CS-866 Deep Reinforcement Learning

MDP Solutions - II
Model-Free Learning

Nazar Khan
Department of Computer Science
University of the Punjab

Model-Free Learning: Motivation

- So far: Value Iteration (model-based) computes the policy using the transition model.
- Problem: In many environments, the transition probabilities are **unknown**.
- Solution: Use **model-free algorithms** that learn directly from experience.
- Key milestone: Enabled reinforcement learning to work in real-world problems.

Model-Free Learning: Overview

- Focus: **value-based** model-free algorithms.
- Instead of knowing the transition model:
  - The agent interacts with the environment.
  - Learns from sampled rewards and state transitions.
- Goal: Learn an optimal policy π^* without knowing transition dynamics.

Tabular Value-Based Approaches

Name	Approach
Value Iteration	Model-based enumeration[1,2]
SARSA	On-policy temporal difference model-free[3]
Q-learning	Off-policy temporal difference model-free[4]

[1] Richard Bellman. *Dynamic Programming*. Courier Corporation, 1957, 2013.

[2] Ethem Alpaydin. *Introduction to Machine Learning*. MIT press, 2009.

[3] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Tech. rep. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[4] Christopher JCH Watkins. 'Learning from Delayed Rewards'. PhD thesis. King's College, Cambridge, 1989.

Principles of Model-Free Learning (1/3)

Principle 1: Reward Sampling

- Estimate value functions by sampling rewards from the environment.
- Two main strategies:
  1. **Monte Carlo sampling:** update after full-episode return.
  2. **Temporal Difference (TD) learning:** update after single-step.

Principles of Model-Free Learning (2/3)

Principle 2: Action Selection

- How does the agent decide which action to take?
- Trade-off:
  - **Exploration:** try new actions to discover rewards.
  - **Exploitation:** choose best known action to maximize reward.
- Examples:
  1. Greedy: $a^* = \arg\max_a Q(s, a)$. Never explores.
  2. ϵ-greedy: with probability ϵ, pick random action, otherwise pick greedily.
  3. Softmax: pick action according to its probability.

$$p(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a')/\tau}}$$

Principles of Model-Free Learning (3/3)

Principle 3: Learning from Rewards

- Two ways of using reward feedback:
  1. **On-policy learning:** Learn about the policy you are following (e.g., SARSA).
  2. **Off-policy learning:** Learn about a different (greedy) policy while following another (e.g., Q-learning).
- Leads to powerful learning algorithms that do not need full transition models.

Principle 1: Reward Sampling

Monte Carlo Sampling: Intuition

- Idea: Learn from **complete episodes**.
- Generate a random episode by interacting with the environment.
- Use its return to update the value function at the visited states.
- Named *Monte Carlo* after the famous casino, due to random sampling.

Two Loops in Monte Carlo Learning

1. Loop over time steps of the episode.
2. Loop over many episodes until values converge.

Monte Carlo Sampling

Pseudocode

1. **Initialization:** Start with arbitrary $Q(s, a)$ values.
2. **Episode loop:** Generate many episodes.
3. **Within each episode:**
   - Collect (s, a, r) tuples until terminal state.
4. **Return calculation:** Work backwards to compute G_t for each time step $t \in \{T, T-1, \ldots, 1, 0\}$.

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

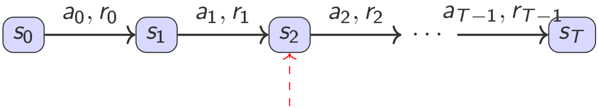
5. **Update rule:**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \underbrace{\left(G_t - Q(s_t, a_t)\right)}_{\text{Monte Carlo Error}}$$

Note

Uses **incremental implementation**, suitable for non-stationary environments.

Monte Carlo Sampling: Illustration



$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

Key Idea

At the end of the episode, compute G_t for each state s_t visited and use it to update the value function.

Monte Carlo Sampling: Code Structure

```
def monte_carlo(n_samples, ep_length, alpha, gamma):
    # 0: initialize
    t = 0; total_t = 0
    Qsa = []

    # sample n_times
    while total_t < n_samples:

        # 1: generate a full episode
        s = env.reset()
        s_ep = []
        a_ep = []
        r_ep = []
        for t in range(ep_length):
            a = select_action(s, Qsa)
            s_next, r, done = env.step(a)
            s_ep.append(s)
            a_ep.append(a)
```

Monte Carlo Sampling: Code Structure

```
            r_ep.append(r)

            total_t += 1
            if done or total_t >= n_times:
                break;
            s = s_next

        # 2: update Q function with a full episode (incremental
        # implementation)
        g = 0.0
        for t in reversed(range(len(a_ep))):
            s = s_ep[t]; a = a_ep[t]
            g = r_ep[t] + gamma * g
            Qsa[s,a] = Qsa[s,a] + alpha * (g - Qsa[s,a])

    return Qsa

def select_action(s, Qsa):
```

Monte Carlo Sampling: Code Structure

```
    # policy is egreedy
    epsilon = 0.1
    if np.random.rand() < epsilon:
        a = np.random.randint(low=0,high=env.n_actions)
    else:
        a = argmax(Qsa[s])
    return a

env = gym.make('Taxi-v3')
monte_carlo(n_samples=10000, ep_length=100, alpha=0.1, gamma
   =0.99)
```

Monte Carlo Sampling: Pros and Cons

Advantages

- Conceptually simple.
- Works without knowing transitions.
- Easy to implement.

Disadvantages

- Must wait until end of episode to update values.
- Inefficient in long episodes.
- High variance in estimates.

Motivation for Next Step

Leads to **Temporal Difference (TD) learning**, which updates values after each step by bootstrapping.

Temporal Difference (TD) Learning

Bootstrapping and Model-Free Updates

- Recall: In **Value Iteration**, Bellman's equation computes values recursively using successor states.
- In model-free RL, we don't have the transition model $T(s, a, s')$.
- But we can still refine estimates step by step from sampled experience.
- This is called **bootstrapping**: refine old estimates with new updates.

Idea

Use the difference between successive time steps to update the current value estimate.

Bootstrapping Explained

- "Pull yourself up by your bootstraps" $\rightarrow$ refine estimates iteratively.
- Bellman recursion is itself a form of bootstrapping.
  - The value of a state depends on the values of its successor states.

$$V(s) = \max_a \mathbb{E}[r + \gamma V(s')]$$

- Model-based RL: compute expectation $\mathbb{E}$ using transition probabilities.
- Model-free RL: use sample transitions (s, r, s') instead of knowing transition probabilities.

Key Question

How can we compute the value of a state using only sampled transitions?

TD Learning Update Rule

From Sutton, 1988

$$V(s_t) \leftarrow V(s_t) + \alpha\Big[\underbrace{r_{t+1} + \gamma V(s_{t+1}) - V(s_t)}_{\text{temporal difference error } \delta}\Big]$$

Interpretation

- $V(s_t)$ predicts future reward from *now*.
- $V(s_{t+1})$ predicts future reward from the *next time step*.
- $r_{t+1} + \gamma V(s_{t+1})$ represents a *one-step lookahead* estimate of the total return from s_t.
- $\delta = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is the difference between the estimate after looking one-step ahead and the estimate now.
- *TD Learning* updates current estimate by adding the (scaled) *temporal difference error*.

Alternative Formulation of TD Learning

$$V(s_t) \leftarrow \alpha\big[r_{t+1} + \gamma V(s_{t+1})\big] + (1-\alpha)V(s_t)$$

- Weighted average between:
  - $V(s_t)$: current estimate of future reward
  - $r_{t+1} + \gamma V(s_{t+1})$: new estimate of future reward *after looking one-step into the future*.
- **No transition model needed** $\Rightarrow$ model-free!

Implementation

Learning takes place on Q, not V

While the TD update equation is theoretically introduced in terms of V, in learning implementations, it's applied to Q.

- Leads to two different TD Learning implementations.
  1. **SARSA (On-policy learning)**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \underbrace{Q(s_{t+1}, a_{t+1})}_{\text{reward under } \pi} - Q(s_t, a_t) \right]$$

  2. **Q-Learning (Off-policy learning)**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \underbrace{\max_{a'} Q(s_{t+1}, a')}_{\text{reward under } \pi^*} - Q(s_t, a_t) \right]$$

SARSA

```
# Temporal Difference SARSA
Q = np.zeros((n_states, n_actions))

for episode in range(n_episodes):
    s = env.reset()
    a = epsilon_greedy(Q, s, epsilon)
    done = False

    while not done:
        s_next, r, done, _ = env.step(a)
        a_next = epsilon_greedy(Q, s_next, epsilon)

        # TD update (SARSA)
        Q[s,a] = Q[s,a] + alpha * (
            r + gamma * Q[s_next, a_next] - Q[s,a]
        )

        s, a = s_next, a_next
```

Q-learning

```
# Temporal Difference Q-learning
Q = np.zeros((n_states, n_actions))

for episode in range(n_episodes):
    s = env.reset()
    done = False

    while not done:
        a = epsilon_greedy(Q, s, epsilon)
        s_next, r, done, _ = env.step(a)

        # TD update (Q-learning)
        Q[s,a] = Q[s,a] + alpha * (
            r + gamma * np.max(Q[s_next,:]) - Q[s,a]
        )

        s = s_next
```

Advantages and Impact

- TD combines ideas of **Monte Carlo** (sample-based) and **Dynamic Programming** (bootstrapping).
- More efficient than full-episode Monte Carlo: updates can occur at each time step.
- Enabled model-free learning in complex domains.

Famous Application

TD-Gammon[a] beat world champions in Backgammon using TD learning.

[a] Gerald Tesauro. 'Temporal difference learning and TD-Gammon'. In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

Bias–Variance Trade-off

Monte Carlo vs. Temporal Difference

- Key difference between Monte Carlo (MC) and Temporal Difference (TD):
  - MC: no bootstrapping
  - TD: uses bootstrapping
- Bootstrapping introduces a trade-off between **bias** and **variance**.

Monte Carlo Characteristics

- MC waits until the **end of an episode** to update values.
- Uses many random action choices $\rightarrow$ updates are **unbiased**.
- Randomness across full episodes $\rightarrow$ **high variance** in returns.

Monte Carlo

Low Bias / High Variance

Temporal Difference Characteristics

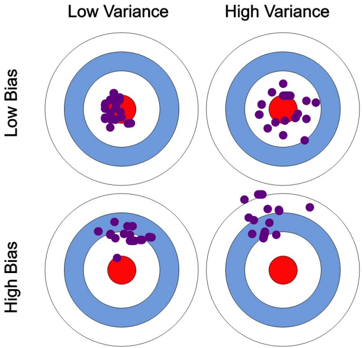
- TD updates the value function **after every step**.
- Old values are reused in updates $\rightarrow$ **bias is introduced**.
- But because updates are incremental, variance is **lower**.

Temporal Difference

High Bias / Low Variance

Bias–Variance Illustrated

The Dartboard Analogy

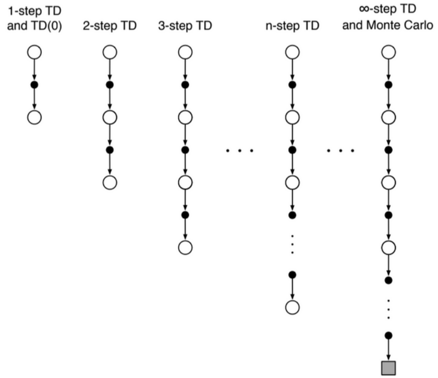


- High bias $\rightarrow$ shots far from target center.
- High variance $\rightarrow$ shots spread out.
- Goal: balance accuracy (low bias) and consistency (low variance).

Finding Middle Ground: N-step Methods

- Can we combine the best of MC and TD?
- Idea: use n-**step returns**.
  - Not a full episode (like MC).
  - Not just one step (like TD).
  - Instead: update after n steps.
- Results in **medium bias** and **medium variance**.

MC, TD, and N-step Compared



- Monte Carlo $\rightarrow$ full-episode updates.
- TD $\rightarrow$ single-step bootstrapping.
- n-step $\rightarrow$ compromise: updates after n steps.

Finding a Policy from Value Functions
Value-based Learning

- Goal of reinforcement learning: construct a policy π with the highest cumulative reward.
- In the **value-based approach**, we use $V(s)$ or $Q(s,a)$ to guide action selection.
- In discrete action spaces:
  - At least one action has the highest value.
  - That action becomes the best choice in the policy.

Optimal Policy

$$\pi^\star = \max_{\pi} V^\pi(s) = \max_{a,\pi} Q^\pi(s,a)$$

$$a^\star = \arg\max_{a \in A} Q^\star(s,a)$$

Value-based Policy Extraction

- The optimal policy sequence $\pi^\star(s)$ is recovered by:
  1. Learning $Q^\star(s, a)$ or $V^\star(s)$.
  2. Selecting $a^\star = \arg\max_a Q^\star(s, a)$ at each state.
- This is why methods are called **value-based**: the policy comes from values.

Key Idea

Value function $\longrightarrow$ Best actions $\longrightarrow$ Policy

Principle 2: Action Selection

Exploration in Model-free RL

- In model-free settings, no transition model T is available.
- Agents must sample the environment directly.
- Sampling is often **expensive** (e.g., real-world robot actions).
- Hence, smart action selection is needed to:
  - Avoid wasting samples.
  - Find good policies faster.

Greedy Action Selection

- Idea: always select the action with the current highest Q-value.
- Pros: exploits current knowledge.
- Cons:
  - **Short-sighted**: may converge to local maxima.
  - High bias: based on few samples.
  - Risk of circular reinforcement: policy only reinforces what it already does.

Problem

A purely greedy agent may miss better long-term strategies.

Exploration vs. Exploitation

- To avoid local maxima, agents must sometimes try less-known actions.
- This introduces the **exploration–exploitation trade-off**:
  - **Exploitation**: use current best policy (max Q-values).
  - **Exploration**: try random actions to gather new information.
- Smart policies mix both to balance:
  - Learning speed.
  - Policy quality.

Preview

The ϵ-greedy strategy is one common way to achieve this balance.

Bandit Theory: The Exploration/Exploitation Trade-off

- Fundamental question:
  - How to obtain the most reliable information at the least cost?
- Studied extensively in literature for single-step decision problems
  - Known as the **multi-armed bandit problem**.
- A *bandit* $\Rightarrow$ casino slot machine with many arms
  - Each arm has an unknown payout probability
  - Each trial costs a coin
  - Goal: Find strategy to identify the best arm with minimal cost

Bandit Theory as Reinforcement Learning

- Multi-armed bandit is:
  - A single-state, single-decision RL problem
  - A one-step, non-sequential decision-making problem
- Actions $\Rightarrow$ arms of the bandit
- Simplified model $\Rightarrow$ allows in-depth study of exploration vs. exploitation

Bandit Applications: Clinical Trials

- Example: Testing new drugs in clinical trials
- Bandit $\Rightarrow$ the trial setup
- Arms $\Rightarrow$ choice of assigning subjects to:
  - Experimental drug
  - Placebo
- Serious implication: **human lives at stake**

Fixed vs. Adaptive Trials

Fixed Randomized Controlled Trial

- Group sizes fixed in advance
- Duration and confidence interval fixed
- Risk: More people exposed to harmful drug or deprived of beneficial drug

Adaptive Trial (Bandit Setup)

- Group sizes adapt during trial
- More subjects get promising drug
- Fewer subjects get ineffective/harmful drug

Adaptive Clinical Trial Illustration

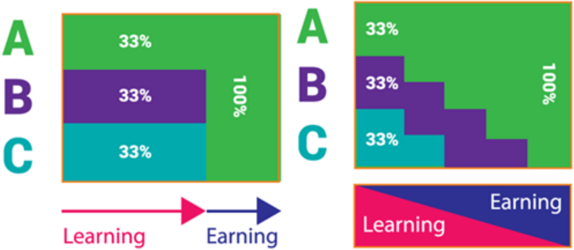


Figure: Adaptive trial: balancing exploration vs. exploitation[5]

[5] Abhishek. *Multi-Arm Bandits: a potential alternative to A/B tests* *https://medium.com/brillio-data-science/multi-arm-bandits-a-potential-alternative-to-a-b-tests-a647d9bf2a7e*. 2019.

ϵ-Greedy Exploration

- Simple pragmatic strategy:
  - Choose greedy action (highest estimated value) most of the time
  - With probability ϵ, explore another random action
- Example: $\epsilon = 0.1$
  - 90% exploit best-known action
  - 10% explore random actions
- ϵ-greedy is a **soft policy**: non-zero probability for all actions

Exploration/Exploitation Trade-off

- Central concept in reinforcement learning
- Determines:
  - How much confidence in outcomes
  - How quickly variance is reduced
- Variants:
  - **Adaptive** ϵ: decay over time or based on statistics
  - Add **Dirichlet noise**[6] to prior probabilities of actions for exploration
  - Use **Thompson sampling**[7] for Bayesian exploration

[6] Samuel Kotz, Narayanaswamy Balakrishnan, and Norman L Johnson. *Continuous Multivariate Distributions, Volume 1: Models and Applications*. John Wiley & Sons, 2004.
[7] Daniel Russo et al. 'A tutorial on Thompson sampling'. In: *Found. Trends Mach. Learn.* 11.1 (2018), pp. 1–96.

Principle 3: Learning from Rewards

Learning Methods in Reinforcement Learning

- Beyond action selection, a key design question is:
  - Which **learning method** to use?
- RL is about learning an **action-policy** from rewards
- Two main approaches:
  1. **On-policy learning**
  2. **Off-policy learning**

On-policy Learning

- Agent selects an action using the current policy
- The **value of that chosen action** is used to update the policy
- Learning is tied directly to the behavior of the policy

Key Idea

Update policy values using the **action actually taken**.

Off-policy Learning

- Learning uses values of **another action**, not necessarily the chosen one
- Makes sense during exploration:
  - Behavior policy may select a *non-optimal* action
  - On-policy learning would back up its inferior value
  - Off-policy learning instead backs up the **best action's value**
- Advantage:
  - Avoids "polluting" the policy with bad exploratory actions

On-Policy SARSA

Idea

- **On-policy** algorithm: learns from the action actually taken.
- Uses the same policy for both:
  - **Action selection (behavior policy)**
  - **Target updates (learning policy)**
- Typical choice: ϵ-greedy exploration.

SARSA Update Rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \Big]$$

- Uses the next action a_{t+1} *chosen by the current policy*.
- Predictive: learns directly from behavior values.

SARSA Intuition

- Agent follows its policy π (possibly ϵ-greedy).
- Updates Q-values using *the same action it just took*.
- Policy gradually improves while respecting its own exploration.

Off-Policy Q-Learning

Idea

- **Off-policy** algorithm: learns as if it always followed a greedy policy.
- Behavior policy may explore, but updates are from the *best possible action*.

Q-Learning Update Rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \Big]$$

- Uses $\max_a Q(s_{t+1}, a)$ instead of $Q(s_{t+1}, a_{t+1})$.
- Learns from **greedy action**, not the exploratory one.

Q-Learning Intuition

- Behavior policy may try exploratory actions.
- But updates pretend the agent acted greedily.
- Leads to convergence to the **optimal policy**.

SARSA vs. Q-Learning

SARSA (On-policy)

- Learns values of *current behavior*.
- More stable (lower variance).
- May converge to sub-optimal policy if ϵ is fixed.

Q-Learning (Off-policy)

- Learns values of *greedy policy*.
- Converges to optimal $Q^\star$ (low bias).
- Can be unstable with function approximation (max operator).

On-policy vs. Off-policy (Summary)

On-policy	Off-policy
Updates from the action actually taken	Updates from the *best* action
Tied to behavior policy	Separate behavior + target policy
Exploration actions may lower value estimates	More efficient during exploration
SARSA[8]	**Q-learning**

[8] Name from the update tuple (s, a, r, s', a')

Convergence Behavior

- Proven convergence in tabular RL when policy is:
  - Greedy in the limit with infinite exploration (GLIE)
- Off-policy methods:
  - Learn from greedy rewards
  - $\Rightarrow$ Converge to optimal policy after enough samples
- On-policy methods:
  - With fixed ϵ, never fully converge (keep exploring)
  - With decaying $\epsilon \to 0$, do converge to greedy policy

Sparse vs Dense Rewards

- **Dense reward:** Every state has a reward.
  - Example: supermarket (cost per step $\rightarrow$ negative reward).
- **Sparse reward:** Rewards only at special states.
  - Example: chess (only win/draw/loss at terminal positions).

Challenges of Sparse Rewards

- Harder to find good policies.
- Reward landscape: flat with rare sharp peaks.
- Gradient often zero $\Rightarrow$ optimization difficult.

Reward Shaping

- Modify reward function $\rightarrow$ easier optimization.
- Encodes heuristic knowledge into MDP.
- Common in board games (heuristics in chess, checkers).
- Classic reference: Ng et al. (1999)[9].

[9] Andrew Y Ng, Daishi Harada, and Stuart Russell. 'Policy invariance under reward transformations: Theory and application to reward shaping'. In: *International Conference on Machine Learning*. Vol. 99. 1999, pp. 278–287.

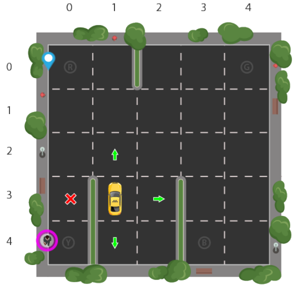
Hands-On: Q-Learning on Taxi

Hands-On: Q-learning on Taxi

- **Value Iteration:** works if transition model is known.
- **Q-learning:** model-free; learns by sampling.
- Stores rewards in a **Q-table**, approximating $Q(s, a)$.
- Once best actions are known for all states $\rightarrow$ optimal policy.

Taxi Environment Setup

- Grid world: $5 \times 5 = 25$ locations.
- State space size:
  25 (taxi positions) $\times$ 5 (passenger states) $\times$ 4 (destinations) $= 500$
- Actions: up, down, left, right, pick-up, drop-off.
- Rewards (Gym Taxi):
  - $+20$: successful drop-off.
  - -1: each time step.
  - -10: illegal drop-off.



Q-learning Intuition

- Goal: learn a policy $\pi(s)$ maximizing cumulative reward.
- Q-values = expected rewards for (s, a).
- Stored in array $Q(s, a)$, updated with experience.
- Use ϵ-greedy policy:
  - Best action most of the time.
  - Random action occasionally (exploration).

Q-learning Update Rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \Big]$$

- α: learning rate ($0 < \alpha \leq 1$).
- γ: discount factor ($0 \leq \gamma \leq 1$).
- Bootstrapping: update current Q using next state's Q.
- Q-table initialized randomly; values converge over time.

Q-learning Implementation

```
# Q learning for OpenAI Gym Taxi environment
import gymnasium as gym
import numpy as np
import random
#Environment Setup
env = gym.make("Taxi-v2")
env.reset()
env.render()
# Q[state,action] table implementation
Q = np.zeros([env.observation_space.n, env.action_space.n])
gamma = 0.7   # discount factor
alpha = 0.2   # learning rate
epsilon = 0.1 # epsilon greedy
for episode in range(1000):
    done = False
    total_reward = 0
    state = env.reset()
    while not done:
```

Q-learning Implementation

```
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample() # Explore state
                space
        else:
            action = np.argmax(Q[state]) # Exploit learned
                values
        next_state, reward, done, info = env.step(action) #
            invoke Gym
        next_max = np.max(Q[next_state])
        old_value = Q[state,action]

        new_value = old_value + alpha * (reward + gamma *
            next_max - old_value)

        Q[state,action] = new_value
        total_reward += reward
        state = next_state
    if episode % 100 == 0:
```

Q-learning Implementation

```
        print("Episode {} Total Reward: {}".format(episode,
            total_reward))
```

Algorithm Summary

1. Initialize Q-table randomly.
2. Choose initial state s.
3. Select action a from s:
   - Greedy or ϵ-random.
4. Execute a, observe r, s', update Q.
5. Repeat until terminal state.
6. Continue until Q-table converges.

Evaluating the Learned Policy

```
total_epochs, total_penalties = 0, 0
ep = 100
for _ in range(ep):
    state = env.reset()
    epochs, penalties, reward = 0, 0, 0
    done = False
    while not done:
        action = np.argmax(Q[state])
        state, reward, done, info = env.step(action)
        if reward == -10:
            penalties += 1
        epochs += 1
    total_penalties += penalties
    total_epochs += epochs
print(f"Results after {ep} episodes:")
print(f"Average timesteps per episode: {total_epochs / ep}")
print(f"Average penalties per episode: {total_penalties / ep}")
```

Tuning Hyperparameters

- Exploration ϵ: balance between exploration/exploitation.
- Discount γ: close to 1 for long-term reward.
- Learning rate α: small values stabilize learning.
- **Warning:** high α can cause divergence.

Tip: Start with $\gamma \approx 0.9$, $\alpha \approx 0.1$, $\epsilon \approx 0.1$.

Takeaways

- Q-learning is model-free and effective in discrete problems.
- Builds Q-table of expected rewards $\rightarrow$ optimal policy.
- Taxi world: small, fast, builds intuition.
- Key to mastery: experiment with hyperparameters!

Summary

- Value functions can be learned **without a transition model**, by sampling the environment.
- **Model-free methods:**
  - Use irreversible actions.
  - Sample states and rewards using exploration/exploitation trade-off.
  - Apply backup rules with bootstrapping.
- On-policy (SARSA): follows the chosen behavior policy, including explorative actions.
- Off-policy (Q-learning): always follows the value of the best action.
- Both use tabular representations of the value function.

Next: Function approximation with deep neural networks for high-dimensional state spaces.