tau)) \right] \end{aligned}$$

↓  
POLICY GRADIENT  
THEOREM

## DERIVATIVE OF PERFORMANCE OBJECTIVE

$$\nabla_\theta J(\pi_\theta) = \sum_{t=0}^H \nabla_\theta \log \pi_\theta(a_t | S_t) \cdot R(\tau)$$

Note : For simplicity, assuming  $\bar{R}_t = R(\tau)$

what happens in between ..

$$J(\pi_\theta) = \sum_{\tau} P(\tau; \theta) R(\tau)$$

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \sum_{\tau} P(\tau; \theta) R(\tau)$$

$$\nabla_\theta J(\pi_\theta) = \sum_{\tau} (\nabla_\theta P(\tau; \theta)) (R(\tau))$$

$$\nabla_\theta J(\pi_\theta) = \sum_{\tau} \left( \frac{P(\tau; \theta)}{P(\tau; \theta)} \right) (\nabla_\theta P(\tau; \theta)) (R(\tau))$$

$$\nabla_\theta J(\pi_\theta) = \sum_{\tau} (P(\tau; \theta)) \left( \frac{\nabla_\theta P(\tau; \theta)}{P(\tau; \theta)} \right) (R(\tau))$$

Based on Derivative log trick or known as REINFORCE trick,

$$\nabla_x \log f(x) = \frac{\nabla_x f(x)}{f(x)}$$

$$\nabla_{\theta} J(\pi_{\theta}) = \sum_{\tau} \left( P(\tau; \theta) \right) \left( \nabla_{\theta} \log (P(\tau; \theta)) \right) (R(\tau)) - E_g \textcircled{1}$$

From Equation ①, extract  $\nabla_{\theta} \log (P(\tau; \theta))$

$$\nabla_{\theta} \log (P(\tau; \theta))$$

Note :  $\pi(s_0)$  is the initial state distribution

$$\begin{aligned} &= \nabla_{\theta} \log (\pi(s_0) \cdot \prod_{t=0}^H P(s_{t+1} | s_t, a_t) \pi_{\theta}(a_t | s_t)) \\ &= \nabla_{\theta} \left( \log \pi(s_0) + \log \prod_{t=0}^H P(s_{t+1} | s_t, a_t) + \log \prod_{t=0}^H \pi_{\theta}(a_t | s_t) \right) \\ &= \nabla_{\theta} \left( \underbrace{\log \pi(s_0)}_{0} + \sum_{t=0}^H \log (P(s_{t+1} | s_t, a_t)) + \sum_{t=0}^H \log \pi_{\theta}(a_t | s_t) \right) \\ &= 0 + 0 + \nabla_{\theta} \sum_{t=0}^H \log \pi_{\theta}(a_t | s_t) \\ &= \nabla_{\theta} \sum_{t=0}^H \log \pi_{\theta}(a_t | s_t) \\ &= \sum_{t=0}^H \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) - E_g \textcircled{2} \end{aligned}$$

Both terms are not derived with respect to  $\theta$

Substitute Eq② into Eq① :

$$\mathbb{V}_\theta J(\pi_\theta) = \sum_{\tau} \left( P(\tau; \theta) \right) \left( \sum_{t=0}^H \gamma_\theta \log \pi_\theta(a_t | s_t) \right) (R(\tau))$$

$P(\tau; \theta)$  can be estimated by sampling multiple trajectories...

$$\mathbb{V}_\theta J(\pi_\theta) = \frac{1}{m} \sum_{i=0}^m \sum_{t=0}^H \gamma_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) \cdot R(\tau^{(i)})$$

# ACTUAL IMPLEMENTATION OF DERIVATIVE OF PERFORMANCE OBJECTIVE

$$\nabla_{\theta} J(\pi_{\theta}) = -\frac{1}{m} \sum_{i=0}^m \sum_{t=0}^H \nabla_{\theta} \ln \pi_{\theta}(a_t^{(i)} | s_t^{(i)}) \cdot G_t$$

In actual implementation, a few modifications were made.

(1)  $R(\tau^{(i)})$  is replaced by  $G_t$

$R(\tau^{(i)}) \rightarrow$  discounted return for the entire episode, starting from  $t=0$

$G_t \rightarrow$  discounted return starting from  $t=t$  (The current time step of this action)

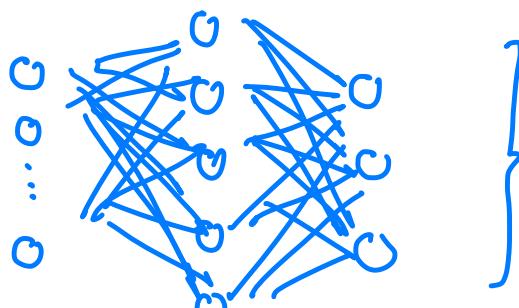
(2) In PyTorch, it is easier in implementing to minimize something rather than maximizing it.

Therefore, a negative sign is added

(3) using  $\ln$  instead of  $\log$

# Design of policy network to use Reinforce

## a) discrete space

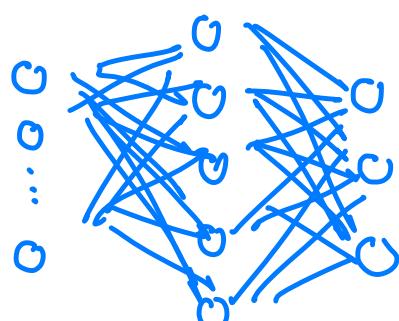


Each output node represents probability of taking a discrete action

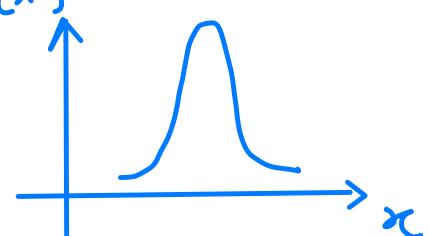
can be applied into easily

$$J_{\theta}[\pi_{\theta}] = - \frac{1}{m} \sum_{i=0}^m \sum_{t=0}^H \nabla_{\theta} \ln \pi_{\theta}(a_t^{(i)} | s_t^{(i)}) \cdot G_t$$

## b) Continuous Space



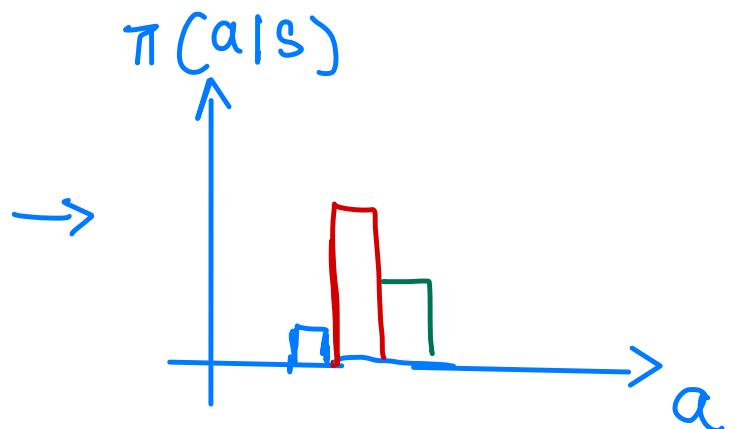
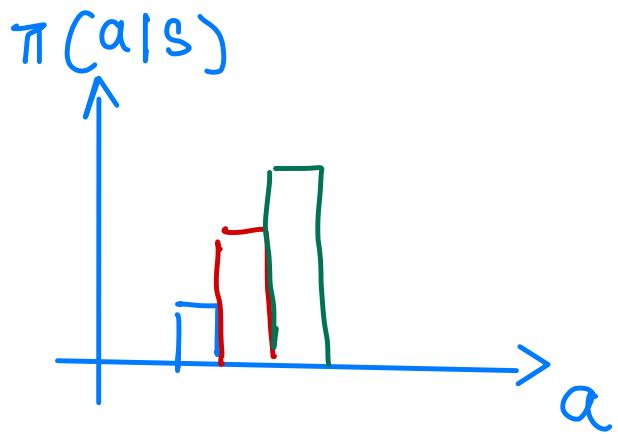
Each pair of output nodes represent mean & std-dev of a Gaussian Probability distrib for an continuous action



- When taking an action, you sample it from this distribution

- Based on the distribution, the probability of the action can also be determined

## Visualization of policy update in discrete actions

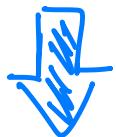
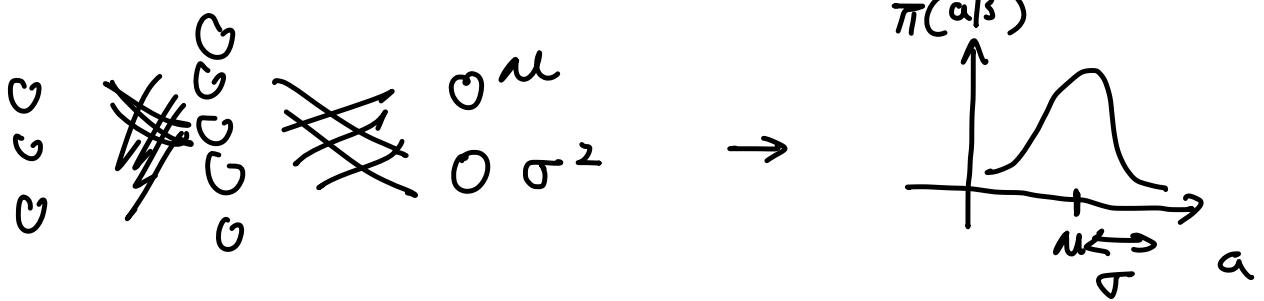


- Due to the evaluation made by the advantage,  $\Psi$ ,  $a_2$  is deemed a good action, receive a high  $\Delta J(\pi_\theta)$  where the parameter in the policy network are tuned in such a way that leads to higher  $\pi(a_2|s)$ .
- Because the policy neural network's output will go through softmax, when  $\pi(a_2|s)$ ,  $\pi(a_1|s)$  &  $\pi(a_3|s)$  decrease.

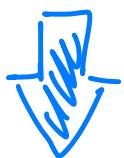
\*  $a_2 \rightarrow$  get positive advantage

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

When policy network is to account for continuous actions



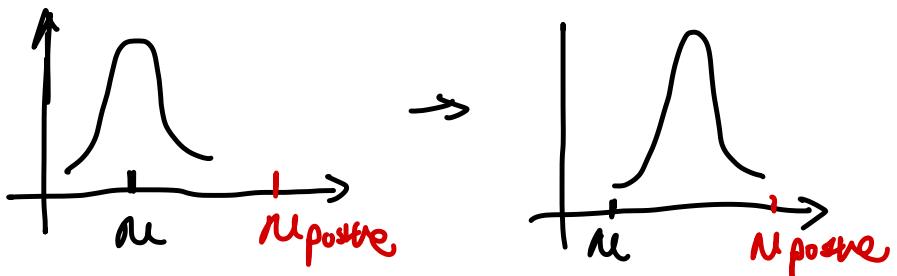
- Compute  $\Delta_\theta J(\pi_\theta)$
- Update policy network via Gradient Descent



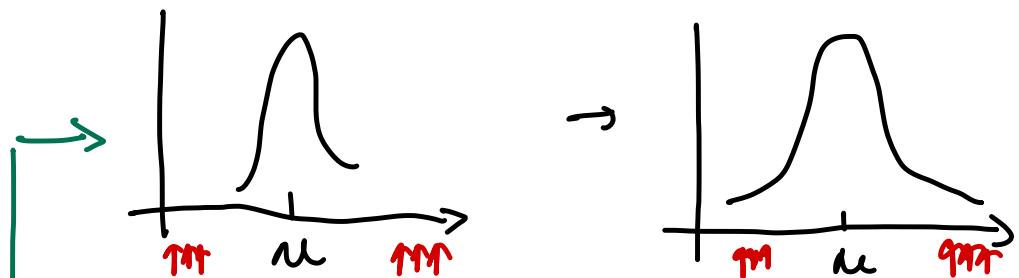
Visualization of policy update in continuous actions

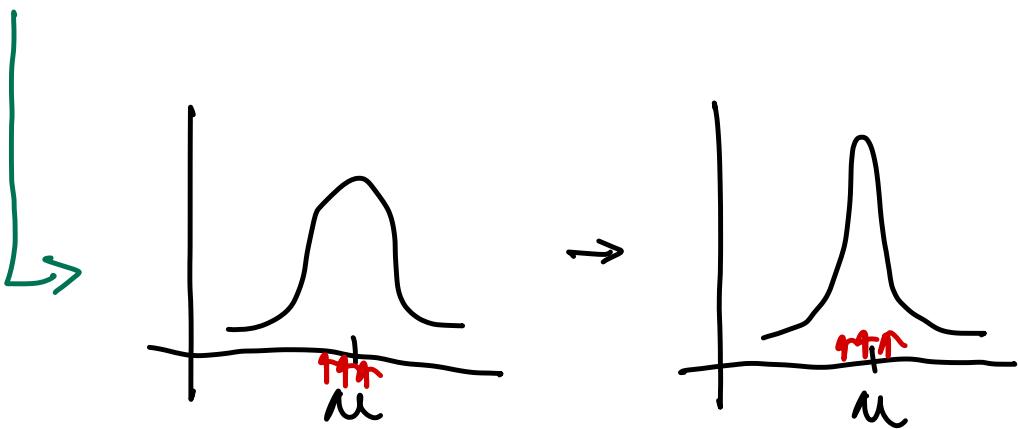
when get positive advantage

Adjust mean  
to be closer



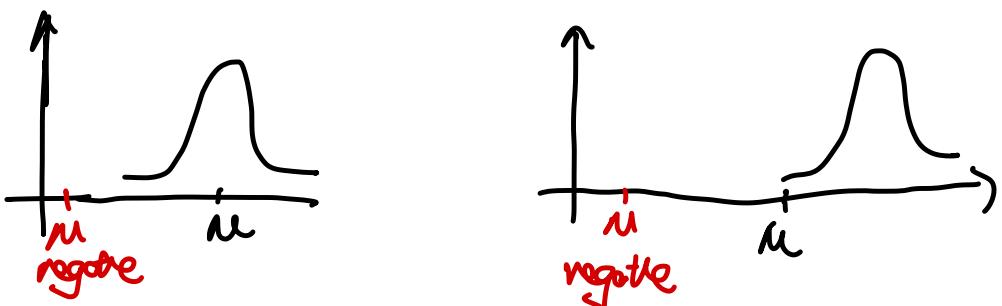
Adjust  $\sigma^2$   
to be closer



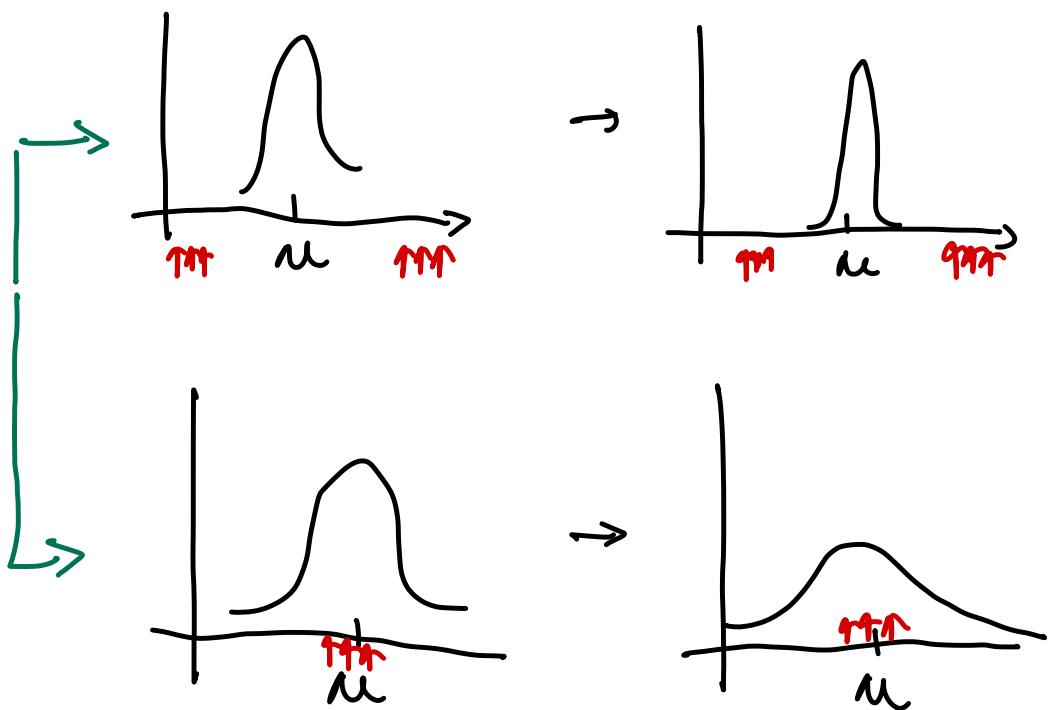


When get negative advantage

adjust mean  
to be further



adjust  $\sigma^2$   
to be  
further

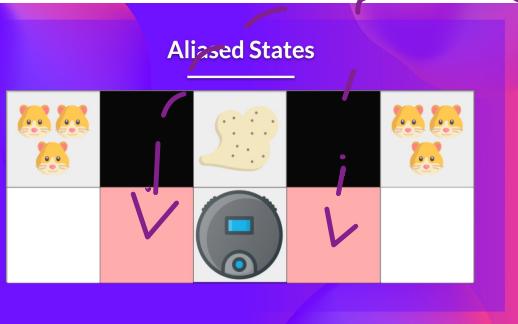


# Advantage of Policy Gradient over Deep Q Learning

i) Can learn a stochastic policy

- No longer need to compute epsilon-greedy policy manually

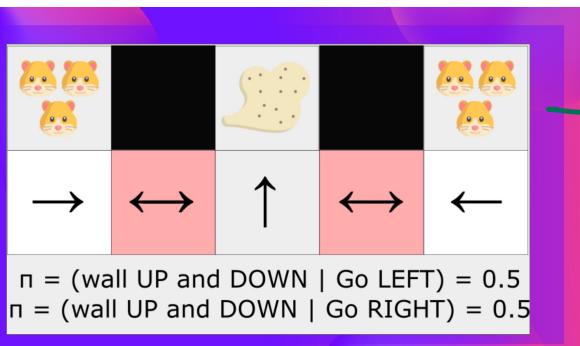
- Get rid of perceptual aliasing



When agent is in a state, it might be observing the same observation state. But on the grand scale, it is not

i) If using a deterministic policy, will always take the same direction.

ii) If using quasi-deterministic policy (epsilon-greedy), will take a long time to find the most reward



iii) Policy Gradient outputs a stochastic policy & therefore can find the most reward faster

2) More efficient for continuous action space

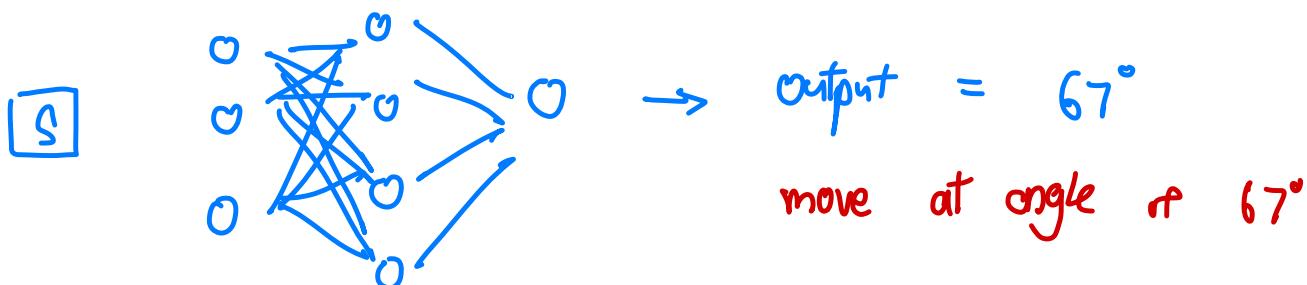
In Deep Q learning.

$$\text{Loss} = \text{Loss\_fn}\left(r_t + \max_{a_{t+1}} Q_{\theta^*}(s_{t+1}, a_{t+1}) - Q_{\theta}(s_t, a_t)\right)$$

↳ need to apply ↓  
onto every possible action  
in every single state

! highly inefficient in continuous action space !

In Policy Gradient



3) Smoother convergence probability

because the action space is in continuous, the type of actions taken by the agent will also shift more gradually, allowing smoother convergence

Chapter 6 :

Actor - Critic

Advantage Actor Critic

Actor-Critic is a combination of

- Q-Learning / Deep Q Learning

→ Involving a **Critic** that uses action value or state value to evaluate the quality of actions taken by the agent

→ However, the benchmark to evaluate is not necessarily always state / action value. It can also be using the discounted return as discussed in reinforcement algorithm.

- Policy Gradient

→ Involving an **actor** that samples its action through the assistance of a policy

before diving deeper into this topic, it is advisable to refresh on Chapter 5 : Policy Gradient where an algorithm named Reinforce was extensively discussed

Policy Gradient is a form of Actor - Critic

The ultimate goal :

$$\max_{\Theta} J(\pi_{\Theta}) = \underset{\tau \sim \pi_{\Theta}}{E}[R(\tau)]$$

$$\nabla_{\Theta} J(\pi_{\Theta}) = \sum_{t=0}^H \nabla \log \pi_{\Theta}(a_t | s_t) \cdot \underline{R(\tau)}$$

which can  
be generalised  
into



↳ discounted return starting from this state,  $s_t$   
- also written as  $g_t(\tau)$

$$\nabla_{\Theta} J(\pi_{\Theta}) = \sum_{t=0}^H \nabla \log \pi_{\Theta}(a_t | s_t) \cdot \underline{\Psi_t}$$



Advantage

• Advantage of taking this action in this stage compared to other actions

As displayed above,  $\bar{q}_t = R(\tau) + g_t(\tau)$   
in reinforce algorithm.

There are many options for  $\bar{q}_t$  (Advantage), let's evaluate the possible options :

p/s : a) & c) are not advantage as no baseline is introduced

a)  $R(\tau)/g_t$  : 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 (Discussed further in Chapter 11 : Bias vs 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)  $Q_{\pi}(S_t, A_t)$

✓ 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  $\begin{array}{c} \xrightarrow{\text{policy}} \\ \xrightarrow{Q} \\ \xrightarrow{V} \end{array}$

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

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

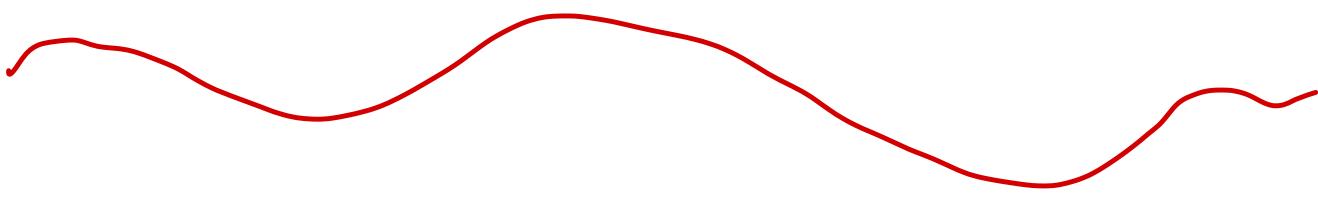
\* Conventionally,  
this is  
usually referred  
to as  
**advantage**

✓ 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  $\begin{array}{l} \xrightarrow{\text{update}} \text{policy network through} \\ \text{advantage (critic)} \\ \xrightarrow{\text{update}} \text{value network (actor)} \end{array}$

# Flow of an actor-critic algorithm



## Step 1 :

- Pass a state information,  $s_t$  to the actor / policy
- Sample an action,  $a_t$  from the policy network's output

## Step 2 :

- Use the critic to evaluate the action

value,  $\hat{Q}_w(s_t, a_t)$

$\hat{Q}$  with w parameter

## Step 3 :

- Perform the action,  $a_t$  & step in the environment to collect reward,  $r_t$  & next state,  $s_{t+1}$

Step 4 :

- Update Actor / Policy Network via Gradient Ascent
- Notice : Similar to Reinforce, except the selection of Advantage Function

$$\nabla_{\theta} (\pi_{\theta}) = \nabla_{\theta} \log \pi_{\theta}(s, a) \cdot \bar{A}$$
$$= \nabla_{\theta} \log \pi_{\theta}(s, a) \cdot \frac{\hat{Q}_w(s, a)}{\text{Advantage function}}$$

selected here is  
the action value

$$\pi_{\theta} = \pi_{\theta} + \alpha \nabla_{\theta} (\pi_{\theta})$$

$\alpha$  Learning rate

Note : You are free to replace the advantage function here, but if it involves state value instead of action value, Step 5 needs to be adjusted accordingly.

## Step 5 :

- Update critic via Gradient Descent
- In this case, the critic is a Q network
- Update using SARSA (from Q learning: Temporal Difference)

TD error

$$\nabla_w = \beta \left( \frac{r_t + \gamma \hat{Q}_w(s_{t+1}, a_{t+1}) - \hat{Q}_w(s_t, a_t)}{\text{Gradient of this Return}} \right)$$

TD target

Learning rate

reward

discount factor

action

value of the next state

(requires actor to sample a new action)

$\hat{Q}_w = \hat{Q}_w - \nabla_w$

## Benefits of using Advantage Actor Critic

\* Advice : Please go through Chapter 11 before proceeding with this following section

If the advantage that you're using are :

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

$g_t$  : Sampled by Monte Carlo  $\rightarrow$  high variance

- By subtracting all  $g_t$  from their corresponding state value, resulting in a more stable & fair critic value for all state in an episode
- Because every step is criticised at a similar range gives they are centered around the state value respectively.
- Variance is reduced !

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

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

- Also help to reduce variance, given that the critic value are centered !
- However, although b) & c) are temporal difference based, where it has high bias problem, it is not dealt with using advantage

Essentially,

- Advantage introduces baseline to the critic value , leading to reduced variation
- It does not deal with bias !

Chapter 7 :

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

Chapter 8 :

Curiosity - Intrinsic  
Curiosity  
Model  
(ICM)

There are 2 major problems in RL

### 1) Sparse Reward

- Most reward does not contain information, hence are set to 0.
- Lack of access to meaningful reward becomes difficult for an agent to learn if their actions are good or bad.

### 2) Extrinsic Reward needs to be handmade

- In each environment, a human needs to manually design & implement the extrinsic reward function.

Solution : Intrinsic Reward

- A function that encourages & rewards the agent to explore area that has high uncertainty (hard to be predicted)
- Simply formulated

$$IR = \left| \left| \frac{\hat{S}_{\text{ext}}}{\downarrow \text{Predicted next state}} - \frac{S_{\text{ext}}}{\downarrow \text{Actual next state}} \right| \right|$$

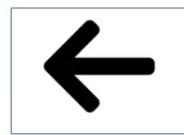
## Further discovery of problem :

- However, if a model is trained to predict a complete state, the state's dimensionality maybe too high & computationally expensive to be predicted

It's hard to predict the pixels directly



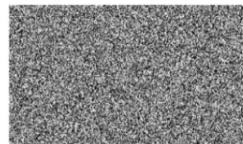
$s_t$



Move left



$s_{t+1}$



Predicted  $s_{t+1}$

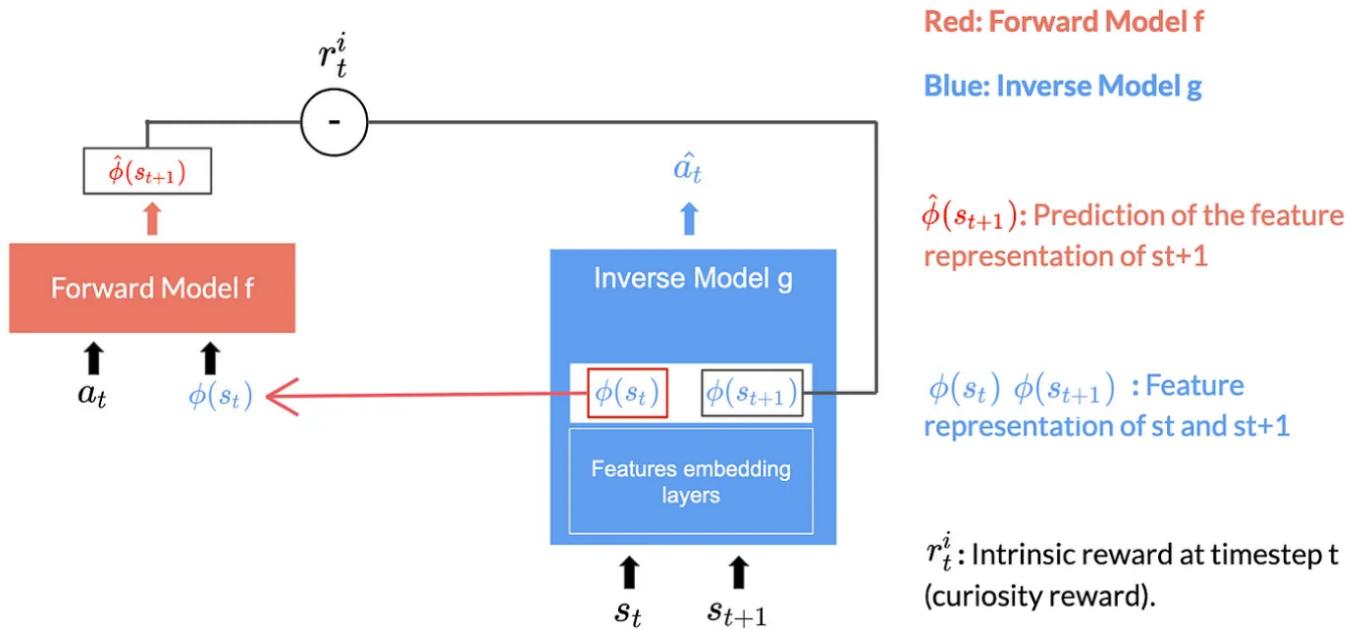


Needs to predict (248\*248)  
61504 pixels!

## Solution :

- Therefore instead of directly predicting the full state, the model should predict the corresponding feature representation
- A few rules to follow in modelling the state's feature representation are :
  - a) Model things that can be controlled by agents
  - b) Model things that cannot be controlled by agents but can affect them.
  - c) Do not model things that cannot be controlled by agents nor affecting the agents

# Introducing Intrinsic Curiosity Module (ICM) :



## Inverse Model, $g$

- responsible to compress state into lower dimensional feature representation

$$\hat{a}_t = g(s_t, s_{t+1}; \Theta_I)$$

↓                    ↓                    ↓                    ↓  
 predict            current            next            weight  
 action taken      state             state            of inverse model

- The loss that is associated with this model is as following :

$$L_I = H(\hat{a}_t, a_t)$$

↓  
 Entropy Loss between predicted  
 actions & actual action taken

- Since this model is designed to predict the actual action to be taken, it will compress the state input into feature representations that will affect the action selection decision.



$\phi(S_t)$ , Feature representation of  $S_t$

$\phi(S_{t+1})$ , Feature representation of  $S_{t+1}$

## Forward model, f

- Responsible to predict the feature representation of the next state

$$\hat{\phi}(S_{t+1}) = f(\underline{\phi(S_t)}, \underline{a_t}; \underline{\Theta_f})$$

$\downarrow$   
 Predicted feature representation of next state

$\downarrow$   
 Feature representation of current state

$\downarrow$   
 Action taken

$\downarrow$   
 Weight of forward model

- The loss that is associated with this model :

$$L_F = \frac{1}{2} \left\| \underline{\hat{\phi}(S_{t+1})} - \underline{\phi(S_{t+1})} \right\|_2^2$$

$\downarrow$   
 Predicted feature representation of next state

$\downarrow$   
 Actual feature representation of next state

## Intrinsic Curiosity

$$r_t^i = \frac{\eta}{2} \left\| \hat{\phi}(s_{t+1}) - \phi(s_{t+1}) \right\|_2^2$$

$\downarrow$   
Predicted  
latent representation  
next state
 $\downarrow$   
Actual  
latent representation  
of next state

Scale factor,  $> 0$

\* It may look similar to forward model loss, but it is designed to be maximized as shown below ↴

## Overall optimization problem

$$\min_{\theta_P, \theta_I, \theta_F} \left[ -\lambda \mathbb{E}_{\pi(s_t; \theta_P)} [\sum_t r_t] + (1 - \beta) L_I + \beta L_F \right]$$

Average reward for episodes / trajectories when the agent follows policy  $\pi$

$\lambda > 0$ , is a scalar that weighs the importance of the policy gradient loss against the importance of learning the intrinsic reward

$\beta$  Scalar (between 0 and 1) that weighs the inverse model loss against the forward model loss

Optimize parameters of policy, inverse model and forward model

Inverse Model Loss	Forward Model Loss
--------------------	--------------------

Note :

Although some articles show that with intrinsic reward / curiosity, the agent can already perform well, sometimes external reward is still needed for exploration

Chapter 9 :

Random

Network

Distillation

There are still some problems with Intrinsic Curiosity Module

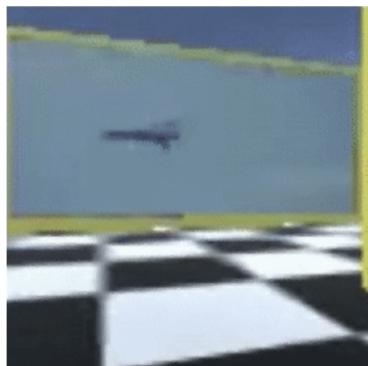
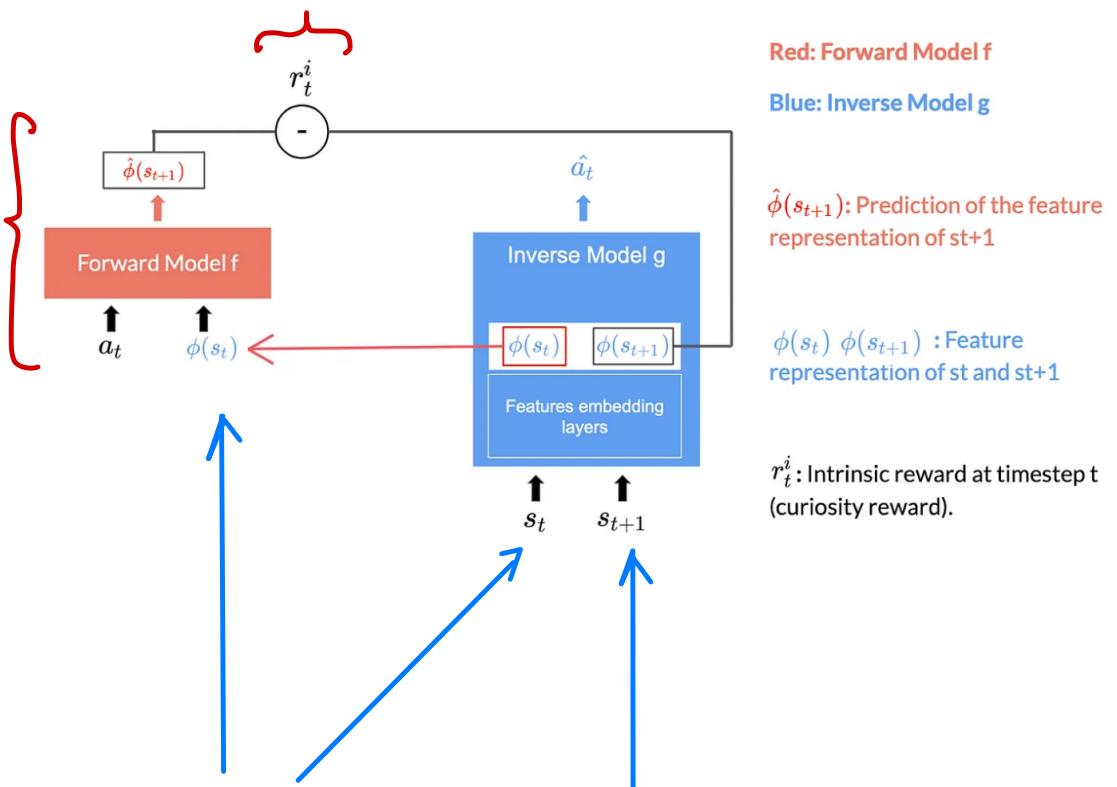
### 1) Noisy TV

- Since the curiosity is calculated by predicting the next state, where the harder it is to predict the next state, the higher the intrinsic reward / curiosity, the even more likely that the agent will explore that area
- However, when the next state is random & stochastic, i.e. a TV problem, the agent has a hard time predicting it & thought there's high curiosity, therefore drawn to it & stopped performing meaningful exploration
- The agent has a hard time predicting because inverse model rely on the prediction of action taken to tweak the inverse model's parameter which subsequently affect the feature representation.

With TU Noise, since there's no clear pattern of a specific action leading to a specific next state, the agent got confused with constantly experiencing a high curiosity.

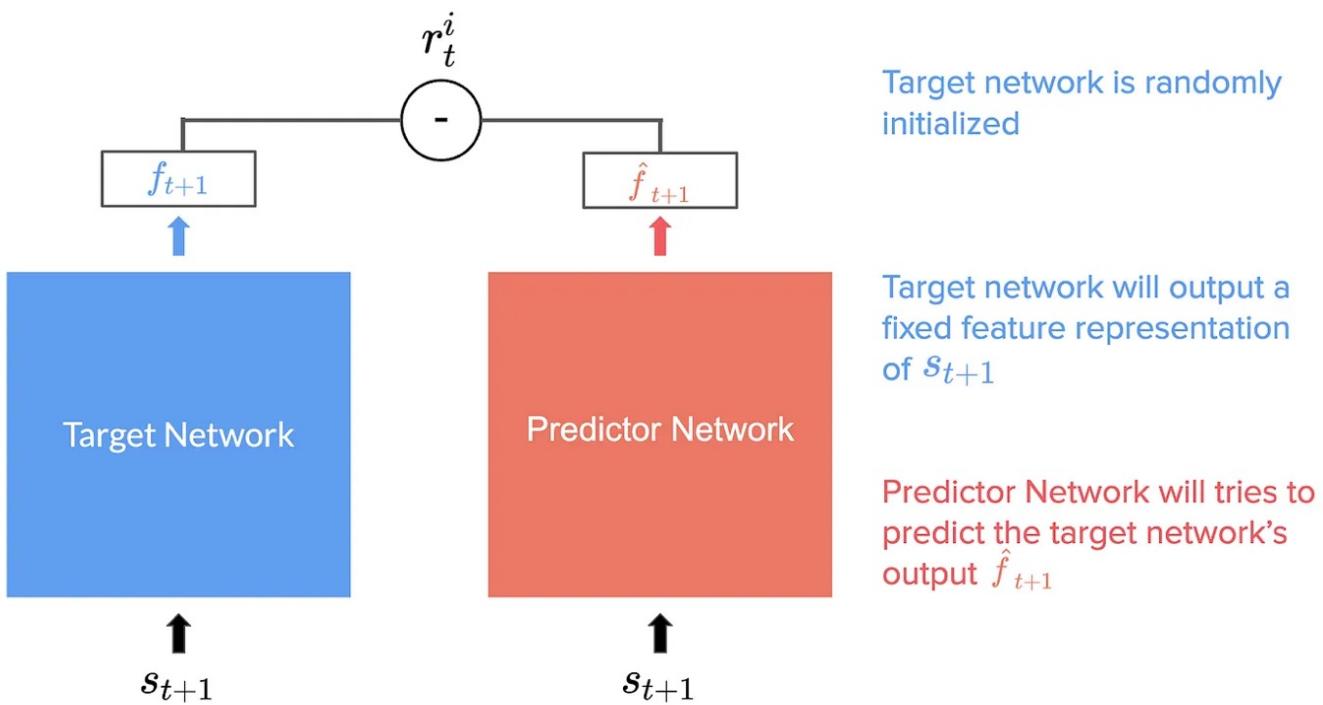
Always high curiosity

Has a hard time predicting the next state



Next state is always random

# Solution : Introducing Random Distillation Network



## Target network

- Randomly initialized
- Never Trained &  $l$  is fixed
- Unlike in ICM, the target network (inverse model) is trained by predicting the action taken, which can result in the feature representation of target to be volatile.
- Since target network is not trained here, any state (no matter how random is it) has no association with its previous state or any actions but instead the resultant representation will be

churned out based on these weight. (Fixed)

- As a result, this is easier for the predictor network to match the target feature representation.
- In long run, even if it faces problems or TV noise, due to ease of matching, the agent's curiosity of it will decrease & it moves on to performing other tasks.

Intrinsic reward

$$\tilde{r}_t^i = \frac{\|\hat{f}(s_{t+1}) - f(s_{t+1})\|_2^2}{\|f(s_{t+1})\|_2^2}$$

$\downarrow$                                      $\downarrow$   
predicted feature  
representation  
of next state  
(output by predictor  
network)  
Actual feature  
representation of  
the next state  
(output by target network)

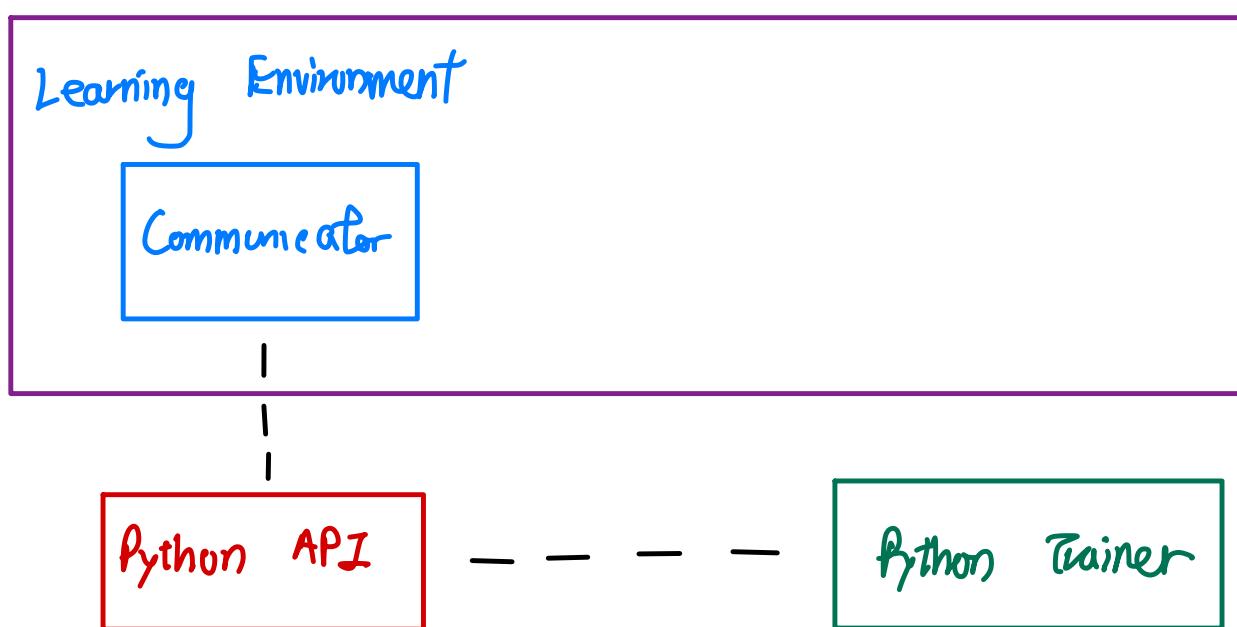
Chapter 0:

Introduction to Unity ML

# Unity ML

- A toolkit that allows
- creation of environments using Unity
  - Train an agent in that environment using RL algorithm

## Main component in Unity ML



### ① Learning environment

- Contains Unity Scene
- Contains Environment element [ agent & interactable objects ]

## ② External communicator

- Connect Environment (coded in C#) with low level Python API

## ③ Python API

- Low level Python Interface for interacting with the environment & launching training

## ④ Python Trainers

- RL algorithm developed with PYTORCH

## ⑤ Gym Wrapper

- Encapsulate learning environment in a gym wrapper

## ⑥ Perting Zoo Wrapper

- Multi agent versions of gym wrappers

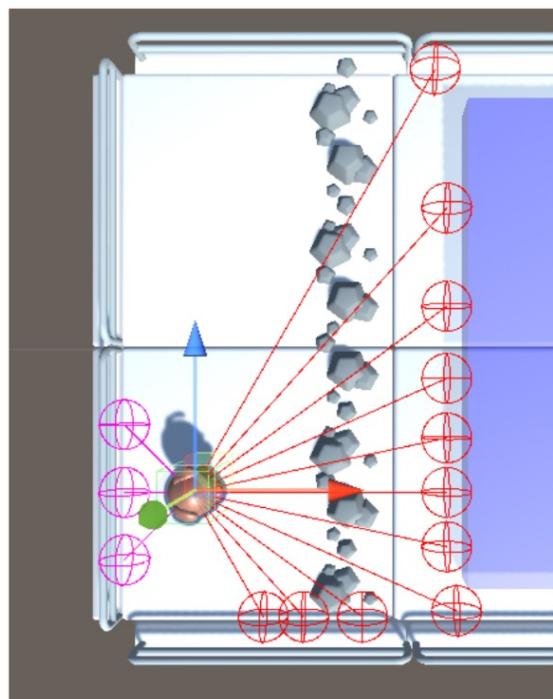
within learning components

- a) BRAIN : Policy of the agent
- b) ACADEMY : Handle requests from Python API & order agents to be in sync

State information in a Unity Environment is collected using Raycast, a laser that will detect if it passes through an object  
*(mimicking vision)*

1 set of back raycasts:

- Detect invisible walls



3 sets of front raycasts:

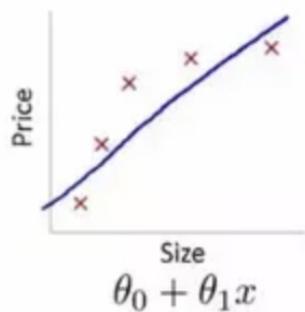
- One set to detect snowballs
- One set to detect targets
- One set to detect frontier (the rocks) and invisible walls

Chapter 11:

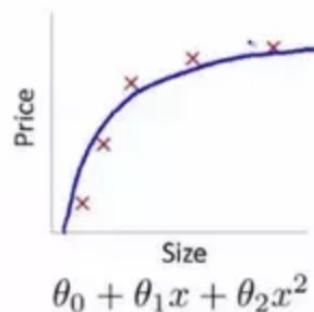
Bias &  
Variance

Trade off in RL

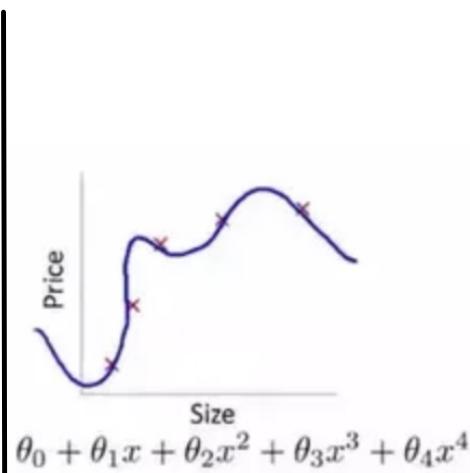
As a refresher, bias & variance tradeoff also exist in the world of supervised machine learning



High bias  
(underfit)



"Just right"



High variance  
(overfit)

### Underfit

- good for test data
- low variance
- high bias

- It has a very high bias of where it wants to be with a low variance in response to the data point

### Overfit

- good for training data
- Low bias
- High variance

- It has no bias of where to go, but is as varied as possible in following all data

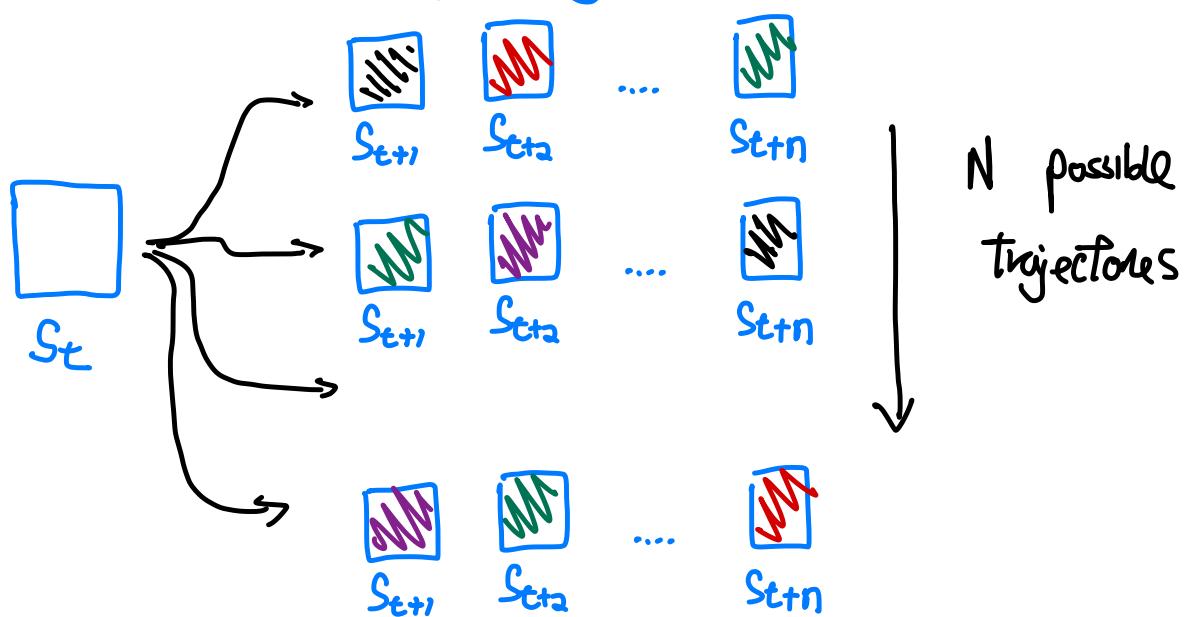
Similarly, bias & variance problem also exists in RL, but in a different way

The bias-variance tradeoff reflects how well the reinforcement signals reflect the true reward the agent should receive from the environment

## ① Monte Carlo

- Sampling reinforcement signals ( $s_t, a_t, r_t$ ) by waiting till the end of episodes
- Resulting in Low Bias, High Variance

Explanation 1 : Why high variance?



- As a result, the discounted reward / reward that will determine the state value or action value are highly volatile, given so many different possible subsequent next states

As a result, lead to **High variance**

- Usually can be resolved by sampling  $N$  number of episodes to reduce the variance

## Explanation 2 : Why low bias ?

- It does not really make any estimates. All the reinforcement reward signals that was sampled were actual happenings from the environment.
- Hence, it samples under the influence of 0 bias.

(2)

## Temporal Difference

- Sampling reinforcement signals  $(S_t, a_t, r_t)$  in between every 2 steps
- Result in low variance & high bias

### Explanations 1 : Why low variance?

- Unlike Monte Carlo, where the rewards at every state can be affected by many different possible state in the remaining of the episodes
- If estimates  $g_t$  by only comparing with the subsequent state. For instance :

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

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

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

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

Approach 3 : Q-Learning

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

Hence,

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

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

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

- As a result, the factor that can influence the variation is a lot less in comparison to Monte Carlo, resulting in low variance.

## Explanation 2 : Why high bias ?

- because it samples by relying on the estimation of next state using a certain algorithm, where the estimation will definitely be less accurate than actual signals from the environment
- Therefore, the signal that is sampled will definitely be affected by some bias introduced by the estimation algorithm.

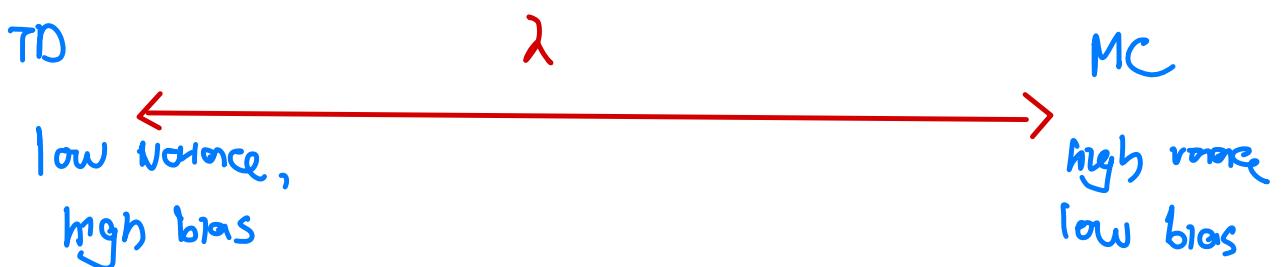
So, how can it be handled?

### Factor 1: Discount Factor, $\gamma$

- For environment where the agent needs to think very far into the future to secure a reward, a discount factor of  $0.999$  is recommended.
- For environment where the agent does not need to think as far into it to secure a reward, a discount factor of  $0.9$  is sufficient.

### Factor 2: GAE Lambda, $\lambda$

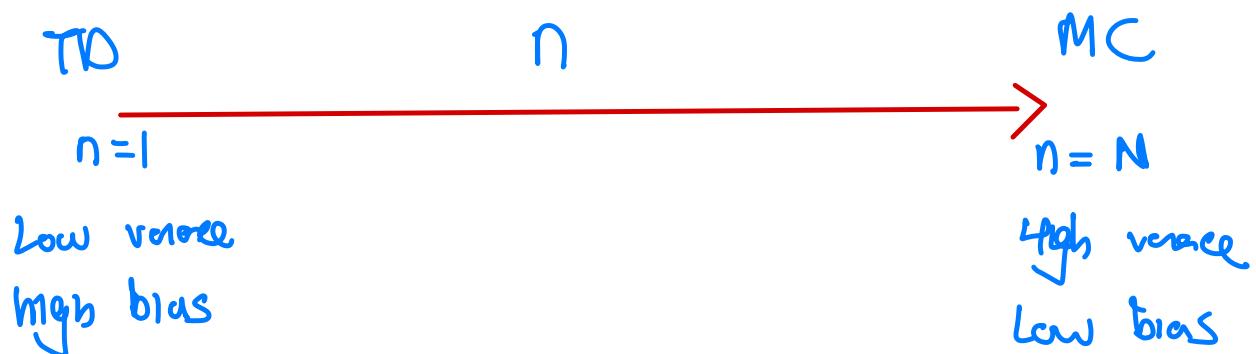
- Typically, advised to be kept in the range of  $0.95 - 0.99$
- The lower it is, the closer it behaves as TD.



- Refer to Chapter 12 for details

## Factor 3 : Time Horizon / n-steps, $N$

- Should be long enough for the agent to receive meaningful rewards
- But can't be too long that it leads to high variance
  - Refer to Chapter 13
- Can be used in combination for GAE



Chapter 12 :

Generalised

Adventoge

Estimate

(GAE)

Recall that in policy gradient, the policy network is updated via Gradient Ascend :

$$\nabla_{\theta} J(\pi_{\theta}) = \sum_{t=0}^H \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \cdot \underline{g(t)}$$

However, as :

- a) discounted return can only be sampled via full trajectories (Monte Carlo) which is inefficient in sampling.
- b) discounted return via Monte Carlo has high variance (as discussed in Chapter 11)

Subsequently, advantage is introduced in Actor-Critic :

$$\nabla_{\theta} J(\pi_{\theta}) = \sum_{t=0}^H \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \cdot \underline{A_t}$$

where :

$$A_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

It introduces

- a) a baseline which is helpful in reducing variance as all the actions taken in

each state is evaluated in comparison to their current state

b) TD Sampling which inherently has lower variance

As observed, advantage successfully solved the issue of high variance. But inherently, TD sampling also has high bias that wasn't handled well.

GAE aims to solve by allowing tunable bias & variance tradeoff :

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

$$A_t^{\text{GAE}} = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l}$$

TD sampled Advantage

Gamma  
Discount Factor

Lambda  
- Tunable parameter  
to adjust bias & variance

When  $\lambda = 0$

$$\begin{aligned} A_t^{\text{GAE}} &= \sum_{l=0}^{\infty} (0)^l \delta_{t+l} \\ &= \underbrace{0^0 \delta_t}_1 + \underbrace{0^1 \delta_{t+1} + 0^2 \delta_{t+2} \dots}_{0} \dots \\ &= \delta_t \\ &= r_t + \gamma V(S_{t+1}) - V(S_t) \end{aligned}$$

When  $\lambda = 0$ ,

- Advantage is TD sampled advantage :
- Inherently, **high bias, but low variance**

When  $\lambda = 1$ ,

$$\begin{aligned} A_t^{\text{GAE}} &= \sum_{l=0}^{\infty} (\gamma)^l \delta_{t+l} \\ &= \underbrace{\delta_t}_1 + \underbrace{\gamma \delta_{t+1}}_{\gamma^1 \delta_{t+1}} + \underbrace{\gamma^2 \delta_{t+2}}_{\gamma^2 \delta_{t+2}} \dots \\ &= \underbrace{r_t + \gamma V(S_{t+1}) - V(S_t)}_{+ \gamma r_{t+1} + \gamma^2 V(S_{t+2}) - \gamma V(S_{t+1})} \\ &\quad \underbrace{+ \gamma^2 r_{t+2} + \gamma^3 V(S_{t+3}) - \gamma^2 V(S_{t+2}) \dots}_{\gamma^3 \delta_{t+3}} \end{aligned}$$

If observed closely,

$\delta_t$  : The advantage of state  $t+1$  in comparing with the state  $t$

$\delta_{t+1}$  : The advantage of state  $t+2$  in comparing with the state  $t$

$\delta_{t+2}$  : The advantage of state  $t+3$  in comparing with the state  $t$

because these advantages are calculated with consideration of the quality of MULTIPLE SUBSEQUENT states, it can lead to higher variance, but low bias.

When  $\lambda = 1$ ,

- high variance, low bias

Summary :

$\lambda = 0$

Low variance

high bias

$\lambda = 1$

High variance

Low bias

A simple way to generalise this algorithm for implementation is as follows ....

$$A_0^{GAE(r, \lambda)} = \sum_{l=0}^{\infty} (r\lambda)^l \delta_l \\ = \delta_0 + (r\lambda)\delta_1 + (r\lambda)^2\delta_2 + (r\lambda)^3\delta_3 \dots$$

$$A_1^{GAE(r, \lambda)} = \sum_{l=0}^{\infty} (r\lambda)^l \delta_{1+l} \\ = \delta_1 + (r\lambda)\delta_2 + (r\lambda)^2\delta_3 \dots$$

$$A_0^{GAE(r, \lambda)} = \delta_0 + (r\lambda) A_1^{GAE(r, \lambda)}$$

$$A_2^{GAE(r, \lambda)} = \sum_{l=0}^{\infty} (r\lambda)^l \delta_{2+l} \\ = \delta_2 + (r\lambda)\delta_3 + (r\lambda)^2\delta_4 \dots$$

$$A_1^{GAE(r, \lambda)} = \delta_1 + r\lambda A_2^{GAE(r, \lambda)}$$

Generalised :  $A_t^{GAE(r, \lambda)} = \delta_t + r\lambda A_{t+1}^{GAE(r, \lambda)}$

How to use GAE in combination with n-value bootstrapping?

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

$$A_t^{GAE} = \sum_{\ell=0}^{\infty} (\gamma \lambda)^\ell \delta_{t+\ell} \rightarrow A_t^{GAE} = \sum_{\ell=0}^n (\gamma \lambda)^\ell \delta_{t+\ell}$$

Example of  $n = 2$

$$A_t^{GAE} = \sum_{\ell=0}^2 (\gamma \lambda)^\ell \delta_{t+\ell}$$

$$= \underline{(\gamma \lambda)^0 \delta_t} + \underline{(\gamma \lambda)^1 \delta_{t+1}} + \underline{(\gamma \lambda)^2 \delta_{t+2}}$$

$$= \underline{\delta_t} + \underline{(\gamma \lambda) \delta_{t+1}} + \underline{(\gamma \lambda)^2 \delta_{t+2}}$$

$$= \underline{r_t + \gamma V(s_{t+1}) - V(s_t)}$$

$$+ \underline{(\gamma \lambda)(r_{t+1} + \gamma V(s_{t+2}) - V(s_{t+1}))}$$

$$+ \underline{(\gamma \lambda)^2(r_{t+2} + \gamma V(s_{t+3}) - V(s_{t+2}))}$$

Chapter 13 :

n - step

Bootstrapping

As previously discussed in chapter 11:

Monte Carlo : High variance, low bias

Temporal Difference , Low variance, high bias

It is observed that

Monte Carlo calculates the target using  
N steps , where

$$N = \text{Final time step} - \text{Current time step}$$

Meanwhile

Temporal Difference calculates the target using  
1 step

If the sampling & update can be done using  
n step were

$$1 \leq n \leq N$$

TD

↓ variance

↑ bias

monte carlo

↑ variance

↓ bias

We will use SARSA as an example to discuss how to integrate n-step bootstrapping

In Temporal Difference .

Approach 1 : SARSA

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

Hence,

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

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

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



In SARSA,

$$\text{Target} = r_t + \gamma Q(S_{t+1}, a_{t+1})$$

with n-step bootstrapping

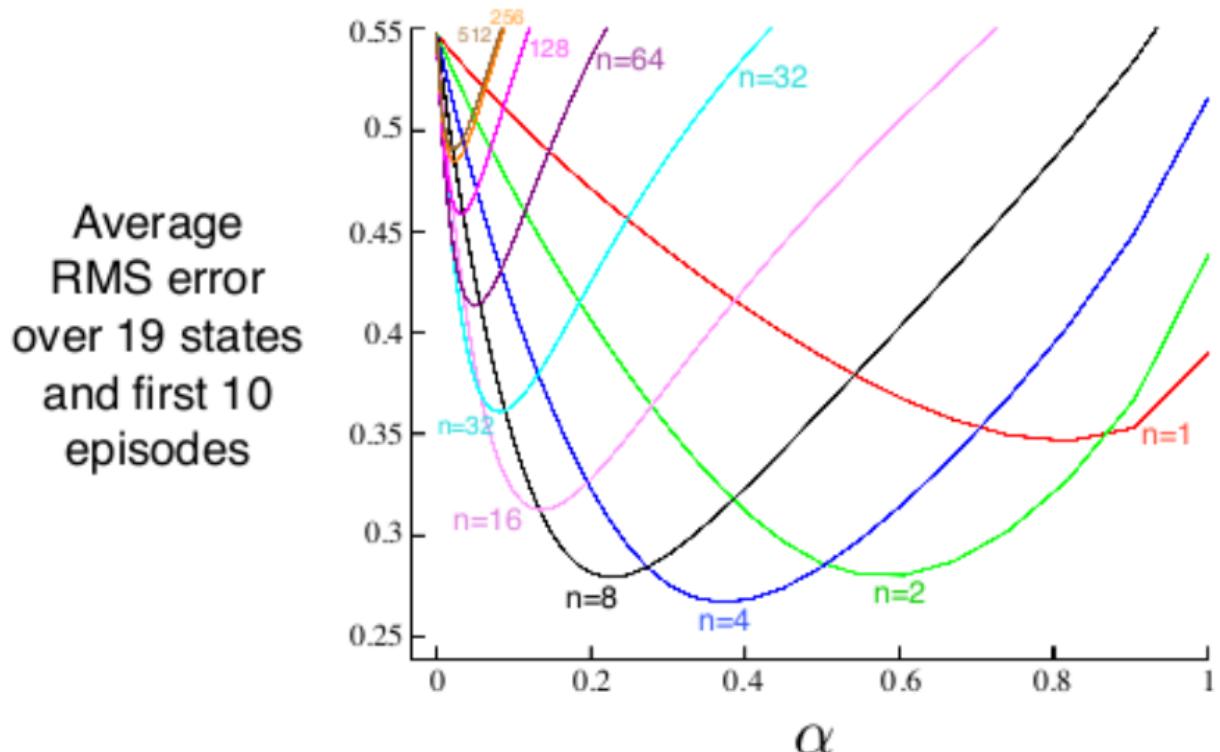
$$\text{Target} = \left[ \sum_{l=0}^n \gamma^l r_{t+l} \right] + \gamma^{n+1} Q(S_{t+n+1}, a_{t+n+1})$$

Assuming if  $n = 2$

$$\text{Target} = \left[ \sum_{l=0}^2 \gamma^l r_{t+l} \right] + \gamma^{n+1} Q(S_{t+1+n}, a_{t+1+n})$$

$$= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 Q(S_{t+3}, a_{t+3})$$

With the addition of  $n$ , it provides another degree of control as showcased below :



Chapter 14 :

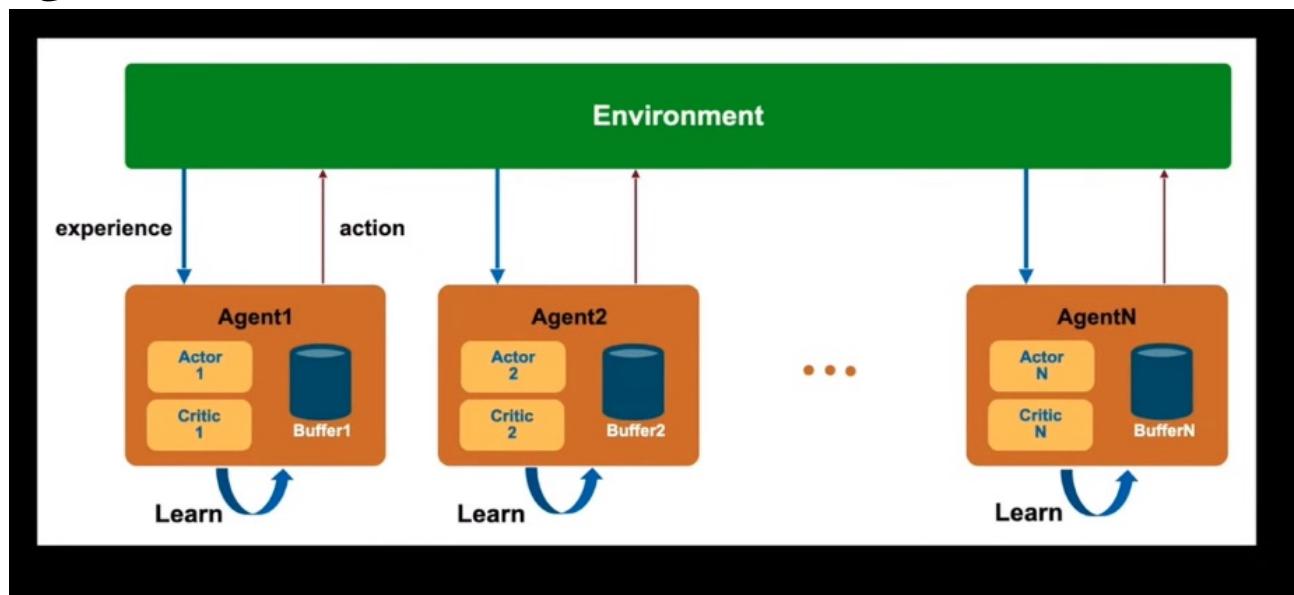
Multi - Agent

Reinforcement Learning

( MARL )

This section serves to give a high level overview of design issues for Reinforcement Learning that involves multiple agent

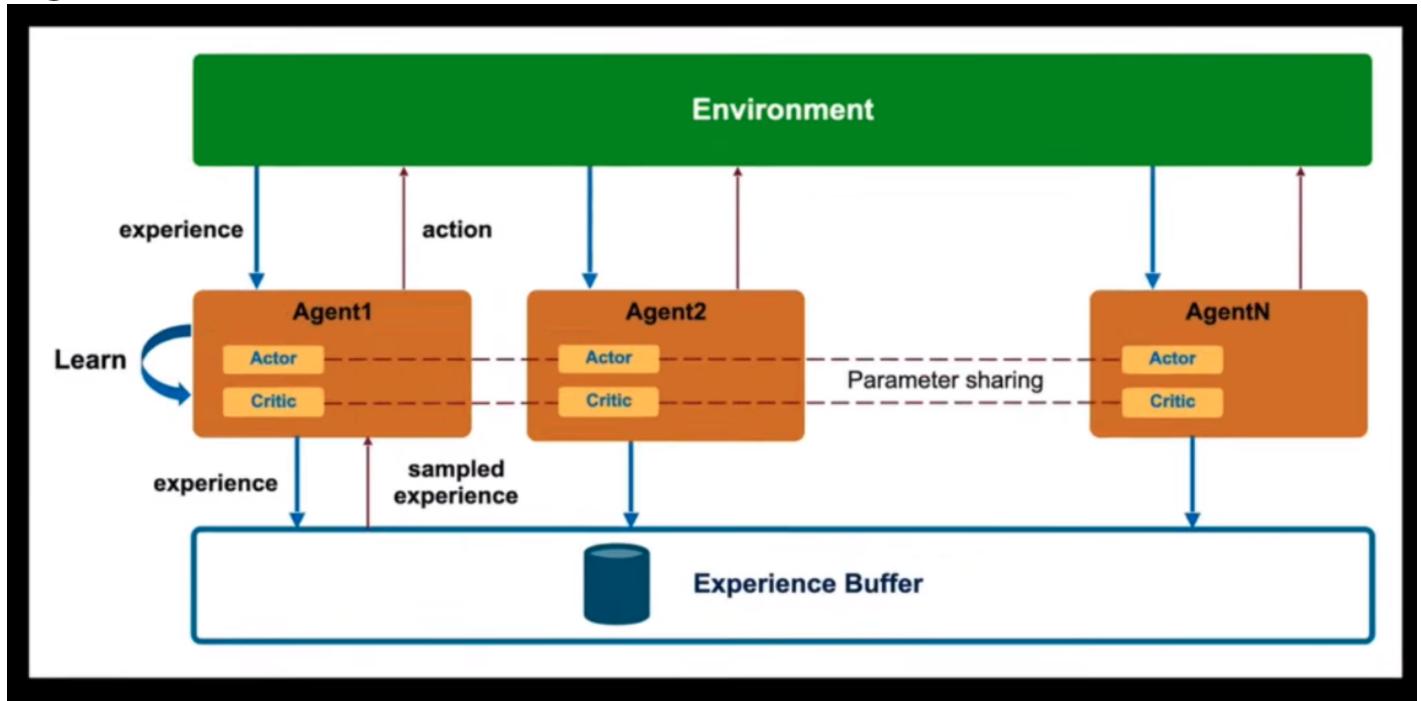
## Design Choice 1: Decentralised



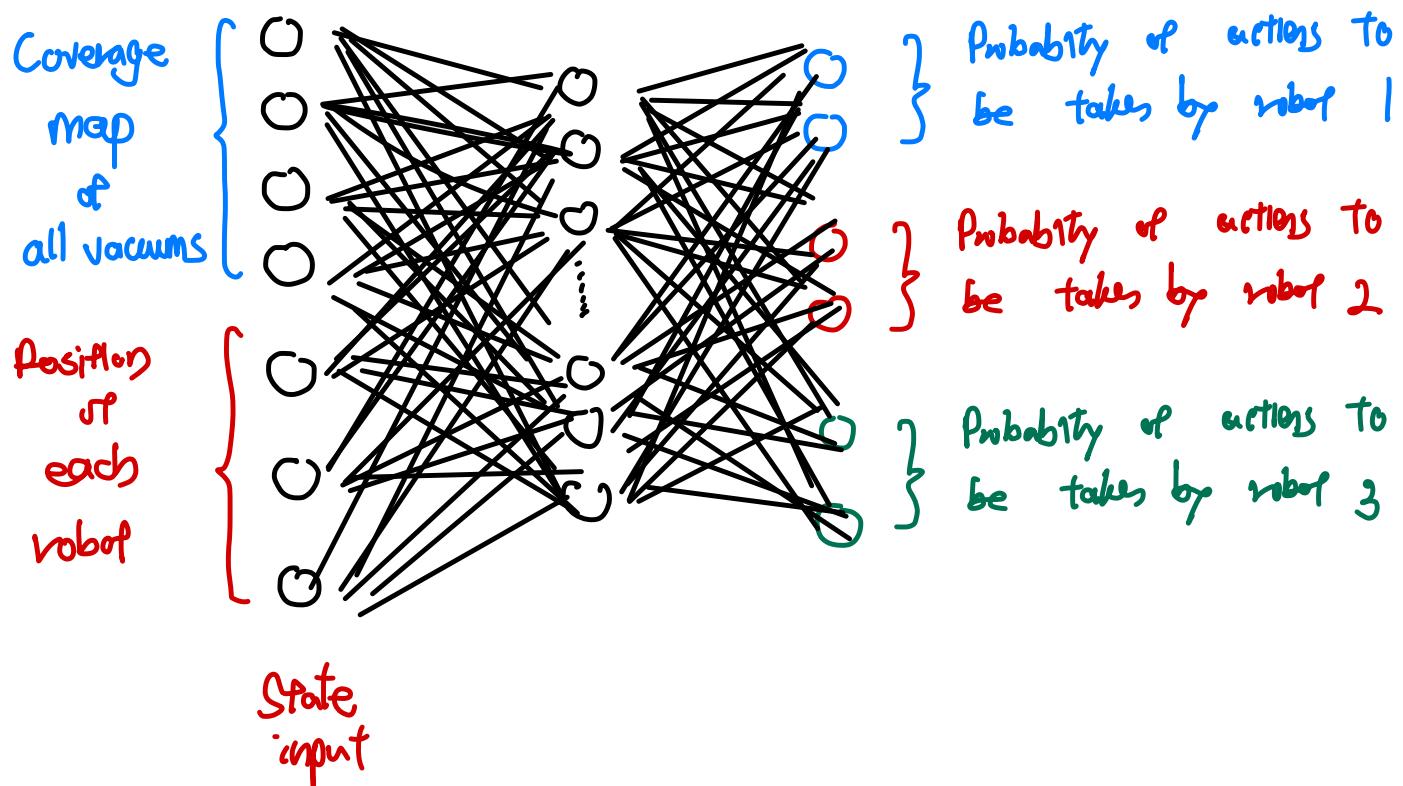
- Each agent is trained independently.
  - ⇒ with their own policy network (Actor), value network (Critic) & buffer
- The interactions of other agents with the environment
  - ⇒ is not known by the agent
  - ⇒ is treated as part of the environment when the result is treated as part of the environment dynamics

- As a result,
  - ⇒ the agents have to learn in a "non-stationary environment" which is very difficult for convergence given many RL algorithms are built on the foundation of Markov Decision Process (MDP)
  - ⇒ The learning is also inefficient given the individual buffer where the experience of one agent is not shared with another agent, despite sharing the same objective.

## Design Choice 2 : Centralised



- All the agents shared the same policy network (actor), value network (critic) & experience buffer
- For example , the policy network can be designed as following where the environment involve multiple vacuums robots in aiming to cover the whole floor plan



- Advantage of this design is that
  - Since state information is shared, the agents can now coordinate with one & another in terms of taking actions, avoiding clashing
  - Another advantage is that they can cover the entire area more efficiently, given the shared coverage map.

## ◦ Disadvantage ?

- When deploying this design, the number of agents to be used is fixed because the policy network has a fixed input shape as shown above. ( In this case, position of the robot is an input that will be affected by number of robots )

Chapter 15 :

Self - Play

In MARL,

- agents can be cooperative with one & another  
→ maximizing common benefits
- agents can also be competitive with one & another  
→ minimizing opponent's benefits to maximize one's benefits

Self-play is a technique used to train competitive agents.

- Use a copy of former self as an opponent
- This way, the agent gets to play against another agents on similar skill level, thus maximizing the learning effectiveness
- As the agent improves, the former self (the more recent copies) also is supposed to be at similar level.

Based on the implementation of Self play in ML-Agents, let's go through some hyperparameters which ought to introduce better intuition.

<https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Training-Configuration-File.md#self-play>

## 1) Save steps

- Number of trainer steps between saving a snapshot of the current policy network.
- This saved snapshot can potentially be loaded as a future opponent.
- High save steps lead to opponents with wider range of skills lead to more difficult learning but can lead to more generalizable strategies
- Default : 20 000
- Typical range : 10 000 - 100 000

## 2) Team change

- Number of trainer steps between switching the team that will undergo learning.
- A large value of team change lead to a team playing the same set of possible opponents

For longer can lead to overfitting

- Also can determine how many snapshots being saved as future opponent, recommended to be set as a function of save steps
- Default :  $5 \times \text{Save\_steps}$   
Typical range :  $4x - 10x \text{ Save\_steps}$

### 3) Swap steps

- Number of ghost steps between swapping the opponents policy with a different policy snapshot
- When an agent is the learning team, it is taking trainer step
- If the agent is not the learning team, it is taking ghost step
- Formula of swap-step

$$\frac{\text{Num\_agents}}{\text{Num\_opposite\_agents}} \times \frac{\text{Team\_change}}{\text{Swap\_steps}} = x$$

where

$x$  : Number of policy swap for the opponents during each team-change interval

→ Hence,

- the higher the number of agents on a team, the more frequent the opponents will have to change their policy
- because more agents means collecting more observation, tend to overfit faster.

→ Default : 10 000

Typical range : 10 000 - 100 000

#### 4) play-against-latest-model-ratio

→ Probability of playing the opponent with the latest policy

→ If this probability is not hit, it will then play against a opponent with policy sampled from window.

→ The larger this value is, the more likely it will play the latest agent which is constantly updated, which can lead to framing instability

→ Default : 0.5

Typical range : 0 - 1.0

## 5) Window

- Number of past snapshots to be stored which can potentially be used as an opponent.
- The larger the window is, the wider the range of skill levels of opponents, the more difficult the training is, but can potentially produce an agent with more generalizable strategies.

## Downside with Self-Play

- ① More suited for 1v1 because it evaluates only the total reward
- ② In 2v2 and above, it does not give credit to a particular agent in the winning team. Therefore, it may result to one agent carrying the whole team while the rest is not improving.

Chapter 16

ELO

- In adversarial MARL cases, ELO rating is a better way to evaluate the performance of an agent.
- It does not eliminate the necessity of designing a reward function in updating the agent.
- It is just another way that can coexist with reward in evaluating the agent.

### Advantages of ELO system

- 1) Points are always balanced because it is zero sum
- 2) Self-corrected system where the more the difference in skill levels, the more the stronger player will lose point or the less the stronger player will earn point
- 3) Works with team game because it can average ELO point of each member in the team

## Disadvantages of ELO System

- 1) Does not account for individual contribution
- 2) Rating Deflation : A good rating requires skills over time to maintain
- 3) Can't compare rating in history

For each player, they will start off with an initial ELO, usually 1200

Before the game, the expected scores for each player will be calculated

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}} \quad E_A, E_B : \text{Expected score}$$

$$E_B = \frac{1}{1 + 10^{(R_A - R_B)/400}} \quad R_A, R_B : \text{Actual elo rating of players}$$

When the match ends, their ELO gets updated

$$R'_A = R_A + K(S_A - E_A)$$

$$R'_B = R_B + K(S_B - E_B)$$

$K$ :  $K$  factor, a fixed hyperparameter, to decide the magnitude of adjustment, usually  $K=16$  or  $K=32$

$R'_A, R'_B$  : Updated elo rating of players

$S_A, S_B$  : Match result

$$S_A / S_B = 1 \quad \text{if win}$$

$$S_A / S_B = 0.5 \quad \text{if tie}$$

$$S_A / S_B = 0 \quad \text{if lose}$$

Chapter 17 :

Multi-Agent

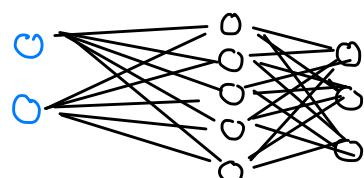
Posthumous Credit Assignment

(MA-PCA)

On a very high level, the architecture design of MA-POCA is as following

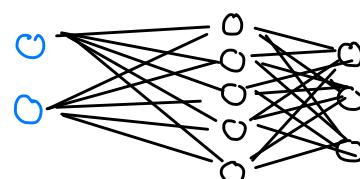
### a) Policy Network

Agent 1



The JT network

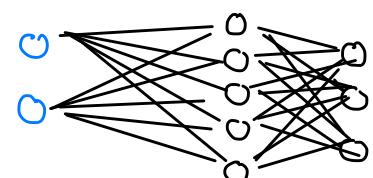
Agent 2



The JT network

=

Agent n

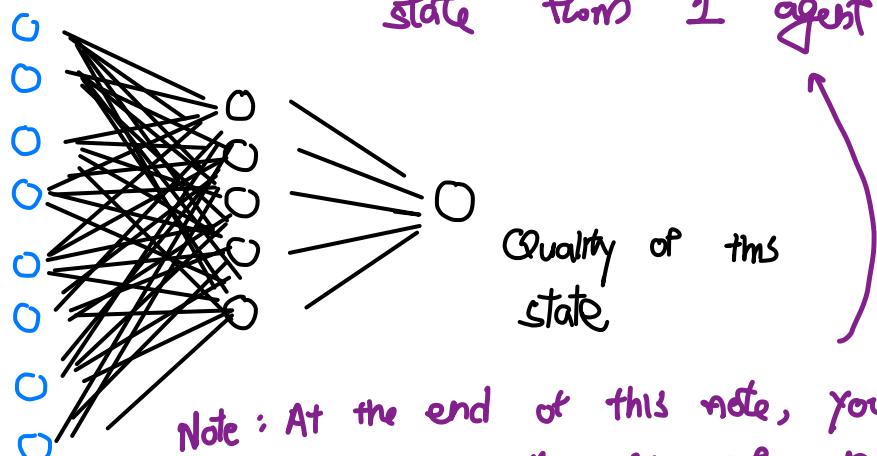


The JT network

- All agent shares the same policy network with the same parameter
- The networks takes in state input relative to the agent that it currently governs only
- The network then outputs the probability or taking each action for its agent.

### b) Value Network

State information collected by only 1 agent



But still it's accurate that value network only take input state from 1 agent

Note: At the end of this note, you will realise that this is not an accurate illustration of the architectures

- The network takes in the state information  
that is collected only one agent  
(which individually will be fed into their respective policy network)
  - It then outputs a value that estimate the quality of this state.
  - It is also worth pointing out that all agents share the same value network.
- 
- More nuanced architectural design will be discussed later on.
  - But having this knowledge will help to understand the following discussion.

MA-POCA aims to resolve the following issues

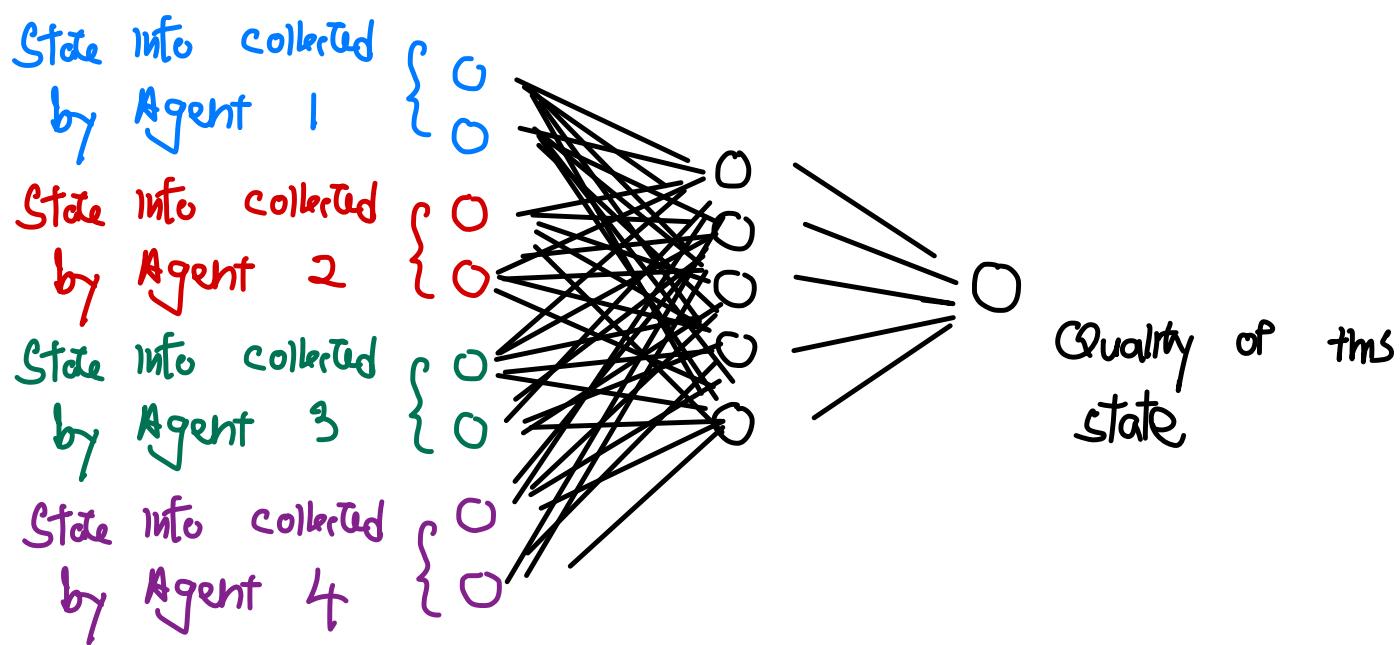
### ① Variable number of agents → Absorbing states

- In a multi-agent case, some agents may have its episode terminated early or some agents may only be introduced later in an episode.
- Therefore, the number of agents across various timestep in an episode is not constant
- As a result, for both the policy network (actor) & value network (critic), these networks need to know the maximum number of agents that are possible. This is because this information allows the networks to
  - To accommodate a state input size that is corresponding to a timestep where there's maximum number of agents
  - When the number of existing agents is less than the maximum number of agents, the

corresponding extra state can be zero'd out (or assigned a certain value that is beyond the possible range of state)

Conventionally,

A V network, assuming maximum number of agent is 4

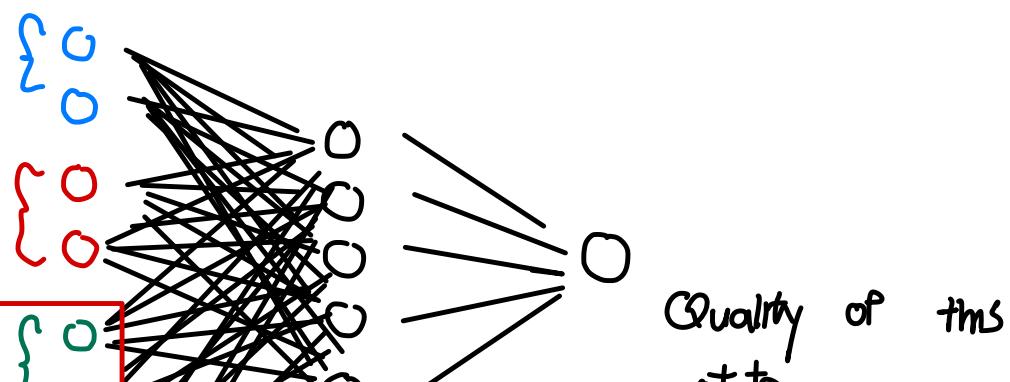


Agent still alive, collecting state

Agent still alive, collecting state

Agents are dead, assign a placeholder value

Agents are dead, assign a placeholder value.



→ When the agents are 'dead' but are assigned a placeholder value, these agents are said to be in Absorbing State which has the following problems:

- a) Training inefficiency & inefficient resource use
- b) Introduces element into the input distribution that makes the estimation by the neural network to be more challenging

• It can be argued that why not have only 1 value network with its parameter shared across agents? (similar to the policy / actor network)

→ However, when both actor & critic network are taking in state information relative to the currently involved agent only, it cannot achieve coordination.

→ With the current setup where

- a) a shared policy network that takes in only the state information relative to the currently involved agent

b) a value network that takes in the state information relative to every single involved agent which allows it to evaluate the quality of the whole state.

At least, the value network can help to coordinate by overseeing all agents.

The only problem is that we need to deal with the inefficiency caused by absorbing states when number of involved agent is less than number of maximum agent.

## ② Posthumous Credit Assignment

- For agents that are terminated early, it will not get to experience the rewards given to the group after termination
- Therefore, these agents will have hard time to learn if their actions are valuable for the group
- Absorbing state allows the state value beyond an agent's termination to be propagated all the way back.

• For instance,

→ To update a value network, you need to compute some form of target

→ In most simplified example with SARSA,

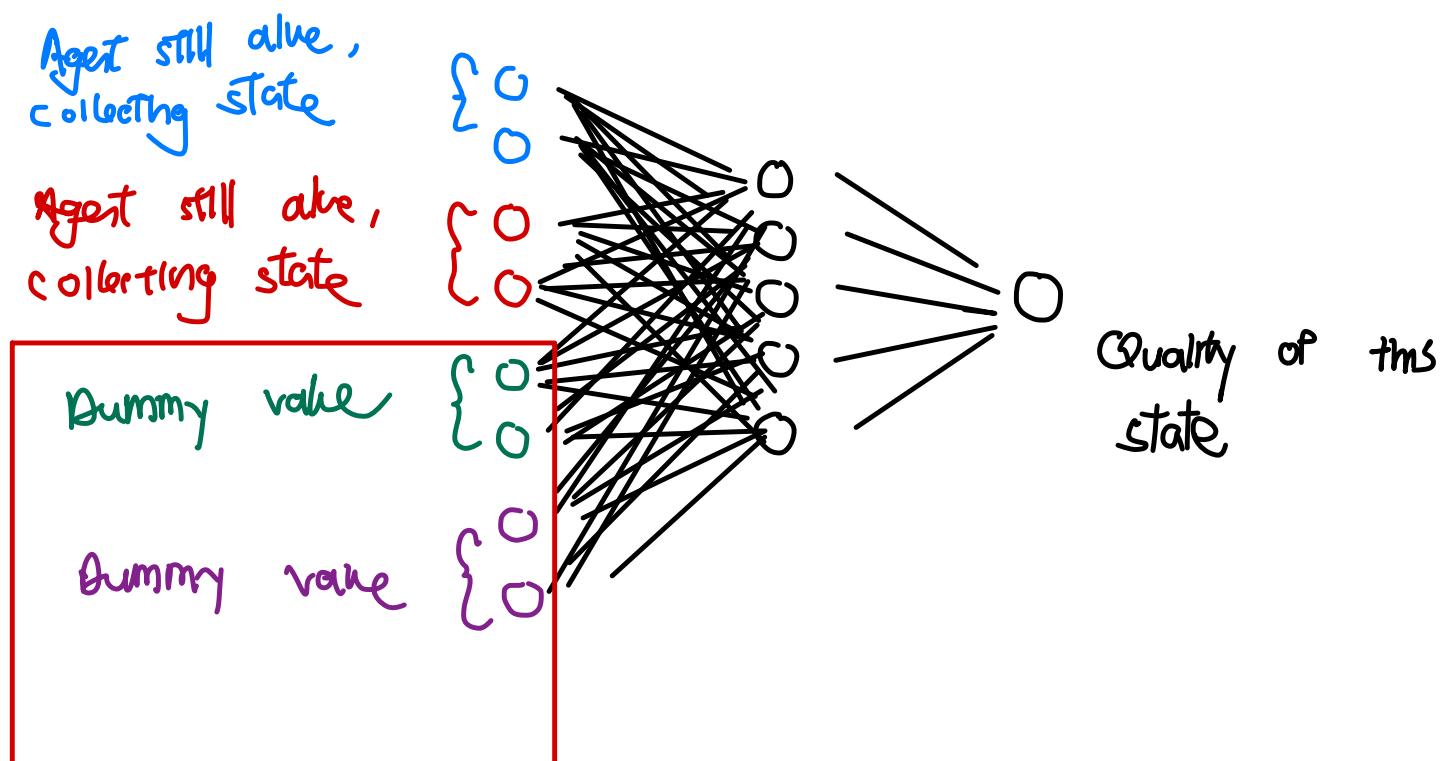
$$\text{Target} = r_t + \gamma V(S_{t+1})$$

→ Despite these agents are already terminated, they're still in absorbing state, passing in dummy value

→ Therefore, as the network is adjusting its parameter to achieve the desired target, it is still shaping the

distribution in such a way that still consider the state information of those agents that was before termination

→ Therefore, allowing the reward post termination to be propagated to the state before termination, which is important to the critic's advantage calculation



- Though absorbing state solve this problem, it remains disadvantageous as previously discussed

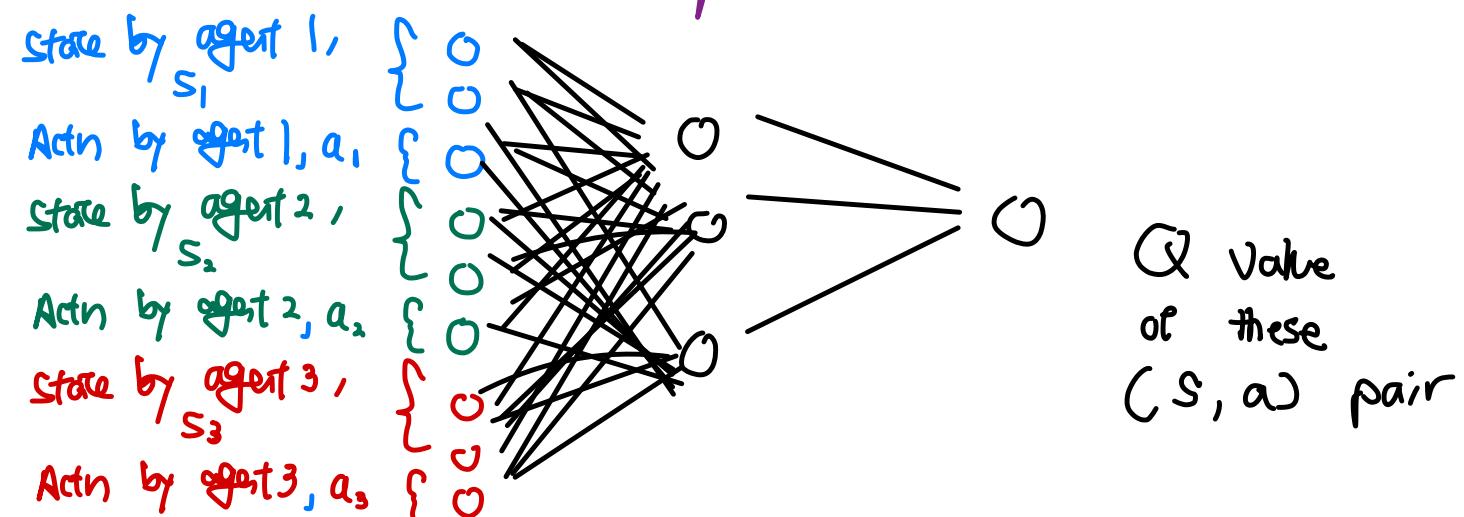
∴ As a summary, as much as absorbing state helps to solve many problems in Multi-Agent Learning, it is not the most optimal solution

before finally introducing MA-PCCA which can optimise these situations, let's introduce one more pre-requisite knowledge - Counterfactual Baseline !

Counterfactual baseline

- Introduce per agent baseline
- So the advantage reflects individual agent's contribution to the total reward

For the following discussion, it is better to visualise the Q network this way



action vector  
 $a = (a_1, a_2, a_3 \dots a_n)$   
 → consists of action information from all agents

Expected value  
 $b_i(s, a) = \underset{a' \sim \pi_i(\cdot | s_i)}{\mathbb{E}} [Q^\pi(s, (a^{-i}, a'))]$

action vector that has action information from all agents except agent i

baseline for agent i  
 state vector  
 $s = (s_1, s_2, s_3 \dots s_n)$   
 → consists of state information from all agents

The action taken by agent i, which can be any of the possible actions offered by the policy network for current state

Further derivation,

$$b_i(s, a) = \sum_{a'} \frac{\pi_i(a' | s_i) \cdot Q^\pi(s, (a^{-i}, a'))}{\text{Probability of taking action } a' \text{ based on the policy with the current state of } s_i \text{ for agent } i}$$

$$\text{Value of when agent } i \text{ takes action } a' \text{ while all other agent takes fixed action as recorded in } a^{-i}$$

Essentially, this baseline can be thought of as

what is the expected value of when

- all other agents are taking fixed agent
- while agent  $i$  takes many different actions?

which can effectively serve as a baseline to examining the actual advantage earned when the agents takes a set of actions collectively

state vector

$$s = (s_1, s_2, s_3, \dots, s_n)$$

→ consists of state information from all agents

action vector

$$a = (a_1, a_2, a_3, \dots, a_n)$$

→ consists of action information from all agents

$$\text{Advantage}_i = Q^\pi(s, a) - b_i(s, a)$$

Therefore, the update for the policy network will be as following :

$$\begin{aligned} \nabla_{\theta_i} J(\theta_i) \\ = \underset{s \sim p^{\pi}}{E} \left[ \nabla_{\theta} \log \pi_i(a_i | o_i) \cdot \text{Advantage}_i \right] \\ a^i \sim \pi_i \end{aligned}$$

Considering the policy network used has its parameter shared across agent,

$$\theta_{\text{update}} = \theta_{\text{old}} + \frac{\alpha}{\text{Sum of gradients across all agents}} \sum_i^n \nabla_{\theta_i} J(\theta_i)$$

Learning rate

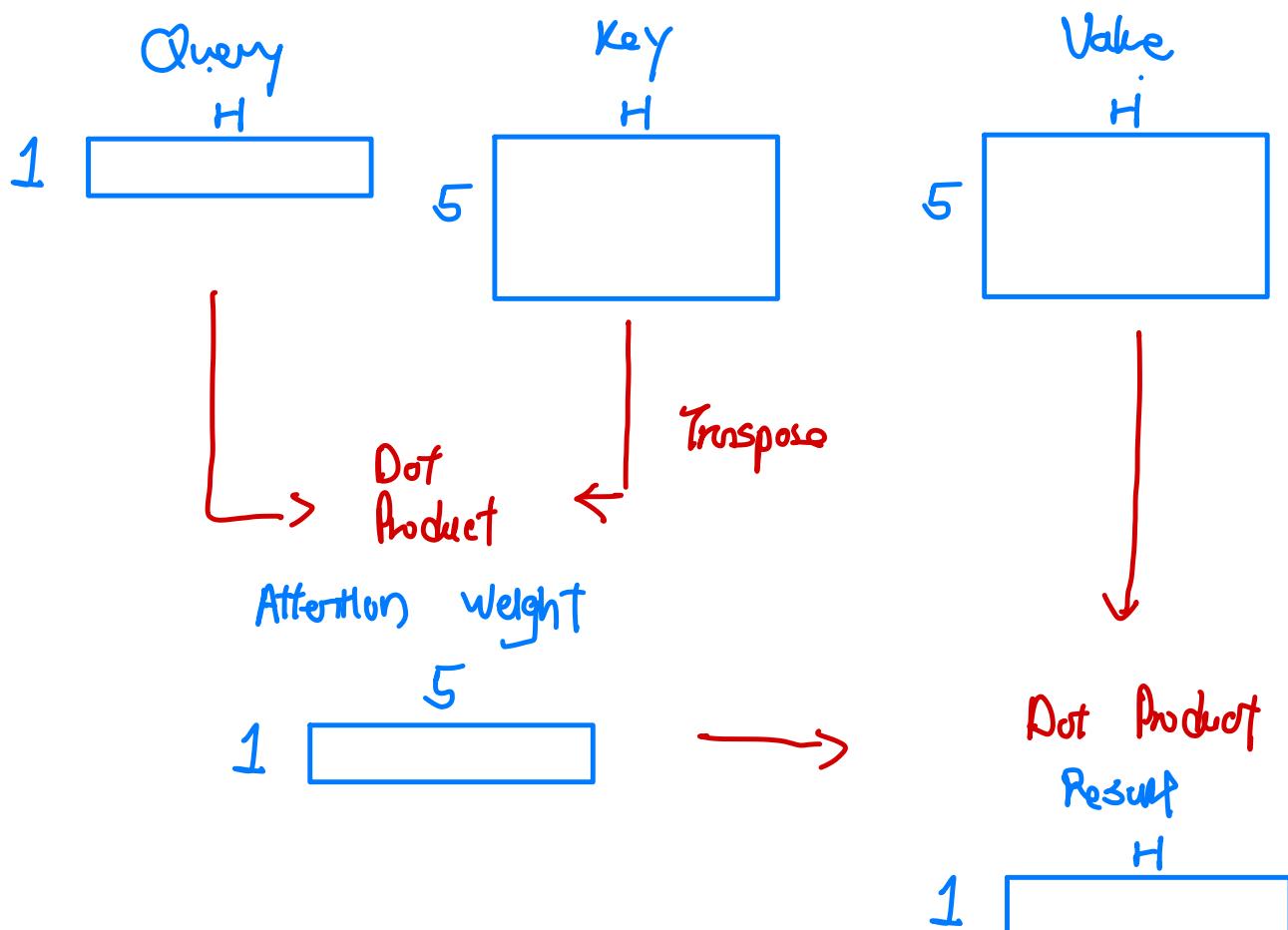
# MA-POCA introduces ...

Attention - block to replace Neural Network

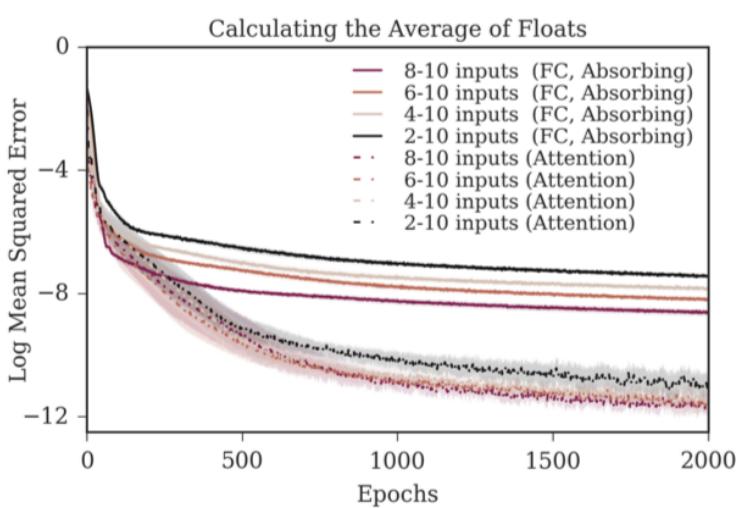
a) Can be used even when there's variable number of agents

- Assuming the state's information collected by each agent is encoded into an embedding with the size of  $1 \times H$
- Assuming cross attention is involved ...

Query : 1 agent's state information :  $1 \times H$   
Key / Value : 5 agent's state information :  $5 \times H$



- As observed, as long as the state information can be encoded into  $1 \times H$  embedding, number of agent involved is not an issue when a floating block is involved.
- b) Eliminate the use of absorbing state which leads to improved estimation / prediction



- An experiment is conducted ...
- Both Neural Network & Attention architectures are designed to predict the mean of a group of inputs with different size
- In NN, when an input of 8-10 is used, that means it might have 0-2 node that needs to be in absorbing state.
- While in attention, no matter what input size is given, as long as each 1 input

is encoded into  $1 \times H$ , it can handle without absorbing state.

- As observe, the error in the estimation made by Attention is much lower than NN
- In NN, from 8-10 input to 2-10 input, the absorbing state increases, lead to higher error

In short,

introduction of attention allow the input to be in the shape of

$$N \times H$$

where

$N$ : number of agents, can be any value

$H$ : hidden dimension of the input information collected by 1 agent

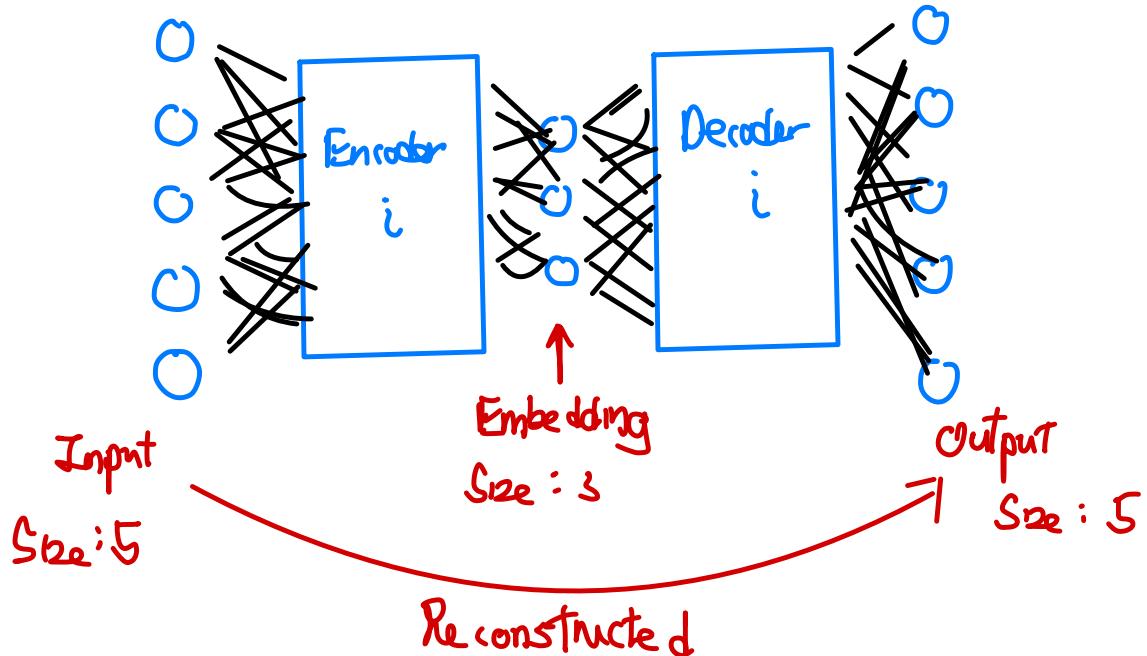
As observed above,  $H$  must still be uniform

Therefore, the paper made the following assumption

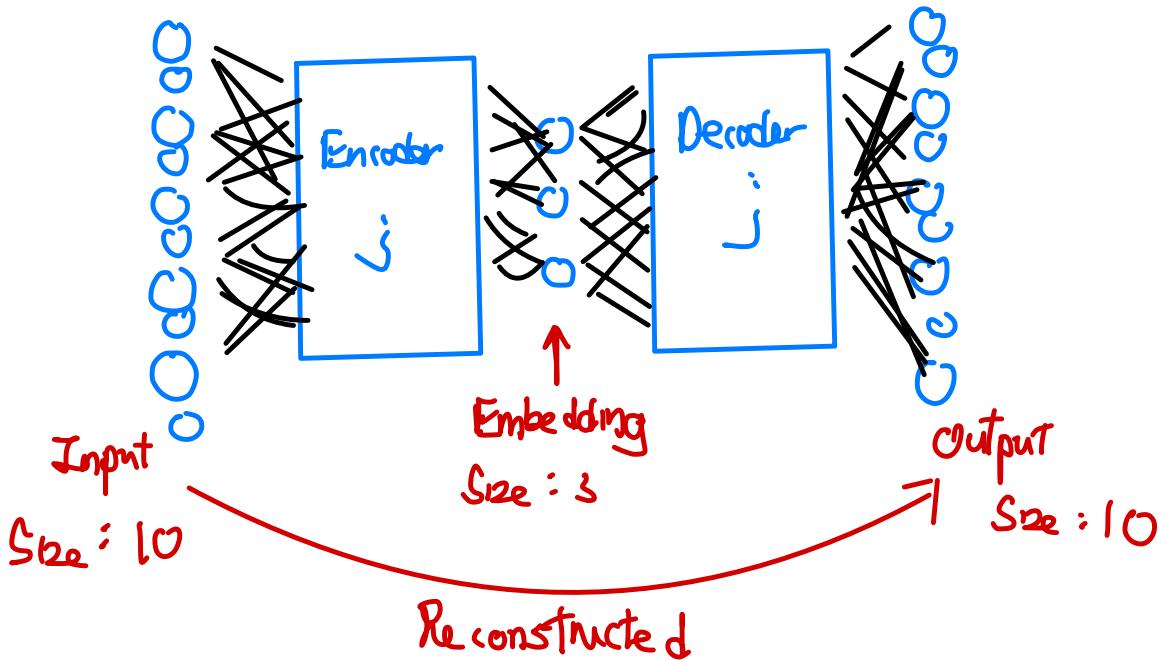
If agent  $i$  & agent  $j$  shares the same observation space, they will use the same encoder. Else, they will use different encoder

For instance,

agent  $i$  may be observing a state vector  
that has 5 elements



↓  
different observations space  
but leads to same  
embedding dimensions



With that out of the way, let's go through the 3 main components in MA-PoCA

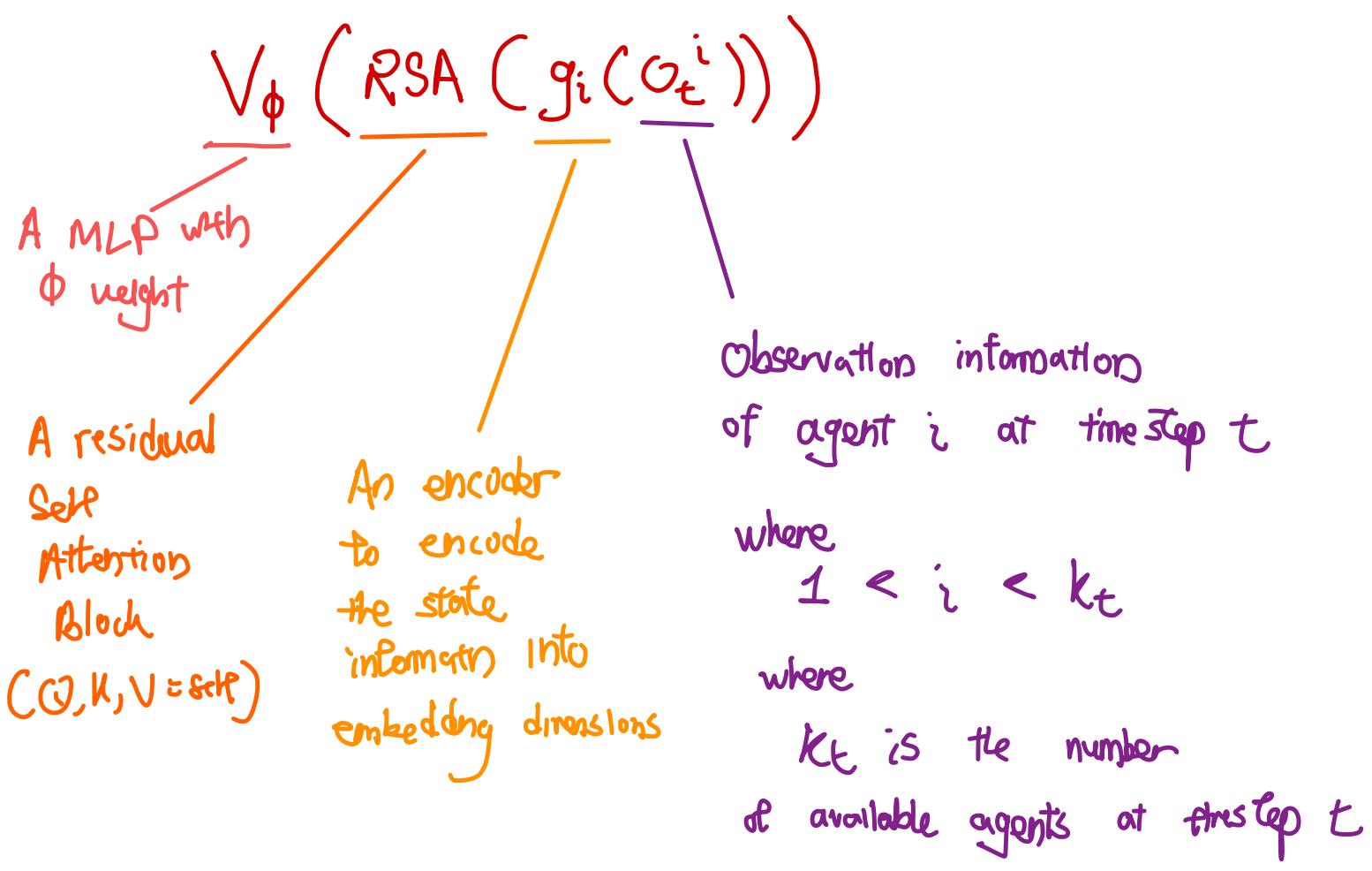
a) Critic - A value network

b) Baseline - For calculations of advantage

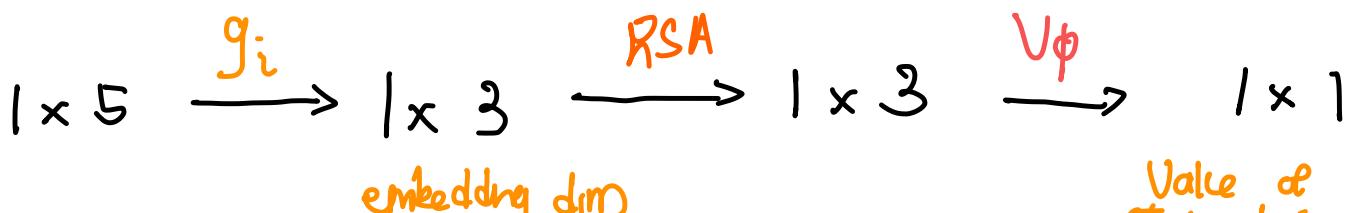
c) Actor - A policy network

a) Critic - All agent share the same critic b/w

The structure :



Assuming  $O_t^i$  has a size of  $1 \times 5$



How to train ?

- By minimizing  $J(\phi)$

$$J(\phi) = \left[ \frac{V_\phi(RSA(g_i(o_t^i)))}{\text{Predicted value of this state / observation}} - \frac{\gamma^{(\lambda)}}{\text{Target value of this state / observation}} \right]^2$$

where

$$\gamma^{(\lambda)} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

Although  $n$  can be up to  $\infty$ , but that will be MC, so user can done it to be lower

lambda, refer to nAE.  
 $\lambda = TD$ ,  $1 = MC$

Discount factor

Discarded return  
String from timestep,  $t$  with  $n$ -value bootstrapping

where

$$G_t^{(n)} = \sum_{l=1}^n \gamma^{l-1} r_{t+l} + \frac{\gamma^n V_\phi(RSA(g_i(o_{t+n}^i)))}{\text{The value of next state after bootstrap for } n \text{ step}}$$

- With this design, it is expected that even if the agent terminates at  $t$ , reward from state  $t+n$  still can be propagated backward, thus helping to evaluate  $S_t$  & the corresponding contribution of this agent

## b) Baseline network

- As you may notice now, if value & policy network both are only going to take in the state information, how does it solve COORDINATION?
- That is with the help of a baseline network we will talk about its intuition later on.

Structure :

Encoder,  
 → different from  $g_j$   
 → because it is also  
 encoding action)

A MLP  
 with  $y_j$  weight

observation of agent  $i$  at  
 time step  $t$   
 where  $1 \leq j \leq k_t$   
 where  $k_t$  is the number of agents  
 at time step  $t$   
 But  $i \neq j$

$$Q \left( \underbrace{\text{RSA} \left( g_j(O_t^j) \right)}_{\text{Encoder}}, \underbrace{f_i(O_t^i, a_t^i)}_{\text{Residual Cross Attention}} \right)$$

Encoder

Observations of  $j$  at time step  $t$   
 where  $1 \leq j \leq k_t$   
 where  $k_t$  is the number of agents  
 at time step  $t$   
 But  $j \neq i$

Residual Cross Attention

$$\begin{cases} Q : g_j(O_t^j) : 1 \times H \rightarrow \text{Refer to agent } j \\ V, K : f_i(O_t^i, a_t^i) : N \times H \end{cases}$$

With the help of  $g_j$  &  $f_i$   
 respectively, all the  
 embedding has the same  
 size of  $H$

↳ Refer to all  
 agents that  
 are present at  
 time  $t$

↳ Except agent  $j$   
 ↳  $N = k_t - 1$

How to train ?

- By minimizing  $J(\psi)$

$$J(\psi) = \frac{\left( Q_\psi(RSA(g_j(o_t^j), f_i(o_t^i, a_t^i)) - y^{(2)} \right)^2}{\text{Predicted baseline}}$$

Target baseline

- Overall how much return can this state bring back
- Same as previously discuss

So what is the intuition of this baseline network?

$$Q_\psi(RSA(g_j(o_t^j), f_i(o_t^i, a_t^i)))$$

Key      Key, Value

①

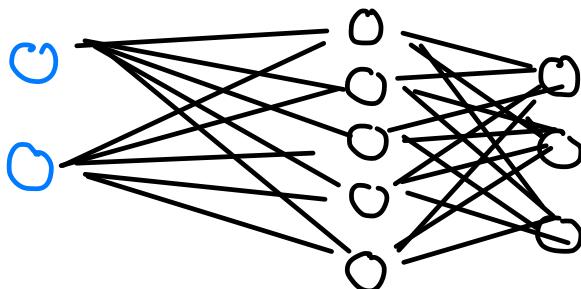
→ This setup will basically return the value of this state that agent  $j$  is in after comparing or paying attention to the state & action of all other agents available at this time step

→ As a result, that can effectively serve as a baseline

- ② → The existence of this baseline also allows coordination to take place since it allows each agent to compare fairly regarding their current value
- This baseline allows the calculation of advantage (which will be used to update policy) to be done with consideration of what other agents are doing
- which needs the decision making (done via policy network) is made by considering presence of other agents, hence give birth to coordination!

c) Policy Network - All agent shares the same policy network

- The design of the policy network is rather simple



$$\pi(O_t^i)$$

→ Taking in the state information at timestep t

→ Output the probability of taking each action

How to train it?

$$\nabla_{\theta_i} J(\theta_i) = \underset{s \sim p^\pi}{E} \left[ \nabla_\theta \log \pi_i(a_i | o_i) \cdot \text{Advantage}_i \right]$$

$a^i - \pi_i$

Considering the policy network used has its parameter shared across agent,

$$\theta_{\text{update}} = \theta_{\text{old}} + \frac{\alpha}{N} \sum_i \nabla_{\theta_i} J(\theta_i)$$

Learning rate

Sum of  
gradient across  
all agents

$$\text{Advantage}_j = \frac{\gamma^{(2)} - Q_\theta(\text{RSA}(g_j(o_t^j), f_i(o_t^i, a_t^i)))}{\text{Baseline for agent } j}$$

1  
Discounted return