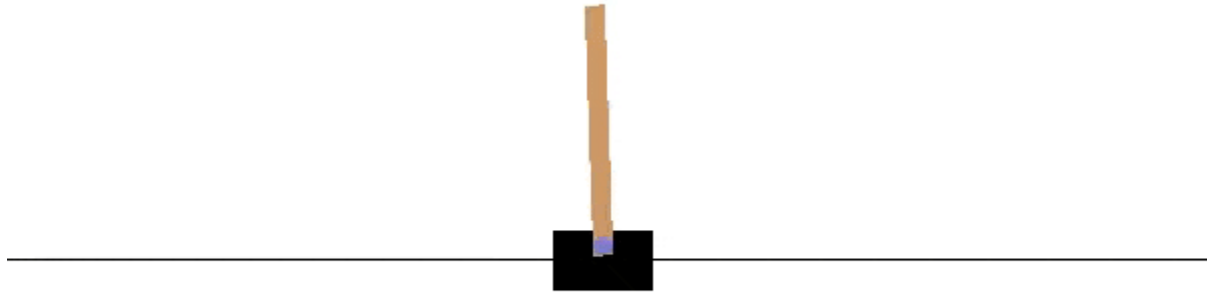
Q Learning Hands-on

Prof. Seungchul Lee

Industrial AI Lab.

CartPole-v0

- A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track.
- The system is controlled by applying a force of +1 or -1 to the cart.
- The pendulum starts upright, and the goal is to prevent it from falling over.
- A reward of +1 is provided for every timestep that the pole remains upright.
- The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.



env.reset()

- The process gets started by calling reset(), which returns an initial observation.

```
import gym

env = gym.make('CartPole-v0')
observation = env.reset() # observation = state

print(observation)
```

```
[ 0.04713564  0.01468255  0.03148437 -0.01945374]
```

```
env.reset()

for _ in range(100):
    env.render()
    action = env.action_space.sample() # your agent here (this takes random actions)
    observation, reward, done, info = env.step(action)

    print(action, observation, reward, done)

env.close()
```

```
1 [-0.06675232 -0.99327954  0.1310663   1.49187636] 1.0 False
1 [-0.08661791 -0.79997362  0.16090383  1.2428277 ] 1.0 False
0 [-0.10261738 -0.99675255  0.18576038  1.58128523] 1.0 False
0 [-0.12255243 -1.19353599  0.21738609  1.92568222] 1.0 True
1 [-0.14642315 -1.00130837  0.25589973  1.70753637] 0.0 True
0 [-0.16644932 -1.1982987   0.29005046  2.06781574] 0.0 True
```

env.step(action)

```
observation, reward, done, info = env.step(action)
```

- observation: an environment-specific object representing your observation of the environment.
- reward: amount of reward achieved by the previous action. The scale varies between environments, but the goal is always to increase your total reward.
- done: whether it’s time to reset the environment again. Most (but not all) tasks are divided up into well-defined episodes, and done being True indicates the episode has terminated. (For example, perhaps the pole tipped too far, or you lost your last life.)

```
for i_episode in range(2):
    observation = env.reset()

    for k in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)

        if done:
            print("Episode finished after {} timesteps".format(k+1))
            break

env.close()
```

Discretized States

```
def map_discrete_state(state):
    pos, vel, ang, anv = state

    idx_pos = np.where(np.histogram(np.clip(pos,-2,2), bins = n_bins_pos, range = (-4,4))[0] == 1)[0][0]
    idx_vel = np.where(np.histogram(np.clip(vel,-2,2), bins = n_bins_vel, range = (-2,2))[0] == 1)[0][0]
    idx_ang = np.where(np.histogram(np.clip(ang,-0.4,0.4), bins = n_bins_ang, range = (-0.4,0.4))[0] == 1)[0][0]
    idx_anv = np.where(np.histogram(np.clip(anv,-2,2), bins = n_bins_anv, range = (-2,2))[0] == 1)[0][0]

    states = np.zeros([n_bins_pos, n_bins_vel, n_bins_ang, n_bins_anv])
    states[idx_pos, idx_vel, idx_ang, idx_anv] = 1

    states = states.reshape(-1,1)

    s = np.where(states == 1)[0][0]

    return s
```

```
n_bins_pos = 10
n_bins_vel = 10
n_bins_ang = 10
n_bins_anv = 10
```

```
print(env.observation_space.low)
print(env.observation_space.high)
```

```
[-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38]
[4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38]
```

Exploration vs. Exploitation, and TD

```
n_states = n_bins_pos*n_bins_vel*n_bins_ang*n_bins_anv
n_actions = 2

Q_table = np.random.uniform(0, 1, (n_states, n_actions))
```

```
# Exploration vs. Exploitation for action
epsilon = 0.1
if np.random.uniform() < epsilon:
    action = np.random.randint(2)
else:
    action = np.argmax(Q_table[s, :])
```

$$\pi(s) = \begin{cases} \max_{a \in A} Q_k(s,a) & \text{with probability } 1-\varepsilon \\ \text{random action} & \text{otherwise} \end{cases}$$

```
# Temporal Difference Update
Q_table[s, action] = (1 - alpha)*Q_table[s, action] + alpha*(reward + gamma*np.max(Q_table[next_s, :]))
```

$$Q_*(s,a) \leftarrow (1-\alpha)\left(Q_*(s,a)\right) + \alpha\left(R_t + \gamma \max_{a'} Q_*\left(s', a'\right)\right)$$

Q Learning

```
alpha = 0.3
gamma = 0.9

Q_table = np.random.uniform(0, 1, (n_states, n_actions))

for episode in range(801):

    done = False
    state = env.reset()

    while not done:

        s = map_discrete_state(state)

        # Exploration vs. Exploitation for action
        epsilon = 0.1
        if np.random.uniform() < epsilon:
            a = env.action_space.sample()
        else:
            a = np.argmax(Q_table[s, :])

        # next state and reward
        next_state, reward, done, _ = env.step(a)

        if done:
            reward = -100
            Q_table[s, a] = reward

        else:
            # Temporal Difference Update
            next_s = map_discrete_state(next_state)
            Q_table[s, a] = (1 - alpha)*Q_table[s, a] + alpha*(reward + gamma*np.max(Q_table[next_s, :]))

        state = next_state

env.close()
```

Q Learning

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε greedy)
    Take action a, observe R, s'
    $Q_*(s,a) \leftarrow (1-\alpha)\left(Q_*(s,a)\right) + \alpha\left(R_t + \gamma \max_{a'} Q_*\left(s', a'\right)\right)$
    $s \leftarrow s'$
until s is terminal

Learned Optimal Policy

```
state = env.reset()

done = False

while not done:

    env.render()

    s = map_discrete_state(state)
    action = np.argmax(Q_table[s,:])

    next_state, _, done, _ = env.step(action)
    state = next_state

env.close()
```