深度強化學習在連鎖反應遊戲中的應用：基於 AlphaZero 架構的實作與分析

摘要 (Abstract)

本研究旨在探討利用深度強化學習（Deep Reinforcement Learning, DRL）解決「連鎖反應」（Chain Reaction）遊戲策略問題的可行性。連鎖反應遊戲具有高度的不穩定性與動態特徵，單一步驟可能引發全盤局勢的劇烈變化，這使得傳統的啟發式搜尋演算法難以精確評估盤面價值。本專案實作了一個基於 AlphaZero 的人工智慧代理（Agent），核心架構包含卷**積殘差網路（Convolutional ResNet）與蒙地卡羅樹搜尋（MCTS）**。本文將詳細剖析環境建模、神經網路設計、MCTS 演算法細節，並比較基於 Gradio 的網頁端與基於 Pygame 的本機端兩種實作方式。

---

1. 引言 (Introduction)

1.1 研究動機

受到西洋棋 AI Stockfish 與 Google DeepMind 所推出的電影《The Thinking Game》所啟發，我對於人工智慧在完全資訊賽局中的決策極限產生了濃厚興趣。AlphaZero 展示了透過自我對弈超越人類經驗的可能性，然而，相較於西洋棋的穩定結構，「連鎖反應」(Chain Reaction) 具有極高的動態不確定性。單一格子的臨界爆炸往往能引發連鎖效應，瞬間翻轉盤面優勢，這使得傳統基於固定估值函數的搜尋演算法難以精確應對。

1.2 研究目的

本研究旨在探討將卷積殘差網路 (ResNet) 結合蒙地卡羅樹搜尋 (MCTS) 應用於此類高非線性遊戲的可行性，期望透過深度強化學習，訓練出能掌握複雜爆炸規律並制定長遠策略的智能代理。

1.3 Chain Reaction 遊戲規則說明

連鎖反應（Chain Reaction）是一種在 $N \times N$(在此研究中設定為5) 網格上進行的零和策略遊戲。規則如下:

- 玩家在第一局可以在人一空白位置放置三個原子
- 第一局後，玩家必須在己方顏色的格子中任取一個放置一個原子
- 當一個格子的原子數達到臨界值(在此研究中設定為4)該格回發生「爆炸」現象:
    - 爆炸的格子原子數歸零
    - 該格子四周的格子點數皆加一(觸及邊界則忽略此規則)，若等於4，則連鎖爆炸
    - 爆炸四周的格子所有權轉為己方所有

遊戲規則雖然簡單，原子數似乎也看似是不變量(每局增加2個原子)，但是受到邊界限制以及所有權轉移，這種連鎖效應導致遊戲狀態空間（State Space）雖然離散，但狀態轉移（State Transition）極度非線性。

---

2. 環境建模與遊戲邏輯 (Environment Modeling)

在強化學習中，精確的環境模擬是訓練的基礎。本專案透過 `chain_reaction_env .py` 定義了遊戲的物理規則。

2.1 狀態空間表示 (State Representation)

為了讓神經網路能夠處理遊戲資訊，我們將盤面狀態張量化。在 `ChainReactionEnv` 類別中，狀態被定義為一個 $5 \times 5 \times 2$ 的 NumPy 陣列：

- **維度 $(5, 5, 2)$：** 對應 5×5 的棋盤。
- **通道 0 (Atom Count)：** 紀錄每個格子目前的原子數量（整數 0-3）。
- **通道 1 (Ownership)：** 紀錄格子的歸屬權。我們使用 `1` 代表紅色（先手），`-1` 代表藍色（後手/AI），`0` 代表中立空地。

這種編碼方式使得卷積神經網路（CNN）能夠同時提取「原子密度」與「勢力分佈」的空間特徵。

2.2 動態轉移與連鎖爆炸

遊戲最關鍵的邏輯在於 `get_next_state` 方法。當玩家執行動作後，環境會進行以下運算：

1. **放置原子：** 若為首手，根據規則給予額外原子（+3 Bonus）；否則1必須放置於與自己同色的位置，該位置原子數 +1。
2. **連鎖判定：** 程式進入 `BFS(廣度優先探索)`，持續偵測盤面上是否有格子達到臨界質量（Critical Mass = 4）。

   ```
   # 尋找所有達到爆炸條件的格子
   exploding_cells = np.where(new_state[:, :, 0] >= self.critical_mass)
   ```

3. **爆炸擴散：** 爆炸格子的原子歸零，並向上下左右四個方向擴散(觸及邊界及忽視)。最重要的是，擴散會觸發「所有權轉移」，將鄰居格子強制轉為爆炸方的顏色。

```
# Spread to neighbors
                directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
                for dr, dc in directions:
                    nr, nc = er + dr, ec + dc

                    # Check bounds
                    if 0 <= nr < self.grid_size and 0 <= nc < self.grid_size:
                        # Give atom
                        new_state[nr, nc, 0] += 1
                        # Claim neighbor's color (ownership transfer)
                        new_state[nr, nc, 1] = explosion_color
```

2.3 終局判定

`get_value_and_terminated` 函數負責判斷遊戲是否結束。由於連鎖反應的勝利條件是「消滅對手所有原子」，程式會計算盤面上紅藍雙方的格子總數。若某一方格子數歸零，則判定另一方獲勝（Reward = 1）。

---

3. 神經網路架構 (Neural Network Architecture)

本專案的神經網路模型定義於 `neural_structure.py`，採用了經典的 ResNet 結構，並分為策略頭（Policy Head）與價值頭（Value Head）。

3.1 殘差骨幹 (Residual Backbone)

為了捕捉棋盤上的空間特徵，網路首先通過一個卷積層將輸入的 2 個通道升維至 64 個通道，隨後經過數個 `ResNetBlock`。

```
class ResNetBlock(nn.Module):
    def __init__(self, channels):
        super(ResNetBlock, self).__init__()
        self.conv1 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(channels)
        self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(channels)

    def forward(self, x):
        residual = x
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += residual
        return F.relu(out)
```

殘差連接（`out += residual`）允許梯度直接流向淺層網路，有效解決了深層網路訓練退化的問題，確保 AI 能理解長距離的連鎖反應潛力。

3.2 雙頭輸出 (Dual-Head Output)

網路在骨幹之後分為兩個分支：

1. 策略頭 (Policy Head)：
    - 目的： 預測下一步在 25 個格子落子的機率分布。
    - 結構： 卷積層(`Conv2d`) $\rightarrow$ 批次正規化(`BatchNorm2d`) $\rightarrow$ 全連接層 $\rightarrow$ LogSoftmax。
    - 輸出： 形狀為 $(Batch, 25)$ 的向量，代表每個位置的落子優先度。

```
# --- Policy Head ---
self.policy_conv = nn.Conv2d(num_channels, 2, kernel_size=1)
self.policy_bn = nn.BatchNorm2d(2)
self.policy_fc = nn.Linear(2 * grid_size * grid_size, grid_size * grid_size)
```

2. 價值頭 (Value Head)：
    - 目的： 評估當前盤面的勝率。
    - 結構： 卷積層 $\rightarrow$ 全連接層 $\rightarrow$ Tanh。
    - 輸出： 範圍在 $[-1, 1]$ 的純量。1 代表當前玩家必勝，-1 代表必敗。

```
# --- Value Head ---
self.value_conv = nn.Conv2d(num_channels, 1, kernel_size=1)
self.value_bn = nn.BatchNorm2d(1)
self.value_fc1 = nn.Linear(grid_size * grid_size, 64)
self.value_fc2 = nn.Linear(64, 1)
```

4. 蒙地卡羅樹搜尋 (MCTS)

MCTS 是連接神經網路預測與實際決策的橋樑。在 `app.py` 中，雖然具體的 MCTS 類別被封裝引用，但在 `ai_move` 函數中我們可以觀察到其應用流程：

```
def ai_move(state, move_count):
    args = {'num_simulations': MCTS_SIMS, 'c_puct': 1.0}
    mcts = MCTS(GAME, MODEL, args)
    probs = mcts.search(state, -1, move_count)
    return np.argmax(probs)
```

以下詳細說明 MCTS 在本 AI 中的四個核心步驟（Selection, Expansion, Evaluation, Backup）：

4.1 選擇 (Selection)

演算法從根節點（當前盤面）出發，向下選擇子節點，直到到達一個葉節點。選擇的依據是 PUCT (Predictor + Upper Confidence Bound applied to Trees) 公式，該公式平衡了「利用」（選擇高勝率節點）與「探索」（選擇少被訪問的節點）。

$$a_t = \operatorname{argmax}_a (Q(s, a) + U(s, a))$$

其中 $Q(s, a)$ 是目前的平均勝率，$U(s, a)$ 是探索項，與神經網路輸出的先驗機率 $P(s, a)$ 成正比，並隨訪問次數 $N(s, a)$ 增加而衰減。參數 `c_puct` (設定為 1.0) 用於控制探索的積極程度。

4.2 擴展與評估 (Expansion & Evaluation)

當搜尋到達一個尚未展開的葉節點 S_L 時，我們呼叫神經網路 `MODEL` 進行評估：

$$ (\mathbf{p}, v) = f_{\theta}(S_L) $$

- 擴展： 利用策略頭輸出的機率向量 $\mathbf{p}$ 來初始化該節點下所有合法動作的先驗機率。
- 評估： 價值頭輸出的 v 直接作為該盤面的評分，不需要像傳統 MCTS 那樣進行隨機模擬（Random Rollout）直到遊戲結束。這大幅提升了搜尋效率。

4.3 反向傳播 (Backpropagation)將評估得到的價值 v 沿著搜尋路徑反向傳回根節點。對於路徑上的每個節點，更新其統計數據：

- 訪問次數：$N \leftarrow N + 1$
- 平均價值：$Q \leftarrow \frac{N_{old} \cdot Q_{old} + v}{N_{new}}$

4.4 決策 (Decision)

經過 `MCTS_SIMS` (50次) 的模擬後，AI 根據根節點各動作的訪問次數 N 來決定最終移動。在 `app.py` 中，直接選擇訪問次數最多（機率最高）的動作：`np.argmax(probs)`。

5. 系統實作與介面比較 (System Implementation)

本專案提供了兩種不同的互動介面，分別適用於網頁部署與本機遊玩。

5.1 網頁介面 (Gradio Implementation)

`app.py` 使用 Gradio 框架建構了一個 Web UI。

- 視覺化： 透過 CSS (`CUSTOM_CSS`) 將 HTML 按鈕排版成 5×5 網格，並根據原子歸屬動態切換 `.cell-red` 與 `.cell-blue` 樣式。
- 事件驅動： on_click 函數處理完整的遊戲回合邏輯：
    1. 驗證玩家（紅方）點擊的合法性。
    2. 執行玩家移動並檢查勝負。
    3. 若遊戲未結束，立即觸發 AI（藍方）進行運算。
    4. 更新前端頁面顯示。

 image

5.2 本機介面 (Pygame Implementation)

`color_war_game.py` 提供了一個基於 Pygame 的圖形介面，適合流暢的本機體驗。

- 渲染循環： 使用 `while running` 迴圈與 `clock.tick(FPS)` 保持 60 FPS 的畫面更新。
- 動畫效果： `draw_atoms` 函數根據原子數量（1-4顆）繪製不同排列方式的圓點，提供了比網頁版更直觀的視覺反饋。
- 雙人對戰/邏輯驗證： 此版本主要實作了 `ChainReactionGame` 類別，包含完整的 `add_atom` 與 `explode` 遞迴邏輯，適合作為 AI 訓練前的環境驗證工具。

 image

6. 結論 (Conclusion)

透過結合卷積殘差網路與蒙地卡羅樹搜尋，我們成功構建了一個具備長遠規劃能力的連鎖反應遊戲 AI。環境模擬 (`color_war_env`) 精確捕捉了遊戲的動態規則，神經網路 (`neural_structure`) 有效提取了盤面特徵，而 MCTS (`app.py`) 則確保了在複雜局勢下的決策穩定性。未來的改進方向可包括增加 MCTS 的模擬次數，或引入多執行緒並行搜尋以提升回應速度。