Day 2. Tabular Q-learning

SAMSUNG AI

Reinforcement Learning

June 17, 2022

Jaeuk Shin, Mingyu Park

CORE
Control + Optimization Research Lab

Tabular Q-learning

With Q, we don't need a policy!

$$Q^*(s,a) = \underbrace{r(s,a)}_{\text{immediate reward}} + \gamma \underbrace{\sum_{s' \in S} p(s'|s,a) v^*(s')}_{\text{optimal value of next state}}$$

$$= r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \max_{a' \in A} Q^*(s',a')$$

- Define the Bellman operator $\mathcal{T}$ for Q-functions by

$$(\mathcal{T}Q)(\boldsymbol{s},\boldsymbol{a}) := \boxed{r(\boldsymbol{s},\boldsymbol{a}) + \gamma \sum_{\boldsymbol{s}' \in S} p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) \max_{\boldsymbol{a}' \in A} Q(\boldsymbol{s}',\boldsymbol{a}').}$$

Then, it is a monotone contraction mapping.

- Bellman equation:

$$Q = \mathcal{T}Q.$$

Q-Learning (tabular):

Initialize Q;

Replace model with data!

1. Take some action and $\boxed{\text{observe } (s,a,s',r);}$
2. Set $Q(s,a) \leftarrow (1-\alpha) \underbrace{Q(s,a)}_{\text{old estimate}} + \alpha \underbrace{\boxed{\left[r + \gamma \max_{a'} Q(s',a')\right]}}_{\text{new estimate}};$
3. Repeat until convergence;

Can you see the difference?

Tabular Q-learning

MDP definition

```python
class MyEnv:
    num_actions = 4

    def __init__(self):
        pass

    def reset(self):
        pass

    def step(self, action):
        pass
```

s_t is kept internally, and is updated in **step** method

sample an initial state $s_0 \sim \rho_0(s)$

agent-env interaction:
$s_{t+1} \sim p(\cdot|s_t, a_t), \;\; r_t = r(s_t, a_t)$

Tabular Q-learning

Example - Pendulum

MDP as a Python class(gym.Env)

```python
class PendulumEnv(gym.Env):
metadata = {
    'render.modes': ['human', 'rgb_array'],
    'video.frames_per_second': 30
}

def __init__(self, g=10.0):
    self.max_speed = 8
    self.max_torque = 2.
    self.dt = .05
    self.g = g
    self.m = 1.
    self.l = 1.
    self.viewer = None

    high = np.array([1., 1., self.max_speed], dtype=np.float32)
    self.action_space = spaces.Box(
        low=-self.max_torque,
        high=self.max_torque, shape=(1,),
        dtype=np.float32
    )
```

https://github.com/openai/gym/blob/master/gym/envs/classic_control/pendulum.py

CORE Control + Optimization Research Lab

Tabular Q-learning

Example - Pendulum

action input a_t (torque applied at t)

state s_t: $s_t = (\theta_t, \dot{\theta}_t)$

compute reward $r_t = r(s_t, a_t)$

compute the next state s_{t+1}

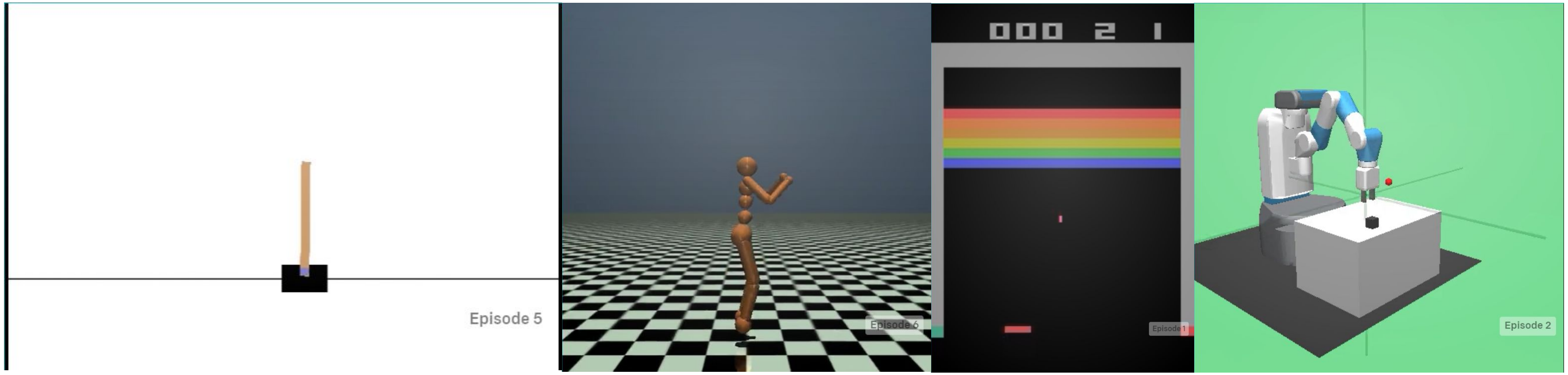
return next state s_{t+1} & reward r_t

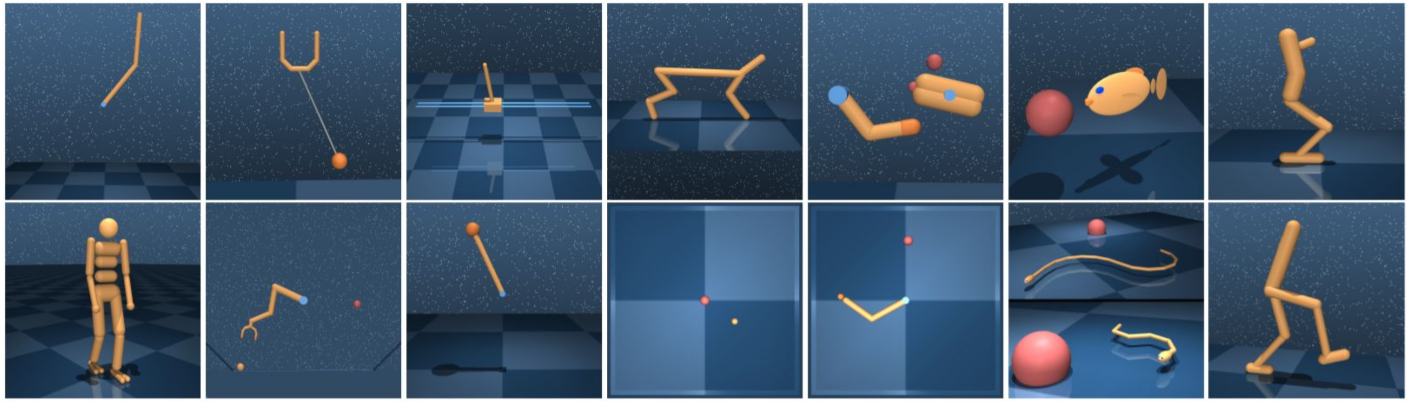
```
def step(self, u):
    th, thdot = self.state  # th := theta

    g = self.g
    m = self.m
    l = self.l
    dt = self.dt

    u = np.clip(u, -self.max_torque, self.max_torque)[0]
    self.last_u = u  # for rendering
    costs = angle_normalize(th) ** 2 + .1 * thdot ** 2 + .001 * (u ** 2)

    newthdot = thdot + (-3 * g / (2 * l) * np.sin(th + np.pi) + 3. / (m * l ** 2) * u) * dt
    newth = th + newthdot * dt
    newthdot = np.clip(newthdot, -self.max_speed, self.max_speed)

    self.state = np.array([newth, newthdot])
    return self._get_obs(), -costs, False, {}
```

https://github.com/openai/gym/blob/master/gym/envs/classic_control/pendulum.py

Tabular Q-learning

More examples - OpenAI Gym, Deepmind Control Suite, etc.





Q-learning - Implementation

Algorithm Implementation

```python
class QTable:
    def __init__(self, num_states, num_actions, gamma=0.99):
        self.gamma = gamma
        self.Q = np.zeros(shape=(num_states, num_actions))

    def update(self, state, action, reward, next_state, alpha):
        target = reward + self.gamma * np.max(self.Q[next_state]) - self.Q[state, action]
        self.Q[state, action] += alpha * target

    def act(self, state):
        return np.argmax(self.Q[state])
```

store Q-function as a table

Q-learning update!

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t(s_t, a_t) \underbrace{\left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\right)}_{\text{error}}$$

greedy action $a_t = \arg\max_a Q(s_t, a)$

Q-learning - Implementation

Complete Outline

```
learner = QTable(num_states=env.observation_space.n, num_actions=env.action_space.n, gamma=gamma)
rollout_len = 1000000
visit_count = np.zeros(shape=(num_states, num_actions))      # save visit counts N(s, a) of all state-action pairs
alpha = VisitCountStepsizeSchedule(deg=0.5001)
epsilon = LinearExplorationSchedule(rollout_len, final_epsilon=0.4)

s = env.reset()
for t in tqdm(range(rollout_len + 1)):
    u = np.random.rand()
    if u < epsilon(t):
        a = env.action_space.sample()
    else:
        a = learner.act(state=s)
    s_next, r, _, _ = env.step(action=a)
    n = visit_count[s, a]
    learner.update(state=s, action=a, reward=r, next_state=s_next, alpha=alpha(n))
    visit_count[s, a] += 1
    s = s_next
```

stepsize rule : $\alpha_t(s,a) = \frac{1}{n_t(s,a)^d}$

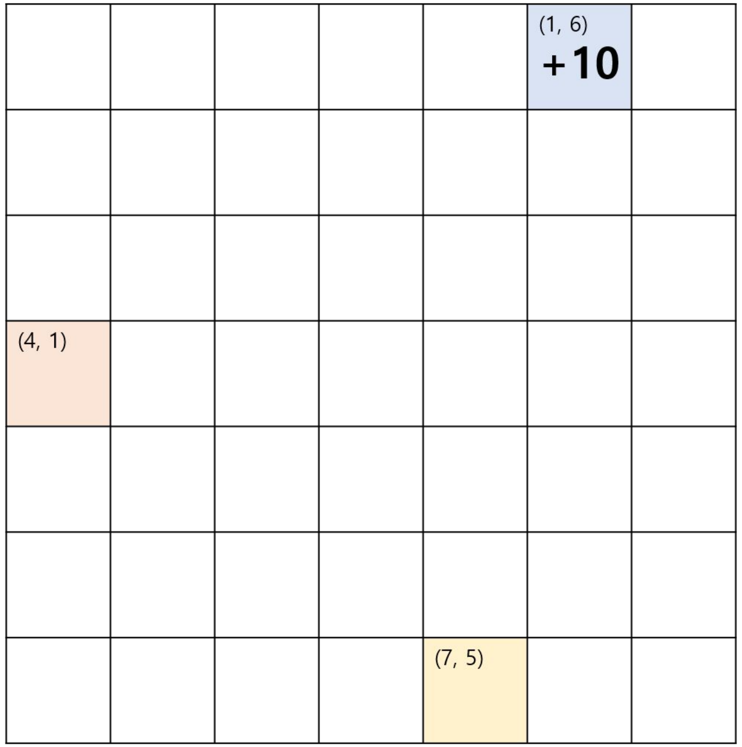
$1/2 < d < 1$

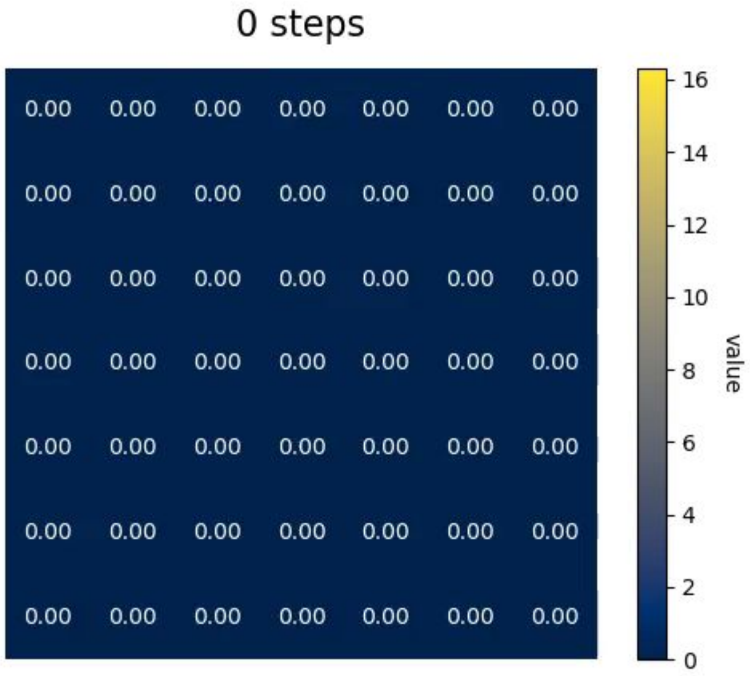
exploration strategy : start with large ε, and decrease it.

ε-greedy action selection

Practice1 - GridWorld

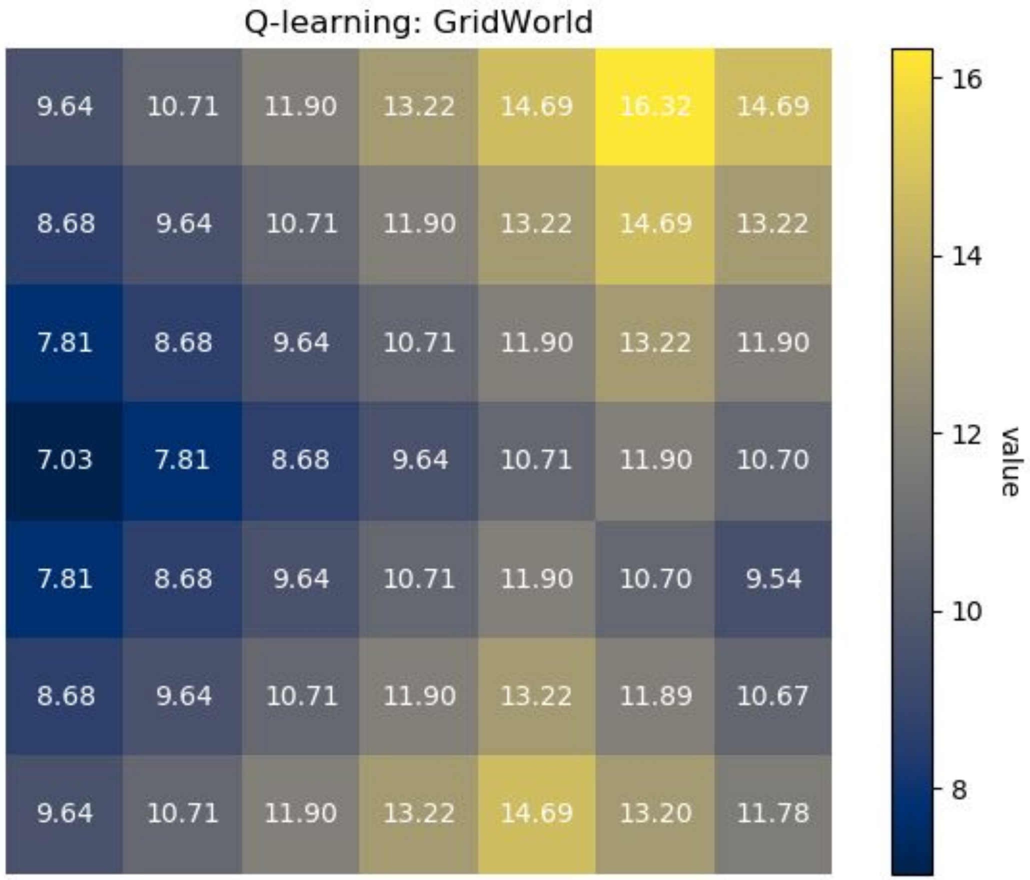
In Day 1, we learned how to compute the value function of GridWorld via **value iteration** when the model is completely **known**.

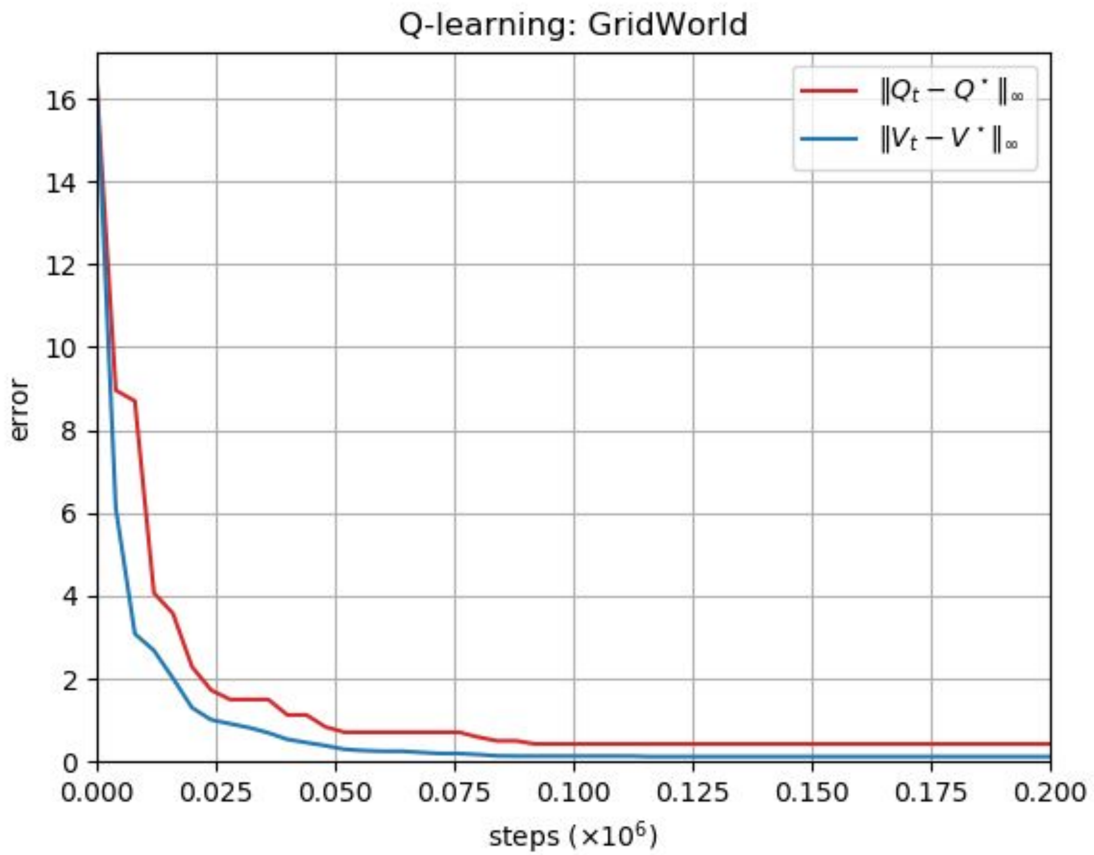




CORE
Control + Optimization Research Lab

Practice1 - GridWorld



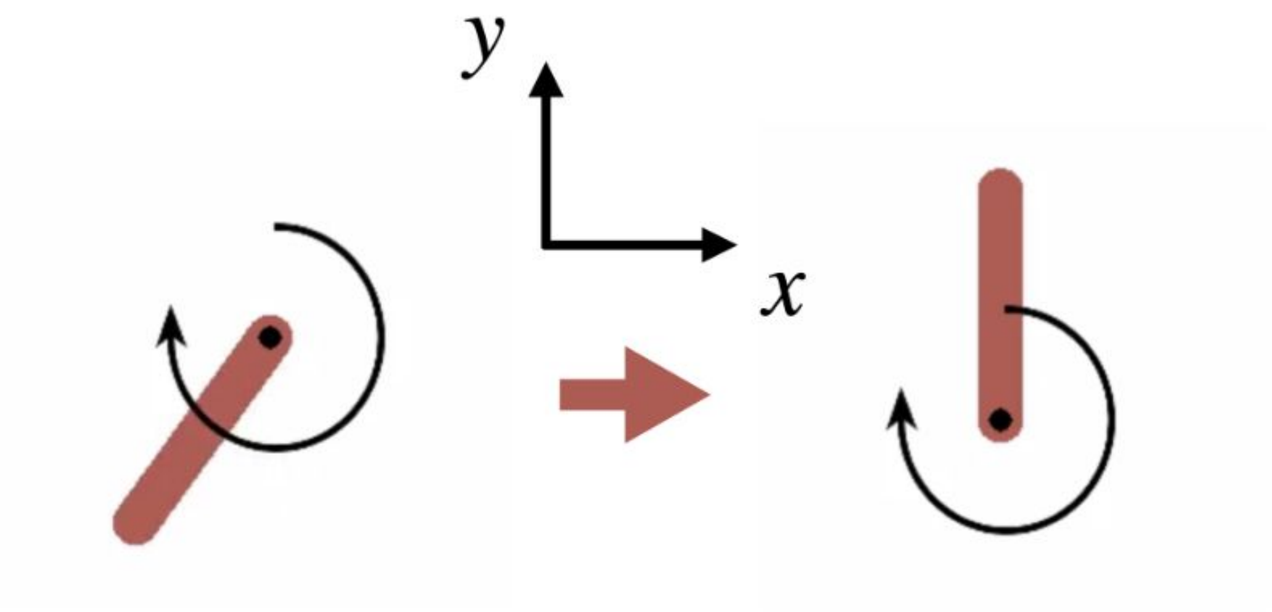


Practice2 – Discrete Pendulum

$$s := (x, y, \dot{\theta}) \quad \begin{cases} x = \cos\theta : \text{x coordinate of the pendulum tip} \\ y = \sin\theta : \text{y coordinate of the pendulum tip} \\ \dot{\theta} \qquad\quad : \text{Angular velocity} \end{cases} , \qquad a := \ddot{\theta},$$

Where the origin is set to the joint of the pendulum, and $-\pi \leq \theta \leq \pi$ as $\theta = 0$ is set to the $+y$ direction.
Finally, as control objective is $\theta = \dot{\theta} = 0$, we will give the reward of

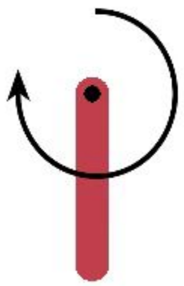
$$r := -(\theta^2 + 0.1\dot{\theta}^2 + 0.001\ddot{\theta}^2).$$



CORE
Control + Optimization Research Lab

Practice2 – Discrete Pendulum

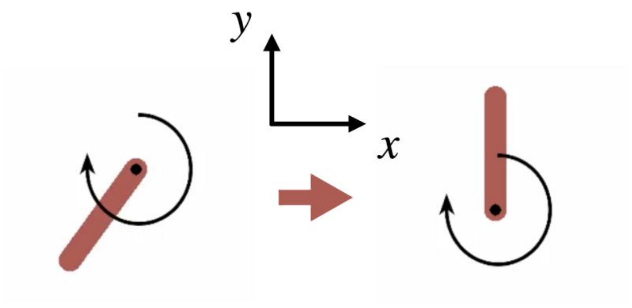
- discretize pendulum dynamics
  - 2-dim. state $(\theta, \dot{\theta})$
  - 1-dim action τ
- goal : apply a torque τ to a joint for swing-up
- 2460 discretized states & 15 discretized actions



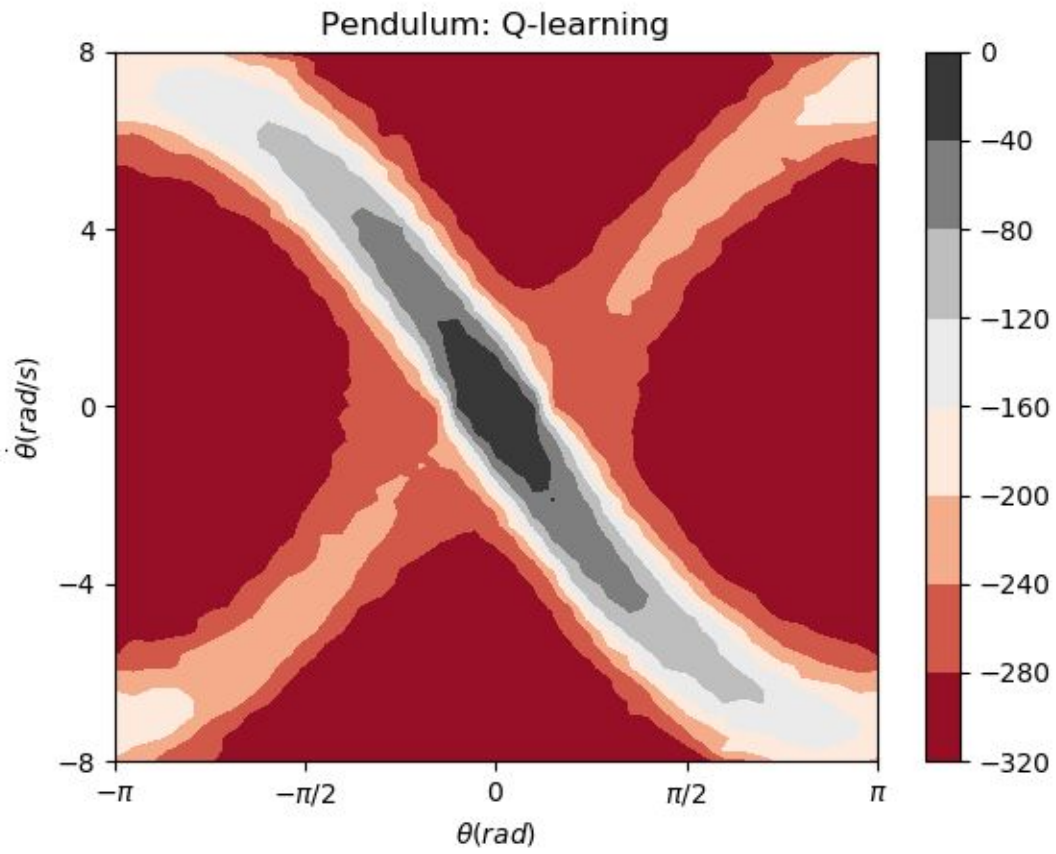
$$s := (x, y, \dot{\theta}) \quad \begin{cases} x = \cos\theta : \text{x coordinate of the pendulum tip} \\ y = \sin\theta : \text{y coordinate of the pendulum tip} \\ \dot{\theta} \qquad\quad : \text{Angular velocity} \end{cases}, \qquad a := \ddot{\theta},$$

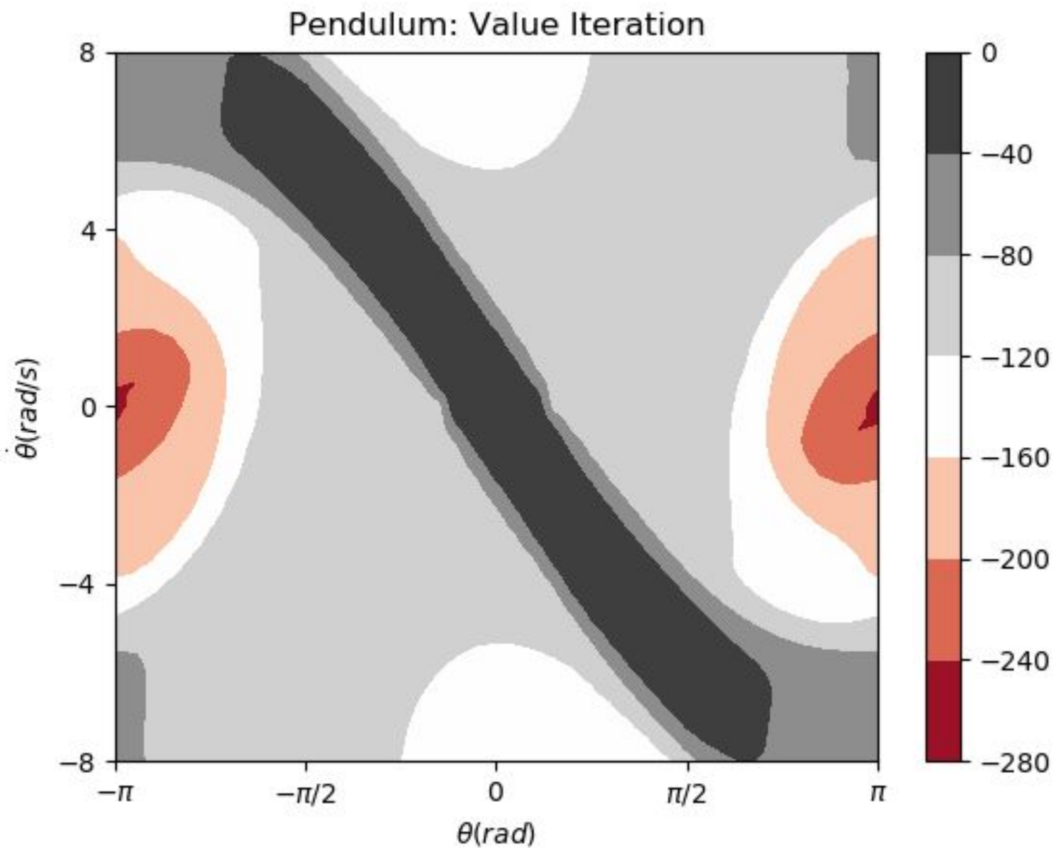
Where the origin is set to the joint of the pendulum, and $-\pi \leq \theta \leq \pi$ as $\theta = 0$ is set to the $+y$ direction.
Finally, as control objective is $\theta = \dot{\theta} = 0$, we will give the reward of

$$r := -(\theta^2 + 0.1\dot{\theta}^2 + 0.001\ddot{\theta}^2).$$



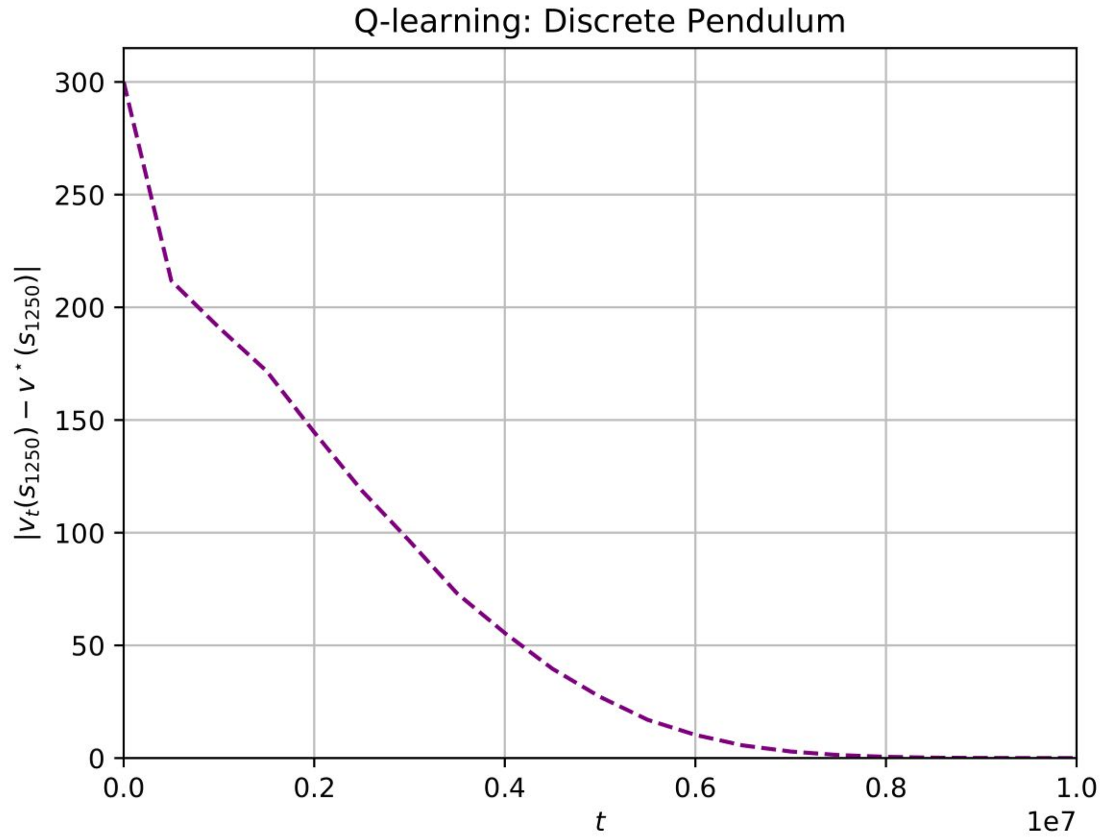
Practice2 – Discrete Pendulum





Control + Optimization Research Lab

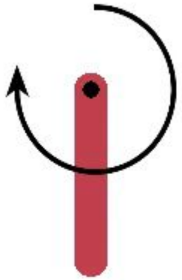
Practice2 – Discrete Pendulum



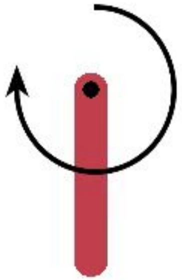
Why Deep Q-Network?

- It seems like Q-learning works well in these examples.
- Why **deep reinforcement learning** then?

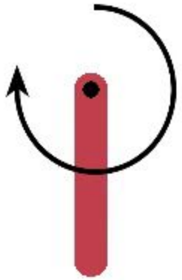
$\Longrightarrow$ For most real-world problems, discretization is not a good strategy...



$n = 160, m = 5$(**coarse**)



$n = 620, m = 10$



$n = 2460, m = 15$(**fine**)