Expected SARSA

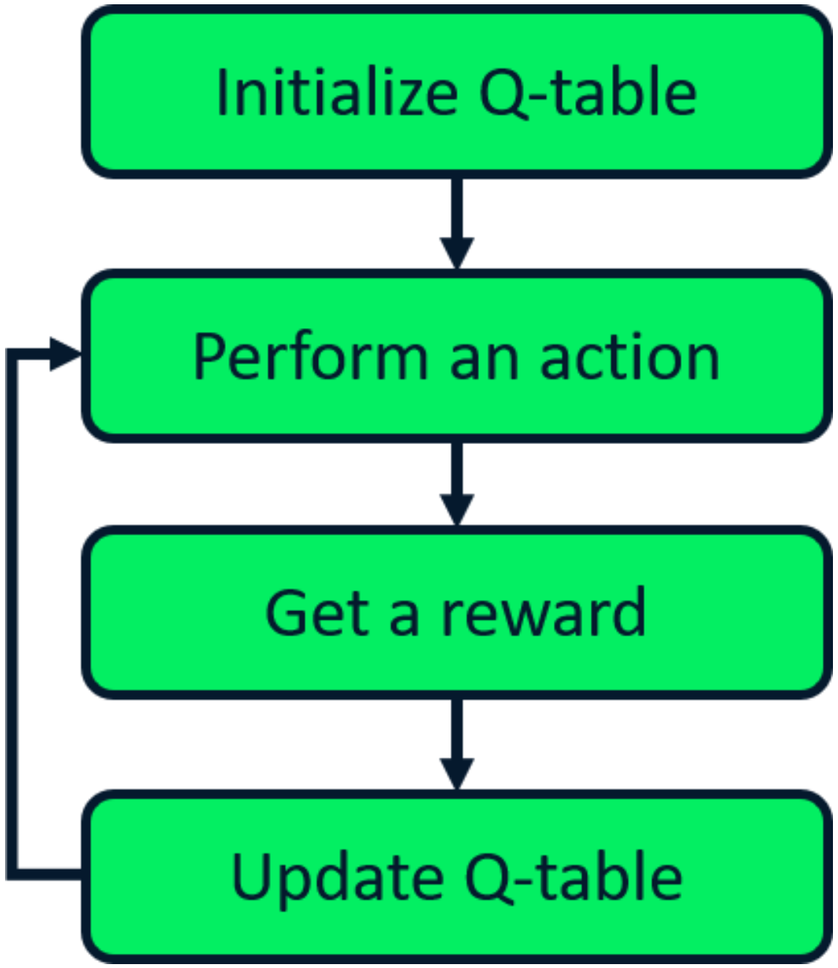
REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Fouad Trad
Machine Learning Engineer

datacamp

Expected SARSA

- TD method
- Model-free technique
- Updates Q-table differently than SARSA and Q-learning



Expected SARSA update

SARSA

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

Q-learning

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

Expected SARSA

$$\underbrace{Q(s,a)}_{\text{New Q value}} = (1-\alpha)\underbrace{Q(s,a)}_{\text{Old Q value}} + \alpha[r + \gamma \underbrace{E\{Q(s',A)\}}_{\substack{\text{Expected Q value} \\ \text{for next state} \\ \text{based on all actions}}}]$$

Expected value of next sate

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

New Q value

Old Q value

Expected Q value for next state based on all actions

- Takes into account all actions

$$E\{Q(s',A)\} = Sum(Prob(a) * Q(s',a) \;\; for\; a\; in\; A)$$

- Random actions → equal probabilities

$$E\{Q(s',A)\} = Mean(Q(s',a) \;\; for\; a\; in\; A)$$

Implementation with Frozen Lake

```
env = gym.make('FrozenLake-v1',
               is_slippery=False)

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

gamma = 0.99
alpha = 0.1
num_episodes = 1000
```

Expected SARSA update rule

```
def update_q_table(state, action, next_state, reward):
    expected_q = np.mean(Q[next_state])
    Q[state, action] = (1-alpha) * Q[state, action]
                       + alpha * (reward + gamma * expected_q)
```

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

New Q value

Old Q value

Expected Q value for next state based on all actions

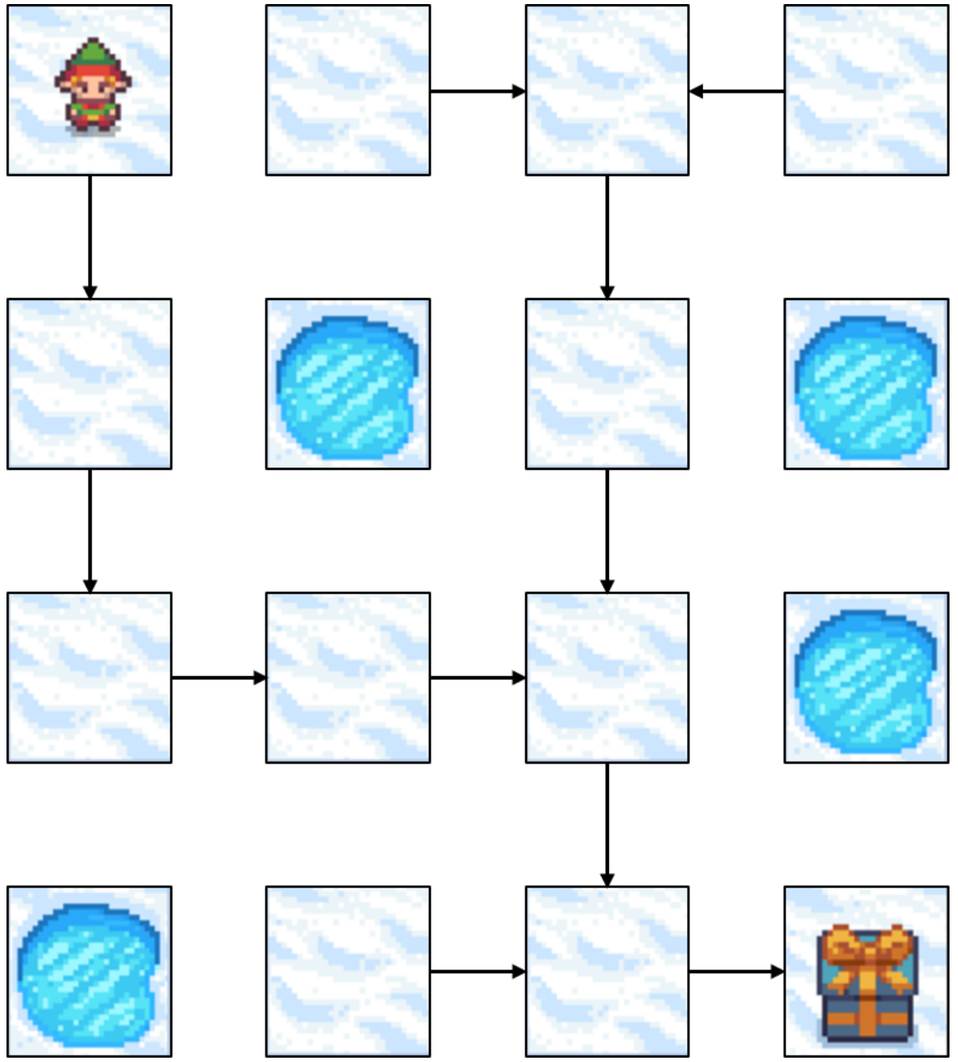
Training

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

Agent's policy

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

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



Let's practice!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Double Q-learning

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Fouad Trad
Machine Learning Engineer

datacamp

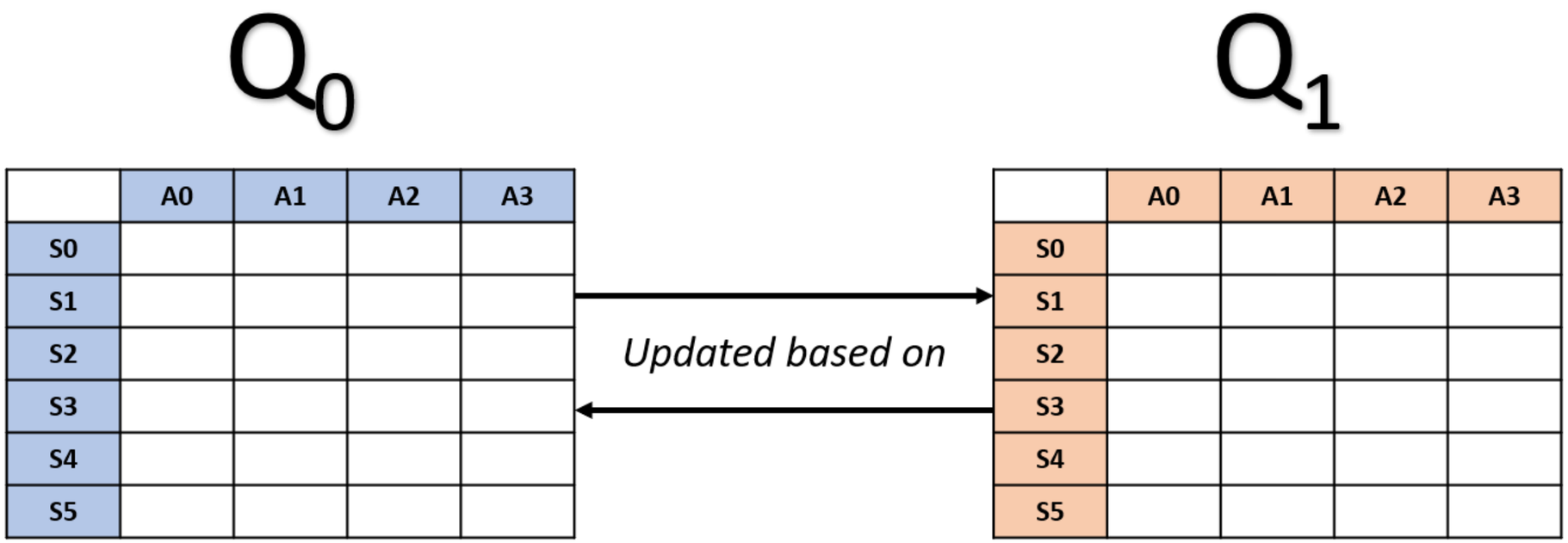
Q-learning

- Estimates optimal action-value function
- Overestimates Q-values by updating based on max Q
- Might lead to suboptimal policy learning

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

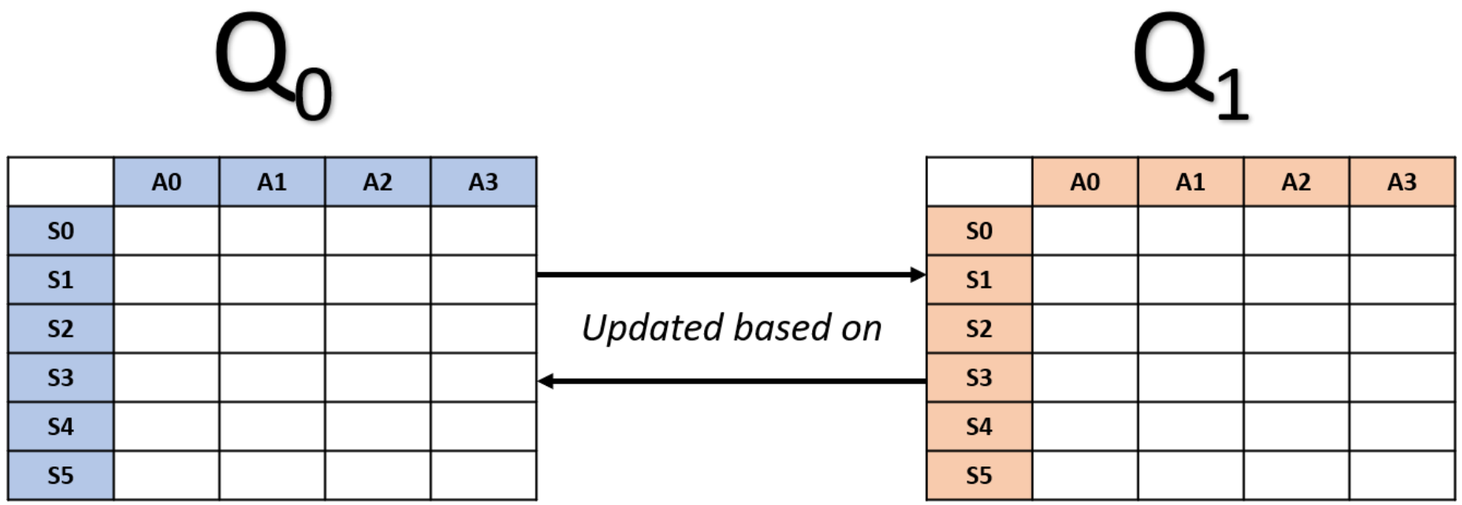
Double Q-learning

- Maintains two Q-tables
- Each table updated based on the other
- Reduces risk of Q-values overestimation

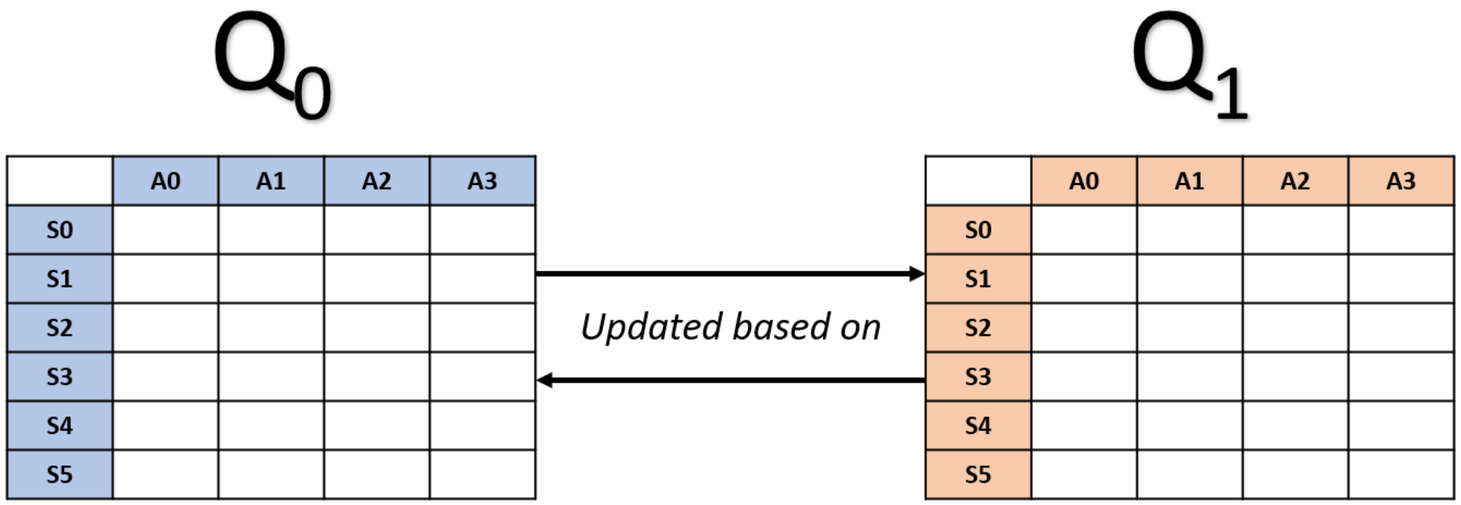


Double Q-learning updates

- Randomly select a table



Q0 update



$$max_a = argmax(Q_0(s'))$$

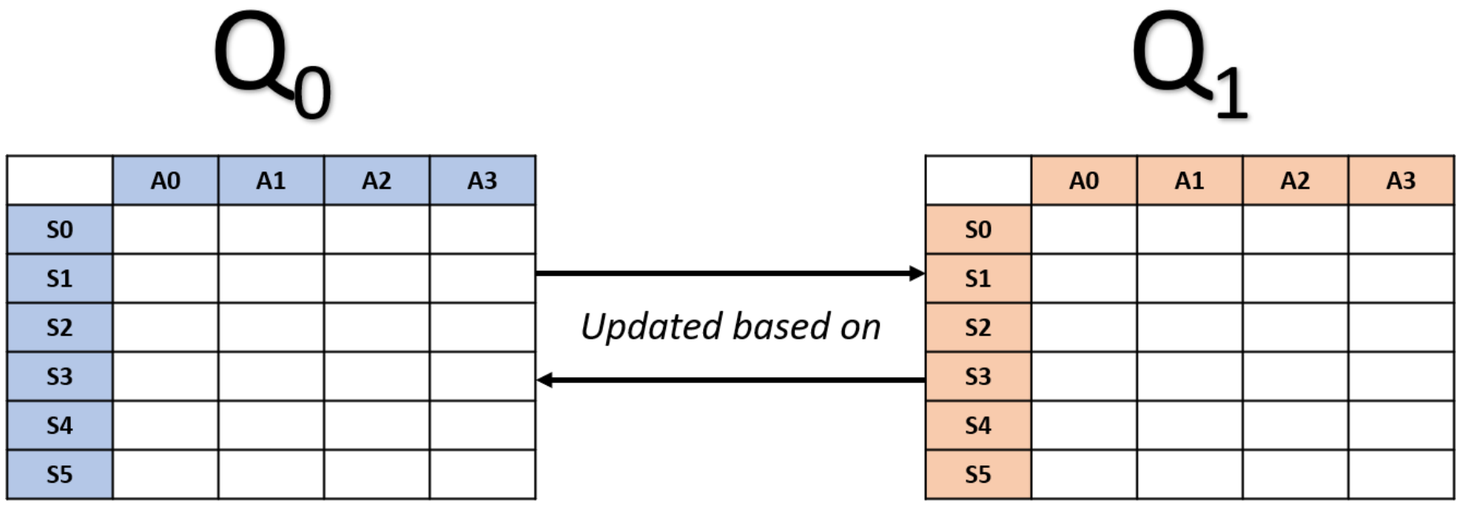
$$Q_0(s,a) = (1-\alpha)Q_0(s,a) + \alpha[r + \gamma Q_1(s', max_a)]$$

New Q value

Old Q value

Maximum Q value given the next state

Q1 update



$$max_a = argmax(Q_1(s'))$$

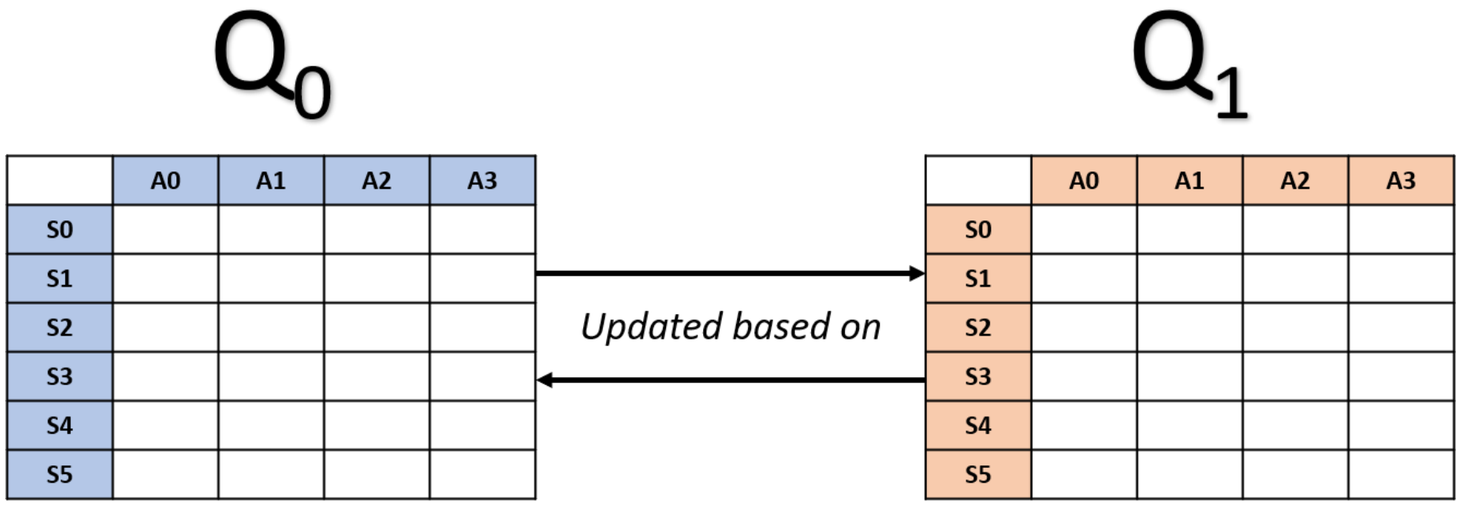
$$Q_1(s,a) = (1-\alpha)Q_1(s,a) + \alpha[r + \gamma Q_0(s', max_a)]$$

New Q value

Old Q value

Maximum Q value given the next state

Double Q-learning



- Reduces overestimation bias
- Alternates between Q0 and Q1 updates
- Both tables contribute to learning process

Implementation with Frozen Lake

```
env = gym.make('FrozenLake-v1',
               is_slippery=False)

num_states = env.observation_space.n
n_actions = env.action_space.n
Q = [np.zeros((num_states, n_actions))] * 2

num_episodes = 1000
alpha = 0.5
gamma = 0.99
```

Implementing update_q_tables()

```
def update_q_tables(state, action, reward, next_state):
    # Select a random Q-table index (0 or 1)
    i = np.random.randint(2)
    # Update the corresponding Q-table
    best_next_action = np.argmax(Q[i][next_state])
    Q[i][state, action] = (1 - alpha) * Q[i][state, action]
                          + alpha * (reward + gamma * Q[1-i][next_state, best_next_action])
```

$$\underbrace{Q_0(s,a)}_{\text{New Q value}} = (1-\alpha)\underbrace{Q_0(s,a)}_{\text{Old Q value}} + \alpha[r + \gamma \underbrace{Q_1(s', max_a)}_{\substack{\text{Maximum Q value} \\ \text{given the next state}}}]$$

$$\underbrace{Q_1(s,a)}_{\text{New Q value}} = (1-\alpha)\underbrace{Q_1(s,a)}_{\text{Old Q value}} + \alpha[r + \gamma \underbrace{Q_0(s', max_a)}_{\substack{\text{Maximum Q value} \\ \text{given the next state}}}]$$

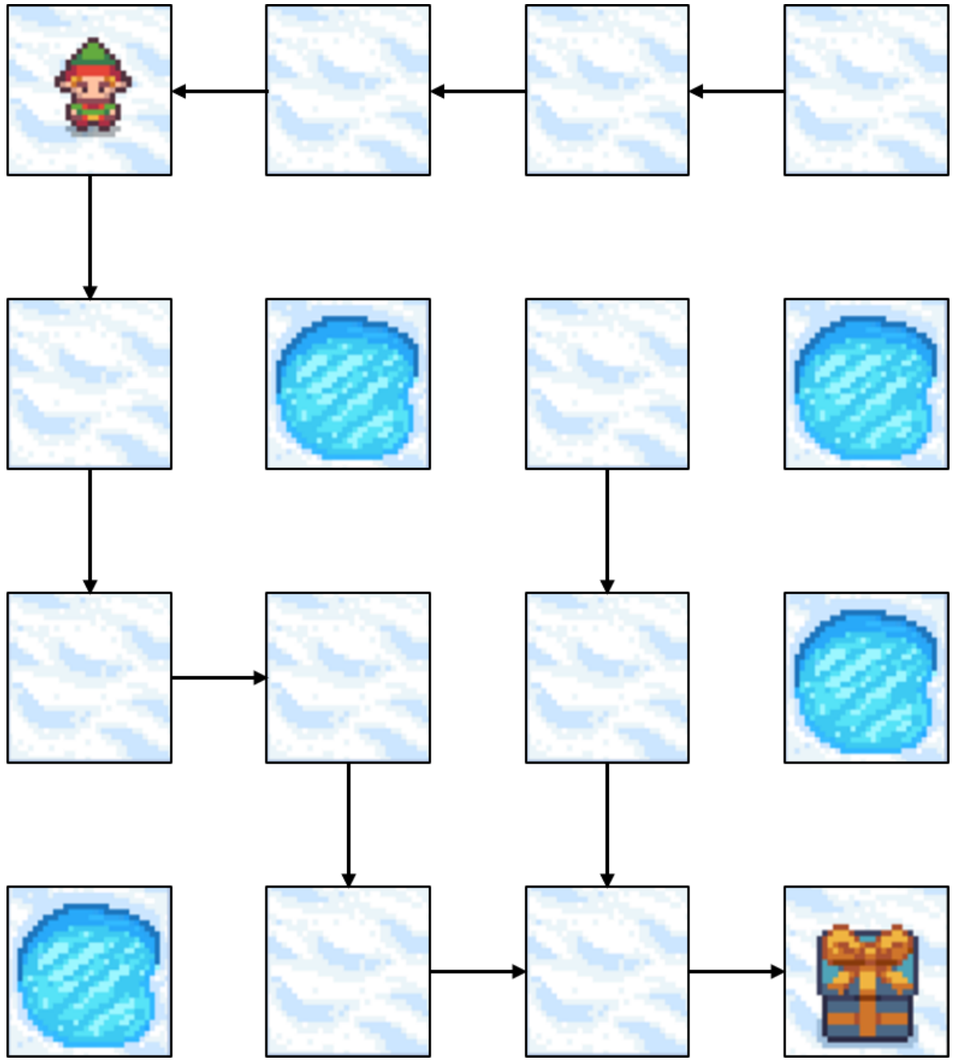
Training

```
for episode in range(num_episodes):
    state, info = env.reset()
    terminated = False

    while not terminated:
        action = np.random.choice(n_actions)
        next_state, reward, terminated, truncated,  info = env.step(action)
        update_q_tables(state, action, reward, next_state)
        state = next_state

final_Q = (Q[0] + Q[1])/2
# OR
final_Q = Q[0] + Q[1]
```

Agent's policy

```
policy = {state: np.argmax(final_Q[state])
          for state in range(num_states)}
print(policy)
```

```
{ 0: 1,  1: 0,  2: 0,  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

Balancing exploration and exploitation

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Fouad Trad
Machine Learning Engineer

datacamp

Training with random actions

- Agent explores environment
- No strategy optimization based on learned knowledge
- Agent uses knowledge when training done

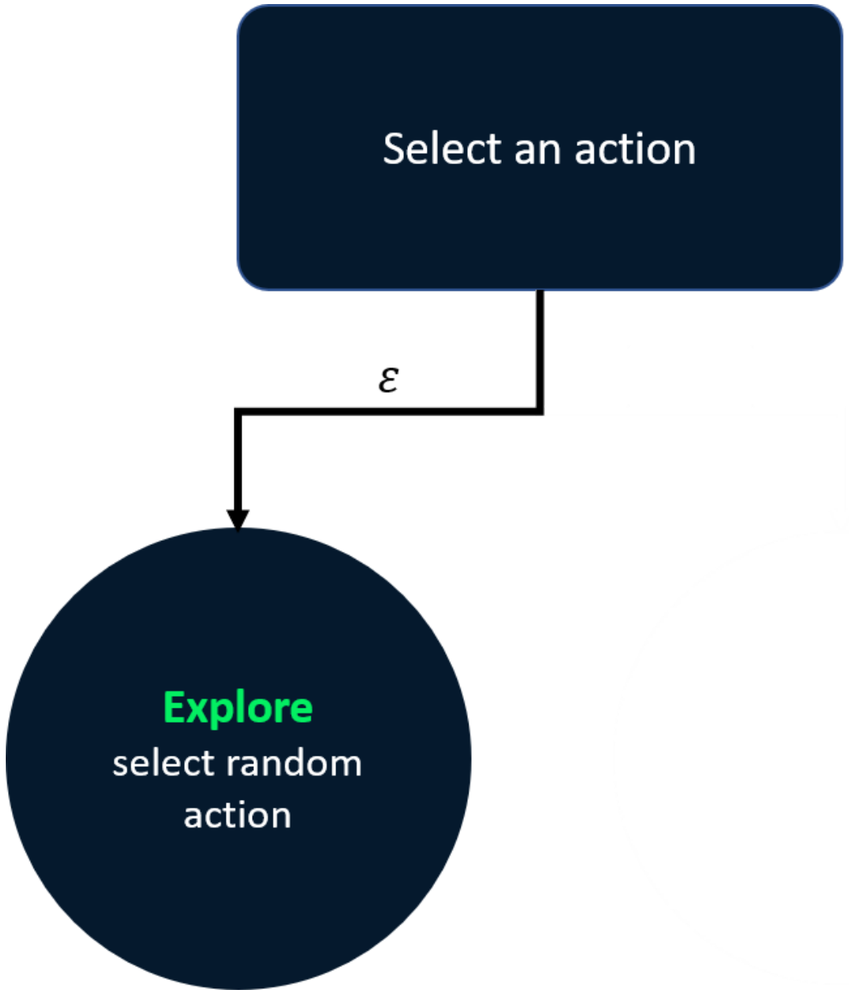
Exploration-exploitation trade-off

- Balances exploration and exploitation
- Continuous exploration prevents strategy refinement
- Exclusive exploitation misses undiscovered opportunities

Dining choices

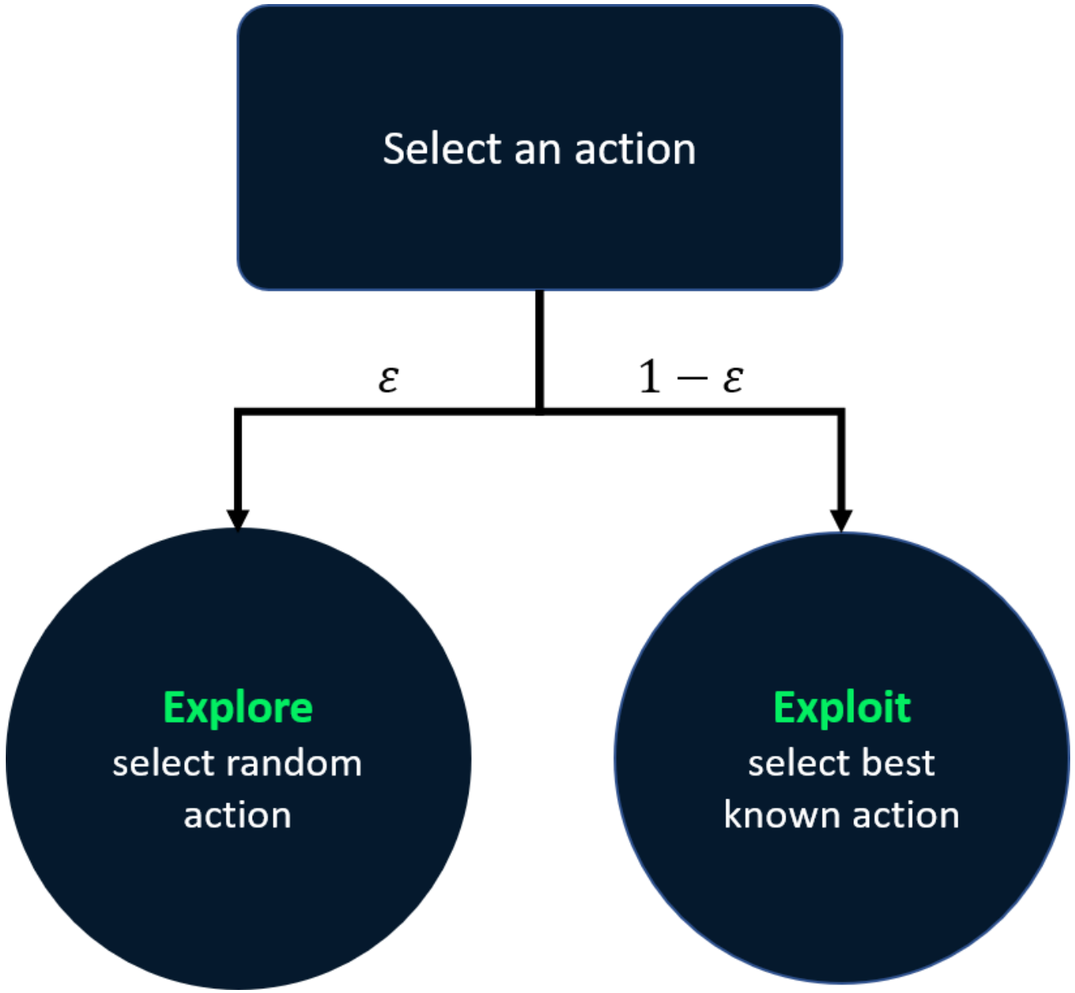
Epsilon-greedy strategy

- Explore with probability epsilon



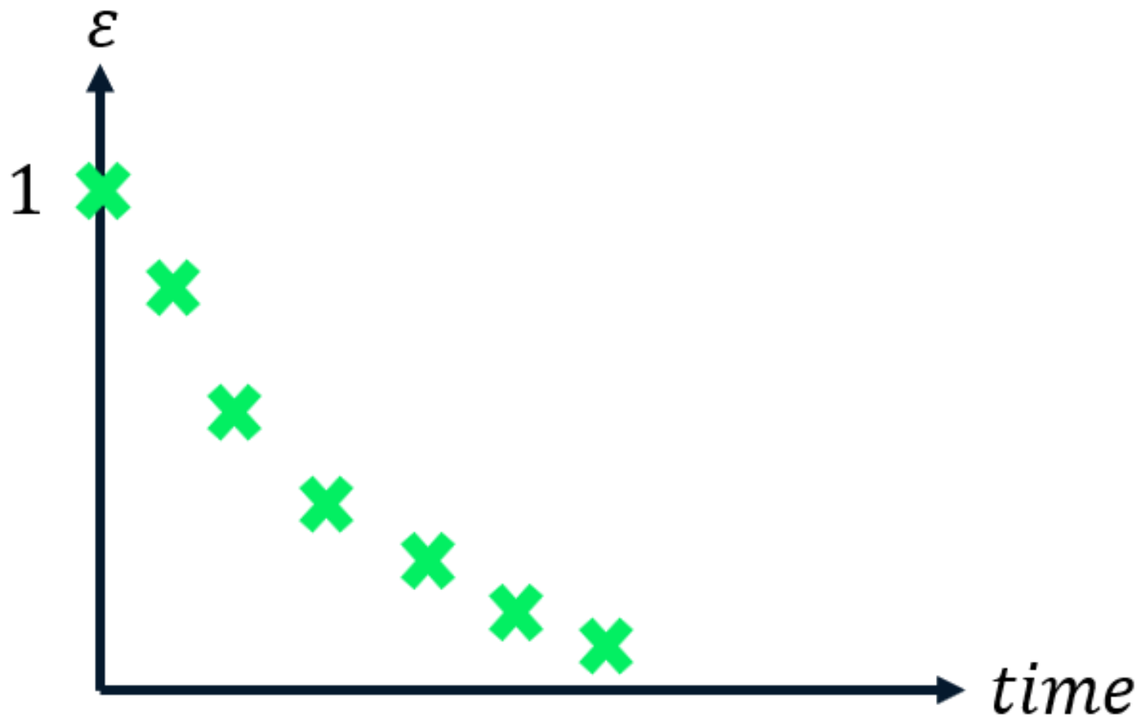
Epsilon-greedy strategy

- Explore with probability epsilon
- Exploit with probability 1-epsilon
- Ensures continuous exploration while using knowledge



Decayed epsilon-greedy strategy

- Reduces epsilon over time
- More exploration initially
- More exploitation later on
- Agent increasingly relies on its accumulated knowledge



Implementation with Frozen Lake

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

action_size = env.action_space.n
state_size = env.observation_space.n
Q = np.zeros((state_size, action_size))

alpha = 0.1
gamma = 0.99
total_episodes = 10000
```

Implementing epsilon_greedy()

```
def epsilon_greedy(state):
    if np.random.rand() < epsilon:
        action = env.action_space.sample()  # Explore
    else:
        action = np.argmax(Q[state, :])  # Exploit
    return action
```

Training epsilon-greedy

```
epsilon = 0.9   # Exploration rate
rewards_eps_greedy = []
```

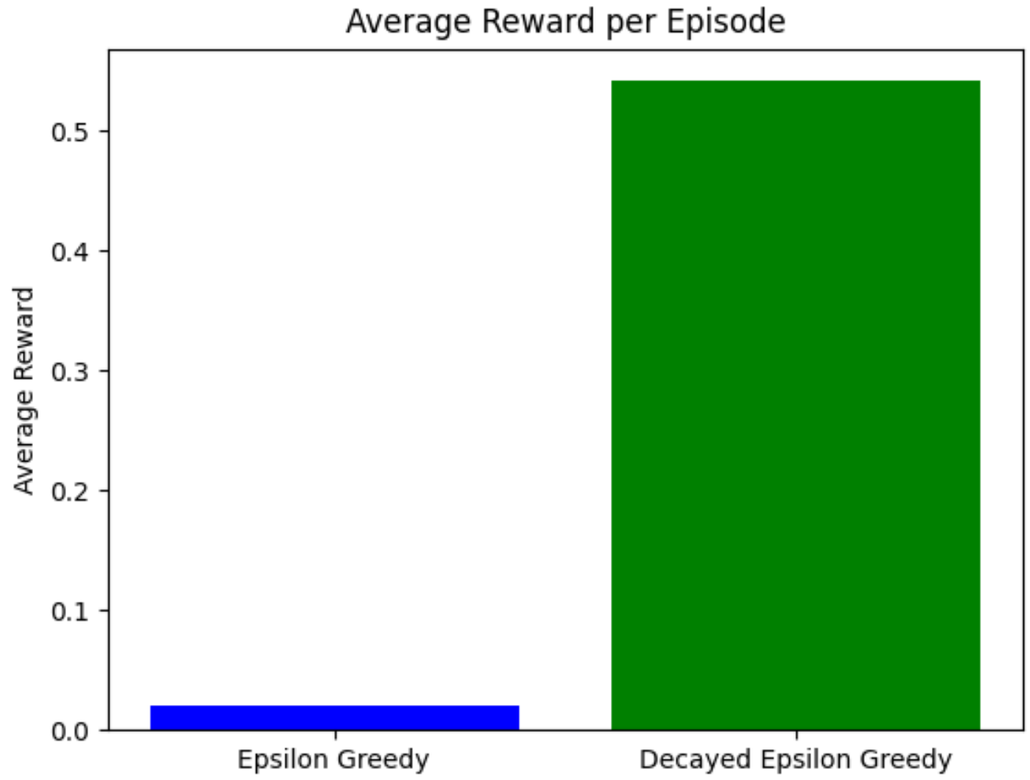
```
for episode in range(total_episodes):
    state, info = env.reset()
    terminated = False
    episode_reward = 0
    while not terminated:
        action = epsilon_greedy(state)
        new_state, reward, terminated, truncated, info = env.step(action)
        Q[state, action] = update_q_table(state, action, new_state)
        state = new_state
        episode_reward += reward
    rewards_eps_greedy.append(episode_reward)
```

Training decayed epsilon-greedy

```
epsilon = 1.0   # Exploration rate
epsilon_decay = 0.999
min_epsilon = 0.01
rewards_decay_eps_greedy = []
for episode in range(total_episodes):
    state, info = env.reset()
    terminated = False
    episode_reward = 0
    while not terminated:
        action = epsilon_greedy(state)
        new_state, reward, terminated, truncated, info = env.step(action)
        episode_reward += reward
        Q[state, action] = update_q_table(state, action, new_state)
        state = new_state
    rewards_decay_eps_greedy.append(episode_reward)
    epsilon = max(min_epsilon, epsilon * epsilon_decay)
```

Comparing strategies

```
avg_eps_greedy= np.mean(rewards_eps_greedy)
avg_decay = np.mean(rewards_decay_eps_greedy)
plt.bar(['Epsilon Greedy', 'Decayed Epsilon Greedy'],
        [avg_eps_greedy, avg_decay],
        color=['blue', 'green'])
plt.title('Average Reward per Episode')
plt.ylabel('Average Reward')
plt.show()
```



Let's practice!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Multi-armed bandits

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Fouad Trad
Machine Learning Engineer

datacamp

Multi-armed bandits

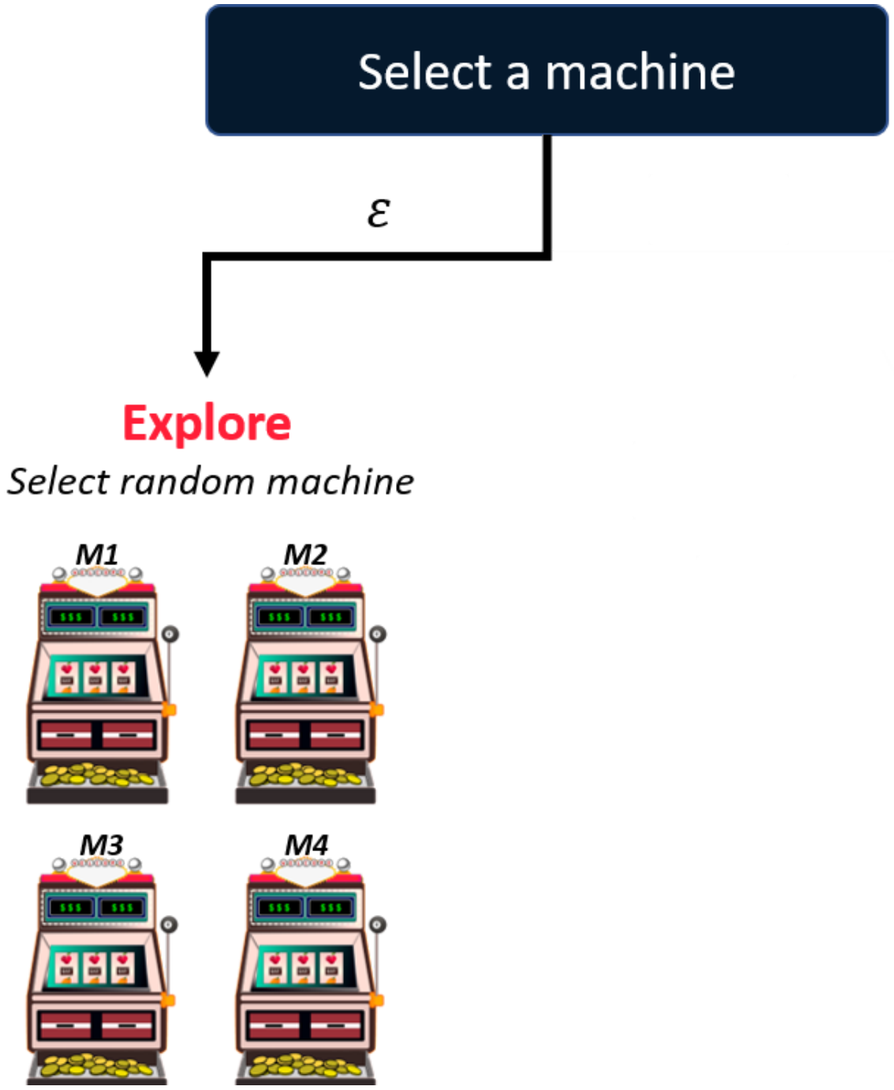
- Gambler facing slot machines
- Challenge → maximize winning
- Solution → exploration-exploitation

Slot machines

- Reward from an arm is 0 or 1
- Agent's goal → Accumulate maximum reward

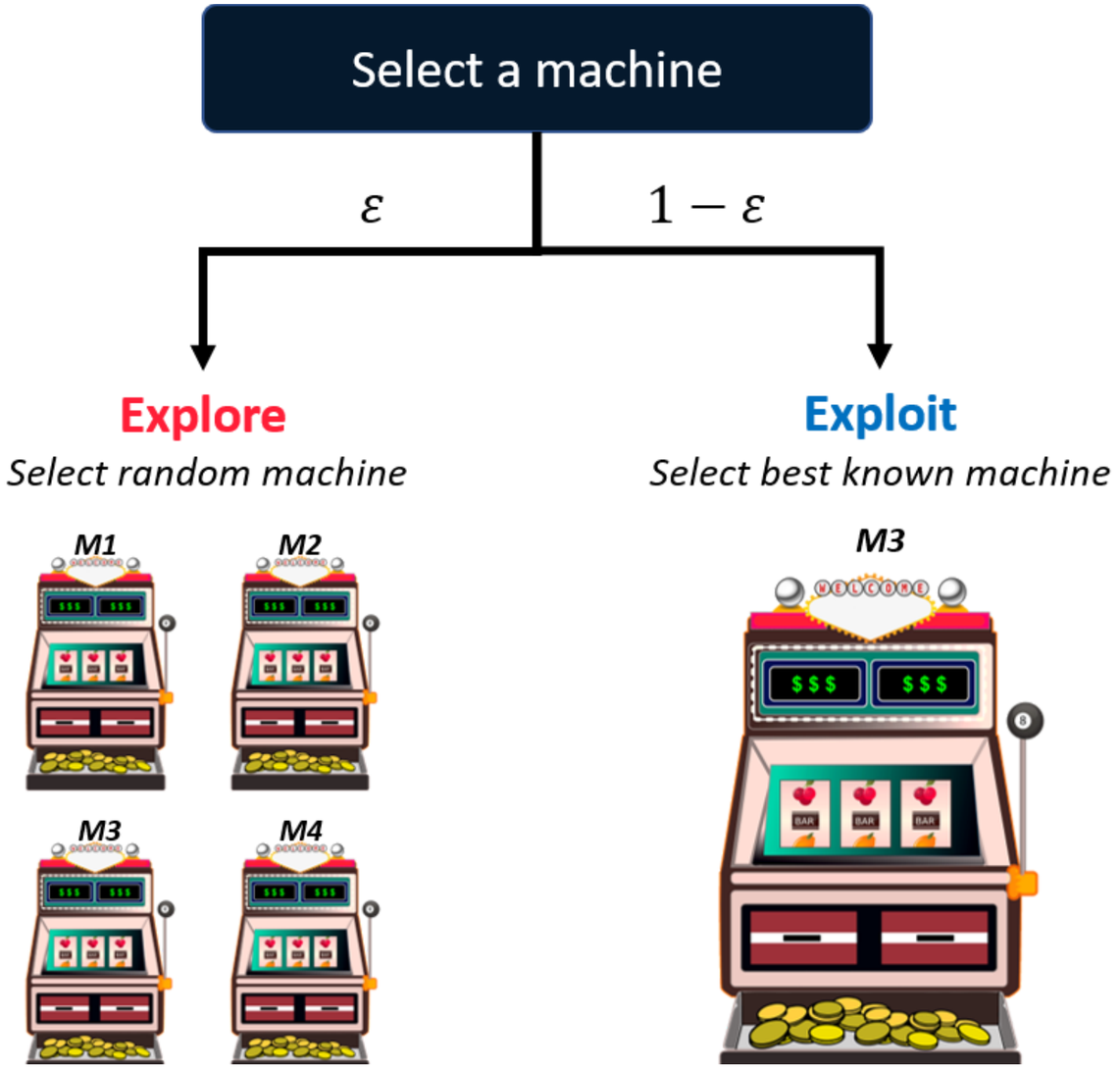
Solving the problem

- Decayed epsilon-greedy
- Epsilon → select random machine



Solving the problem

- Decayed epsilon-greedy
- Epsilon → select random machine
- 1 - epsilon → select best machine so far
- Epsilon decreases over time



Initialization

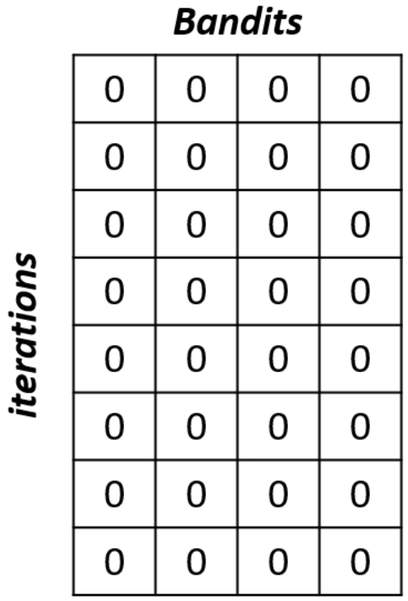
```
n_bandits = 4
true_bandit_probs = np.random.rand(n_bandits)

n_iterations = 100000
epsilon = 1.0
min_epsilon = 0.01
epsilon_decay = 0.999

counts = np.zeros(n_bandits)  # How many times each bandit was played
values = np.zeros(n_bandits)  # Estimated winning probability of each bandit
rewards = np.zeros(n_iterations)  # Reward history
selected_arms = np.zeros(n_iterations, dtype=int)  # Arm selection history
```

Interaction loop

```
for i in range(n_iterations):
    arm = epsilon_greedy()
    reward = np.random.rand() < true_bandit_probs[arm]
    rewards[i] = reward
    selected_arms[i] = arm
    counts[arm] += 1
    values[arm] += (reward - values[arm]) / counts[arm]
    epsilon = max(min_epsilon, epsilon * epsilon_decay)
```

Analyzing selections

```
selections_percentage = np.zeros((n_iterations, n_bandits))
```



Analyzing selections

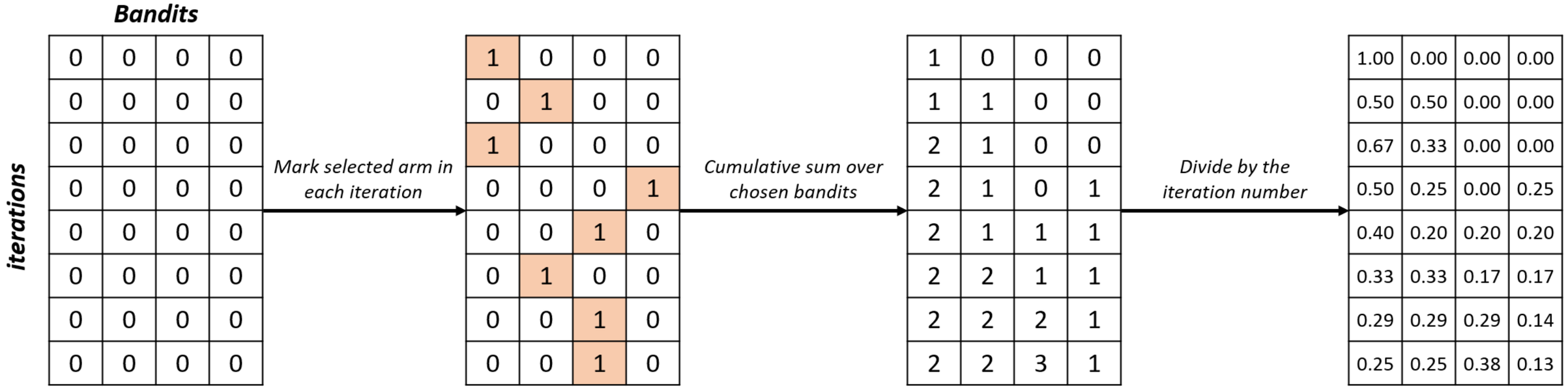
```
selections_percentage = np.zeros((n_iterations, n_bandits))
for i in range(n_iterations):
    selections_percentage[i, selected_arms[i]] = 1
```

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

1	0	0	0
0	1	0	0
1	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
0	0	1	0
0	0	1	0

Analyzing selections

```
selections_percentage = np.zeros((n_iterations, n_bandits))
for i in range(n_iterations):
    selections_percentage[i, selected_arms[i]] = 1
selections_percentage = np.cumsum(selections_percentage, axis=0) /
                        np.arange(1, n_iterations + 1).reshape(-1, 1)
```



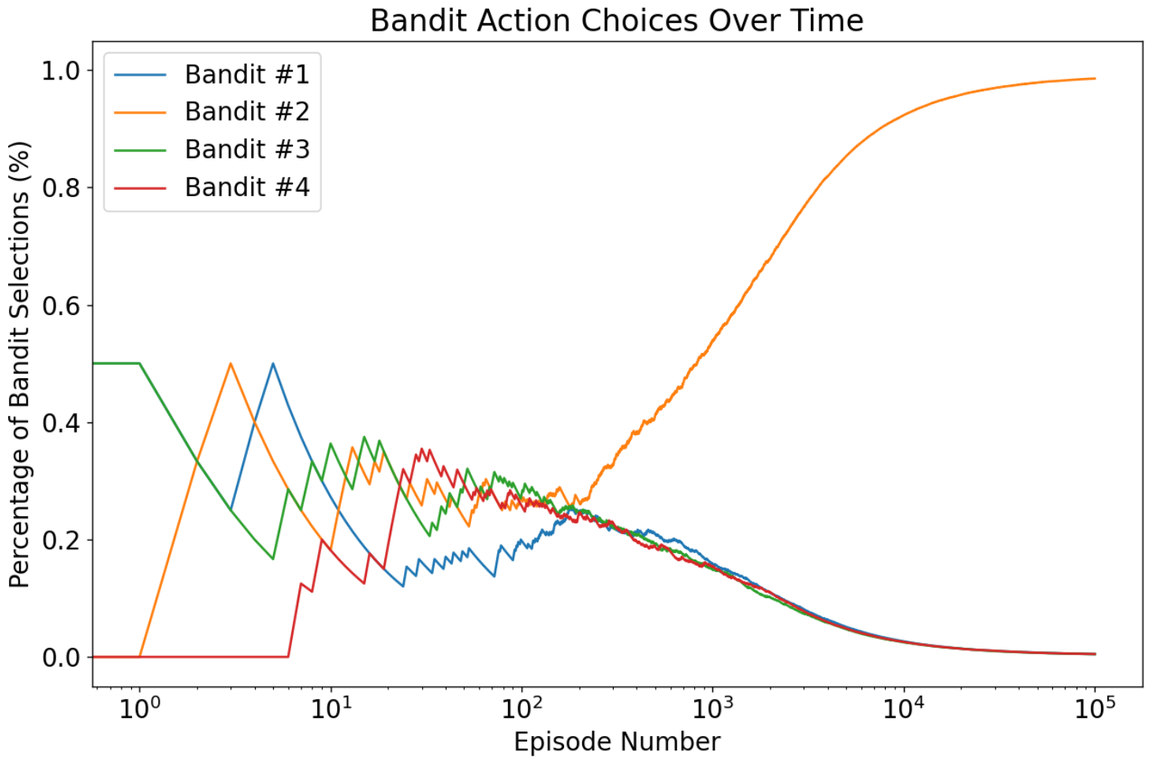
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

1	0	0	0
0	1	0	0
1	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
0	0	1	0
0	0	1	0

1	0	0	0
1	1	0	0
2	1	0	0
2	1	0	1
2	1	1	1
2	2	1	1
2	2	2	1
2	2	3	1

1.00	0.00	0.00	0.00
0.50	0.50	0.00	0.00
0.67	0.33	0.00	0.00
0.50	0.25	0.00	0.25
0.40	0.20	0.20	0.20
0.33	0.33	0.17	0.17
0.29	0.29	0.29	0.14
0.25	0.25	0.38	0.13

Analyzing selections




```python
for arm in range(n_bandits):
    plt.plot(selections_percentage[:, arm], label=f'Bandit #{arm+1}')
plt.xscale('log')
plt.title('Bandit Action Choices Over Time')
plt.xlabel('Episode Number')
plt.ylabel('Percentage of Bandit Selections (%)')
plt.legend()
plt.show()
for i, prob in enumerate(true_bandit_probs, 1):
    print(f"Bandit #{i} -> {prob:.2f}")
```

```
Bandit #1 -> 0.37
Bandit #2 -> 0.95
Bandit #3 -> 0.73
Bandit #4 -> 0.60
```

- Agent learns to select the bandit with highest probability

Let's practice!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Congratulations!

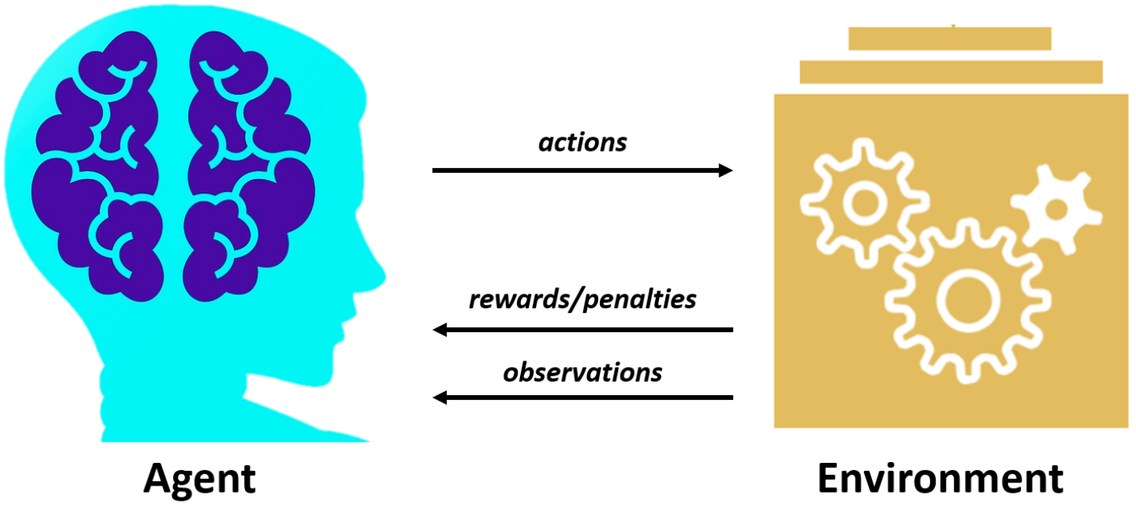
REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Fouad Trad
Machine Learning Engineer

Chapter 1

Introduction to reinforcement learning

- Fundamentals of RL
- RL framework
- Gymnasium



Chapter 2

Model-based learning

- Markove Decision Process
- Policies, value functions
- Policy iteration, value iteration

Chapter 3

Model-Free Learning

- Monte Carlo methods
- Temporal difference learning
  - SARSA
  - Q-learning

Chapter 4

Advanced strategies in model-free RL

- Expected SARSA
- Double Q-learning
- Exploration-exploitation
- Multi-armed bandits

Next steps

- Dive into advanced RL topics
- Deep RL
- Exploring more complex environments
- Creating your own environments

Congratulations!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON