Module: 3 – Temporal Difference Learning:

1. TD (Temporal Difference) Learning is a popular model-free method.

2. It combines the advantages of both DP method and MC method.

Recap of Adv. and Disadv. of DP and MC methods:–

DP: **Adv:** We use Bellman eqn to compute the value of a state, i.e, the value of a state is the sum of the immediate reward and the discounted value of the next state – called bootstrapping. So to compute the value of a state, we don't have to wait till the end of the episode, instead using the Bellman eqn., we can estimate the value of a state, just based on the value of the next state

$$V(s) = \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V(s') \right]$$

Disadv: We can implement DP only when the model dynamics is known [transition prob of the states. model-based method]

Monte-Carlo: **Adv:** model-free method. It doesn't require the model dynamics to estimate the value & Q functions

Disadv. To find the value fn & Q fn of (s,a) we need to wait until the end of the episode and if the episode is long, then it will cost us a lot of time.

→ MC methods cannot be applied to continuous tasks (non-episodic tasks) without a terminal state.

∴ In TD-learning: Adv:

1) We perform bootstrapping, so that we don't have to wait till the end of an episode to find the Q-value & state-value.

2) Like MC, it is a model-free method.

~~We can use TD-learning algom, for both the predn & control.~~

TD-learning can be categorized into:-

1) TD-Prediction and 2) TD-control.

In TD-Prediction

→ a policy is given as an i/p & we try to predict the value fn & Q-fn using the given policy.

→ Using this, the agent will come to know how good it is for the agent to be in each state if it uses the given policy.

→ the agent can understand what is the expected return it can get, if it acts according to the given policy in each state.

In TD-control: we are not given a policy as i/p, but the goal is to find the optimal policy.

∴ we initialize a random policy & then we try to find the optimal policy iteratively.

∴ This optimal policy will give the max. return.

TD Prediction:-

→ A policy is given as i/p, & we try to estimate the value fn of each state using the given policy.

→ TD used ~~bootstrapping~~ like DP, so it doesn't have to wait till the end of the episode

→ Like MC method, it doesn't require the model dynamics of the envt to find the value & Q fns.

→ the update rule of TD takes these advantages into account.

In the MC method,

$$V(s) \approx R(s)$$

∵ a single return, cannot approximate the value of a state, perfectly, we take the mean of the return over N episodes.

$$V(s) \approx \frac{1}{N} \sum_{i=1}^{N} R_i(s)$$

∵ TD learning doesn't have to wait till the end of an episode, it uses bootstrapping to estimate the value of a state without using the model dynamics

$$V(s) \approx r + \gamma V(s')$$

↳ immediate reward r

∵ a single value V(s) cannot approximate the value of a state perfectly, we take the incremental mean as,

$$V(s) = V(s) + \alpha (R - V(s)) \leftarrow \left\{ \text{used in } \underline{MC} \right.$$

In <u>TD</u>:

$$V(s) = V(s) + \alpha \left(r + \gamma V(s') - V(s) \right)$$

Difference betn the update rule in MC & TD methods. -

In MC method

$$V(s) = V(s) + \underbrace{\alpha}_{\text{learning rate or step size}} (\underbrace{R}_{\text{full return, computed using the full episode}} - V(s))$$

In TD Learning $V(s) = V(s) + \alpha(\underbrace{r + \gamma V(s')}_{\text{Bootstrap estimate}} - V(s))$

computed without having to wait till the end of the episode; r is the immediate reward.

w.r.t: $r + \gamma \cdot V(s')$ ... an estimate of $V(s)$ called TD target.

∴ TD learning update rule is

$$V(s) = V(s) + \underbrace{\alpha}_{\text{learning rate}} [\underbrace{\underbrace{r + \gamma V(s')}_{\text{TD target}} - \underbrace{V(s)}_{\text{Predicted value}}}_{\text{TD error}}]$$

Using TD-learning in Prediction tasks:-

In TD predn, a policy is taken as i/p, & using the update rule of TD-learning, the V(s) is updated.

Finally, we get the expected return an agent can obtain in each state, if it acts according to the given policy.

W.K.T TD-learning update rule is

$$V(s) = V(s) + \alpha\left(r + \gamma V(s') - V(s)\right)$$

Ex: Applying TD Prediction in Frozen Lake Env

Pg: 196:

Given policy:

State	Action
(1,1)	Right
(1,2)	R
(1,3)	L
⋮	
(4,4)	Down

Initialize the value table to random values:

State	value
(1,1)	0.9
(1,2)	0.6
(1,3)	0.8
⋮	⋮
(4,4)	0.7

Say we are in state (1,1).

Step 1: Find the value of state (1,1) as per the policy

As per the given policy, take right & next state is (1,2), reward = 0.

Assume $\alpha = 0.1$ and $\gamma = 1$.

$$\therefore V(S) = V(S) + \alpha \left(r + \gamma V(S') - V(S) \right)$$

$$\therefore V(1,1) = V(1,1) + 0.1 \left(0 + 1\, V(1,2) - V(1,1) \right)$$

$$= 0.9 + 0.1 (0.6 - 0.9)$$

$$\therefore V(1,1) = 0.87$$

$\therefore$ we update this in state table.

State	Value
(1,1)	0.87
(1,2)	0.6
(1,3)	0.8
(4,4)	0.7

Step 2:

Current state is (1,2)

As per policy:-

$s = (1,2)$; $a =$ right,

$r = 0$; $s' = (1,3)$

Update $V(1,2)$

$$\therefore V(1,2) = V(1,2) + 0.1 \left(0 + 1 \cdot V(1,3) - V(1,2) \right)$$

$$= 0.6 + 0.1 (0.8 - 0.6)$$

$$V(1,2) = 0.62$$

update the state value table

State	Value
(1,1)	~~0.8~~ 0.87
(1,2)	0.62
(1,3)	0.8
⋮	
(4,4)	0.7

Step 3:

current state is (1,3) ; ~~s'~~ = a = left ;

r = 0 ; s' = (1,2)

∴ updated value of (1,3) is

$$V(1,3) = V(1,3) + 0.1\left(0 + 1 \cdot V(1,2) - V(1,3)\right)$$

$$= 0.8 + 0.1\left(1 \cdot 0.62 - 0.8\right)$$

$$V(1,3) = 0.782$$

∴ updated value table :-

State	Value
(1,1)	0.87
(1,2)	0.62
(1,3)	0.782
⋮	
(4,4)	0.7

||ly, we calculate the value of every state using this policy.

However, to be more accurate we repeat this over several episodes & get the accurate estimates of the state value fn.

∴ The TD predn algom is:-

1. Initialize the value fn of all states $V(s)$ with random values. Take a policy π as input.

2. For each episode:-
   1. Initialize state s.
   2. For each step in the episode:
      1. Perform an action 'a' in state 's' according to the given policy π, get the reward r, and move to the next state s'.
      2. Update the value of the state to
         $$V(s) = V(s) + \alpha\left(r + \gamma V(s') - V(s)\right)$$
      3. Update $s = s'$ (next state becomes the current state)
      4. If s is not the terminal state, repeat steps 1 to 4.

TD predn algorm in the FZE:-

```
# import libraries
import gymnasium as gym
import pandas as pd

# create the env
env = gym.make('Frozen

# define the random policy
def random_policy():
    return env.action_space.sample()

# define a dictionary to store the value of states
V = {}
for s in range(env.observation_space.n):
    V[S] = 0.0

# Initialize the discount factor γ and
# learning rate α
alpha = 0.85
gamma = 0.90

# set the no. of episodes and timesteps in each
                                      episode
num_episodes = 50000
num_timesteps = 1000
```

```
# Compute V(s) all states using the given
# policy

# for each episode
for i in range(num_episodes):
    # Initialize the state by resetting the env
        s = env.reset()

    # for every step in this episode
        for t in range(num_timesteps):
    # Select an action according to random policy
            a = random_policy()
    # Perform the action and get the next
    # state, and reward, r
            s_, r, done, _ = env.step(a)
    # Compute the updated value of state s'
    # using TD-learning update rule
    # V(s) = V(s) + α [ r + γ V(s') - V(s))
            V(s) += alpha * [r + gamma *
                          V[s_] - V[s])
    # update next state current state
            s = s_
```

if s is terminal state, then break

```
        if done:
            break
```

∴ After all iterations, we will have the value of all states, according to the given policy.

Evaluating the values of the states.

```
# convert the values dictionary to a pandas
#data frame.
df = pd.DataFrame(list(V.items()),
                  columns=['state', 'value'
```

```
# print the value of the states
df
```

check the output on page Pg: 205.

TD Control:- Here, the agent starts with a random policy, & finds the optimal policy iteratively.

Control method is of 2 categories:-

1) on-policy control:- the agent behaves using one policy and tries to improve the same policy. ie, episodes are generated using one policy, and the agent improves it iteratively to find the optimal policy

Ex. SARSA: on-policy TD-control algm

SARSA:- state-Action-Reward-State-Action

2) off-policy control:- the agent behaves using one policy, ~~using~~ but tries to improve a different policy. ie, episodes are generated using one policy, and the agent improves a different policy iteratively, to find the optimal policy.

Ex: Q-learning:

On-Policy TD Control - SARSA :-

In TD-control, our goal is to find the optimal policy using the Q-function.

Once we have the Q-fn, we extract the policy, by selecting the action in each state that has the max Q-value.

How to compute the Q function in TD learning

Same as the update rule in TD learning for V(s),

The update rule in TD-learning for the Q fn is

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma Q(s',a') - Q(s,a)\right)$$

This is also known as the SARSA update rule. Using this rule, the Q-table is updated in each time step. Over multiple episodes, we extract the optimal policy from the Q-table finally.

But, without any policy, how to react in the envt?

∴ we initialize the Q table with random values or zeros.

Extract a policy from this randomly initialized Q fn to act in the env.

Our initial policy will not be optimal, ∵ it is generated from a randomly initialized Q table.

But every episode, keeps updating the Q table.

So, in every episode, we use the updated Q table to extract a new policy

Thus, after a series of episodes, we get an optimal policy.

In SARSA method, the policy used to select an action in each state is epsilon-greedy instead of greedy.

In greedy approach/policy :- we always select the action that has the max. Q value

In epsilon-greedy policy :- we select a random action with a prob of epsilon and we select the best action with a prob of 1 − epsilon.

Ex: SARSA in FZE:-

Initial Q-table with random values:-

State	Action	Value
(1,1)	up	0.5
(1,2)	...	
⋮		
(4,2)	up	0.3
(4,2)	down	0.5
(4,2)	left	0.1
$\boxed{(4,2)}$	$\boxed{\text{right}}$	$\boxed{0.8}$
⋮		
(4,4)	right	0.5

FZE

	1	2	3	4
1	S	F	F	F
2	F	H	F	H
3	F	F	F	H
4	H	F	F	G

initial state

Assume we are in state (4,2). Select an action in this state using epsilon-greedy policy; ie, with a prob ε we select a random action and with a prof of $1-\varepsilon$ we select the best action (action with a max Q value).

Assume we select the best action in (4,2)

$\therefore$ a = ~~down~~ right ; $s' = (4,3)$; $r = 0$

$\alpha = 0.1$; $\gamma = 1$.

update Q-value of (4,2) using SARSA update rule:-

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma Q(s',a') - Q(s,a)\right)$$

$\therefore Q\left[(4,2), \text{right}\right) = Q\left[(4,2), \text{right}\right) + 0.1\left(0 + 1 \times Q\left[(4,3), a'\right) - Q\left[(4,2), \text{right}\right)\right.$

$$= 0.8 + 0.1\left(1 \times Q\left[(4,3), a'\right) - 0.8\right)$$

What is $Q[(4,3), a')$? Q-value of next-state action pair. Use the same epsilon-greedy policy. Assume we select a random action in the next state.

Assume we choose action right in $(4,3)$

$\therefore Q[(4,2), \text{right}) =$

$0.8 + 0.1(0 + 1(0.9) - 0.8)$

$\therefore Q[(4,2), \text{right}) = 0.81$

State	action	value
(1,1)	up	0.5
⋮		
(4,2)	right	0.8
(4,3)	up	0.1
(4,3)	down	0.3
(4,3)	left	1.0
$\boxed{(4,3)}$	$\boxed{\text{right}}$	$\boxed{0.9}$
⋮		
(4,4)	Right	0.5

Similarly, we update the Q-table ~~of~~ ~~the~~ ~~as~~ in each step of ~~the~~ an episode. After one episode, we extract a new policy from the updated Q-table. & use this policy to act in the envt during the next episode..

Repeat this several episodes to get the optimal policy.

The SARSA algm is:-

1. Initialize the Q fn Q(s,a) with random values.

2. For each episode:

   1. Initialize state s

   2. Extract a policy from Q(s,a), select an action a to perform in state s. [Use epsilon greedy policy]

   3. For each step in the episode:

      1. Perform the action 'a', move to next state 's'', get the reward 'r'.

      2. In state 's'', select action a', using epsilon-greedy policy.

      3. Update the Q value of Q(s,a) to

      $$Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$$

      4. Update $s = s'$, $a = a'$ (update the next state-action as the current state-action

      5. If s is not a terminal state, repeat steps 1 to 5.

To find optimal policy Using SARSA in the FLE

```
# import the libraries
import gym
import random
# create the envt
env = gym.make(                    )
# define a dictionary for the Q table
# initialize the Q value of all (s,a) pairs
# to 0.0
Q = {}
for s in range(env.observation_space.n):
    for a in range(env.action_space.n):
        Q[(s,a)] = 0.0
# define the epsilon-greedy policy. we
# generate a random number from the Uniform
# distbn and if the random no. is less than
# epsilon, we select a random action,
# else, we select the best action
def epsilon-greedy(state, epsilon):
    if random.uniform(0,1) < epsilon:
        return env.action_space.sample()
    else
        return max(list(range(env.action_space.n
        )), key = lambda x: Q[(state, x)]
```

P8(

```
# Initialize α, γ, epsilon
alpha = 0.85
gamma = 0.90
epsilon = 0.8
# set the no. of opsiodes and the no. of steps in each
# episode
num_episodes = 50000
num_timesteps = 1000
# compute the policy
# for each episode
    for i in range(num_episodes):
# Initialize the state by resetting the env
        s = env.reset()
# select the action using the epsilon-greedy
                                                policy
        a = epsilon_greedy(s, epsilon)
# for each step in the episode:
        for t in range(num_timesteps):
# Perform the selected action and get the next
# state and reward
            s_, r, done, _ = env.step(a)
# select the action in next state using
# epsilon_greedy policy
            a_ = epsilon_greedy(s_, epsilon)
```

update the Q value of (s,a) using SARSA rule

Q[(s,a)] += alpha * (r + gamma *
Q[(s_, a_)] - Q[(s,a)])

update next state-action as current

s = s_
a = a_

if current state is terminal, then break

if done:
    break

∴ from the Q-value after all episodes, we can extract the optimal policy by selecting the action that has the max. Q value in each state.

Off-policy TD control - Q-learning

This is a very popular off-policy algm for finding the optimal policy.

It uses 2 different policies, one policy for behaving in the envt (selecting an action in the envt) and the other for finding the optimal policy.

In SARSA, we select an action 'a' in state s using the epsilon-greedy policy, move to the next state s', update the Q value using the update rule as:

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma Q(s',a') - Q(s,a)\right)$$

To find the Q-value of next state-action pair, Q(s',a'), we need to select an action in the next state using the same epsilon-greedy policy, and update the Q(s,a).

But in Q-learning:- we use 2 different policies,

To select an action in the current state of the envt we use an epsilon-greedy policy

But to update the Q(s',a') we use a greedy policy.

The update rule of SARSA is:-

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma Q(s',a') - Q(s,a)\right)$$

In Q-learning, to find $Q(s',a')$ we use greedy policy, ie, we choose the action that gives max. Q-value.

$\therefore$ The update rule of Q-learning is:

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

we choose the action that gives the max Q-value

Ex: Q-learning in FZE.-

	1	2	3	4
1	S	F	F	F
2	F	H	F	H
3	F	F	F	H
4	H	F	F	G

initial state

Randomly initialize the state

s	a	value
(1,1)	up	0.5
⋮	⋮	⋮
(3,2)	up	0.1
(3,2)	down	0.8
(3,3)	left	0.5
(3,4)	right	0.6
(4,4)	right	0.5

→ Initial state is (3,2)

→ To select an action in state (3,2), we use epsilon-greedy policy; [with prob ε, we choose a random action and with prob $1-\varepsilon$ we choose the best action' that has the max. Q-value]

→ Assume, we use prob $1-\varepsilon$ and select the best action

∴ we choose 'down' in (3,2)

∴ next state $s' = (4,2)$; $r=0$; $\alpha = 0.1$;
$\gamma = 1$.

→ update the Q-value? using ~~the~~ Q-learning rule:

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

$$\therefore Q((3,2),\text{down}) = Q((3,2),\text{down}) + \alpha\left(r + \gamma \max_{a'} Q((4,2),a') - Q((3,2),\text{down})\right)$$

$$= 0.8 + 0.1\left(0 + 1 \times \max_{a'} Q((4,2),a') - 0.8\right)$$

~~Q((3,2), d~~

To find the Q-value (s', a'), we use the greedy policy. ∴ choose $a' = \text{right}$; $Q(s',a') = 0.8$

∴ our update rule becomes:

$$Q((3,2), down) = 0.8 + 0.1[0 + 1 \times Q((4,2), right) - 0.8)$$

$$= 0.8 + 0.1(0 + 1 \times 0.8 - 0.8)$$

$$= 0.8$$

Illy we update the Q-fn in each step of an episode.

we extract a new policy at the end of an episode using the updated Q-fn. & use this policy.

After several episodes, we will have the optimal Q fn

∴ The Q-learning algom is:-

1. Initialize a Q-fn $Q(s,a)$ with random values.
2. for each episode
   1. Initialize state s
   2. for each step in the episode:
      1. Extract a policy from $Q(s,a)$ and select an action a to perform in state s, using epsilon-greedy policy

2. Perform the action a^*, move to s', get the reward r

3. Update the Q-value as

$$Q[s,a] = Q[s,a] + \alpha\left(r + \gamma \max_{a'} Q[s',a'] - Q[s,a]\right)$$

using greedy policy

4. Update $s = s'$ (update next state as current state)

5. If this state s is not a terminal state; repeat steps 1 to 5.

Computing optimal policy using Q-learning for FZE

```python
# import the libraries
import gym
import numpy as np
import random
# Create the FZE
env = gym.make('                ')
# Initialize the dict for Q-value of (s,a) to
# zeros
```

```
Q = {}
for s in range(env.observation_space.n):
    for a in range(env.action_space.n):
        Q[(s,a)] = 0.0

# define the epsilon-greedy policy

def epsilon_greedy(state, epsilon):
    if random.uniform(0,1) < epsilon:
        return env.action_space.sample()
    else:
        return max(list(range(env.action_space.n)),
                   key = lambda x: Q[(state,x)])

# initialize γ, α, and epsilon value

alpha = 0.85
gamma = 0.90
epsilon = 0.8
# set the no. of episodes and the no. of steps
# in each episode

num_episodes = 50000
num_timesteps = 1000
```

```
# Compute the policy

for i in range(num_episodes):
# initialize the env
    s = env.reset()
    for t in range(num_timesteps):
# Select the action using the epsilon-greedy policy
        a = epsilon_greedy(s, epsilon)

# Perform the action, get s' and r
        s_, r, done, _ = env.step(a)

# update Q(s,a) using greedy policy for Q(s',a')
# find the next action a' using greedy policy
        a_ = np.argmax([Q[(s_, a)]
              for a in range(env.action_space.n)])

# update Q(s,a)
        Q[(s,a)] += alpha * (r + gamma *
                               Q[(s_, a)] -
                                Q[(s,a)])

# update s = s'
        s = s_
        if done:
            break
```

After all itns. we will have the optimal Q fns. Using this we can extract the optimal policy, by selecting the action that has the max Q-value in each state.

Difference betn Q-learning and SARSA.

SARSA	Q-learning
1. It is an on-policy algom	1. It is an off-policy algm
2. To select an action in the envt, and to find the Q-value of next state action $Q(s',a')$ pair, it uses a single epsilon-greedy policy	2. To select an action in the envt it uses epsilon-greedy policy; and to find the $Q(s',a')$ pair, it uses a greedy policy
3. The update rule of SARSA is $Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$	3. The update rule of Q-learning is $Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$

Summary of DP, MC and TD methods.

The various RL algoms are:-

DP — value and policy iteration

MC methods

TD learning methods
- TD prediction
- TD control
  - SARSA
  - Q-learning

DP is a model-based method; To find the optimal policy, it uses the model dynamics of the envt. when we don't have the model dynamics, we cannot apply the DP method.

MC is a model-free method. So, it finds the optimal policy without using the model dynamics of the envt. But MC is applied only to episodic tasks & not to continuous tasks.

TD learning combines the advantages of DP and MC methods. It updates the value & Q fns using bootstrapping (doesn't wait till the end of an episode). It doesn't use the model dynamics (transition probs of states) to update the value & Q fns.