CS4910: Deep Learning for Robotics

David Klee
klee.d@northeastern.edu

T/F, 3:25-5:05pm
Behrakis Room 204

https://www.ccs.neu.edu/home/dmklee/cs4910_s22/index.html

https://piazza.com/northeastern/spring2022/cs4910a/home

Reinforcement Learning:
Q-Learning

Some slide material taken from CS7180 F18
For more details see: Sutton & Barto 2018

Today's Agenda

1. HW2 Questions
2. MDPs and Environment
3. TD-Learning
4. Q-Learning

Reinforcement Learning (RL) is learning through trial-and-error *without a model of the world*

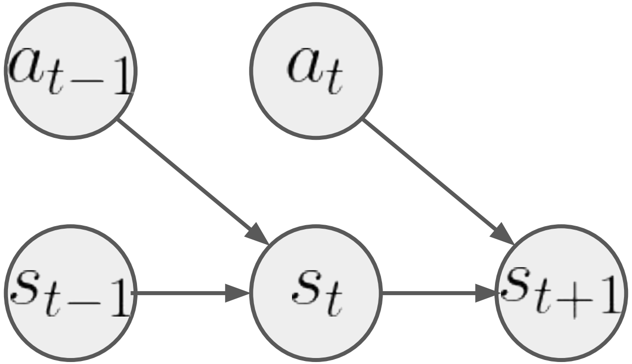
Instead, we learn value functions...

$$V(s) = \mathbb{E}[r|s]$$
$$Q(a) = \mathbb{E}[r|a]$$

$$V^{\pi}(s_t) = \mathbb{E}_{\pi}[G_t|s_t]$$
$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi}[G_t|s_t, a_t]$$

To extend to temporal sequences, we place value on the return (i.e. future rewards) and the value is defined by a policy (i.e. how future actions are chosen)

Markov Decision Process



MDP = < S, A, R, T, gamma >

State: state of the system

Action: action space

R: reward function R(s,a,s’)

T: transition function T(s’|s,a)

Gamma: discount factor

The objective of RL is to learn a policy that maximizes discounted future rewards

Deterministic policy maps state to action

$$\pi(s) \rightarrow a$$

Stochastic policy assigns probability to each action

$$\pi(a|s) \rightarrow (0, 1)$$

The key insight here is that the policy will only be effective if the Markov property holds: the current state contains all information needed to make a decision

Example: express driving to the store as MDP

Multiple levels of abstraction are possible

Ensure that the Markov property holds

Implementing MDP as an Environment

Attributes:

- observation_space
- action_space

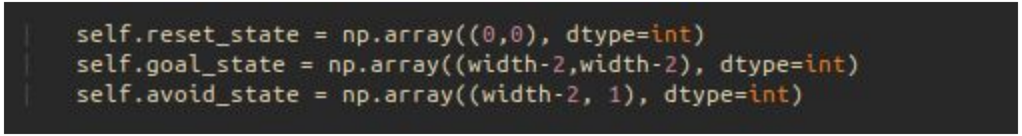
Methods:

- reset -> obs
- step (action) -> obs, reward, done, info
- render -> None

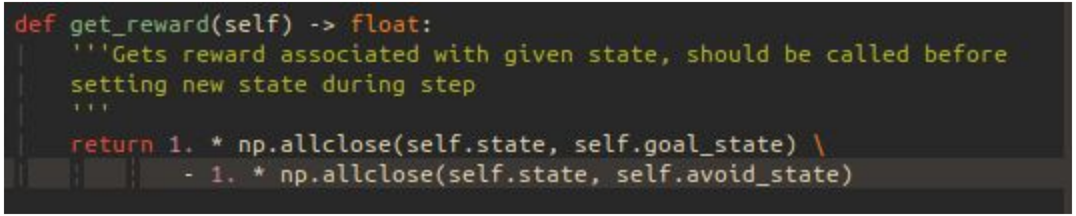
Grid world environment

See ‘examples/tabular_rl.py’

Let’s add an avoid state, where the agent receives a reward of -1. We will place it in the same column as the goal state



```
self.reset_state = np.array((0,0), dtype=int)
self.goal_state = np.array((width-2,width-2), dtype=int)
self.avoid_state = np.array((width-2, 1), dtype=int)
```



```
def get_reward(self) -> float:
    '''Gets reward associated with given state, should be called before
    setting new state during step
    '''
    return 1. * np.allclose(self.state, self.goal_state) \
            - 1. * np.allclose(self.state, self.avoid_state)
```

Value of a state is the expected *return* when following a given policy from the state

$$\boxed{V^{\pi}(s) = \mathbb{E}_{\pi}[G_t | s_t = s]}$$

Return (G) is the sum of future discounted rewards

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \end{aligned}$$

Exercise: calculate returns for trajectories

Trajectory 1

s_t	r_{t+1}	G_t
s_1	-0.1	
s_2	0.5	
s_3	0	
s_4	0	
s_5	-1	

Trajectory 2

s_t	r_{t+1}	G_t
s_1	1	
s_3	0	
s_4	0.1	
s_5	-2	

Exercise: calculate returns for trajectories

Trajectory 1

s_t	r_{t+1}	G_t
s_1	-0.1	$-0.1+\gamma 0.5+\gamma^2 0+\gamma^3 0+\gamma^4(-2)$
s_2	0.5	$0.5+\gamma 0+\gamma^2 0+\gamma^3(-2)$
s_3	0	$0+\gamma 0+\gamma^2(-2)$
s_4	0	$0+\gamma(-1)$
s_5	-1	-1

Trajectory 2

s_t	r_{t+1}	G_t
s_1	1	$1 + \gamma 0+\gamma^2 0.1+\gamma^3(-2)$
s_3	0	$0+\gamma 0.1+\gamma^2(-2)$
s_4	0.1	$0.1+\gamma(-2)$
s_5	-2	-2

$$V^\pi(s_3) = ?$$

Exercise: calculate returns for trajectories

Trajectory 1

s_t	r_{t+1}	G_t
s_1	-0.1	$-0.1+\gamma 0.5+\gamma^2 0+\gamma^3 0+\gamma^4(-1)$
s_2	0.5	$0.5+\gamma 0+\gamma^2 0+\gamma^3(-1)$
s_3	0	$0+\gamma 0+\gamma^2(-1)$
s_4	0	$0+\gamma(-1)$
s_5	-1	-1

Trajectory 2

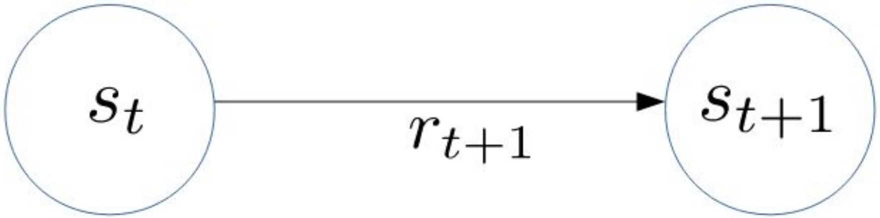
s_t	r_{t+1}	G_t
s_1	1	$1 + \gamma 0+\gamma^2 0.1+\gamma^3(-2)$
s_3	0	$0+\gamma 0.1+\gamma^2(-2)$
s_4	0.1	$0.1+\gamma(-2)$
s_5	-2	-2

Do you notice any pattern in the calculation of returns?

$$V^\pi(s_3) = ?$$

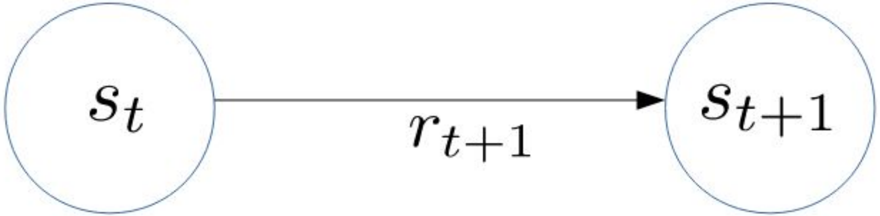
$$V^\pi(s_3) = \frac{1}{2}[\gamma 0.1 + \gamma^2(-1) + \gamma^2(-2)]$$

More efficient value function calculations with dynamic programming



How can we express the value function of s_t in terms of the value function of s_{t+1}?

More efficient value function calculations with dynamic programming



$$V^\pi(s_t) = \mathbb{E}_\pi\left[G_t | s_t\right]$$

$$V^\pi(s_t) = \mathbb{E}_\pi\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t\right]$$

$$V^\pi(s_t) = \mathbb{E}_\pi\left[r_{t+1} + \gamma G_{t+1} | s_t\right]$$

$$V^\pi(s_t) = \mathbb{E}_\pi\left[r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t\right]$$

Bellman Update Equation

Express in terms of $\pi(a|s)$ and $p(s', r|s, a)$:

$$V^{\pi}(s_t = s) = \mathbb{E}_{\pi}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1})|s_t\right]$$

$$V^{\pi}(s_t = s) = \sum_{a} \pi(a|s) \sum_{s', r} p(s', r|s, a)\left[r_{t+1} + \gamma V^{\pi}(s_{t+1} = s')\right]$$

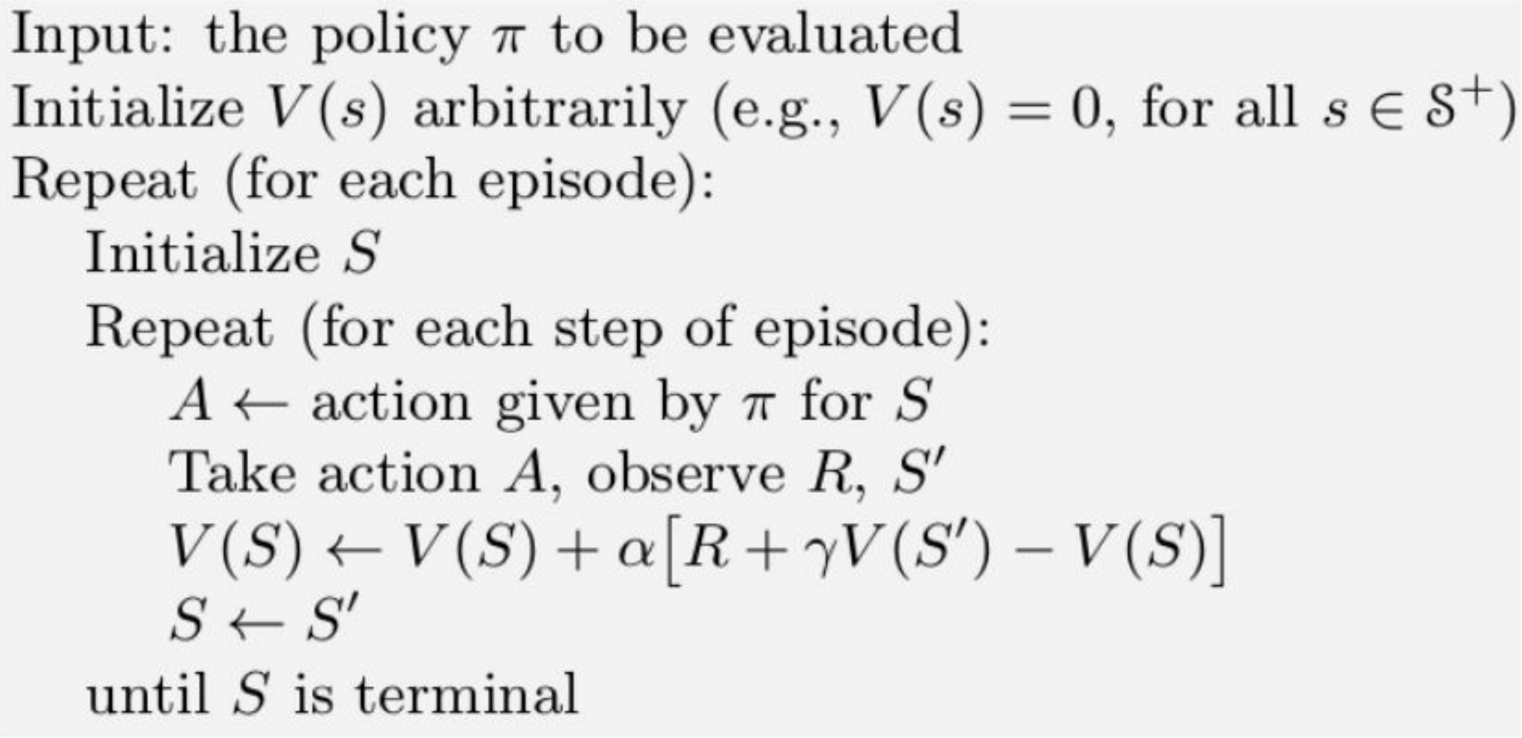
Why is it impractical to calculate value functions like this?

Learning without transition model using temporal difference learning (TD-learning)

$$V^\pi(s_t) \leftarrow (1-\alpha)V^\pi(s_t) + \alpha[r_{t+1} + \gamma V^\pi(s_{t+1})]$$

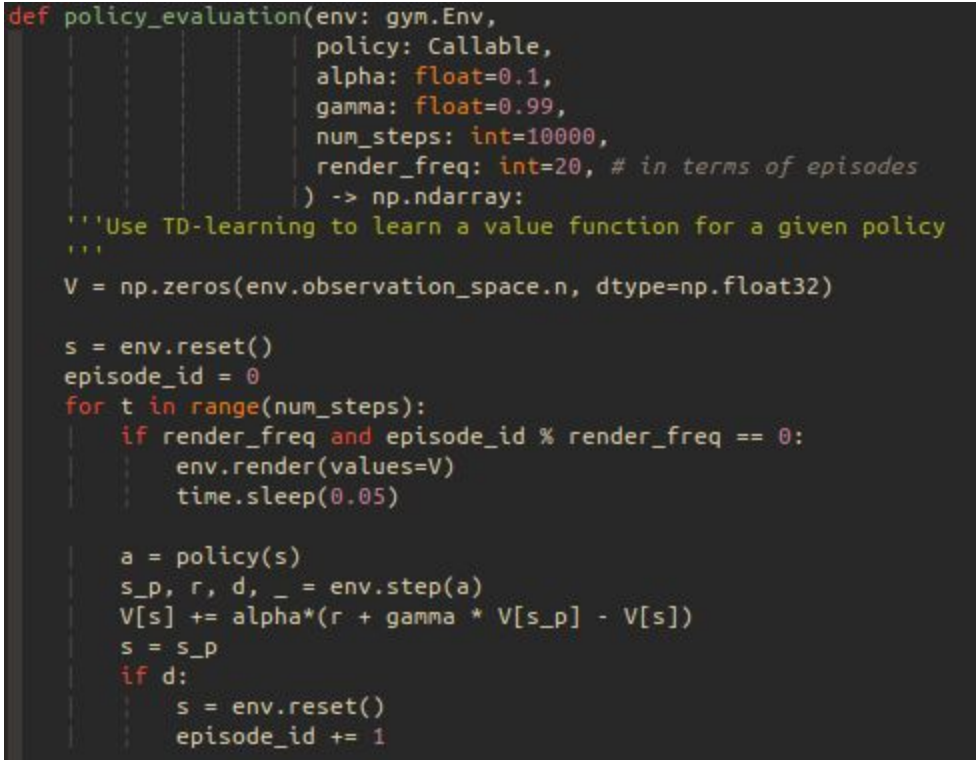
$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha\underbrace{[r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)]}_{\textbf{TD error}}$$

Policy evaluation: learn value function associated with a given policy



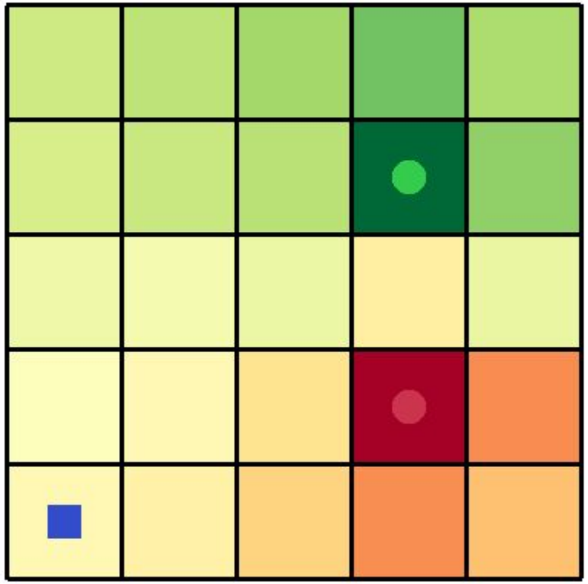
```
Input: the policy $\pi$ to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0$, for all $s \in \mathcal{S}^+$)
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Implement policy evaluation for 2d grid world



```
def policy_evaluation(env: gym.Env,
                      policy: Callable,
                      alpha: float=0.1,
                      gamma: float=0.99,
                      num_steps: int=10000,
                      render_freq: int=20, # in terms of episodes
                     ) -> np.ndarray:
    '''Use TD-learning to learn a value function for a given policy
    '''
    V = np.zeros(env.observation_space.n, dtype=np.float32)

    s = env.reset()
    episode_id = 0
    for t in range(num_steps):
        if render_freq and episode_id % render_freq == 0:
            env.render(values=V)
            time.sleep(0.05)

        a = policy(s)
        s_p, r, d, _ = env.step(a)
        V[s] += alpha*(r + gamma * V[s_p] - V[s])
        s = s_p
        if d:
            s = env.reset()
            episode_id += 1
```

Result of running policy_evaluation



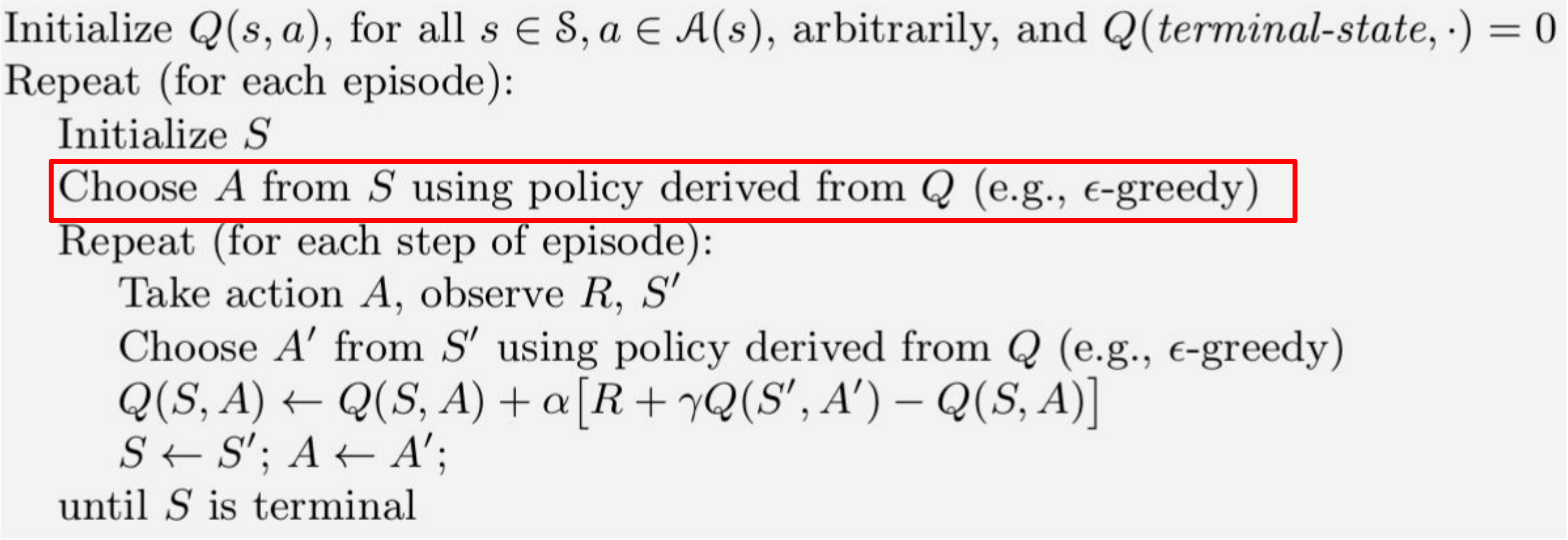
Extending to policy evaluation to action-value function

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha[r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)]$$

⇓

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha[r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t)]$$

Extending to policy evaluation to action-value function

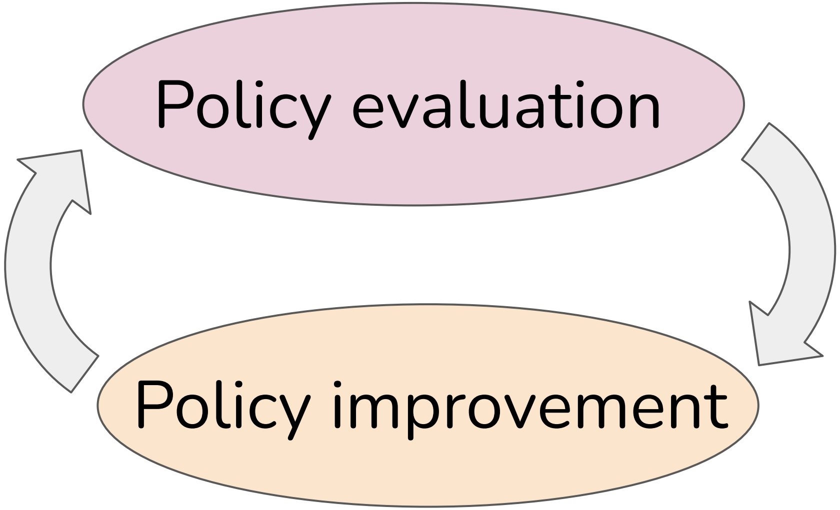


```
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma Q(S', A') - Q(S, A)\right]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

Policy Evaluation: given policy, estimate action-value function based on trajectories

Policy Improvement: generate a new policy by selection actions that have higher values in a given state

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

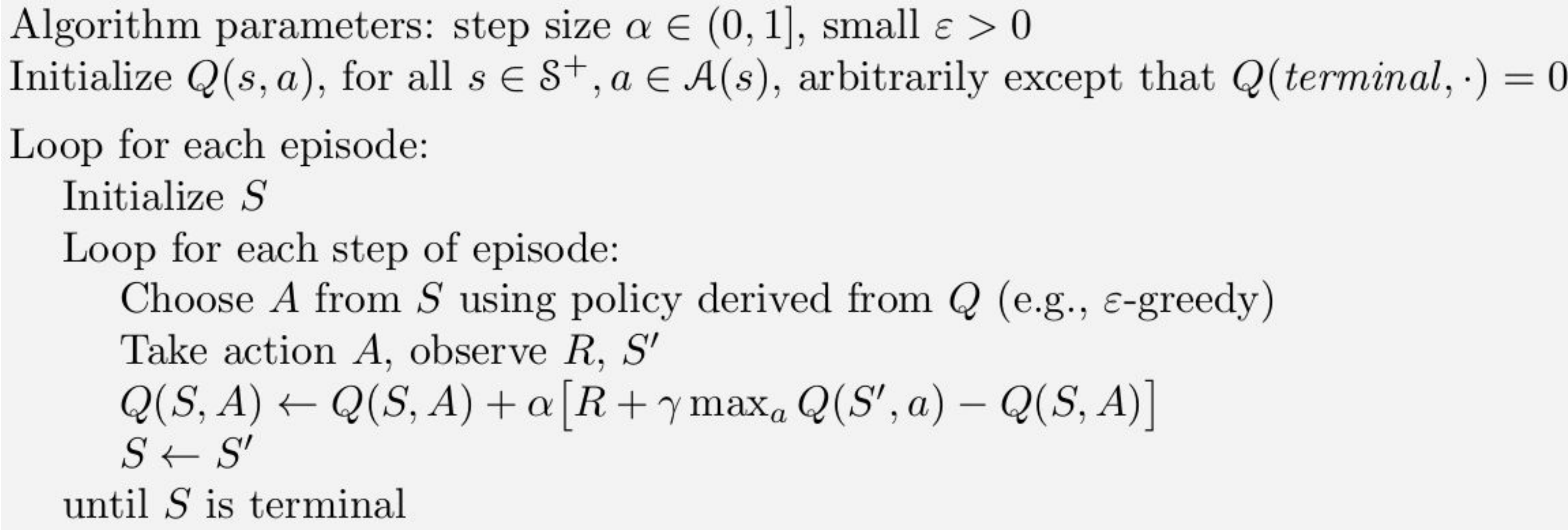

Policy evaluation

Policy improvement

Q-learning update for off-policy learning

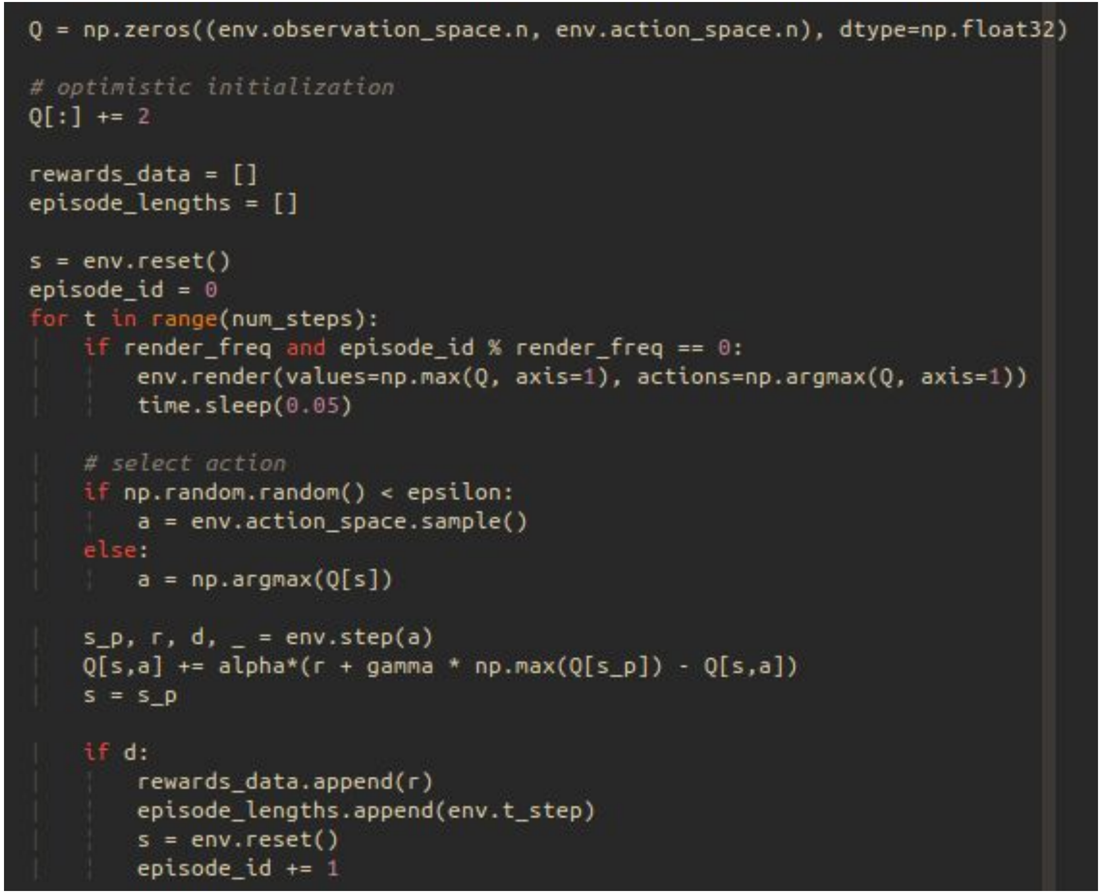
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]$$

Q-learning Algorithm



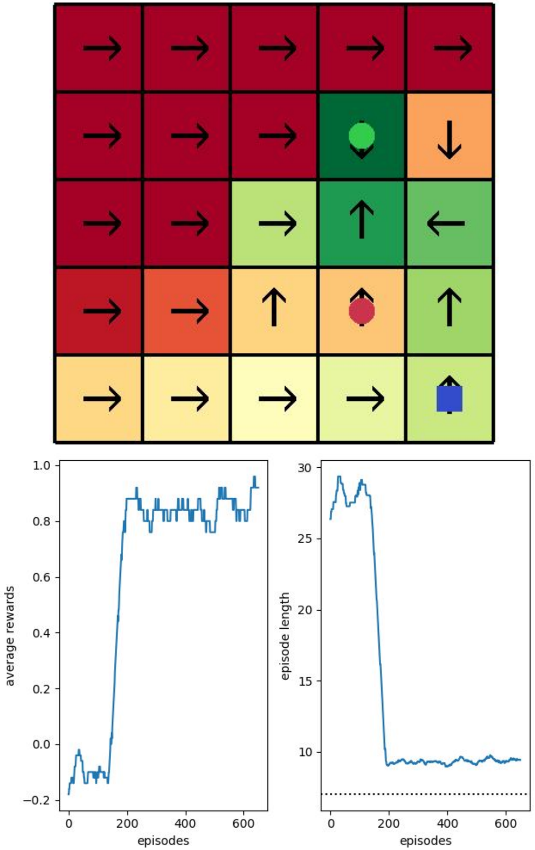
```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Implement Q-learning for 2d navigation problem

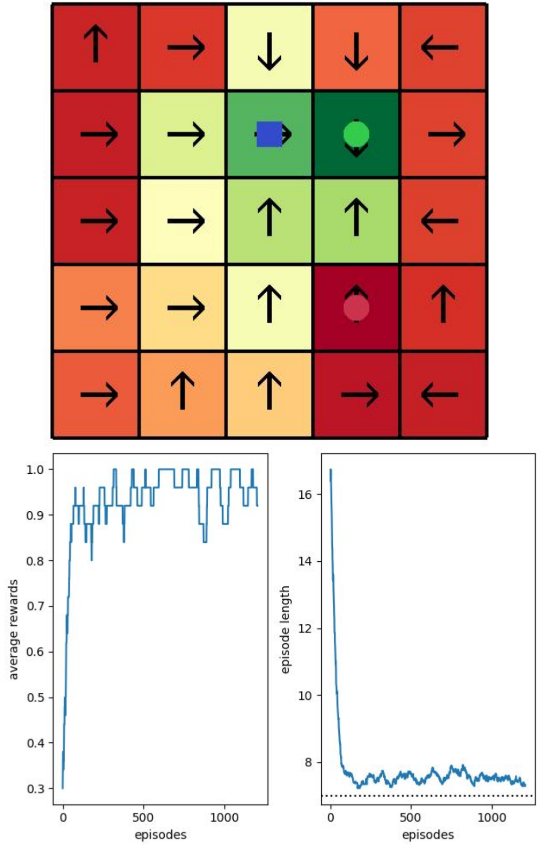


```
Q = np.zeros((env.observation_space.n, env.action_space.n), dtype=np.float32)

# optimistic initialization
Q[:] += 2

rewards_data = []
episode_lengths = []

s = env.reset()
episode_id = 0
for t in range(num_steps):
    if render_freq and episode_id % render_freq == 0:
        env.render(values=np.max(Q, axis=1), actions=np.argmax(Q, axis=1))
        time.sleep(0.05)

    # select action
    if np.random.random() < epsilon:
        a = env.action_space.sample()
    else:
        a = np.argmax(Q[s])

    s_p, r, d, _ = env.step(a)
    Q[s,a] += alpha*(r + gamma * np.max(Q[s_p]) - Q[s,a])
    s = s_p

    if d:
        rewards_data.append(r)
        episode_lengths.append(env.t_step)
        s = env.reset()
        episode_id += 1
```

Implement Q-learning for 2d navigation problem



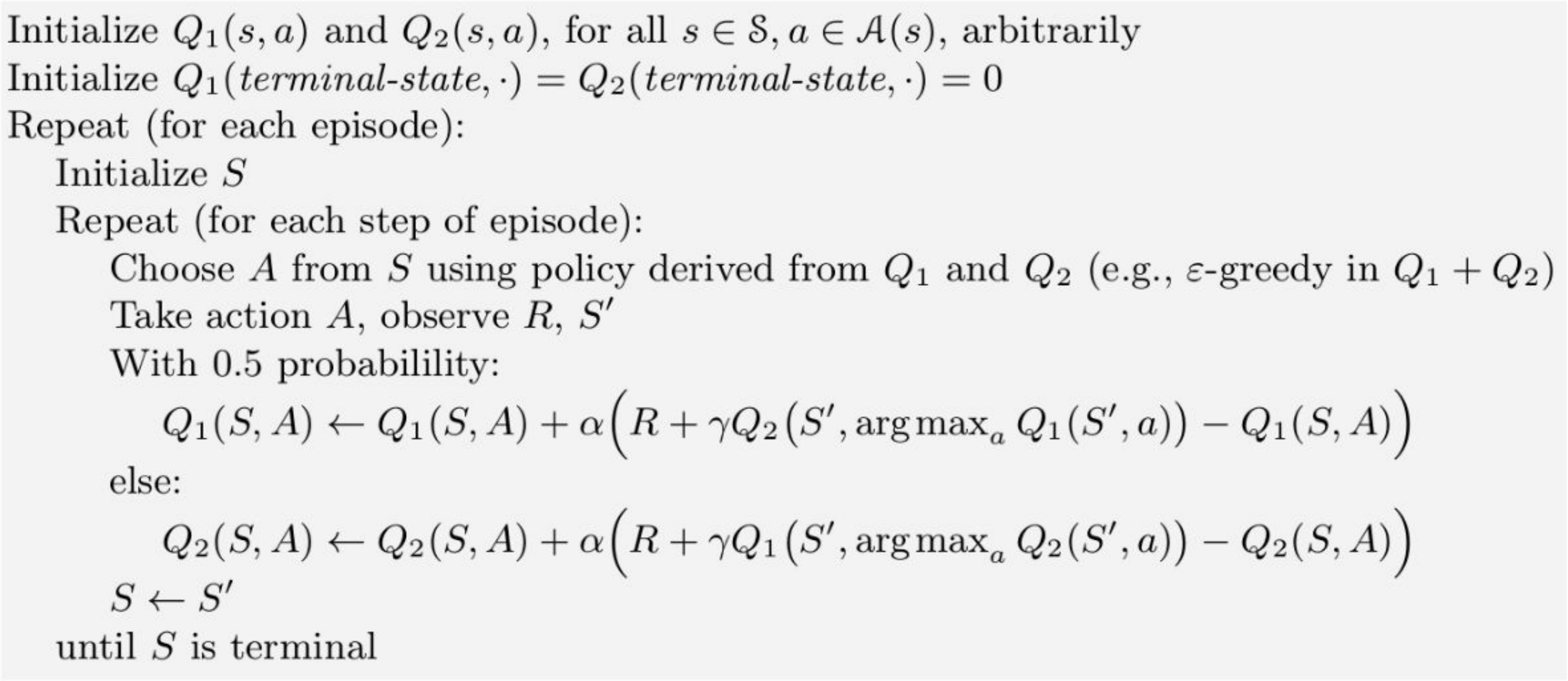
Tuning exploration or using optimistic q-function initialization speeds up learning and finds optimal path



Maximization bias of Q-learning

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]$$

Double Q-learning reduces overestimation of q-targets and improves learning



```
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily
Initialize $Q_1(\textit{terminal-state}, \cdot) = Q_2(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q_1$ and $Q_2$ (e.g., $\varepsilon$-greedy in $Q_1 + Q_2$)
        Take action $A$, observe $R$, $S'$
        With 0.5 probabilility:
            $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
        else:
            $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
        $S \leftarrow S'$
    until $S$ is terminal
```

Next Class

1. Tabular to Deep Q-learning
2. Debugging DQN and relevant hyperparameters
3. Case studies on formulating environments

Survey to provide feedback

https://forms.gle/a2KasSG5UsPzVzqQ6