5강.
Temporal Difference(TD) Methods

Contents

- Temporal Difference
- On-Policy TD Methods
- Off-Policy TD Methods
- Code Exercise
- N-Step TD Methods

TD

- 2nd family of methods that learn the optimal v*(s) or q*(s, a) values based on experience
  - No need of a model of the environment

$$\pi(s) \leftarrow \arg\max_{a} Q(s, a)$$

- Combination of Monte Carlo methods and dynamic programming
  - The agent learns from example: S_0, A_0, R_1, S_1, A_1, … , R_T
  - Use bootstrapping

TD

- Monte Carlo methods wait until the return Gt is available before updating Q(s, a):

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-t-1} R_T$$

- TD update Q and π every time the agent takes an action, during the episode without waiting until the end
  - Learning process is constant and uniform. Now we learn every step. This is advantage because the learning at the beginning of the episode influences the policy during the rest of the episode, improving its decision making.

TD

- We keep a table with q-value estimates for each S_t, A_t pair:

$$Q(S_t, A_t)$$

- Remember the bellman equations:

$$q_\pi(s,a) = E_\pi[\, Gt \,|S_t = s, At = a]$$

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\ [\, r + \gamma\, v_\pi(s')]$$

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\ [\, r + \gamma \sum_{a'} \pi(a'|s')\, q_\pi(s',a')\,]$$

TD

- Remember the bootstrapping

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\ [\, r + \gamma \sum_{a'} \pi(a'|s')\ q_\pi(s',a')\ ]$$

$$\begin{cases} r,\ s' : R_{t+1},\ S_{t+1} \\ \pi(a'|s') : A_{t+1} \\ q_\pi(s',a') : Q \end{cases}$$

- $q_\pi(S_t, A_t)$ can also be estimates as:

$$R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$$

TD

- Temporal difference error between the new one and old one
  - Compare the two estimates. Note the new one incorporates real information from the environment

$$Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$

- Estimates are updated based on the temporal difference error:

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

TD

- Similar to the constant-α Monte Carlo:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)]$$

- Estimating $G_t \approx R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ allows us to update $Q(S_t, A_t)$ at the time t+1

- The update moves $Q(S_t, A_t)$ α percent in the direction of $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

Advantage of TD

- Monte Carlo methods need to wait until an episode ends.
  So, no Q(s, a) table update in-between.

- TD method can start to update the Q-value table immediately after taking the first action
  So, the actions taken at the beginning of the episode start influencing the behavior of the agent immediately.

On-policy TD

SARSA

- The name, SARSA comes from the five values (S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}) involved in the update rule:

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

- The ϵ-greedy policy picks, A_{t+1}, to update $Q(S_t, A_t)$
  - $\begin{cases} \text{prob. } \epsilon : & a \sim \text{random} \\ \text{prob. } 1-\epsilon : & a \leftarrow \arg\max_a Q(s,a) \end{cases}$

SARSA

Algorithm 1 SARSA

Input: α learning rate, ϵ random action probability, γ discount factor
$\pi \leftarrow \epsilon$-greedy policy w.r.t $Q(s, a)$
Initialize $Q(s, a)$ arbitrarily, with $Q(terminal, \cdot) = 0$
for episode $\in 1..N$ **do**
    Reset the environment and observe S_0
    $A_0 \sim \pi(S_0)$
    for $t \in 0..T-1$ **do**
        Execute A_t in the environment and observe S_{t+1}, R_{t+1}
        $A_{t+1} \sim \pi(S_{t+1})$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$
    end for
end for
Output: Near optimal policy π and action values $Q(s, a)$

SARSA

Create the environment ¶

```
env = Maze()
```

Create the $Q(s, a)$ table

```
action_values = np.zeros(shape=(5, 5, 4))
```

Create the policy $\pi(s)$

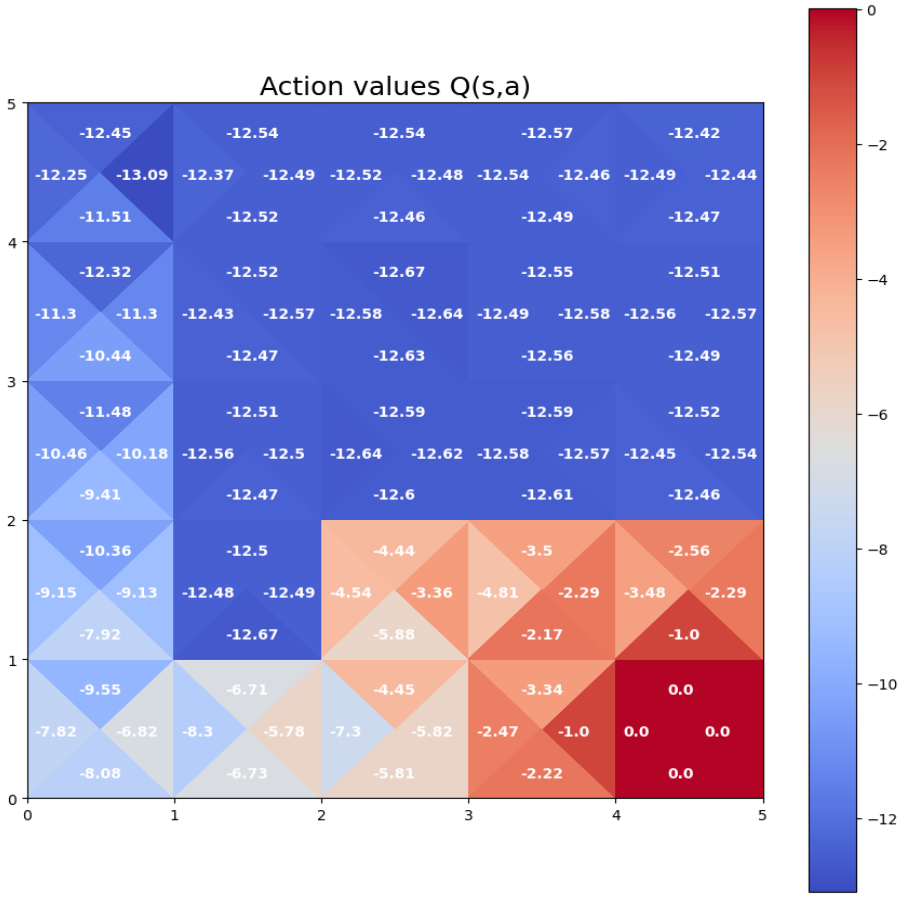
```
def policy(state, epsilon=0.):
    if np.random.random() < epsilon:
        return np.random.randint(4)
    else:
        av = action_values[state]
        return np.random.choice(np.flatnonzero(av == av.max()))
```

SARSA

```
def sarsa(action_values, policy, episodes, alpha=0.1, gamma=0.99, epsilon=0.2):

    for episode in range(1, episodes + 1):
        state = env.reset()
        action = policy(state, epsilon)
        done = False
        while not done:
            next_state, reward, done, _ = env.step(action)
            next_action = policy(next_state, epsilon)

            qsa = action_values[state][action]
            next_qsa = action_values[next_state][next_action]
            action_values[state][action] = qsa + alpha * (reward + gamma * next_qsa - qsa)
            state = next_state
            action = next_action
```

```
sarsa(action_values, policy, 100)
```

SARSA



D	L	D	R	U
D	L	L	L	D
D	D	U	R	L
D	L	R	D	D
R	R	U	R	U

Code Ex.

Off-policy TD

Q-learning

- Off-policy:

Exploration policy ≠ Target Policy

- Exploration policy (how to interact with the environment):

$$b(\mathrm{a} \mid \mathrm{s})$$

- Target Policy (how to participate in the learning process):

$$\pi(\mathrm{s}) \leftarrow \arg\max_{\mathrm{a}} \mathrm{Q}(\mathrm{s}, \mathrm{a})$$

Q-learning

- We'll choose action A_{t+1} according to the target policy
  - Update rule:

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

- Target policy update $Q(S_t, A_t)$ based on the best available action (defines how to update the Q values)

Q-learning

Algorithm 2 Q-Learning

Input: α learning rate, γ discount factor
$\pi \leftarrow$ greedy policy w.r.t $Q(s, a)$
$b \leftarrow$ exploratory policy with coverage of π
Initialize $Q(s, a)$ arbitrarily, with $Q(terminal, \cdot) = 0$
for episode $\in 1..N$ **do**
    Reset the environment and observe S_0
    for $t \in 0..T-1$ **do**
        $A_t \sim b(S_t)$
        Execute A_t in the environment and observe S_{t+1}, R_{t+1}
            $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\,[R_{t+1} + \gamma Q(S_{t+1}, \pi(S_{t+1})) - Q(S_t, At)]$
    end for
end for
Output: Approximately optimal policy π and action values $Q(s, a)$

Q-learning

Create the environment

```
env = Maze()
```

Create the $Q(s, a)$ table

```
action_values = np.zeros((5,5,4))
```

Create the target policy $\pi(s)$

```
def target_policy(state):
    av = action_values[state]
    return np.random.choice(np.flatnonzero(av == av.max()))
```

Create the exploratory policy $b(s)$

```
def exploratory_policy(state):
    return np.random.randint(4)
```

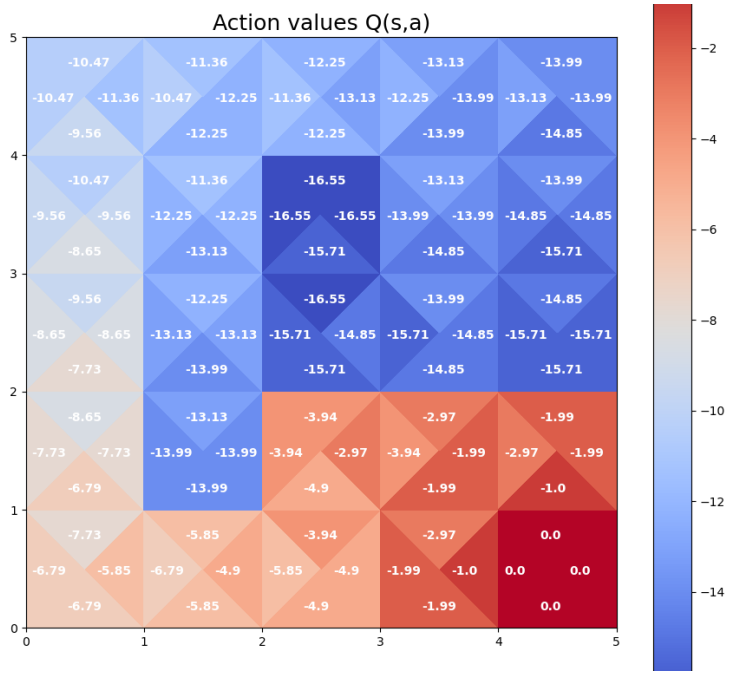
Q-learning

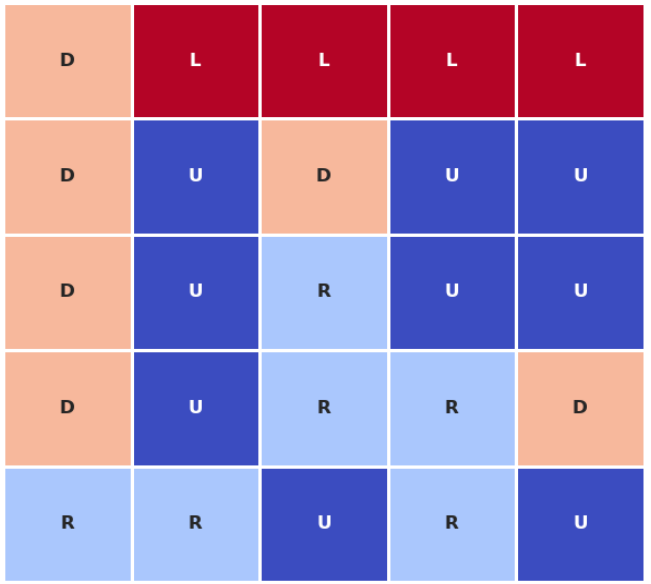
```
def q_learning(action_values, exploratory_policy, target_policy, episodes, alpha=0.1, gamma=0.99):

    for episode in range(1, episodes + 1):
        state = env.reset()
        done = False

        while not done:
            action = exploratory_policy(state)
            next_state, reward, done, _ = env.step(action)
            next_action = target_policy(next_state)

            qsa = action_values[state][action]
            next_qsa = action_values[next_state][next_action]
            action_values[state][action] = qsa + alpha * (reward + gamma * next_qsa - qsa)

            state = next_state
```

```
q_learning(action_values, exploratory_policy, target_policy, 1000)
```

Q-learning

- Note that now in all states, the policy also leads us to the goal.
- It's because the exploratory policy has visited quite frequently, all the states.





Q-learning

- Exploratory policy has been a random policy throughout the entire process giving the agent the opportunity to explore all of these states and get a good idea of the real Q values and optimal action in all of them
- Advantage is we can separate the learning process from the exploration.
- We can use either a policy that selects the optimal action most of the time and only every once in a while explores (SARSA), or we can choose a policy that is much more aggressive in its exploration (Q-learning)

Advantage of TD

- Unlike Monte Carlo,

  TD allows us to update Q(s, a) while experience is being collected

  This means that the decision making of the agent can be improved during the episode without to wait until the end. In practice they converge faster.

- Unlike Dynamic Programming,

  More efficient. focusing the effort on the states that lead to goals.

  Don’t require a model of the environment

Code Ex.

N-Step TD

N-step TD

- Note bootstrapping in SARSA

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

- After performing an action, we replace the remaining rewards by the $Q(S_{t+1}, A_{t+1})$ estimate, applying our estimate one step in the future
  - Advantage is we don't have to wait until the end of the episode to obtain the remaining rewards because we use estimate to replace them.
  - But, it induces bias

N-step TD

- Note that TD is a mix of MC and Dynamic Programming
- How many actual rewards vs. how many estimate using Q val.

2-step bootstrapping

$$R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, A_{t+2})$$

3-step bootstrapping

$$R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+1} + \gamma^3 Q(S_{t+3}, A_{t+3})$$

n-step bootstrapping

$$R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n})$$

N-step TD

- N-step return estimate:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^{n} Q(S_{t+n}, A_{t+n})$$

- N-step return as our target (N-step TD):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_{t:t+n} - Q(S_t, A_t)]$$

- Since we need to observer n rewards, we need to wait until time t+n to update the present state, $Q(S_t, A_t)$ because we have to collect those n rewards to compute $G_{t:t+n}$

N-step TD

- If $n \geq T$: $G_t = G_{t:t+n}$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G - Q(S_t, A_t)]$$

- If $n = 1$: $G_t = Rt_{+1+} \gamma Q(S_{t+1}, A_{t+1})$

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

- If $n = k$: $G_{t:t+k} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^k Q(S_{t+k}, A_{t+k})$

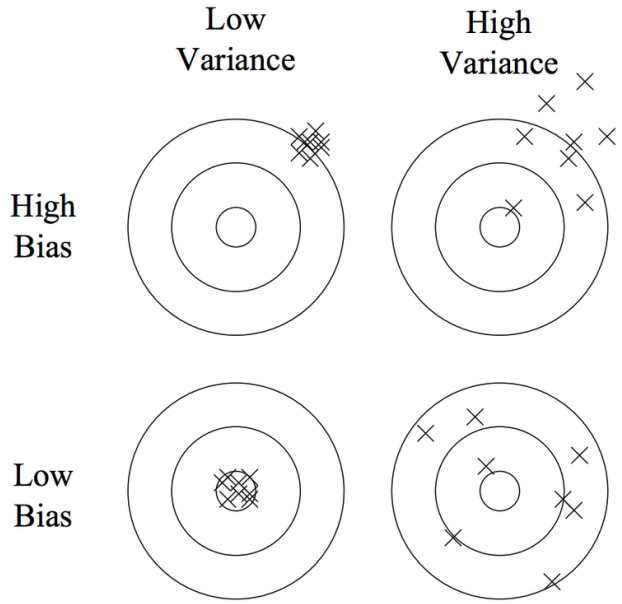
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_{t:t+k} - Q(S_t, A_t)]$$

Bias vs Variance Tradeoff

- $Q(S_{t+1}, A_{t+1})$ is an estimate of future rewards. The estimate improves throughout the learning process.

$$R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$$

- $Q(S_{t+1}, A_{t+1})$ introduces bias in the estimate.
- The higher n, the more heavily discounted the estimate $Q(S_{t+n}, A_{t+n})$ by γ^{n}.
- The higher the n, the lower the bias

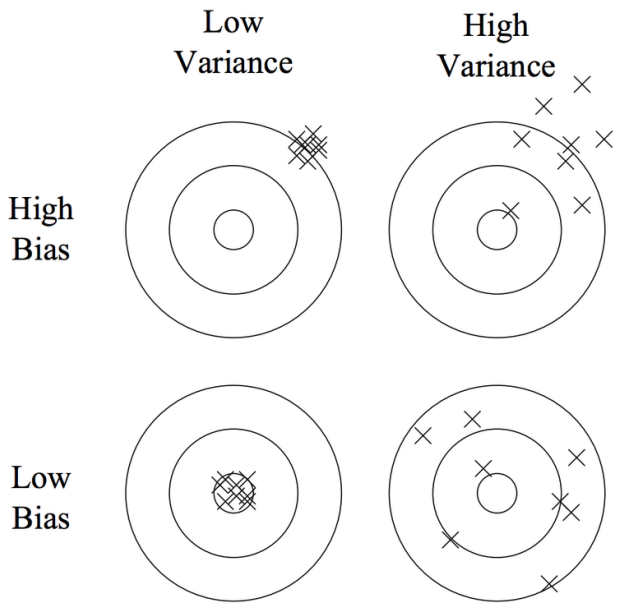


Bias vs Variance Tradeoff

- Each reward is a random variable that depends on the state s and action a preceding it.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{n-1} R_{t+n}$$

- Even if they have the same expected value, every observation of the return, G_t, samples will be very different from each other.
- The higher the n, the higher the variance



* If a policy choose a different action at the beginning of the episode, the rewards that will obtain throughout the rest can vary a lot because we'll visit different states and probably choose other actions.

N-step SARSA

- Combining SARSA with n-step bootstrapping

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^{n} Q(S_{t+n}, A_{t+n})$$

- Update rule: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_{t:t+n} - Q(S_t, A_t)]$

- On-policy learning strategy with ϵ-greedy policy:
  - $\begin{cases} \text{prob. } \epsilon : & a \sim \text{random} \\ \text{prob. } 1-\epsilon : & a \leftarrow \arg\max_{a} Q(s,a) \end{cases}$

Algorithm 1 n-step SARSA

Input: α learning rate, ϵ random action probability,
    γ discount factor, n bootstrap timestep
$\pi \leftarrow \epsilon$-greedy policy w.r.t $Q(s, a)$
Initialize $Q(s, a)$ arbitrarily, with $Q(terminal, \cdot) = 0$
for episode $\in 1..N$ **do**
    Reset the environment and observe S_0
    $A_0 \sim \pi(S_0)$
    while $t - n < T$ **do**
        if $t < T$ **then**
            Take action A_t and observe R_{t+1}, S_{t+1}
            $A_{t+1} \sim \pi(S_{t+1})$
        end if
        if $t \geq n$ **then**
            $B = Q(S_{t+1}, A_{t+1})$ if $t + 1 < T$, else 0
            $G = R_{t-n+1} + \gamma R_{t-n+2} + \cdots + \gamma^{n-1} R_{t+1} + \gamma^n B$
            $Q(S_{t-n}, A_{t-n}) \leftarrow Q(S_{t-n}, A_{t-n}) + \alpha \left[G - Q(S_{t-n}, A_{t-n})\right]$
        end if
    end while
end for
Output: Near optimal policy π and action values $Q(s, a)$

N-step SARSA

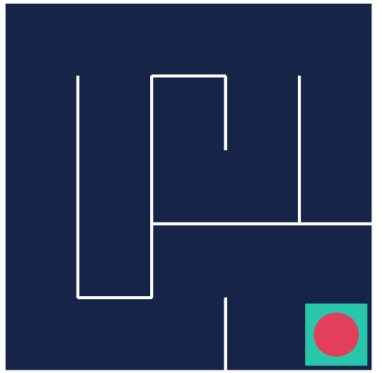
- So on the one hand, using SARSA, we have to wait and steps until we are able to start the learning process.

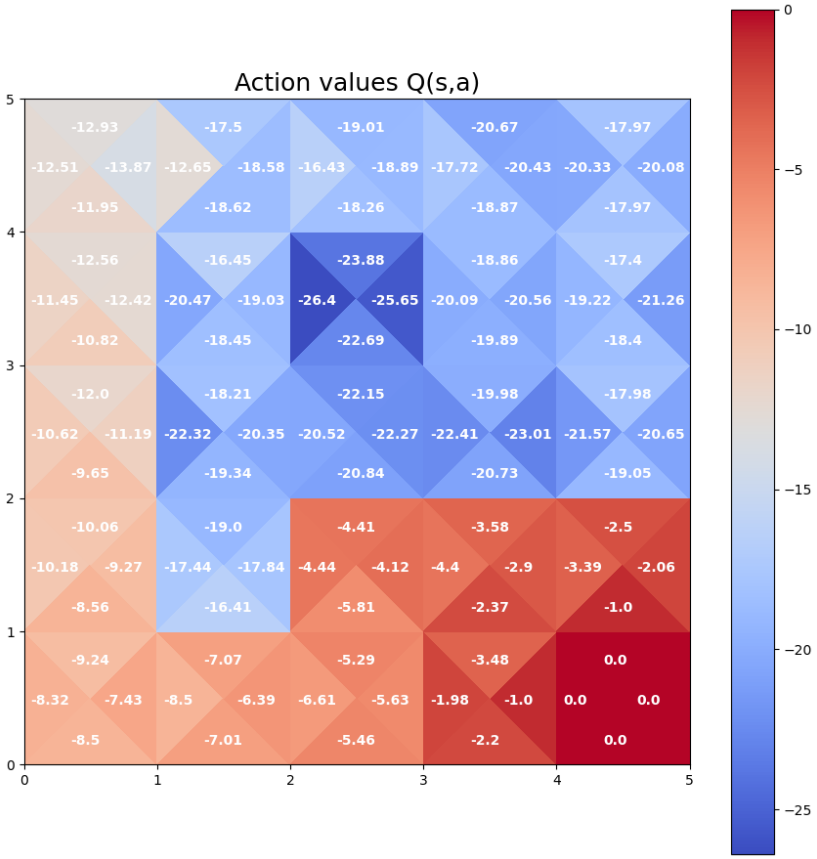
- But on the other hand, those estimates will include more information from the environment instead of a single reward, they will include n.

- And that will be a more reliable estimate of the return.

```python
def n_step_sarsa(action_values, policy, episodes, alpha=0.1, gamma=0.99, epsilon=0.2, n=8):

    for episode in range(1, episodes + 1):
        state = env.reset()
        action = policy(state, epsilon)
        transitions = []
        done = False
        t = 0

        while t-n < len(transitions):

            if not done:
                next_state, reward, done, _ = env.step(action)
                next_action = policy(next_state, epsilon)
                transitions.append([state, action, reward])

            if t >= n:
                G = (1 - done) * action_values[next_state][next_action]
                for state_t, action_t, reward_t in reversed(transitions[t-n:]):
                    G = reward_t + gamma * G
                action_values[state_t][action_t] += alpha * (G - action_values[state_t][action_t])

            t += 1
            state = next_state
            action = next_action
```

N-step SARSA





D	L	L	L	U
D	U	D	U	U
D	U	L	U	U
D	D	R	D	D
R	R	U	R	U