Monte Carlo methods

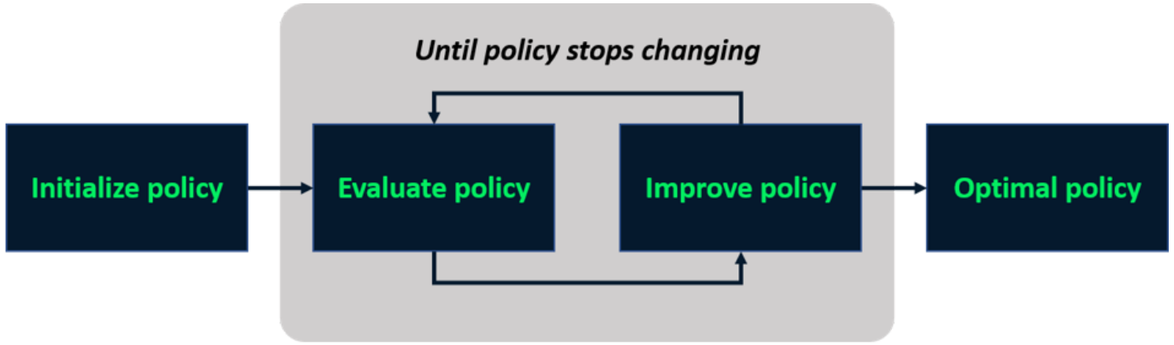
REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Fouad Trad
Machine Learning Engineer

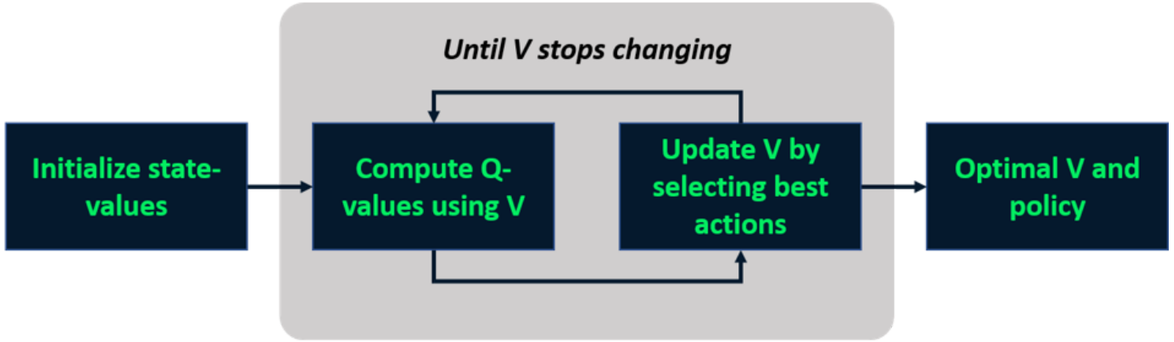
datacamp

Recap: model-based learning

- Rely on knowledge of environment dynamics
- No interaction with environment



Policy iteration



Value iteration

Model-free learning

- Doesn't rely on knowledge of environment dynamics
- Agent interacts with environment
- Learns policy through trial and error
- More suitable for real-world applications

Monte Carlo methods

- Model-free techniques
- Estimate Q-values based on episodes

Collecting random episodes

State	*Action*	*Reward*	*Return*

Monte Carlo methods

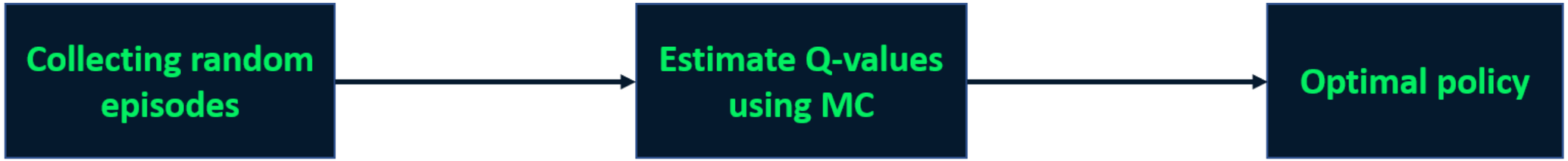
- Model-free techniques
- Estimate Q-values based on episodes

State	*Action*	*Reward*	*Return*

	A0	A1	A2	A3
S0				
S1				
S2				
S3				
S4				
S5				

Monte Carlo methods

- Model-free techniques
- Estimate Q-values based on episodes



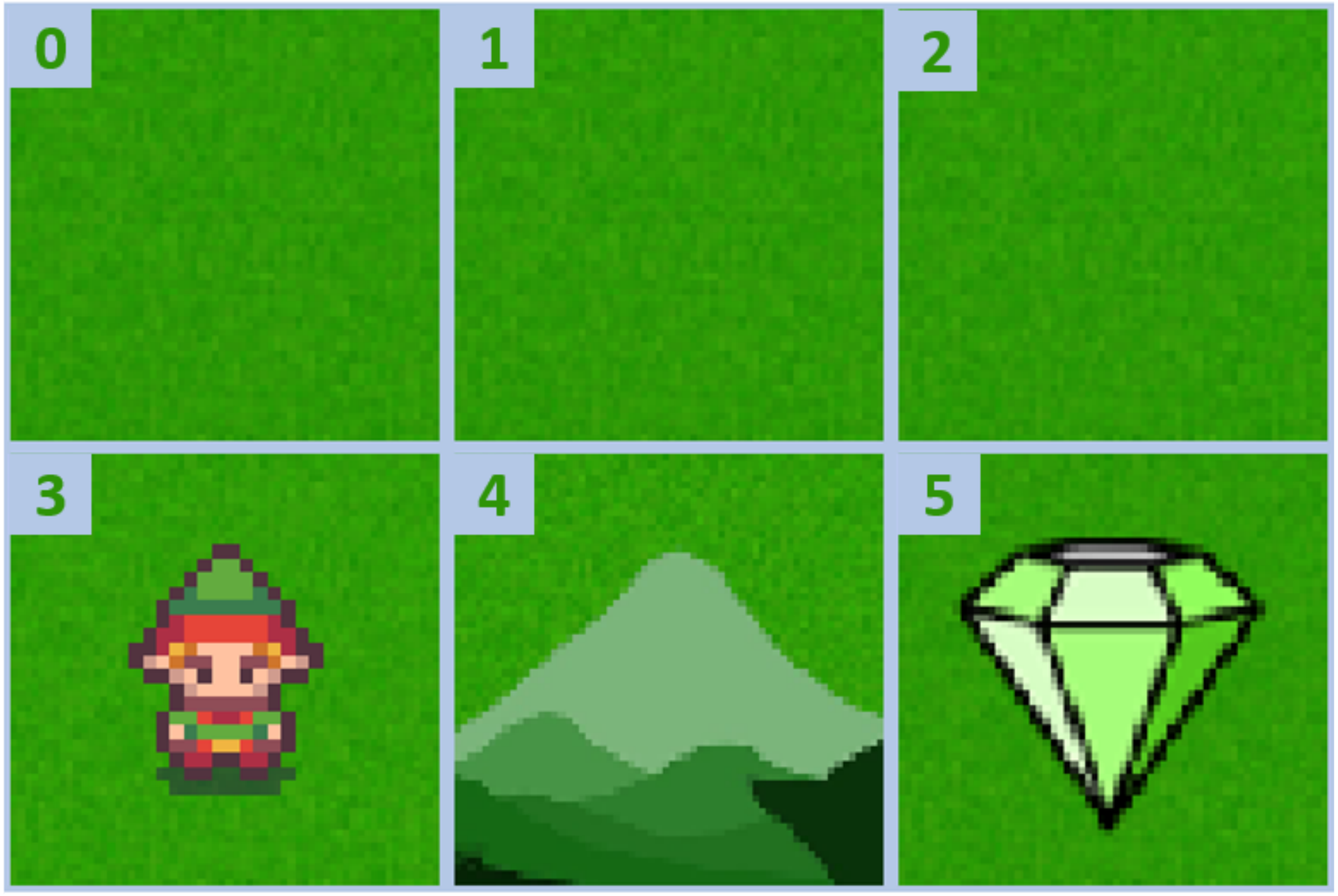
State	*Action*	*Reward*	*Return*

	A0	A1	A2	A3
S0				
S1				
S2				
S3				
S4				
S5				

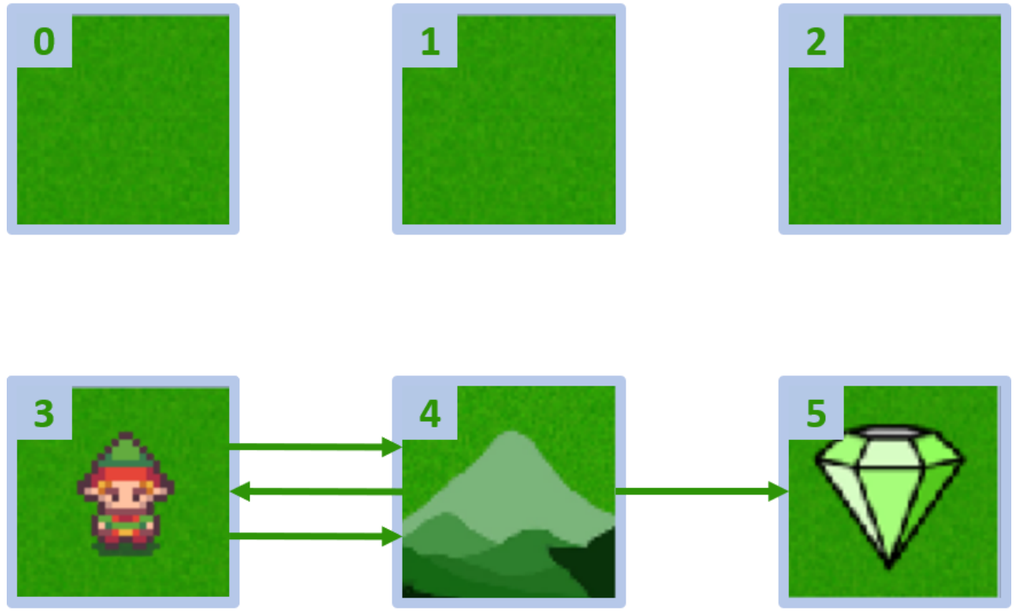
State	Optimal action
S0	
S1	
S2	
S3	
S4	
S5	

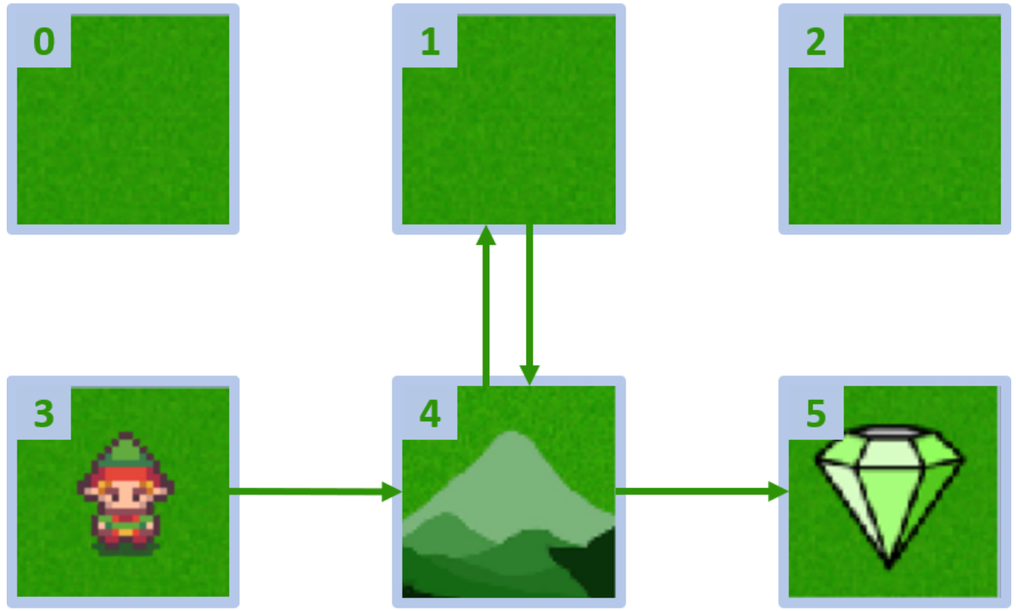
- Two methods: **first-visit**, **every-visit**

Custom grid world



Collecting two episodes





State	Action	Reward	Return
3	Right	-2	5
4	Left	-1	7
3	Right	-2	8
4	Right	10	10

State	Action	Reward	Return
3	Right	-2	5
4	Up	-1	7
1	Down	-2	8
4	Right	10	10

Estimating Q-values

State	Action	Reward	Return
3	Right	-2	5
4	Left	-1	7
3	Right	-2	8
4	Right	10	10

State	Action	Reward	Return
3	Right	-2	5
4	Up	-1	7
1	Down	-2	8
4	Right	10	10

- Q-table: table for Q-values

	Left	Down	Right	Up
0				
1				
2				
3				
4				
5				

Q(4, left), Q(4, up), and Q(1, down)

State	Action	Reward	Return
3	Right	-2	5
4	Left	-1	7
3	Right	-2	8
4	Right	10	10

State	Action	Reward	Return
3	Right	-2	5
4	Up	-1	7
1	Down	-2	8
4	Right	10	10

- (s,a) appears once -> fill with return

	Left	Down	Right	Up
0				
1		8		
2				
3				
4	7			7
5				

Q(4, right)

State	Action	Reward	Return
3	Right	-2	5
4	Left	-1	7
3	Right	-2	8
4	Right	10	10

State	Action	Reward	Return
3	Right	-2	5
4	Up	-1	7
1	Down	-2	8
4	Right	10	10

- (s,a) occurs once per episode -> average

	Left	Down	Right	Up
0				
1		8		
2				
3				
4	7		10	7
5				

Q(3, right) - first-visit Monte Carlo

State	Action	Reward	Return
3	Right	-2	5
4	Left	-1	7
3	Right	-2	8
4	Right	10	10

State	Action	Reward	Return
3	Right	-2	5
4	Up	-1	7
1	Down	-2	8
4	Right	10	10

- Average first visit to (s,a) within episodes

	Left	Down	Right	Up
0				
1		8		
2				
3			5	
4	7		10	7
5				

Q(3, right) - every-visit Monte Carlo

State	Action	Reward	Return
3	Right	-2	5
4	Left	-1	7
3	Right	-2	8
4	Right	10	10

State	Action	Reward	Return
3	Right	-2	5
4	Up	-1	7
1	Down	-2	8
4	Right	10	10

- Average every visit to (s,a) within episodes

	Left	Down	Right	Up
0				
1		8		
2				
3			6	
4	7		10	7
5				

Generating an episode

```
def generate_episode():
    episode = []
    state, info = env.reset()
    terminated = False
    while not terminated:
        action = env.action_space.sample()
        next_state, reward, terminated, truncated, info = env.step(action)
        episode.append((state, action, reward))
        state = next_state
    return episode
```

First-visit Monte Carlo

```
def first_visit_mc(num_episodes):
    Q = np.zeros((num_states, num_actions))
    returns_sum = np.zeros((num_states, num_actions))
    returns_count = np.zeros((num_states, num_actions))

    for i in range(num_episodes):
        episode = generate_episode()
        visited_states_actions = set()
        for j, (state, action, reward) in enumerate(episode):
            if (state, action) not in visited_states:
                returns_sum[state, action] += sum([x[2] for x in episode[j:]])
                returns_count[state, action] += 1
                visited_states_actions.add((state, action))

    nonzero_counts = returns_count != 0
    Q[nonzero_counts] = returns_sum[nonzero_counts] / returns_count[nonzero_counts]
    return Q
```

Every-visit Monte Carlo

```
def every_visit_mc(num_episodes):
    Q = np.zeros((num_states, num_actions))
    returns_sum = np.zeros((num_states, num_actions))
    returns_count = np.zeros((num_states, num_actions))

    for i in range(num_episodes):
        episode = generate_episode()

        for j, (state, action, reward) in enumerate(episode):

            returns_sum[state, action] += sum([x[2] for x in episode[j:]])
            returns_count[state, action] += 1



    nonzero_counts = returns_count != 0
    Q[nonzero_counts] = returns_sum[nonzero_counts] / returns_count[nonzero_counts]
    return Q
```

Getting the optimal policy

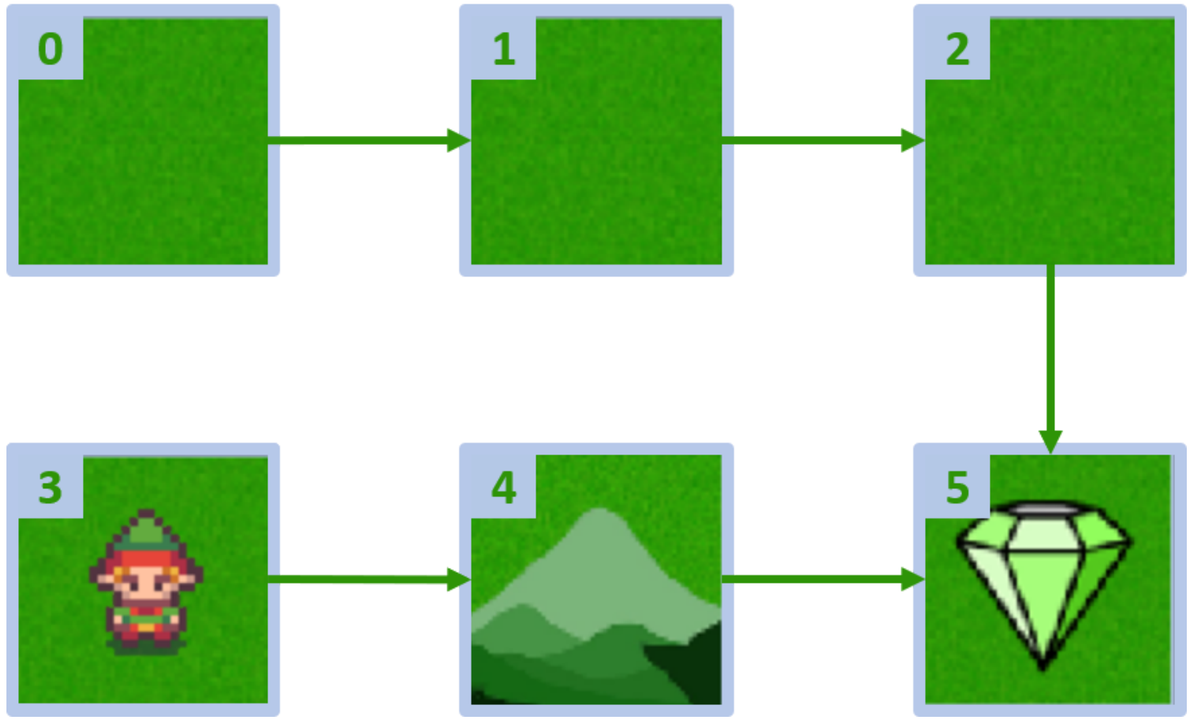
```
def get_policy():
    policy = {state: np.argmax(Q[state]) for state in range(num_states)}
    return policy
```

Putting things together

```
Q = first_visit_mc(1000)
policy_first_visit = get_policy()
print("First-visit policy: \n", policy_first_visit)
Q = every_visit_mc(1000)
policy_every_visit = get_policy()
print("Every-visit policy: \n", policy_every_visit)
```

```
First-visit policy:
{0: 2, 1: 2, 2: 1,
 3: 2, 4: 2, 5: 0}

Every-visit policy:
{0: 2, 1: 2, 2: 1,
 3: 2, 4: 2, 5: 0}
```



Let's practice!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Temporal difference learning

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Fouad Trad
Machine Learning Engineer

TD learning vs. Monte Carlo

TD learning

- Model-free
- Estimate Q-table based on interaction
- Update Q-table each step within episode
- Suitable for tasks with long/indefinite episodes

Monte Carlo

- Model-free
- Estimate Q-table based on interaction
- Update Q-table when at least one episode done
- Suitable for short episodic tasks

TD learning as weather forecasting

SARSA

- TD algorithm
- **On-policy method:** adjusts strategy based on taken actions

SARSA update rule

$$\underbrace{Q(s,a)}_{\text{New Q value}} = (1-\alpha)\underbrace{Q(s,a)}_{\text{Old Q value}} + \alpha[r + \gamma \underbrace{Q(s',a')}_{\text{Q value for next state – action pair}}]$$

- α: learning rate
- γ: discount factor
- Both between 0 and 1

Frozen Lake

Initialization

```
env = gym.make("FrozenLake", is_slippery=False)

num_states = env.observation_space.n
num_actions = env.action_space.n

Q = np.zeros((num_states, num_actions))

alpha = 0.1
gamma = 1
num_episodes = 1000
```

SARSA loop

```
for episode in range(num_episodes):
    state, info = env.reset()
    action = env.action_space.sample()
    terminated = False
    while not terminated:
        next_state, reward, terminated, truncated, info = env.step(action)
        next_action = env.action_space.sample()
        update_q_table(state, action, reward, next_state, next_action)
        state, action = next_state, next_action
```

SARSA updates

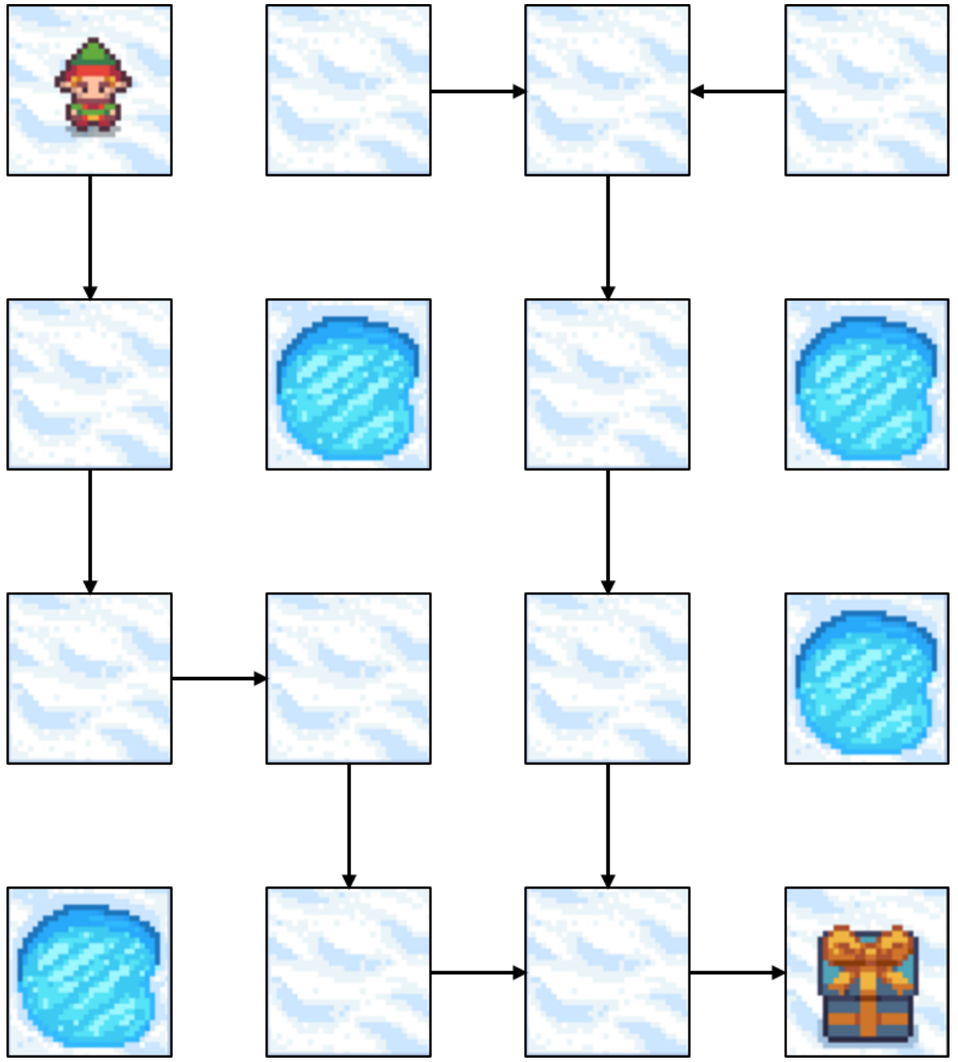
```
def update_q_table(state, action, reward, next_state, next_action):
    old_value = Q[state, action]
    next_value = Q[next_state, next_action]
    Q[state, action] = (1 - alpha) * old_value + alpha * (reward + gamma * next_value)
```

$$\underbrace{Q(s,a)}_{\text{New Q value}} = (1-\alpha)\underbrace{Q(s,a)}_{\text{Old Q value}} + \alpha[r + \gamma \underbrace{Q(s',a')}_{\text{Q value for next state – action pair}}]$$

Deriving the optimal policy

```
policy = get_policy()
print(policy)
```

```
{ 0: 1,  1: 2,  2: 1,  3: 0,
  4: 1,  5: 0,  6: 1,  7: 0,
  8: 2,  9: 1, 10: 1, 11: 0,
 12: 0, 13: 2, 14: 2, 15: 0}
```



Let's practice!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Q-learning

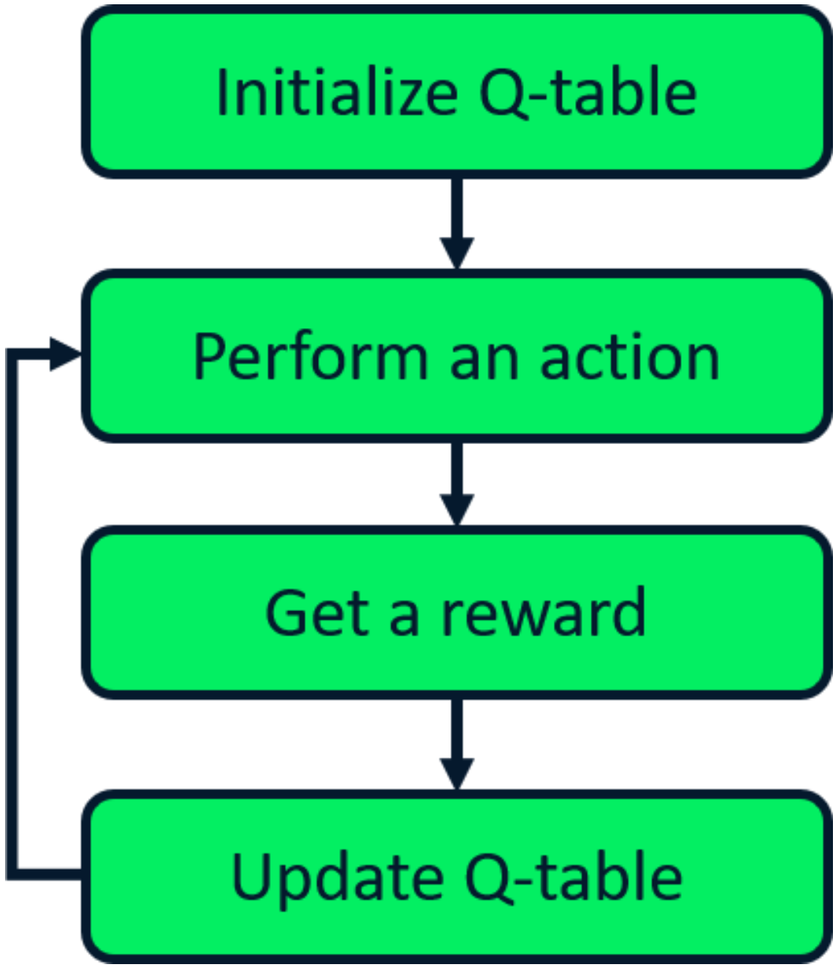
REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Fouad Trad
Machine Learning Engineer

datacamp

Introduction to Q-learning

- Stands for **quality learning**
- Model-free technique
- Learns optimal Q-table by interaction



Q-learning vs. SARSA

SARSA

$$\underbrace{Q(s,a)}_{\text{New Q value}} = (1-\alpha)\underbrace{Q(s,a)}_{\text{Old Q value}} + \alpha[r + \gamma \underbrace{Q(s',a')}_{\substack{\text{Q value for next} \\ \text{state – action pair}}}]$$

- Updates based on taken action
- On-policy learner

Q-learning

$$\underbrace{Q(s,a)}_{\text{New Q value}} = (1-\alpha)\underbrace{Q(s,a)}_{\text{Old Q value}} + \alpha[r + \gamma \underbrace{\max_{a'} Q(s',a')}_{\substack{\text{Maximum Q value} \\ \text{given the next state}}}]$$

- Updates independent of taken actions
- Off-policy learner

Q-learning implementation

```
env = gym.make("FrozenLake", is_slippery=True)

num_episodes = 1000
alpha = 0.1
gamma = 1

num_states, num_actions = env.observation_space.n, env.action_space.n
Q = np.zeros((num_states, num_actions))

reward_per_random_episode = []
```

Q-learning implementation

```
for episode in range(num_episodes):
    state, info = env.reset()
    terminated = False
    episode_reward = 0
    while not terminated:
        # Random action selection
        action = env.action_space.sample()
        # Take action and observe new state and reward
        new_state, reward, terminated, truncated, info = env.step(action)
        # Update Q-table
        update_q_table(state, action, new_state)
        episode_reward += reward
        state = new_state
    reward_per_random_episode.append(episode_reward)
```

Q-learning update

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a')]$$

New Q value

Old Q value

Maximum Q value given the next state

```
def update_q_table(state, action, reward, new_state):
  old_value = Q[state, action]
  next_max = max(Q[new_state])
  Q[state, action] = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
```

Using the policy

```
reward_per_learned_episode = []
```

```
policy = get_policy()
for episode in range(num_episodes):
    state, info = env.reset()
    terminated = False
    episode_reward = 0
    while not terminated:
        # Select the best action based on learned Q-table
        action = policy[state]
        # Take action and observe new state
        new_state, reward, terminated, truncated, info = env.step(action)
        state = new_state
        episode_reward += reward
    reward_per_learned_episode.append(episode_reward)
```

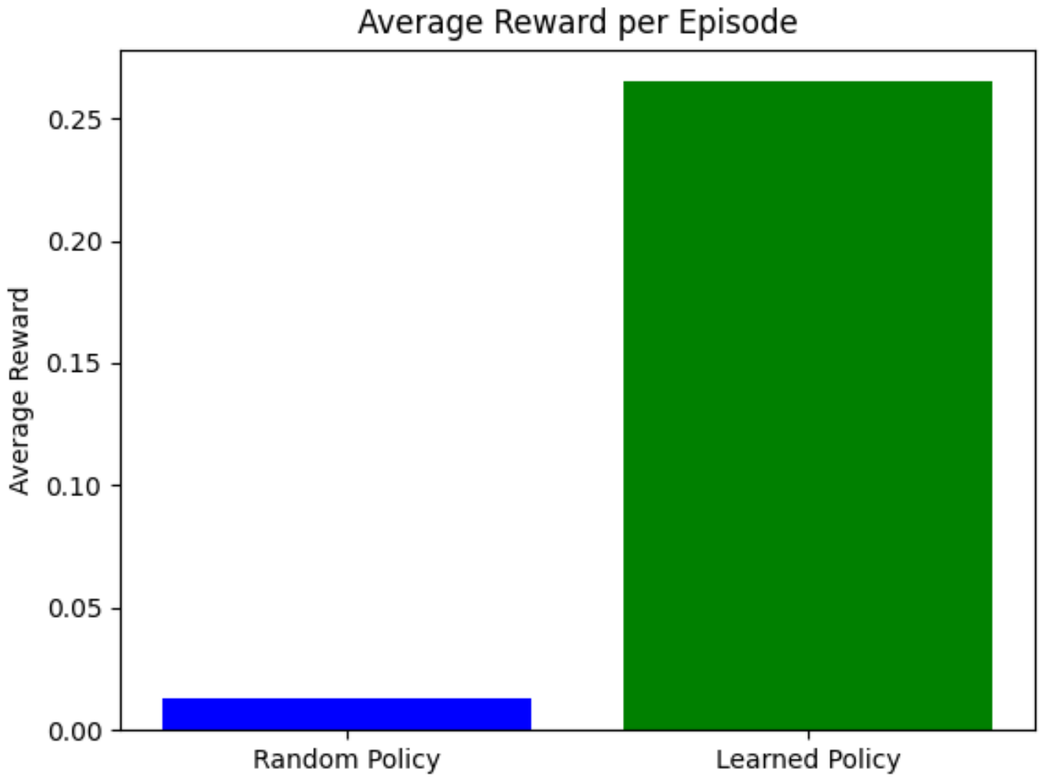
Q-learning evaluation

```
import numpy as np
import matplotlib.pyplot as plt

avg_random_reward = np.mean(reward_per_random_episode)
avg_learned_reward = np.mean(reward_per_learned_episode)

plt.bar(['Random Policy', 'Learned Policy'],
        [avg_random_reward, avg_learned_reward],
        color=['blue', 'green'])

plt.title('Average Reward per Episode')
plt.ylabel('Average Reward')
plt.show()
```



Let's practice!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON