強化學習入門學習計畫

學習目標

- **理解強化學習的基本概念**：熟悉環境、狀態、行動、獎勵等核心要素，以及強化學習如何透過代理（Agent）與環境互動來學習策略 (第3章 有限马尔可夫决策过程 — 强化学习导论 0.0.1 文档)。
- **實作核心演算法**：能夠利用 Python 手寫實現強化學習中的關鍵演算法（如 Q-learning、Deep Q Network、Policy Gradient），加深對理論的理解。
- **強化學習實戰經驗**：透過 OpenAI Gym 等工具進行實驗，累積基礎實戰經驗，為進一步探索更進階的強化學習技術打下基礎。

階段 1：強化學習基本概念與環境構成

重點內容：首先了解強化學習問題的構成，包括環境（Environment）、代理（Agent）、狀態（State）、行動（Action）和獎勵（Reward）。強化學習是透過代理在不同狀態下選擇行動影響環境，從環境獲得獎勵作為回饋，進而更新策略以最大化累積獎勵 (第3章 有限马尔可夫决策过程 — 强化学习导论 0.0.1 文档)。環境包含代理以外的所有事物，代理在每個時間步根據當前**狀態**選擇一個**行動**，環境對行動做出反應並給予下一個狀態及**獎勵**作為回饋 (第3章 有限马尔可夫决策过程 — 强化学习导论 0.0.1 文档)。這樣的互動形成了**狀態-行動-獎勵**的序列，代理的目標是學習一個選擇行動的策略以最大化長期獎勵總和。

- **環境（Environment）**：代理所處的外部系統。環境在代理採取行動後產生新的狀態和獎勵訊號，是代理學習的來源 (第3章 有限马尔可夫决策过程 — 强化学习导论 0.0.1 文档)。例如，在遊戲中環境即為遊戲世界。
- **狀態（State）**：環境在某時刻的情況描述。用於讓代理了解目前所處情境，例如遊戲中的畫面或物體位置。 (强化学习（一）动态规划、蒙特卡洛方法和时序差分 | Hexo)
- **行動（Action）**：代理可以對環境採取的操作。每個狀態下代理可選擇的行動集合可能不同，例如在遊戲中可按下的按鍵。 (强化学习（一）动态规划、蒙特卡洛方法和时序差分 | Hexo)

- **獎勵（Reward）**：環境給予代理的即時回饋信號，通常以數值表示。獎勵衡量當前行動的好壞，代理的目標是最大化累積獎勵總和 (第3章 有限马尔可夫决策过程 — 强化学习导论 0.0.1 文档)。例如遊戲中的得分變化可以作為獎勵。

推薦實作練習：安裝 OpenAI Gym 並嘗試建立一個簡單的強化學習環境，體驗代理與環境互動的過程。例如，可以使用經典的 CartPole 遊戲環境，讓代理隨機採取行動，觀察環境狀態變化和得到的獎勵：

```
import gym

# 建立 CartPole 遊戲環境
env = gym.make('CartPole-v1')
state = env.reset()
print("初始狀態:", state)

# 隨機行動測試代理與環境互動
for t in range(10):
    action = env.action_space.sample()              # 隨機選擇一個行動
    next_state, reward, done, info = env.step(action)  # 執行行動
    print(f"步驟{t}: 執行行動{action}, 獎勵={reward}, 下個狀態=
{next_state}")
    if done:
        print("遊戲結束, 重置環境")
        state = env.reset()
```

透過上述隨機試玩，理解環境提供的狀態資訊、行動空間以及獎勵反饋機制，這將為後續編寫演算法打下基礎。

階段 2：馬可夫決策過程（MDP）

重點內容：強化學習通常建模為**馬可夫決策過程（Markov Decision Process, MDP）**。MDP 是強化學習的理論基礎，它以數學形式定義了環境和代理的交互。MDP 包含以下要素：

- **狀態空間 $\mathcal{S}$**：所有可能的環境狀態集合。
- **行動空間 $\mathcal{A}$**：代理可採取的行動集合。
- **狀態轉移概率 $P(s'|s,a)$**：在狀態 s 下執行行動 a 後轉移到狀態 s' 的概率 (第3章 有限马尔可夫决策过程 — 强化学习导论 0.0.1 文档)。**馬可夫性**假設成立，即下一狀態只取決於當前狀態和行動，而與過去歷史無關。

- **獎勵函數 $R(s,a)$**：在狀態 s 執行行動 a 所得到的預期獎勵。
- **折扣因子 γ**：介於0和1之間，決定未來獎勵的重要程度。γ 越接近1，代理越重視長期回報；反之更關注短期回報。

透過 MDP，我們能夠精確定義強化學習問題： (第3章 有限马尔可夫决策过程 — 强化学习导论 0.0.1 文档)。 这些依赖于状态的量对于准确地为个人行动选择的长期结果分配信用至关重要。)) (第3章 有限马尔可夫决策过程 — 强化学习导论 0.0.1 文档)： 當代理在狀態 s 採取行動 a 後，環境根據轉移動態轉到新狀態 s' 並給出即時獎勵 r。代理行動的目的是最大化**累積回報（Return）**，通常定義為各時間步獎勵的折扣和：$G_t = R*{t+1} + \gamma R*{t+2} + \gamma^2 R_{t+3} + \dots$。

理解 MDP 有助於掌握後續演算法的前提。例如，若環境動態轉移概率和獎勵函數已知，則可使用**動態規劃**等方法求解；否則需採用**模型無關（model-free）**的方法來進行學習 (强化学习（一）动态规划、蒙特卡洛方法和时序差分 | Hexo。

推薦實作練習：嘗試將一個真實問題抽象為 MDP。例如，以棋盤上的移動為例，自行設計一個簡單的網格世界：定義格子為狀態、代理可移動的方向為動作、移動的獎勵規則，並寫出該 MDP 的狀態轉移和獎勵規則。這有助於鞏固對 MDP 元素的理解。

階段 3：策略與價值函數

重點內容：代理在強化學習中透過**策略（Policy）*來決定行動選擇。策略通常記為 $\pi(a|s)$，表示代理在狀態 s 下以概率 $\pi(a|s)$ 選擇行動 a (强化学习（一）动态规划、蒙特卡洛方法和时序差分 | Hexo)。策略可以是*確定性*的（每個狀態對應唯一行動）或*隨機*的（給出動作的機率分佈）。強化學習的目標是找到一個*最優策略 π^***，使得在所有狀態下的長期累積獎勵最大。

為評價策略的好壞，我們定義**價值函數（Value Function）**：

- **狀態價值函數 $V_{\pi}(s)$**：在策略 π 下，從狀態 s 開始所能獲得的**預期累積回報** (强化学习（一）动态规划、蒙特卡洛方法和时序差分 | Hexo)。直觀地說，$V_{\pi}(s)$衡量在狀態 s 下後續表現的好壞。
- **行動價值函數 $Q_{\pi}(s,a)$**：在策略 π 下，於狀態 s 執行行動 a 後所能獲得的預期累積回報 (强化学习（一）动态规划、蒙特卡洛方法和时序差分 | Hexo)。$Q_{\pi}(s,a)$評估在狀態 s 採取特定行動的好壞。

價值函數滿足遞迴的**貝爾曼方程（Bellman Equation）**關係：
$V*{\pi}(s) = \sum*{a \in \mathcal{A}} \pi(a|s) \sum*{s',r} P(s',r \mid s,a)\, [\,r + \gamma V*{\pi}(s')\,] ,$
該方程說明當前狀態的價值等於對可能行動的加權（由策略決定）後續價值和，即當前獎

勵加上折扣後未來價值的期望。

透過價值函數，我們可以衡量並改進策略——這就是強化學習許多演算法的基礎。後續演算法將重複進行**策略評估**（計算 $V{\pi}$ 或 $Q{\pi}$）和**策略改善**（根據價值選擇更優的行動）。

推薦實作練習：給定一個小型的 MDP（例如只有幾個狀態的轉移系統），假設一個隨機策略，手算或編寫簡單腳本計算每個狀態的 $V_{\pi}(s)$。這將加深對價值函數含義的理解。例如，可以模擬一個簡單遊戲，用蒙地卡羅方法反覆採樣多次來估計某策略的狀態價值。

階段 4：動態規劃方法

重點內容：在強化學習中，如果**環境模型（狀態轉移概率和獎勵函數）已知**且狀態空間不大，我們可以使用**動態規劃（Dynamic Programming, DP）**求解最優策略 (强化学习（一）动态规划、蒙特卡洛方法和时序差分 | Hexo。動態規劃利用貝爾曼方程進行確定性的策略評估與改進，包括兩個典型算法：

- **策略疊代（Policy Iteration）**：首先假設一個初始策略，進行**策略評價**計算該策略的價值函數，接著根據價值函數對每個狀態採用贏面最大的行動（即貪心選擇）來**改進策略**。如此反覆評估-改善，直到策略收斂 (强化学习（一）动态规划、蒙特卡洛方法和时序差分 | Hexo)。最終得到的即是最優策略和對應的價值函數。
- **價值疊代（Value Iteration）**：這是另一種等價的方法，每一步直接通過貝爾曼最優方程更新狀態價值：$V*{new}(s) = \max*{a \in \mathcal{A}} \sum_{s',r} P(s',r|s,a)[\,r + \gamma V(s')\,] .$ 重複這個更新直至 V 收斂，即可得到最優狀態價值，從而衍生出最優策略（對每個狀態選擇使右端取最大值的動作）。

動態規劃方法計算精確、收斂快 (Basic Concepts — DI-engine 0.1.0 documentation) (强化学习（一）动态规划、蒙特卡洛方法和时序差分 | Hexo)。但其前提是**已知環境模型**且狀態空間規模允許完全計算。現實中許多問題不符合這些條件，因此需要下面介紹的其他方法（如蒙特卡羅、時序差分）來應對。

推薦實作練習：在紙上或者使用程式，實現一個簡單網格世界的策略疊代演算法。例如，考慮一個 3×3 的網格迷宮，設定終點有正獎勵、陷阱有負獎勵，其餘步移為零獎勵。明確狀態轉移規則後，編寫程式反覆計算網格的價值函數並改進策略，驗證最終策略是否符合預期（如朝著高獎勵區域移動）。

階段 5：蒙特卡羅方法（Monte Carlo Methods）

重點內容：蒙特卡羅方法是一類通過**採樣試驗**來解決強化學習的演算法，適用於**模型未知**的情況 (强化学习（一）动态规划、蒙特卡洛方法和时序差分 | Hexo。與動態規劃不同，蒙特卡羅方法不需要預先知道狀態轉移概率，而是讓代理在環境中**實際體驗**（例如玩多次遊戲），從經驗中估計價值。 (第5章 蒙特卡洛方法 — 强化学习导论 0.0.1 文档)

蒙特卡羅方法的核心思想是：反覆讓代理從起始狀態按照當前策略遊玩直至終局（形成一個**episode**），然後利用該過程中得到的累積獎勵來估計經歷過的狀態或狀態-行動的價值 (第5章 蒙特卡洛方法 — 强化学习导论 0.0.1 文档)。重複大量回合後，取平均回報作為價值的近似。例如，**首次訪問蒙特卡羅**策略評價算法會在每次狀態第一次出現時記錄從該狀態開始的實得回報，經多次採樣取平均來估計 $V_{\pi}(s)$。

蒙特卡羅方法適用於**情節型任務**（有明確終止狀態的任務），因為需要等待一個情節結束才能根據最終回報更新估計 (第5章 蒙特卡洛方法 — 强化学习导论 0.0.1 文档)。它的優點是概念直觀、在模型未知時依然有效 (第5章 蒙特卡洛方法 — 强化学习导论 0.0.1 文档)；缺點是在每回合結束前無法利用當前資訊（需等待終局），且對長回合或無終止任務不直接適用。

> *術語：有時「回報」（Return）也稱為「累積獎勵」，與「獎勵」（Reward）有所區別。回報是整個後續得到的獎勵總和（通常折扣後），而獎勵指單一步的即時回饋。*

推薦實作練習：使用 OpenAI Gym 的**Blackjack**（二十一點）環境（`gym.make('Blackjack-v1')`）作為蒙特卡羅方法的練習。這個環境天然是情節型任務：每輪遊戲自然結束。讓代理使用隨機策略進行多次玩牌，記錄每個狀態（例如玩家手牌總點數、莊家明牌等）的最終輸贏結果，統計平均收益。這樣可以透過試驗體會蒙特卡羅估計價值的過程，為引入學習策略（如蒙特卡羅控制算法，用 ϵ-貪心策略逐步改進）做準備。

階段 6：時序差分學習（Temporal-Difference, TD）與 Q-learning

重點內容：時序差分（Temporal-Difference, TD）學習融合了動態規劃和蒙特卡羅兩者的思想，是強化學習領域的核心創新之一 (第6章 时序差分学习 — 强化学习导论 0.0.1 文档)。像蒙特卡羅一樣，TD 不需要環境模型，可直接從實際經驗中學習；又如動態規劃一般，TD 通過**自舉（Bootstrapping）***利用當前估計來更新價值，而無需等到情節結束才更新 (第6章 时序差分学习 — 强化学习导论 0.0.1 文档)。TD 方法每一步根據**部分**後續獎勵**進行更新，典型代表就是 **TD(0)** 演算法，其更新規則為：

$V(S_t) \leftarrow V(S_t) + \alpha \big[\,R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\,\big] ,$

其中 α 為學習率。這表示利用當前估計的下一狀態價值來改進當前狀態的價值估計。

在策略已給定的前提下，TD 方法可以用於**策略評價**（如 TD(0) 更新上述 V 值）。更重要的是，TD 思想可用於**控制**問題，尋找最優策略。例如，**Q-learning** 就是最具代表性的 TD 控制演算法之一。

Q-learning 演算法：(第6章 时序差分学习 — 强化学习导论 0.0.1 文档)
這是一種*離線策略（off-policy）*TD控制方法。它通過直接學習最優動作價值函數 $Q^*(s,a)$ 來逼近最優策略。其更新公式為：
$Q(s,a) \leftarrow Q(s,a) + \alpha \Big[\, R_{t+1} + \gamma \max_{a'} Q(s',a') - Q(s,a)\,\Big] ,$
其中 s' 為執行動作 a 後到達的新狀態，$\max_{a'} Q(s',a')$代表在新狀態下估計的最佳後續價值。 (第6章 时序差分学习 — 强化学习导论 0.0.1 文档)這個更新規則意味著：用當前獎勵加上下一狀態的最高預估價值作為目標，不斷調整目前的 $Q(s,a)$ 值。最終，在充分探索各狀態-行動並反覆更新下，Q 值將收斂到最優動作價值函數 (第6章 时序差分学习 — 强化学习导论 0.0.1 文档)。

在實作 Q-learning 時，常採用 ϵ-貪心策略來平衡**探索**（Exploration）和**利用**（Exploitation）：以小概率ϵ隨機選擇動作探索未知狀態，其餘情況選擇目前 Q 值最大的動作。這可確保算法持續探索新的狀態-行動，同時逐步傾向已有經驗。

Python 實作範例：以下程式碼實現了一個簡化的 Q-learning，用於 OpenAI Gym 提供的 FrozenLake 環境（4x4 冰湖，目標是找到不掉入冰窟的路徑）。我們將建立一個 Q 表格，反覆與環境互動並更新表格值。

```
import gym
import numpy as np

env = gym.make('FrozenLake-v1', is_slippery=False)  # 建立冰湖環境（關閉滑動隨機性以簡化問題）
n_states = env.observation_space.n   # 狀態數量
n_actions = env.action_space.n       # 行動數量

# 初始化Q表為零
Q = np.zeros((n_states, n_actions))
alpha = 0.1    # 學習率
gamma = 0.99   # 折扣因子
epsilon = 0.1  # 探索機率

for episode in range(500):  # 訓練500回合
    state = env.reset()
```

```
    done = False
    while not done:
        # 選擇行動：ε-貪心策略
        if np.random.rand() < epsilon:
            action = env.action_space.sample()          # 探索：隨機選
        else:
            action = np.argmax(Q[state])                # 利用：選擇目
前最優行動
        next_state, reward, done, info = env.step(action)  # 執行行
動
        # Q-learning 更新
        best_next = np.max(Q[next_state])
        Q[state, action] += alpha * (reward + gamma * best_next -
Q[state, action])
        state = next_state

# 訓練結束後，輸出學到的最優策略（每個狀態最優行動）
optimal_actions = np.argmax(Q, axis=1)
print("各狀態學到的最優行動索引:", optimal_actions)
```

在上述代碼中，代理會在 FrozenLake 上反覆嘗試。透過 Q-learning 更新，Q 表格逐步逼近最優值函數。最後從 Q 表中提取的 `optimal_actions` 即為各狀態下代理認為最好的行動策略。

練習建議：執行上述程式，觀察代理在 FrozenLake 是否學會避開陷阱、到達目標。嘗試調整學習率 α 和探索率 ϵ，觀察收斂速度和結果的變化。另外，可以將 `is_slippery=True`（冰面滑動）再訓練，體會隨機性增加對學習的影響。這些經驗有助於理解強化學習演算法在各種環境下的表現差異。

階段 7：深度 Q 網路（Deep Q-Network, DQN）實戰

重點內容：傳統 Q-learning 使用表格存儲 Q 值，當狀態空間很大甚至連續時（例如遊戲畫面的像素狀態），表格方法將無法應對。**深度 Q 網路（DQN）*將 *Q-learning* 與深度學習結合，使用*神經網路**逼近動作價值函數，以處理高維或連續狀態空間 (一图看懂DQN(Deep Q-Network)深度强化学习算法_深度强化学习算法的结构示意图-CSDN博客)。DQN 是深度強化學習的開創性算法之一，由 DeepMind 團隊提出並成功應用於雅達利遊戲 (一图看懂DQN(Deep Q-Network)深度强化学习算法_深度强化学习算法的结构示意图-CSDN博客)。

DQN 的主要特點包括 (一图看懂DQN(Deep Q-Network)深度强化学习算法_深度强化学习算法的结构示意图-CSDN博客) (DQN 算法 - 动手学强化学习)：

- 使用神經網路 $Q(s,a;\theta)$ 來近似 Q 值。輸入為狀態，輸出對應各個動作的預估價值。通過調整參數 θ，讓網路預測的 Q 值接近真實的回報。
- 引入**經驗回放（Experience Replay）**機制：將代理與環境交互得到的經驗 (s,a,r,s') 存儲起來，在訓練時從記憶庫中隨機抽樣小批資料進行學習。這打破了數據之間的相關性，提高訓練穩定性。
- 引入**目標網路（Target Network）**：建立一個拷貝的 Q 網路作為目標，每隔固定步數同步一次參數。訓練時計算目標值 $y = r + \gamma \max_{a'} Q_{\text{target}}(s',a')$，用於更新主要網路。目標網路的使用減少了訓練過程中目標移動的振盪，提高收斂性。

透過以上技術，DQN 有效地解決了傳統 Q-learning 在大型狀態空間下的不穩定和收斂難題，使強化學習能應用到更複雜的任務上 (一图看懂DQN(Deep Q-Network)深度强化学习算法_深度强化学习算法的结构示意图-CSDN博客)。

Python 實作範例：以下將實作一個簡化的 DQN，用來解決經典的 CartPole 平衡桿問題。CartPole 的狀態由4個連續變量組成（小車位置/速度、杆子角度/角速度），行動為向左或向右施力兩種。我們使用一個簡單的前饋神經網絡作為 Q 網路，並在線更新參數（未使用經驗回放和目標網路以簡化實作）。

```
import gym
import numpy as np
from tensorflow import keras

# 建立 CartPole 環境和 Q 網絡
env = gym.make('CartPole-v1')
state_dim = env.observation_space.shape[0]   # 狀態維度 = 4
num_actions = env.action_space.n             # 行動數量 = 2

# 定義一個簡單的神經網路模型作為Q函數逼近
model = keras.Sequential([
    keras.layers.Dense(24, activation='relu', input_shape=
(state_dim,)),
    keras.layers.Dense(24, activation='relu'),
    keras.layers.Dense(num_actions, activation='linear')
])
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
loss='mse')
```

```
# DQN 超參數
gamma = 0.95           # 折扣因子
epsilon = 1.0          # 初始探索機率
epsilon_decay = 0.995 # 每回合後探索機率的衰減
epsilon_min = 0.01     # 最低探索機率
episodes = 300         # 訓練迴合數

for episode in range(episodes):
    state = env.reset()
    state = np.reshape(state, [1, state_dim])   # 調整形狀以符合網路輸
入
    done = False
    while not done:
        # ϵ-貪心選擇行動
        if np.random.rand() < epsilon:
            action = env.action_space.sample()
        else:
            q_values = model.predict(state, verbose=0)
            action = np.argmax(q_values[0])
        # 執行行動
        next_state, reward, done, info = env.step(action)
        next_state = np.reshape(next_state, [1, state_dim])
        # 計算目標Q值
        target = reward
        if not done:
            # 非終止狀態，加入未來預期價值
            target += gamma * np.max(model.predict(next_state,
verbose=0)[0])
        # 更新當前動作的Q值朝向目標值
        target_q = model.predict(state, verbose=0)
        target_q[0][action] = target
        model.fit(state, target_q, epochs=1, verbose=0)
        # 移動到下一狀態
        state = next_state
    # 減少探索率
    if epsilon > epsilon_min:
        epsilon *= epsilon_decay

print("訓練完成，探索率=", epsilon)
```

上述程式在每個迴合（episode）中讓代理從頭開始嘗試保持平衡桿，使用 ϵ-貪心策略選擇行動並訓練網路。對每一步，將神經網路對當前狀態的預測 Q 值向目標值進行調整（目標值為當前獎勵加上預估的下一狀態最大 Q）。隨著迴合進行，ϵ 逐漸降低，代理會從最初的隨機探索慢慢轉向利用學到的策略。經過數百回合訓練後，CartPole 問題通常可以收斂到平衡杆子的不錯策略。

> *說明：為了簡潔，上述代碼省略了經驗回放和目標網絡等提升穩定性的技術，因此在一些情況下可能不如完整 DQN 收斂穩定。但即使如此，通常也能在 CartPole 這樣的簡單任務上取得成功。讀者可將此代碼作為起點，進一步完善為完整的 DQN 算法。*

練習建議：觀察每回合代理存活的時間步數是否逐漸增加，以評估學習效果。嘗試修改神經網路的結構（例如隱藏層大小）或調整超參數（如學習率、ϵ 衰減速度）來改善訓練表現。如果有能力，也可以實現經驗回放機制：使用一個記憶體儲存轉移樣本，每步從中隨機抽取小批量樣本來訓練，這將提升學習的穩定性和效率。

階段 8：策略梯度方法（Policy Gradient）與應用

重點內容：前述方法大多屬於**價值函數為基礎（Value-Based）**的強化學習——先估計價值函數，再從中導出策略 (第13章 策略梯度方法 — 强化学习导论 0.0.1 文档， 如 (\hat{v}(s%2C \mathbf{w})) 所示。))。**策略梯度（Policy Gradient）方法**提供了直接以策略為基礎進行學習的途徑 (强化学习(十三) 策略梯度(Policy Gradient) - 刘建平Pinard - 博客园)。這類方法直接為策略建立參數化表示（例如以參數向量 θ 表示的策略 $\pi(a|s;\theta)$），通過梯度上升優化策略參數以最大化期望回報 (第13章 策略梯度方法 — 强化学习导论 0.0.1 文档。換言之，策略梯度方法不需要明確學習價值函數即可選擇行動（雖然可以同時學一個價值函數作輔助），而是通過提升策略本身的表現指標來求得最優策略 (第13章 策略梯度方法 — 强化学习导论 0.0.1 文档， 如 (\hat{v}(s%2C \mathbf{w})) 所示。))。

最基本的策略梯度演算法是**REINFORCE**（又稱蒙特卡羅策略梯度）(强化学习(十三) 策略梯度(Policy Gradient) - 刘建平Pinard - 博客园)。其演算法步驟概要：

1. 從當前策略出發進行多次遊戲（episode）模擬，記錄每次在狀態 s_t 選擇行動 a_t 並獲得的獎勵序列。
2. 對於每一步t，計算從該步開始的**折扣累積回報** G_t。
3. 調整策略參數 θ，方向為提高含有行動 a_t 的策略概率的方向，其調整幅度與G_t成正比。這可透過梯度公式實現：$\theta \leftarrow \theta + \alpha, G_t, \nabla_{\theta} \log \pi(a_t|s_t;\theta)$。

直觀理解：若某行動最終帶來高回報，則增加其在當前狀態下被選擇的概率；反之則降低。經過足夠次的樣本平均，這個更新期望等價於提升整體回報的梯度方向 (第13章 策略梯度方法 — 强化学习导论 0.0.1 文档。

需要注意，純蒙特卡羅策略梯度方法（如 REINFORCE）**高方差**且每次更新需要一整個情節數據 (强化学习(十三) 策略梯度(Policy Gradient) - 刘建平Pinard - 博客园)。因此實務中常會引入**基線（Baseline）**減少方差——最常見的是減去狀態價值函數作為基線（對應 Actor-Critic 演算法思想）。Actor-Critic 方法同時學習一個價值函數（Critic）來輔助策略更新，能加速收斂。

Python 實作範例：以下以 CartPole 問題演示一個簡單的 REINFORCE 策略梯度實作。我們使用 PyTorch 建立策略網路，採用蒙特卡羅方式估計每回合的回報，並在回合結束後更新策略參數。

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

import gym

# 定義策略網路（輸入為狀態，輸出為各動作的機率分佈）
class PolicyNet(nn.Module):
    def __init__(self):
        super(PolicyNet, self).__init__()
        self.fc1 = nn.Linear(4, 16)     # CartPole 狀態維度4
        self.fc2 = nn.Linear(16, 2)     # 兩個動作（向左或向右）
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.softmax(self.fc2(x), dim=-1)  # 輸出對應兩個動作的概率
        return x

env = gym.make('CartPole-v1')
policy_net = PolicyNet()
optimizer = optim.Adam(policy_net.parameters(), lr=0.01)
gamma = 0.99

for episode in range(300):
    state = env.reset()
    log_probs = []  # 儲存每步對數機率
    rewards = []    # 儲存每步即時獎勵
```

```
    # 產生一回合資料
    done = False
    while not done:
        state_tensor = torch.FloatTensor(state)
        action_probs = policy_net(state_tensor)
        # 根據策略網路輸出的機率分布採樣行動
        action_dist = torch.distributions.Categorical(action_probs)
        action = action_dist.sample()
        log_probs.append(action_dist.log_prob(action))  # 儲存
log(prob)以便訓練時使用
        next_state, reward, done, info = env.step(action.item())
        rewards.append(reward)
        state = next_state
    # 計算每步的折扣累積回報（從後往前計算G_t）
    returns = []
    G = 0
    for r in reversed(rewards):
        G = r + gamma * G
        returns.insert(0, G)
    returns = torch.tensor(returns)
    # 將回報標準化，減少方差（optional，但常見處理）
    returns = (returns - returns.mean()) / (returns.std() + 1e-9)
    # 計算策略梯度的損失函數
    policy_loss = []
    for log_prob, R in zip(log_probs, returns):
        policy_loss.append(-log_prob * R)
    policy_loss = torch.stack(policy_loss).sum()
    # 反向傳播更新策略網路參數
    optimizer.zero_grad()
    policy_loss.backward()
    optimizer.step()
```

在此程式中，`PolicyNet` 輸出一個對兩個動作的機率分佈，透過採樣決定行動。每個episode結束後，我們計算該回合每一步的累積回報 `returns`，並根據 REINFORCE 的策略梯度公式來計算損失（其實就是負的累積回報加權的對數概率）。最後執行梯度下降（對應梯度上升於獎勵）來更新策略。經過多次迭代後，策略網路會提高那些帶來高回報的行動概率，使得杆子平衡的時間越來越長。

練習建議：嘗試調整學習率或網路結構對訓練過程的影響，例如將隱藏層從16個神經元增減，看收斂速度和最終表現是否有變化。也可以將 CartPole 的獎勵修改為每一步 -1（倒掉為0）來轉變為一個**最小化**平衡時間的問題，觀察策略梯度方法是否仍能正常工作（此時需要對獎勵取負來進行梯度上升）。另外，建議了解並實作簡單的 Actor-Critic：在上述程式基礎上增加一個價值網路作為 Baseline，看看是否能更快收斂。

以上八個階段涵蓋了強化學習從基礎概念、核心理論到重點演算法實作的入門路線圖。按照順序學習並完成對應的練習，學習者將能夠：

1. 清楚理解強化學習問題的構成與各種經典解法的原理，
2. 動手實作並調試強化學習演算法的細節，
3. 獲得將理論應用於實際環境的經驗。

在此基礎上，學習者可以進一步探索更進階的主題，例如深度強化學習中的演算法優化（Double DQN、Dueling DQN、PPO、DDPG 等）、策略梯度的改進方法以及多智能體強化學習等，不斷拓展強化學習的知識版圖。祝學習順利！ (一图看懂DQN(Deep Q-Network)深度强化学习算法_深度强化学习算法的结构示意图-CSDN博客)