



Reinforcement Learning Algorithms

Reinforcement Learning (RL) is a powerful paradigm in machine learning where an agent learns to make decisions by interacting with an environment. In this chapter, we delve into the fundamental concepts of RL, exploring the relationship between agents, environments, and rewards. We then introduce Markov Decision Processes (MDPs) as a formal framework to model RL problems. Finally, we explore three prominent algorithms in RL: Q-learning, Deep Q Networks (DQNs), and **Policy Gradients**.

Agents: In RL, an agent is an entity that interacts with an environment to achieve specific goals. The agent takes actions in the environment, receives feedback in the form of rewards, and learns to make better decisions over time. The goal is for the agent to discover a policy, a strategy that maps states to actions, maximizing the cumulative reward.


Environment: The environment represents the external system with which the agent interacts. It encapsulates the dynamics of the system and provides feedback to the agent based on its actions. Environments in RL are often modeled as Markovian, meaning the future state depends only on the current state and action, not on the path taken to reach that state.

Rewards: Rewards are the feedback mechanism for the agent. They quantify the immediate benefit or cost associated with an action taken in a particular state. The agent's objective is to learn a policy that maximizes the cumulative reward over time. Reinforcement learning, therefore, involves finding the optimal policy that leads to the highest possible total reward.

Markov Decision Processes (MDPs)

Markov Decision Processes provide a formal framework for modeling sequential decision-making problems. An MDP is defined by a tuple (S, A, P, R) , where:

- S : Set of states
- A : Set of actions
- P : Transition probability function $P(s'|s, a)$
- R : Reward function $R(s, a, s')$



The transition probability function P defines the probability of transitioning to state s' given that the agent is in state s and takes action a . The reward function R provides the immediate reward received after taking action a in state s and transitioning to state s' .

MDPs assume the Markov property, which states that the future is independent of the past given the present state. This simplifying assumption allows for efficient computation and enables the use of dynamic programming techniques.

Types of Reinforcement algorithm:

- **Q-learning** is a model-free reinforcement learning algorithm used to find the optimal action-selection policy for a given finite Markov decision process.
- **DQNs (Deep Q Networks)** extend Q-learning by incorporating deep neural networks to handle high-dimensional state spaces, such as images.
- **Policy Gradients** aim to directly optimize the policy function that maps states to actions, without explicitly estimating value functions.

Let's discuss the most important algorithm using an example code.

Algorithms in Reinforcement Learning

- **Q-learning:** Q-learning is a model-free RL algorithm that aims to learn the optimal action-value function $Q(s, a)$. The Q-function represents the expected cumulative reward of taking action a in state s and following the optimal policy thereafter. The Q-learning update rule is given by:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [R(s, a, s') + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)]$$

Where:

- α is the learning rate
- γ is the discount factor

Q-learning explores the environment by iteratively updating the Q-values based on the observed rewards. Over time, the agent converges to the optimal Q-function, allowing it to make informed decisions.

Example Code: Q-learning in a Grid World

Consider a simple grid world where the agent navigates to reach a goal. The states are the cells in the grid, and the actions are movements (up, down, left, right). The agent receives a positive reward at the goal and a negative reward for each step. Q-learning enables the agent to learn the optimal policy for reaching the goal efficiently.

```
# Q-learning implementation
import numpy as np

class SimpleEnvironment:
    def __init__(self):
        self.n_states = 5
        self.n_actions = 2
        self.transitions = np.array([[1, 2], [3, 4], [0, 1],
[2, 3], [4, 0]])
        self.rewards = np.array([0, 0, 0, 1, 10])

    def reset(self):
        self.state = np.random.randint(self.n_states)
        return self.state

    def step(self, action):
        next_state = self.transitions[self.state, action]
        reward = self.rewards[next_state]
        done = (next_state == 4) # Terminal state is state
4
        self.state = next_state
        return next_state, reward, done

# Q-learning implementation
def q_learning(env, num_episodes=1000, alpha=0.1, gamma=0.9,
epsilon=0.1):
    num_states = env.n_states
    num_actions = env.n_actions
    Q = np.zeros((num_states, num_actions))
```



```

for episode in range(num_episodes):
    state = env.reset()
    done = False

    while not done:
        # Epsilon-greedy exploration
        if np.random.rand() < epsilon:
            action = np.random.choice(num_actions)
        else:
            action = np.argmax(Q[state, :])

        next_state, reward, done = env.step(action)

        # Q-learning update
        Q[state, action] += alpha * (reward + gamma *
np.max(Q[next_state, :]) - Q[state, action])

        state = next_state

    return Q

# Create a simple environment
env = SimpleEnvironment()

# Apply Q-learning
learned_Q = q_learning(env)

# Print the learned Q-values
print("Learned Q-values:\n", learned_Q)

```

Output:

```

Learned Q-values:
[[77.66844639 33.50809439]
 [56.82378856 88.27026678]
 [49.26261679 78.49009434]
 [69.67746352 47.64594428]
 [87.38607957 36.32515556]]

```

1. **Initialization:** The Q-table (Q) is initialized with zeros, where each entry (state, action) represents the estimated cumulative future reward for taking action action in state state.



2. **Episode Loop:** The algorithm iterates through a fixed number of episodes (num_episodes).
3. **State Reset:** The environment is reset to its initial state at the beginning of each episode.
4. **Time Step Loop:** The algorithm iterates through time steps within each episode until termination.
5. **Exploration-Exploitation Strategy:** At each time step, the algorithm chooses an action using an epsilon-greedy strategy. With probability epsilon, a random action is chosen (exploration), and with probability $1 - \text{epsilon}$, the action with the highest Q-value for the current state is chosen (exploitation).
6. **Action Execution and Environment Interaction:** The chosen action is executed, and the next state and reward are observed from the environment.
7. **Q-learning Update:** The Q-value for the current state-action pair is updated using the Q-learning update equation, which incorporates the observed reward and the maximum Q-value for the next state.
8. **State Transition:** The current state is updated to the next state for the next iteration.
9. **Q-table Return:** The learned Q-table is returned after the specified number of episodes.

The Q-learning update rule is based on the Bellman equation, and it helps the algorithm learn the optimal action-value function by iteratively updating the Q-values based on observed rewards and estimated future rewards. The epsilon-greedy strategy balances exploration and exploitation during the learning process.