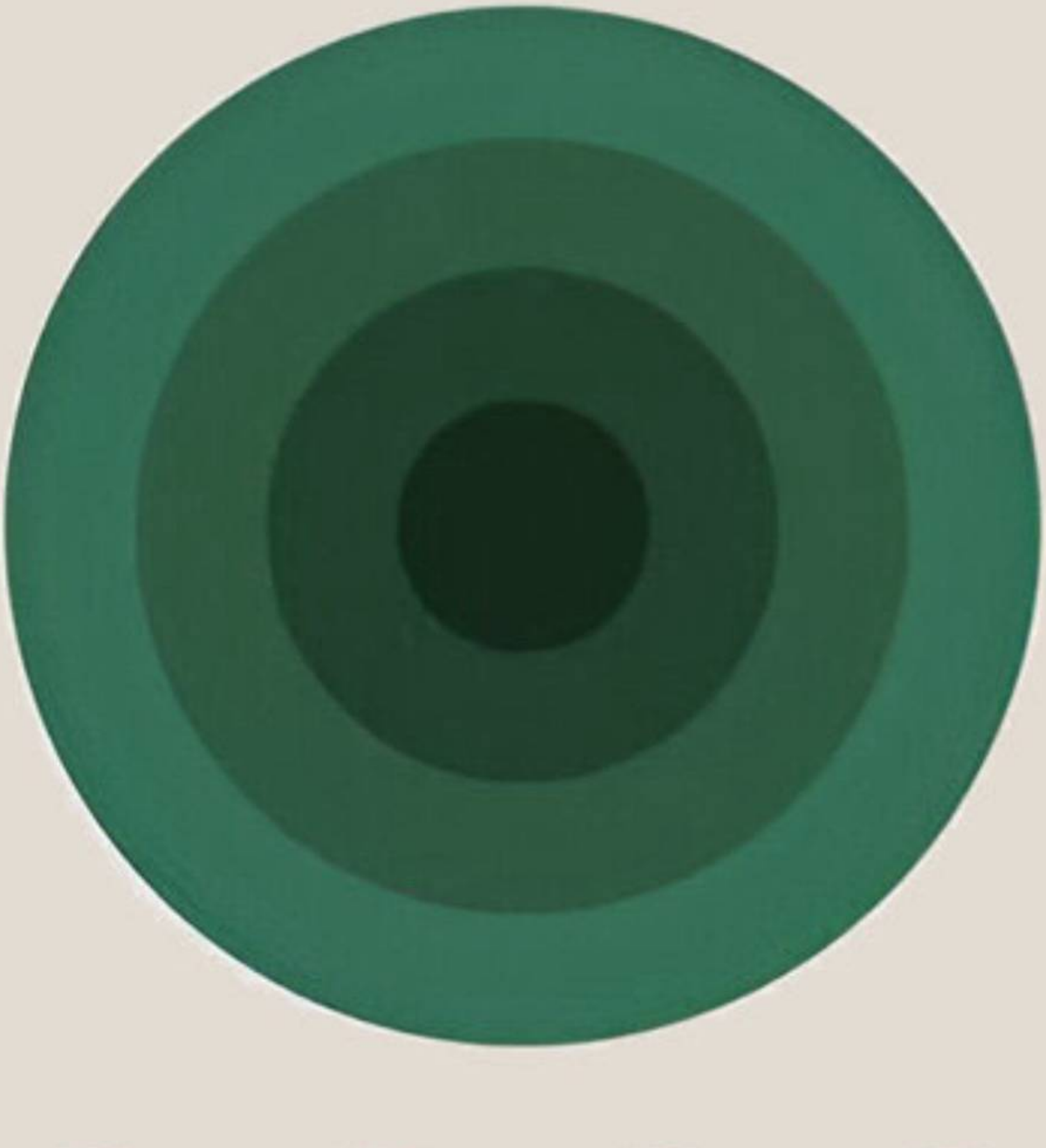
Reinforcement Learning with Python

With Code Examples



save for later

Introduction to Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment. The agent receives rewards or penalties based on its actions, aiming to maximize cumulative rewards over time. This process mimics how humans and animals learn through trial and error.

```python
import gym
import numpy as np

env = gym.make('CartPole-v1')
n_episodes = 1000
max_steps = 500

for episode in range(n_episodes):
    state = env.reset()
    for step in range(max_steps):
        action = env.action_space.sample()  # Random action
        next_state, reward, done, _ = env.step(action)
        if done:
            break
    print(f"Episode {episode + 1} completed in {step + 1} steps")
```

Swipe next →

Key Components of Reinforcement Learning

The main components of RL are the agent, environment, state, action, and reward. The agent is the learner that interacts with the environment. The environment is the world in which the agent operates. The state represents the current situation of the agent in the environment. Actions are the decisions the agent can make, and rewards provide feedback on the quality of those actions.

Swipe next ⟶

save for later

Example

```python
class Agent:
    def __init__(self, action_space):
        self.action_space = action_space

    def choose_action(self, state):
        return self.action_space.sample()

class Environment:
    def __init__(self):
        self.state = 0

    def step(self, action):
        if action == 1:
            self.state += 1
        else:
            self.state -= 1
        reward = 1 if self.state == 5 else 0
        done = abs(self.state) >= 5
        return self.state, reward, done

agent = Agent(gym.spaces.Discrete(2))
env = Environment()

state = env.state
for _ in range(10):
    action = agent.choose_action(state)
    next_state, reward, done = env.step(action)
    print(f"State: {state}, Action: {action}, Next State: {next_state}, Reward:
{reward}")
    state = next_state
    if done:
        break
```

Swipe next ⟶

The RL Process

save for later

The RL process is a continuous cycle of interaction between the agent and the environment. The agent observes the current state, chooses an action, and receives a reward and the next state from the environment. This cycle repeats until a terminal state is reached or a maximum number of steps is completed.

```python
import random

class SimpleAgent:
    def __init__(self, n_actions):
        self.n_actions = n_actions

    def choose_action(self, state):
        return random.randint(0, self.n_actions - 1)

class SimpleEnvironment:
    def __init__(self):
        self.state = 0

    def step(self, action):
        self.state += action - 1
        reward = -abs(self.state)
        done = abs(self.state) >= 5
        return self.state, reward, done

agent = SimpleAgent(3)
env = SimpleEnvironment()

state = env.state
total_reward = 0

for _ in range(20):
    action = agent.choose_action(state)
    next_state, reward, done = env.step(action)
    total_reward += reward
    print(f"State: {state}, Action: {action}, Next State: {next_state}, Reward:
{reward}")
    state = next_state
    if done:
        break

print(f"Total Reward: {total_reward}")
```

Swipe next →

follow for more

Markov Decision Processes (MDPs)

Markov Decision Processes provide a mathematical framework for modeling decision-making in RL. An MDP consists of a set of states, actions, transition probabilities, and rewards. The Markov property states that the next state depends only on the current state and action, not on the history of previous states and actions.

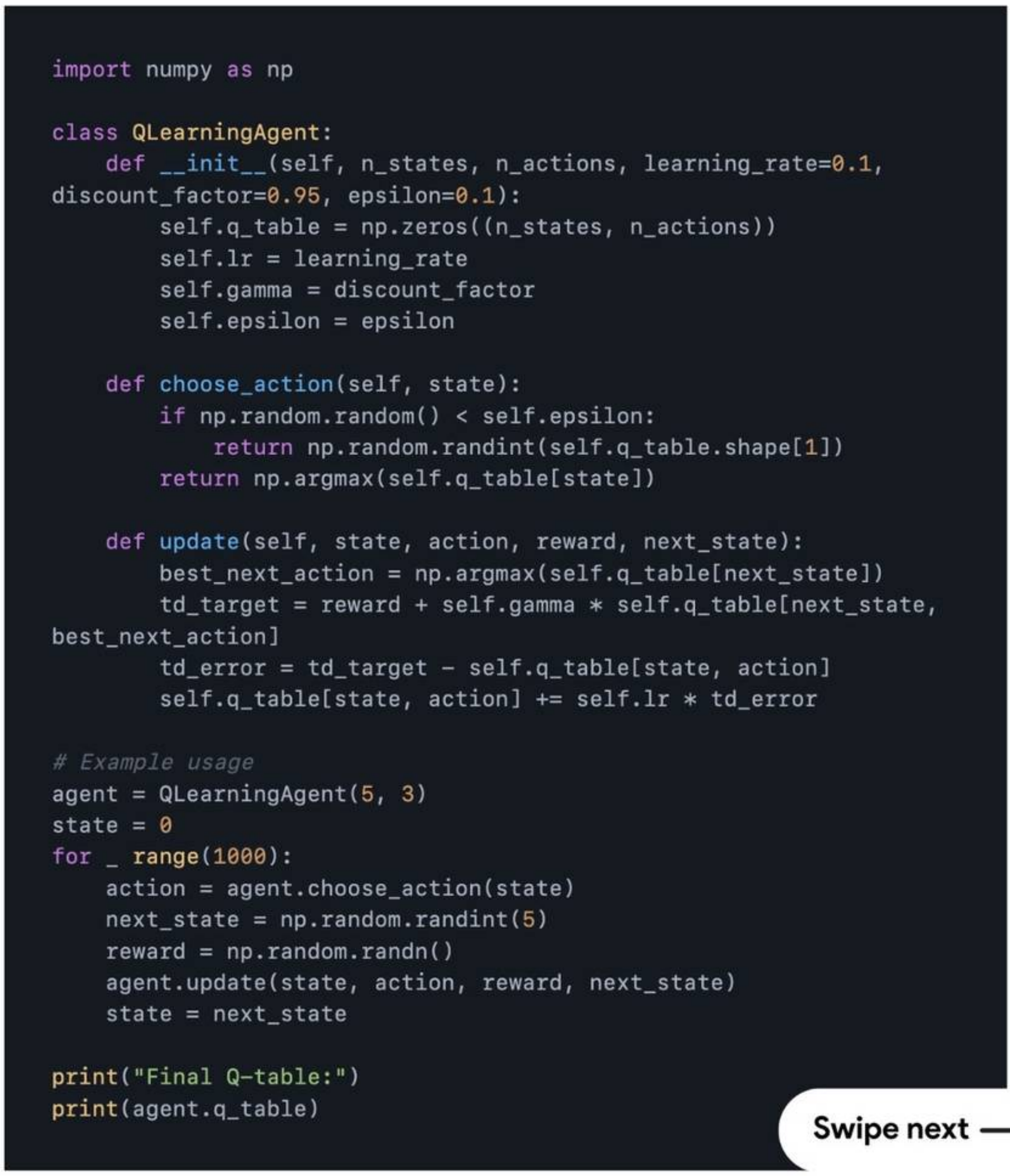
```python
import numpy as np

class SimpleMDP:
    def __init__(self, n_states, n_actions):
        self.n_states = n_states
        self.n_actions = n_actions
        self.transition_probs = np.random.rand(n_states, n_actions, n_states)
        self.transition_probs /= self.transition_probs.sum(axis=2, keepdims=True)
        self.rewards = np.random.randn(n_states, n_actions, n_states)

    def step(self, state, action):
        next_state = np.random.choice(self.n_states, p=self.transition_probs[state,
action])
        reward = self.rewards[state, action, next_state]
        return next_state, reward

mdp = SimpleMDP(5, 3)
state = 0

for _ in range(10):
    action = np.random.randint(mdp.n_actions)
    next_state, reward = mdp.step(state, action)
    print(f"State: {state}, Action: {action}, Next State: {next_state}, Reward:
{reward:.2f}")
    state = next_state
```

Swipe next ⟶

save for later

Q-Learning: A Value-Based RL Algorithm

Q-Learning is a popular value-based RL algorithm that learns to estimate the quality of actions in different states. It maintains a Q-table that stores the expected cumulative reward for each state-action pair. The agent uses this table to make decisions, balancing exploration and exploitation.



```python
import numpy as np

class QLearningAgent:
    def __init__(self, n_states, n_actions, learning_rate=0.1,
discount_factor=0.95, epsilon=0.1):
        self.q_table = np.zeros((n_states, n_actions))
        self.lr = learning_rate
        self.gamma = discount_factor
        self.epsilon = epsilon

    def choose_action(self, state):
        if np.random.random() < self.epsilon:
            return np.random.randint(self.q_table.shape[1])
        return np.argmax(self.q_table[state])

    def update(self, state, action, reward, next_state):
        best_next_action = np.argmax(self.q_table[next_state])
        td_target = reward + self.gamma * self.q_table[next_state,
best_next_action]
        td_error = td_target - self.q_table[state, action]
        self.q_table[state, action] += self.lr * td_error

# Example usage
agent = QLearningAgent(5, 3)
state = 0
for _ range(1000):
    action = agent.choose_action(state)
    next_state = np.random.randint(5)
    reward = np.random.randn()
    agent.update(state, action, reward, next_state)
    state = next_state

print("Final Q-table:")
print(agent.q_table)
```

Swipe next →

save for later

Policy Gradient Methods

Policy gradient methods are another class of RL algorithms that directly learn the policy without maintaining a value function. These methods optimize the policy by estimating the gradient of the expected cumulative reward with respect to the policy parameters. REINFORCE is a simple policy gradient algorithm.

```python
import numpy as np

class REINFORCEAgent:
    def __init__(self, n_states, n_actions, learning_rate=0.01):
        self.n_actions = n_actions
        self.lr = learning_rate
        self.theta = np.zeros((n_states, n_actions))

    def softmax(self, x):
        exp_x = np.exp(x - np.max(x))
        return exp_x / exp_x.sum()

    def choose_action(self, state):
        probs = self.softmax(self.theta[state])
        return np.random.choice(self.n_actions, p=probs)

    def update(self, episode):
        for t, (state, action, reward) in enumerate(episode):
            G = sum([r for (_, _, r) in episode[t:]])
            probs = self.softmax(self.theta[state])
            grad = np.zeros_like(self.theta[state])
            grad[action] = 1
            grad -= probs
            self.theta[state] += self.lr * G * grad

# Example usage
agent = REINFORCEAgent(5, 3)
episode = [(0, 1, 1), (2, 0, -1), (1, 2, 2)]  # [(state, action, reward), ...]
agent.update(episode)

print("Updated policy parameters:")
print(agent.theta)
```

Swipe next ⟶

Deep Q-Networks (DQN)

Deep Q-Networks combine Q-learning with deep neural networks to handle high-dimensional state spaces. DQNs use a neural network to approximate the Q-function, allowing them to generalize across similar states and handle complex environments like Atari games.

Swipe next ⟶

Example

save for later

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

class DQN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_dim, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

class DQNAgent:
    def __init__(self, state_dim, action_dim, learning_rate=0.001, gamma=0.99,
epsilon=0.1):
        self.q_network = DQN(state_dim, action_dim)
        self.target_network = DQN(state_dim, action_dim)
        self.target_network.load_state_dict(self.q_network.state_dict())
        self.optimizer = optim.Adam(self.q_network.parameters(), lr=learning_rate)
        self.gamma = gamma
        self.epsilon = epsilon

    def choose_action(self, state):
        if np.random.random() < self.epsilon:
            return np.random.randint(self.q_network.fc3.out_features)
        with torch.no_grad():
            q_values = self.q_network(torch.FloatTensor(state))
            return q_values.argmax().item()

    def update(self, state, action, reward, next_state, done):
        state = torch.FloatTensor(state)
        next_state = torch.FloatTensor(next_state)
        q_values = self.q_network(state)
        next_q_values = self.target_network(next_state)

        q_value = q_values[action]
        next_q_value = next_q_values.max()
        expected_q_value = reward + self.gamma * next_q_value * (1 - done)

        loss = nn.MSELoss()(q_value, expected_q_value.detach())
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def update_target_network(self):
        self.target_network.load_state_dict(self.q_network.state_dict())

# Example usage
agent = DQNAgent(4, 2)  # 4 state dimensions, 2 actions
state = np.random.rand(4)
action = agent.choose_action(state)
next_state = np.random.rand(4)
reward = 1
done = False
agent.update(state, action, reward, next_state, done)
```

Swipe next →

save for later

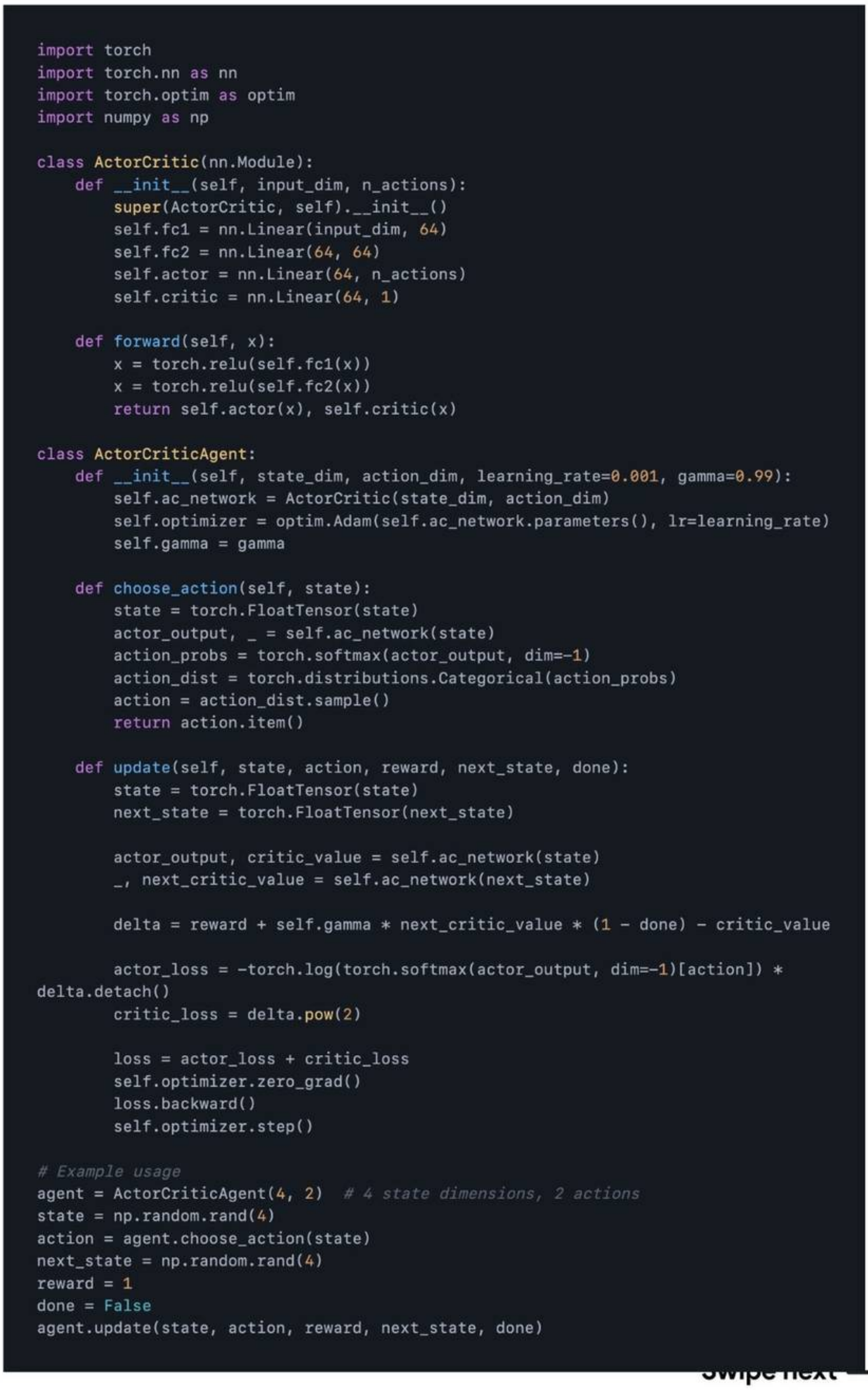
Actor-Critic Methods

Actor-Critic methods combine the strengths of both value-based and policy-based approaches. They use two networks: an actor that learns the policy, and a critic that estimates the value function. This combination often leads to more stable and efficient learning.

Swipe next ⟶

follow for more

Example



```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

class ActorCritic(nn.Module):
    def __init__(self, input_dim, n_actions):
        super(ActorCritic, self).__init__()
        self.fc1 = nn.Linear(input_dim, 64)
        self.fc2 = nn.Linear(64, 64)
        self.actor = nn.Linear(64, n_actions)
        self.critic = nn.Linear(64, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.actor(x), self.critic(x)

class ActorCriticAgent:
    def __init__(self, state_dim, action_dim, learning_rate=0.001, gamma=0.99):
        self.ac_network = ActorCritic(state_dim, action_dim)
        self.optimizer = optim.Adam(self.ac_network.parameters(), lr=learning_rate)
        self.gamma = gamma

    def choose_action(self, state):
        state = torch.FloatTensor(state)
        actor_output, _ = self.ac_network(state)
        action_probs = torch.softmax(actor_output, dim=-1)
        action_dist = torch.distributions.Categorical(action_probs)
        action = action_dist.sample()
        return action.item()

    def update(self, state, action, reward, next_state, done):
        state = torch.FloatTensor(state)
        next_state = torch.FloatTensor(next_state)

        actor_output, critic_value = self.ac_network(state)
        _, next_critic_value = self.ac_network(next_state)

        delta = reward + self.gamma * next_critic_value * (1 - done) - critic_value

        actor_loss = -torch.log(torch.softmax(actor_output, dim=-1)[action]) *
delta.detach()
        critic_loss = delta.pow(2)

        loss = actor_loss + critic_loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

# Example usage
agent = ActorCriticAgent(4, 2)  # 4 state dimensions, 2 actions
state = np.random.rand(4)
action = agent.choose_action(state)
next_state = np.random.rand(4)
reward = 1
done = False
agent.update(state, action, reward, next_state, done)
```

Example

follow for more

```python
import numpy as np

class ExplorationAgent:
    def __init__(self, n_actions, method='epsilon_greedy'):
        self.n_actions = n_actions
        self.method = method
        self.q_values = np.zeros(n_actions)
        self.action_counts = np.zeros(n_actions)
        self.total_steps = 0
        self.epsilon = 0.1
        self.temperature = 1.0

    def choose_action(self):
        self.total_steps += 1
        if self.method == 'epsilon_greedy':
            if np.random.random() < self.epsilon:
                return np.random.randint(self.n_actions)
            return np.argmax(self.q_values)
        elif self.method == 'softmax':
            probs = np.exp(self.q_values / self.temperature)
            probs /= np.sum(probs)
            return np.random.choice(self.n_actions, p=probs)
        elif self.method == 'ucb':
            ucb_values = self.q_values + np.sqrt(2 * np.log(self.total_steps) /
(self.action_counts + 1e-5))
            return np.argmax(ucb_values)

    def update(self, action, reward):
        self.action_counts[action] += 1
        self.q_values[action] += (reward - self.q_values[action]) /
self.action_counts[action]

# Example usage
agent = ExplorationAgent(5, method='softmax')
rewards = [0, 0.2, 0.5, 0.1, 1.0]  # Example rewards for each action

for _ in range(1000):
    action = agent.choose_action()
    reward = rewards[action]
    agent.update(action, reward)

print("Final Q-values:", agent.q_values)
```

Swipe next ⟶

save for later

Exploration vs. Exploitation

The exploration-exploitation dilemma is a fundamental challenge in RL. Exploration involves trying new actions to gather information about the environment, while exploitation means using known information to maximize rewards. Balancing these aspects is crucial for effective learning. Common strategies include epsilon-greedy, softmax exploration, and upper confidence bound (UCB) algorithms.

Swipe next ⟶

save for later

Function Approximation in RL

Function approximation allows RL algorithms to handle large or continuous state spaces by generalizing from observed states to unseen ones. This is typically achieved using neural networks or other parametric models to represent value functions or policies.

Swipe next ⟶

Example

follow for more

```python
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

class ValueNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(ValueNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

class FunctionApproximationAgent:
    def __init__(self, state_dim, learning_rate=0.01):
        self.value_network = ValueNetwork(state_dim, 64)
        self.optimizer = optim.Adam(self.value_network.parameters(),
lr=learning_rate)

    def estimate_value(self, state):
        state_tensor = torch.FloatTensor(state)
        return self.value_network(state_tensor).item()

    def update(self, state, target_value):
        state_tensor = torch.FloatTensor(state)
        predicted_value = self.value_network(state_tensor)
        loss = nn.MSELoss()(predicted_value, torch.FloatTensor([target_value]))
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

# Example usage
agent = FunctionApproximationAgent(4)  # 4-dimensional state space
state = np.random.rand(4)
target_value = 10.0  # Example target value

for _ in range(1000):
    agent.update(state, target_value)

print("Estimated value:", agent.estimate_value(state))
```

Swipe next ⟶

Multi-Agent Reinforcement Learning

save for later

Multi-Agent Reinforcement Learning (MARL) extends RL to environments with multiple agents. These agents can be cooperative, competitive, or a mix of both. MARL introduces new challenges such as non-stationarity, coordination, and credit assignment.

```python
import numpy as np

class SimpleMARL:
    def __init__(self, n_agents, n_actions):
        self.n_agents = n_agents
        self.n_actions = n_actions
        self.q_values = np.zeros((n_agents, n_actions))

    def choose_actions(self, epsilon=0.1):
        actions = []
        for agent in range(self.n_agents):
            if np.random.random() < epsilon:
                actions.append(np.random.randint(self.n_actions))
            else:
                actions.append(np.argmax(self.q_values[agent]))
        return actions

    def update(self, actions, rewards, learning_rate=0.1):
        for agent in range(self.n_agents):
            self.q_values[agent, actions[agent]] += learning_rate * (rewards[agent]
- self.q_values[agent, actions[agent]])

# Example usage
marl = SimpleMARL(2, 3)  # 2 agents, 3 actions each

for _ in range(1000):
    actions = marl.choose_actions()
    rewards = np.random.rand(2)  # Example rewards
    marl.update(actions, rewards)

print("Final Q-values:")
print(marl.q_values)
```

Swipe next ⟶

Hierarchical Reinforcement Learning

follow for more

Hierarchical Reinforcement Learning (HRL) decomposes complex tasks into simpler subtasks, allowing agents to learn and operate at multiple levels of abstraction. This approach can significantly speed up learning and improve generalization in complex environments.

```python
import numpy as np

class HierarchicalAgent:
    def __init__(self, n_high_level_actions, n_low_level_actions):
        self.high_level_policy = np.zeros(n_high_level_actions)
        self.low_level_policies = [np.zeros(n_low_level_actions) for _ in
range(n_high_level_actions)]

    def choose_high_level_action(self, epsilon=0.1):
        if np.random.random() < epsilon:
            return np.random.randint(len(self.high_level_policy))
        return np.argmax(self.high_level_policy)

    def choose_low_level_action(self, high_level_action, epsilon=0.1):
        if np.random.random() < epsilon:
            return
np.random.randint(len(self.low_level_policies[high_level_action]))
        return np.argmax(self.low_level_policies[high_level_action])

    def update(self, high_level_action, low_level_action, reward,
learning_rate=0.1):
        self.high_level_policy[high_level_action] += learning_rate * (reward -
self.high_level_policy[high_level_action])
        self.low_level_policies[high_level_action][low_level_action] +=
learning_rate * (reward - self.low_level_policies[high_level_action]
[low_level_action])

# Example usage
agent = HierarchicalAgent(3, 4)  # 3 high-level actions, 4 low-level actions

for _ in range(1000):
    high_action = agent.choose_high_level_action()
    low_action = agent.choose_low_level_action(high_action)
    reward = np.random.rand()  # Example reward
    agent.update(high_action, low_action, reward)

print("High-level policy:", agent.high_level_policy)
print("Low-level policies:", agent.low_level_policies)
```

Swipe next ⟶

follow for more

Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning (HRL) decomposes complex tasks into simpler subtasks, allowing agents to learn and operate at multiple levels of abstraction. This approach can significantly speed up learning and improve generalization in complex environments.

```
import numpy as np

class HierarchicalAgent:
    def __init__(self, n_high_level_actions, n_low_level_actions):
        self.high_level_policy = np.zeros(n_high_level_actions)
        self.low_level_policies = [np.zeros(n_low_level_actions) for _ in
range(n_high_level_actions)]

    def choose_high_level_action(self, epsilon=0.1):
        if np.random.random() < epsilon:
            return np.random.randint(len(self.high_level_policy))
        return np.argmax(self.high_level_policy)

    def choose_low_level_action(self, high_level_action, epsilon=0.1):
        if np.random.random() < epsilon:
            return
np.random.randint(len(self.low_level_policies[high_level_action]))
        return np.argmax(self.low_level_policies[high_level_action])

    def update(self, high_level_action, low_level_action, reward,
learning_rate=0.1):
        self.high_level_policy[high_level_action] += learning_rate * (reward -
self.high_level_policy[high_level_action])
        self.low_level_policies[high_level_action][low_level_action] +=
learning_rate * (reward - self.low_level_policies[high_level_action]
[low_level_action])

# Example usage
agent = HierarchicalAgent(3, 4)  # 3 high-level actions, 4 low-level actions

for _ in range(1000):
    high_action = agent.choose_high_level_action()
    low_action = agent.choose_low_level_action(high_action)
    reward = np.random.rand()  # Example reward
    agent.update(high_action, low_action, reward)

print("High-level policy:", agent.high_level_policy)
print("Low-level policies:", agent.low_level_policies)
```

Swipe next ⟶

save for later

Inverse Reinforcement Learning

Inverse Reinforcement Learning (IRL) aims to recover the reward function of an agent given its observed behavior. This is useful in scenarios where the reward function is unknown or difficult to specify, such as in robotic imitation learning or autonomous driving.

```python
import numpy as np

class SimpleIRL:
    def __init__(self, n_states, n_actions):
        self.n_states = n_states
        self.n_actions = n_actions
        self.reward_weights = np.random.rand(n_states)
        self.feature_counts = np.zeros(n_states)

    def estimate_reward(self, state):
        return self.reward_weights[state]

    def update(self, expert_trajectory, learning_rate=0.01):
        # Compute feature counts from expert trajectory
        expert_counts = np.zeros(self.n_states)
        for state, _ in expert_trajectory:
            expert_counts[state] += 1

        # Update reward weights
        grad = expert_counts - self.feature_counts
        self.reward_weights += learning_rate * grad

        # Update feature counts
        self.feature_counts = np.zeros(self.n_states)
        for state, _ in expert_trajectory:
            self.feature_counts[state] += 1

# Example usage
irl = SimpleIRL(5, 2)  # 5 states, 2 actions

# Simulated expert trajectory
expert_trajectory = [(0, 1), (1, 0), (2, 1), (3, 0), (4, 1)]

for _ in range(100):
    irl.update(expert_trajectory)

print("Estimated reward weights:", irl.reward_weights)
```

Swipe next ⟶

Real-life Applications of Reinforcement Learning

Reinforcement Learning has found applications in various domains, demonstrating its versatility and power. Two prominent examples are:

1. Game Playing: RL has achieved superhuman performance in complex games like Go (AlphaGo) and Dota 2. These successes showcase RL's ability to learn intricate strategies in high-dimensional state spaces.
2. Robotics: RL enables robots to learn complex motor skills through trial and error, such as grasping objects or walking. This approach allows robots to adapt to new environments and tasks without explicit programming.

Swipe next ⟶

save for later

Example

```python
import numpy as np

class SimpleRobot:
    def __init__(self, n_joints, n_actions):
        self.n_joints = n_joints
        self.n_actions = n_actions
        self.q_table = np.zeros((n_joints, n_actions))

    def choose_action(self, joint, epsilon=0.1):
        if np.random.random() < epsilon:
            return np.random.randint(self.n_actions)
        return np.argmax(self.q_table[joint])

    def update(self, joint, action, reward, learning_rate=0.1):
        self.q_table[joint, action] += learning_rate * (reward -
self.q_table[joint, action])

# Example usage: Robot learning to grasp
robot = SimpleRobot(3, 4)  # 3 joints, 4 actions per joint

for episode in range(1000):
    total_reward = 0
    for joint in range(robot.n_joints):
        action = robot.choose_action(joint)
        reward = np.random.randn()  # Simulated reward
        robot.update(joint, action, reward)
        total_reward += reward

    if episode % 100 == 0:
        print(f"Episode {episode}, Total Reward: {total_reward}")

print("Final Q-table:")
print(robot.q_table)
```

Swipe next ⟶

Additional Resources

For those interested in diving deeper into Reinforcement Learning, here are some valuable resources:

1. "Reinforcement Learning: An Introduction" by Richard S. Sutton and Andrew G. Barto (2nd Edition, 2018) ArXiv link: https://arxiv.org/abs/1603.02199
2. "Deep Reinforcement Learning: An Overview" by Yuxi Li (2017) ArXiv link: https://arxiv.org/abs/1701.07274
3. "A Survey of Deep Reinforcement Learning in Video Games" by Kai Arulkumaran et al. (2019) ArXiv link: https://arxiv.org/abs/1912.10944
4. OpenAI Gym: A toolkit for developing and comparing reinforcement learning algorithms GitHub repository: https://github.com/openai/gym
5. DeepMind's educational resources on RL: https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver