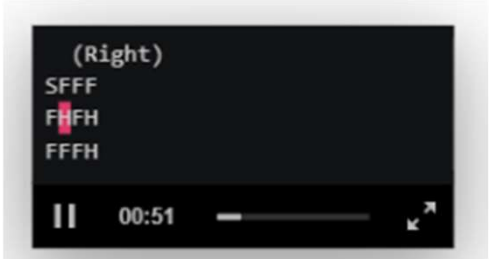
The OpenAI Gym Environment

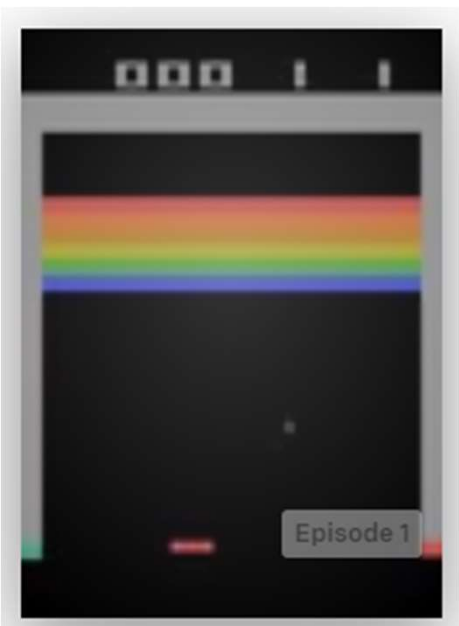
The Gym Environment

- A set of environments to test reinforcement learning algorithms
  - https://gym.openai.com/envs/

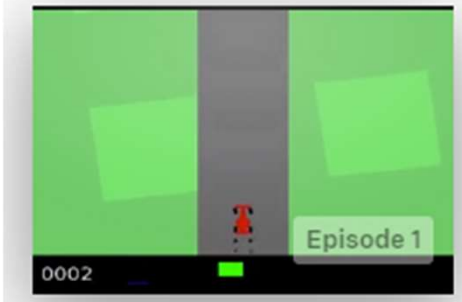


FrozenLake-v0
Find a safe path across a grid of ice and water tiles.

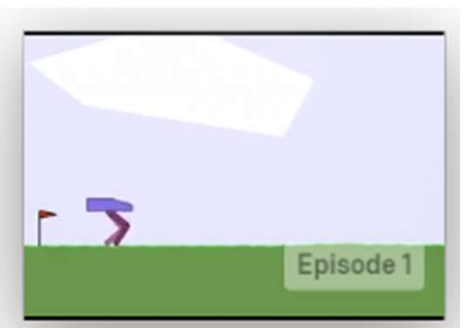
CartPole-v1
Balance a pole on a cart.



Breakout-ram-v0
Maximize score in the game Breakout, with RAM as input



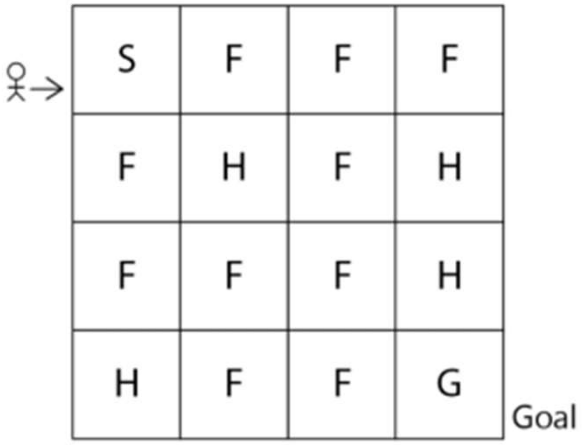
CarRacing-v0
Race a car around a track.



BipedalWalker-v2
Train a bipedal robot to walk.

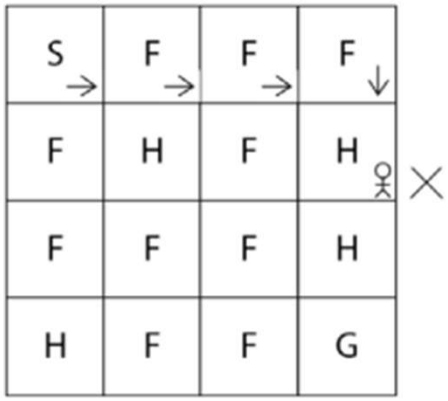
Frozen Lake

- A "hello world" environment in RL
- The goal of the agent is to start from the initial state S to the final state G.
- Four type of states
  - S: starting state
  - F: frozen state
  - H: hole state
  - G: goal state

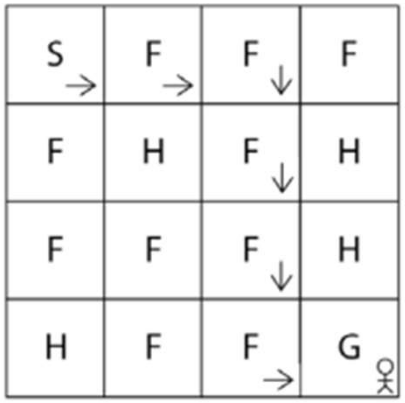


Frozen Lake

- If the agent comes to a hole state, the agent dies and the episode ends there.



- The agent must reach the goal state without visiting a hole state.

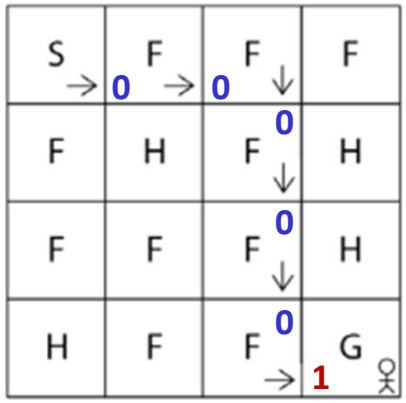


Frozen Lake: States, Actions, Rewards

- States
  - 16 states (current location of the agent)

- Actions
  - 4 actions: left, down, right, up

- Reward
  - The goal of training is to make the agent reach the goal state without visiting a hole state.
  - When the agent reaches the goal state, it receives a +1 reward.
  - In all other cases, the reward is 0.



Creating the Environment

- We can use the gym library to create the Frozen Lake environment.
  - gym.make("FrozenLake-v1"): create and return a Frozen Lake environment.
  - env.render(): visualize the environment

```
import gym

env = gym.make("FrozenLake-v1")
env.render()
```

[1] ✓ 0.3s

```
...

SFFF
FHFH
FFFH
HFFG
```

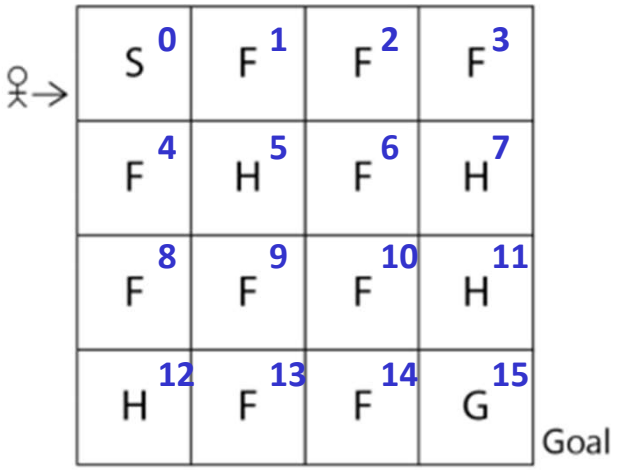
Exploring the Environment

- States
  - env.observation_space
  - shows that there are 16 discrete states

```
print(env.observation_space)

✓ 0.1s
```

```
Discrete(16)
```



- Actions
  - env.action_space
  - shows that there are 4 discrete actions
  - encodings: 0 (left), 1 (down), 2(right), 3 (up)

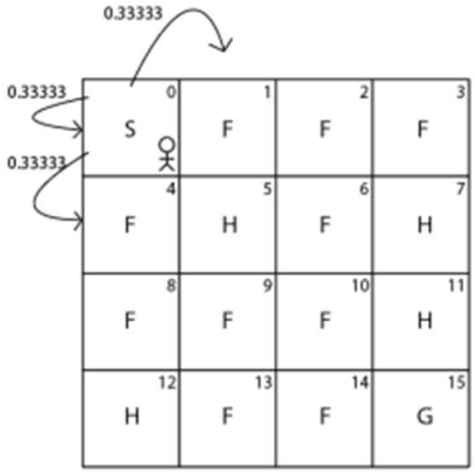
```
print(env.action_space)

✓ 0.2s
```

```
Discrete(4)
```

Exploring the Environment

- Transition Probability
  - FrozenLake-v1 is a stochastic environment.
  - For example, if the agent "moves right" from state 0, the agent does not always reach state 1. Instead, the agent's next state is based on a probability distribution.
    - The agent reaches state 1 with 1/3 probability.
    - The agent reaches state 4 with 1/3 probability.
    - The agent stays at state 0 with 1/3 probability.



Exploring the Environment

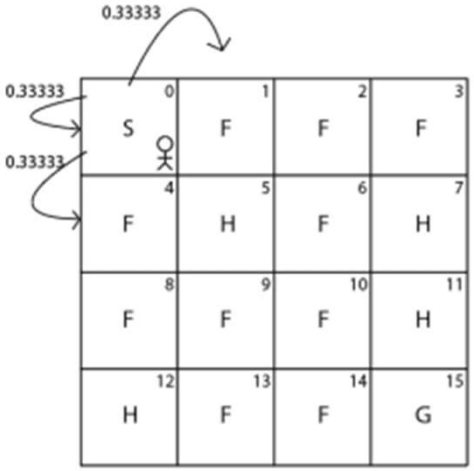
- Transition Probability
  - The variable P stores the transition probability and the reward for every state, action pair.
  - For example, P[0][2] is the transition probability and the reward for the case where the agent performs "move right" in state 0.

```
print(env.P[0][2])
```

✓ 0.6s

```
[(0.3333333333333333, 4, 0.0, False), (0.3333333333333333, 1, 0.0, False), (0.3333333333333333, 0, 0.0, False)]
```

  - The first element (0.3333, 4, 0.0, False) means:
    - Transition probability (0.3333)
    - The next state (4)
    - Reward (0)
    - Is terminal state? (False)



Exploring the Environment

- Transition probability for a hole state
  - The next state is always itself
  - Since episode ends as soon as the agent arrives here, the transition probability out of state 5 is not used.

```
print(env.P[5][0])

✓ 0.1s
```

```
[(1.0, 5, 0, True)]
```

- Transition probability for the goal state
  - Similar to a hole state
  - The reward +1 is acquired when the agent moves from another state to the goal state.

```
print(env.P[15][0])

✓ 0.5s
```

```
[(1.0, 15, 0, True)]
```

Generating an Episode

- Initializing the state
  - Calling env.reset() will put the agent back to the initial state.

```
state = env.reset()
env.render()
```

✓ 0.5s

```
SFFF
FHFH
FFFH
HFFG
```

Generating an Episode

- Performing an action
  - Calling env.step(action) will perform an action.

```
env.step(2)    # '2' is right

✓ 0.1s
```

```
(1, 0.0, False, {'prob': 0.3333333333333333})
```

  - The output of env.step(action) means:
    - The agent reached state 1
    - The agent received reward 0.0
    - State 1 is not a terminal state, so the episode continues
    - The probability of reaching this state as a result of the action was 0.33.

  - We can store this information as:

```
next_state, reward, done, info = env.step(2)

✓ 0.1s
```

Generating an Episode

- If we do not know which action to perform, we can sample a random action from the action space.
  - In the example below, the agent randomly chose to move down, but it ended up in state 1.

```
random_action = env.action_space.sample()
next_state, reward, done, info = env.step(random_action)
print(next_state, reward, done, info)
env.render()
```

✓ 0.1s

```
1 0.0 False {'prob': 0.3333333333333333}
  (Down)
SFF
FHFH
FFFH
HFFG
```

Generating an Episode

- We can generate an episode using a random policy.
  - The agent performs random action in every state it visits.
  - num_timesteps = 20 limits number of moves in the episode to 20.

```
1   import gym
2
3   env = gym.make("FrozenLake-v1")
4   state = env.reset()
5
6   print('Time step 0: ')
7   env.render()
8
9   num_timesteps = 20
10  for t in range(num_timesteps):
11      random_action = env.action_space.sample()
12
13      new_state, reward, done, info = env.step(random_action)
14      print('Time Step {}: '.format(t+1))
15
16      env.render()
17
18      if done:
19          break
```

Generating an Episode [ex001]

- Let's run multiple episode and see how many times the agent can get to the goal state with random policy!

```python
from tqdm import tqdm
import gym
env = gym.make("FrozenLake-v1")

num_episodes = 100
num_timesteps = 50
total_reward = 0
total_timestep = 0

for i in tqdm(range(num_episodes)):
    state = env.reset()

    for t in range(num_timesteps):
        random_action = env.action_space.sample()
        new_state, reward, done, info = env.step(random_action)
        total_reward += reward

        if done:
            break

    total_timestep += t

print("Number of successful episodes: %d / %d"%(total_reward, num_episodes))
print("Average number of timesteps per episode: %.2f"%(total_timestep/num_episodes))
```

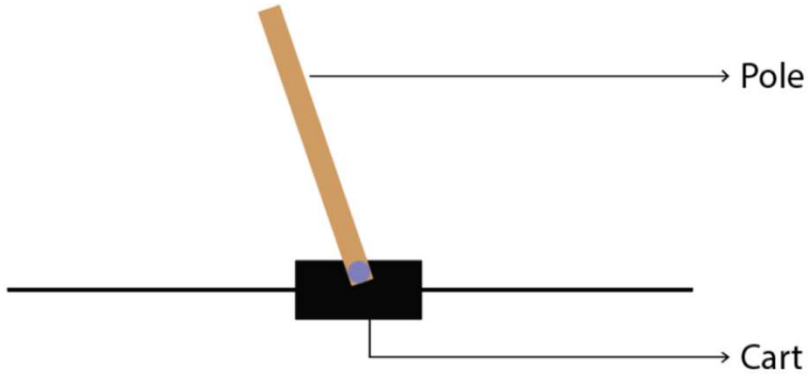
Cart-Pole Balancing

- The goal of our agent is to keep the pole standing straight up on the cart.
- The agent pushes the cart left and right to keep the pole standing.

```
import gym
env = gym.make("CartPole-v1")
env.render()
```

✓ 0.4s

  - (If you are working on a remote server via SSH, you would need display forwarding to render this environment.)



Cart-Pole Balancing

- State space
  - A state must capture the following:
    - cart position
    - cart velocity
    - pole angle
    - pole velocity at the tip
  - All these positions are continuous
    - The value of the cart position ranges from -4.8 to 4.8
    - The value of the cart velocity ranges from -Inf to Inf
    - The value of the pole angle ranges from -0.418 to 0.418 radians
    - The value of the pole velocity at the tip ranges from -Inf to Inf
  - In CartPole, the state space is an array of 4 continuous values.

```
print(env.observation_space)
```

✓ 0.4s

```
Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
```

Cart-Pole Balancing

- Initial state
  - env.reset() sets the four values to random numbers within the range.
  - (cart position, cart velocity, pole angle, pole velocity)

```
env.reset()
```

✓ 0.3s

```
array([ 0.00072647, -0.01207986, -0.02150378, -0.00809408])
```

- Action space
  - In CartPole, the agent only has two possible discrete actions

```
env.action_space
```

```
Discrete(2)
```

Number	Action
0	Push cart to the left
1	Push cart to the right

Cart-Pole Balancing

- Reward
  - The agent acquires +1 reward for every timestep that the pole remains upright. In other words, +1 reward is given in every timestep until the episode ends.

- Terminating condition
  - The pole is more than 15 degrees from vertical
  - The cart moves more than 2.4 units from the center

Cart-Pole Balancing

- CartPole with random policy

```python
import gym
env = gym.make('CartPole-v1')

num_episodes = 100
num_timesteps = 50

for i in range(num_episodes):
    r = 0   # return
    state = env.reset()

    for t in range(num_timesteps):
        random_action = env.action_space.sample()
        next_state, reward, done, info = env.step(random_action)
        r += reward

        if done:
            break

    if i%10 == 0:
        print('episode: {}, return: {}'.format(i, r))
```

The Bellman Equation and Dynamic Programming

The Bellman Equation

Bellman Equation

- Equation used to solve a Markov Decision Process
  - Solving MDP means we find the optimal policy

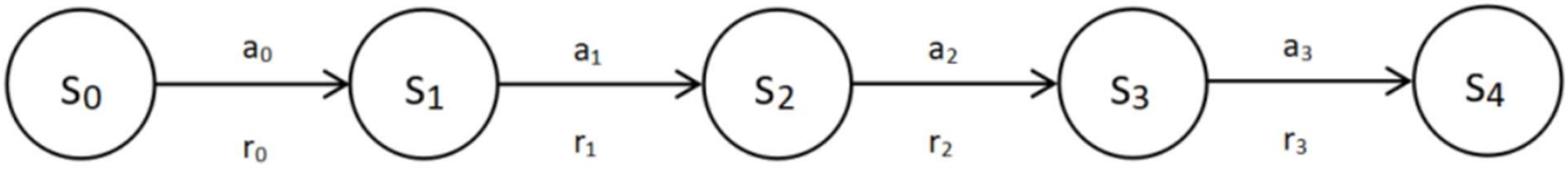
- Bellman equation for the value function
  - Value of a state can be obtained as a sum of the immediate reward and the discounted value of the next state.

$$V(s) = R(s, a, s') + \gamma V(s')$$

  - $R(s, a, s')$: immediate reward obtained by performing an action a in state s and moving to the next state s'
  - γ: the discount factor
  - $V(s')$: the value of the next state

Bellman Equation of the Value Function

- Suppose we generate the following trajectory τ using a policy π.

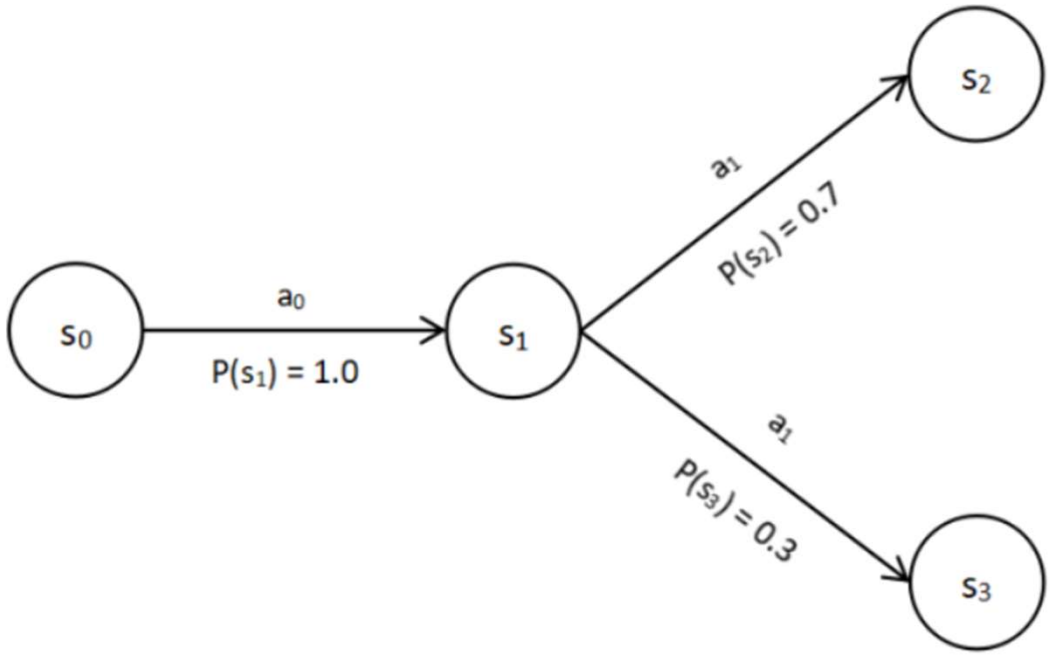


- To compute the value of s_2, the Bellman Equation says:
  - $V(s_2) = R(s_2, a_2, s_3) + \gamma V(s_3) = r_2 + \gamma V(s_3)$

- In general, the value function can be defined as:
  - $V^{\pi}(s) = R(s, a, s') + \gamma V^{\pi}(s')$
  - This implies that we are using policy π.
  - The right-hand side of the equation is called Bellman backup.

Bellman Equation of the Value Function

- In the previous slide, the environment was deterministic.
- In a stochastic environment, when we perform action a in state s, it is not guaranteed that the next state will be s'.



Bellman Equation of the Value Function

- For a stochastic environment, we need to modify the Bellman equation to use expectations (the weighted average).

  - $V^{\pi}(s) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')]$

  - $P(s'|s,a)$: probability of reaching s' by performing a in state s.
  - $[R(s,a,s') + \gamma V^{\pi}(s')]$: Bellman backup for state s'.

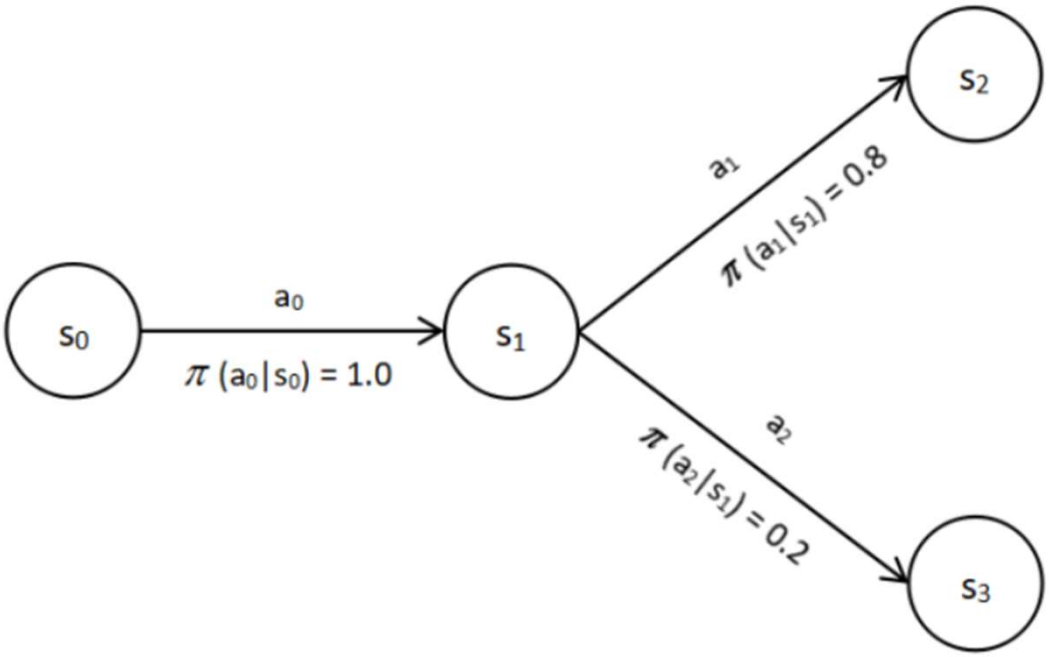
- In the example shown in the previous slide,

$$V(s_1) = P(s_2|s_1,a_1)[R(s_1,a_1,s_2) + V(s_2)] + P(s_3|s_1,a_1)[R(s_1,a_1,s_3) + V(s_3)]$$

$$V(s_1) = 0.70[R(s_1,a_1,s_2) + V(s_2)] + 0.30[R(s_1,a_1,s_3) + V(s_3)]$$

Bellman Equation of the Value Function

- What if the policy is also stochastic?
  - We also use the expectations for the policy similar to the environment.



Bellman Equation of the Value Function

- Value function with stochastic environment and stochastic policy
  - This called the Bellman expectation equation

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')]$$

  - We can also express this equation in expectation form.

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\substack{a \sim \pi \\ s' \sim P}} [R(s,a,s') + \gamma V^{\pi}(s')]$$

    - In the equation, $P = P(s'|s,a)$ and $\pi = \pi(a|s)$.

Bellman Equation of the Q function

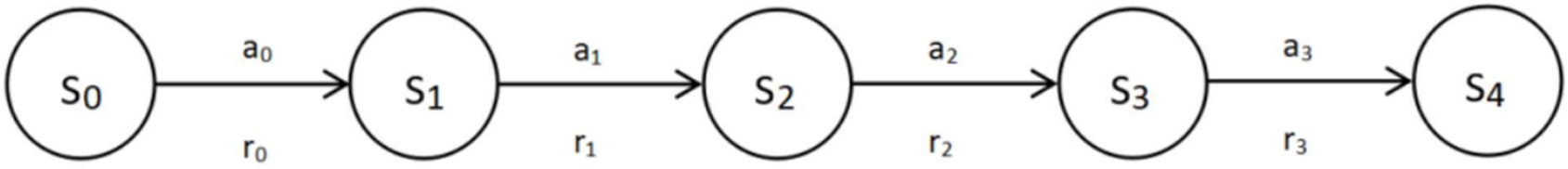
- Similar to the value function, the Q function can be also represented using the Bellman Equation.
  - Immediate reward + discounted Q value of the next state-action pair

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

  - $R(s,a,s')$: immediate reward obtained by performing an action a in state s and moving to the next state s'
  - γ: discount factor
  - $Q(s',a')$: the Q value of the next state-action pair

Bellman Equation of the Q Function

- Suppose we generate the following trajectory τ using a policy π.



- To compute the Q value of (s_2, a_2), the Bellman Equation says:
  - $Q(s_2, a_2) = R(s_2, a_2, s_3) + \gamma Q(s_3, a_3)$
- In general, the value function can be defined as:
  - $Q^{\pi}(s, a) = R(s, a, s') + \gamma Q^{\pi}(s', a')$
  - This implies that we are using policy π.
  - The right-hand side of the equation is called Bellman backup.

Bellman Equation of the Q Function

- For stochastic environment, we use the expectation to compute the Q function.

$$Q^{\pi}(s,a) = \sum_{s'} P(s'|s,a)\,[R(s,a,s') + \gamma Q^{\pi}(s',a')]$$

Bellman Equation of the Q Function

- Recall that when describing Bellman Equation of the value function, we used expectations to consider stochastic policy.

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')]$$

- However, in the Bellman Equation of the Q function, we don't need to use expectations for policies because the policy a is given with the state s.
- Instead, we need to use expectation on the next action a' if the policy is stochastic.

$$Q^{\pi}(s,a) = \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q^{\pi}(s',a') \right]$$

- We can also express the equation in the expectation form

$$Q^{\pi}(s,a) = \mathbb{E}_{s' \sim P}[R(s,a,s') + \gamma \mathbb{E}_{a' \sim \pi} Q^{\pi}(s',a')]$$

The Bellman Equation and Dynamic Programming

The Bellman Optimality Equation

Bellman Optimality Equation for Value Function

- The Bellman equation of the value function

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\substack{a \sim \pi \\ s' \sim P}} [R(s, a, s') + \gamma V^{\pi}(s')]$$

- The value function depends on the policy π.
- In other words, depending on the policy, many different value function exists.
- The optimal value function, $V^{*}(s)$, is the one that yields the maximum value compared to all other value functions.

Bellman Optimality Equation for Value Function

- What is the optimal value function in Bellman equation?
  - In state s, the agent performs an action based on the policy.
  - We let the agent choose an action that will maximize the Bellman backup.
  - Once it reaches the next state s', we'll assume that the agent will choose actions that results in the maximum value function of s'.

- The optimal value function

$$V^*(s) = \max_a \mathbb{E}_{s' \sim P}[R(s, a, s') + \gamma V^*(s')]$$

  - expectation over actions $a \sim \pi$ is removed because we are selecting a policy (that maximizes the Bellman backup) here.
  - To find $V^*(s)$, we can compute the state value for all possible actions and take the maximum.

Bellman Optimality Equation for Q Function

- The Bellman equation of the Q function

$$Q^{\pi}(s,a) = \mathbb{E}_{s' \sim P}[R(s,a,s') + \gamma \mathbb{E}_{a' \sim \pi} Q^{\pi}(s',a')]$$

- The optimal Bellman Q function

$$Q^{*}(s,a) = \mathbb{E}_{s' \sim P}\left[R(s,a,s') + \gamma \max_{a'} Q^{*}(s',a')\right]$$

  - $R(s,a,s')$: the immediate reward given for the state-action pair
  - $\gamma \max_{a'} Q*(s',a')$: maximum Q value at the next state s'

  - Since (s,a) is fixed, we cannot select an action in state s.
  - However, action at the next state s' is not decided.
  - We take the action that will gives us the maximum $Q^{*}(s',a')$

Bellman Optimality Equation for Q Function

- Summary

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

The Bellman Equation and Dynamic Programming

Relationship between Value and Q Function

The Value Function and the Q Function

- Previously, we learned the following equations

- Value of a state: $V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_0 = s]$

- Q value of a state-action pair: $Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a]$

- Optimal value function: $V^{*}(s) = \max_{\pi} V^{\pi}(s)$

- Optimal Q function: $Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a)$

The Value Function and the Q Function

- What is the relationship between the optimal value function and the optimal Q function?

$$V^*(s) = \max_{\pi} V^{\pi}(s) \qquad Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

  - The optimal value function: "the maximum expected return when we start from state s"
  - The optimal Q function: "the maximum expected return when we start from state s perfoming action a"

- We can say that the optimal value function is the maximum of optimal Q value over all possible actions.

$$V^*(s) = \max_{a} Q^*(s,a)$$

  - We can derive V from Q.

The Value Function and the Q Function

- The Bellman equation for the optimal Q function Q^* is:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

- Since $V^*(s) = \max_a Q^*(s,a)$, we can say that

$$Q^*(s,a) = \sum_{s'} P\,(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$$

  - We can derive Q from V.

- Also, from the previous equations, we can derive $V^*(s)$ as well.

  - $V^*(s) = \max_a Q^*(s,a) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$

The Bellman Equation and Dynamic Programming

Dynamic Programming - Value Iteration

Dynamic Programming

- Once we represent the value function and Q function using the Bellman equation, we can find the optimal policy using dynamic programming.

- Dynamic programming is a type of algorithm where we break a problem into sub-problems.
  - For each sub-problem, we compute and store the solution.
  - If the same sub-proglem occurs, we don't recompute; instead, we use the already computed solution.

- Here we look at two methods that use DP to find the optimal policy.
  - Value iteration
  - Policy iteration

- Note: dynamic programming is a model-based method.
  - Can be used when model dynamics is known

Value Iteration

- The value iteration algorithm
  1. Compute the (current) optimal value function iteratively.
  2. Extract the (current) optimal policy from the computed optimal value function

- To execute the method, we need a value table and a Q table.

State	Value
A	
B	
C	

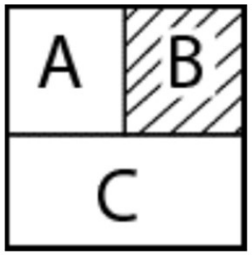
Value table

State	Action	Value
A	0	
A	1	
B	0	
B	1	
C	0	
C	1	

Q table

Value Iteration

- Suppose we have a Grid world environment with 3 states.
  - The initial state is state A, and the agent should move to state C without visiting the shaded state B.
  - We have two actions: 0-left/right, 1-up/down.
  - The optimal policy would be to choose '1' in state A.



- Model dynamics of state A

State (s)	Action (a)	Next State (s')	Transition Probability $P(s'\|s,a)$ or $P^a_{ss'}$	Reward Function $R(s,a,s')$ or $R^a_{ss'}$
A	0	A	0.1	0
A	0	B	0.8	-1
A	0	C	0.1	1
A	1	A	0.1	0
A	1	B	0.0	-1
A	1	C	0.9	1

Value Iteration

- We first need to compute values of all states.
- This can be done if we have the Q values for all state-action pairs because:

$$V^*(s) = \max_a Q^*(s, a)$$

- Now, the Q value for state s and action a can be computed as:

$$Q(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')] \qquad Q(s,a) = \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V(s')\right]$$

- In this equation, $P(s'|s,a)$ and $R(s,a,s')$ is known from the model dynamics, and γ is a parameter.

- What we do not know is the value of the next state $V(s')$.

Value Iteration

- Iteration 1
  - Since we do not know $V(s')$, we just use "some" value.
  - We initialize the value table with random values or zeros.

State	Value
A	0
B	0
C	0

  - Using this table, we compute the Q values of state A.
  - Since there two actions 0 and 1, we need to compute:
    - Q(A, 0)
    - Q(A, 1)
  - (Here we assume $\gamma = 1$ in this example.)

Value Iteration

– The Q values at state A

$$
\begin{aligned}
Q(A,0) &= P_{AA}^{0}[R_{AA}^{0} + \gamma V(A)] + P_{AB}^{0}[R_{AB}^{0} + \gamma V(B)] + P_{AC}^{0}[R_{AC}^{0} + \gamma V(C)] \\
&= 0.1(0+0) + 0.8(-1+0) + 0.1(1+0) \\
&= -0.7
\end{aligned}
$$

$$
\begin{aligned}
Q(A,1) &= P_{AA}^{1}[R_{AA}^{1} + \gamma V(A)] + P_{AB}^{1}[R_{AB}^{1} + \gamma V(B)] + P_{AC}^{1}[R_{AC}^{1} + \gamma V(C)] \\
&= 0.1(0+0) + 0.0(-1+0) + 0.9(1+0) \\
&= 0.9
\end{aligned}
$$

– Once we calculate the Q values, we update the Q table.

State	Action	Value
A	0	- 0.7
A	1	0.9
B	0	
B	1	
C	0	
C	1	

Value Iteration

– Once we update the Q table, we use the Q values to update the value table.

State	Value
A	0.9
B	0
C	0

– We do the same thing for state B and C and update the value table.

State	Value
A	0.9
B	- 0.2
C	0.5

Value Iteration

- Iteration 2
  - In iteration 1, we used the initial value table for $V(s')$.

$$V^*(s) = \max_a Q^*(s,a)$$

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

Random Values

  - In iteration 2, we repeat the same thing, but this time we use the updated value table for $V(s')$.

$$V^*(s) = \max_a Q^*(s,a)$$

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

Use state values from 1st iteration

Value Iteration

- Iteration 2
  - Q value of state A

$$
\begin{aligned}
Q(A,0) &= P_{AA}^{0}[R_{AA}^{0} + \gamma V(A)] + P_{AB}^{0}[R_{AB}^{0} + \gamma V(B)] + P_{AC}^{0}[R_{AC}^{0} + \gamma V(C)] \\
&= 0.1(0 + 0.9) + 0.8(-1 - 0.2) + 0.1(1 + 0.5) \\
&= -0.72
\end{aligned}
$$

$$
\begin{aligned}
Q(A,1) &= P_{AA}^{1}[R_{AA}^{1} + \gamma V(A)] + P_{AB}^{1}[R_{AB}^{1} + \gamma V(B)] + P_{AC}^{1}[R_{AC}^{1} + \gamma V(C)] \\
&= 0.1(0 + 0.9) + 0.0(-1 - 0.2) + 0.9(1 + 0.5) \\
&= 1.44
\end{aligned}
$$

  - Based on the results, $V(A)$ becomes 1.44.
  - We update the value table for all other states.

State	Value
A	1.44
B	- 0.50
C	1.0

Value Iteration

- Iteration 3
  - We repeat the same procedure and update the value table.

State	Value
A	1.94
B	- 0.70
C	1.3

- Iteration 4
  - Repeat.

State	Value
A	1.94
B	- 0.70
C	1.3

Value Iteration

- How many iterations should we go through?
  - We can observe that the value table after iteration 3 and iteration 4 are very similar.
  - This means that the value function has converged.
  - We can stop the iterations at this point.

State	Value
A	1.94
B	- 0.70
C	1.3

after iteration 3

State	Value
A	1.94
B	- 0.70
C	1.3

after iteration 4

Value Iteration

- Once we are done updating the value table, we extract a policy from the value table.
  - Similar to what we've done in each iteration, we calculate Q value for each state.

$$
\begin{aligned}
Q(A,0) &= P_{AA}^{0}[R_{AA}^{0} + \gamma V(A)] + P_{AB}^{0}[R_{AB}^{0} + \gamma V(B)] + P_{AC}^{0}[R_{AC}^{0} + \gamma V(C)] \\
&= 0.1(0 + 1.95) + 0.8(-1 - 0.72) + 0.1(1 + 1.3) \\
&= -0.951 \\
Q(A,1) &= P_{AA}^{1}[R_{AA}^{1} + \gamma V(A)] + P_{AB}^{1}[R_{AB}^{1} + \gamma V(B)] + P_{AC}^{1}[R_{AC}^{1} + \gamma V(C)] \\
&= 0.1(0 + 1.95) + 0.0(-1 - 0.72) + 0.9(1 + 1.3) \\
&= 2.26
\end{aligned}
$$

  - From the Q table, we get the optimal policy.

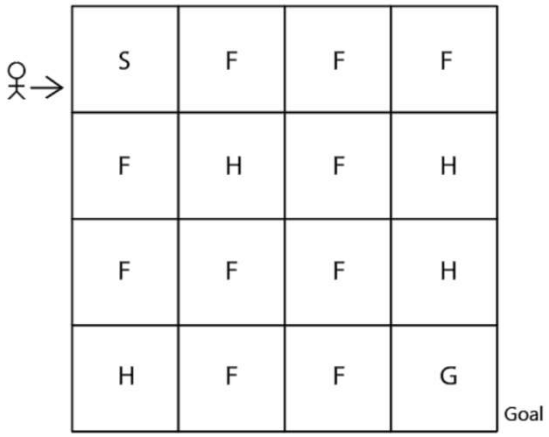
State	Action	Value
A	0	- 0.95
A	1	2.26
B	0	- 0.5
B	1	0.5
C	0	- 1.1
C	1	1.4

$$\pi^* = \arg\max_a Q(s,a)$$

Solving Frozen Lake with Value Iteration [ex002]

- Let's use value iteration to get an optimal policy for the Frozen Lake environment.

```
1   import numpy as np
2   import gym
3   env = gym.make('FrozenLake-v1')
```



Solving Frozen Lake with Value Iteration

- Function for value iteration

```python
def value_iteration(env):

    num_iterations = 1000
    threshold = 1e-20
    gamma = 1.0

    value_table = np.zeros(env.observation_space.n)

    for i in range(num_iterations):

        updated_value_table = np.copy(value_table)

        for s in range(env.observation_space.n):

            Q_values = [sum([prob*(r + gamma * updated_value_table[s_])
                             for prob, s_, r, _ in env.P[s][a]])
                                for a in range(env.action_space.n)]

            value_table[s] = max(Q_values)

        if np.sum(np.fabs(updated_value_table - value_table)) <= threshold:
            break

    return value_table
```

$$Q(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$$

$$V^*(s) = \max_a Q^*(s,a)$$

Solving Frozen Lake with Value Iteration

- Function for extracting policy from the value table

```
def extract_policy(value_table):

    gamma = 1.0
    policy = np.zeros(env.observation_space.n)
    for s in range(env.observation_space.n):

        Q_values = [sum([prob*(r + gamma * value_table[s_])
                                for prob, s_, r, _ in env.P[s][a]])
                                    for a in range(env.action_space.n)]

        policy[s] = np.argmax(np.array(Q_values))

    return policy
```

$$\pi^* = \arg\max_a Q(s, a)$$

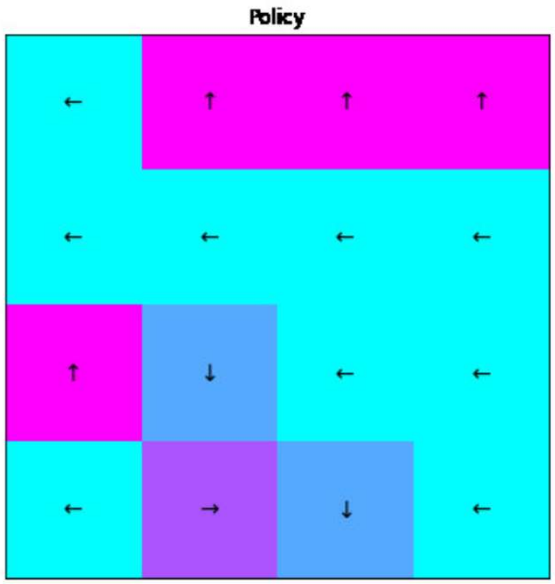
Solving Frozen Lake with Value Iteration

- Using the functions to obtain the optimal policy
  - Run value iteration
  - Extract policy from the converged value table

```
optimal_value_function = value_iteration(env)
optimal_policy = extract_policy(optimal_value_function)
print(optimal_policy)
```

- Result
  - Actions to take in all 16 states.
  - 0: left, 1: down, 2: right, 3: up

```
(rl) jso1@ics1-37:~/teaching/RL$ python3 ex002.py
[0. 3. 3. 3. 0. 0. 0. 0. 3. 1. 0. 0. 0. 2. 1. 0.]
```



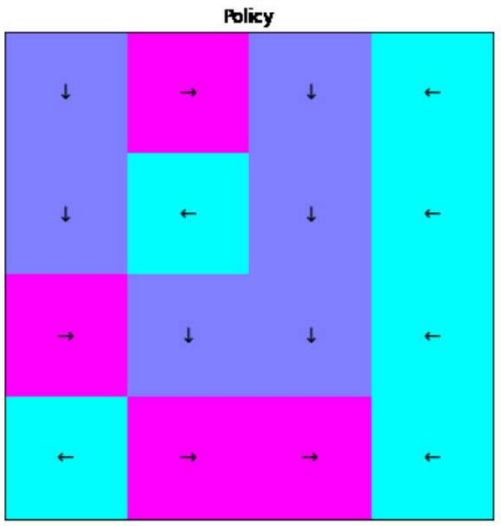
Solving Frozen Lake with Value Iteration

- We can make the environment non-slippery by giving an option.

```
import numpy as np
import gym
env = gym.make('FrozenLake-v1', is_slippery=False)
```

- When we apply value iteration, we must set γ to a value smaller than 1.
  - If $\gamma = 1$, the agent doesn't care whether it moves towards a wall and stays in the same state.
  - If $\gamma < 1$, the agent prefers moving directly towards the goal than staying in the same state
    - It is giving more value to the immediate reward.
    - You can set $\gamma = 0.9$ or $\gamma = 0.99$

- Optimal policy for non-slippery Frozen Lake



Solving Frozen Lake with Value Iteration

- Evaluating the policy
  - The policy calculated from value iteration achieves approximately 75% win ratio.

```
def evaluate_policy(policy):
    num_episodes = 1000
    num_timesteps = 1000
    total_reward = 0
    total_timestep = 0

    for i in range(num_episodes):
        state = env.reset()

        for t in range(num_timesteps):
            if policy is None:
                action = env.action_space.sample()
            else:
                action = policy[state]
            state, reward, done, info = env.step(action)
            total_reward += reward

            if done:
                break

        total_timestep += t

    print("Number of successful episodes: %d / %d"%(total_reward, num_episodes))
    print("Average number of timesteps per episode: %.2f"%(total_timestep/num_episodes))
```

```
optimal_value_function = value_iteration(env)
optimal_policy = extract_policy(optimal_value_function)
evaluate_policy(None)
evaluate_policy(optimal_policy)
```

The Bellman Equation and Dynamic Programming

Dynamic Programming - Policy Iteration

Policy Iteration

- Policy Iteration is another dynamic programming method to find the optimal policy

- In the value iteration method, we compute the optimal value function by taking the maximum over the Q function iteratively.
  - Once we find the optimal value function, we extract optimal policy from there.
  - * We did not use a specific policy for updating $V(s)$ in the value iteration method.

- In policy iteration, we try to compute the optimal value function using the policy iteratively.
  - Once we find the optimal value function, we extract optimal policy from there.

Policy Iteration

- Suppose we have a deterministic policy π. Then, the value function can be calculated as:

$$V^{\pi}(s) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')] \quad \text{or} \quad V^{\pi}(s) = \sum_{s'} P_{ss'}^{a}\left[R_{ss'}^{a} + \gamma V(s')\right]$$

- Iteration 1: Since we don't have any knowledge on value of the states, we use a random policy. We use this policy to calculate the value function V^{π_0}. From V^{π_0}, we extract a new policy π_1.
- Iteration 2: Using the new policy π_1, we compute the new value function V^{π_1}. From V^{π_1}, we extract a new policy π_2.
- Iteration 3: Using the new policy π_2, we compute the new value function V^{π_2}. From V^{π_2}, we extract a new policy π_3.
- We continue until we think the optimal value function is found.
  - Possible stop condition: The new policy is the same as the previous one.

$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \pi_2 \rightarrow V^{\pi_2} \rightarrow \pi_3 \rightarrow V^{\pi_3} \rightarrow \cdots \rightarrow \pi^* \rightarrow V^{\pi^*}$$

Policy Iteration

- The policy iteration algorithm

1. Initialize a random policy.

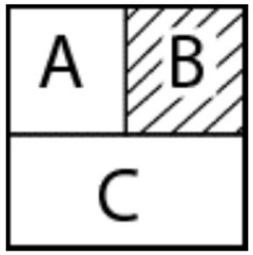
2. Compute the optimal value function using the given policy iteratively.

3. Extract a new policy using the value function obtained from step 2.

4. If the extracted policy is the same as the policy used in step 2, then stop, else send the extracted new policy to step 2 and repeat steps 2 through 4.

Policy Iteration

- This is the same example scenario used in the value iteration method.
- Suppose we have a Grid world environment with 3 states.
  - The initial state is state A, and the agent should move to state C without visiting the shaded state B.
  - We have two actions: 0-left/right, 1-up/down.
  - The optimal policy would be to choose '1' in state A.



State (s)	Action (a)	Next State (s')	Transition Probability $P(s'\|s,a)$ or $P^a_{ss'}$	Reward Function $R(s,a,s')$ or $R^a_{ss'}$
A	0	A	0.1	0
A	0	B	0.8	-1
A	0	C	0.1	1
A	1	A	0.1	0
A	1	B	0.0	-1
A	1	C	0.9	1

Policy Iteration: Step 1

- We compute $V^{\pi}(s)$ using the equation:

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

- Initially, we choose a random policy

$$A \rightarrow 1$$
$$B \rightarrow 0$$
$$C \rightarrow 1$$

- Also, we initialize the value function to random values or zeros since we do not have any information

State	Value
A	0
B	0
C	0

Policy Iteration: Step 2

- Iteration 1
  - We compute the value function using the current policy and the current value table.
  - According to the policy, the agent chooses action '1' in state A.

$$\begin{aligned} V(A) &= P^1_{AA}[R^1_{AA} + \gamma V(A)] + P^1_{AB}[R^1_{AB} + \gamma V(B)] + P^1_{AC}[R^1_{AC} + \gamma V(C)] \\ &= 0.1(0+0) + 0.0(-1+0) + 0.9(1+0) \\ &= 0.9 \end{aligned}$$

  - Similarly, we compute the value for all the states based on the current policy.

State	Value
A	0.9
B	- 0.2
C	0.1

Policy Iteration: Step 2

- Iteration 2
  - We compute the value function using the current policy and the value table updated from Iteration 1.
  - According to the policy, the agent chooses action '1' in state A.

$$\begin{aligned} V(A) &= P^1_{AA}[R^1_{AA} + \gamma V(A)] + P^1_{AB}[R^1_{AB} + \gamma V(B)] + P^1_{AC}[R^1_{AC} + \gamma V(C)] \\ &= 0.1(0 + 0.9) + 0.0(-1 - 0.2) + 0.9(1 + 0.1) \\ &= 1.08 \end{aligned}$$

  - Similarly, we compute the value for all the states based on the current policy.

State	Value
A	1.08
B	- 0.5
C	0.5

Value Iteration

- Iteration 3
  - We repeat the same procedure and update the value table.

State	Value
A	1.45
B	- 0.9
C	0.6

- Iteration 4
  - Repeat.

State	Value
A	1.46
B	- 0.9
C	0.61

- We stop here because there is little change in the value table.

Policy Iteration: Step 3

- Based on the updated V, we extract a new policy that we think is optimal.
  - We compute Q values for state-action pairs using the equation

$$Q(s,a) = \sum_{s'} P^a_{SS'}[R^a_{SS'} + \gamma V(s')]$$

  - For state A,

$$\begin{aligned} Q(A,0) &= P^0_{AA}[R^0_{AA} + \gamma V(A)] + P^0_{AB}[R^0_{AB} + \gamma V(B)] + P^0_{AC}[R^0_{AC} + \gamma V(C)] \\ &= 0.1(0 + 1.46) + 0.8(-1 - 0.9) + 0.1(1 + 0.61) \\ &= -1.21 \\ Q(A,1) &= P^1_{AA}[R^1_{AA} + \gamma V(A)] + P^1_{AB}[R^1_{AB} + \gamma V(B)] + P^1_{AC}[R^1_{AC} + \gamma V(C)] \\ &= 0.1(0 + 1.46) + 0.0(-1 - 0.9) + 0.9(1 + 0.61) \\ &= 1.59 \end{aligned}$$

  - We compute Q table for all state-action pairs

State	Action	Value
A	0	- 1.21
A	1	1.59
B	0	0.1
B	1	0.0
C	0	0.5
C	1	0.0

  - From the table, we obtain the new policy

$$\begin{aligned} A &\rightarrow 1 \\ B &\rightarrow 0 \\ C &\rightarrow 0 \end{aligned}$$

Policy Iteration: Step 4

- Check whether the new policy is the same as the previous policy.
  - If they are the same, then stop.
  - If not, then go back to step 2.

Solving Frozen Lake with Policy Iteration [ex003]

- Let's use policy iteration to get an optimal policy for the Frozen Lake environment.

```
import numpy as np
import gym
env = gym.make("FrozenLake-v1")
```

Solving Frozen Lake with Policy Iteration

- Function for computing value function from a policy

```
def compute_value_function(policy):

    num_iterations = 1000
    threshold = 1e-20
    gamma = 1.0

    value_table = np.zeros(env.observation_space.n)

    for i in range(num_iterations):
        updated_value_table = np.copy(value_table)

        for s in range(env.observation_space.n):
            a = policy[s]

            value_table[s] = sum(
                [prob * (r + gamma * updated_value_table[s_])
                    for prob, s_, r, _ in env.P[s][a]])

        if np.sum(np.fabs(updated_value_table - value_table)) <= threshold:
            break

    return value_table
```

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

Solving Frozen Lake with Policy Iteration

- Function for extracting policy from the value table
  - This is the same function used for value iteration

```
def extract_policy(value_table):

    gamma = 1.0
    policy = np.zeros(env.observation_space.n)
    for s in range(env.observation_space.n):

        Q_values = [sum([prob*(r + gamma * value_table[s_])
                         for prob, s_, r, _ in env.P[s][a]])
                            for a in range(env.action_space.n)]

        policy[s] = np.argmax(np.array(Q_values))

    return policy
```

$$\pi^* = \arg\max_a Q(s, a)$$

Solving Frozen Lake with Policy Iteration

- Function for performing policy iteration

```
def policy_iteration(env):

    num_iterations = 1000

    policy = np.zeros(env.observation_space.n)

    for i in range(num_iterations):

        value_function = compute_value_function(policy)

        new_policy = extract_policy(value_function)

        if np.all(policy == new_policy):
            break

        policy = new_policy

    return policy
```

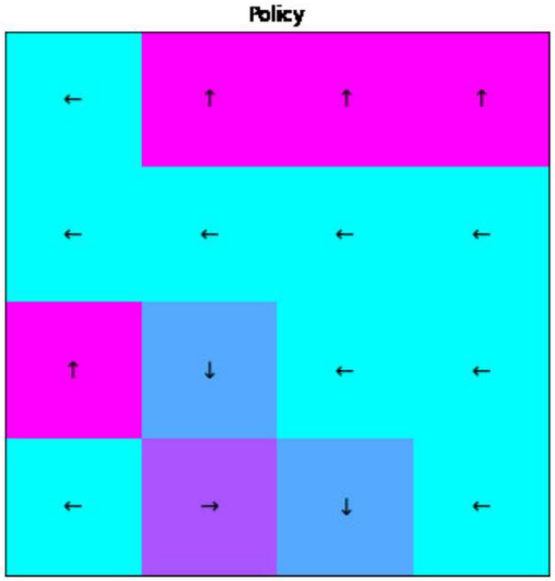
Solving Frozen Lake with Policy Iteration

- Using the functions to obtain the optimal policy

```
optimal_policy = policy_iteration(env)
print(optimal_policy)
```

- Result
  - Same as one obtained by value iteration
  - Actions to take in all 16 states.
  - 0: left, 1: down, 2: right, 3: up

```
(rl) jso1@icsl-37:~/teaching/RL$ python3 ex003.py
[0. 3. 3. 3. 0. 0. 0. 0. 3. 1. 0. 0. 0. 2. 1. 0.]
```



Is DP applicable to all environments?

- In value iteration, we need to use the following equation to compute Q values for all state-action pairs.

$$Q(s,a) = \sum_{s'} P^a_{SS'}[R^a_{SS'} + \gamma V(s')]$$

- In policy iteration, we need to use the following equation to compute value function based on a policy.

$$V^\pi(s) = \sum_{s'} P^a_{SS'}[R^a_{SS'} + \gamma V^\pi(s')]$$

- To compute these values, we need to know the model dynamics.
  - transition probability and reward function

- Thus, DP is a model-based method.
- In the next chapter, we learn the Monte Carlo method which is a model-free method.

End of Chapter

- Can you use value iteration and policy iteration methods to train an agent in the Frozen Lake environment?

- Can you write [ex002] and [ex003] yourself?

End of Class

Questions?

Email: jso1@sogang.ac.kr