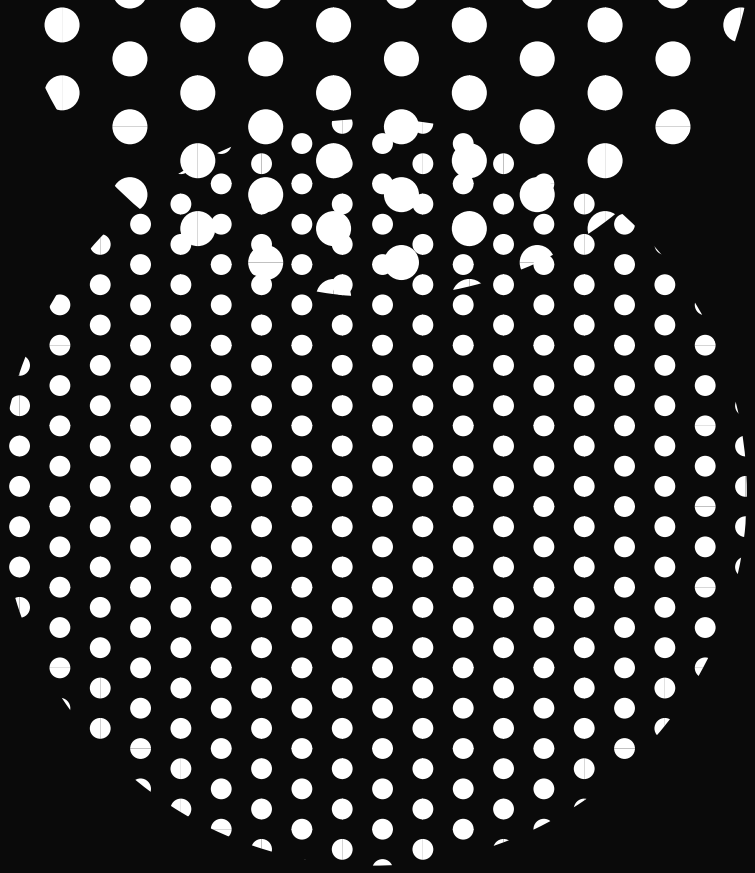
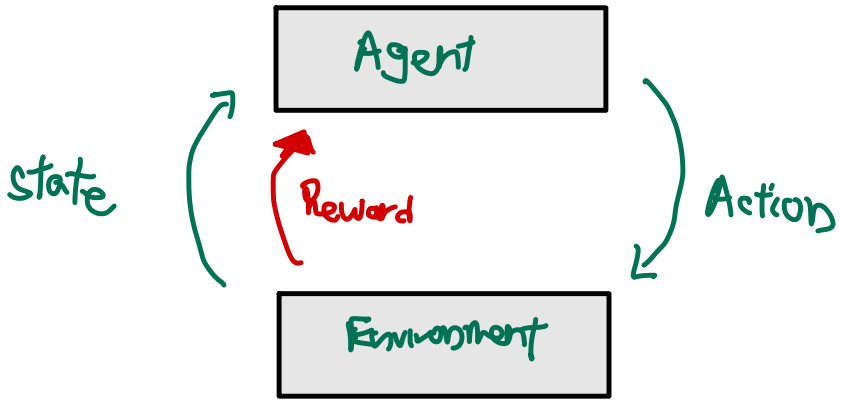


Chapter 1 :

Terminology in Markov Decision Process

Agent : Something that can be directly controlled

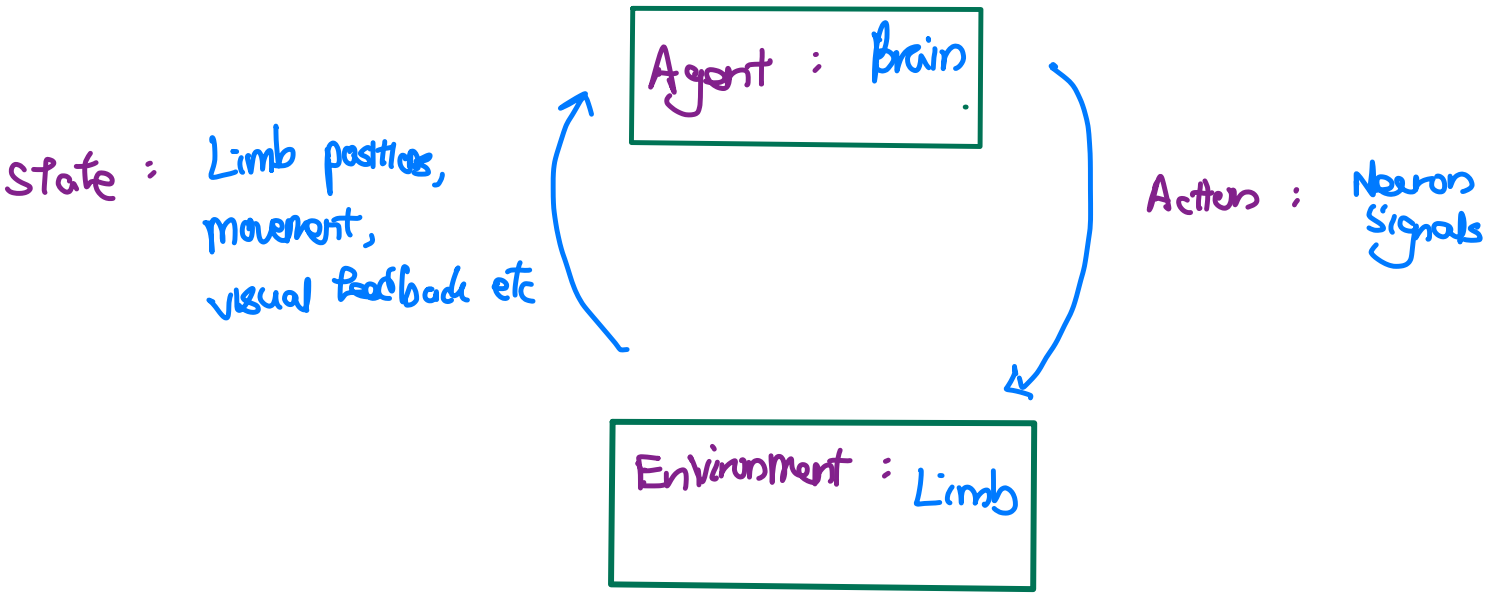
Environment : Something that cannot be controlled,
but can be interacted with through agent



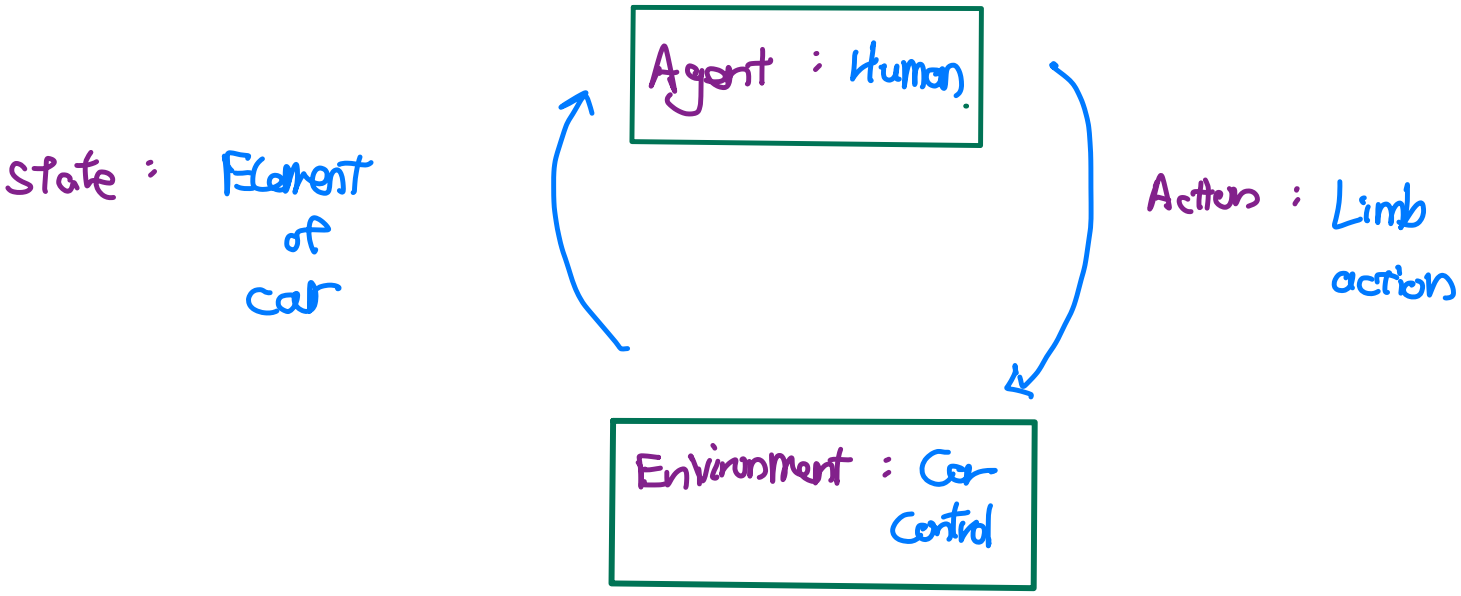
How to determine agent, environment, action & state can be quite arbitary

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

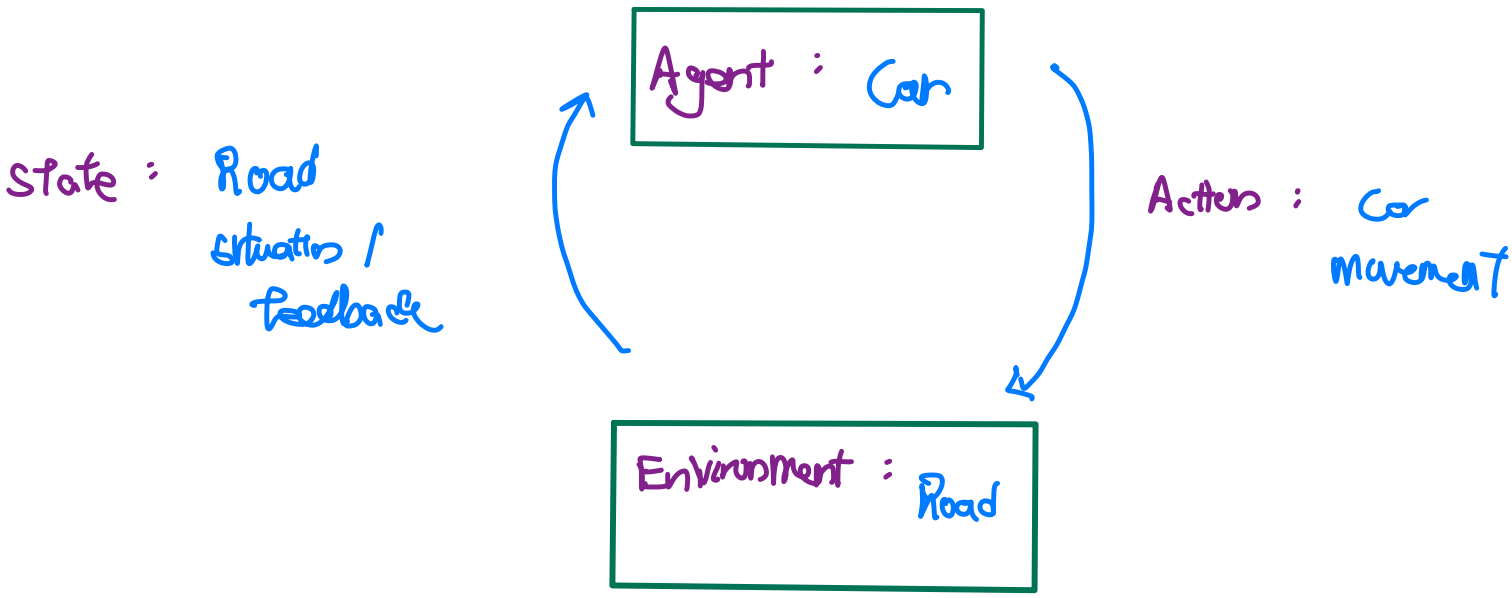
Program A : Control limb movement



Program B : Control interaction with the vehicle



Program C : Control the driving



Markov Decision Process :

- made up of a series of

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

S_0: 0^{th} state; a_0: 0^{th} action; r_0: 0^{th} reward

Note: In other sources, this is defined in

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

- 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

$S_0 \rightarrow a_0 \rightarrow r_0 \uparrow$

- This is done through :

a) Policy, π

$\pi(a|s)$ = 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

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

- where

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

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

Q - Learning

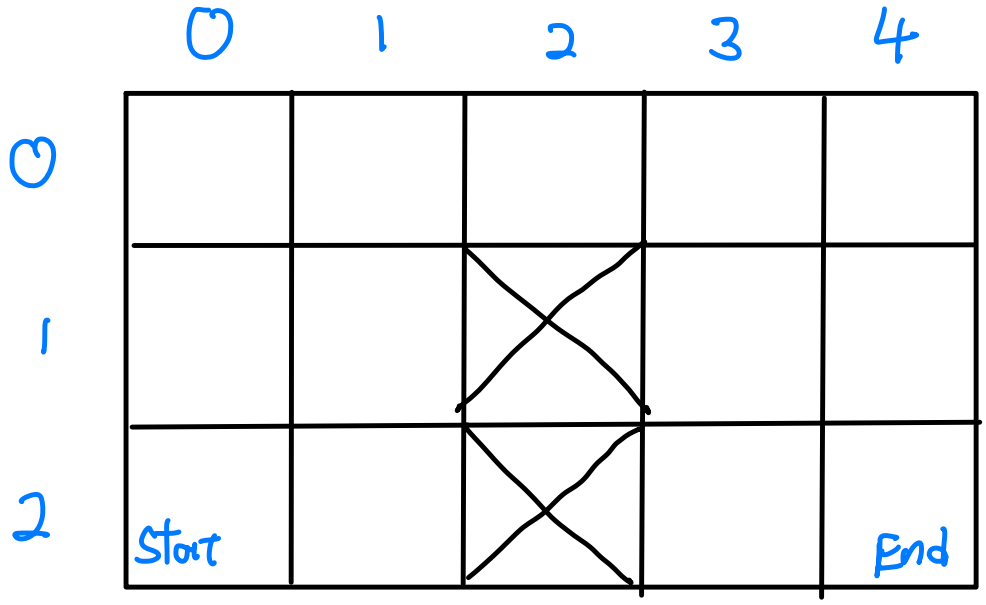
Chapter 2 :

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 → (x, y) coordinates

Action : 1 number → 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:

$$p(s', r \mid s, a)$$

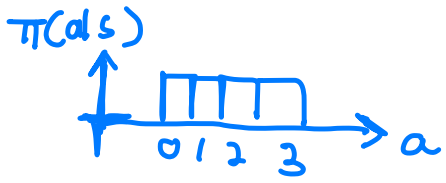
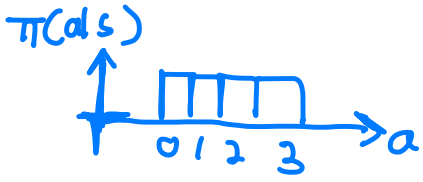
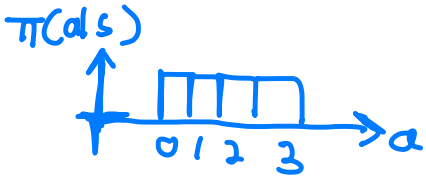
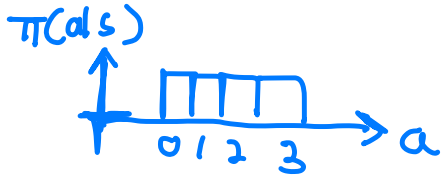
→ the subsequent state & reward given the previous state & action

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 word model access

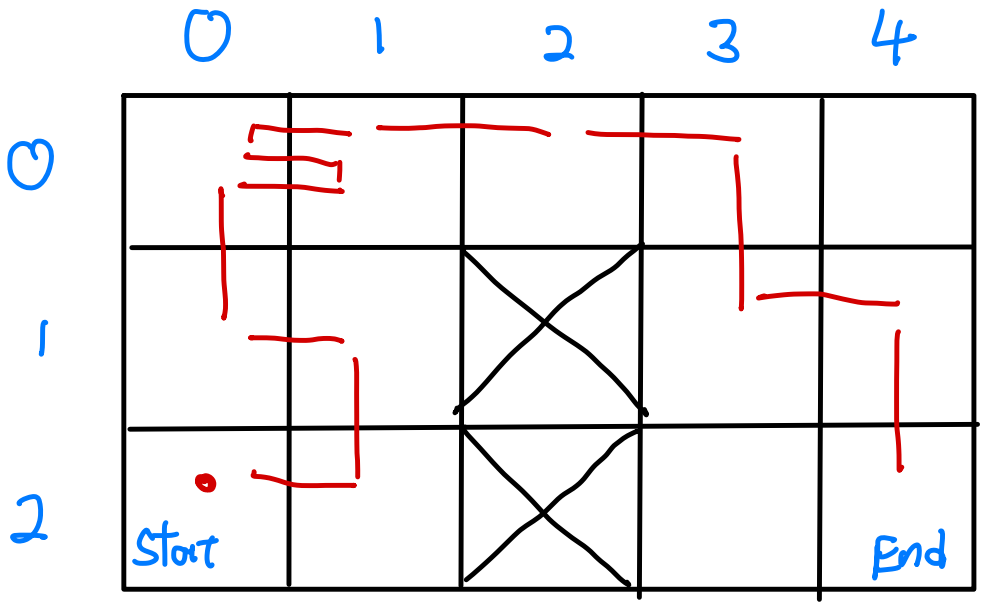
State	Policy, $\pi(a \mid s)$
0, 0	
0, 1	
0, 2	
0, 3 ⋮	

otherwise, if we have access to the model, $\pi(a \mid s)$ 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 backward



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

Given $\gamma = 0.8$

Why is it easier to calculate return from back?

$$g_9 = r_9 = 0$$

$$g_8 = r_8 + \gamma r_9 = -1 + 0.8\,(0)$$
$$= -1$$

$$g_7 = r_7 + \gamma r_8 + \gamma^2 r_9 = -1 + \gamma(r_8 + \gamma r_9)$$
$$= -1 + \gamma g_8$$
$$= -1 + (0.8)(-1)$$
$$= -1.8$$

$$g_6 = r_6 + \gamma r_7 + \gamma^2 r_8 + \gamma^3 r_9$$
$$= r_6 + \gamma(r_7 + \gamma r_8 + \gamma^2 r_9)$$
$$= r_6 + \gamma(g_7)$$
$$= -1 + (0.8)(-1.8)$$
$$= -2.44$$

In short,

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

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, lets renew what are action-value function

- State-value function, $V_\pi(s)$
  → Average return (G_t) expected when following a certain policy (π) in a certain state (s)

- Action-value function, $Q_\pi(s, a)$
  → Average return (G_t) expected when following a certain policy (π) 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 & action

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

→ Initialized at 0

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

From every step in the sampled trajectory / episode:

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

new value = old value + (New return − old_value) × Δ%

Let set Δ% 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:

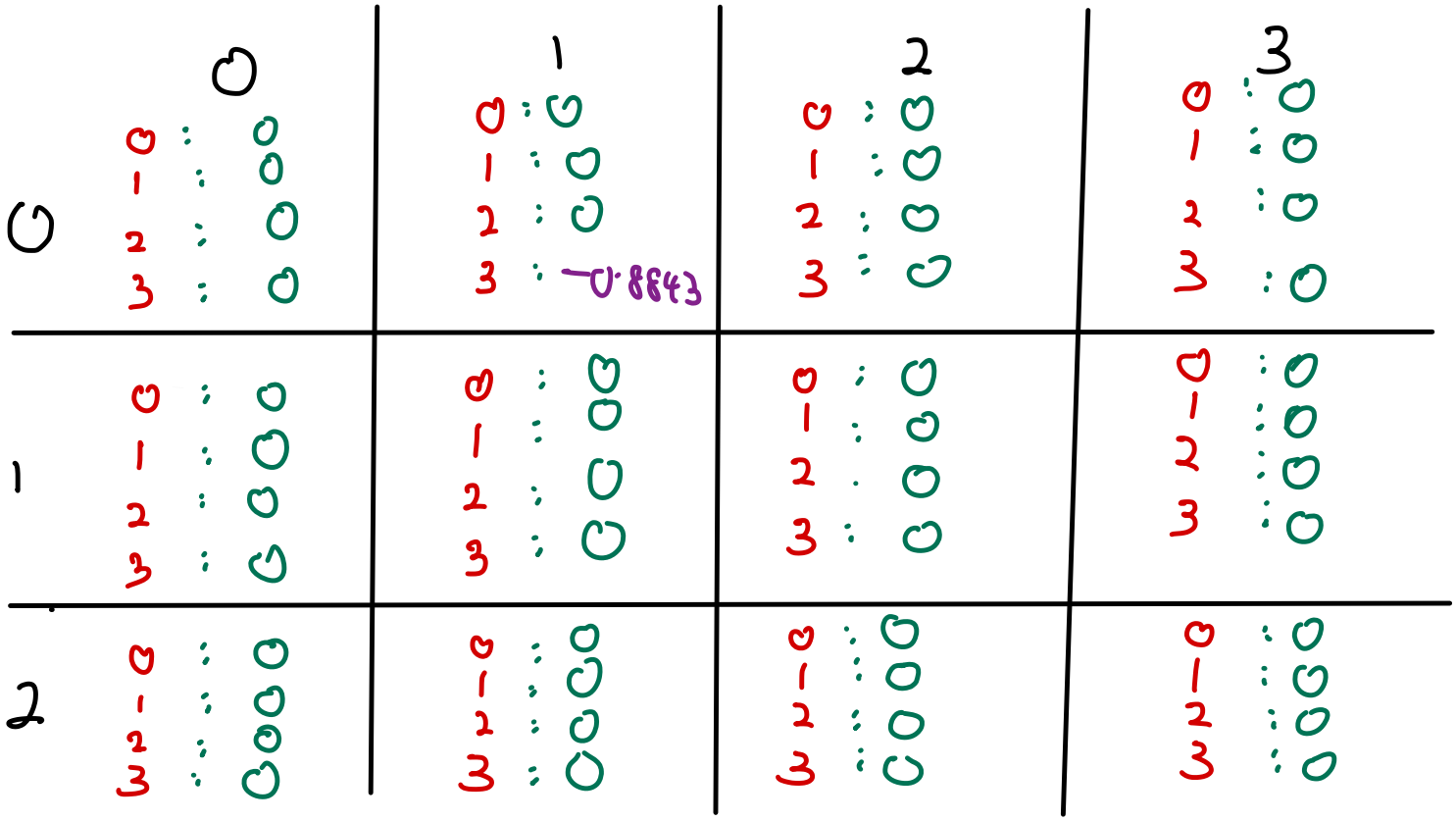
$$\text{new value} = 0 + (-4.57 - 0)(0.1)$$

$$= -0.457$$

At time step 4:

$$\text{new value} = -0.457 + (-4.73 - (-0.457))(0.1)$$

$$= -0.8843$$



Back to updating the policy,

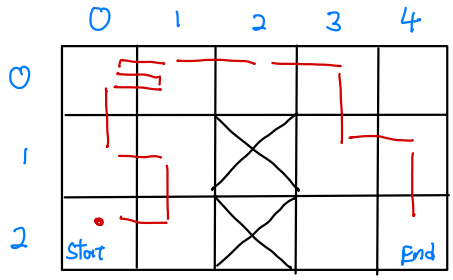
This can be done in numerous ways:

a) Policy Gradient method

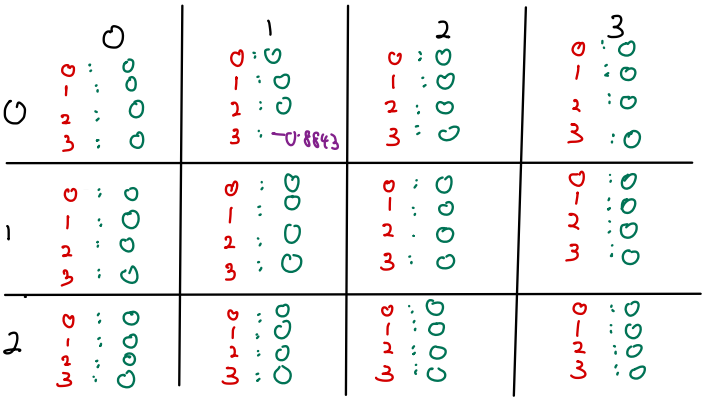
b) Policy Gradient method, with value function

c) Only using action-value function

c) 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



- Update action-value, $Q_\pi(s,a)$
  - via Monte Carlo (shown 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

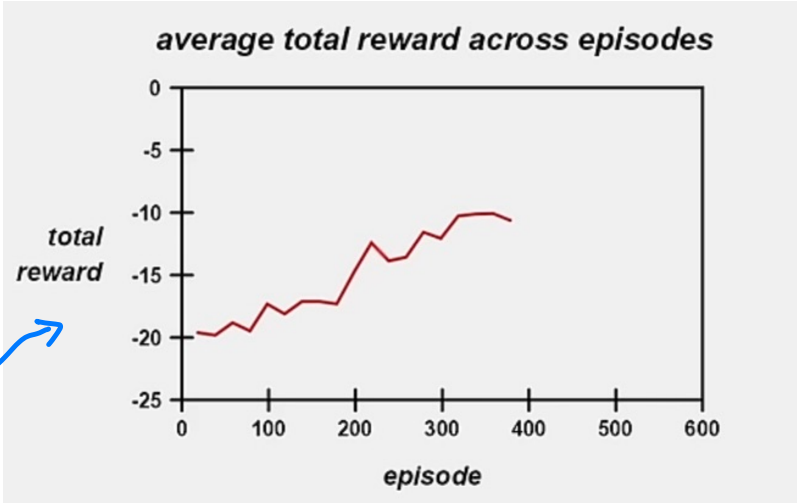
- Compute return (discounted)

↑ This cycle is known as generalized policy iteration

Approximating Q_π → Approximating Q_* (optimal action value)

Policy, π, based on Q, also approaches π_* (optimal policy)

P/s : A good way to visualize the leaning of the agent is as following :



Total reward used here is undiscanted

$$G_t = r_t + r_{t+1} + r_{t+2} \ldots$$

- Which is why start at -20 because maximum step allowed is 20 steps.
- When the agent imprares, 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, ε is introduced
  - where ε : 1) proportion of time that actions are picked randomly
    2) decreases over time as policy improves and less exploration is needed
    3) ie. 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

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

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

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

where $\mu_{k-1} = \frac{1}{k-1}\sum_{j=1}^{k-1} x_j$ and $(k-1)\,\mu_{k-1} = \sum_{j=1}^{k-1} x_j$

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

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

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

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, x_k
- But, by setting $\frac{1}{k}$ to 0.1 (α), 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 calculatable 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:

* α is learning rate

$$Q(S_t, a_t) = Q(S_t, a_t) + \alpha (g_t - Q(S_t, a_t))$$

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

which can be written as

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

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:

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

In Temporal Difference:

Approach 1: SARSA

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

Hence,

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

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

Approach 2: Expected SARSA

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,

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

$$Q(S_t, a_t) \xleftarrow{\text{Assign to}} Q(S_t, a_t) + \alpha\left(r_t + \gamma \sum_a \pi(a|S_{t+1})\, Q(S_{t+1}, a) - Q(S_t, a_t)\right)$$

Approach 3 : Q-Learning

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

Hence,

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

$$Q(S_t, a_t) \xleftarrow{\text{Assign to}} Q(S_t, a_t) + \alpha\left(r_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, a_t)\right)$$

On-Policy / Off-Policy

SARSA & Expected SARSA

- On-Policy

  because

  a) they learn from the action they already toole

  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 ε)

  b) behaviour policy $\neq$ target policy

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

Increasing sampling efficiency

Monte Carlo

SARSA

Expected SARSA

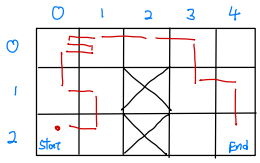
} because of temporal difference

Q-Learning

} because of temporal difference & off-policy

In Monte Carlo :

→ update of action value done once every end of episode



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

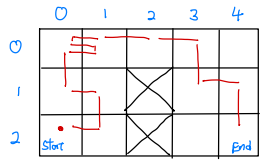
- Update action-value, $Q_\pi(S,a)$ via Monte Carlo (shown above)

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)

In Temporal difference

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



Sample 1 step

Update action value using

a) SARSA

b) Expected SARSA

c) Q learning

Chapter 4 :

Deep Q Learning

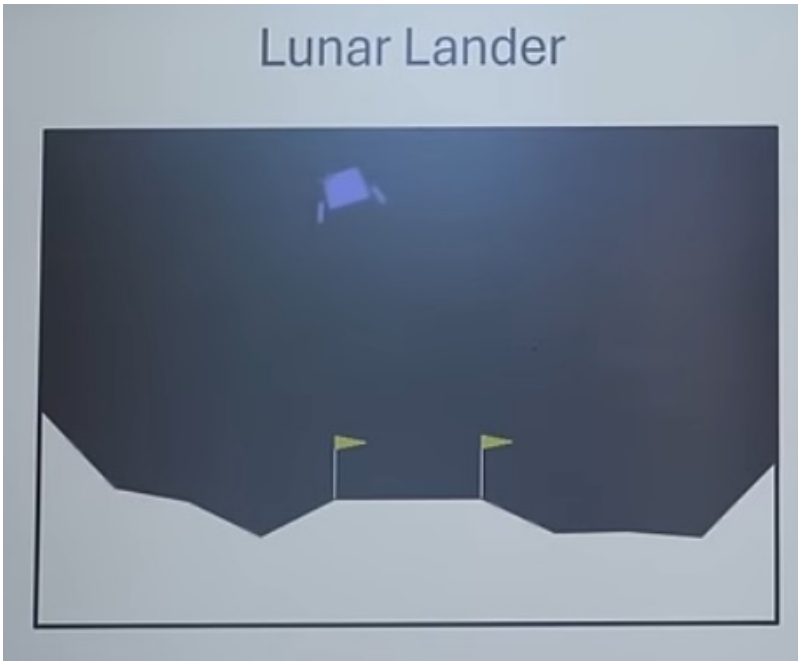
Deep Q Learning basically replaces the action value table used in Monte Carlo / Temporal Difference with a Deep Learning Network

The advantage of this modification 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 , θ
Angular Velocity , ω

} Continuous

Left Leg in contact with Ground , L
Right Leg in contact with Ground , R

} Discrete / Booleans

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

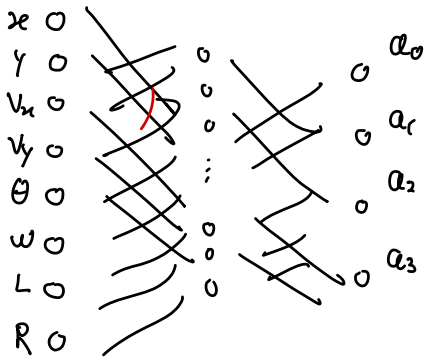
f) Decrease by 0.3 each step the main engine is firing

g) +100 if land safely or -100 for crashing

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

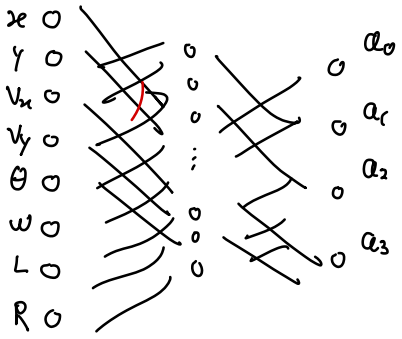
How is Deep Q Learning applied here ?

Initialization



θ
Main Network

→ use to decide actions to be taken during sampling

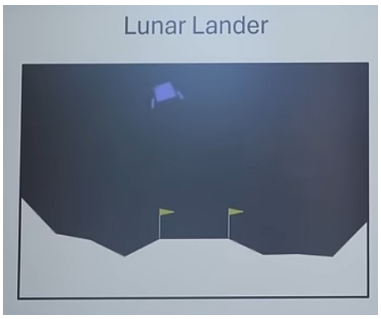


θ^-
Target Network

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

- Initialize γ (discount)
- Initialize α (learning rate)
- Initialize ε (exploration rate)

Sampling Process



Collect state information (S)

↑

S = S'

↑

Store [S, a, r, S'] in replay buffer

↑

- Receive Reward (r)
- Receive next state (S')

←

Select a = Argmax [$Q_\theta(S,a)$]

else :

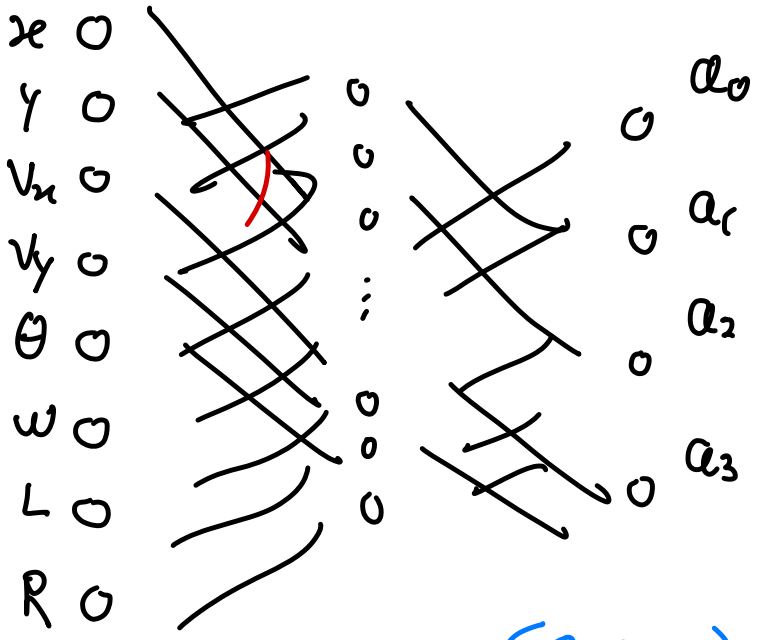
a = Randomly selected

→ Perform selected action

Sample a probability (P)

if P > ε :

→



(S)

($Q_\theta(S,a)$)

θ

Main Network

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

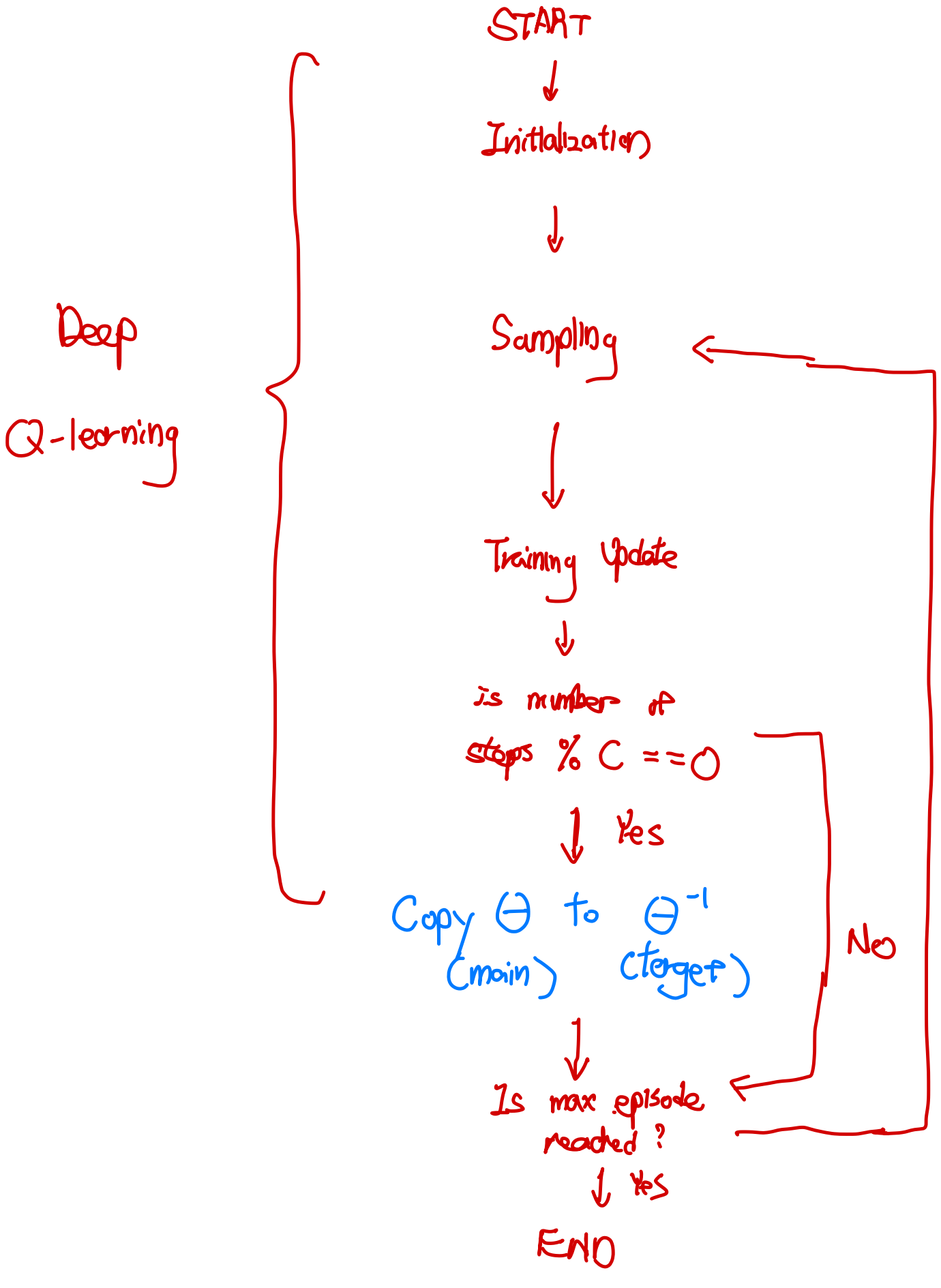
1) Pass next state, s' to the target network, θ^{-1}
2) To get a list of action value $Q_{\theta^{-1}}(s', a')$
3) From these action values, pick the one that is maximum ($\max_{a'}$)

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

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

* E → expected value → average

③ Perform Backpropagation & Gradient Descent to update the parameter in θ



Comparison between Q-Learning & Deep Q Learning

In Q-Learning :

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

$$Q(S_t, a_t) \xleftarrow{\text{Assign to}} Q(S_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, a_t) \right)$$

In Deep Q-Learning :

$$L = E\left(Y - Q_\theta(S, a) \right)^2$$

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

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,

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

② Sample a random number

- If more than epsilon, sample an action with highest probability by feeding current state to the Q network, Q_θ
- 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, termination_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, Termination_Flag, infos

- Compute target

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

- 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 :

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

(4) When t = Number of episode to explore

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

Therefore, $\varepsilon = \varepsilon_{end}$

(5)

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

- 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

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

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

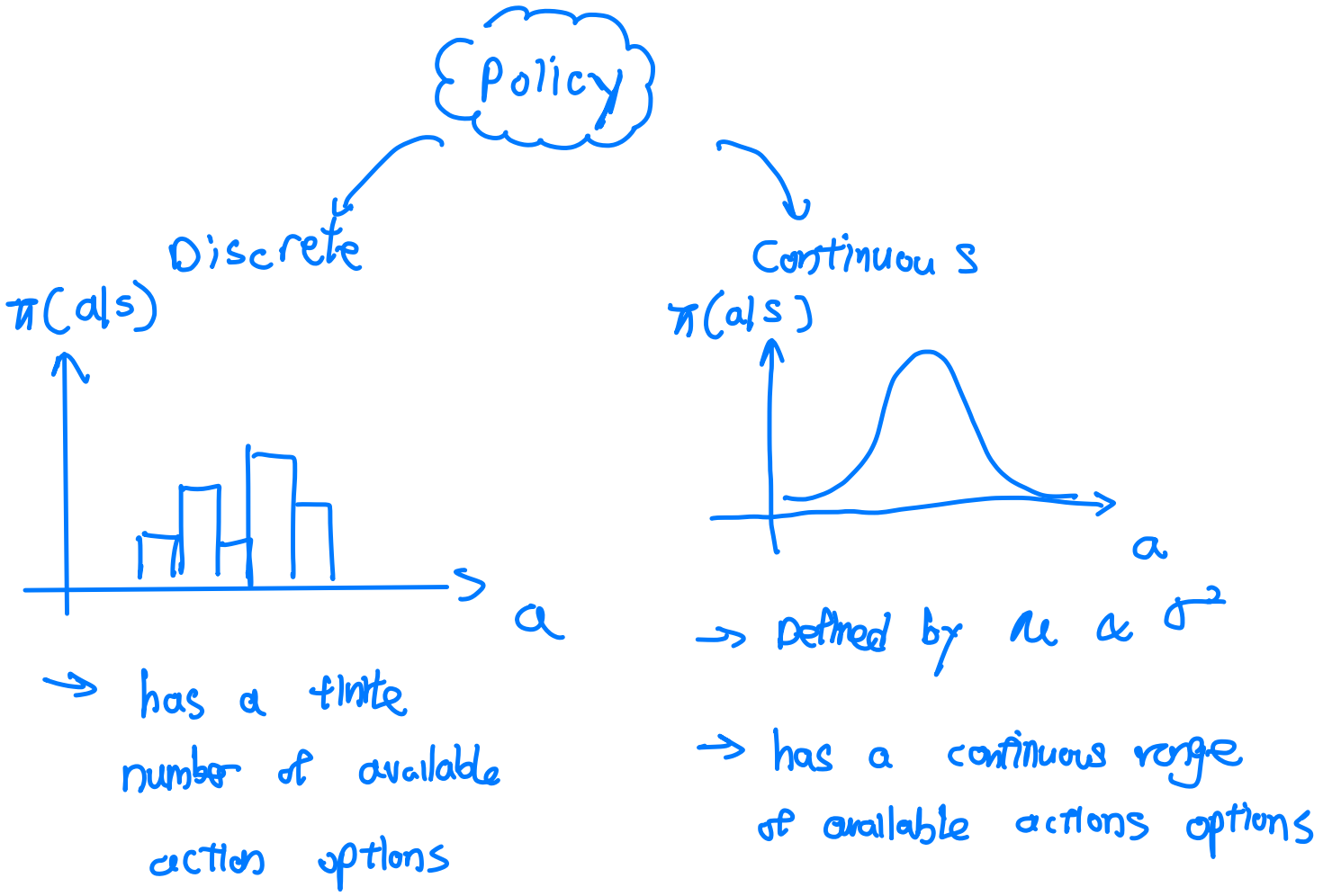
Chapter 5 :

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 ε for exploration)



→ has a finite number of available action options

→ Defined by μ & σ^2

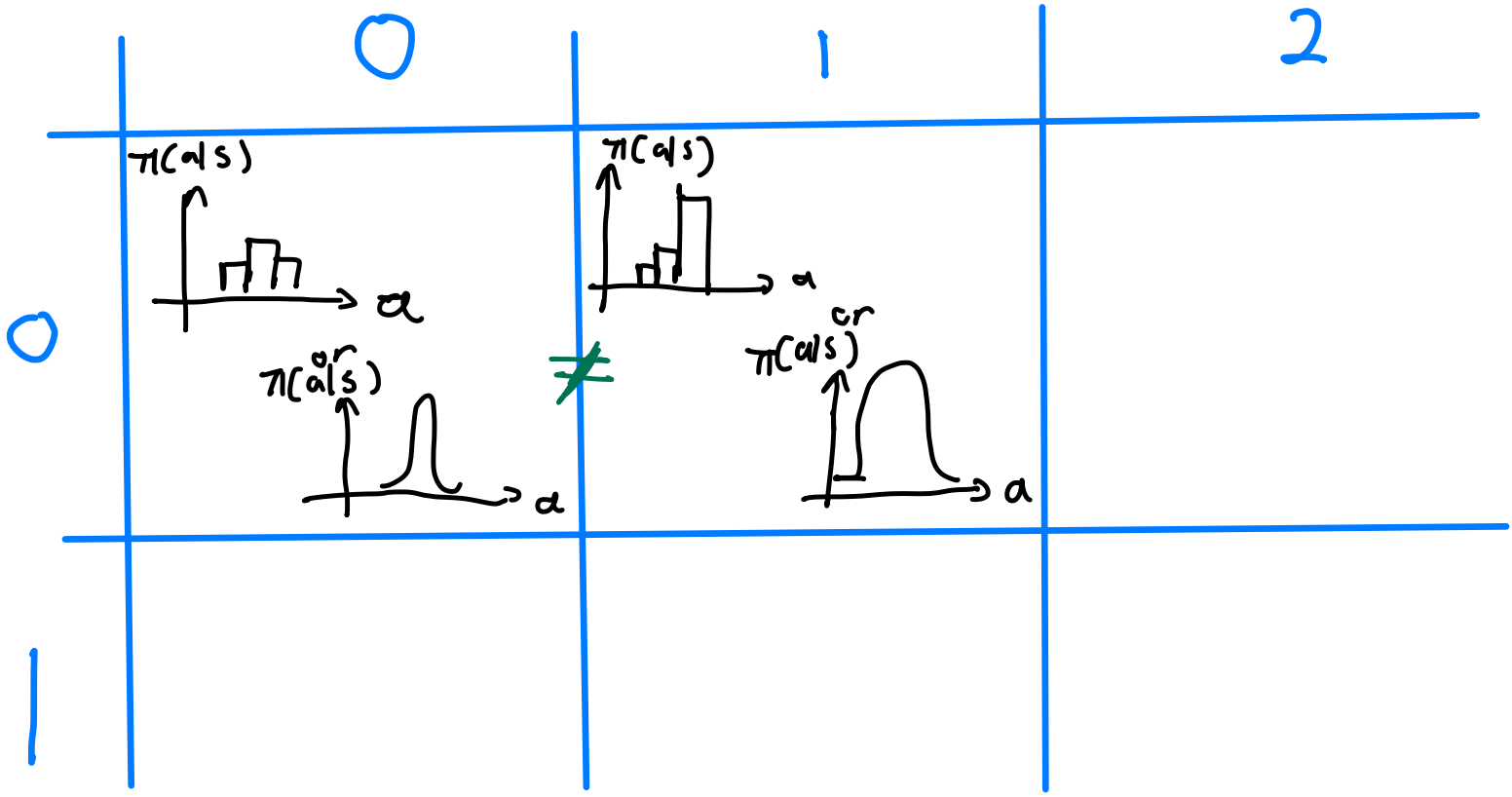
→ has a continuous range of available actions options

Some key concept :

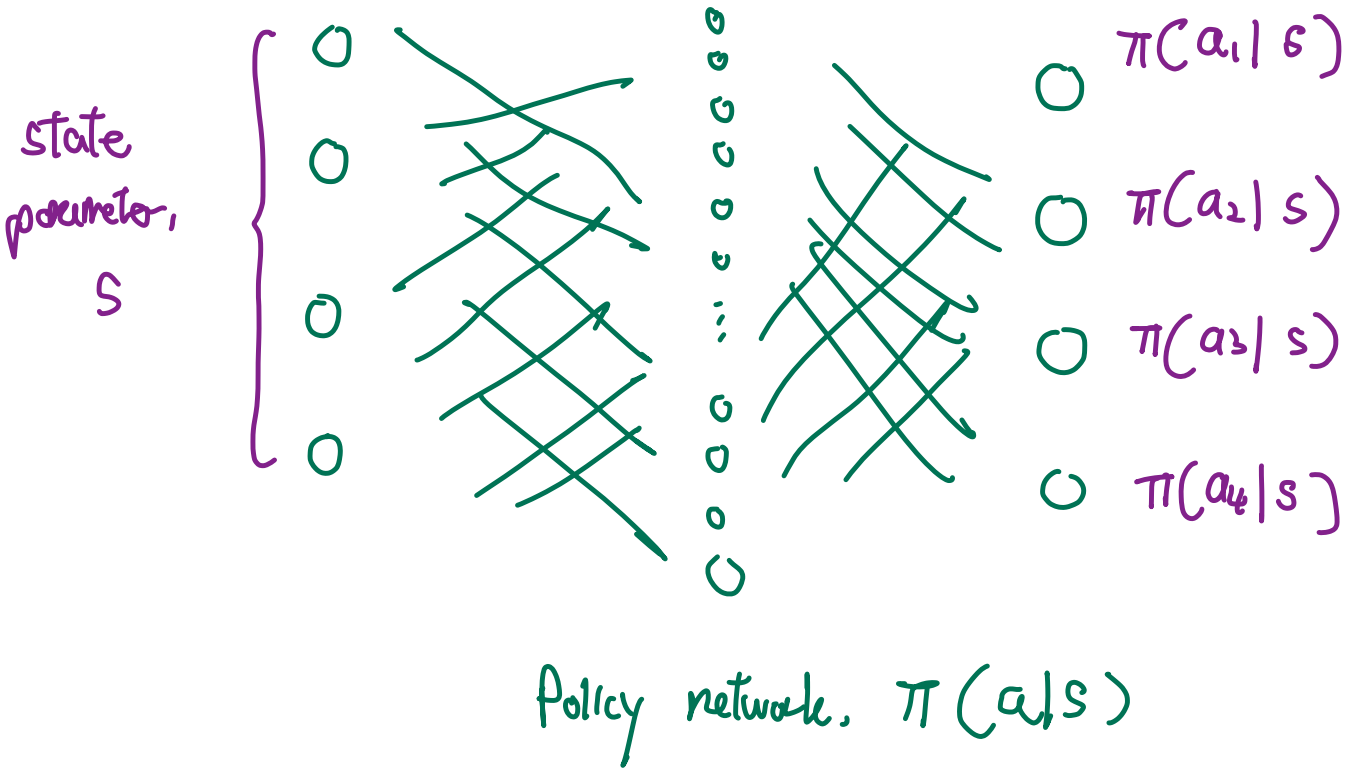
a) When a policy is involved, when decision making. a random action will be sampled from the distribution.

* 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



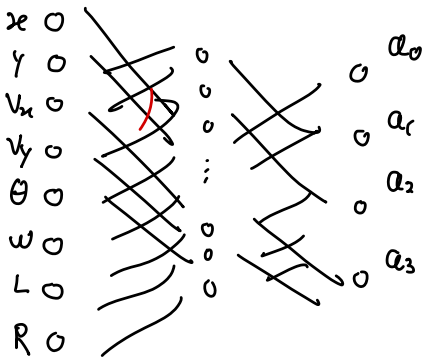
However, when a neural network is introduced to govern the policy, only 1 neural network would ever be needed



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

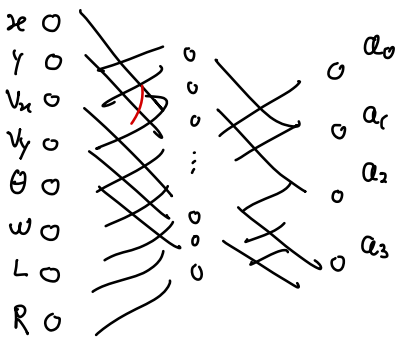
But how do you train this policy network?

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



θ
Main Network

$Q(s,a)$



θ^-
Target Network

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

[TARGET]

Compute
Loss
↓
Adjust parameter
in main network, θ
using
Gradient Descent

- But there is no Ground Truth nor Target
- in the training of the policy network, π

→ 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

$$J(\pi_\theta)$$

J ← Performance objective

π_θ ↳ of a policy network, π with θ parameter

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

① $J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{E}[R(\tau)]$

- Expected value of the reward at each step in all possible trajectories, τ, by following the π_θ policy

- $\sum$ (All possible trajectories) [Probability of a trajectory (Probability of taking action that leads to this trajectory) $\times$ Total Reward in that trajectory]

$\pi(a_1|s_0)$ → multiply
↳ where a_1 leads to s_1

$\times$

$\pi(a_3|s_1)$ → multiply
↳ where a_3 leads to s_5

$\times$

↳ $\pi(a_5|s_5)$

...

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 π with a weight of θ

τ : A trajectory / episode

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

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

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

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

γ : A discount factor

Further breakdown,

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

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

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

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

$\pi_\theta(a_t \mid S_t)$: Probability of the policy network, π with a weight of θ in asking the agent to take the action a at time step t provided an input of state s at time step t

$P(S_{t+1} \mid S_t, a_t)$: Environment dynamic. The probability of the agent ending in a state s, in the subsequent timestep, $t+1$ given that the agent took action a at timestep t when being in state s, at time step t.

In full breakdown

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

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

$$= \sum_{\tau} \left[\prod_{t} \left(P(s_{t+1} \mid s_t, a_t) \pi_\theta(a_t \mid s_t) \right) \cdot R(\tau) \right]$$

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) $P(s_{t+1} \mid s_t, a_t)$ 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 $J(\pi_\theta) = V_\pi(S_0)$

- The expected value of cumulative reward starting from initial state, S_0 following policy, π

Can be calculated using

a) 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

b) Temporal Difference - using SARSA

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

Hence,

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

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

Note:

- This method is still running average $\simeq$ average $\simeq$ 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 $V_\pi(S_0)$

③ $J(\pi_\theta) = \sum_s d_\pi(s) V_\pi(s)$

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

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

- The expected value of reward, weighted by the probability of the action & the frequency of 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 π

Performance objective, $J(\pi_\theta)$

↓ Through policy gradient theorem

For a single trajectory,

$$\nabla_\theta J(\pi_\theta) = \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \Psi_t \Big]$$

For multiple trajectories,

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

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

$\mathbb{E}\left[\sum_{t=0}^{\infty}\right]$: 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

Ψ_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

x Does not consider state difference as a)

d) $Q_\pi(S_t, a_t) - V_\pi(S_t)$

✓ Consider state difference

x Heavy computational, as requires 3 networks → policy, → Q, → V

e) $r_t + \gamma V_\pi(S_{t+1}) - V_\pi(S_t)$

Target V — Current V

* 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

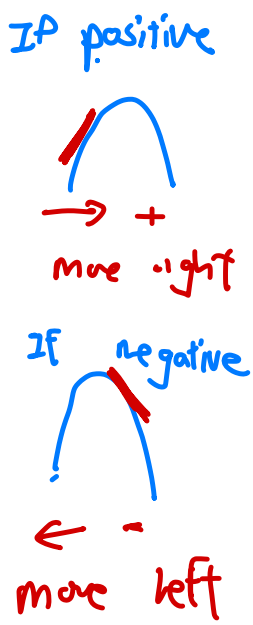
TD Error → update policy network through advantage (critic)

→ update value network (actor)

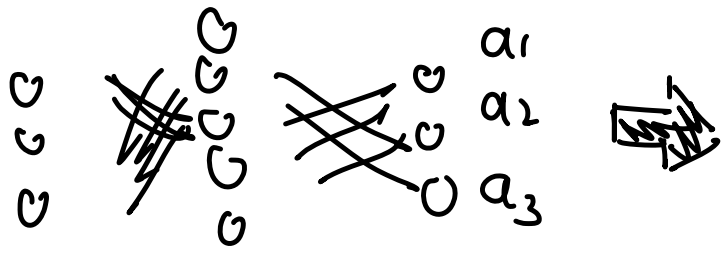
Once $\nabla_\theta J(\pi_\theta)$ is obtained, GRADIENT ASCEND can then be conducted:

$$\theta \leftarrow \theta + \alpha \Delta_\theta J(\pi_\theta)$$

α: learning rate



When policy network is to account for discrete actions



- $\pi(a_1|s)$
- $\pi(a_2|s)$
- $\pi(a_2|s)$

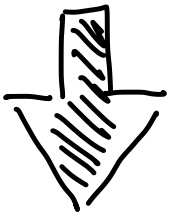
- Compute $\Delta_\theta J(\pi_\theta)$
- Update policy network via Gradient Descent

Derivation of Derivative of PERFORMANCE OBJECTIVE

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

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

$$= \sum_\tau \left[\left(\prod_{t=0}^{\infty} P(s_{t+1} | s_t, a_t) \pi_\theta(a_t | s_t) \right) (R(\tau)) \right]$$



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 $\hat{R}_t = R(\tau)$

What happens in between ...

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

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

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

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

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

Based on Derivative log trick or known as REINFORCE trick,

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

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

— Eq (1)

From Equation (1), extract $\nabla_\theta \log\left(P(\tau;\theta)\right)$

$$\nabla_\theta \log\left(P(\tau;\theta)\right)$$

$$= \nabla_\theta \log\left(\mu(S_0) \cdot \prod_{t=0}^{H} P(S_{t+1}|S_t, a_t)\,\pi_\theta(a_t|S_t)\right)$$

Note : $\mu(S_0)$ is the initial state distribution

$$= \nabla_\theta \left(\log \mu(S_0) + \log \prod_{t=0}^{H} P(S_{t+1}|S_t, a_t) + \log \prod_{t=0}^{H} \pi_\theta(a_t|S_t)\right)$$

$$= \nabla_\theta \left(\log \mu(S_0) + \sum_{t=0}^{H} \log\left(P(S_{t+1}|S_t, a_t)\right) + \sum_{t=0}^{H} \log \pi_\theta(a_t|S_t)\right)$$

Both terms are not derivated with respect to θ

$$= 0 + 0 + \nabla_\theta \sum_{t=0}^{H} \log \pi_\theta(a_t|S_t)$$

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

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

— Eq (2)

Substitute Eq ② into Eq ① :

$$\nabla_\theta J(\pi_\theta) \simeq \sum_\tau \left(P(\tau;\theta)\right)\left(\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\right)\left(R(\tau)\right)$$

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

$$\nabla_\theta J(\pi_\theta) \simeq \frac{1}{m} \sum_{i=0}^{m} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)}) \cdot R(\tau^{(i)})$$

ACTUAL IMPLEMENTATION OF DERIVATIVE OF PERFORMANCE OBJECTIVE

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

In actual implementation, a few modifications were made

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

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

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

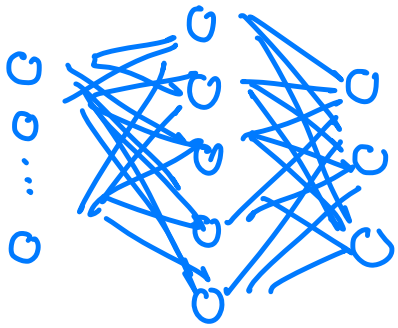
(2) In PyTorch, it is easier in implementing to minimize something rather than maximizing it.

Therefore, a negative sign is added

(3) using ln instead of log

Design of policy network to use Reinforce

a) Discrete space

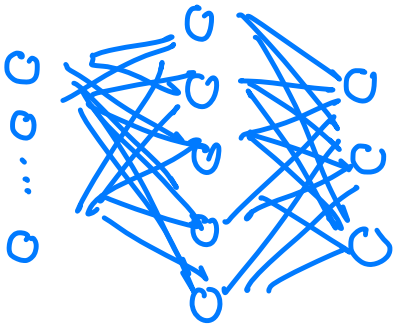


Each output node represents probability of taking a discrete action

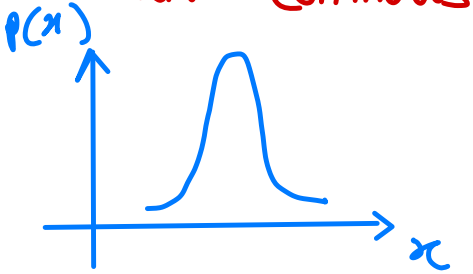
can be applied into easily

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

b) Continuous space



Each pair of output nodes represent mean & std_dev of a Gaussian probability distribution for an continuous action

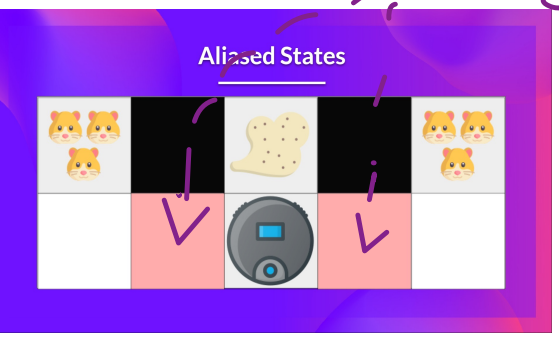


- When taking an action, you sample it from the distribution
- Based on the distribution, the probability of the action can also be determined

Advantage of Policy Gradient over Deep Q Learning

1) Can learn a stochastic policy

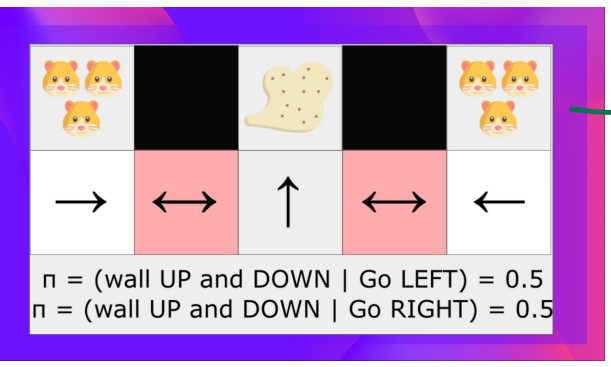
- No longer need to compute epsilon-greedy policy manually
- Get rid of perceptual aliasing



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

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

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



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

2) More efficient for continuous action space

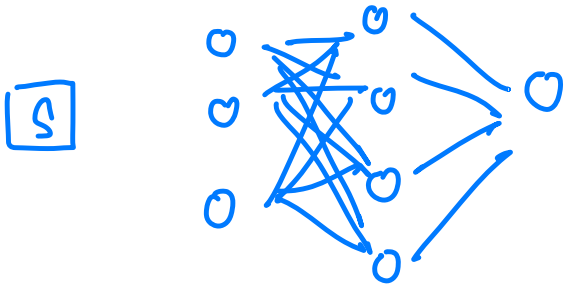
In Deep Q learning.

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

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

❗ highly inefficient in continuous action space ❗

In Policy Gradient



→ output = 67°

move at angle of 67°

3) Smoother convergence probability

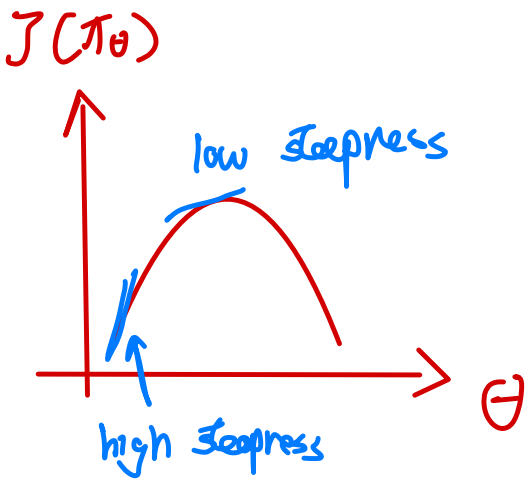
Because the action space is in continuous, the type of actions taken by the agent will also shift more gradually, allowing smoother convergence

Further clarification of the intuition behind how policy gradient works.

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

$$\theta_{new} = \theta_{old} + \alpha \nabla_\theta J(\pi_\theta)$$

- Gradient with respect to policy network's parameter
- Measuring the steepness of the landscape of probability of taking a certain action
- When it is very steep, it means that the maximum performance, is still far, as a result, $\nabla_\theta J(\pi_\theta)$ leads to a large update step
- Similary, when it is not very steep, it indicates that maximum performance is nearly achieved, hence a smaller step is taken



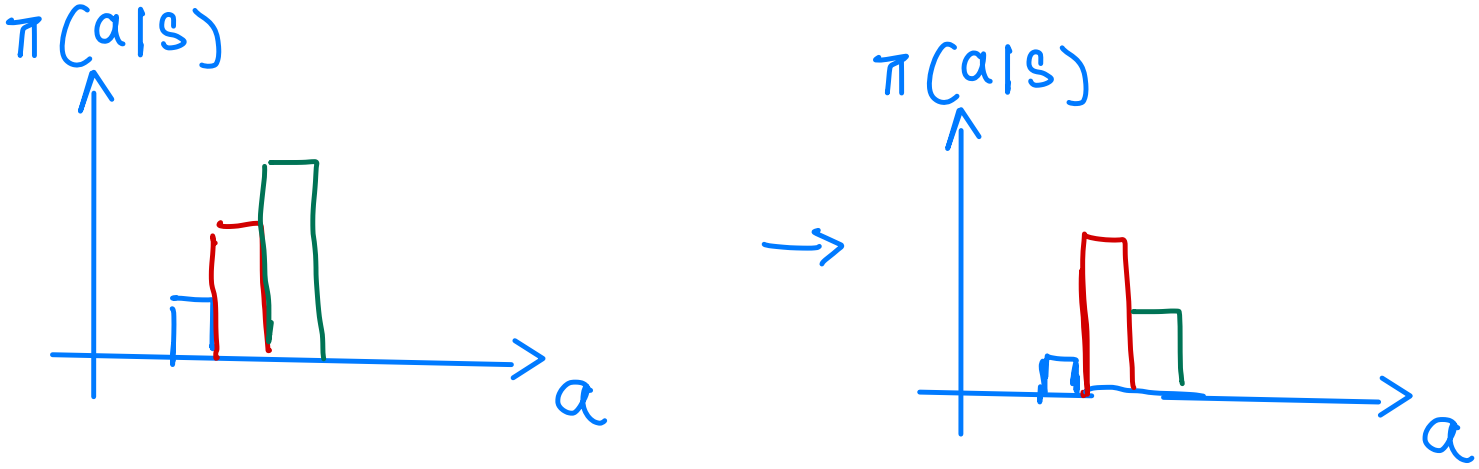
→ Discounted return

→ which can also be advantage

→ when the advantage is positive, the parameter will be updated TOWARDS it

→ Else, if the advantage is negative, the parameter will be updated AWAY from it.

Visualization of policy update in discrete actions



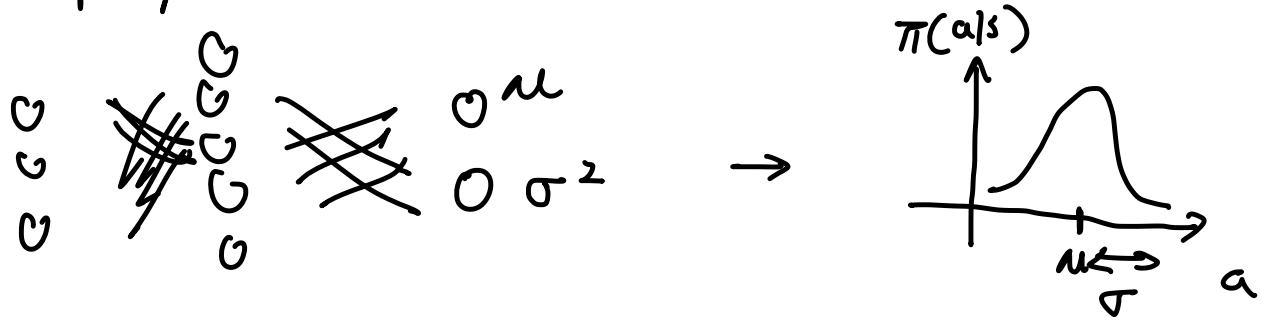
- Due to the evaluation made by the advantage, 卐, a_2 is deemed a good action, receive a high $\Delta_\theta J(\pi_\theta)$ where the parameter in the policy network are tuned in such a way that leads to higher $\pi(a_2|s)$.

- 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.

卐 a_2 → get positive advantage

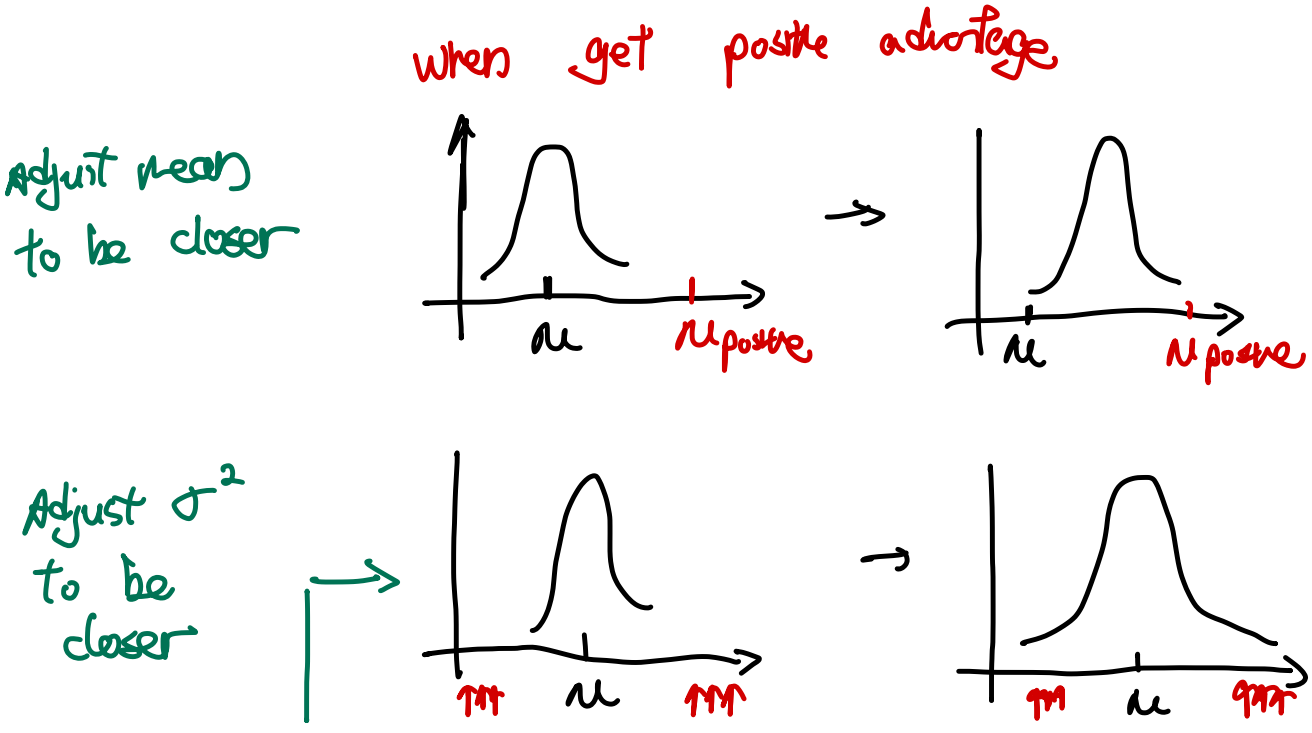
卐 a_1, a_3 → get negate advantage

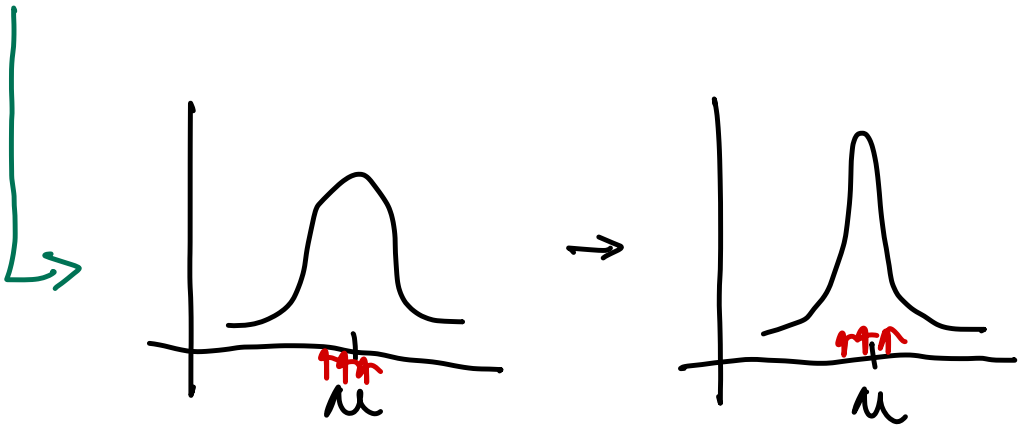
When policy network is to account for continuous actions



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

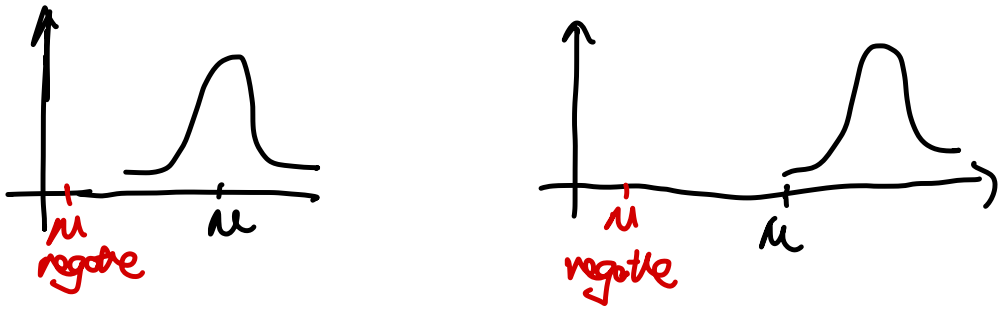
Visualization of policy update in continuous actions



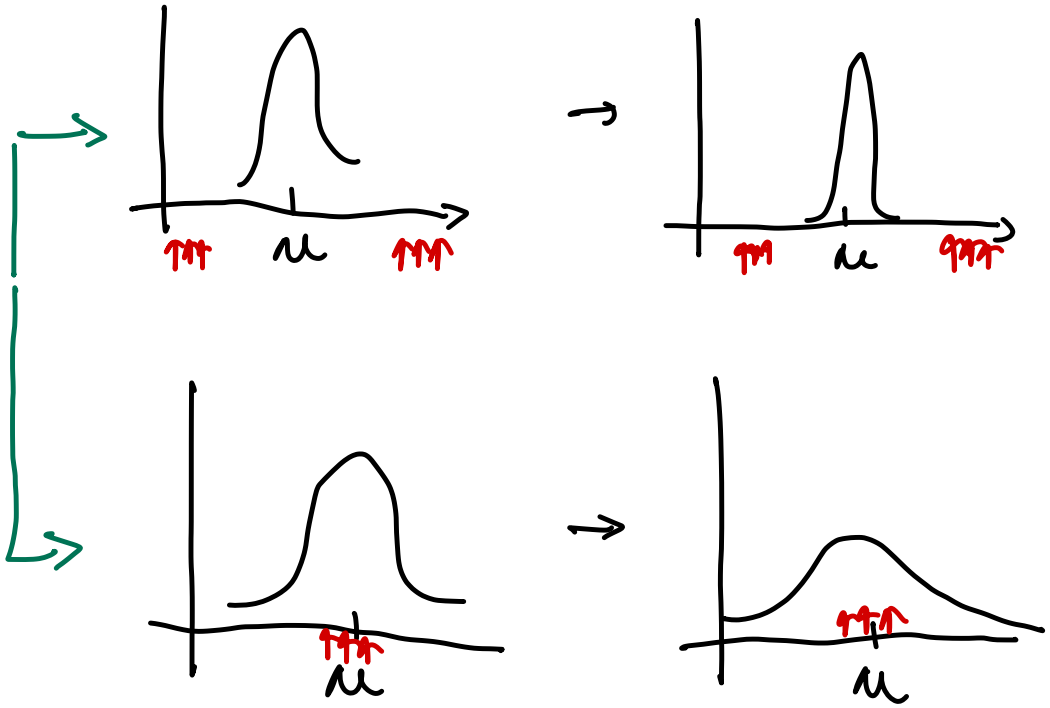


When get negative advantage

Adjust mean to be further



Adjust σ^2 to be further



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 action 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 reinforce 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:

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

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

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

which can be generalised into

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

A_t → Advantage
- Advantage of taking this action in this stage compared to other actions

As displayed above, $\Psi_t = R(\tau) = g_t(\tau)$ in reinforce algorithm.

There are many options for Ψ_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

- ✗ bad because some state is naturally closer to the goal, hence has higher g_t
- ✗ Not a good way to evaluate each action
- ✗ 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
- ✗ 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 → policy, → Q, → V

e) $\underbrace{r_t + \gamma V_\pi(S_{t+1})}_{\text{Target } V} - \underbrace{V_\pi(S_t)}_{\text{Current } V}$

* 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 → update policy network through advantage (critic)

→ update value network (actor)

How 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, $\hat{Q}_w(S_t, a_t)$

  $\hat{Q}_w$ — An action value network, $\hat{Q}$ 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

$$\nabla_\theta (\pi_\theta) = \nabla_\theta \log \pi_\theta (S, a) \cdot \tilde{\psi}$$

$$= \nabla_\theta \log \pi_\theta (S, a) \cdot \underbrace{\hat{Q}_w (S, a)}$$

Advantage function selected here is the action value

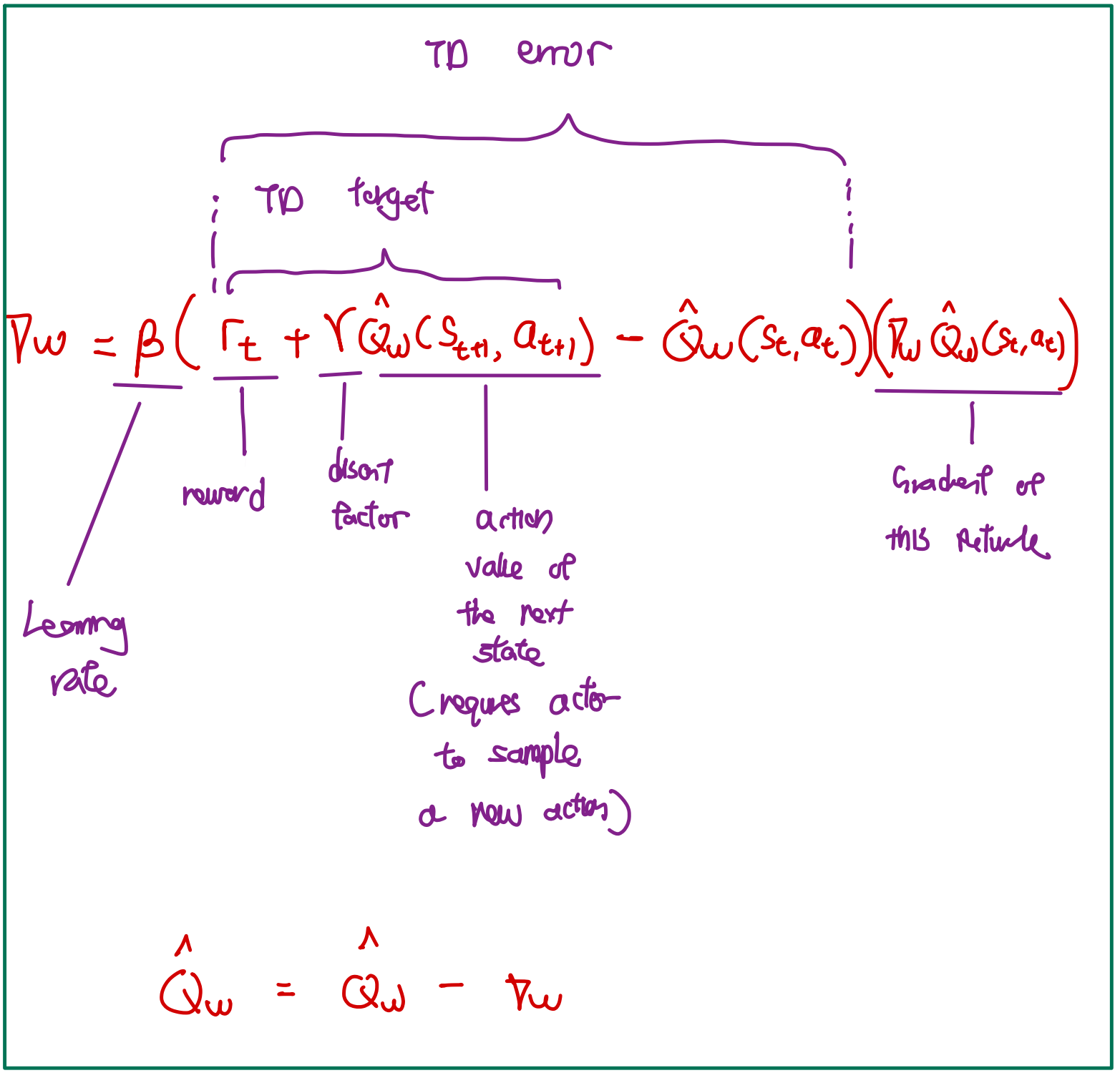
$$\pi_\theta = \pi_\theta + \underbrace{\alpha} \nabla_\theta (\pi_\theta)$$

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)



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 → 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 given 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 & 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 & errors 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 human annotate the right action to take under such noise

↳ Bad because tedious annotation

Chapter 8 :

Curiousity - Intrinsic Curiousity Module (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.

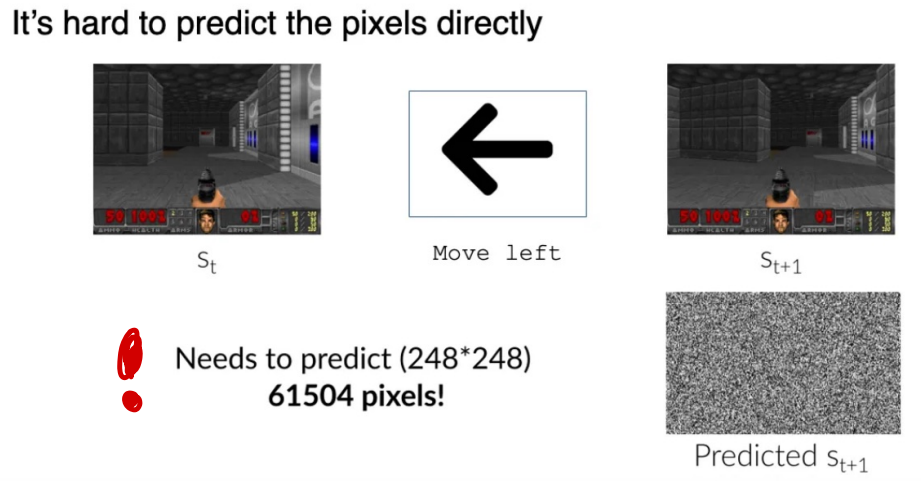
Solution : Intrinsic Reward

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

$$IR = \| \underbrace{\hat{s}_{t+1}}_{\text{Predicted next state}} - \underbrace{s_{t+1}}_{\text{Actual next state}} \|$$

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



Solution :

- Therefore instead of directly predicting the full state, the model should predict the corresponding feature representation

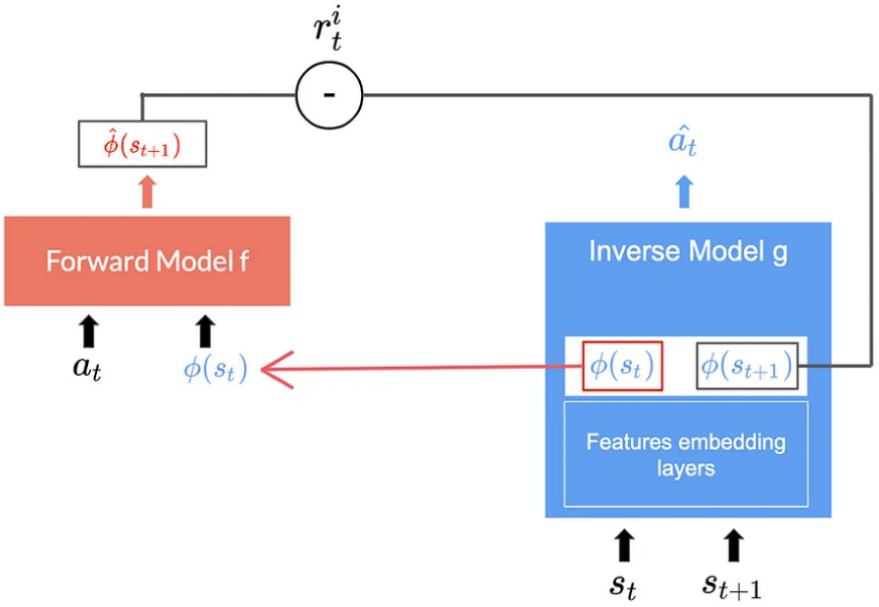
- A few rules to follow in modelling the state's feature representation are :

  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 Curiousity Module (ICM):



Red: Forward Model f

Blue: Inverse Model g

$\hat{\phi}(s_{t+1})$: Prediction of the feature representation of st+1

$\phi(s_t)$ $\phi(s_{t+1})$: Feature representation of st and st+1

r_t^i: Intrinsic reward at timestep t (curiosity reward).

Inverse Model, g

- responsible to compress state into lower dimensional feature representation

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

- $\hat{a}_t$: predict action taken
- s_t: current state
- s_{t+1}: next state
- θ_I: weight of inverse model

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

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

- $H(\hat{a}_t, a_t)$: 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

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

- $\hat{\phi}(S_{t+1})$: Predicted feature representation of next state
- $\phi(S_t)$: Feature representation of current state
- a_t: Action taken
- θ_f: Weight of forward model

- The loss that is associated with this model:

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

- $\hat{\phi}(S_{t+1})$: Predicted feature representation of next state
- $\phi(S_{t+1})$: Actual feature representation of next state

Intrinsic Curiosity

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

- η: Scale factor, > 0
- $\hat{\phi}(s_{t+1})$: Predicted feature representation of next state
- $\phi(s_{t+1})$: Actual feature representation of next state

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

Overall optimization problem

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

- $\min_{\theta_P, \theta_I, \theta_F}$: Optimize parameters of policy, inverse model and forward model
- $\mathbb{E}_{\pi(s_t; \theta_P)}[\Sigma_t r_t]$: Average reward for episodes / trajectories when the agent follows policy π
- $\lambda > 0$, is a scalar that weighs the importance of the policy gradient loss against the importance of learning the intrinsic reward
- L_I: Inverse Model Loss
- L_F: Forward Model Loss
- β: Scalar (between 0 and 1) that weighs the inverse model loss against the 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 Curiousity Module

1) Noisy TV

- Since the curiousity is calculated by predicting the next state, where the harder it is to predict the next state, the higher the intrinsic reward / curiousity, the even more likely that the agent will explore that area

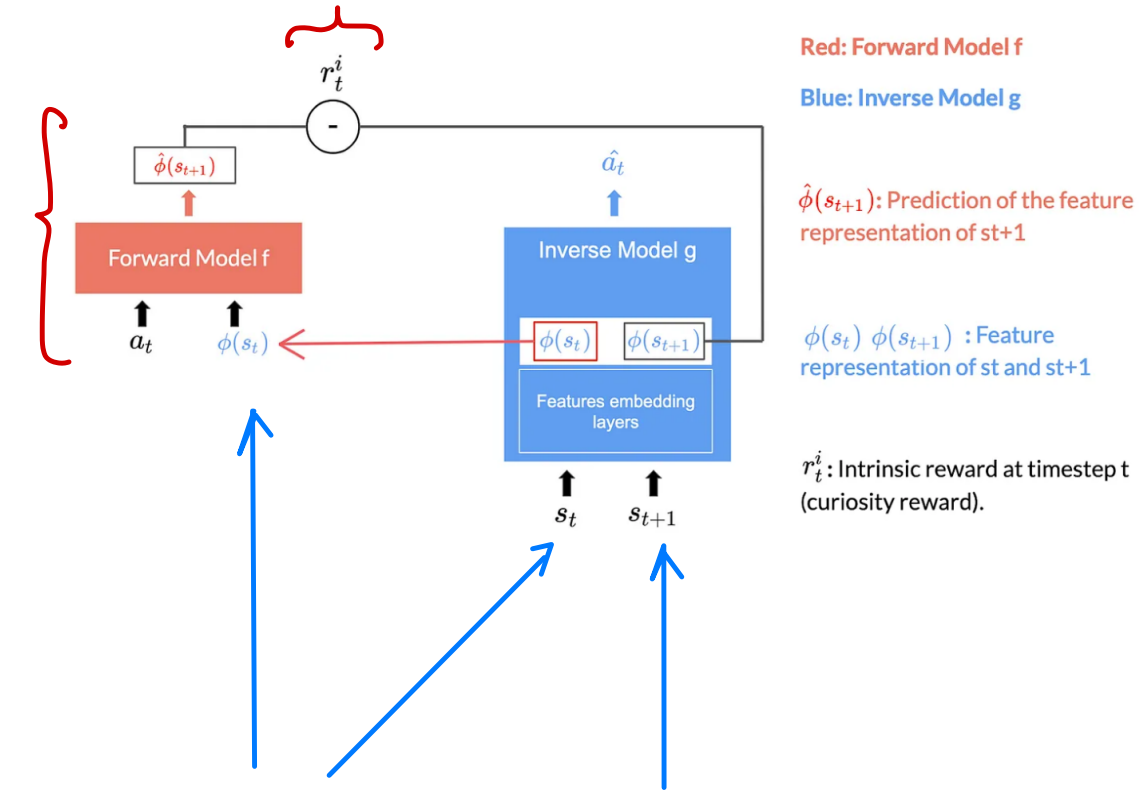
- However, when the next state is random & stohastic, ie. a TV problem, the agent has a hard time predicting it & thought there's high curiousity, therefore drawn to it & stopped performing meaningful exploration

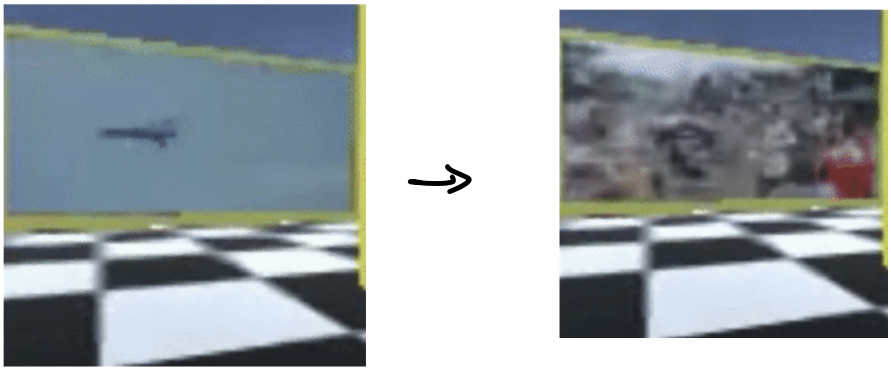
- The agent has a hard time predicting because inverse model rely on the prediction of action taken to tweak the inverse model's parameter which subsequently affect the feature representation.

- With TV 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

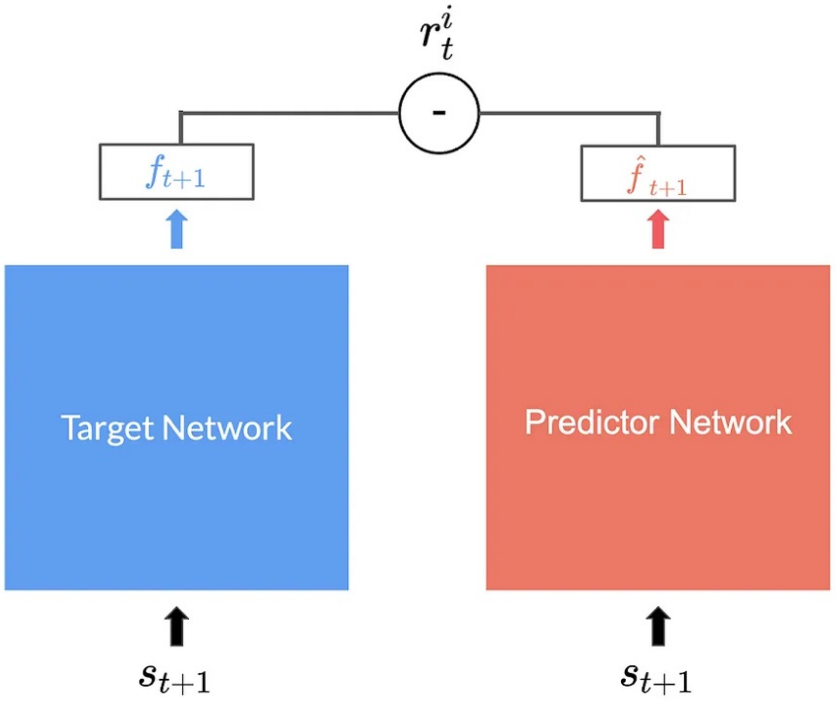
Has a hard time predicting the next state





Next state is always random

Solution : Introducing Random Distillation Network



Target network is randomly initialized

Target network will output a fixed feature representation of s_{t+1}

Predictor Network will tries to predict the target network's output $\hat{f}_{t+1}$

Target network

- Randomly initialized
- Never trained & 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

chanced 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 of TV Noise, due to ease of matching, the agent's curiosity of it will decrease & it moves on to perform other tasks.

Intrinsic reward

$$r_t^i = \| \underbrace{\hat{f}(s_{t+1})}_{\text{Predicted feature representation of next state (output by predictor network)}} - \underbrace{f(s_{t+1})}_{\text{Actual feature representation of the next state (output by target network)}} \|_2^2$$

Chapter 10:

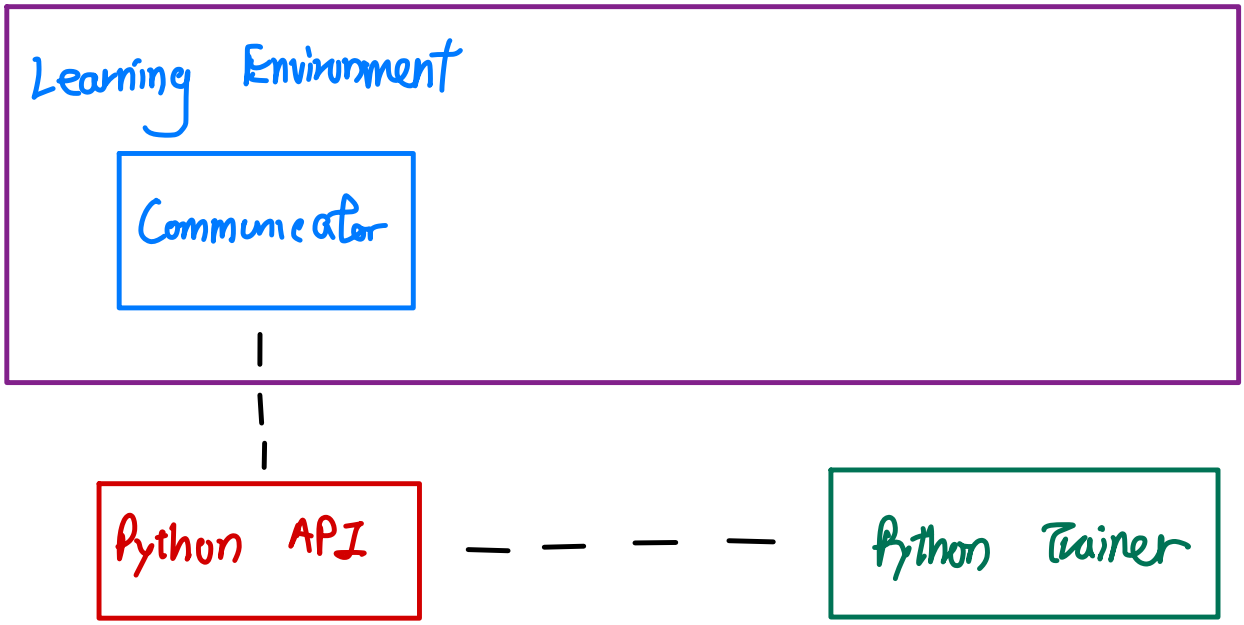
Introduction to Unity ML

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

- Contain Unity Scene
- Contain 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 algorithms developed with PYTORCH

(5) Gym Wrapper

- Encapsulate learning environment in a gym wrapper

(6) Porting Zoo Wrapper

- Multi agent versions of gym wrappers

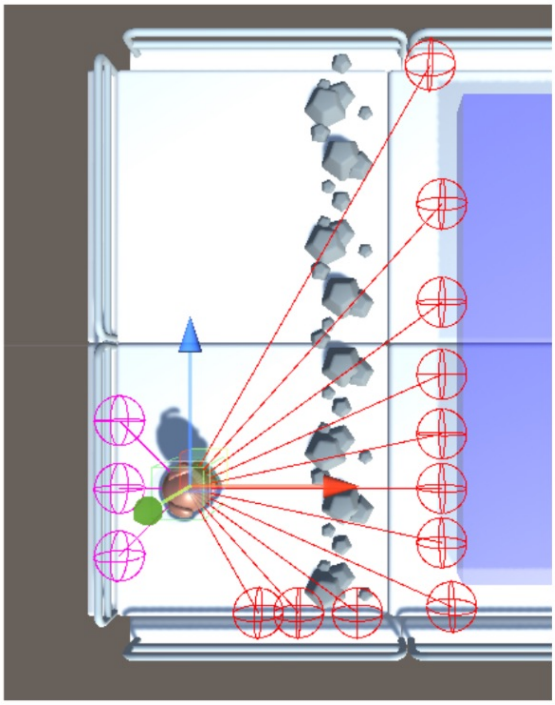
within learning components

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:

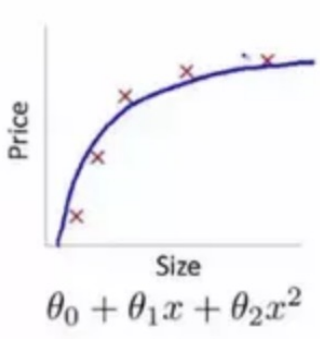
Bias & Variance

Tradeoff in RL

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

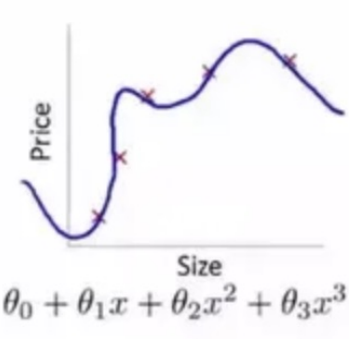
$\theta_0 + \theta_1 x$

High bias
(underfit)



$\theta_0 + \theta_1 x + \theta_2 x^2$

"Just right"



$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

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

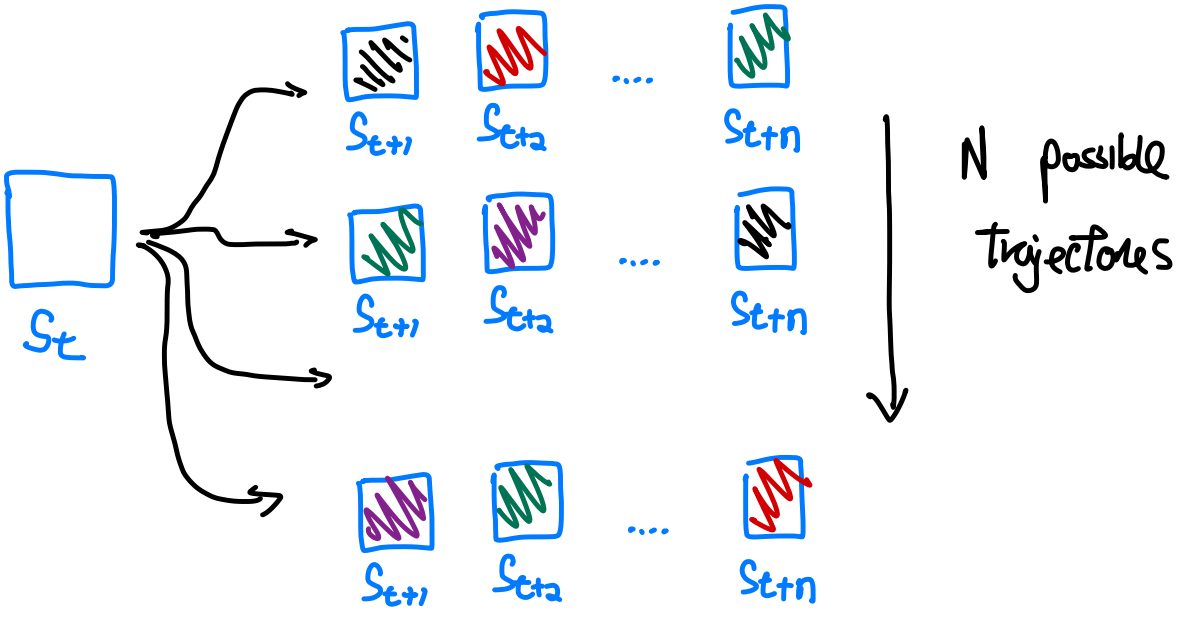
Similarly, bias & variance problem also exist in RL, but in a different way

> 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

Explanation 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
- It estimates g_t by only comparing with the subsequent state. For instance :

Approach 1 : SARSA

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

Hence,

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

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

Approach 2 : Expected SARSA

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,

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

$$Q(S_t, a_t) \xleftarrow{\text{Assign to}} Q(S_t, a_t) + \alpha\left(r_t + \gamma \sum_a \pi(a|S_{t+1})\, Q(S_{t+1}, a) - Q(S_t, a_t)\right)$$

Approach 3 : Q-Learning

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

Hence,

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

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

- 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.

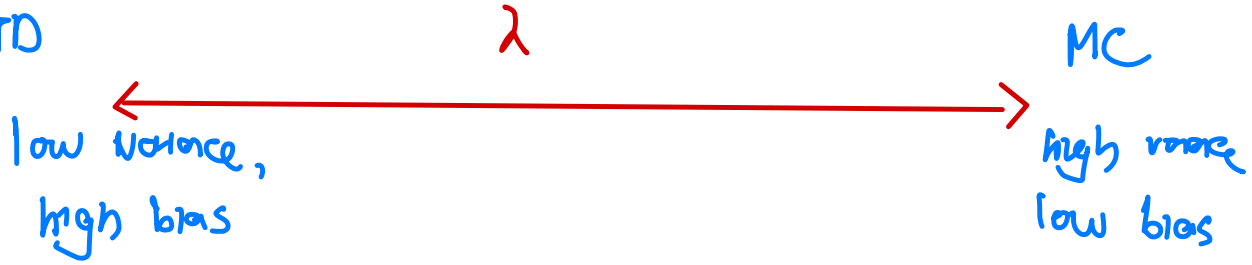
So, how can it be handled?

Factor 1: Discount Factor, γ

- 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, λ

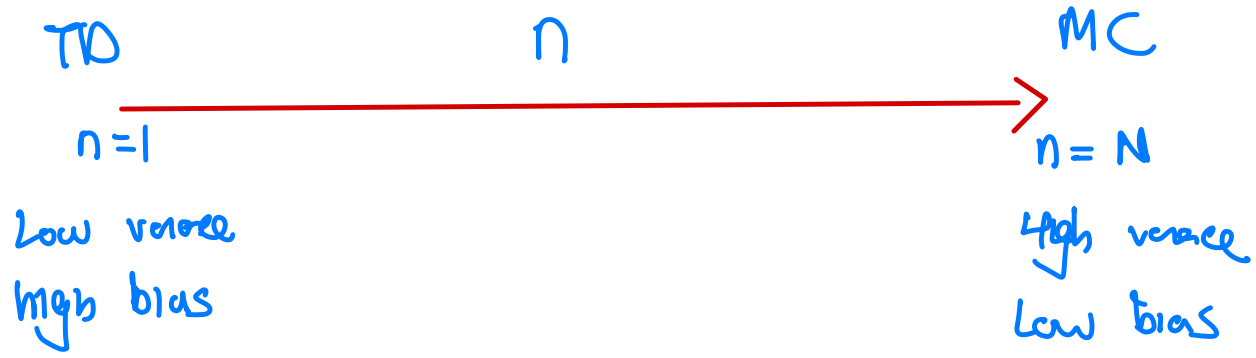
- Typically, advised to be kept in the range of 0.95 – 0.99
- The lower it is, the closer it behaves as TD.



- 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 cant be too long that it leads to high variance
  - Refer to Chapter 13
  - Can be used in combination for GAE



Chapter 12:

Generalised Advantage Estimate (GAE)

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

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

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 :

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

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 :

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

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

δ_{t+l} : TD sampled Advantage

γ : Gamma Discount Factor

λ : Lambda — Tunable parameter to adjust bias & variance

When $\lambda = 0$

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

$$= \underbrace{0^0}_{1} \delta_t + \underbrace{0^1 \delta_{t+1} + 0^2 \delta_{t+2} \cdots}_{0}$$

$$= \delta_t$$

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

When $\lambda = 0$,

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

When $\lambda = 1$,

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

$$= \delta_t + \gamma \delta_{t+1} + \gamma^2 \delta_{t+2} \cdots$$

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

$$+ \gamma r_{t+1} + \gamma^2 V(S_{t+2}) - \gamma V(S_{t+1})$$

$$+ \gamma^2 r_{t+2} + \gamma^3 V(S_{t+3}) - \gamma^2 V(S_{t+2}) \cdots$$

If observed closely,

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

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

δ_{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 :

$\lambda = 0$ → $\lambda = 1$

Low variance

high bias

High Variance

Low Bias

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

$$A_0^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_l$$

$$= \delta_0 + (\gamma\lambda)\delta_1 + (\gamma\lambda)^2\delta_2 + (\gamma\lambda)^3\delta_3 \ldots\ldots$$

$$A_1^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{1+l}$$

$$= \delta_1 + (\gamma\lambda)\delta_2 + (\gamma\lambda)^2\delta_3 \ldots\ldots$$

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

$$A_2^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{2+l}$$

$$= \delta_2 + (\gamma\lambda)\delta_3 + (\gamma\lambda)^2\delta_4 \ldots\ldots$$

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

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

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

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

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

Example of $n = 2$

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

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

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

$$= r_t + \gamma V(s_{t+1}) - V(s_t) + (\gamma\lambda)(r_{t+1} + \gamma V(s_{t+2}) - V(s_{t+1})) + (\gamma\lambda)^2 (r_{t+2} + \gamma V(s_{t+3}) - V(s_{t+2}))$$

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 = Final time step − Current time step

Meanwhile

Temporal Difference calculates the target using 1 step

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

$$1 \leq n \leq N$$

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,

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

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

In SARSA,

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

with n-step bootstrapping

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

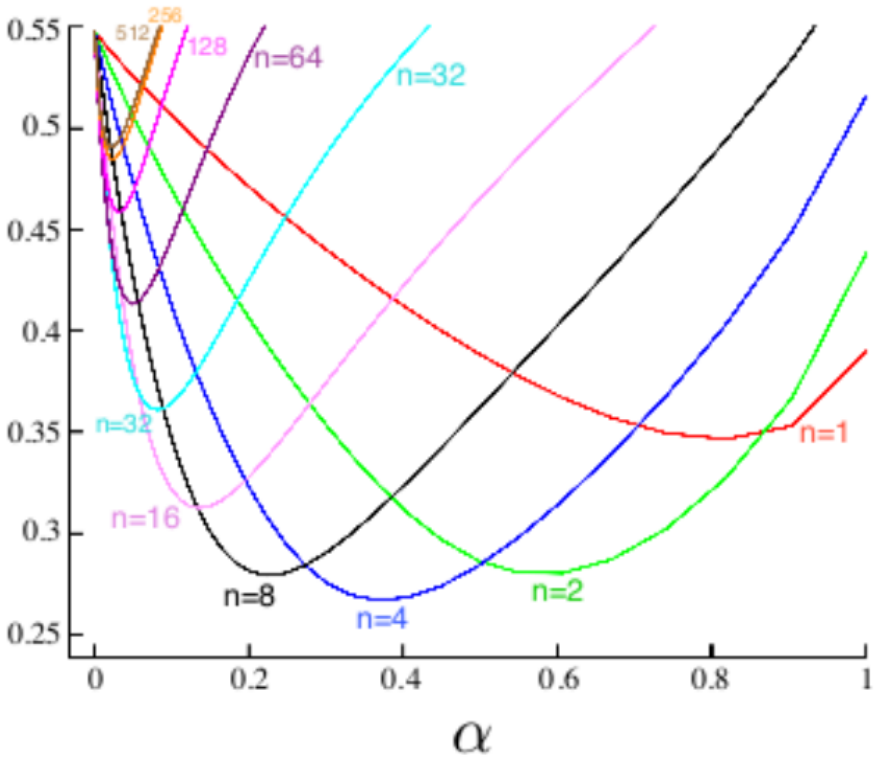
Assuming if $n = 2$

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

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

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

Average RMS error over 19 states and first 10 episodes



Chapter 14 :

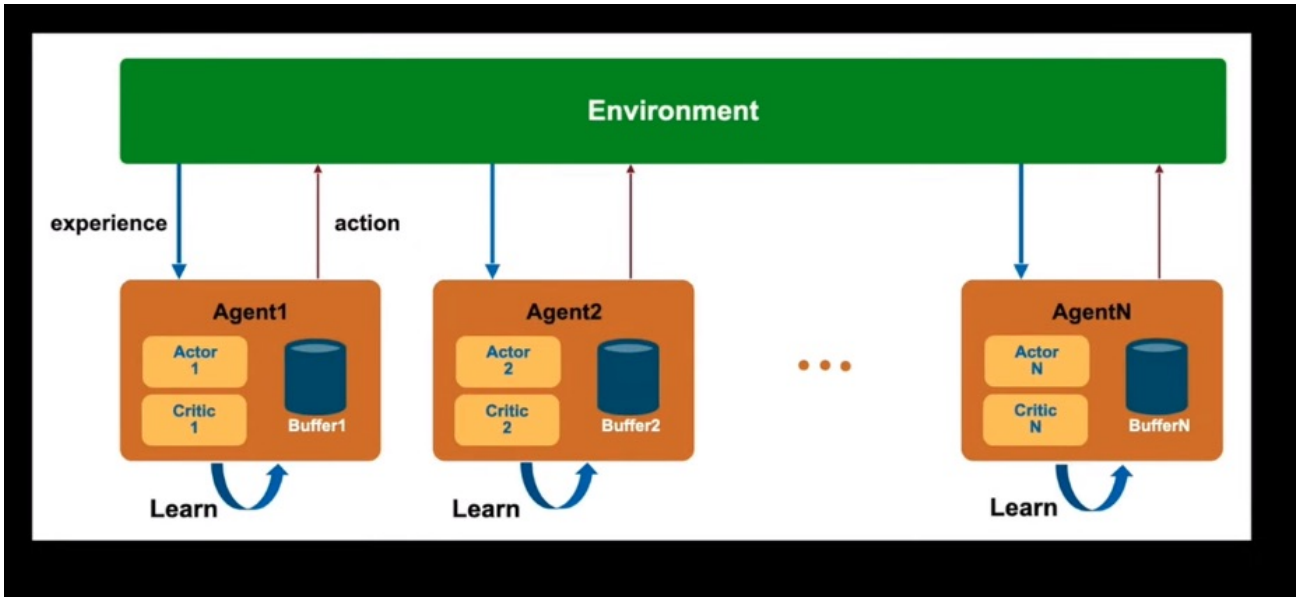
Multi - Agent

Reinforcement Learning

(MARL)

This section serves to give a high lever 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

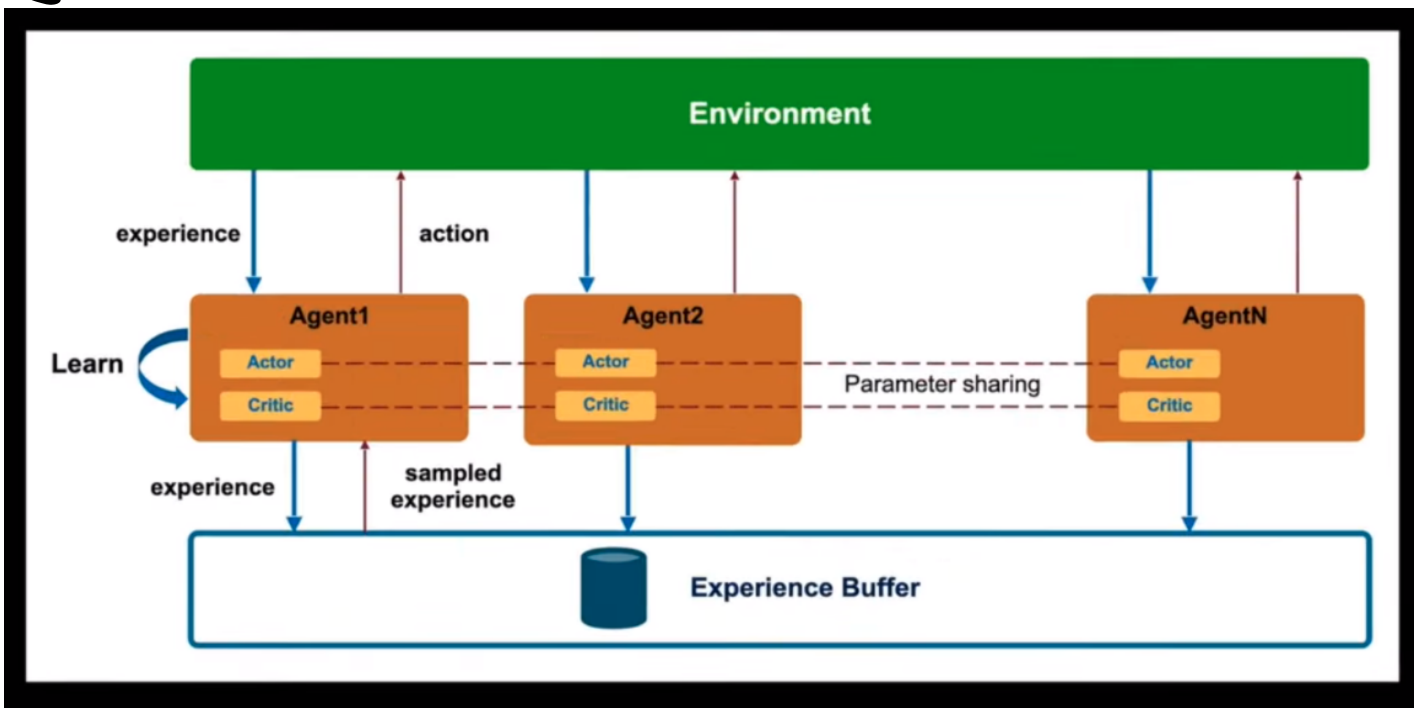
- The interaction of other agents with the environment
  => is not known by the agent
  => is treated as part of the environment when the result is treated as part of the environment dynamics

- As a result,

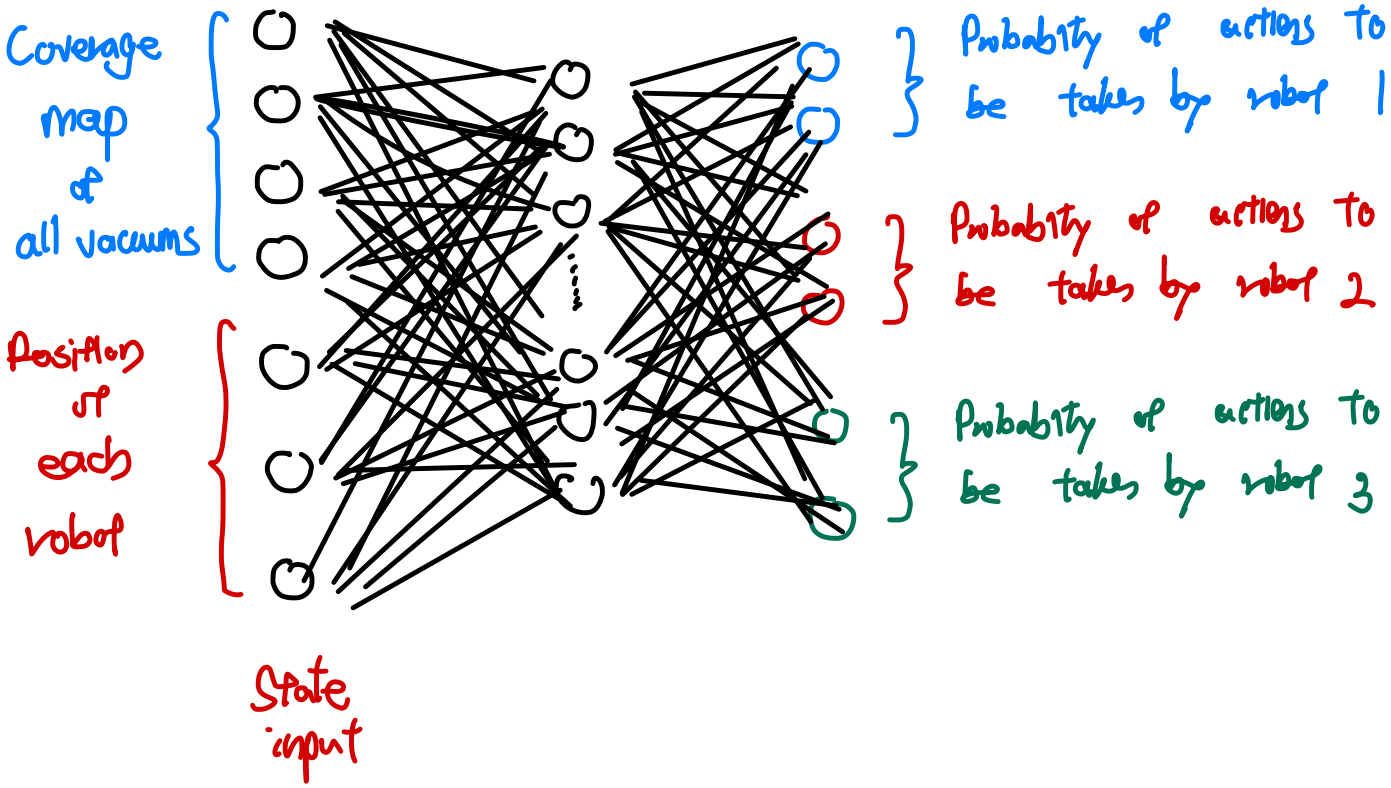
  => the agents have to learn in a "non-stationary environment" which is very difficult for convergence given many RL algorithms are built on the foundation of Markov Decision Process (MDP)

  => 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 agent 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 hyperparameter 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 x save_steps
Typical range : 4x – 10x 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 ther 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

→ Probabilty 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 training unstability

→ 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 level 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) Cant 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

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

$$E_B = \frac{1}{1 + 10^{(R_A - R_B)/400}}$$

E_A, E_B : Expected score

R_A, R_B : 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

$S_A / S_B = 1$ if win

$S_A / S_B = 0.5$ if tied

$S_A / S_B = 0$ if lose

Chapter 17 :

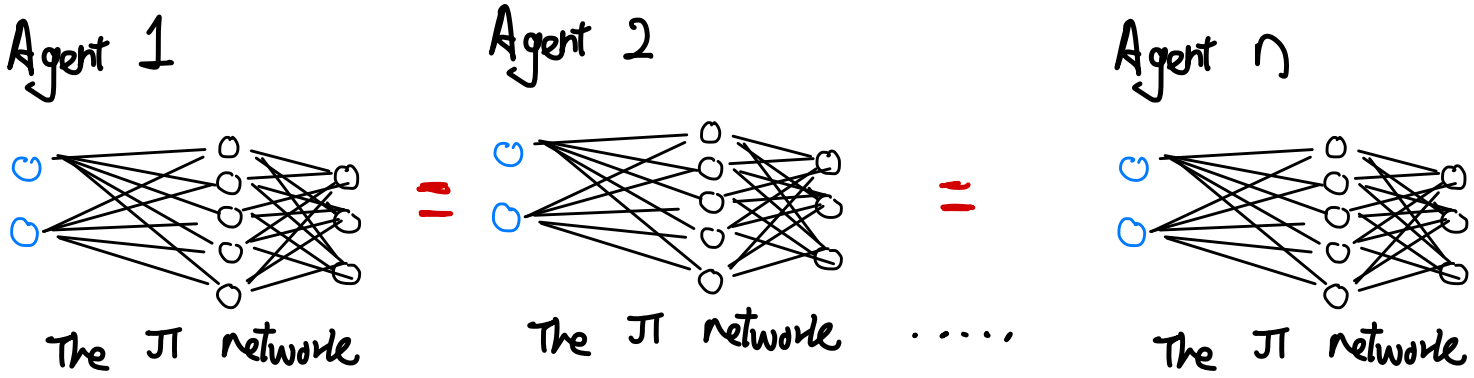
Multi-Agent

Posthumous Credit Assignment

(MA-POCA)

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

a) Policy Network

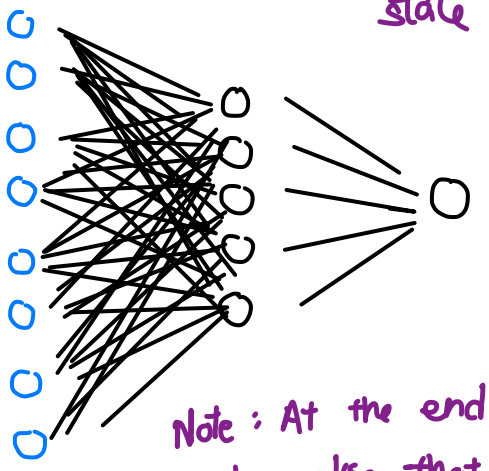


- All agent shares the same policy network with the same parameter
- The networks takes in state input relative to the agent that it currently govern only
- The network then outputs the probability of taking each action for its agent.

b) Value Network

But still its accurate that value network only take input state from 1 agent

State information collected by only 1 agent



Quality of this state

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 networks.

- 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

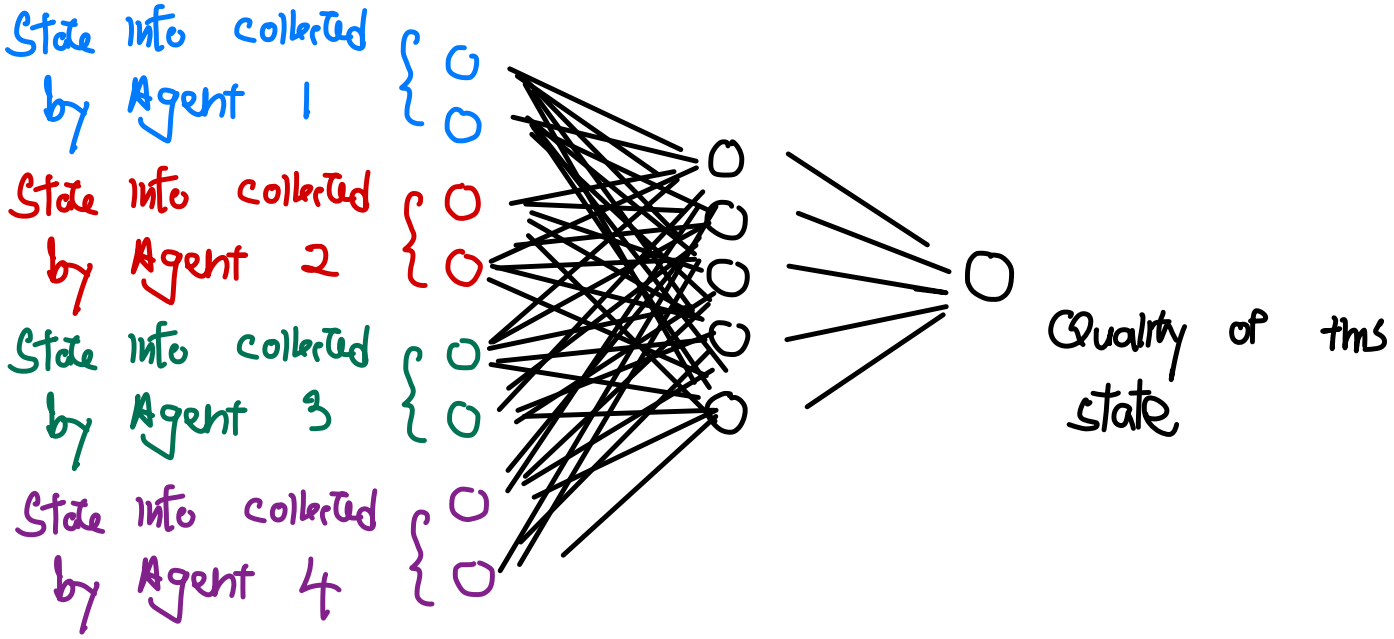
  → To accomodate a state input size that is corresponding to a time step where there's maximum number of agents

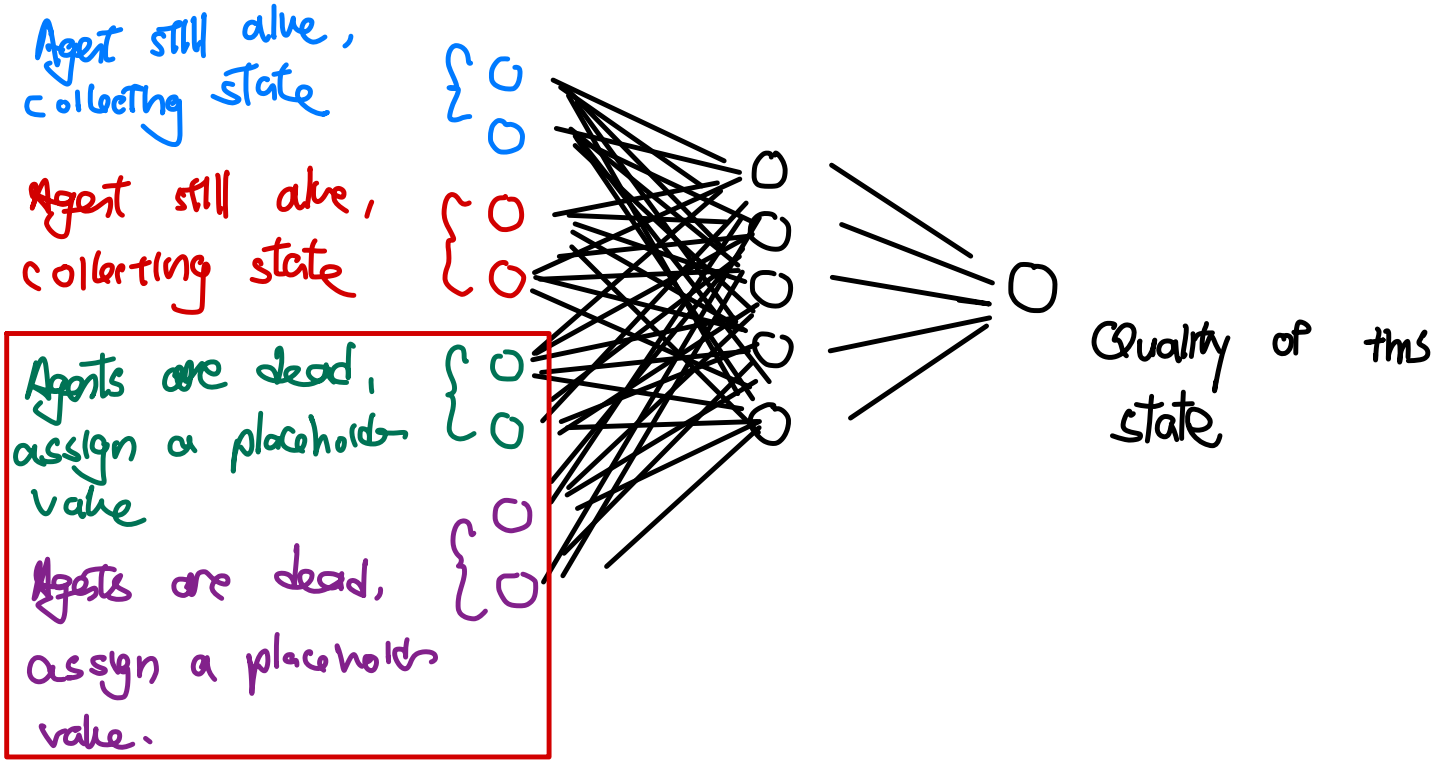
  → When the number of existing agents is less than the maximum number of agents, the

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





→ 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

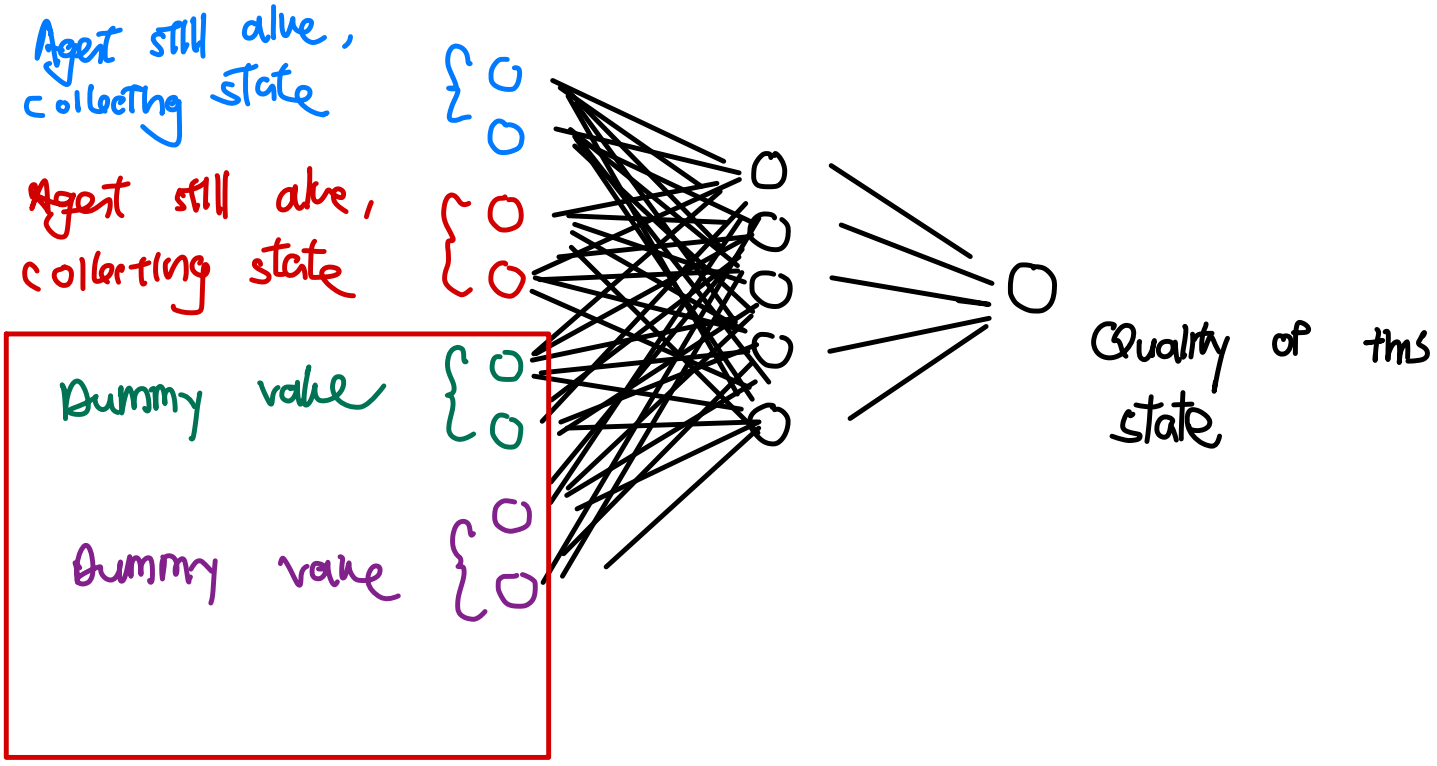
- For agents that are terminated early, it will not get to experience the rewards given to the group after termination
- Therefore, these agents will have hard time to learn if their actions are valuable for the group
- Absorbing state allows the state value beyond an agent's termination to be propagated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

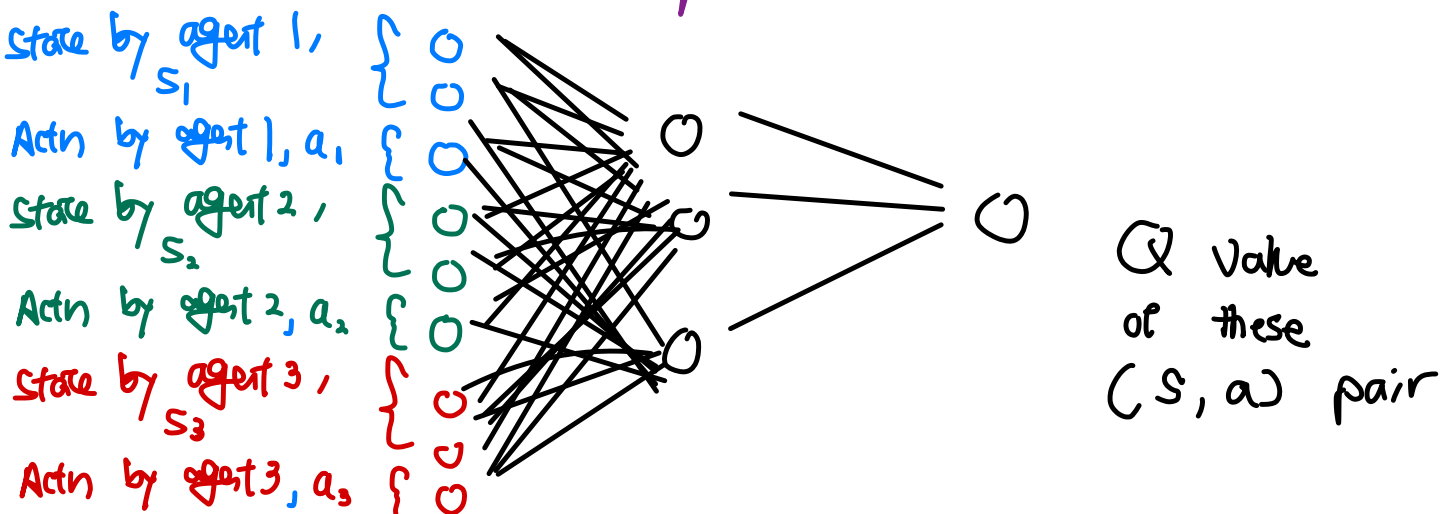
∴ As a summary, as much as absorbing state helps to solve many problems in Multi-Agent Learning, it is not the most optimal solution

Before finally introducing MA-POCA 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 bette 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

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

$$b_i(s, a) = \underset{a' \sim \pi_i(\cdot|o_i)}{E}\left[Q^{\pi}(s, (a^{-i}, a'))\right]$$

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,

$$b_i(s, a) = \sum_{a'} \pi_i(a_i' | s_i) \cdot Q^{\pi}(s, (a^{-i}, a'))$$

$\pi_i(a_i' | s_i)$: Probability of taking action a' based on the policy with the current state of s_i for agent i

$Q^{\pi}(s, (a^{-i}, a'))$: Value of when agent i takes action a' 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 examine 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

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

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

$$\nabla_{\theta_i} J(\theta_i) = \mathop{E}_{\substack{s \sim \rho^{\pi} \\ a^i \sim \pi_i}} \left[\nabla_{\theta} \log \pi_i(a_i | o_i) \cdot \text{Advantage}_i \right]$$

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

$$\theta_{update} = \theta_{old} + \underbrace{\alpha}_{\text{Learning rate}} \underbrace{\sum_i^N \nabla_{\theta_i} J(\theta_i)}_{\text{Sum of Gradient across all agents}}$$

MA-POCA introduces ...

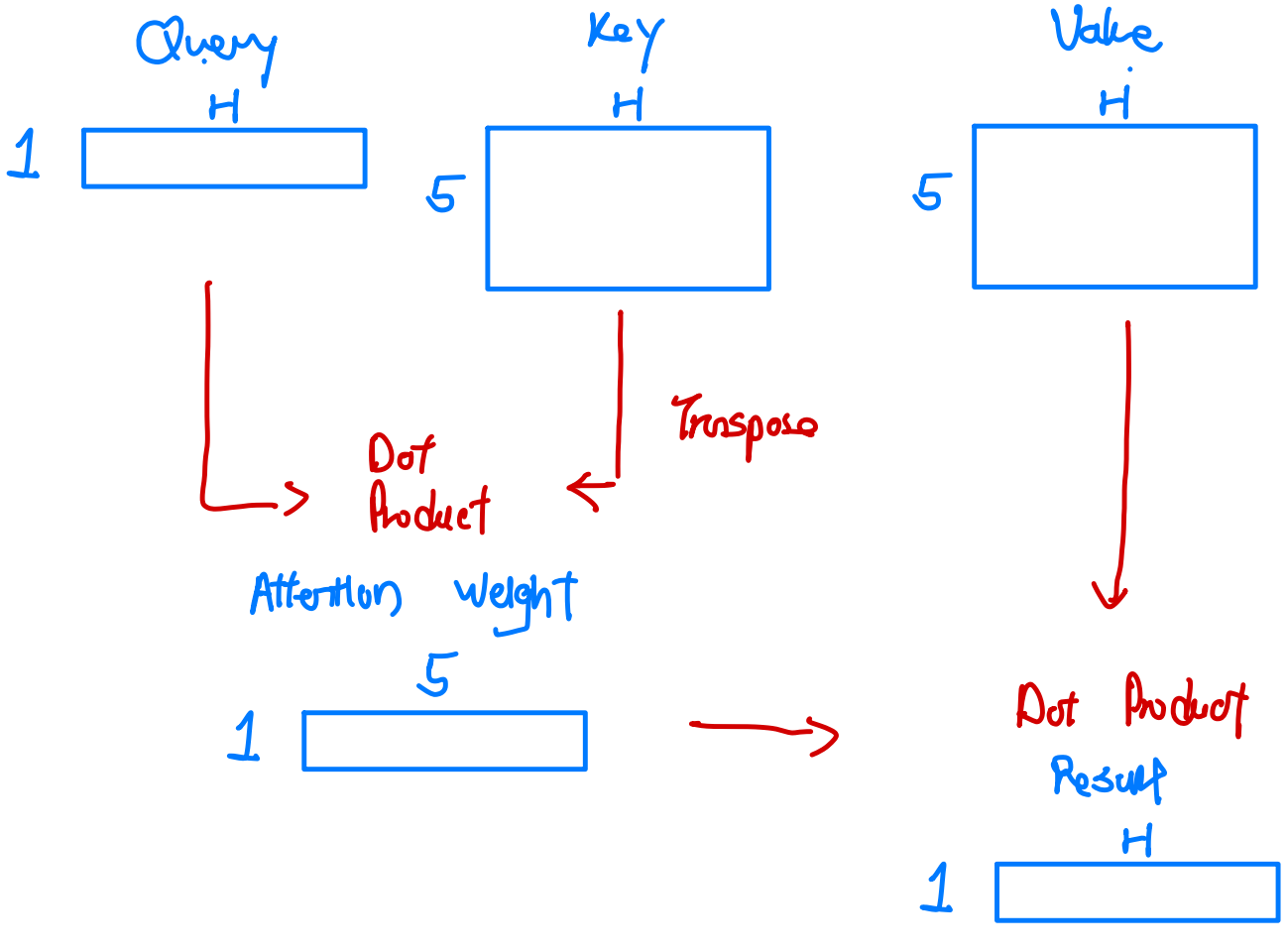
Attention - block to replace Neural Network

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 ...

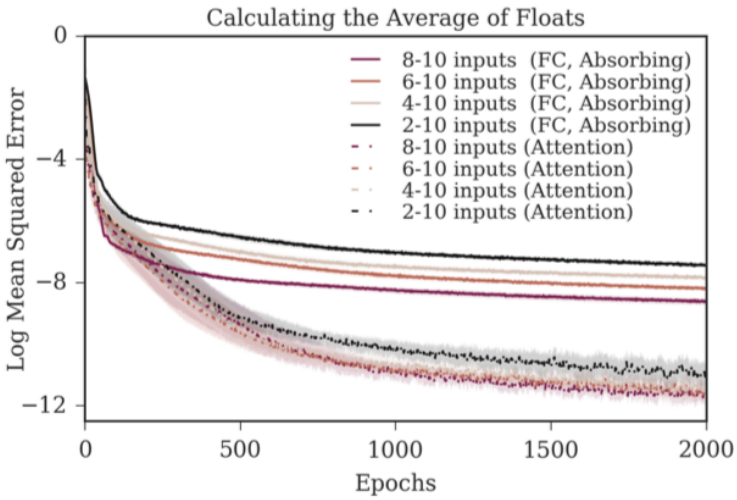
Query : 1 agent's state information : $1 \times H$

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



- As observed, as long as the state information can be encoded into 1 x H embedding, number of agent involved is not an issue when attention 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 x H, it can handle without absorving 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

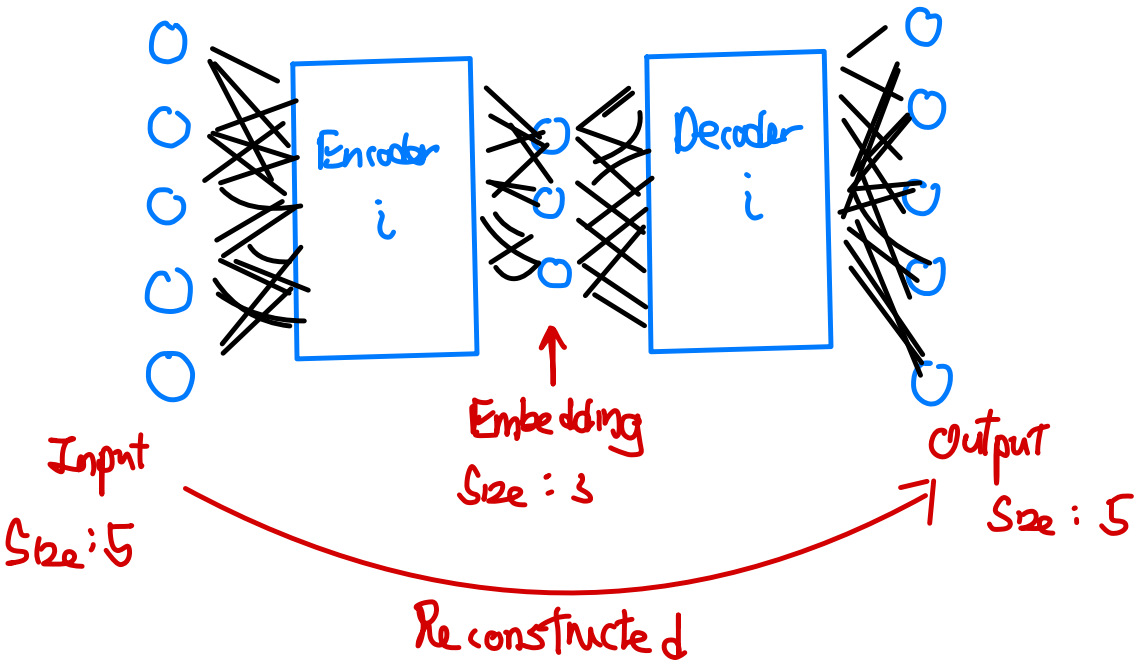
As observed above, H must still be uniform

Therefore, the paper made the following assumption

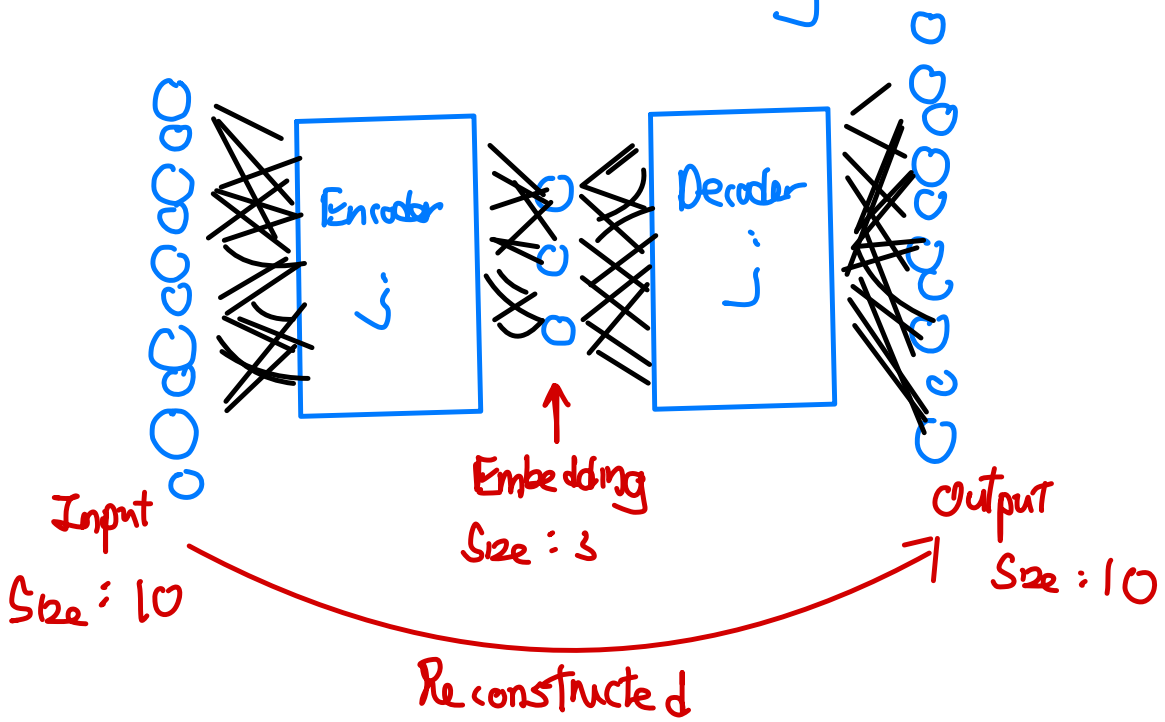
If agent i & agent j shares the same observation space, they will use the same encoder. Else, they will use different encoder

For instance,

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



different observation 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 calculation of advantage

c) Actor – A policy network

a) Critic – All agent share the same critic btw

The structure :

$$V_\phi\left(RSA\left(g_i\left(o_t^i\right)\right)\right)$$

V_ϕ: A MLP with ϕ weight

RSA: A residual Self Attention Block (Q, K, V = self)

g_i: An encoder to encode the state information into embedding dimensions

o_t^i: Observation information of agent i at timestep t

where $1 < i < k_t$

where k_t is the number of available agents at timestep t

Assuming o_t^i has a size of 1×5

$$1 \times 5 \xrightarrow{g_i} 1 \times 3 \xrightarrow{RSA} 1 \times 3 \xrightarrow{V_\phi} 1 \times 1$$

1×3: embedding dim

1×1: Value of this state / observation

How to train ?

- By minimizing $J(\phi)$

$$J(\phi) = \left[V_\phi(\text{RSA}(g_i(o_t^i))) - \gamma^{(\lambda)} \right]^2$$

$V_\phi(\text{RSA}(g_i(o_t^i)))$: Predicted value of this state / observation

$\gamma^{(\lambda)}$: Target value of this state / observation

where

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

lambda, refer to GAE. 0 = TD, 1 = MC

Although n can be up to ∞, but that will be MC, so user can define it to be lower

$G_t^{(n)}$: Discounted return starting from timestep, t with n-value bootstrapping

Discount factor

where

$$G_t^{(n)} = \sum_{l=1}^{n} \gamma^{l-1} r_{t+l} + \gamma^n V_\phi(\text{RSA}(g_i(o_{t+n}^i)))$$

$V_\phi(\text{RSA}(g_i(o_{t+n}^i)))$: The value of next state after bootstrap for n 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

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 $i \neq j$

A MLP with ψ weight

$$Q_\psi\left(\text{RSA}\left(g_j(o_t^j), f_i(o_t^i, a_t^i)\right)\right)$$

Encoder

Observation 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

$Q : g_j(o_t^j) : 1 \times H$ → Refer to agent j only

$V, K : f_i(o_t^i, a_t^i) : N \times H$
↳ Refer to all agents that are present at time t
↳ Except agent j
↳ $N = k_t - 1$

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

How to train ?

- By minimizing $J(\psi)$

$$J(\psi) = \Big(\underbrace{Q_\psi(\text{RSA}(g_j(o_t^j), f_i(o_t^i, a_t^i))}_{\text{Predicted baseline}} - \underbrace{y^{(2)}}_{\text{Target baseline}} \Big)^2$$

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

So what is the intuition of this baseline network ?

$$Q_\psi(\text{RSA}(\underbrace{g_j(o_t^j)}_{\text{Query}}, \underbrace{f_i(o_t^i, a_t^i)}_{\text{Key, Value}}))$$

(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

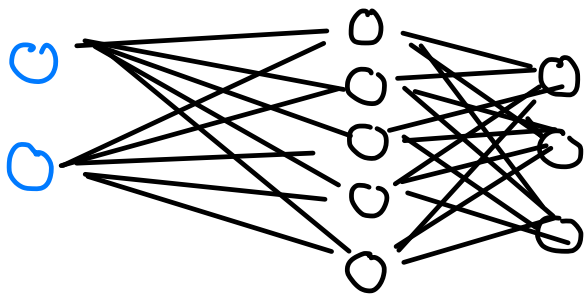
(2) → The existence of this baseline also allow Coordination to take place since it allows each agent to compare fairly regarding their correct value

→ 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 means 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?

$$\nabla_{\theta_i} J(\theta_i) = \mathop{E}_{\substack{s\sim\rho^\pi \\ a^i\sim\pi_i}} \left[\nabla_\theta \log \pi_i(a_i|o_i) \cdot \text{Advantage}_i \right]$$

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

$$\theta_{update} = \theta_{old} + \underbrace{\alpha}_{\text{Learning rate}} \underbrace{\sum_i^N \nabla_{\theta_i} J(\theta_i)}_{\substack{\text{Sum of} \\ \text{Gradient across} \\ \text{all agents}}}$$

$$\text{Advantage}_j = \underbrace{y^{(\lambda)}}_{\text{Discounted return}} - \underbrace{Q_\psi\left(\text{RSA}\left(g_j(o_t^j), f_i(o_t^i, a_t^i)\right)\right)}_{\text{Baseline for agent } j}$$

Chapter 18 :

Proximal Policy Optimization

What is the motivation behind PPO?

- It aims at fixing a few flaws in the most basic Policy Gradient Method (PGM) - REINFORCE

  → In REINFORCE, during the data collection phase, it will involving using an agent to sample 1 full trajectory.

  → Afterwards, the algorithm will enter the update phase where they compute the performance objective & perform backward propagation to update the policy network

Derivative of performance objectives for update of a policy network

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

$$= E\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) \cdot R(\tau^{(i)}) \right]$$

Note: E refers to expected value, which can also be treated as the average over multiple trajectory.

↳ The problem with such approach in REINFORCE is that each sample that was collected by the agent during data collection step can only to be used for 1 backward propagation & update.

→ Or in other words, 1 step of data collection can only collect data that can be used for 1 epoch of update.

→ This subsequently leads to issues of inefficient usage of sampled data for training.

→ The more ideal approach would be:

a) 1 step of data collection

b) The collected data is used for multiple step of update to ensure complete usage of the data for learning

↳ However, in the current frame work of REINFORCE, this would lead to too big of an update that could cause training instability.

Therefore, PPO is introduced to achieve the following objectives

a) Allow all data that was collected in the previous data collection step to be used for multiple step of update

b) While during each update, the update step are kept to be small to avoid huge update step

Components in PPO

a) An actor - a policy network

b) A critic - a value network that would be responsible to calculate advantage

Performance objective in PPO

$$L = L^{CLIP} - vL^{V} + hH$$

- L^{CLIP}: Clipped surrogate objective
- v: A constant for weightage
- L^{V}: Value Loss
- h: A constant for weightage
- H: Entropy bonus

L^{CLIP} - Clipped surrogate objective

$$L^{CLIP} = E_t\left[\min\left(p_t(\theta)A_t,\ clip\left(p_t(\theta), 1-\varepsilon, 1+\varepsilon\right)A_t\right)\right]$$

where:

① $$p_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta\text{-old}}(a_t|s_t)}$$

- $\pi_\theta(a_t|s_t)$ → Probability of taking an action given the current state following the latest policy
- $\pi_{\theta\text{-old}}(a_t|s_t)$ → Probability of taking an action given the current state following the policy that was used in data collection

Why are there 2 policies?

→ because after data collection step, the algorithm will enter the update step.

→ The update step consists of multiple epochs.

→ One epoch means that all the data that was collected in the last data collection step had been backward propagation.

→ During one epoch, it will be further divided into multiple step of backward propagation where each backward propagation indicates that a mini-batch of data has been sampled.

→ After x number of mini-batches had been sampled to go through the entire dataset, it indicates that 1 epoch had been completed.

→ Therefore, as soon as the first mini batch is backward-propagated, the policy network will be updated and become different from the policy network that was used to collect the sample data.

Therefore:

π_θ → the latest policy network

$\pi_{\theta\text{-old}}$ → the policy network that was used in the last data collection step

$$p_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta\text{-old}}(a_t|s_t)}$$

In short, this term represents that how likely that the latest policy network taking the same action given the current state compared to the policy network used in data collection

If $p_t(\theta) > 1$, it means more likely / probable.

If $p_t(\theta) < 1$, it means less likely / probable.

② $$A_t = V_t^{target} - V_{w\text{-old}}(s_t)$$

where

A_t : Advantage

$$V_t^{target} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \dots \gamma^{n-1} r_{t+n-1} + \gamma^n V_{w\text{-old}}(s_{t+n})$$

- As observed, the advantage is calculated using n-value bootstrapping.
- If s_{t+n} is the terminal state, $V_{w\text{-old}}(s_{t+n})$ will be set to 0 because there is no longer expected return beyond termination

- Similarly before, V_{w_old} refers to the value network that was involved in data collection steps. As soon as the first step of back propagation is completed, $V_w \neq V_{w\_old}$

(3) With the components understood, let's pivot back to the full picture of L^{CLIP}

$$L^{CLIP} = E_t\left[\min\left(\rho_t(\theta) A_t,\ clip\left(\rho_t(\theta), 1-\varepsilon, 1+\varepsilon\right) A_t\right)\right]$$

- As observed above, if we omit E_t for now,

$$L^{CLIP} = \rho_t(\theta) A_t \quad \text{or} \quad L^{CLIP} = clip\left(\rho_t(\theta), 1-\varepsilon, 1+\varepsilon\right) \cdot A_t$$

- depends on whichever is smaller.

- Usually, ε is a value between 0 to 1. By default, it is set to 0.2.

- Because of this, a total of 6 possible scenarios can happen

1) When A_t is positive

1.1) $1-\varepsilon \leq \rho_t(\theta) \leq 1+\varepsilon$

$$L^{CLIP} = \rho_t(\theta) A_t$$

→ updated as usual

1.2) $\rho_t(\theta) > 1+\varepsilon$

$$L^{CLIP} = (1+\varepsilon) A_t$$

→ This L^{CLIP} has no θ, the parameter of policy network π_θ

→ When differentiated against θ, this term gave 0

→ Essentially, means it does not influence the update of π_θ

1.3) $\rho_t(\theta) < 1-\varepsilon$

$$L^{CLIP} = \rho_t(\theta) \cdot A_t$$

→ updated as usual

2) When A_t is negative

2.1) $1-\varepsilon \leq \rho_t(\theta) \leq 1+\varepsilon$

$$L^{CLIP} = \rho_t(\theta) \cdot A_t$$

→ updated as usual

2.2) $\rho_t(\theta) > 1+\varepsilon$

$$L^{CLIP} = \rho_t(\theta) \cdot A_t$$

→ updated as usual

2.3) $\rho_t(\theta) < 1-\varepsilon$

$$L^{CLIP} = (1-\varepsilon) \cdot A_t$$

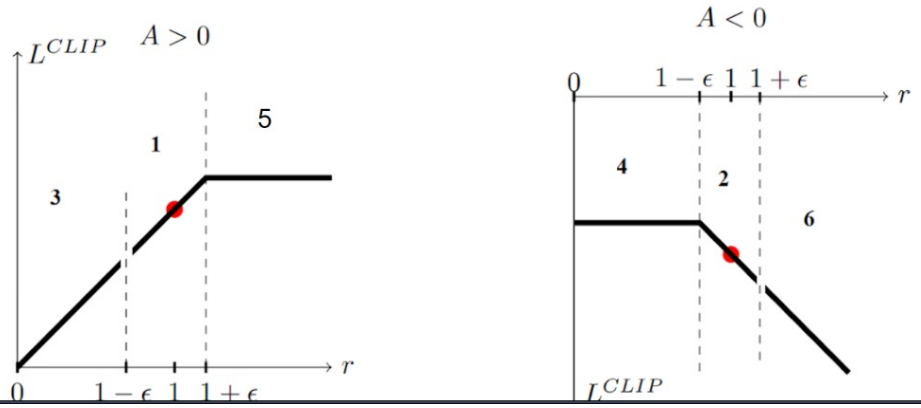
→ This L^{CLIP} has no θ, the parameter of policy network π_θ

→ when differentiated against θ, this term gave 0

→ Essentially, means it does not influence the update of π_θ

- As a summary:

	$p_t(\theta)\ > 0$	A_t	Return Value of *min*	Objective is Clipped	Sign of Objective	Gradient
1	$p_t(\theta) \in [1-\epsilon, 1+\epsilon]$	+	$p_t(\theta)A_t$	no	+	✓
2	$p_t(\theta) \in [1-\epsilon, 1+\epsilon]$	−	$p_t(\theta)A_t$	no	−	✓
3	$p_t(\theta) < 1-\epsilon$	+	$p_t(\theta)A_t$	no	+	✓
4	$p_t(\theta) < 1-\epsilon$	−	$(1-\epsilon)A_t$	yes	−	**0**
5	$p_t(\theta) > 1+\epsilon$	+	$(1+\epsilon)A_t$	yes	+	**0**
6	$p_t(\theta) > 1+\epsilon$	−	$p_t(\theta)A_t$	no	−	✓



By referring to the diagram above, the following intuition can be developed

Region 1 & 3

1) When Advantage is positive, L^{CLIP} encourages the policy network to be updated towards it where as long as $p_t(\theta) \leq 1+\epsilon$, therefore

$$L^{CLIP} = p_t(\theta) \cdot A_t$$

→ where there's a positive derivative that drives the policy's update towards it

2) However when Advantage is positive & $p_t(\theta) > 1+\epsilon$, intuitively, this means that the probability of the

Regn 5 new policy network is way too likely in doing that action, therefore it stopped this sample from influencing the update of the network.

$$L^{CLIP} = (1+\varepsilon) \cdot A_t$$

→ where there's a zero derivative with respect to the parameter of policy network, π_θ

Regn 2 & 6 3) When Advantage is negative, L^{CLIP} encourages the policy network to be updated away from it. Where as long as $p_t(\theta) \geq 1-\varepsilon$, therefore

$$L^{CLIP} = p_t(\theta) \cdot A_t$$

→ where there's a negative derivative that does the policy's update towards it

Regn 4 4) However when Advantage is negative, & $p_t(\theta) < 1-\varepsilon$, intuitively, this means that the probability of the new policy network is way less likely in doing that action, therefore it stopped this sample from influencing the update of the network.

$$L^{CLIP} = (1-\varepsilon) \cdot A_t$$

→ where the's a zero derivate with respect to the parameter of policy network, π_θ

Intuitively, this L^{CLIP} achieves marginal update, therefore allowing the data collected from the same data collection step to be reused for multiple epoch.

Finally, Let's discuss the expected value ...

$$L^{CLIP} = E_t\left[\min\left(\rho_t(\theta)A_t,\ clip\left(\rho_t(\theta), 1-\varepsilon, 1+\varepsilon\right)A_t\right)\right]$$

$$\approx \frac{1}{H}\sum_{t=1}^{H}\left[\min\left(\rho_t(\theta)A_t,\ clip\left(\rho_t(\theta), 1-\varepsilon, 1+\varepsilon\right)A_t\right)\right]$$

The expected value here refer to the average over the minibatch where each minibatch contains of H samples

Each sample is tied together with a timestep & an action as following

$$\left(S_t,\ a_t,\ \pi_{\theta\text{-old}}(a_t|S_t),\ S_{t+1},\ r_t,\ V_t^{target},\ A_t\right)$$

With that said, let's move onto the second component in the performance objective

L^V - Squared value loss

$$L^V = E_t\left[\left(V_w(S_t) - V_t^{target}\right)^2\right]$$

$$= \frac{1}{H}\sum_{t=1}^{H}\left[\left(V_w(S_t) - V_t^{target}\right)^2\right]$$

- As discussed before,

$$V_t^{target} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots \gamma^{n-1} r_{t+n-1} + \gamma^n V_{w,old}(S_{t+n})$$

V_w is the latest value network

- Very simply, that this function is try to improve the state value prediction made by the value network

- Also, same as before, the expected value here refers to the average over a mini-batch which has H samples.

As for the final components ... —

H - Entropy Bonus

$$H = E_t\left[-\sum_{m=1}^{M} \phi_{t,m} \log \phi_{t,m} \right]$$

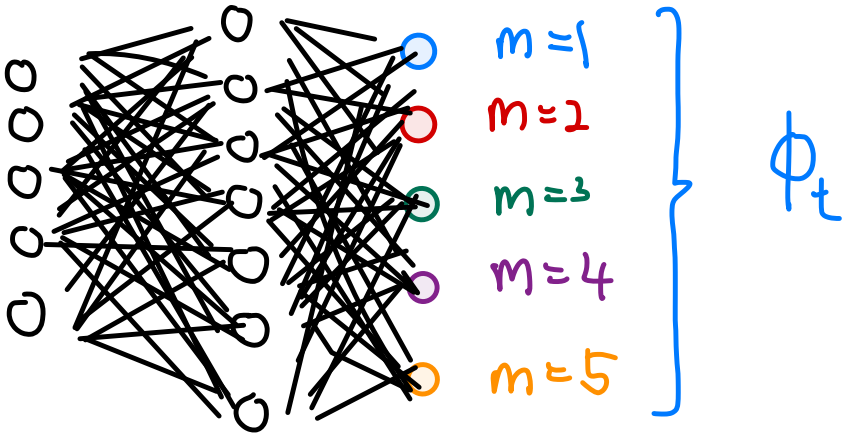
$$= \frac{1}{H}\sum_{t=1}^{H}\left[-\sum_{m=1}^{M} \phi_{t,m} \log \phi_{t,m} \right]$$

where

ϕ_t is a vector of normalized probability estimates (associated with a single training example taken from a minibatch)

$\phi_{t,m}$ refers to the m^{th} elements

Essentially, this entropy bonus is only suitable to be applied in discrete actions. because the policy network in discrete actes space is as such:



↳ Each representing the probability of taking an action

It is recommended to study Statistics chapter 2: Entropy for a better grasp of intuition here. But essentially

- $H = -\sum_{m=1}^{M} \phi_{t,m} \cdot \log \phi_{t,m}$ is basically an entropy function
- where the more balanced of the spread of probability among each $\phi_{t,m}$, the higher the result of entropy. If
- which means that
  a) when entropy is high
  b) $\phi_{t,m}$ of each m elements are similar
  c) Probability of taking any of the actions are roughly similar
  d) Therefore, it encourages Exploration
- Basically, this terms gave a slight boost to the performance objective for high entropy, which leads to more exploration.

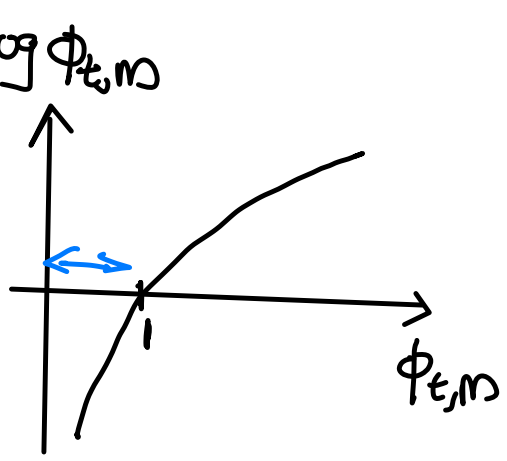
- Note:

$$H = E_t\left[-\sum_{m=1}^{M} \phi_{t,m} \log \phi_{t,m} \right]$$

Although here is a negate sign

This term will also always be negate

Resulting in a posne term



$0 \leq \phi_{t,m} \leq 1$, therefore, $\log \phi_{t,m} < 0$

- Note:
  - The above method is only suitable for discrete action space.
  - In continuous actin space, the policy network would only predict the mean for a Gaussian Distribution that will be used for sampling an action. Meanwhile, the standard deviation for this distribution would be a fixed hyperparameter.
  - In continuy actin space, to encourage the exploration, it may require the policy network to also predict the standard deviation, and from there to encourage the network to explore

- Regardless, that is beyond the discussion scope here.

Therefore, overall, the performance objective is as such:

If Gradient Ascend

$$L = L^{CLIP} - vL^{V} + hH$$

If Gradient Descend

$$L = -L^{CLIP} + vL^{V} - hH$$

Note:

- Both policy network π_θ & value network V_w shares the same performance objectives
- But for instance, L^V has no impact on π_θ as it does not contains any θ parameter during its computation
- L^{CLIP} has impact on both network's update
- H only has impact on π_θ
- Regardless, in PyTorch implementation, it has only 1

Loss / Performance objective for both network

- The backprop will be handled by 'loss.backward'

Full flow of PPO algorithm

https://fse.studenttheses.ub.rug.nl/25709/1/mAI_2021_BickD.pdf

Algorithm 1 Proximal Policy Optimization (PPO) using Stochastic Gradient Descent (SGD)

Input: N = Number of parallel agents collecting training data, T = Maximal trajectory length, performance criterion or maximal number of training iterations, weighting factors v and h

$\pi_\theta \leftarrow newPolicyNet()$
$V_\omega \leftarrow newStateValueNetwork()$ ▷ Possibly parameter sharing with π_θ
env $\leftarrow newEnvironment()$
optimizer $\leftarrow newOptimizer(\pi_\theta, V_\omega)$
number_minibatches $= \left\lfloor \frac{N*T}{minibatch\_size} \right\rfloor$ ▷ Compute number of minibatches per epoch
while performance criterion not reached **or** maximal number of iterations not reached **do**
  train_data $\leftarrow$ []

(1)

  // Training data collection step. Ideally to be parallelized:
  for actor = 1, 2, ..., N **do**
    train_data $\leftarrow$ []
    $s_{t=1} \leftarrow$ env.$randomlyInitialize()$ ▷ Reset environment to a random initial state
    // Let agent interact with its environment
    // for T time steps & collect training data:
    for t = 1, 2, ..., T **do**
      $a_t \leftarrow \pi_\theta.generate\_action(s_t)$
      $\pi_{\theta_{old}}(a_t|s_t) \leftarrow \pi_\theta.distribution.get\_probability(a_t)$
      $s_{t+1}, r_t \leftarrow env.step(a_t)$ ▷ Advance simulation one time step
      train_data $\leftarrow$ train_data + $tuple(s_t, a_t, r_t, \pi_{\theta_{old}}(a_t|s_t))$
    // Use training data to augment each collected tuple
    // of training data stored in train_data:
    for t = 1, 2, ..., T **do**
      $V_t^{target} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V_\omega(s_T)$
      $A_t = V_t^{target} - V_\omega(s_t)$
      train_data[t] $\leftarrow$ train_data[t] + $tuple(A_t, V_t^{target})$

  $optimizer.resetGradients(\pi_\theta, V_\omega)$
  // Update trainable parameters θ and ω for K epochs:

(2)

  for epoch = 1, 2, ..., K **do**
    train_data $\leftarrow randomizeOrder$(train_data)
    for mini_idx = 1, 2, ..., number_minibatches **do**
      M $\leftarrow getNextMinibatchWithoutReplacement$(train_data, mini_idx)
      for example e $\in$ M **do**
        $s_t, a_t, r_t, \pi_{\theta_{old}}(a_t|s_t), A_t, V_t^{target} \leftarrow unpack(e)$
        $\_ \leftarrow \pi_\theta.generate\_action(s_t)$ ▷ Parameterize policy's probabilit distribution
        $\pi_\theta(a_t|s_t) \leftarrow \pi_\theta.distribution.get\_probability(s_t)$
        $p_t(\theta) \leftarrow \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$
        $\phi_t \leftarrow \pi_{\theta_{stoch}}.get\_parameterization()$ ▷ To be computed in case of discrete action space
      $L^{CLIP} = \frac{1}{|M|} \sum_{t \in \{1,2,...,|M|\}} min(p_t(\theta)A_t, clip(p_t(\theta), 1-\epsilon, 1+\epsilon)A_t)$
      $L^V = \frac{1}{|M|} \sum_{t \in \{1,2,...,|M|\}} (V_\omega(s_t) - V_t^{target})^2$
      $H = -\frac{1}{|M|} \sum_{t \in \{1,2,...,|M|\}} \phi_t \; log \; \phi_t$ ▷ To be computed in case of discrete action space
      $L^{CLIP+V+H} \leftarrow -L^{CLIP} + v * L^V - h * H$
      optimizer.$backpropagate(\pi_\theta, V_\omega, L^{CLIP+V+H})$
      optimizer.$updateTrainableParameters(\pi_\theta, V_\omega)$
return π_θ

① Data collection step

a) Collecting

- S_t, a_t, r_t, $\pi_{\theta_{old}}(a_t|s_t)$, S_{t+1}

Computing

- V_t^{target} & A_t

1 sample : $(S_t, a_t, r_t, S_{t+1}, \pi_{\theta_{old}}(a_t|s_t), V_t^{target}, A_t)$

b). Expects that each agent collect up to T number of samples

- If terminated before reaching T samples, the agent will be reinitialized with a reset environment to start over with a new trajectories

c) If can be achieved with

i) 1 environment with K number of agents
- when 1 agent reaches termination, all agent starts with new trajectories

ii) K environment with 1 agent each
- 1 agent's termination does not affect the trajectories of other agents.

In short, each data collection expects a dataset of

$K \times T$ samples

K: number of agents

T: number of steps taken by each agents

(2) Update step

- multiple epochs with the same collected dataset
- J_{θ} & V_w shares the same objectives

Post notes :

- Similarity between L^{clip} & the performance objective in Reinforce

In REINFORCE

$$J(\pi_\theta) = \log \pi_\theta(a_t|s_t) \cdot r_t$$

$$\nabla_\theta J(\pi_\theta) = \frac{1}{\pi_\theta(a_t|s_t)} \left[\nabla_\theta \log \pi_\theta(a_t|s_t)\right] r_t$$

In PPO :

$$L^{clip} = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta\text{-old}}(a_t|s_t)} A_t$$

$$\nabla_\theta L^{clip} = \frac{1}{\pi_{\theta\text{ old}}(a_t|s_t)} \left[\nabla_\theta \pi_\theta(a_t|s_t) A_t\right]$$

The following section discusses some of the implementation details

(1) Calculation of GAE's advantage & return

```
        # bootstrap value if not done
        with torch.no_grad():
            next_value = agent.get_value(next_obs).reshape(1, -1)
            if args.gae:
                advantages = torch.zeros_like(rewards).to(device)
                lastgaelam = 0
                for t in reversed(range(args.num_steps)):
                    if t == args.num_steps - 1:
                        nextnonterminal = 1.0 - next_done
                        nextvalues = next_value
                    else:
                        nextnonterminal = 1.0 - dones[t + 1]
                        nextvalues = values[t + 1]
                    delta = rewards[t] + args.gamma * nextvalues * nextnonterminal - values[t]
                    advantages[t] = lastgaelam = delta + args.gamma * args.gae_lambda * nextnonterminal * lastgaelam
                returns = advantages + values
            else:
                returns = torch.zeros_like(rewards).to(device)
                for t in reversed(range(args.num_steps)):
                    if t == args.num_steps - 1:
                        nextnonterminal = 1.0 - next_done
                        next_return = next_value
                    else:
                        nextnonterminal = 1.0 - dones[t + 1]
                        next_return = returns[t + 1]
                    returns[t] = rewards[t] + args.gamma * nextnonterminal * next_return
                advantages = returns - values
```

Maths :

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

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

For ease of implementation, the advantage is calculated from the final time step, where $t = T$ which can be showcased as follow

$$A_T^{GAE} = (\gamma\lambda)^0 \delta_T$$

$$= \delta_T$$

$$= r_T - V(s_T)$$

$$A_{T-1}^{GAE} = \sum_{l=0}^{1} (\gamma\lambda)^l \delta_{T-1+l}$$

$$= \delta_{T-1} + (\gamma\lambda)(\delta_T)$$

$$= \delta_{T-1} + (\gamma\lambda)(A_T^{GAE})$$

$$A_{T-2}^{GAE} = \sum_{l=0}^{2} (\gamma\lambda)^l \delta_{T-2+l}$$

$$= \delta_{T-2} + (\gamma\lambda)\delta_{T-1} + (\gamma\lambda)^2 \delta_T$$

$$= \delta_{T-2} + (\gamma\lambda)(\delta_{T-1} + (\gamma\lambda)(\delta_T))$$

$$= \delta_{T-2} + (\gamma\lambda)(A_{T-1}^{GAE})$$

To generalise

$$A_N^{GAE} = \delta_N + \gamma\lambda\left(A_{N+1}^{GAE}\right)$$
$$\delta_N^{GAE} = r_N + \gamma V(S_{N+1}) - V(S_N)$$

} if $N \neq T$

$$A_N^{GAE} = \delta_N$$
$$\delta_N^{GAE} = r_N - V(S_N)$$

} else $N == T$

where T is the final step of a trajectory

To handle the potential variation caused by termination, the final generalization is made as following:

$$A_N^{GAE} = \delta_N + \gamma\lambda \cdot F_{NT} \cdot \left(A_{N+1}^{GAE}\right)$$
$$\delta_N^{GAE} = r_N + \gamma \cdot F_{NT} \cdot V(S_{N+1}) - V(S_N)$$

where

F_{NT} is non termination flag, where

$F_{NT} = 1$ if N is not the final step of a trajectory else $= 0$

The benefits of this generalization is that in implementation

→ The agent will store the data collected in an array continuously ACROSS trajectories

→ But this implementation design will enable the advantage accumulation calculation to be automatically reset once it detects it is the final step of a trajectory (or can be understood that it detects that a new trajectory is detected since the calculation is done from end to first).

Once the advantages are calculated, the expected return of this step can also be calculated as such:

$$\underbrace{\text{returns}[t]}_{\text{will be feed into the squared value loss calculation later on}} = \text{Advantages}[t] + \text{Value}[t]$$

$$L^V = E_t\left[\left(V_w(S_t) - V_t^{\text{target}}\right)^2\right]$$

$$= \frac{1}{H}\sum_{t=1}^{H}\left[\left(V_w(S_t) - V_t^{\text{target}}\right)^2\right]$$

where it was initially defined as :

$$V_t^{target} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \dots \gamma^{n-1} r_{t+n-1} + \gamma^n V_{w,old}(S_{t+n})$$

(2) Value Loss clipping

The definition :

$$L^V = \max\left[\left(V_w(S_t) - V_t^{target}\right)^2, \left(clip\left(V_w(S_t), V_w(S_{t-1}) - \varepsilon, V_w(S_{t-1}) + \varepsilon\right) - V_t^{target} \right)^2 \right]$$

So, what is the intuition here ?

- Essentially, the second term in max() aims to keep the current estimated state value by the value network to be within the range of ε from the value of the state at previous time step ... ...

- Then out of these 2 terms, the one that return higher value will be selected as the loss

- Now, this part of selecting the higher loss is because
  → A higher loss is a result from a high deviation between the estimated state value &

target state value

→ The high deviation is a result of the restricted current estimated value (which may or may not be affected by clipping)

- Essentially, the more restricted the current estimated value, the higher the resultant loss, and therefore it is selected to ensure a more stable shift of estimation made by the value network.

In the actual implementation:

$$\text{clip}(V_w(S_t),\ V_w(S_{t-1}) - \varepsilon,\ V_w(S_{t-1}) + \varepsilon)$$

$$= \text{clip}(V_w(S_t) + (V_w(S_{t-1}) - V_w(S_{t-1})),$$
$$\cancel{V_w(S_{t-1})} - \varepsilon + (V_w(S_{t-1}) - \cancel{V_w(S_{t-1})}),$$
$$\cancel{V_w(S_{t-1})} + \varepsilon + (V_w(S_{t-1}) - \cancel{V_w(S_{t-1})}))$$

$$= \text{clip}(V_w(S_t) + (V_w(S_{t-1}) - V_w(S_{t-1})),$$
$$V_w(S_{t-1}) - \varepsilon,$$
$$V_w(S_{t-1}) + \varepsilon)$$

$$= V_w(S_{t-1}) + \text{clip}\left(V_w(S_t) - V_w(S_{t-1}), -\varepsilon, \varepsilon\right)$$

As observed, this term more intuitively showcase that it is trying to limit the shift of estimation from V_w at S_{t-1} to S_t within a range of $-\varepsilon \le x \le \varepsilon$

$$L^V = \max\left[\left(V_w(S_t) - V_t^{target}\right)^2, \left(V_w(S_{t-1}) + \text{clip}\left(V_w(S_t) - V_w(S_{t-1}), -\varepsilon, \varepsilon\right) - V_t^{target}\right)^2\right]$$

```python
# Value loss
newvalue = newvalue.view(-1)
if args.clip_vloss:
    v_loss_unclipped = (newvalue - b_returns[mb_inds]) ** 2
    v_clipped = b_values[mb_inds] + torch.clamp(
        newvalue - b_values[mb_inds],
        -args.clip_coef,
        args.clip_coef,
    )
    v_loss_clipped = (v_clipped - b_returns[mb_inds]) ** 2
    v_loss_max = torch.max(v_loss_unclipped, v_loss_clipped)
    v_loss = 0.5 * v_loss_max.mean()
else:
    v_loss = 0.5 * ((newvalue - b_returns[mb_inds]) ** 2).mean()
```

③ Debug variable / metrics

a) KL divergence loss

- Measure the deviation of $\pi_{\theta_{old}}(a_t|s_t)$ from $\pi_\theta(a_t|s_t)$
- Refer to Statistics : KL divergence for more details

b) Clip fraction

- Calculate fraction of instances where the clipping in Clipped Surrogate Objective, L^{CLIP} happens

```
with torch.no_grad():
    # calculate approx_kl http://joschu.net/blog/kl-approx.html
    old_approx_kl = (-logratio).mean()
    approx_kl = ((ratio - 1) - logratio).mean()
    clipfracs += [((ratio - 1.0).abs() > args.clip_coef).float().mean().item()]
```

where

$$\text{log ratio} = \ln\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta\text{-old}}(a_t|s_t)}\right)$$

c) Explained variance

$$\text{Explained variance} = 1 - \frac{\text{Var}(y - \hat{y})}{\text{Var}(y)}$$

In this case,

$\hat{y}$: Value estimated by the value network

y : Returns calculated using the actual reward

$\frac{\text{Var}(y - \hat{y})}{\text{Var}(y)}$: Unexplained variance, i.e. how different is the prediction & target

```
y_pred, y_true = b_values.cpu().numpy(), b_returns.cpu().numpy()
var_y = np.var(y_true)
explained_var = np.nan if var_y == 0 else 1 - np.var(y_true - y_pred) / var_y
```

④ Annealing of learning rate

```
if args.anneal_lr:
    frac = 1.0 - (update - 1.0) / num_updates
    lrnow = frac * args.learning_rate
    optimizer.param_groups[0]["lr"] = lrnow
```

$$\text{fraction} = 1 - \frac{\text{Current update count} - 1}{\text{Total number of update}}$$

$$\text{lrnow} = \text{fraction} \times \text{lr}$$

If total number of update is 100

When current update count = 1

Fraction = 1

When current update count = 100

fraction = 0.01

⑤ Early Stopping

- use KL divergence computed above as the deciding metrics
- If its above a certain threshold, stop the training to avoid large update.