

Open in app ↗

Sign up

Sign in

Medium

 Search Write

A Beginner's Guide to Q-Learning: Understanding with a Simple Gridworld Example



Gregory Kovalchuk

Follow

5 min read · Oct 24, 2024



Reinforcement learning (RL) is one of the most exciting fields in machine learning, allowing agents to learn optimal behaviors in uncertain environments. One of the fundamental algorithms in RL is Q-learning — a simple yet powerful method for enabling agents to learn how to make decisions through trial and error.

In this article, we'll break down Q-learning using a simple Python implementation of a gridworld environment. We'll walk through the step-by-step process of how the agent learns and updates its knowledge of the environment to reach a goal.

What is Q-Learning?

Q-learning is a value-based reinforcement learning algorithm. The goal of Q-learning is to learn the optimal action-selection policy for an agent interacting with an environment. The agent learns this by updating a table of

values — called the Q-table — where each entry represents the value of taking a particular action in a given state.

The Q-learning algorithm uses the following formula to update the Q-value for a state-action pair:

$$Q(s, a) = Q(s, a) + \alpha \left[R + \gamma \cdot \max_a Q(s', a') - Q(s, a) \right]$$

Where:

- $Q(s, a)$ is the Q-value for state s and action a .
- α is the learning rate, controlling how much new information overrides old information.
- R is the immediate reward for taking action a in state s .
- γ is the discount factor, representing the importance of future rewards.
- $\max_a Q(s', a')$ is the maximum Q-value for the next state s' , representing the best possible reward achievable from that state.

The Gridworld Environment

Let's illustrate Q-learning using a 4x4 gridworld. The gridworld is a simple environment where an agent must navigate a grid to reach a goal, while avoiding obstacles (walls) that incur penalties.

Here's the layout of the grid:

```
+ - -+ - -+ - -+ - -+
| 0 | 0 | 0 | G | -> Goal = 1 (Reward +10)
+ - -+ - -+ - -+ - -+
| 0 | X | 0 | 0 | -> Wall = X (Reward -1)
+ - -+ - -+ - -+ - -+
```

```

| 0 | 0 | 0 | 0 |
+ - -+ - -+ - -+ - -+
| S | 0 | 0 | 0 | -> Start = (3, 0)
+ - -+ - -+ - -+ - -+

```

- The agent starts at position (3, 0) (bottom-left).
- The goal is located at (0, 3) (top-right), and reaching it gives a reward of +10.
- The walls at (1, 1) provide a negative reward (-1) when hit.
- All other moves result in no reward.

Q-Learning in Python

Let's dive into the Python implementation of the gridworld environment and the Q-learning algorithm.

Environment Setup

```

import numpy as np
import random

# Define the gridworld environment
class GridWorld:
    def __init__(self):
        self.grid = np.array([
            [0, 0, 0, 1], # Goal at (0, 3)
            [0, -1, 0, 0], # Wall with reward -1
            [0, 0, 0, 0],
            [0, 0, 0, 0] # Start at (3, 0)
        ])
        self.start_state = (3, 0)
        self.state = self.start_state

    def reset(self):
        self.state = self.start_state
        return self.state

    def is_terminal(self, state):
        return self.grid[state] == 1 or self.grid[state] == -1

```

```

def get_next_state(self, state, action):
    next_state = list(state)
    if action == 0: # Move up
        next_state[0] = max(0, state[0] - 1)
    elif action == 1: # Move right
        next_state[1] = min(3, state[1] + 1)
    elif action == 2: # Move down
        next_state[0] = min(3, state[0] + 1)
    elif action == 3: # Move left
        next_state[1] = max(0, state[1] - 1)
    return tuple(next_state)

def step(self, action):
    next_state = self.get_next_state(self.state, action)
    reward = self.grid[next_state]
    self.state = next_state
    done = self.is_terminal(next_state)
    return next_state, reward, done

```

The environment is simple. The agent moves up, down, left, or right, and receives rewards based on its location.

Q-Learning Agent

Next, we define the Q-learning agent that interacts with the environment.

```

class QLearningAgent:
    def __init__(self, learning_rate=0.1, discount_factor=0.9, exploration_rate=0.1):
        self.q_table = np.zeros((4, 4, 4)) # Q-values for each state-action pair
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_rate = exploration_rate

    def choose_action(self, state):
        if random.uniform(0, 1) < self.exploration_rate:
            return random.randint(0, 3) # Explore
        else:
            return np.argmax(self.q_table[state]) # Exploit

    def update_q_value(self, state, action, reward, next_state):

```

```
max_future_q = np.max(self.q_table[next_state]) # Best Q-value for next
current_q = self.q_table[state][action]
# Q-learning formula
self.q_table[state][action] = current_q + self.learning_rate * (
    reward + self.discount_factor * max_future_q - current_q
)
```

- The agent has a Q-table initialized to zero.
- The agent uses an epsilon-greedy strategy, meaning it randomly explores with probability epsilon and exploits the current knowledge otherwise.
- The update_q_value function uses the Q-learning update formula to adjust the Q-values after each step.

Training the Agent

Finally, we train the agent by running episodes in the environment.

```
env = GridWorld()
agent = QLearningAgent()

episodes = 1000 # Number of training episodes

for episode in range(episodes):
    state = env.reset() # Reset the environment at the start of each episode
    done = False

    while not done:
        action = agent.choose_action(state) # Choose an action
        next_state, reward, done = env.step(action) # Take the action and observe
        agent.update_q_value(state, action, reward, next_state) # Update Q-value
        state = next_state # Move to the next state
```

In each episode, the agent resets to the start state and takes actions until it either reaches the goal or hits a wall. Over time, the Q-values in the Q-table will converge to the optimal values, leading the agent to consistently choose the best path to the goal.

Understanding the First Few Steps of Learning

Let's walk through the first few steps of learning, illustrating how Q-values are updated based on the agent's interactions with the environment.

Step 1:

- State (3, 0) (Start)
- Choose Action: Right
- Next State: (3, 1)
- Reward: 0

The Q-value update is:

$$Q((3, 0), \text{right}) = 0 + 0.1 [0 + 0.9 \cdot \max(Q((3, 1))) - 0]$$

Since Q(3, 1) values are zero, the Q-value remains zero.

Step 2:

- State (3, 1)
- Choose Action: Up
- Next State: (2, 1)
- Reward: 0

Q-value update:

$$Q((3, 1), \text{up}) = 0 + 0.1 [0 + 0.9 \cdot \max(Q((2, 1))) - 0]$$

Again, the Q-value stays zero.

Step 3:

- State (2, 1)
- Choose Action: Up
- Next State: (1, 1) (Wall)
- Reward: -1

Q-value update:

$$Q((2, 1), \text{up}) = 0 + 0.1 [-1 + 0] = -0.1$$

Now, the Q-value for moving up from state (2, 1) is updated to -0.1, reflecting the negative reward.

Conclusion

This example demonstrates how Q-learning works in a simple gridworld. By continuously updating the Q-values based on the agent's experiences, the agent gradually learns the optimal path to reach the goal.

[Python](#)[Machine Learning](#)[Algorithms](#)[Reinforcement Learning](#)

Written by Gregory Kovalchuk

12 followers · 56 following

[Follow](#)

Data engineer, ML, AI — enthusiast, <https://www.linkedin.com/in/gregory-kovalchuk-5869946a/>


No responses yet



Write a response

What are your thoughts?

More from Gregory Kovalchuk

 Gregory Kovalchuk


A Deep Dive into SQL LATERAL JOIN

In SQL, joins are fundamental operations that allow us to combine data from multiple table...

Nov 6, 2024



23

 Gregory Kovalchuk

Understanding `yield from` in Python Generators: A...

Generators in Python are a powerful tool for creating iterators. They allow you to generat...

May 28, 2024



28

 Gregory Kovalchuk

Using numpy.typing for Type-Safe List Handling in Python

When working with NumPy in Python, ensuring type safety can help catch errors...

Feb 19



11



1

 Gregory Kovalchuk

Troubleshooting `pytest` Hanging: A Comprehensive Guide

When working on Python projects, `pytest` is an indispensable tool for running tests,...

Aug 27, 2024



2

[See all from Gregory Kovalchuk](#)

Recommended from Medium

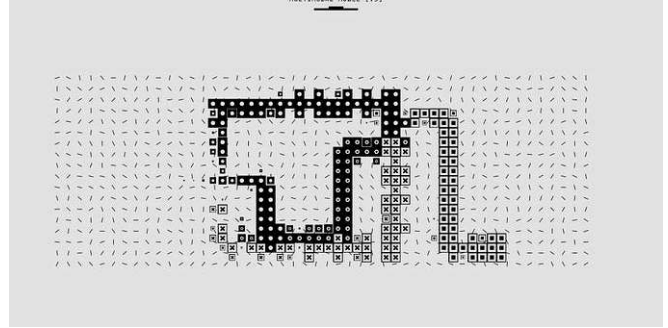


Kaige

How to Test PPO on PettingZoo Turn-Based Game?

This article shows how test ray rllib PPO algorithm on pettingzoo turn-based game....

Mar 1 🖱 50



In Data Science Collective by Oliver S

Planning and Learning in Reinforcement Learning

Dissecting "Reinforcement Learning" by Richard S. Sutton with Custom Python...

★ Feb 11 🖱 284 💬 2



Damini Vadrevu

What are Decision Trees, Random Forest and Gradient Boosting...

An all about Ensemble.



Aakash Chavan Ravindranath, Ph.D

Hidden Markov Model and the Medilian Fund: A Deep Dive into...

Introduction In finance, making informed predictions is key to successful trading. One...

★

Mar 25

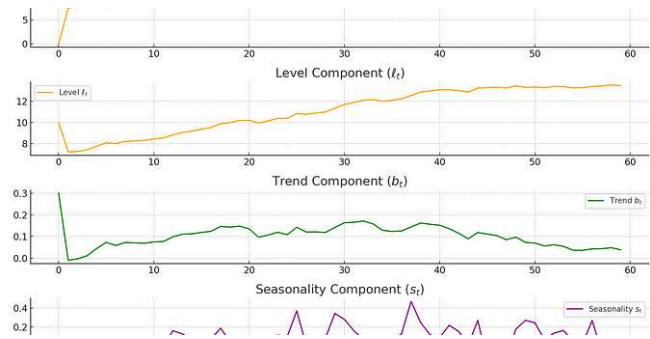
👤 8



★

Mar 2

👤 1



In GoPenAI by Ruth Yang

How ETS and Prophet Really Work: Forecasting Payment Approval...

By Ruth Yang | Data Clarity Series#19

★

Jul 3

👤 4



Ajay Kumar

Mastering Reinforcement Learning: Chapter By Chapter...

Chapter 1: Introduction

★

Apr 18

👤 4



See more recommendations