Intro to Reinforcement Learning Part II

Dr. Dongchul Kim
Department of Computer Science
UTRGV

Prediction Problem

How to estimate value functions when MDP is unknown

Outlines

1. Model-free
2. Monte Carlo
3. Temporal Difference

Outlines

1. **Model-free**
2. Monte Carlo
3. Temporal Difference

Model free

Currently, we lack knowledge of the underlying MDP. This means that critical components such as the **reward function** (R^a_s) and the **transition probability** ($P^a_{ss'}$) remain unknown to us.

Consequently, we face uncertainty regarding the expected compensation for taking specific actions and the likelihood of transitioning to different states before those actions are executed.

“small” environment

We are going to use a “small” environment still.

Our approach involves employing a **table lookup** method.

This method entails the creation of a table wherein we can record and continually update the values associated with various states and actions.

Outlines

1. Model-free
2. **Monte Carlo**
3. Temporal Difference

Monte Carlo Method

The Monte Carlo method for estimating the state value function in an unknown Markov Decision Process (MDP) relies on **generating random episodes** of interactions with the environment, observing the rewards obtained, and then **averaging these returns** over multiple episodes to estimate the expected value of each state.

Step 1 - Initialization

s_0	s_1	s_2	s_3
s_4	s_5	s_6	s_7
s_8	s_9	s_{10}	s_{11}
s_{12}	s_{13}	s_{14}	s_{15}

`(N(s), V(s))`

(0, 0)	(0, 0)	(0, 0)	(0, 0)
(0, 0)	(0, 0)	(0, 0)	(0, 0)
(0, 0)	(0, 0)	(0, 0)	(0, 0)
(0, 0)	(0, 0)	(0, 0)	(0, 0)

Step 2 - Experiencing Episodes

The agent is now at the beginning, and the episode has started. Agents using a random policy go to different states and get rewards randomly. When they reach the final end state, the episode ends, and this is how their path looks.

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots, a_{T-1}, r_T, s_T$$

Step 2 - Experiencing Episodes

You can find *Return*, G$_t$ like this:

$$G_0 = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \ldots + \gamma^{T-1} r_T$$

$$G_1 = r_2 + \gamma r_3 + \gamma^2 r_4 + \ldots + \gamma^{T-2} r_T$$

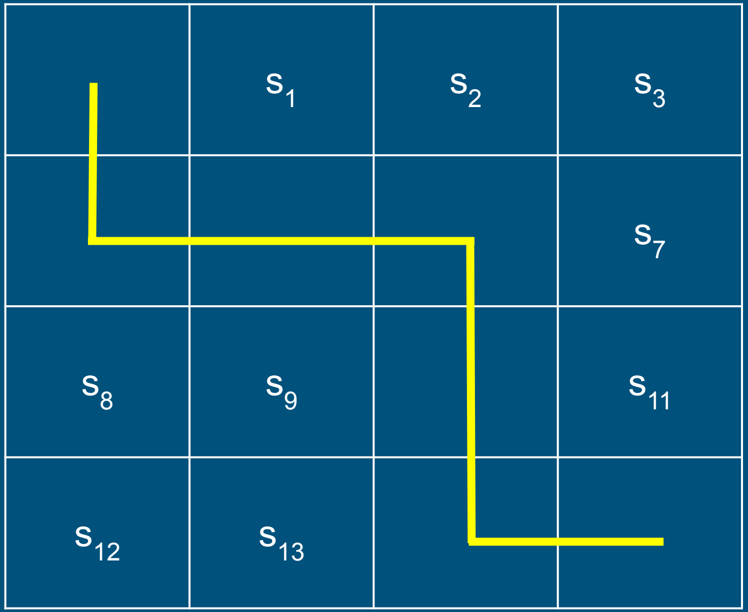
$$\ldots$$

$$G_{T-1} = r_T$$

$$G_T = 0$$

Step 3 - Table Update

$$N(s_t) \leftarrow N(s_t) + 1$$
$$V(s_t) \leftarrow V(s_t) + G_t$$



`(N(s), V(s))`

(1, -6)	(0, 0)	(0, 0)	(0, 0)
(1, -5)	(1, -4)	(1, -3)	(0, 0)
(0, 0)	(0, 0)	(1, -2)	(0, 0)
(0, 0)	(0, 0)	(1, -1)	End

Step 4 - Estimate State Value

$$v_\pi(s_t) = \frac{V(s_t)}{N(s_t)}$$

Update with Learning rate

$$N(s_t) \leftarrow N(s_t) + 1$$
$$V(s_t) \leftarrow V(s_t) + G_t$$

$$v_\pi(s_t) = \frac{V(s_t)}{N(s_t)}$$

$$\Rightarrow$$

$$V(s_t) \leftarrow (1 - \alpha) * V(s_t) + \alpha * G_t$$
$$V(s_t) \leftarrow V(s_t) + \alpha(G_t - V(s_t))$$

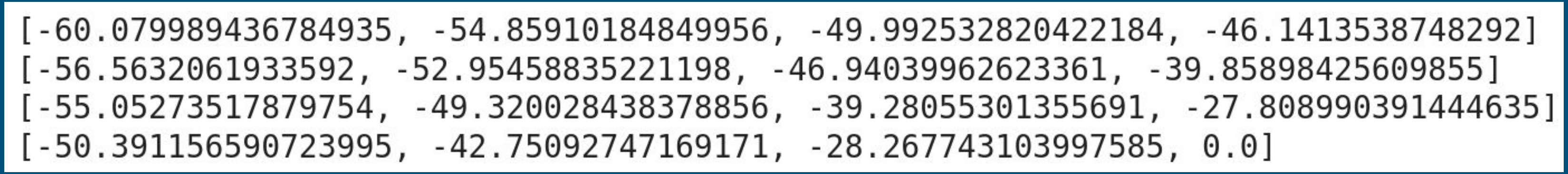
Implementation!

```
class GridWorld():
    def __init__(self):
        self.x=0
        self.y=0

    def step(self, a):
        if a==0:
            self.move_left()
        elif a==1:
            self.move_up()
        elif a==2:
            self.move_right()
        elif a==3:
            self.move_down()

        reward = -1
        done = self.is_done()
        return (self.x, self.y), reward, done

    def move_right(self):
        self.y += 1
        if self.y > 3:
            self.y = 3

    def move_left(self):
        self.y -= 1
        if self.y < 0:
            self.y = 0

    def move_up(self):
        self.x -= 1
        if self.x < 0:
            self.x = 0

    def move_down(self):
        self.x += 1
        if self.x > 3:
            self.x = 3

    def is_done(self):
        if self.x == 3 and self.y == 3:
            return True
        else :
            return False

    def get_state(self):
        return (self.x, self.y)

    def reset(self):
        self.x = 0
        self.y = 0
        return (self.x, self.y)
```

```
class Agent():
    def __init__(self):
        pass

    def select_action(self):
        coin = random.random()
        if coin < 0.25:
            action = 0
        elif coin < 0.5:
            action = 1
        elif coin < 0.75:
            action = 2
        else:
            action = 3
        return action
```

```
def main():
  env = GridWorld()
  agent = Agent()
  data = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
  gamma = 1.0
  reward = -1
  alpha = 0.001

  for k in range(50000):
      done = False
      history = []

      while not done:
          action = agent.select_action()
          (x,y), reward, done = env.step(action)
          history.append((x,y,reward))
      env.reset()

      cum_reward = 0
      for transition in history[::-1]:
          x, y, reward = transition
          data[x][y] = data[x][y] + alpha*(cum_reward-data[x][y])
          cum_reward = reward + gamma*cum_reward

  for row in data:
      print(row)
```

Results



```
[-60.079989436784935, -54.85910184849956, -49.992532820422184, -46.1413538748292]
[-56.5632061933592, -52.95458835221198, -46.94039962623361, -39.85898425609855]
[-55.05273517879754, -49.320028438378856, -39.28055301355691, -27.80899039144635]
[-50.391156590723995, -42.75092747169171, -28.267743103997585, 0.0]
```

Outlines

1. Model-free
2. Monte Carlo
3. **Temporal Difference**

Temporal Difference

The MC method needs the episode to finish before updating the table, which could be a drawback. In certain situations, the episode might not even finish. On the other hand, the **Temporal Difference** method can update the table even if the episode is ongoing.

Temporal Difference

Do you remember Ver. 1 of Bellman Expectation Equation?

$$v_\pi(s_t) = \mathbb{E}_\pi[G_t]$$

$$v_\pi(s_t) = \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(s_{t+1})]$$

Temporal Difference

For example,

$$s_0 \to s_1 \to s_2 \to s_3 \to s_7 \to s_6 \to s_{10} \to s_{11} \to s_{15}$$

MC: $V(s_t) \leftarrow V(s_t) + \alpha(G_t - V(s_t))$

TD: $V(s_t) \leftarrow V(s_t) + \alpha(\underbrace{r_{t+1} + \gamma V(s_{t+1})}_{\text{TD target}} - V(s_t))$

$$V(s_0) \leftarrow V(s_0) + 0.01 * (-1 + V(s_1) - V(s_0))$$

$$V(s_1) \leftarrow V(s_1) + 0.01 * (-1 + V(s_2) - V(s_1))$$

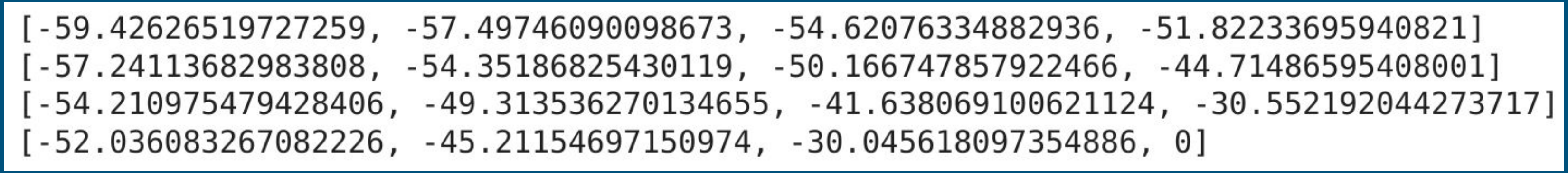
...

$$V(s_{11}) \leftarrow V(s_{11}) + 0.01 * (-1 + V(s_{15}) - V(s_{11}))$$

Implementation!

```
def main():
    #TD
    env = GridWorld()
    agent = Agent()
    data = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
    gamma = 1.0
    reward = -1
    alpha = 0.01

    for k in range(50000):
        done = False
        while not done:
            x, y = env.get_state()
            action = agent.select_action()
            (x_prime, y_prime), reward, done = env.step(action)
            x_prime, y_prime = env.get_state()
            data[x][y] = data[x][y] + alpha*(reward+gamma*data[x_prime][y_prime]-data[x][y])
        env.reset()

    for row in data:
        print(row)
```

Results



```
[-59.42626519727259, -57.49746090098673, -54.62076334882936, -51.82233695940821]
[-57.24113682983808, -54.35186825430119, -50.16674785792246, -44.71486595408001]
[-54.210975479428406, -49.313536270134655, -41.638069100621124, -30.55219204427371 7]
[-52.036083267082226, -45.21154697150974, -30.045618097354886, 0]
```

MC vs TD

- Episodic MDP vs **Non-episodic MDP**
- **Biased** vs Unbiased
- High variance vs **Low variance**

N-step TD

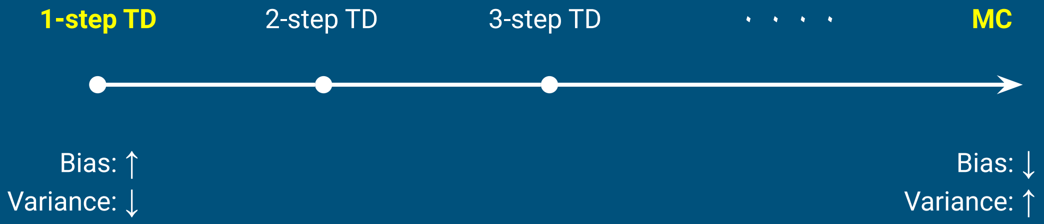
N=1: $r_{t+1} + \gamma V(s_{t+1})$

N=2: $r_{t+1} + \gamma V(s_{t+1}) + \gamma^2 V(s_{t+2})$

N=3: $r_{t+1} + \gamma V(s_{t+1}) + \gamma^2 V(s_{t+2}) + \gamma^3 V(s_{t+3})$

$\dots$

N=n: $r_{t+1} + \gamma V(s_{t+1}) + \gamma^2 V(s_{t+2}) + \dots + \gamma^n V(s_{t+n})$



Control Problem

How to find optimal policy when MDP is unknown

Outlines

1. MC Control
2. TD Control - SARSA
3. TD Control - Q Learning
    a. On-policy vs Off-policy

Outlines

1. **MC Control**
2. TD Control - SARSA
3. TD Control - Q Learning
    a. On-policy vs Off-policy

Policy Iteration in known MDP

Let's revisit the control techniques we previously studied.

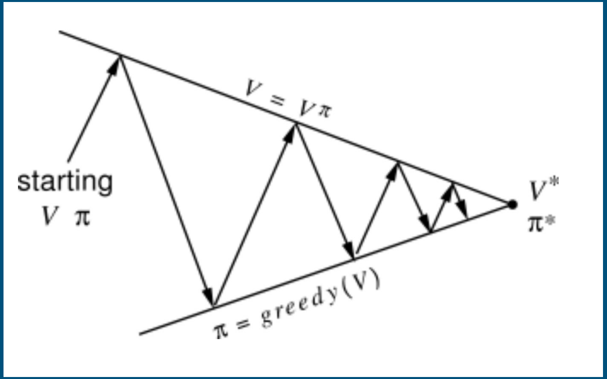
Could you recall the approach for determining the policy function when MDP known? **Policy Iteration** is an approach involving the iteration of steps, starting with an initial random policy, followed by policy evaluation (computing state values), and ultimately devising a new (greedy) policy based on the obtained value estimates.

Is it possible to apply the policy iteration method in model-free?

Policy Iteration in known MDP

In summary,

1. Initialize a random policy.
2. **Evaluate the policy** using iterative policy evaluation.
3. **Improve the policy** by selecting the greedy action based on the value function obtained in step 2.
4. Repeat steps 2-3 until convergence.



http://incompleteideas.net/book/ebook/the-book.html

Policy Iteration in Model-Free

Two reasons we can't use Policy Iteration in model-free

Problem 1 we can't use Bellman Expectation Equation for Policy Evaluation.

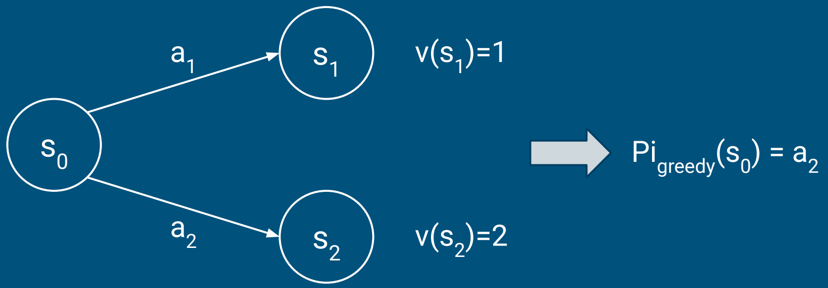
$$v_\pi(s) = \sum_{a \in A} \pi(a|s)\left(r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')\right)$$

reward function (R^a_s) and **transition probability** ($P^a_{ss'}$) are unknown in model-free!!

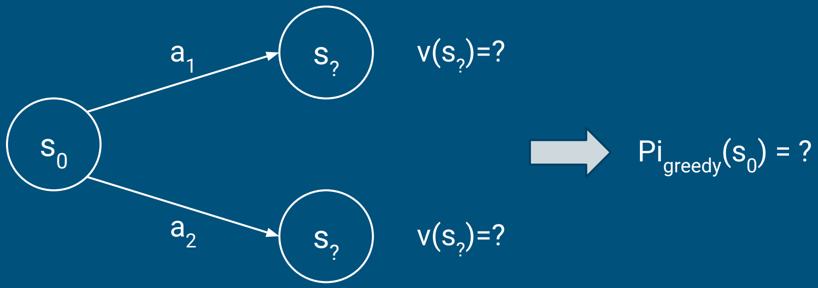
Policy Iteration in Model-Free

Problem 2

Even if we are able to estimate state values (v(s)), updating the policy using a greedy approach is not viable because it's impossible to predict the next state when choosing an action in the current state.



When MDP is known



When MDP is unknown

Solutions

1. (For Problem 1) For policy evaluation, you can employ **Monte Carlo (MC)** instead of the Bellman equation.

2. (For Problem 2) When addressing the policy improvement problem, consider utilizing the **action value function (Q function)** instead of the state value function.

3. Using the action value function introduces an exploration challenge, which can be tackled by **ε-greedy** rather than a purely greedy approach.

Policy Iteration in model-free

Policy Evaluation

  MC to estimate q(s, a)

Policy Improvement

  Greedy policy with q(s, a)

Exploration

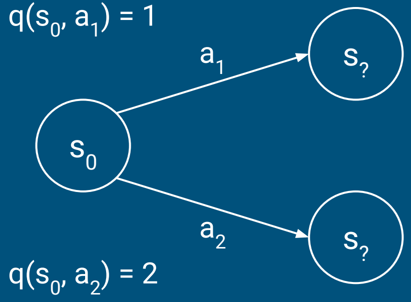
  ε-greedy

Greedy policy with q(s, a)

There was an issue with the state value function *v(s)* during policy improvement in model-free.

By employing the action value function *q(s, a)*, you gain the ability to make action selections.

In essence, while we may not precisely know which state will be reached when choosing an action in state *s*, we can make informed decisions by **selecting the action with the highest *q(s, a)* value**, as we have knowledge of the expected returns for each action.



$\Rightarrow$ $Pi_{greedy}(s_0) = a_2$

q(s, a) causes a problem

In the context of policy iteration, the action-value function q(s, a) starts with an initialization of 0 and is subsequently updated using Monte Carlo (MC) methods.

During this process, for example, $q(s_3, a_1)$ might be updated to 0.1, while the values for the other actions like (s_3, a_2), (s_3, a_3), and so on, remain at 0. Consequently, when we determine the next policy by opting for a greedy approach, **only a_1 gets selected in state s_3**, potentially overlooking better-performing actions.

To address this issue, we introduce an element of randomness by taking actions based on certain probabilities. This strategy is known as **ε-greedy**, allowing us to explore alternatives beyond the purely greedy selection and potentially discover better actions.

ε-greedy

$$\begin{tikzcd}
 & \text{Exploitation (selects the greedy action)} \\
\varepsilon\text{-greedy policy} \arrow[ur, "1-\varepsilon"] \arrow[dr, "\varepsilon"'] & \\
 & \text{Exploration (selects a \textbf{random} action)}
\end{tikzcd}$$

MC Control

$$\begin{tikzcd}
\pi \arrow[r, bend left, "\text{Policy Evaluation} \\ \text{MC}"] & q_{\pi} \arrow[l, bend left, "\text{Policy Improvement} \\ \text{esp-greedy}"]
\end{tikzcd}$$

Implementation

Environment

Grid World

Actions

Left: 0

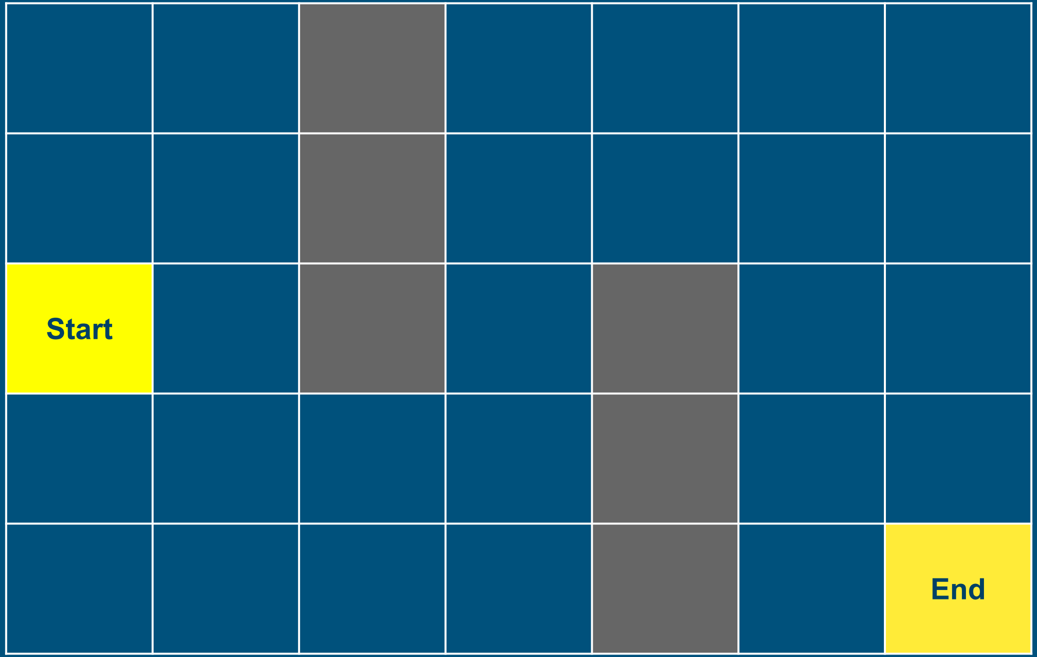
Up: 1

Right: 2

Down: 3

Reward

-1 per move



```python
class GridWorld():
    def __init__(self):
        self.x=0
        self.y=0

    def step(self, a):
        if a==0:
            self.move_left()
        elif a==1:
            self.move_up()
        elif a==2:
            self.move_right()
        elif a==3:
            self.move_down()

        reward = -1
        done = self.is_done()
        return (self.x, self.y), reward, done

    def move_left(self):
        if self.y==0:
            pass
        elif self.y==3 and self.x in [0,1,2]:
            pass
        elif self.y==5 and self.x in [2,3,4]:
            pass
        else:
            self.y -= 1
```

```python
def move_right(self):
    if self.y==1 and self.x in [0,1,2]:
        pass
    elif self.y==3 and self.x in [2,3,4]:
        pass
    elif self.y==6:
        pass
    else:
        self.y += 1

def move_up(self):
    if self.x==0:
        pass
    elif self.x==3 and self.y==2:
        pass
    else:
        self.x -= 1

def move_down(self):
    if self.x==4:
        pass
    elif self.x==1 and self.y==4:
        pass
    else:
        self.x+=1
```

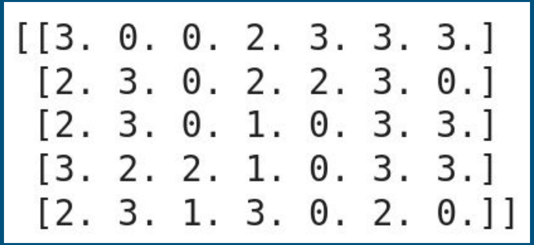
```python
def is_done(self):
    if self.x==4 and self.y==6:
        return True
    else:
        return False

def reset(self):
    self.x = 0
    self.y = 0
    return (self.x, self.y)
```

```python
class QAgent():
    def __init__(self):
        self.q_table = np.zeros((5, 7, 4))
        self.eps = 0.9
        self.alpha = 0.01

    def select_action(self, s): # eps-greedy
        x, y = s
        coin = random.random()
        if coin < self.eps: # random action
            action = random.randint(0,3)
        else:
            action_val = self.q_table[x,y,:]
            action = np.argmax(action_val)
        return action

    def update_table(self, history):
        # Given a single episode, the q-table's values are updated accordingly.
        cum_reward = 0
        for transition in history[::-1]: # from the last transition
            s, a, r, s_prime = transition
            x,y = s
            self.q_table[x,y,a] = self.q_table[x,y,a] + self.alpha * (cum_reward - self.q_table[x,y,a])
            cum_reward = cum_reward + r

    def anneal_eps(self):
        self.eps -= 0.03
        self.eps = max(self.eps, 0.1)
```

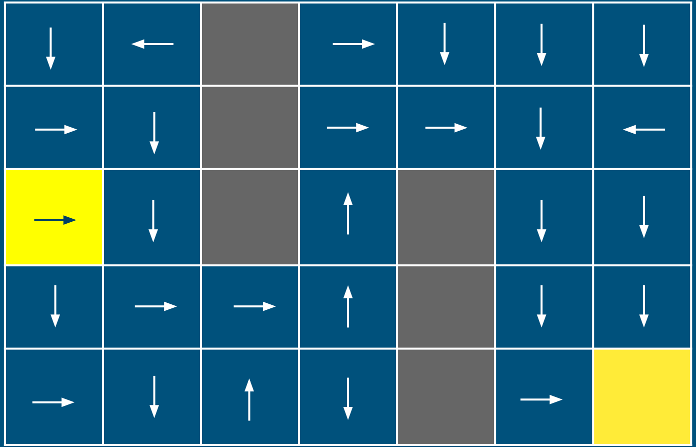
```python
    def show_table(self):
        q_lst = self.q_table.tolist()
        data = np.zeros((5,7))
        for row_idx in range(len(q_lst)):
            row = q_lst[row_idx]
            for col_idx in range(len(row)):
                col = row[col_idx]
                action = np.argmax(col)
                data[row_idx, col_idx] = action
        print(data)
```

```python
def main():
    env = GridWorld()
    agent = QAgent()

    for n_epi in range(1000): # 1000 episodes
        done = False
        history = []

        s = env.reset()
        while not done: # single episode
            a = agent.select_action(s)
            s_prime, r, done = env.step(a)
            history.append((s, a, r, s_prime))
            s = s_prime
        agent.update_table(history) # update table
        agent.anneal_eps()

    agent.show_table()
```

Results



```
[[3. 0. 0. 2. 3. 3. 3.]
 [2. 3. 0. 2. 2. 3. 0.]
 [2. 3. 0. 1. 0. 3. 3.]
 [3. 2. 2. 1. 0. 3. 3.]
 [2. 3. 1. 3. 0. 2. 0.]]
```



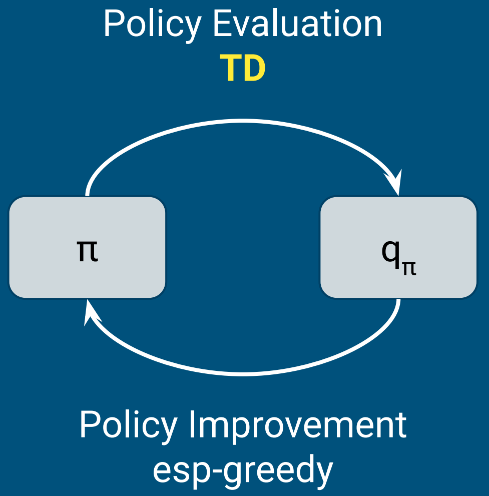
Outlines

1. MC Control
2. **TD Control - SARSA**
3. TD Control - Q Learning
   a. On-policy vs Off-policy

TD Control - SARSA

Similar to MC, **TD** can be employed to compute q(s, a) (policy evaluation), followed by policy improvement through ε-greedy methods.

TD offers the benefit of lower variance and enables learning to occur even before an episode concludes.



TD Control - SARSA

Do you remember Ver. 1?

$$v_\pi(s_t) = \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(s_{t+1})]$$

$$q_\pi(s_t, a_t) = \mathbb{E}_\pi[r_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1})]$$

Bellman Expectation Equations for v and q

TD Control - SARSA

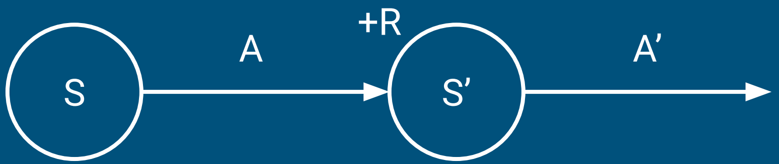
Let's see the TD targets for learning V and Q.

TD for Learning V: $V(s_t) \leftarrow V(s_t) + \alpha(\underbrace{r_{t+1} + \gamma V(s_{t+1})}_{\text{TD target}} - V(s_t))$

TD for Learning Q: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(\underbrace{r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})}_{\text{TD target}} - Q(s_t, a_t))$

TD Control - SARSA

This TD control method has a specific designation: **SARSA**. When you select action "a" in state "s," resulting in a reward "r" and a transition to state "s'" with the subsequent choice of action "a'," these transition events, namely "s," "a," "r," "s'," and "a'," are termed **SARSA**.



*Note that an actual transition (single step) we use in each update does not include a’ but only s, a, r, s’. We use a simulated a’. Please see the following implementation.

Implementation

Environment

Grid World

Actions

Left: 0

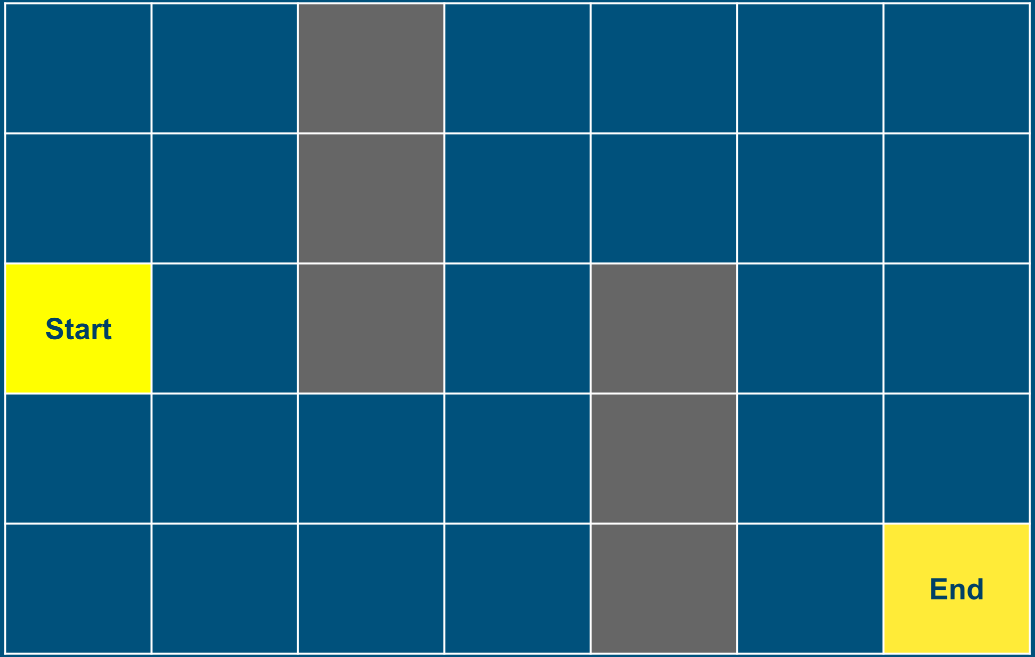
Up: 1

Right: 2

Down: 3

Reward

-1 per move

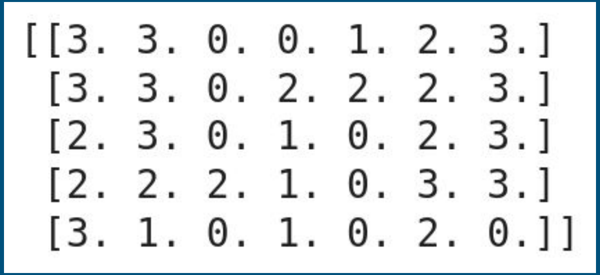


```
class QAgent():
    def __init__(self):
        self.q_table = np.zeros((5, 7, 4))
        self.eps = 0.9

    def select_action(self, s):
        x, y = s
        coin = random.random()
        if coin < self.eps:
            action = random.randint(0,3)
        else:
            action_val = self.q_table[x,y,:]
            action = np.argmax(action_val)
        return action

    def update_table(self, transition):
        s, a, r, s_prime = transition
        x,y = s
        next_x, next_y = s_prime
        a prime = self.select action(s prime)
        self.q_table[x,y,a] = self.q_table[x,y,a] + 0.1 * (r + self.q_table[next_x,next_y,a_prime] - self.q_table[x,y,a])
```

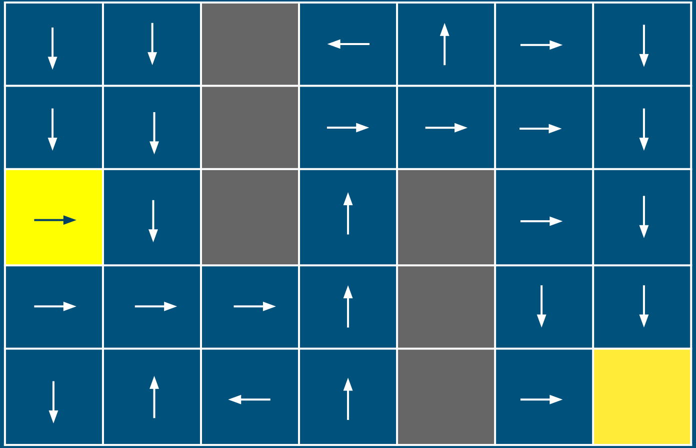
```
    def anneal_eps(self):
        self.eps -= 0.03
        self.eps = max(self.eps, 0.1)

    def show_table(self):
        q_lst = self.q_table.tolist()
        data = np.zeros((5,7))
        for row_idx in range(len(q_lst)):
            row = q_lst[row_idx]
            for col_idx in range(len(row)):
                col = row[col_idx]
                action = np.argmax(col)
                data[row_idx, col_idx] = action
        print(data)
```

```python
def main():
    env = GridWorld()
    agent = QAgent()

    for n_epi in range(1000):
        done = False

        s = env.reset()
        while not done:
            a = agent.select_action(s)
            s_prime, r, done = env.step(a)
            agent.update_table((s,a,r,s_prime))
            s = s_prime
        agent.anneal_eps()

    agent.show_table()
```

Results



```
[[3. 3. 0. 0. 1. 2. 3.]
 [3. 3. 0. 2. 2. 2. 3.]
 [2. 3. 0. 1. 0. 2. 3.]
 [2. 2. 2. 1. 0. 3. 3.]
 [3. 1. 0. 1. 0. 2. 0.]]
```



Outlines

1. MC Control
2. TD Control - SARSA
3. **TD Control - Q Learning**
    a. **On-policy vs Off-policy**

TD Control - Q-learning

The second TD control approach is **Q-learning**.

Q-learning is quite popular, so much so that even people who aren't very familiar with Reinforcement Learning might have come across it casually.

In 2015, the convergence of deep learning and Q-learning gave rise to Deep Q Network (DQN), a groundbreaking paper published in Nature, and this development ultimately paved the way for AlphaGo.

Both Q-learning and SARSA fall under the category of TD control methods. So, what sets them apart? To grasp this distinction, it's essential to first know the concepts of **off-policy** and **on-policy**.

Examples

Let's consider that you're learning League of Legends. If you've never played it before, there are two approaches you can think about for learning the game:

1. **Learning by playing the game yourself**: You learn by playing the game on your own.
2. **Observing a skilled friend play**: You watch a skilled friend play the game and learn from their actions.

You can label the first approach as "**on-policy**," and the second as "**off-policy**."

Target Policy and Behavior Policy

The **target policy** is the policy that you aim to reinforce or improve. It's the policy under consideration, and as it continuously gets updated, it becomes stronger over time.

On the other hand, the **behavior policy** is the policy used when interacting with the environment in practical terms.

On-policy: When the target policy and the behavior policy are the same (e.g., when you personally play the game).

Off-policy: When the target policy and the behavior policy are different (e.g., learning from a friend's experience).

Target Policy and Behavior Policy

On-Policy:

In on-policy learning, the behavioral policy you use when playing the game is the same as the target policy you have in mind for learning.

Off-Policy:

In off-policy learning, the behavioral policy your friend uses when playing the game is different from the target policy you have in mind for learning.

Is Off-policy a Supervised Learning?

Advantages of Off-policy

1. **Reusability of Past Experiences**: Off-policy learning allows for the reuse of past experiences, enabling more efficient learning from historical data.

2. **Utilizing Human Expertise**: It provides the opportunity to incorporate human expertise and experiences into the learning process, which can enhance the quality of the learned policies.

3. **Supports One-to-Many and Many-to-One Learning**: Off-policy learning can handle scenarios where one policy can be learned from multiple different behavior policies (one-to-many) or where multiple different target policies can be learned from a single behavior policy (many-to-one). This flexibility is valuable in various applications.

TD Control - Q-learning

Just as SARSA relies on Bellman's expected equation (version 1) to utilize the Q function, Q-learning employs Bellman's optimality equation (version 1) to determine the TD target.

Bellman's optimality equation (Ver. 1) $q_*(s,a) = \mathbb{E}_{s'}\left[r + \gamma\, max_{a'}\, q_*(s', a')\right]$

SARSA
$$Q(S,A) \leftarrow Q(S,A) + \alpha(\underbrace{R + \gamma Q(S',A')}_{\text{TD target}} - Q(S,A))$$

Q-learning
$$Q(S,A) \leftarrow Q(S,A) + \alpha(\underbrace{R + \gamma max_{A'} Q(S',A')}_{\text{TD target}} - Q(S,A))$$

TD Control - Q-learning

In SARSA, the behavior policy and target policy coincide, whereas in Q-learning, they diverge. The behavior policy employs ε-greedy for exploration, while the target policy adopts a greedy policy, selecting the action with the highest Q-value.

	SARSA	Q-learning
Behavior Policy	ε-greedy	**ε-greedy**
Target Policy	ε-greedy	**greedy**

Implementation

Same as SARSA except update_table function and anneal_eps function.

```
def update_table(self, transition):
    s, a, r, s_prime = transition
    x,y = s
    next_x, next_y = s_prime
    a_prime = self.select_action(s_prime)
    self.q_table[x,y,a] = self.q_table[x,y,a] + 0.1 * (r + np.amax(self.q_table[next_x,next_y,:]) - self.q_table[x,y,a])
```