Module 3

Temporal Difference (TD) Learning

Slides prepared by Dr J Alamelu Mangai

PRESIDENCY UNIVERSITY

Private University Estd. in Karnataka State by Act No. 41 of 2013

Introduction

- TD is a popular model-free method
- It combines the advantages of DP and MC methods
- A recap of pros and cons of DP and MC methods

Dynamic Programming (DP)

Advantages :

- Uses Bellman equation to find the value of a state as

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

- We don't have to wait till the end of an episode to find the value of a state.
- We can estimate the value of a state, just based on the value of the next state – **boot strapping**

Disadvantages:

- DP can be used only when the model dynamics is known

Monte-Carlo (MC)

Advantages :

- It is a model free method.
- We don't require the model dynamics to estimate the value and Q functions.

Disadvantages:

- To find V(s) or Q(s,a), we need to wait till the end of the episode.
- If the episode is too long, it will cost us a lot of time.
- MC methods cannot be applied to continuous tasks/non-episodic tasks, without a terminal state.

Temporal Difference (TD) learning

- We use boot-strapping, and don't have to wait till the end of an episode to find V(s) or Q(s, a)
- Like MC it is a model-free method.
- Two categories of TD learning : TD-prediction and TD-control
- TD- prediction :
  - A policy is given as input and we try to predict the V(s) and Q(s,a) using this policy
  - Helps the agent to understand how good it is for the agent to be in each state, if it uses the given policy
  - The agent can estimate the expected return of each state, if it uses the given policy in that state.

- TD-control:
- We are not given any policy as input, but the goal is to find an optimal policy
- We initialize a random policy and we try to find the optimal policy iteratively.
- This optimal policy will give the maximum return

TD-Prediction

- A policy is given as input and we try to estimate the value function of each state V(s), using the given policy
- TD uses bootstrapping like DP, hence we don't have to wait till the end of an episode to find V(s)
- Like MC, it doesn't require the model dynamics to find V(s) and Q(s,a).
- The update rule of TD takes these advantages into account.
- In MC method, $V(s) \approx R(s)$
- Since a single return cannot approximate V(s) 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 uses bootstrapping and doesn't wait till the end of an episode, to find the value of a state as

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

- It doesn't use the model dynamics.
- In MC, since a single value V(s) cannot approximate the value of a state perfectly, we took the incremental mean

$$V(s) = V(s) + \alpha(R - V(s))$$

- In TD, we use the TD-learning update rule

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

Difference between MC and TD

Monte Carlo Method

$$V(s) = V(s) + \alpha(R - V(s))$$

full return

TD Learning

$$V(s) = V(s) + \alpha(\boxed{r + \gamma V(s')} - V(s))$$

Bootstrap estimate

Figure 5.1: A comparison between MC and TD learning

- $r + \gamma V(s')$ is an estimate of the value of a state – called – TD target.
- Hence $(r + \gamma V(s'))$ - V(s) is (target – predicted) – called the TD error.

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

Our TD learning update rule basically implies:

Value of a state = value of a state + learning rate (reward + discount factor(value of next state) - value of a state)

TD-Prediction using TD-learning

- A policy is taken as input, using the update rule of TD-learning, 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
- The update rule of TD-learning is

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

TD-Prediction in FZLE

- States are represented as numbers as the first state S is denoted by (1,1) and the second state F is denoted by (1,2) and so on to the last state G, which is denoted by (4,4).
- the goal of the agent is to reach the goal state G from the starting state S without visiting the hole states H
- Reward is 1, if the agent reaches the goal state, else reward is 0
- Actions : left, right, up, down

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

- Take a policy as input and evaluate V(s) using this policy
- Assume input policy is

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

- Find V(s) for this input policy using TD-learning

- Initialize the value of all states to some random values.

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

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

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

- Step 1: Current state s = (1,1) a = right as per the input policy

  next state s' = (1,2)

  update V((1,1)) using TD –learning update rule as

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

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

Substituting the value of state $V(s)$ with $V(1,1)$ and the next state $V(s')$ with $V(1,2)$ in the preceding equation, we can write:

$$V(1,1) = V(1,1) + \alpha(r + \gamma V(1,2) - V(1,1))$$

Substituting the reward $r = 0$, the learning rate $\alpha = 0.1$, and the discount factor $\gamma = 1$, we can write:

$$V(1,1) = V(1,1) + 0.1(0 + 1 \times V(1,2) - V(1,1))$$

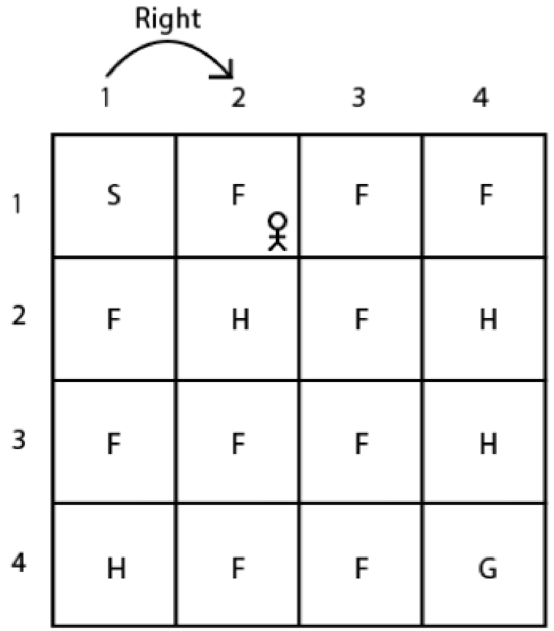
We can get the state values from the value table shown earlier. That is, from the preceding value table, we can observe that the value of state **(1,1)** is 0.9 and the value of the next state **(1,2)** is 0.6. Substituting these values in the preceding equation, we can write:

$$V(1,1) = 0.9 + 0.1(0 + 1 \times 0.6 - 0.9)$$

Thus, the value of state **(1,1)** becomes:

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

- Updated value table after step1 is



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

Figure 5.4: The value of state (1,1) is updated

- Step 2 : s = (1,2) a = right s' = (1,3) r = 0 α = 0.1 and γ = 1

  current value of V((1,2)) = 0.6

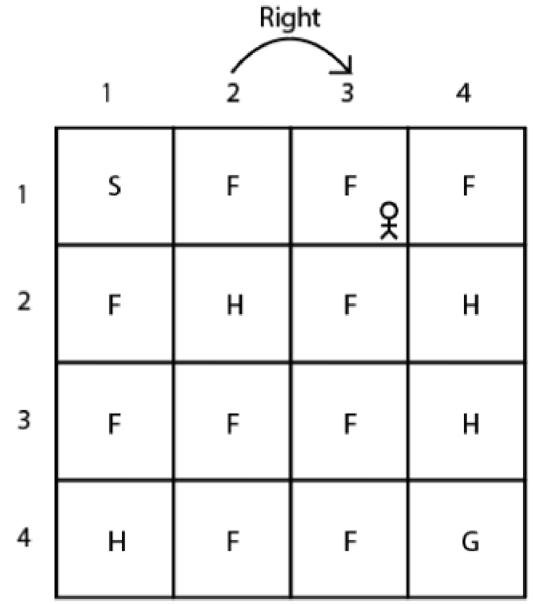
  update V((1,2)) using TD-learning

$$V(1,2) = V(1,2) + \alpha(r + \gamma V(1,3) - V(1,2))$$

$$V(1,2) = V(1,2) + 0.1(0 + 1 \times V(1,3) - V(1,2))$$

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

Value table after step 2 is



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

Figure 5.5: The value of state (1,2) is updated

- Step 3 : s = (1,3) a = left r = 0 s’ = (1,2) current V(1,3) = 0.8 and V(1,2) = 0.62

  Update V(1,3)

$$V(1,3) = V(1,3) + \alpha\big(r + \gamma V(1,2) - V(1,3)\big)$$

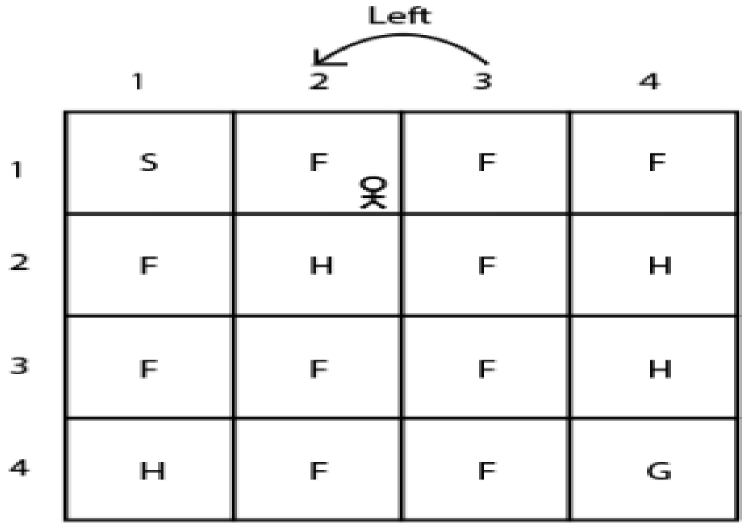
$$V(1,3) = V(1,3) + 0.1(0 + 1 \times V(1,2) - V(1,3))$$

$$V(1,3) = 0.8 + 0.1(0 + 1 \times 0.62 - 0.8)$$

Thus, the value of state **(1,3)** becomes:

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

So, we update the value of state **(1,3)** to **0.782** in the value table, as *Figure 5.6* shows:



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

- Thus, in this way, we compute the value of every state using the given policy.
- However, computing the value of the state just for one episode will not be accurate.
- So, we repeat these steps for several episodes and compute the accurate estimates of the state value (the value function).

TD-Prediction Algorithm

The TD prediction algorithm is given as follows:

1. Initialize a value function $V(s)$ with random values. A policy π is given.
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 given policy π, get the reward r, and move to the next state s'
        2. Update the value of the state to $V(s) = V(s) + \alpha(r + \gamma V(s') - V(s))$
        3. Update $s = s'$ (this step implies we are changing the next state s' to the current state s)
        4. If s is not the terminal state, repeat *steps 1* to *4*

Using TD-Prediction in FZLE

- For an input random policy π, predict the value of the states V(s) using TD prediction

```
import gymnasium as gym

import pandas as pd


#create the envt

env = gym.make("FrozenLake-v1", render_mode = "human")

env.reset()

env.render()
```

```
#define a random input policy
def random_policy():
    return env.action_space.sample()
#initialize the value of all states to zeros
V = {}
for s in range(env.observation_space.n):
    V[s] = 0.0
#initialize the parameters
alpha = 0.85
gamma = 0.90
num_eps = 50
num_steps = 10
```

#generating episodes

```
for i in range(num_eps):
    s = env.reset()
    s = s[0]
    for t in range(num_steps):
        a = random_policy()
        s_, r, done, _, _ = env.step(a)
        #use TD-update rule to update the value of a state
        V[s] += alpha * (r + gamma * V[s_] - V[s])
        s = s_
        if done:
          break
```

#convert the dictionary to a data frame

```
df = pd.DataFrame(list(V.items()), columns = ['state', 'value'])
print(df)
```

Output :

- Value of state 14 is maximum
- Value of all terminal states(hole state and goal state) are zeroes

0 S	1 F	2 F	3 F
4 F	5 H	6 F	7 H
8 F	9 F	10 F	11 H
12 H	13 F	14 F	15 G

Figure 5.7: States encoded as numbers

	state	value
0	0	0.1241807
1	1	0.0024911
2	2	0.0001897
3	3	0.0000000
4	4	0.0242708
5	5	0.0000000
6	6	0.0008208
7	7	0.0000000
8	8	0.1605379
9	9	0.0230677
10	10	0.0035581
11	11	0.0000000
12	12	0.0000000
13	13	0.4063436
14	14	0.8770302
15	15	0.0000000

Figure 5.8: Value table

Note that since we have initialized a random policy, We might get varying results every time we run the previous code

TD-Control

- Start with a random policy and find an optimal policy iteratively
- Types – on-policy and off-policy TD-control
- On-policy control :
  - the agent behaves using one policy and tries to improve the same policy.
  - That is, in the on-policy method, we generate episodes using one policy and improve the same policy iteratively to find the optimal policy
- Off-policy control:
  - the agent behaves using one policy and tries to improve a different policy.
  - That is, in the off-policy method, we generate episodes using one policy and we try to improve a different policy iteratively to find the optimal policy.

On-policy TD-Control - SARSA

- **SARSA** – **s**tate-**a**ction-**r**eward-**s**tate-**a**ction
- in TD control our goal is to find the optimal policy.
- how can we extract a policy?
  - We can extract the policy from the Q function.
  - once we have the Q function then we can extract policy by selecting the action in each state that has the maximum Q value.
- how can we compute the Q function in TD learning?
  - Recall, in TD learning, the value function is computed as:

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

  - We can just rewrite this update rule in terms of the Q function as:

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

- we compute the Q function using the preceding TD learning update rule, and then we extract a policy from them.
- the preceding update rule is also known as **the SARSA update rule**
- In the prediction method, we were given a policy as input, so we acted in the environment using that policy and computed the value function.
- But here, we don't have a policy as input. So how can we act in the environment?

- So, first we initialize the Q function with random values or with zeros.
- Then we extract a policy from this randomly initialized Q function and act in the environment.
- Our initial policy will definitely not be optimal as it is extracted from the randomly initialized Q function, but on every episode, we will update the Q function (Q values).
- So, on every episode, we can use the updated Q function to extract a new policy.
- Thus, we will obtain the optimal policy after a series of episodes.

- One important point we need to note is that in the SARSA method, instead of making our policy act greedily, we use the epsilon-greedy policy.
- In a greedy policy, we always select the action that has the maximum Q value.
- But, with the epsilon-greedy policy we select a random action with probability epsilon, and we select the best action (the action with the maximum Q value) with probability of 1-epsilon.

$$
\text{Action at time}(t) \begin{cases} \max Q_t(a) & \text{with probability } 1-\epsilon \\ \text{any action } (a) & \text{with probability } \epsilon \end{cases}
$$

```
p = random()

if p < ε:
    pull random action
else:
    pull current-best action
```

SARSA in the FZLE

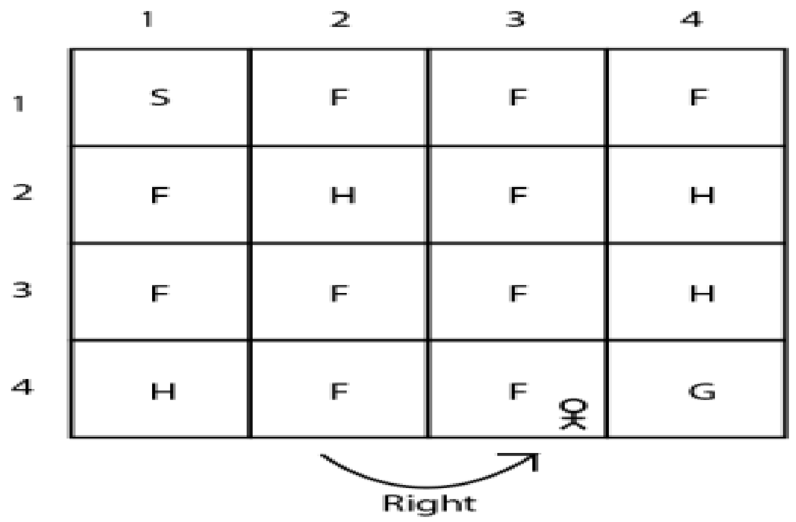
- Initialize the Q function with random values

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

State	Action	Value
(1,1)	Up	0.5
⋮	⋮	⋮
(4,2)	Up	0.3
(4,2)	Down	0.5
(4,2)	Left	0.1
(4,2)	Right	0.8
⋮	⋮	⋮
(4,4)	Right	0.5

Figure 5.9: The Frozen Lake environment and Q table with random values

- Assume we are in state (4,2)
  - **Select an action using epsilon greedy policy**.
  - With probability epsilon, we select a random action and with probability 1-epsilon we select the best action (the action that has the maximum Q value).
  - Suppose we use a probability 1-epsilon and select the best action.
  - So, in state (4,2), we move right as it has the highest Q value compared to the other actions.
  - so, we perform the right action in state (4,2) and move to the next state (4,3) as shown in the fig



State	Action	Value
(1,1)	Up	0.5
⋮	⋮	⋮
(4,2)	Up	0.3
(4,2)	Down	0.5
(4,2)	Left	0.1
$\boxed{(4,2)}$	Right	0.8
⋮	⋮	⋮
(4,4)	Right	0.5

Figure 5.11: We perform the action with the maximum Q value in state (4,2)

- we moved right in state (4,2) to the next state (4,3) and received a reward r of 0, next state s' = (4,3)
- Assume learning rate α at 0.1, and the discount factor γ at 1
- Update the Q((4,2), right), using **SARSA update rule**

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

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

- Substituting α and γ

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

- Substituting Q((4,2), right) = 0.8, from the previous Q –table :

$$Q((4,2),\text{right}) = 0.8 + 0.1\left(0 + 1 \times Q\left((4,3),a'\right) - 0.8\right)$$

- **what about the term $Q((4, 3), a')$?** – Q-value of the next state-action pair?
- We need to choose the action a’ in the next state using epsilon-greedy policy
- we select a random action with a probability of epsilon, or we select the best action that has the maximum Q value with a probability of 1-epsilon.
- Suppose we use probability epsilon and select the random action. In state (4,3), we select the right action randomly, as Figure 5.12 shows.
- As you can see, although the right action does not have the maximum Q value, we selected it randomly with probability epsilon

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

State	Action	Value
(1,1)	Up	0.5
⋮	⋮	⋮
(4,2)	Left	0.1
(4,2)	Right	0.8
(4,3)	Up	0.1
(4,3)	Down	0.3
(4,3)	Left	1.0
(4,3)	Right	0.9
⋮	⋮	⋮
(4,4)	Right	0.5

Figure 5.12: We perform a random action in state (4,3)

- Use the update rule now :

$$Q((4,2),\text{right}) = 0.8 + 0.1\big(0 + 1 \times Q\big((4,3),\text{right}\big) - 0.8\big)$$

- substituting the value of Q((4,3), right) with 0.9

$$Q((4,2),\text{right}) = 0.8 + 0.1(0 + 1(0.9) - 0.8)$$

- Hence the updated Q-value is

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

- in this way, we update the Q function by updating the Q value of the state-action pair in each step of the episode.
- After completing an episode, we extract a new policy from the updated Q function and uses this new policy to act in the environment. (Remember that our policy is always an epsilon-greedy policy).
- We repeat this steps for several episodes to find the optimal policy.

SARSA algorithm

The SARSA algorithm is given as follows:

1. Initialize a Q function $Q(s, a)$ with random values
2. For each episode:
    1. Initialize state s
    2. Extract a policy from $Q(s, a)$ and select an action a to perform in state s
    3. For each step in the episode:
        1. Perform the action a and move to the next state s' and observe the reward r
        2. In state s', select the action a' using the epsilon-greedy policy
        3. Update the Q value to $Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$
        4. Update $s = s'$ and $a = a'$ (update the next state s'-action a' pair to the current state s-action a pair)
        5. If s is not a terminal state, repeat *steps 1 to 5*

Implement SARSA in FZLE

- Find an optimal policy for the FZLE using SARSA

```
#import libraries
import gymnasium as gym
import random

 #create the envt
env = gym.make("FrozenLake-v1", render_mode = "human")
env.reset()
env.render()
```

```
#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 distribution of 0 to 1.

#If the random number 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)])
```

```
# initialize alpha, gamma and epsilon
alpha = 0.85
gamma = 0.90
epsilon = 0.8

#set the no. of episodes and the no. of steps in each episode
 num_eps = 500
num_steps = 100
```

```
#generate episodes
for i in range(num_eps):
    s = env.reset()
    s = s[0]
    a = epsilon_greedy(s,epsilon)
    for t in range(num_steps):
        s_, r, done, _, _ = env.step(a)
        a_ = epsilon_greedy(s_, epsilon)
        predict = Q[(s,a)]
        target = r + gamma * Q[(s_, a_)]
        Q[(s,a)] = Q[(s,a)] + alpha * (target - predict)
        s = s_
        a = a_
        if done:
            break
```

```
#convert the dictionary to a data frame
df = pd.DataFrame(list(Q.items()), columns = ['state-action',  'value'])
print(df)
```

- Convert the Q-table into a frame and print
- Note that on every iteration we update the Q function.
- After all the iterations, we will have the optimal Q function.
- Once we have the optimal Q function then we can extract the optimal policy by selecting the action that has the maximum Q value in each state.

Off-policy TD control – Q-learning

- Q learning is an off-policy algorithm,
- it uses two different policies, one policy for behaving in the environment (selecting an action in the environment) and the other for finding the optimal policy.
- In SARSA, we use epsilon-greedy policy to choose an action in the current state s, then used the SARSA update rule for Q(s,a)
- Also in the next state-action pair Q(s',a'), we used the same epsilon greedy policy to select an action in s' and then updated Q(s',a')

- unlike SARSA, in Q learning, we use two different policies.
- One is the epsilon-greedy policy and the other is a greedy policy.
- **To select an action in the environment we use an epsilon-greedy policy, but while updating the Q value of the next state-action pair Q(s’, a’)we use a greedy policy**

- we select an action a in state s using the epsilon-greedy policy and move to the next state s' and update the Q value using the update rule shown below:

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

- To find Q(s', a') we select an action using the greedy-policy, the policy that has the maximum Q-value in s'
- So the update rule of Q learning is given as:

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

Q-learning in FZLE

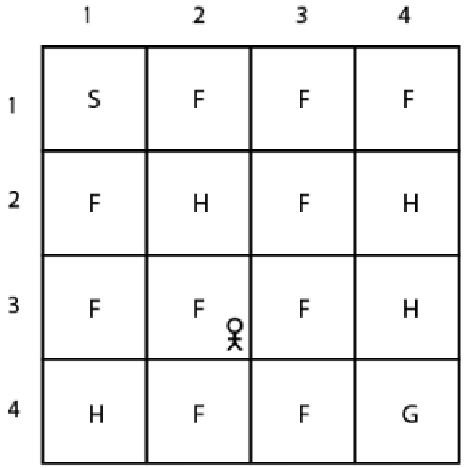
- Initialise the Q-table to random values

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

State	Action	Value
(1,1)	Up	0.5
⋮	⋮	⋮
(3,2)	Up	0.1
(3,2)	Down	0.8
(3,2)	Left	0.5
(3,2)	Right	0.6
⋮	⋮	⋮
(4,4)	Right	0.5

Figure 5.13: The Frozen Lake environment with a randomly initialized Q table

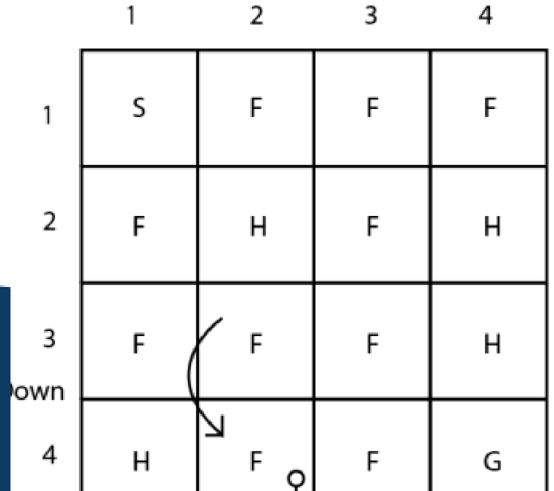
- Assume we are in state (3,2)
- Select an action using epsilon – greedy policy. Assume with prob of 1-epsilon we choose the best action ‘down’



State	Action	Value
(1,1)	Up	0.5
⋮	⋮	⋮
(3,2)	Up	0.1
$\boxed{(3,2)}$	$\boxed{\text{Down}}$	$\boxed{0.8}$
(3,2)	Left	0.5
(3,2)	Right	0.6
⋮	⋮	⋮
(4,4)	Right	0.5

Figure 5.14: We perform the action with the maximum Q value in state (3,2)

- we perform the down action in state (3,2) and move to the next state (4,2), with r =0. Assume e learning rate α as 0.1, and the discount factor γ as 1



State	Action	Value
(1,1)	Up	0.5
⋮	⋮	⋮
(3,2)	Up	0.1
$\boxed{(3,2)}$	$\boxed{\text{Down}}$	$\boxed{0.8}$
(3,2)	Left	0.5
(3,2)	Right	0.6
⋮	⋮	⋮

- Update Q((3,2),down) using Q-learning update rule as

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

Substituting the state-action pair $Q(s,a)$ with $Q((3,2),$ down) and the next state s' with **(4,2)** in the preceding equation, we can write:

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

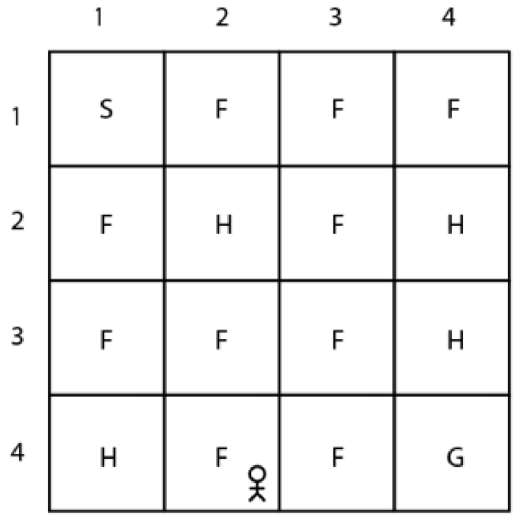
Substituting the reward, $r = 0$, the learning rate $\alpha = 0.1$, and the discount factor $\gamma = 1$, we can write:

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

From the previous Q table, we can observe that the Q value of $Q((3,2),$ down) is **0.8**.

$$Q((3{,}2),\text{down}) = 0.8 + 0.1\left(0 + 1 \times \max_{a'} Q((4,2),a') - 0.8\right)$$

- Use greedy-policy to select a' in s'.
- the right action has the maximum Q value in state (4,2).
- So, we select the right action and update the Q value of the next state-action pair:



State	Action	Value
(1,1)	Up	0.5
⋮	⋮	⋮
(4,2)	Up	0.3
(4,2)	Down	0.5
(4,2)	Left	0.1
(4,2)	Right	0.8
⋮	⋮	⋮
(4,4)	Right	0.5

Figure 5.16: We perform the action with the maximum Q value in state (4,2)

Thus, now our update rule becomes:

$$Q((3,2),\text{down}) = 0.8 + 0.1\big(0 + 1 \times Q\big((4,2),\text{right}\big) - 0.8\big)$$

From the previous Q table, we can observe that the Q value of Q((4,2), right) is **0.8**. Thus, substituting the value of Q((4,2), right) with **0.8**, we can rewrite the above equation as:

$$Q((3,2),\text{down}) = 0.8 + 0.1(0 + 1 \times 0.8 - 0.8)$$

Thus, our Q value becomes:

$$Q((3,2),\text{down} = 0.8$$

Similarly, we update the Q value for all state-action pairs.

- in this way, we update the Q function by updating the Q value of the state-action pair in each step of the episode.
- We will extract a new policy from the updated Q function on every step of the episode and uses this new policy.
- Remember that we select an action in the environment using epsilon-greedy policy but while updating Q value of the next state-action pair we use the greedy policy.
- After several episodes, we will have the optimal Q function

Q-learning algorithm

The Q learning algorithm is given as follows:

1. Initialize a Q function $Q(s, a)$ 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
        2. Perform the action a, move to the next state s', and observe the reward r
        3. Update the Q value as
           $Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s'a') - Q(s,a)\right)$
        4. Update $s = s'$ (update the next state s' to the current state s)
        5. If s is not a terminal state, repeat *steps 1* to *5*

Implement Q-learning in FZLE

```
#import libraries
import gymnasium as gym
import numpy as np
import random
 #create the environment
env = gym.make("FrozenLake-v1", render_mode = "human")
 #initialise the Q table
Q = {}
for s in range(env.observation_space.n):
    for a in range(env.action_space.n):
        Q[(s,a)] = 0.0
```

#define the function for 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)])
```

#initialise the parameters

```
alpha = 0.85
gamma = 0.90
epsilon = 0.8
 num_eps = 500
num_steps = 100
```

#generate episodes

```
for i in range(num_eps):
    s = env.reset()
    s = s[0]
    for t in range(num_steps):
        a = epsilon_greedy(s,epsilon)
        s_, r, done, _, _ = env.step(a)
        a_ = np.argmax([Q[(s,a)]  for a in range(env.action_space.n)])
        Q[(s,a)] += alpha * ( r + gamma * Q[(s_,a_)] - Q[(s,a)])
        s = s_
        if done:
            break
```

Differences between SARSA and Q-learning

S.No	SARSA	Q-learning
1	SARSA is an on-policy algorithm	Q learning is an off-policy algorithm
2	we use a single epsilon-greedy policy for selecting an action in the environment and also to compute the Q value of the next state-action pair	we use an epsilon-greedy policy for selecting an action in the environment, but to compute the Q value of next state-action pair we use a greedy policy
3	The SARSA update rule is : $Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$	The Q-learning update rule is : $Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$

Classical RL algorithms

- To learn an optimal policy :
  - DP – value and policy iteration, MC and TD
- **Comparison of DP, MC and TD methods:**
- Dynamic programming (DP), that is, the value and policy iteration methods,
  - is a model-based method, meaning that we compute the optimal policy using the model dynamics of the environment.
  - We cannot apply the DP method when we don't have the model dynamics of the environment.
- MC
  - is a model-free method, meaning that we compute the optimal policy without using the model dynamics of the environment.
  - But one problem we face with the MC method is that it is applicable only to episodic tasks and not to continuous tasks.
- TD :
  - a model-free method.
  - TD learning takes advantage of both DP by bootstrapping and the MC method by being model free.