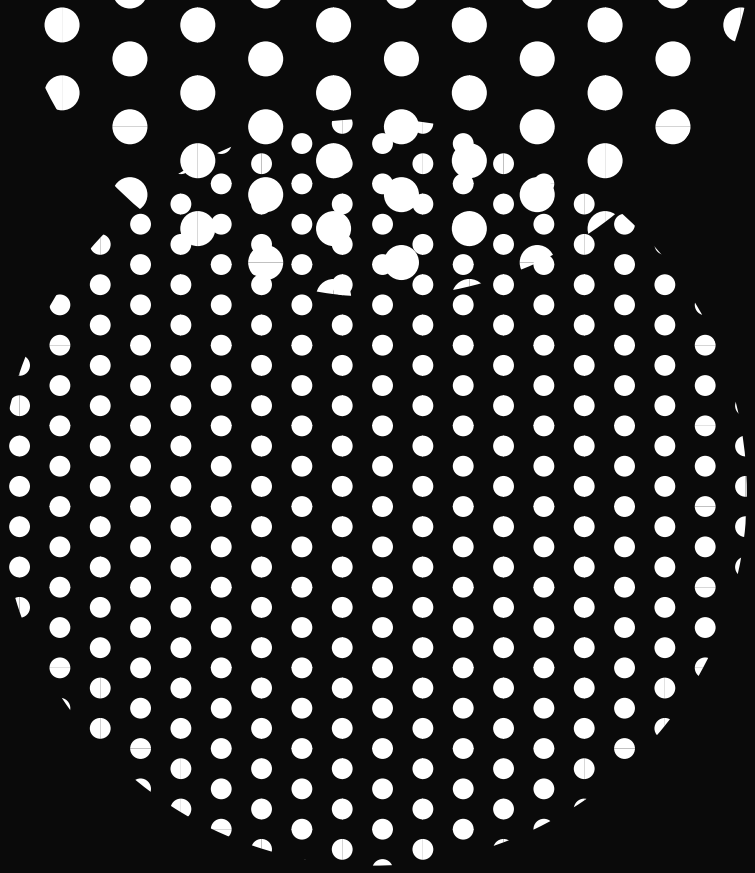
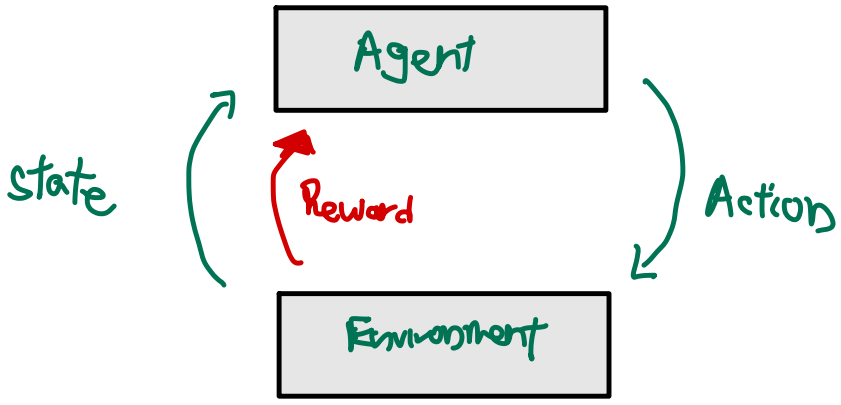


Chapter 1 :

Terminology in Markov Decision Process

Agent : Something that can be directly controlled

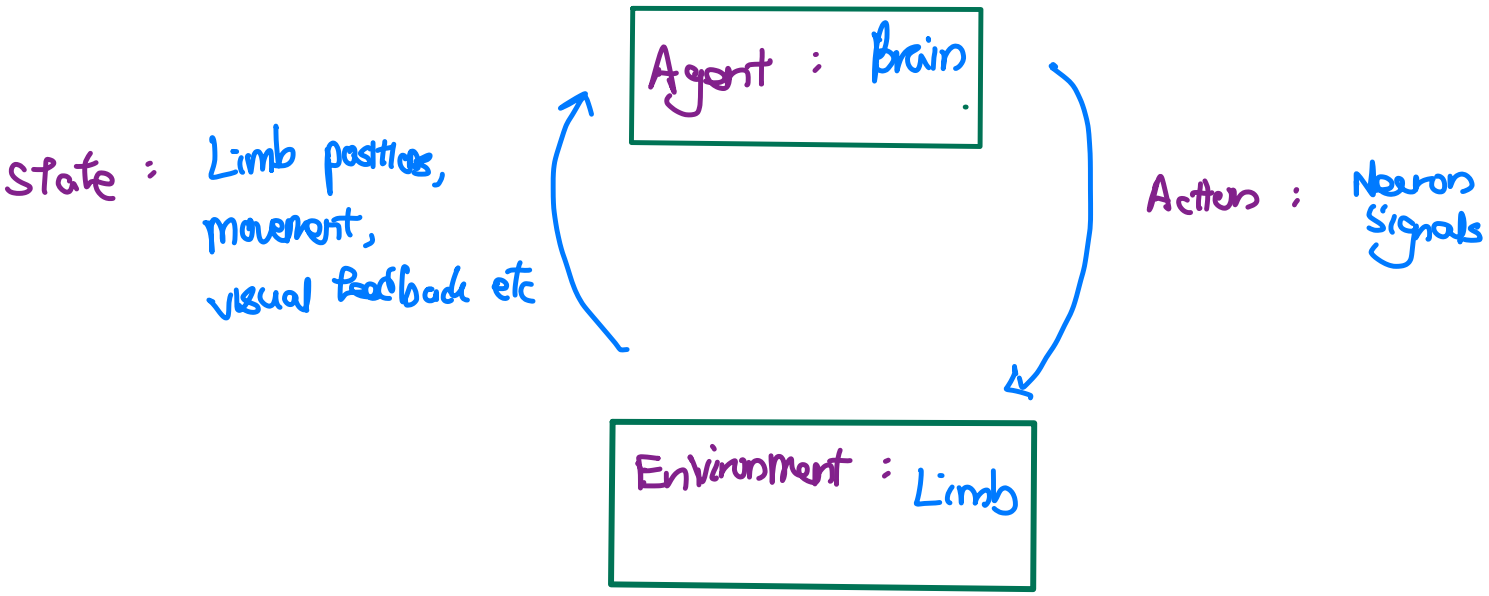
Environment : Something that cannot be controlled,
but can be interacted with through agent



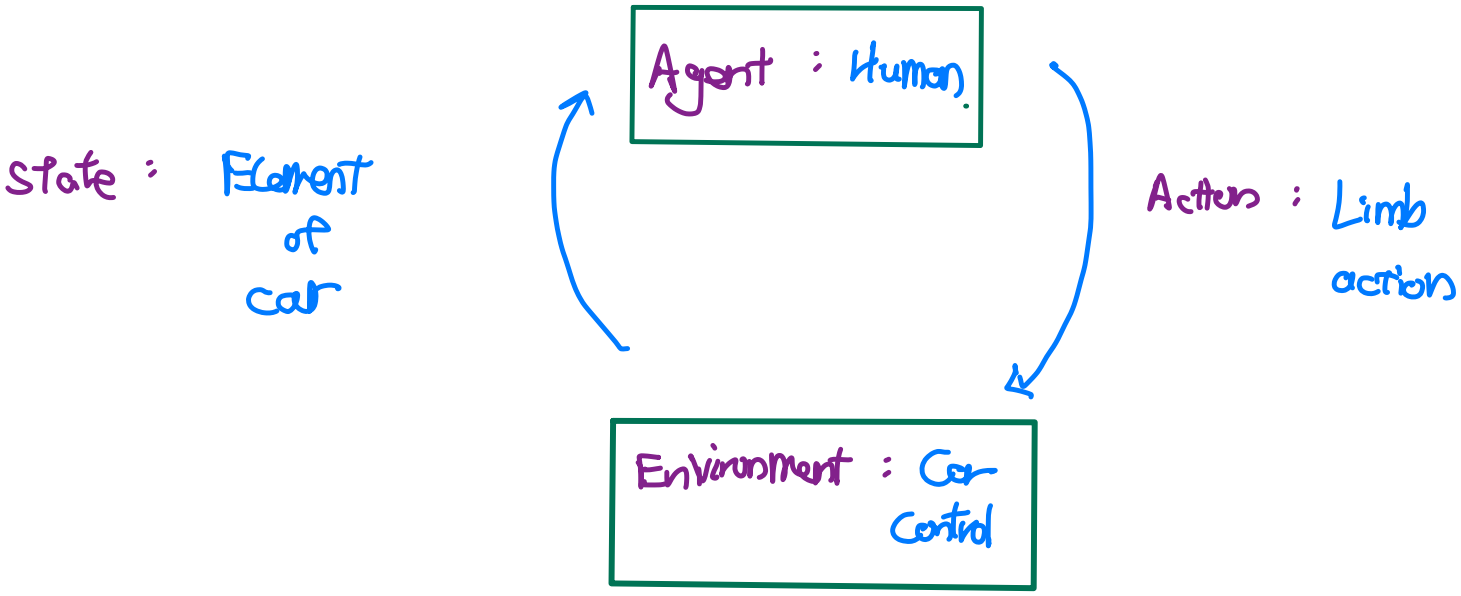
How to determine agent, environment, action & state can be quite arbitary

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

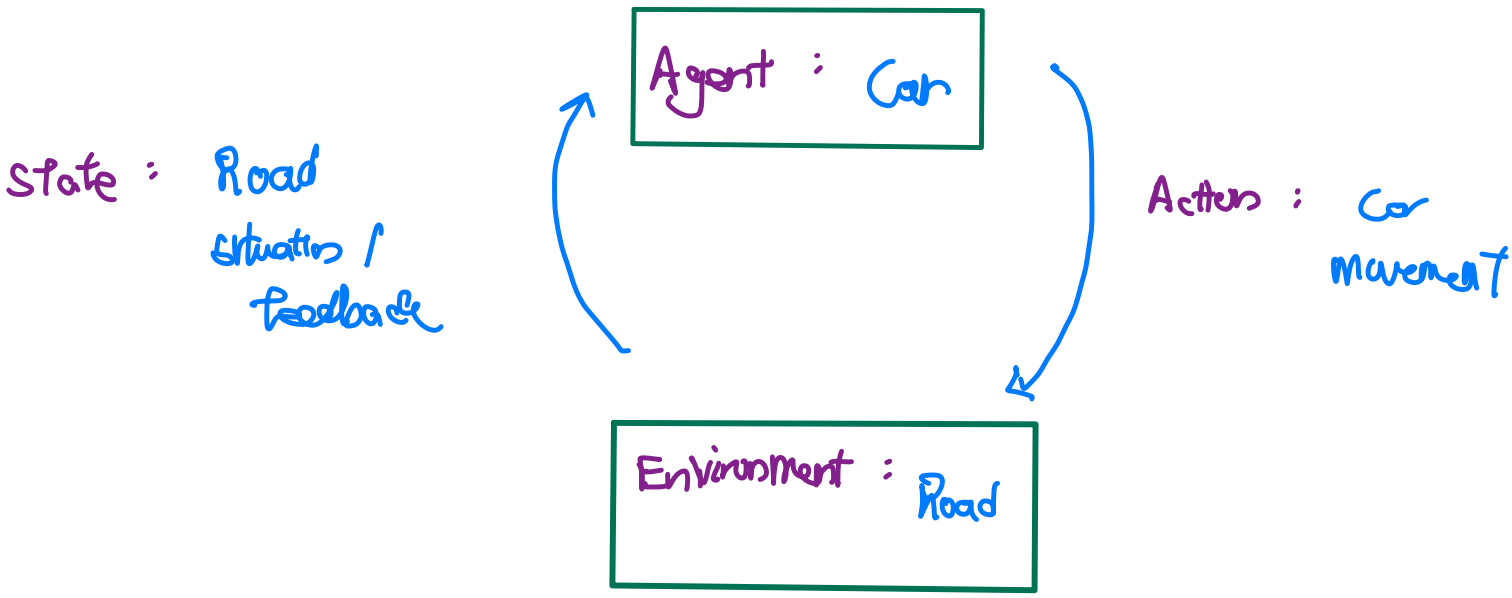
Program A : Control limb movement



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

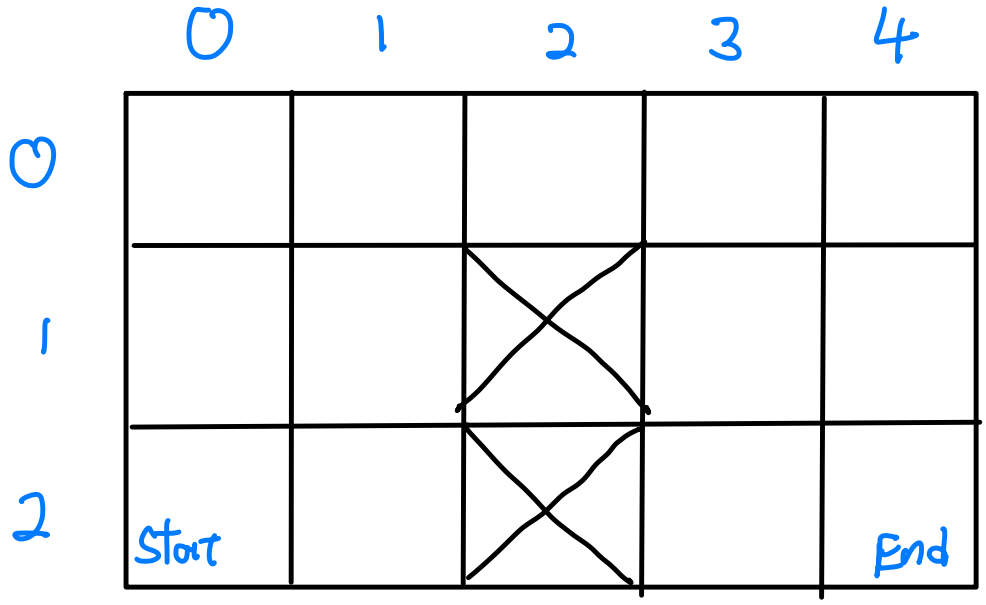
Chapter 2 :

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:

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

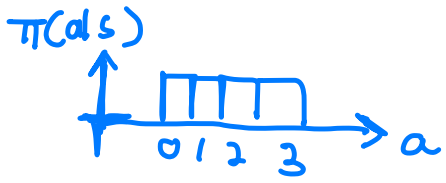
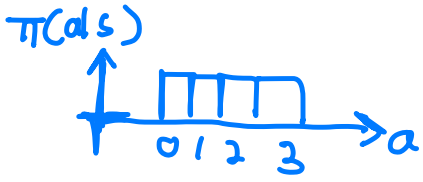
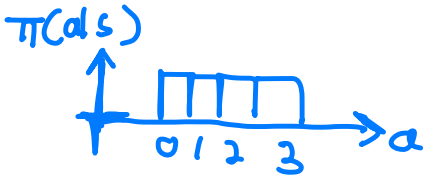
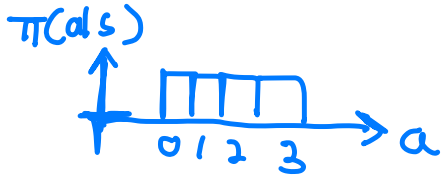
→ the subsequent state & reward given the previous state & action

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

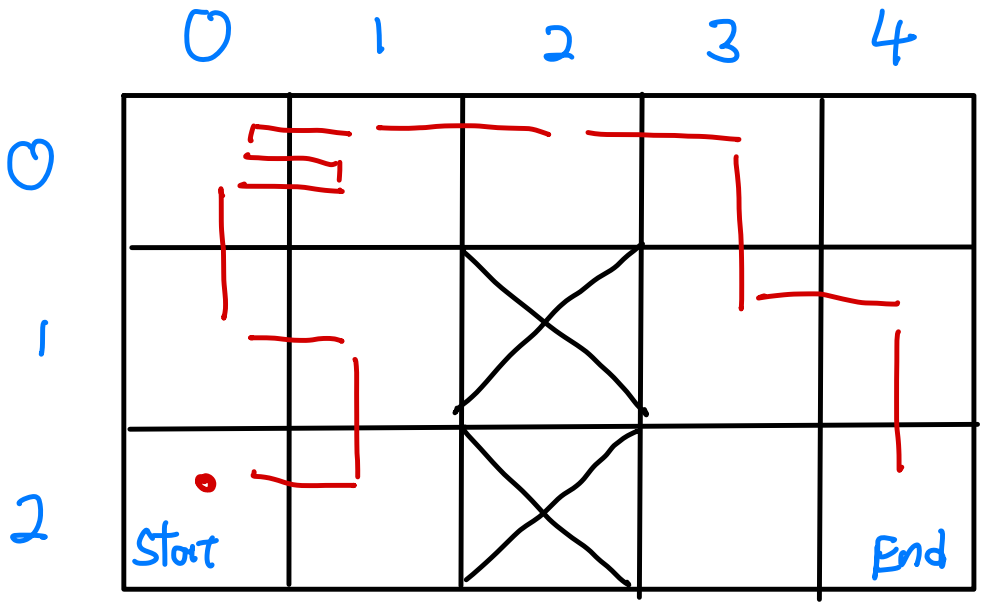
State	Policy, $\pi(a \mid s)$
0, 0	
0, 1	
0, 2	
0, 3 ⋮	

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:

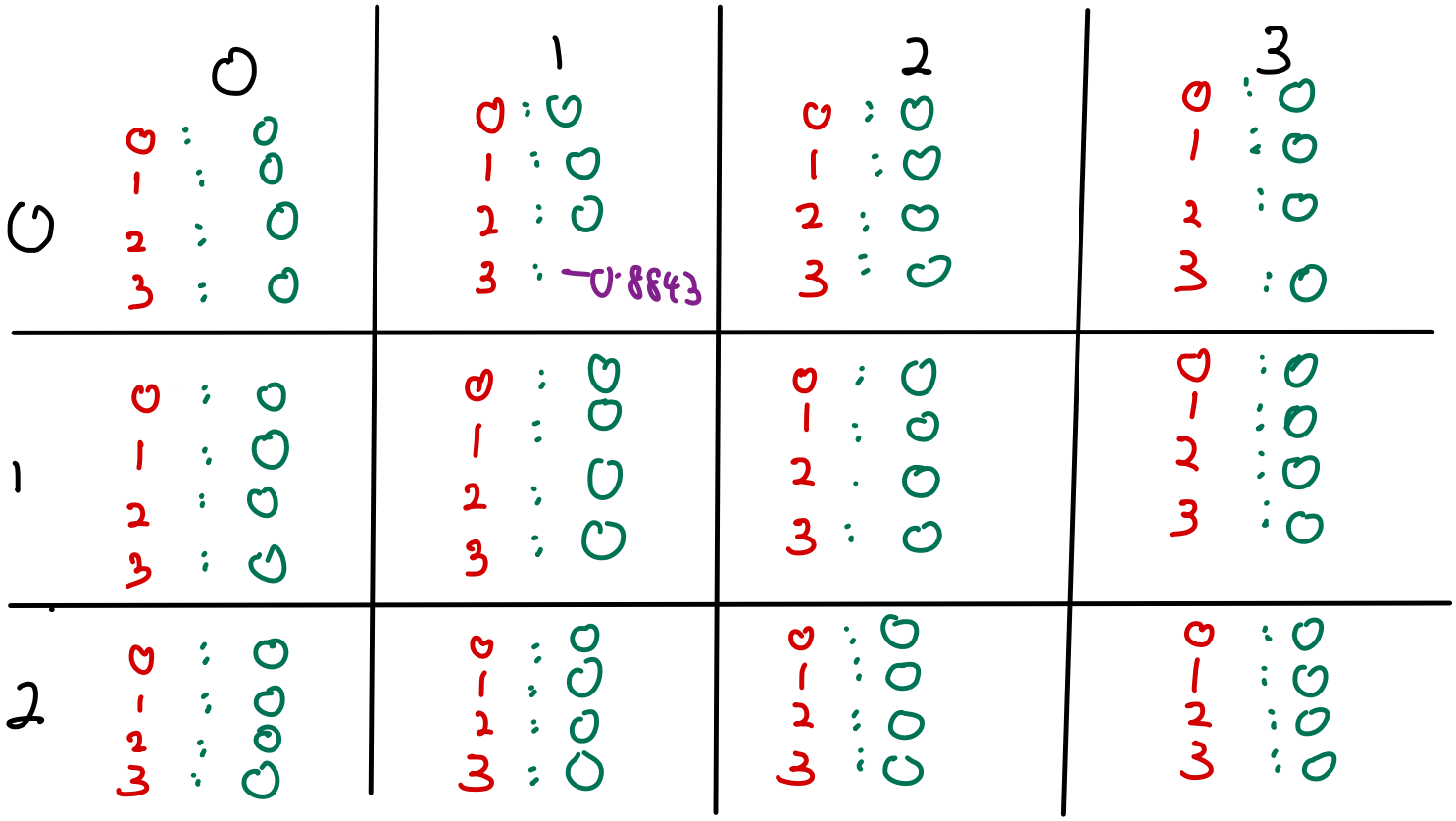
$$\text{new value} = 0 + (-4.57 - 0)(0.1)$$

$$= -0.457$$

At time step 4:

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

$$= -0.8843$$



Back to updating the policy,

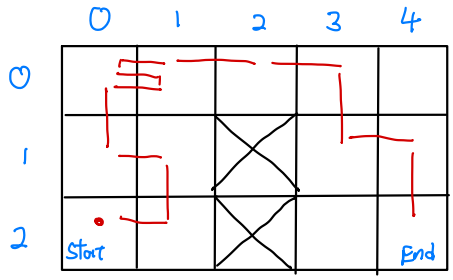
This can be done in numerous ways:

a) Policy Gradient method

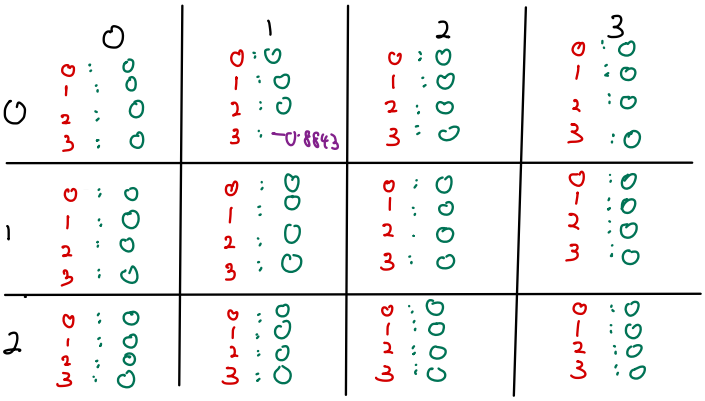
b) Policy Gradient method, with value function

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

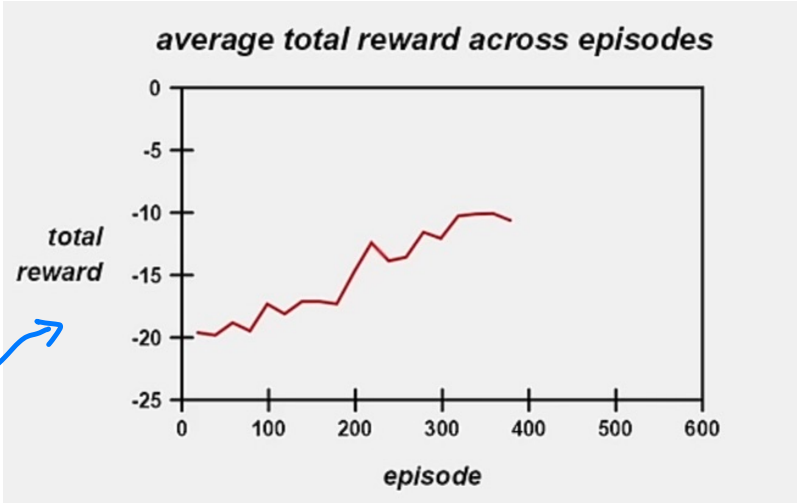
- Compute return (discounted)

↑ 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

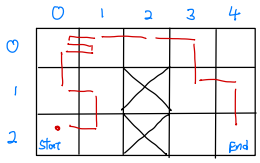
} because of temporal difference

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

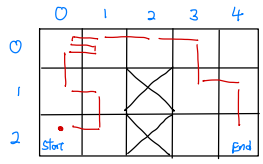
- Update action-value, $Q_\pi(S,a)$ via Monte Carlo (shown above)

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

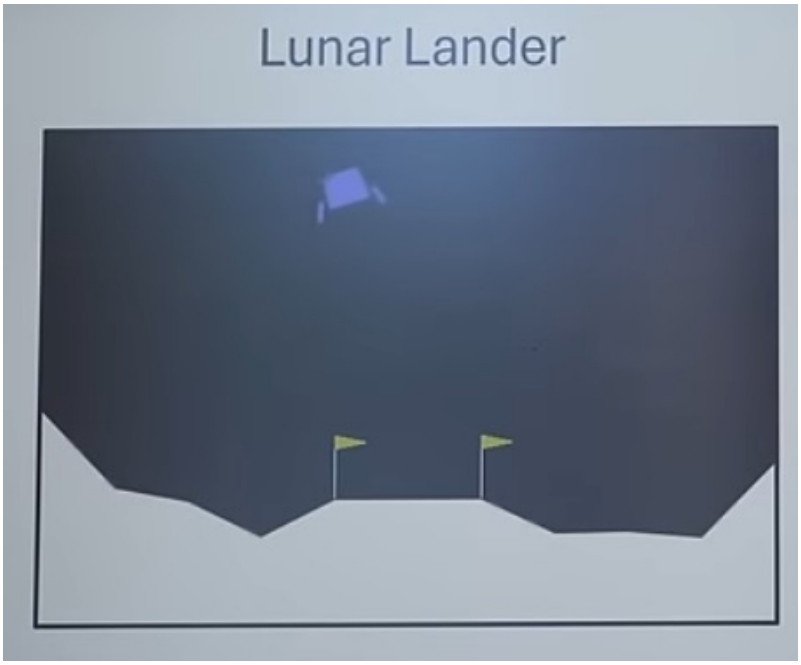
Deep Q Learning basically replaces the action value table used in Monte Carlo / Temporal Difference with a Deep Learning Network

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

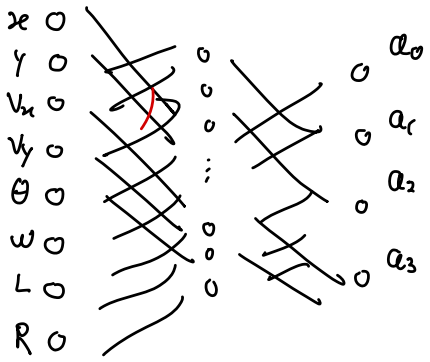
f) Decrease by 0.3 each step the main engine is firing

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

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

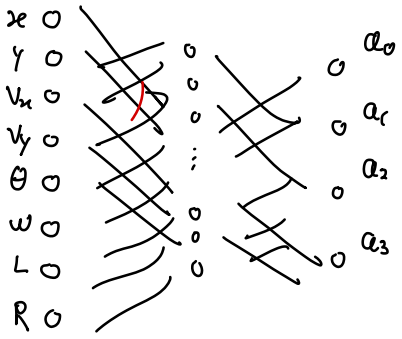
How is Deep Q Learning applied here ?

Initialization



θ
Main Network

→ use to decide actions to be taken during sampling

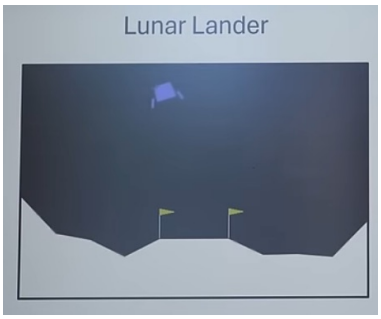


θ^-
Target Network

→ 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 :

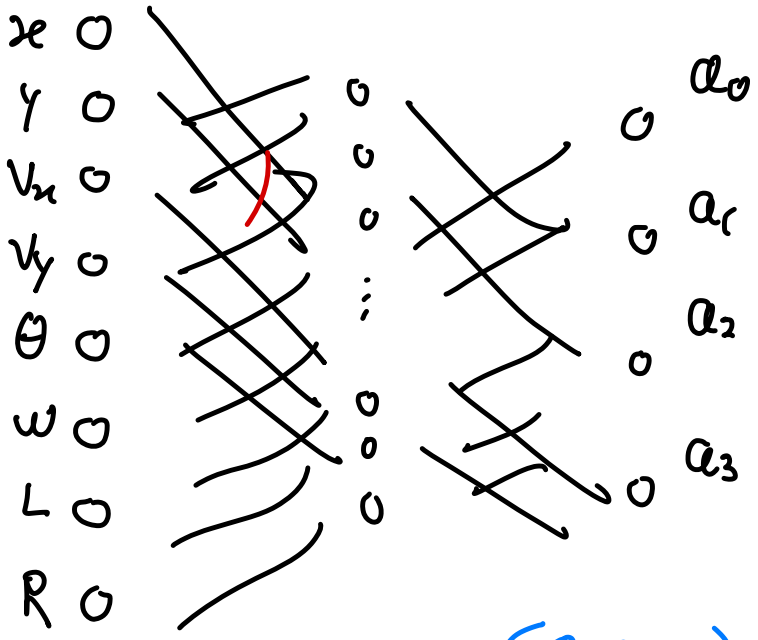
a = Randomly selected

→ 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

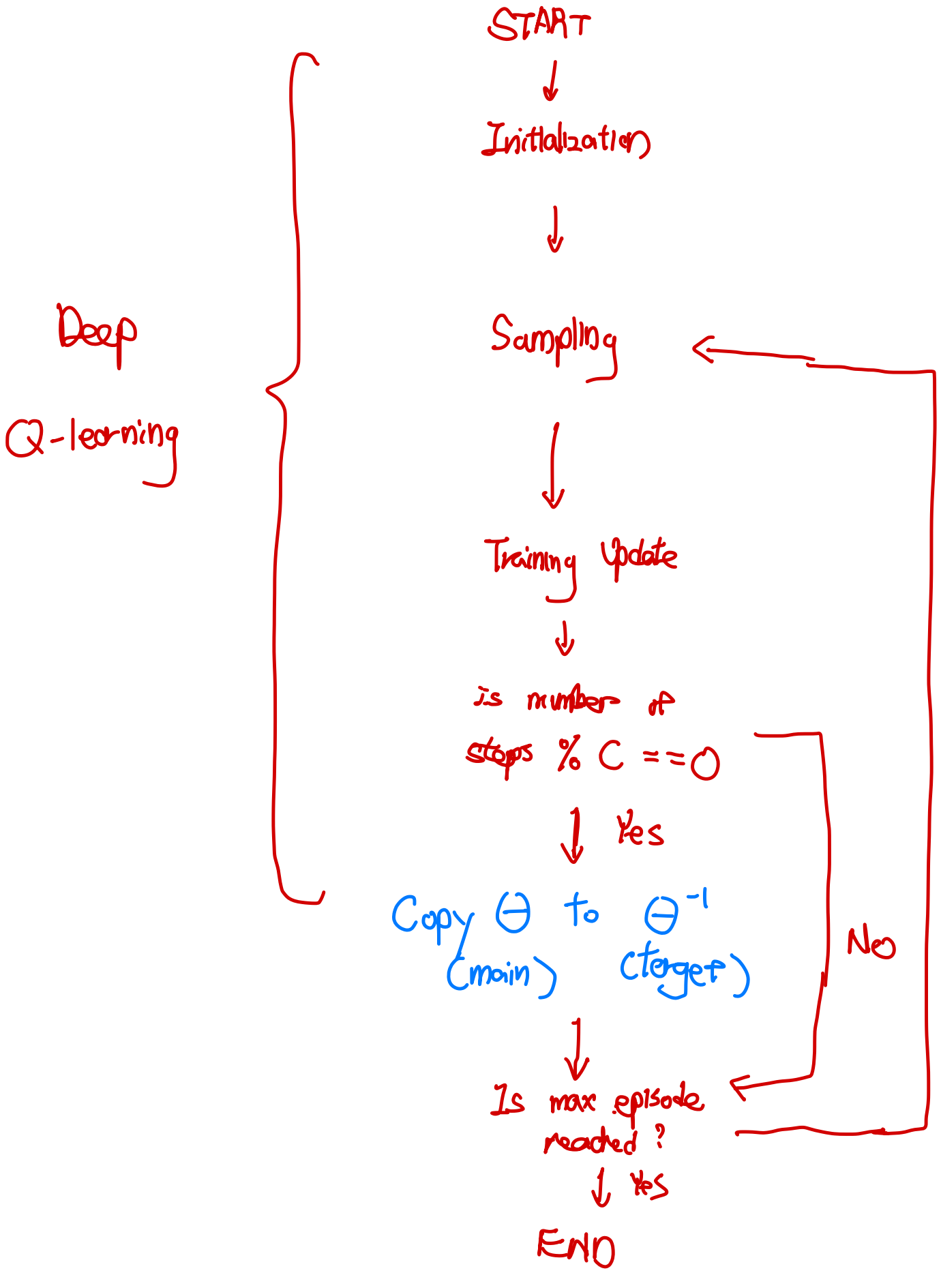
1) Pass next state, s' to the target network, θ^{-1}
2) To get a list of action value $Q_{\theta^{-1}}(s', a')$
3) From these action values, pick the one that is maximum ($\max_{a'}$)

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

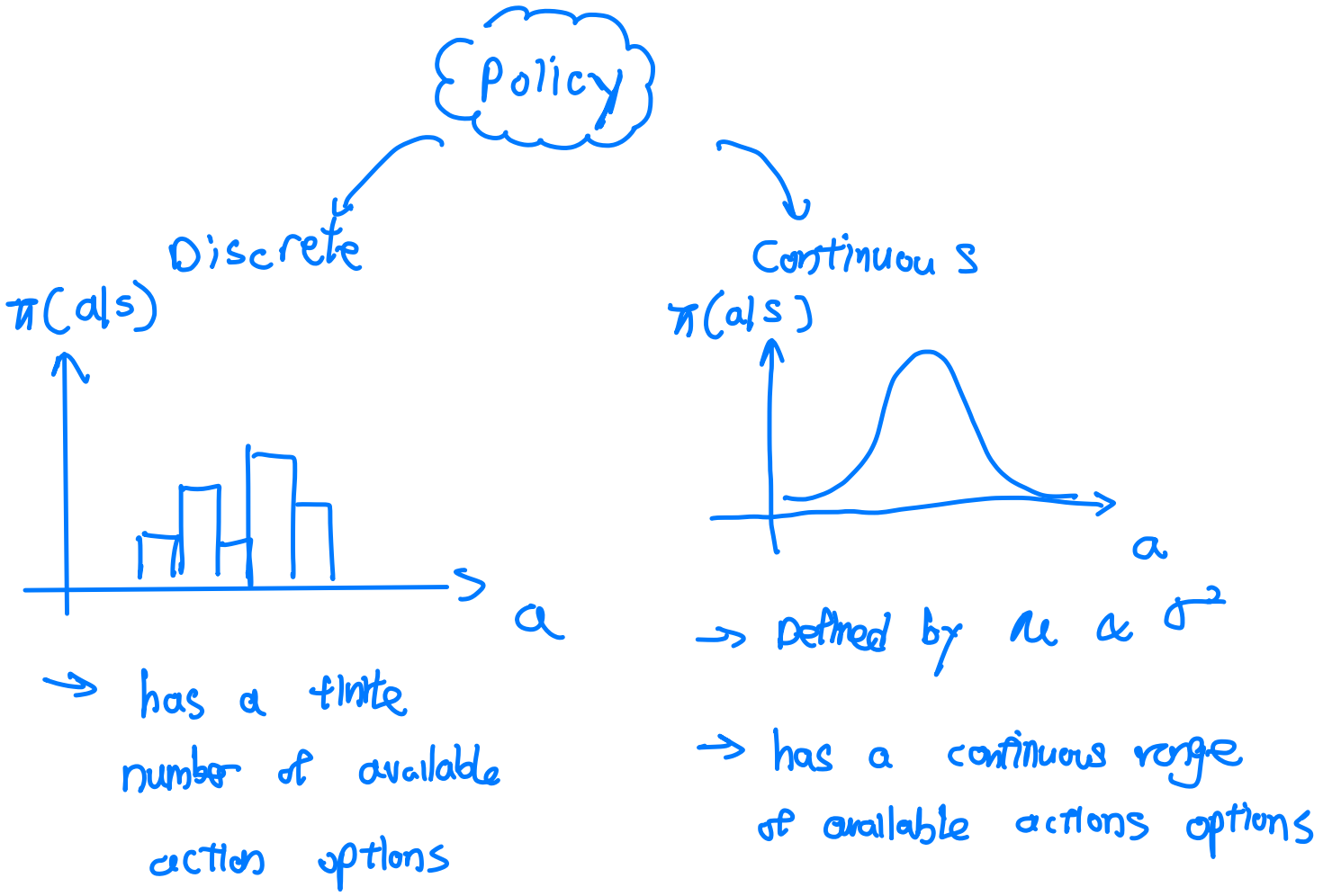
Chapter 5 :

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

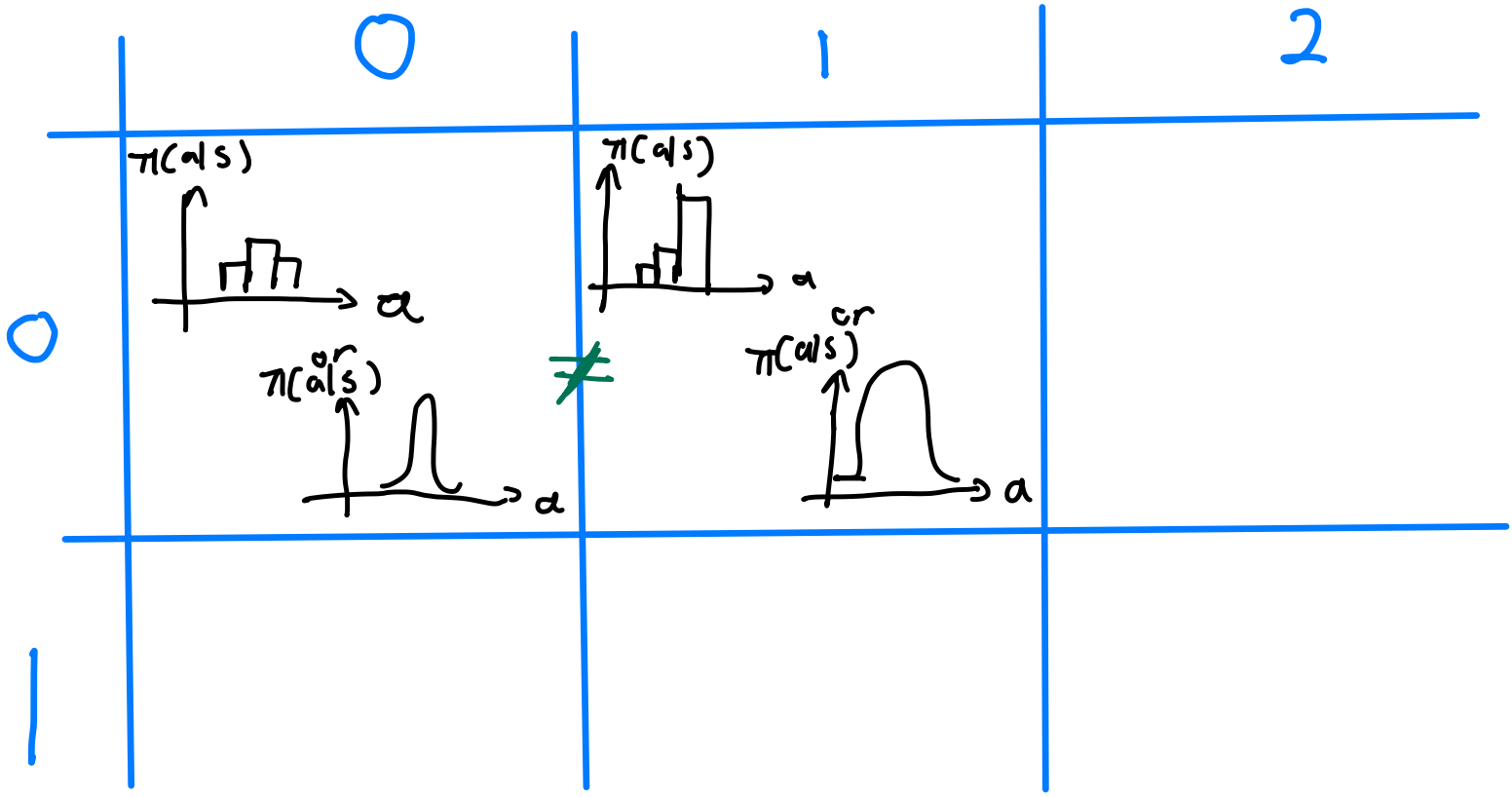
→ has a continuous range of available actions options

Some key concept :

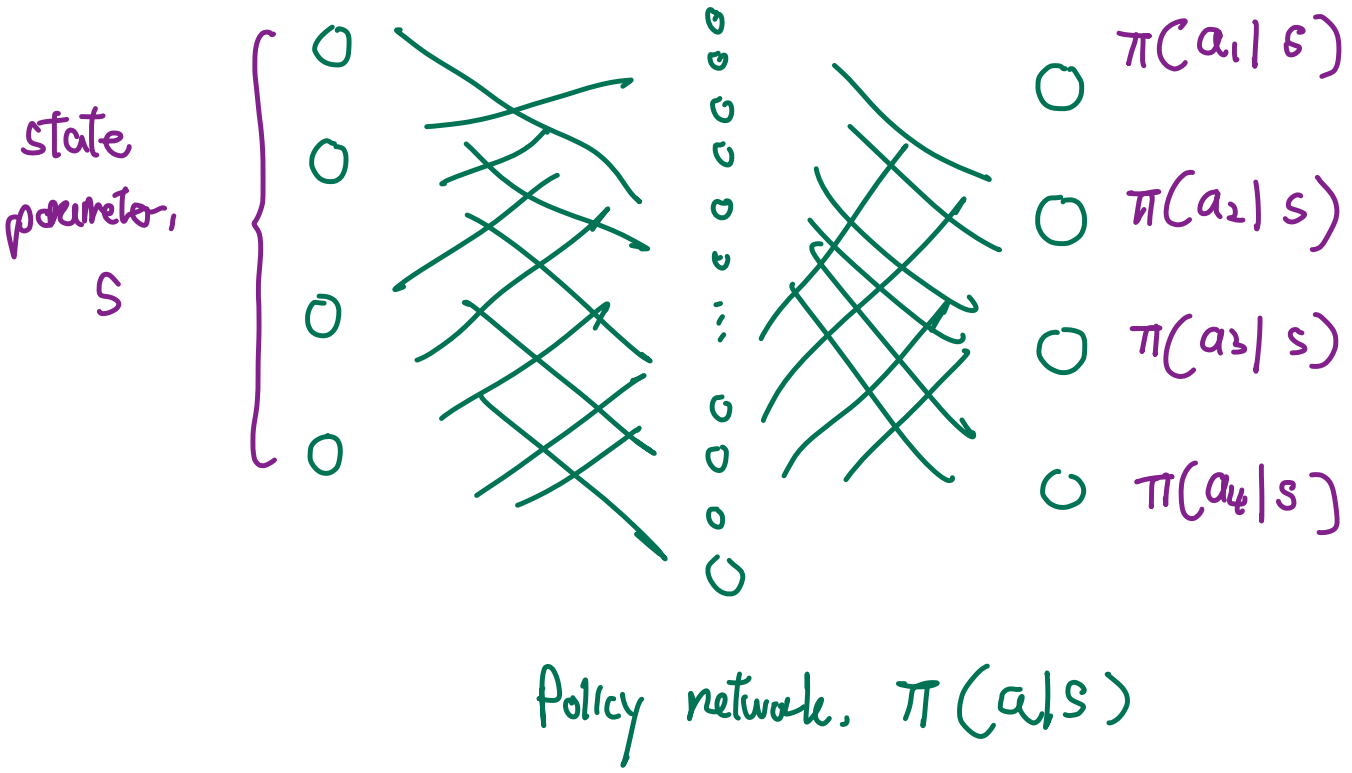
a) When a policy is involved, when decision making. a random action will be sampled from the distribution.

* 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



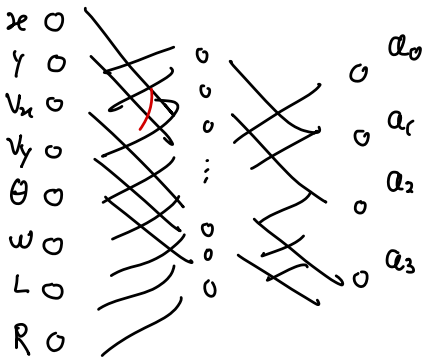
However, when a neural network is introduced to govern the policy, only 1 neural network would ever be needed



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

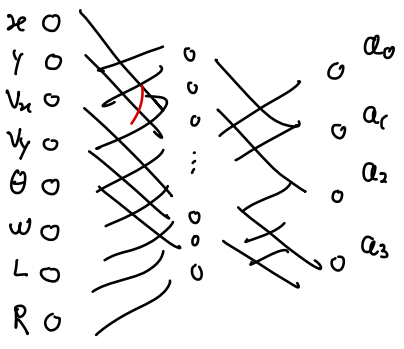
But how do you train this policy network?

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

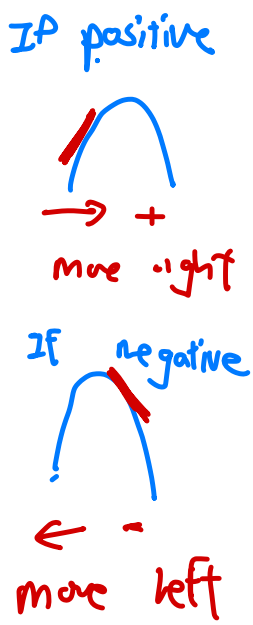
TD Error → update policy network through advantage (critic)

→ update value network (actor)

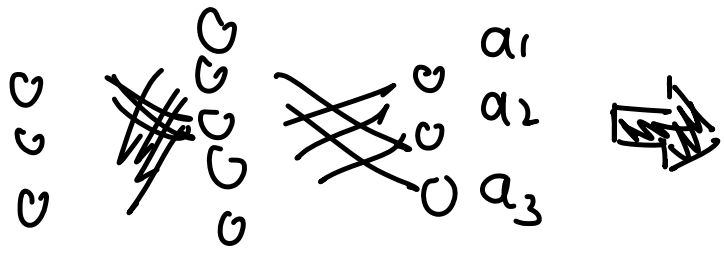
Once $\nabla_\theta J(\pi_\theta)$ is obtained, GRADIENT ASCEND can then be conducted:

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

α: learning rate



When policy network is to account for discrete actions



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

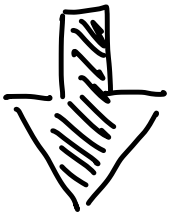
- Compute $\Delta_\theta J(\pi_\theta)$
- Update policy network via Gradient Descent

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)

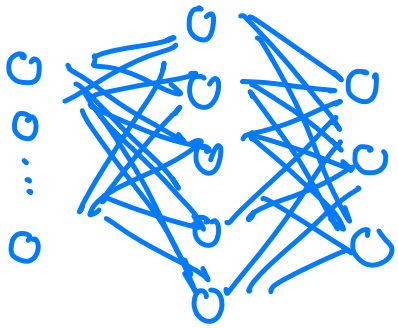
(2) In PyTorch, it is easier in implementing to minimize something rather than maximizing it.

Therefore, a negative sign is added

(3) using ln instead of log

Design of policy network to use Reinforce

a) Discrete space

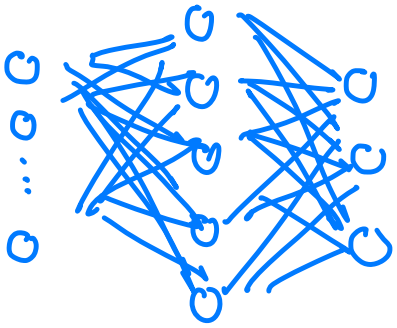


Each output node represents probability of taking a discrete action

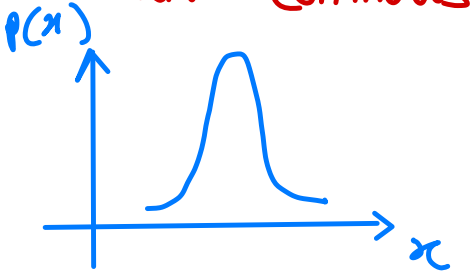
can be applied into easily

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

b) Continuous space

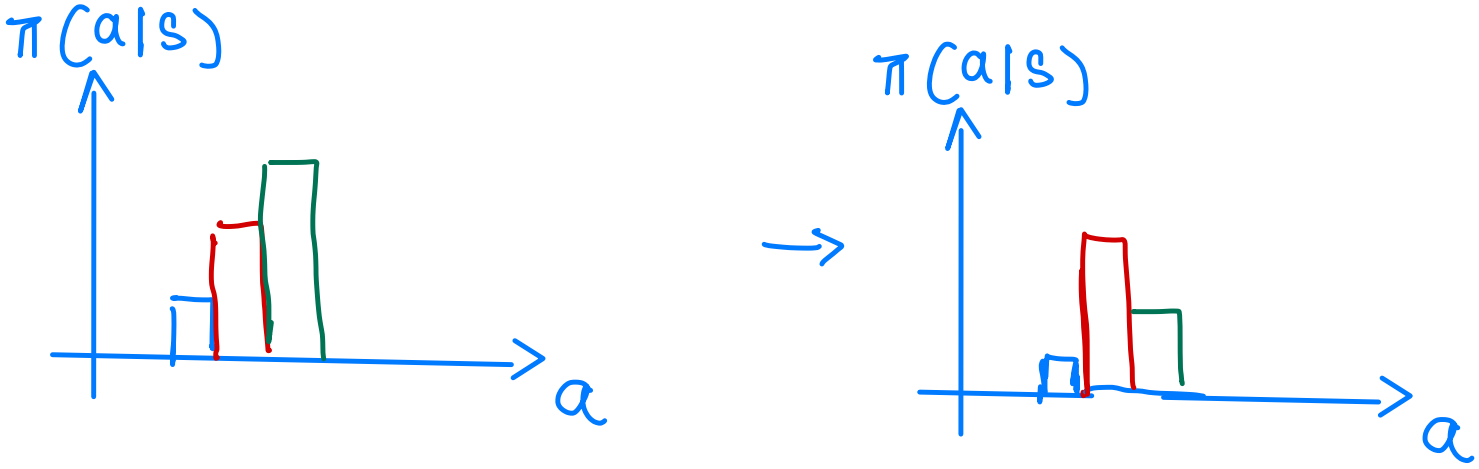


Each pair of output nodes represent mean & std_dev of a Gaussian probability distribution for an continuous action



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

Visualization of policy update in discrete actions



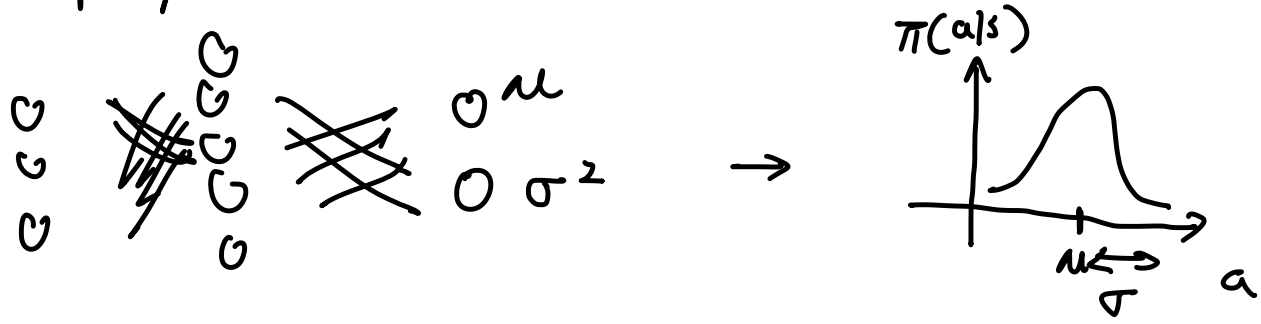
- Due to the evaluation made by the advantage, 卐, a_2 is deemed a good action, receive a high $\Delta_\theta J(\pi_\theta)$ where the parameter in the policy network are tuned in such a way that leads to higher $\pi(a_2|s)$.

- 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

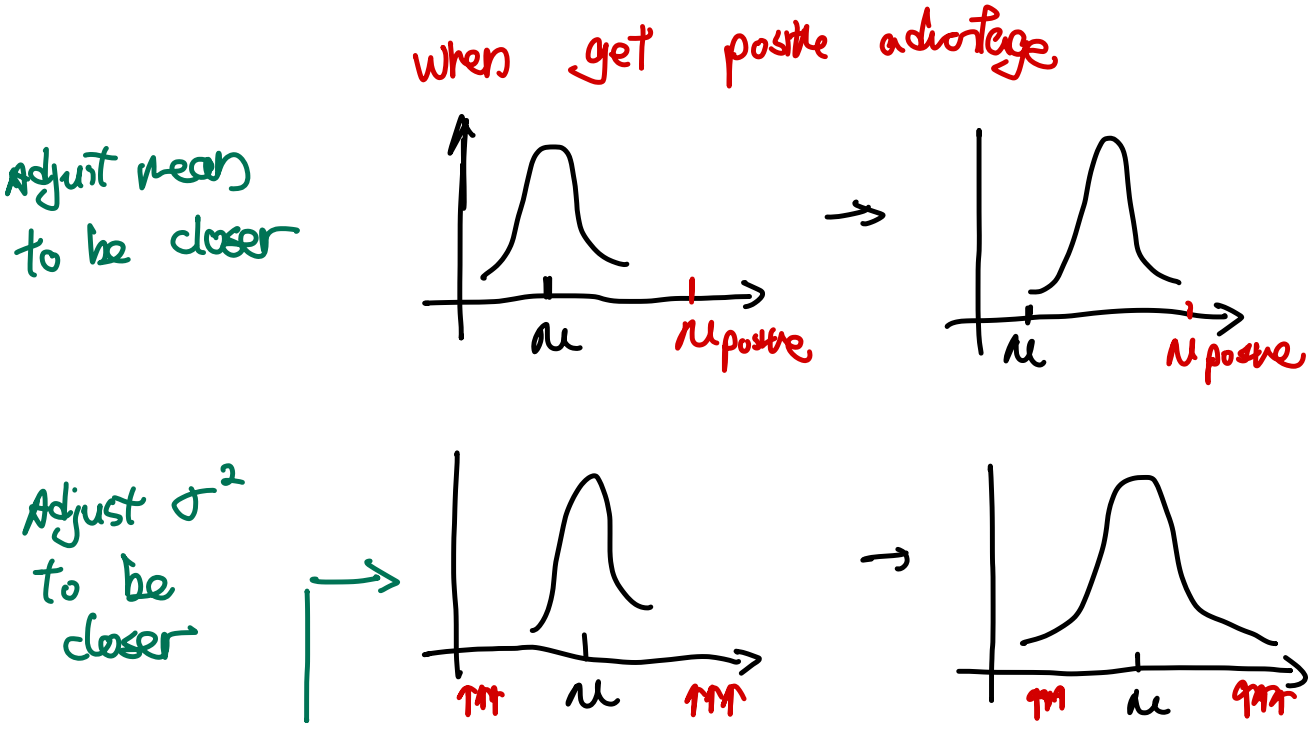
卐 a_1, a_3 → get negate advantage

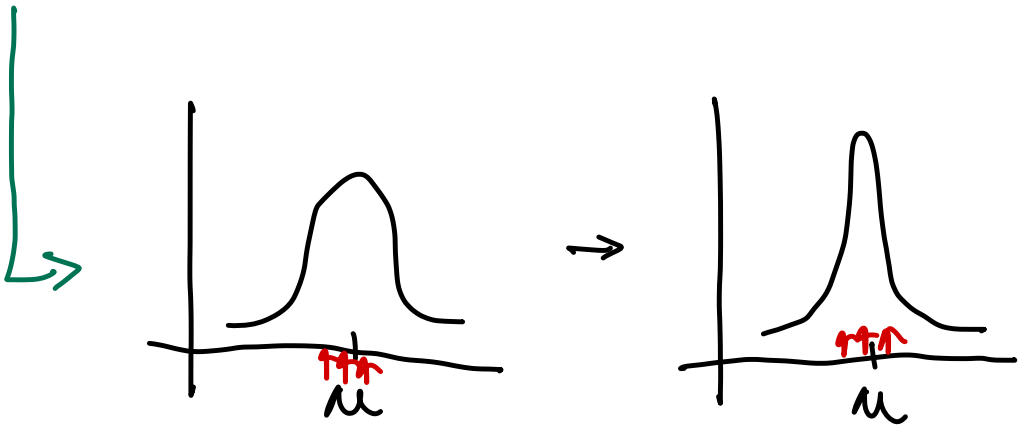
When policy network is to account for continuous actions



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

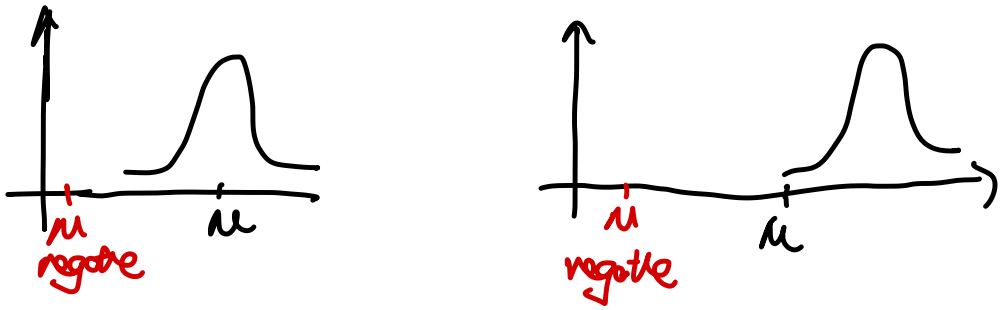
Visualization of policy update in continuous actions



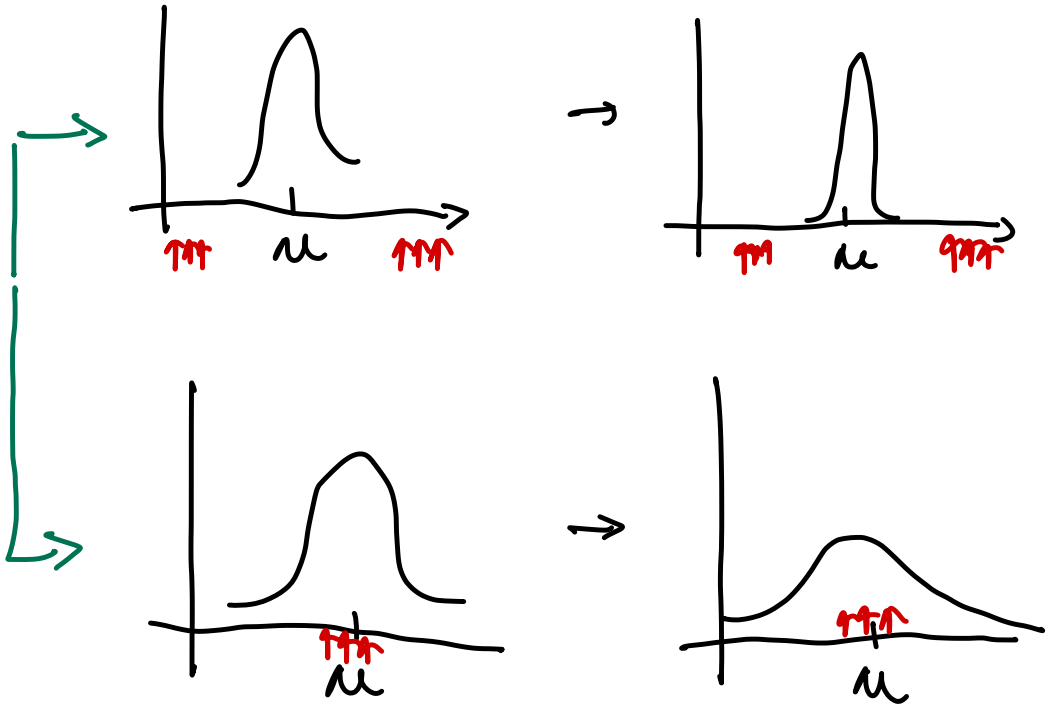


When get negative advantage

Adjust mean to be further



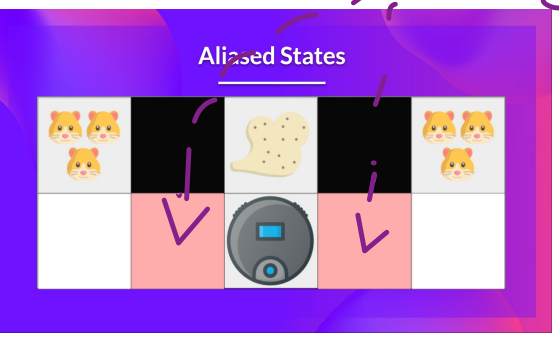
Adjust σ^2 to be further



Advantage of Policy Gradient over Deep Q Learning

1) Can learn a stochastic policy

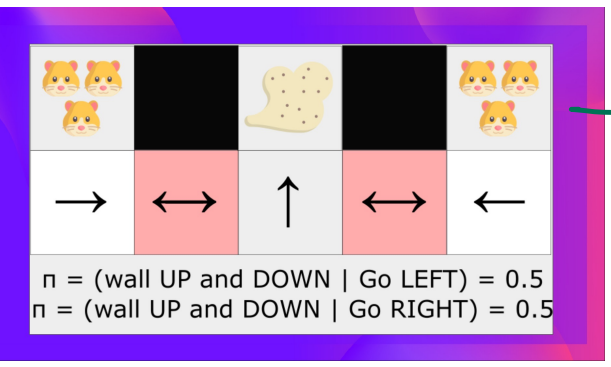
- No longer need to compute epsilon-greedy policy manually
- Get rid of perceptual aliasing



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

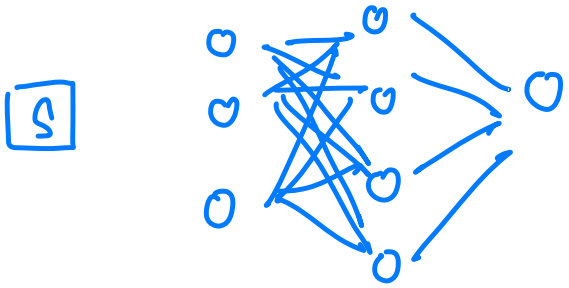
In Deep Q learning.

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

↳ 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

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

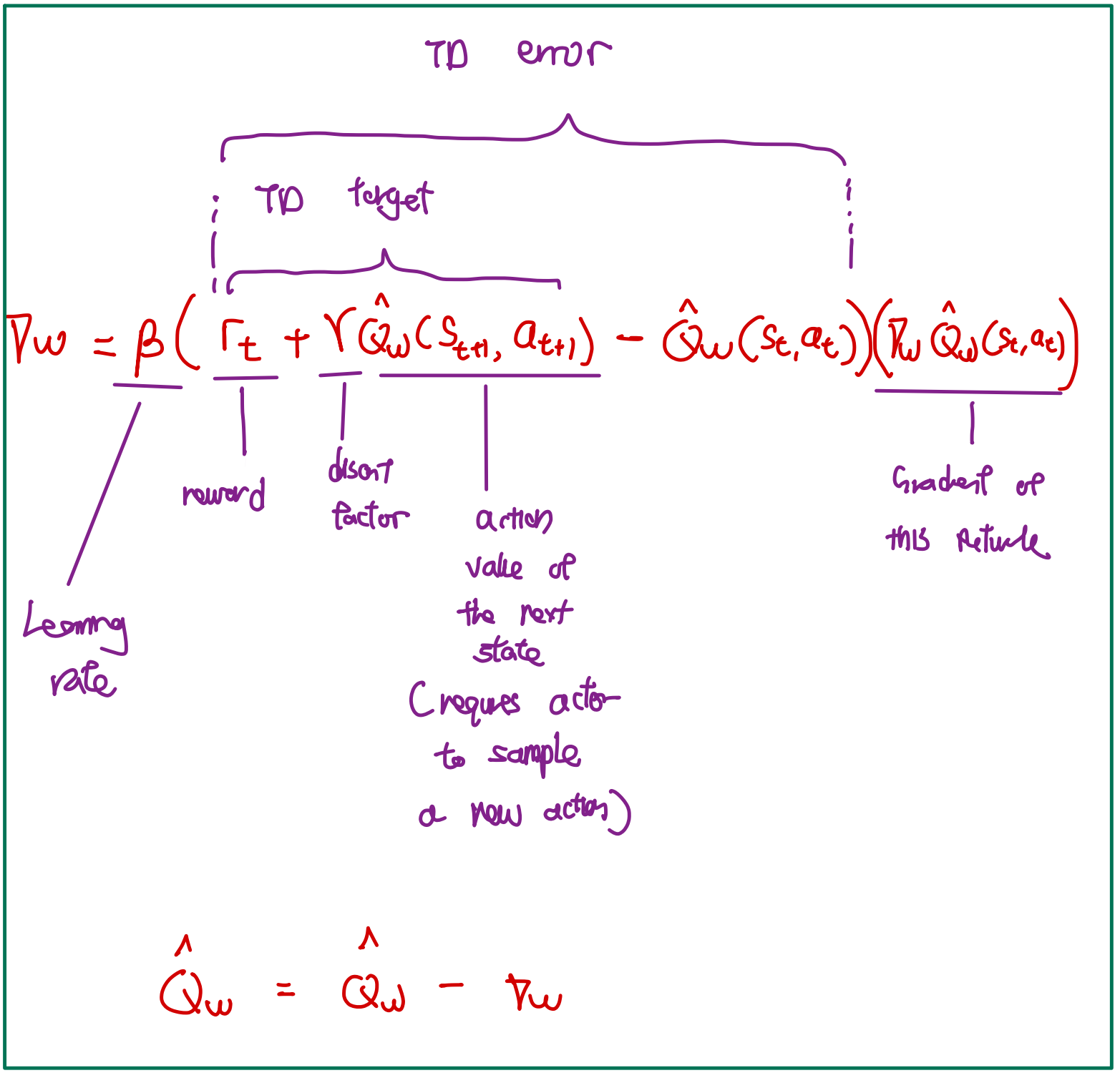
$$\pi_\theta = \pi_\theta + \underbrace{\alpha} \nabla_\theta (\pi_\theta)$$

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.

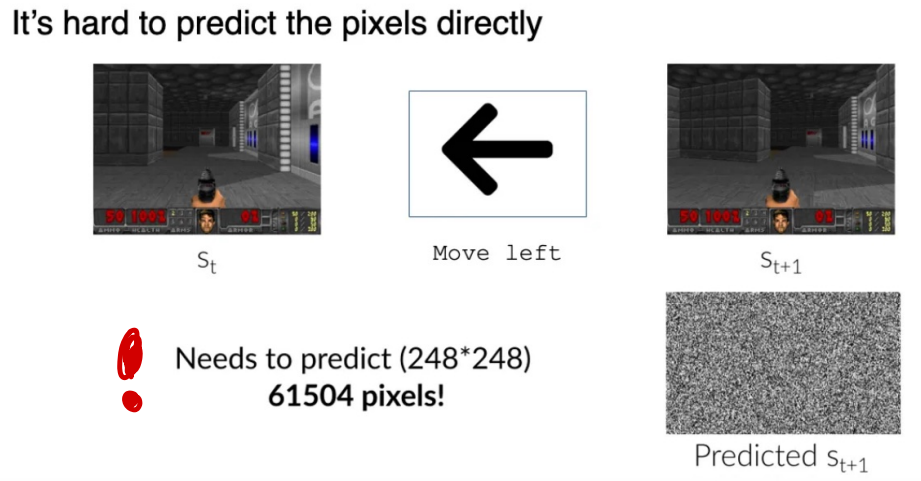
Solution : Intrinsic Reward

- 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

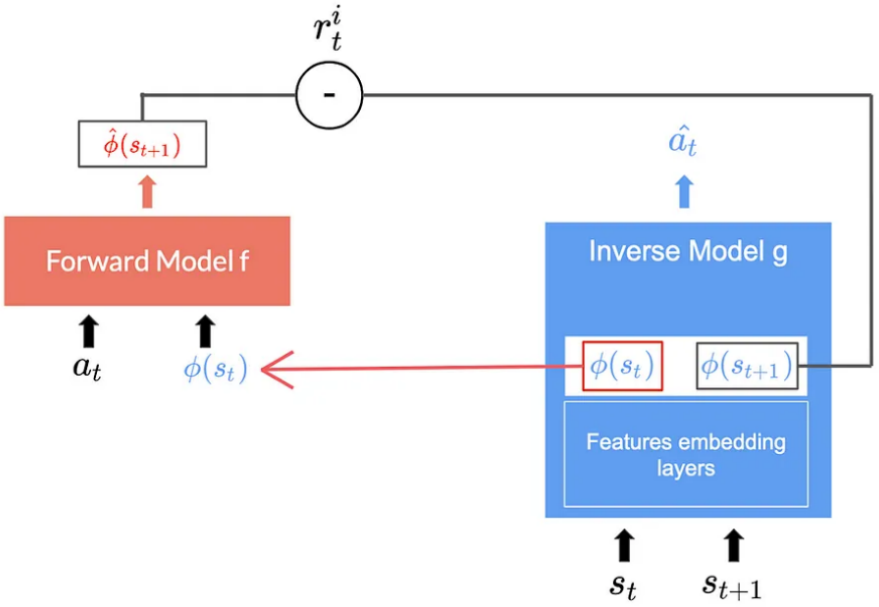
- A few rules to follow in modelling the state's feature representation are :

  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

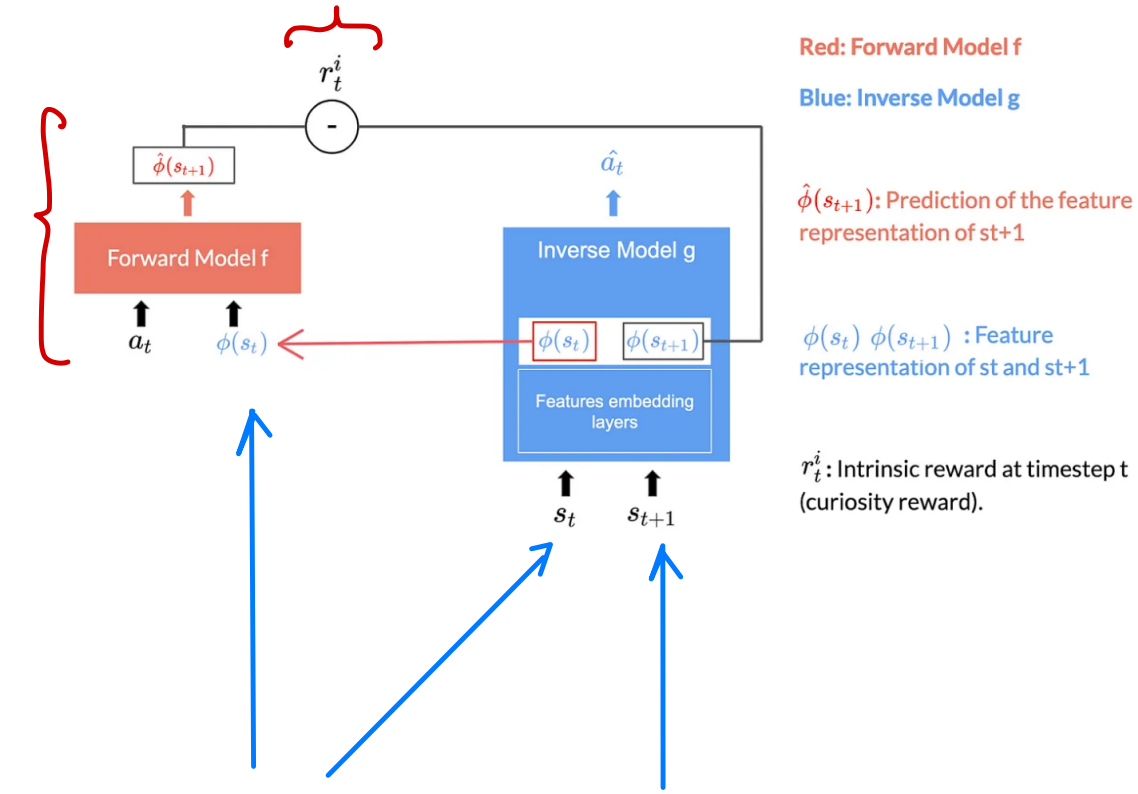
- However, when the next state is random & stohastic, ie. a TV problem, the agent has a hard time predicting it & thought there's high curiousity, therefore drawn to it & stopped performing meaningful exploration

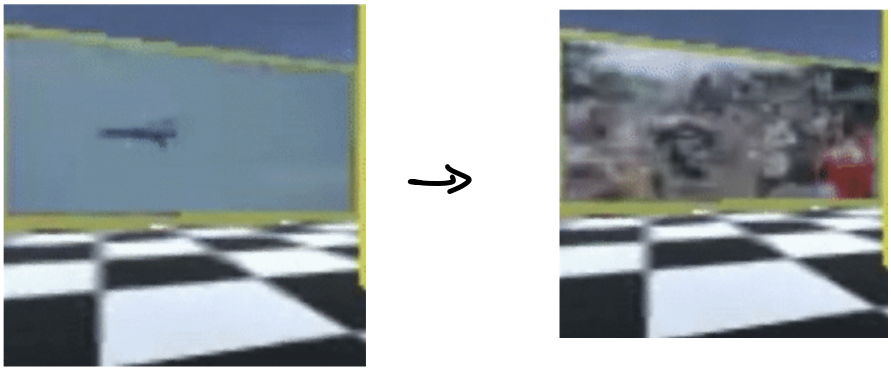
- The agent has a hard time predicting because inverse model rely on the prediction of action taken to tweak the inverse model's parameter which subsequently affect the feature representation.

- 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

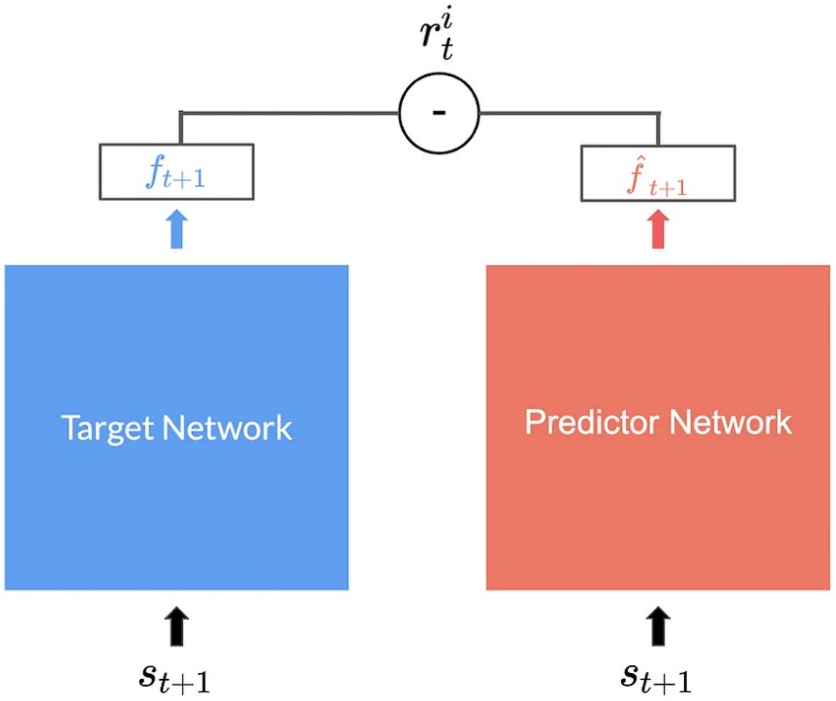
Has a hard time predicting the next state





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:

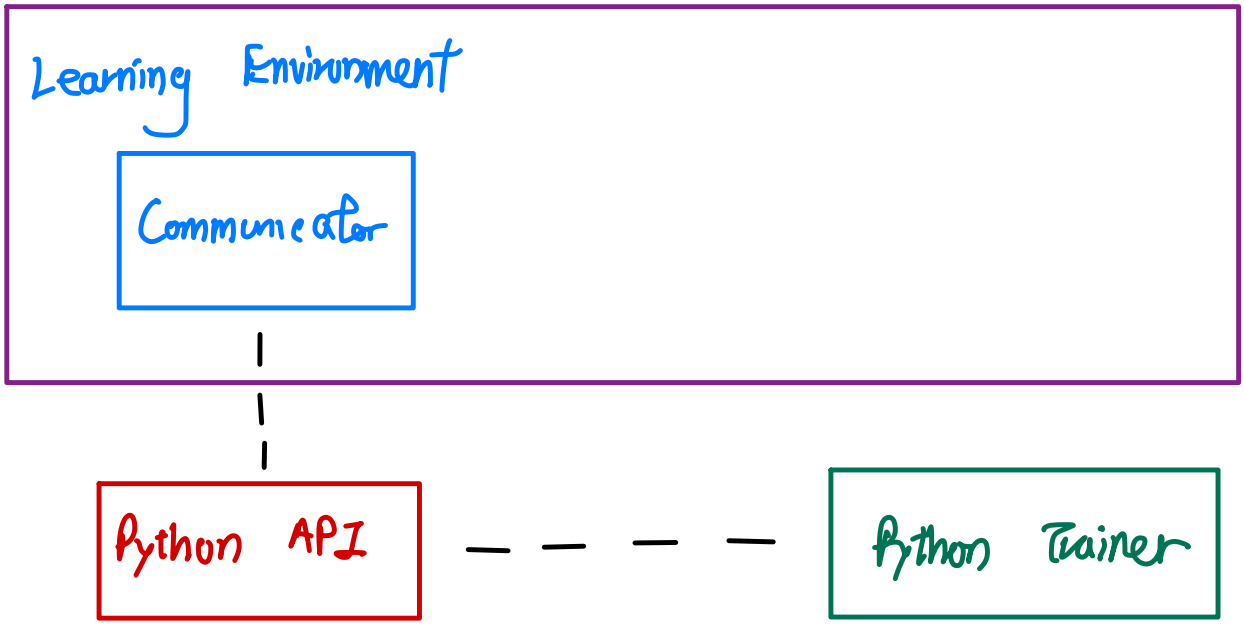
Introduction to Unity ML

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

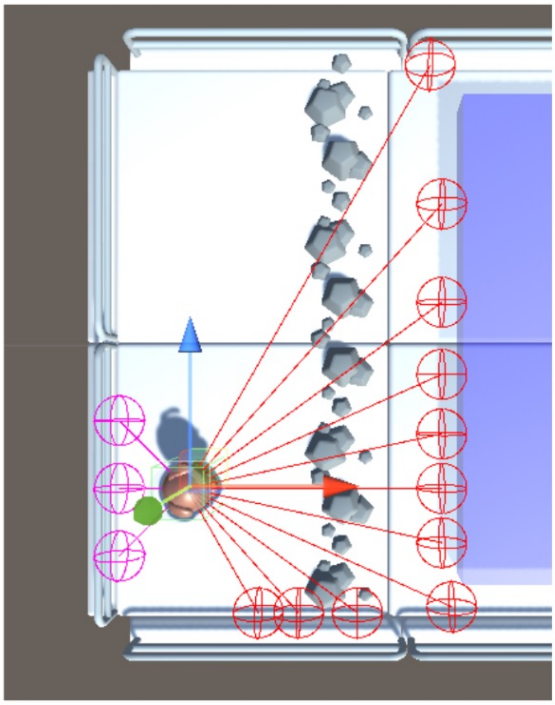
within learning components

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:

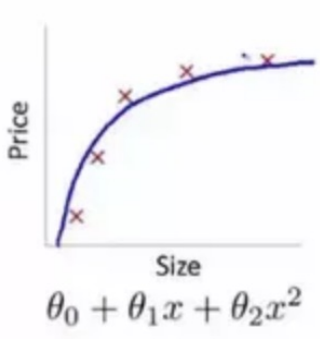
Bias & Variance

Tradeoff in RL

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

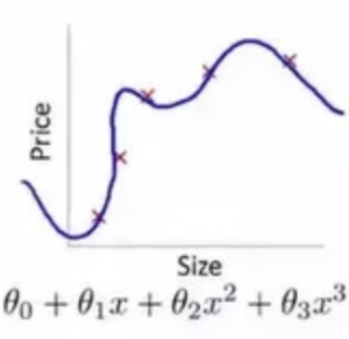
$\theta_0 + \theta_1 x$

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

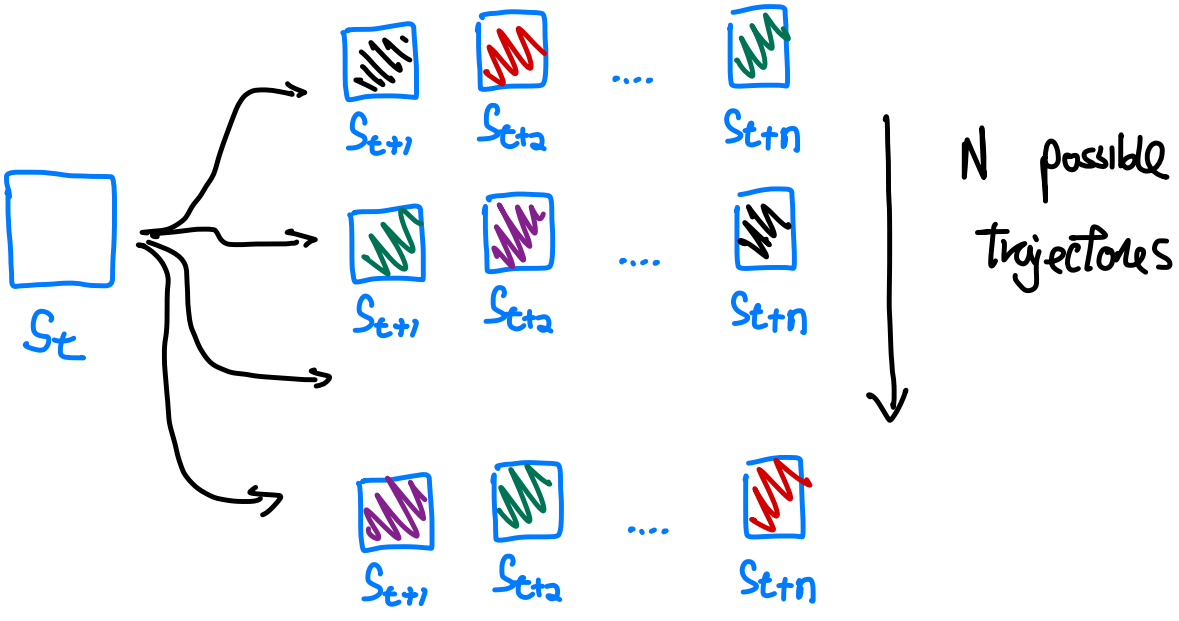
Similarly, bias & variance problem also exist in RL, but in a different way

> 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,

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

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

- 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.