Temporal Difference Learning

Story So Far...

- Dynamic Programming
  - a model-based method: need model dynamics
  - uses the Bellman equation to compute the value of a state
    - state value: sum of the immediate reward + discounted value of the next state
  - bootstrapping: we do not have to wait until the end of episode to calculate $V(s)$.

$$V(s) = \sum_{s'} P_{ss'}^{a} [R_{ss'}^{a} + \gamma V(s')]$$

- Monte-Carlo method
  - a model-free method: does not need model dynamics
  - to estimate state value or Q value, we need to wait until the end of episode
  - cannot be used for continuous tasks

Temporal Difference Learning

- Combines the benefits of DP and MC methods
  - Uses bootstrapping so that we do not have to wait until the end of the episode to compute state value and Q value.
  - A model-free method that does not require model dynamics

Temporal Difference Learning

TD Prediction

Estimating state value in MC

- In the MC method, the value of a state is estimated by running the episode and observing its return.

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

- In order to improve the accuracy of approximation, we generate multiple episodes and take the average return across the episodes.

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

- To calculate $V(s)$, we need to wait until the end of episode.
- It takes a lot of time if the episode is long.

TD Prediction

- The goal of prediction task is to calculate value of states given a policy.
- In TD learning, we use bootstrapping to calculate the value of a state

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

  - We estimate the value of a state using immediate reward r and the discounted value of the next state $\gamma V(s')$.

- We used bootstrapping in dynamic programming. The difference here is that we cannot just calculate $r + \gamma V(s')$ here because we do not know the model dynamics.
  - We do not know what is the reward and which next state the agent will move to.
- We need to actually perform an action in order to obtain $r + \gamma V(s')$.
- Just like Monte Carlo, we can get an approximate of $r + \gamma V(s')$ using a mean across multiple episodes.
- Here we use incremental mean instead of arithmetic mean.

TD Prediction

- Using incremental mean to estimate the state value in TD learning

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

- This equation is called the TD learning update rule.

- Comparison of incremental mean in 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

TD Prediction

- Terms in TD learning update rule
  - TD target: $r + \gamma V(s')$
  - TD error: $r + \gamma V(s') - V(s)$
  - learning rate: α

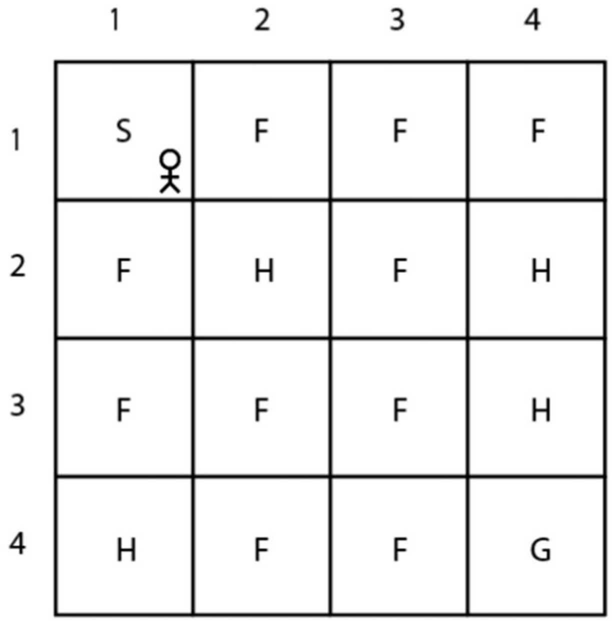
$$V(s) = V(s) + \underbrace{\alpha}_{\text{Learning rate}}(\underbrace{r + \gamma V(s') - V(s)}_{\text{TD error}})$$

- The meaning of TD learning update rule

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

TD Prediction Algorithm

- The Frozen Lake example
- Suppose we are given a policy



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

TD Prediction Algorithm

- For initialization, random values are assigned to states.

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

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

TD Prediction Algorithm

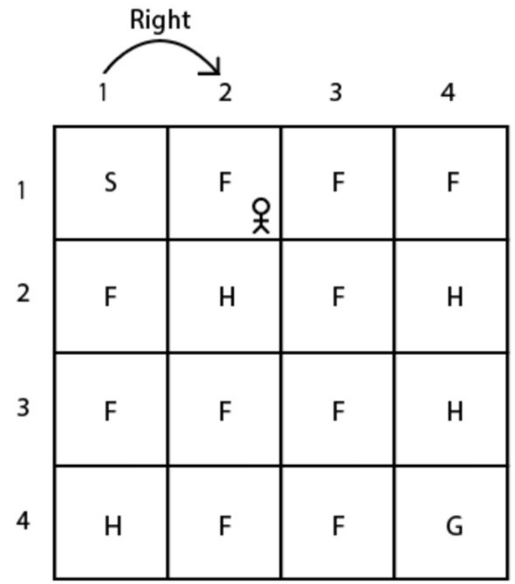
- We start the episode.
- The agent is in state (1,1), and moves right according to the given policy.
  - The agent ends up in state (1,2), and receives a reward of 0.
- Here, we assume the learning rate $\alpha = 0$ and discount factor $\gamma = 1.0$.

State	Value
(1,1)	0.9
(1,2)	0.6

- TD update rule

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

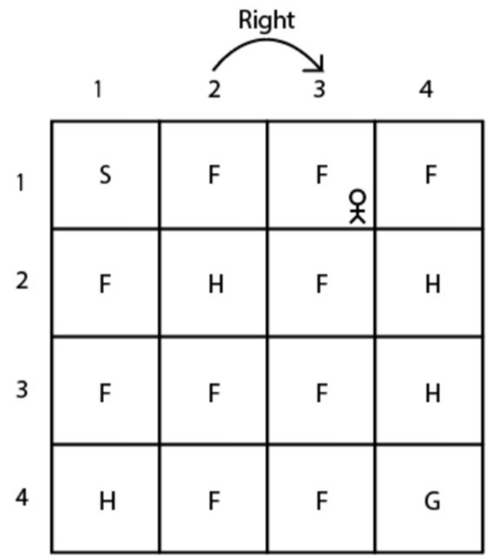
- In this case
  - $V(1,1) = V(1,1) + \alpha(r + \gamma V(1,2) - V(1,1))$
  - $V(1,1) = V(1,1) + 0.1(0 + 1 \times V(1,2) - V(1,1))$
  - $V(1,1) = 0.9 + 0.1(0 + 1 \times 0.6 - 0.9)$
  - $V(1,1) = 0.87$



TD Prediction Algorithm

- From state (1,2) the agent moves right.
  - The agent arrives at state (1,3) and receives a reward of 0.

- TD update
  - $V(s) = V(s) + \alpha(r + \gamma V(s') - V(s))$
  - $V(1{,}2) = V(1{,}2) + \alpha(r + \gamma V(1{,}3) - V(1{,}2))$
  - $V(1{,}2) = V(1{,}2) + 0.1(0 + 1 \times V(1{,}3) - V(1{,}2))$
  - $V(1{,}2) = 0.6 + 0.1(0 + 1 \times 0.8 - 0.6)$
  - $V(1{,}2) = 0.62$



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

TD Prediction Algorithm

- In this way, we compute the value of the states using the given policy.
- We repeat these steps for multiple episodes to improve the accuracy of approximation.
- TD prediction algorithm

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*

TD Prediction Algorithm: Implementation [ex007]

- import packages and create environment

```
import gym
import pandas as pd
from tqdm import tqdm
env = gym.make('FrozenLake-v1')
```

- initialization

```
V = {}
for s in range(env.observation_space.n):
    V[s] = 0.0

alpha = 0.85
gamma = 0.90
```

TD Prediction Algorithm: Implementation [ex007]

- TD learning

```
num_episodes = 500000
num_timesteps = 1000

for i in tqdm(range(num_episodes)):
    s = env.reset()
    for t in range(num_timesteps):
        a = random_policy()
        s_, r, done, _ = env.step(a)
        V[s] += alpha * (r + gamma * V[s_] - V[s])    # TD update rule
        s = s_
        if done:
            break
```

- Organize state values into a Panda data frame

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

TD Prediction Algorithm: Implementation [ex007]

- Result
  - State 14 has a high value because it is next to the goal state.
  - Value of the hole states and the goal state is 0 (not updated from initial value).

	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

Temporal Difference Learning

TD Control

TD Control

- On-policy TD control - SARSA
  - The agent behaves using one policy and tries to improve the same policy.

- Off-policy TD control - Q-learning
  - The agent behaves using one policy and tries to improve a different policy.

On-Policy TD Control: SARSA

- SARSA: "State-Action-Reward-State-Action"
- For control, we need to compute a Q function for state-action pairs

- TD update rule for the value function

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

- TD update rule for the Q function

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

On-Policy TD Control: SARSA

- First, we initialize the Q table to random values.

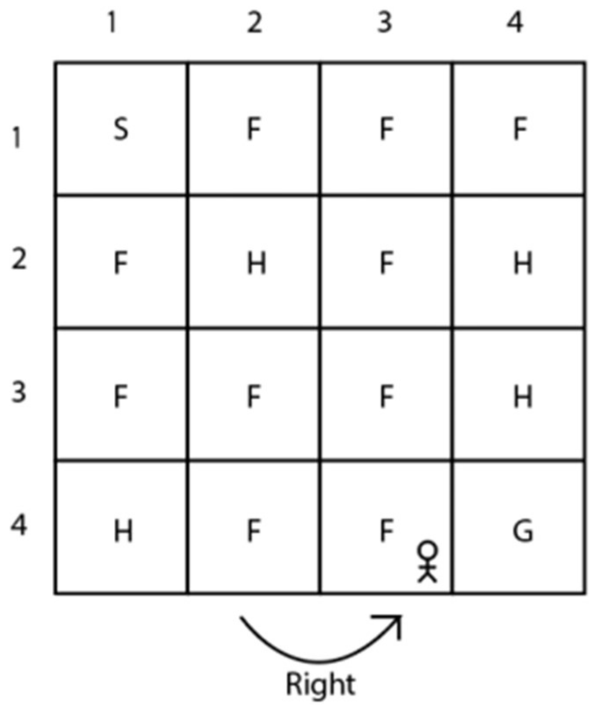
	1	2	3	4
1	S	F	F	F
2	F	H	F	H
3	F	F	F	H
4	H	F	F	G

State	Action	Value
(1,1)	Up	0.5
⋮	⋮	⋮
(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

On-Policy TD Control: SARSA

- When we run an episode, we use the epsilon-greedy policy based on the current Q function.

- For example, the agent decides to move right in state (4,2) according to the epsilon-greedy policy.



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

On-Policy TD Control: SARSA

- Now we update the Q value of state-action pair ((4,2), right).
  - $Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$
  - $Q\big((4,2), right\big) = Q\big((4,2), right\big) + \alpha\left(r + \gamma Q\big((4,3), a'\big) - Q\big((4,2), right\big)\right)$
  - $Q\big((4,2), right\big) = Q\big((4,2), right\big) + 0.1 \times \left(0 + 1 \times Q\big((4,3), a'\big) - Q\big((4,2), right\big)\right)$
  - $Q\big((4,2), right\big) = 0.8 + 0.1 \times \big(0 + 1 \times Q\big((4,3), a'\big) - 0.8\big)$

(4,2)	Up	0.3
(4,2)	Down	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

- How can we obtain $Q((4,3), a')$?
- After arriving at state (4,3), the agent again uses the epsilon-greedy policy to select an action. Suppose the agent selects "Right".
- Then, the update rule becomes:
  - $Q\big((4,2), right\big) = 0.8 + 0.1 \times \big(0 + 1 \times Q\big((4,3), right\big) - 0.8\big)$
  - $Q\big((4,2), right\big) = 0.8 + 0.1 \times (0 + 1 \times 0.9 - 0.8) = 0.81$

On-Policy TD Control: SARSA

- The SARSA algorithm

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*

SARSA: Implementation [ex008]

- Solving FrozenLake with SARSA (1/2)

```
import numpy as np
import gym
env = gym.make('FrozenLake-v1')
from tqdm import tqdm

def action_epsilon_greedy(q, s, epsilon=0.05):
    if np.random.rand() > epsilon:
        return np.argmax(q[s])
    return np.random.randint(4)

def evaluate_policy(q, n=500):
    acc_returns = 0
    for i in range(n):
        done = False
        s = env.reset()
        while not done:
            a = action_epsilon_greedy(q, s, epsilon=0.)
            s, reward, done, _ = env.step(a)
            acc_returns += reward
    return acc_returns / n
```

SARSA: Implementation [ex008]

- Solving FrozenLake with SARSA (2/2)

```
def sarsa(alpha=0.02, gamma=1., epsilon=0.05, q=None, env=env):

    if q is None:
        q = np.ones((16,4))

    nb_episodes = 200000
    steps = 2000
    progress = []
    for i in tqdm(range(nb_episodes)):
        done = False
        s = env.reset()
        a = action_epsilon_greedy(q, s, epsilon=epsilon)
        while not done:
            new_s, reward, done, _ = env.step(a)
            new_a = action_epsilon_greedy(q, new_s, epsilon=epsilon)
            q[s,a] = q[s,a] + alpha * (reward + gamma * q[new_s,new_a] - q[s,a])
            s = new_s
            a = new_a

        if i%steps == 0:
            progress.append(evaluate_policy(q, n=500))

    return q, progress

q, progress = sarsa(alpha=0.02, epsilon=0.05, gamma=0.999)
print(evaluate_policy(q, n=10000))
print(progress)
```

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

Off-Policy TD Control: Q-Learning

- In SARSA, the update rule for the Q function was:

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

- In order to compute the Q value of next state-action pair, $Q(s',a')$, we need to select an action.

- In SARSA, we select an action a' using the same epsilon-greedy policy and update the Q value.

Off-Policy TD Control: Q-Learning

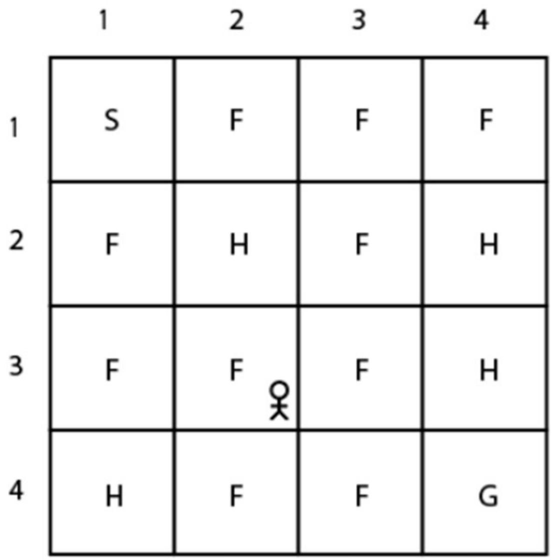
- In Q learning, we use two different policies.
  - To select an action a in state s, we use an epsilon-greedy policy.
  - When selecting an action a' for the next state s', we use a greedy policy.

- The update rule for the Q learning is:

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

Q-Learning: Example

- Suppose the agent is in state (3,2) in the Frozen Lake environment.



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

- Here, the agent selects an action based on the epsilon-greedy policy.
  - Random action with probability ϵ and best action with probability $1 - \epsilon$.

- Suppose the agent selects the best action "down", and moves to state (4,2).

Q-Learning: Example

- Now we need to update $Q((3,2), \mathrm{down})$.
- In Q-learning, the update rule is:

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

- Here, the update rule for $Q((3,2), \mathrm{down})$ will be:

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

Q-Learning: Example

- According to the current Q table, the best action in state (4,2) is "right".
- Assuming $\alpha = 0.1$ and $\gamma = 1.0$, the update equation will be:

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

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

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

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

	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

Q-Learning: The algorithm

- Algorithm

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*

Q-Learning: Implementation [ex009]

- Solving Frozen Lake with Q-learning (1/4)

```
import numpy as np
import gym
from tqdm import tqdm

# Q learning params
ALPHA = 0.1 # learning rate
GAMMA = 0.99 # reward discount
LEARNING_COUNT = 100000
TEST_COUNT = 1000

EPS = 0.1

TURN_LIMIT = 100

class Agent:
    def __init__(self, env):
        self.env = env
        self.episode_reward = 0.0
        self.q_val = np.zeros(16 * 4).reshape(16, 4).astype(np.float32)
```

Q-Learning: Implementation [ex009]

- Solving Frozen Lake with Q-learning (2/4)

```
def learn(self):
    # one episode learning
    state = self.env.reset()
    #self.env.render()

    for t in range(TURN_LIMIT):
        if np.random.rand() < EPS: # explore
            act = self.env.action_space.sample() # random
        else: # exploit
            act = np.argmax(self.q_val[state])
        next_state, reward, done, info = self.env.step(act)
        q_next_max = np.max(self.q_val[next_state])
        # Q <- Q + a(Q' - Q)
        # <=> Q <- (1-a)Q + a(Q')
        self.q_val[state][act] = (1 - ALPHA) * self.q_val[state][act]\
                                 + ALPHA * (reward + GAMMA * q_next_max)

        #self.env.render()
        if done:
            return reward
        else:
            state = next_state
    return 0.0 # over limit
```

Q-Learning: Implementation [ex009]

- Solving Frozen Lake with Q-learning (3/4)

```
def test(self):
    state = self.env.reset()
    for t in range(TURN_LIMIT):
        act = np.argmax(self.q_val[state])
        next_state, reward, done, info = self.env.step(act)
        if done:
            return reward
        else:
            state = next_state
    return 0.0 # over limit
```

Q-Learning: Implementation [ex009]

- Solving Frozen Lake with Q-learning (4/4)

```python
def main():
    env = gym.make("FrozenLake-v1")
    agent = Agent(env)

    print("###### LEARNING #####")
    reward_total = 0.0
    for i in tqdm(range(LEARNING_COUNT)):
        reward_total += agent.learn()
    print("episodes      : {}".format(LEARNING_COUNT))
    print("total reward  : {}".format(reward_total))
    print("average reward: {:.2f}".format(reward_total / LEARNING_COUNT))
    print("Q Value       :{}".format(agent.q_val))

    print("###### TEST #####")
    reward_total = 0.0
    for i in tqdm(range(TEST_COUNT)):
        reward_total += agent.test()
    print("episodes      : {}".format(TEST_COUNT))
    print("total reward  : {}".format(reward_total))
    print("average reward: {:.2f}".format(reward_total / TEST_COUNT))

if __name__ == "__main__":
    main()
```

Summary: SARSA vs. Q-Learning

- SARSA
  - on-policy algorithm
  - a single epsilon-greedy policy for selecting an action in the environment and computing Q-value of the state-action pair
  - update rule

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

- Q-Learning
  - off-policy algorithm
  - a epsilon-greedy policy for selection an action, and a greedy policy for updating the Q value
  - update rule

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

Summary So Far: DP, MC, and TD

- Dynamic Programming (DP)
  - methods: value iteration, policy iteration
  - model-based method: need model dynamics

- Monte Carlo (MC) method
  - on-policy MC, off-policy MC
  - model-free method
  - applicable to episodic tasks and not to continuous tasks

- Temporal Difference (TD) learning
  - SARSA (on-policy TD), Q-learning (off-policy TD)
  - model-free method
  - uses bootstrapping: applicable to continuous tasks as well as episodic tasks

End of Chapter

- Can you use the Temporal Difference learning method to train an agent to solve the Frozen Lake problem?

- Can you write [ex007], [ex008] and [ex009] yourself?

End of Class

Questions?

Email: jso1@sogang.ac.kr