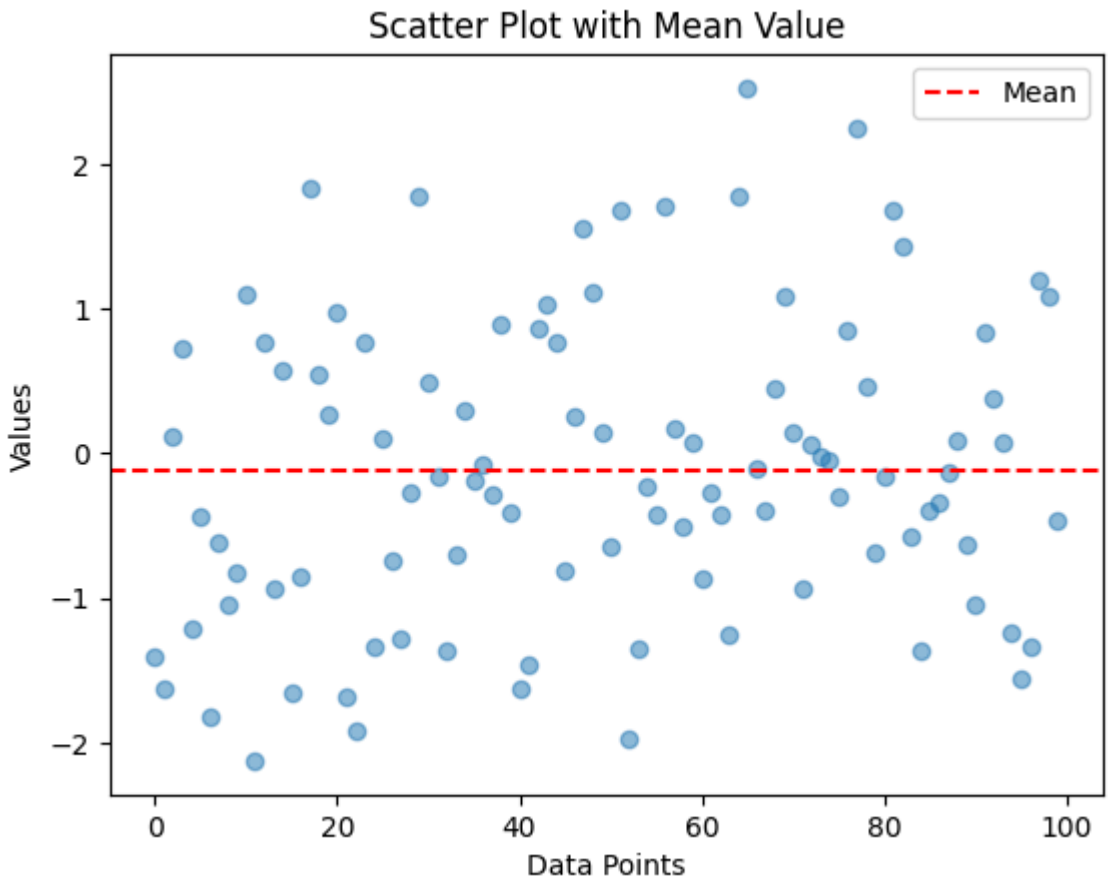
HW4-1

```
import matplotlib.pyplot as plt
import numpy as np

# Generate sample data
data = np.random.randn(100)

# Calculate the mean value
mean_value = np.mean(data)

# Plot the data with alpha level
plt.scatter(range(len(data)), data, alpha=0.5)

# Add a horizontal line for the mean value
plt.axhline(mean_value, color='red', linestyle='--', label='Mean')

# Set plot properties
plt.xlabel('Data Points')
plt.ylabel('Values')
plt.title('Scatter Plot with Mean Value')
plt.legend()

# Display the plot
plt.show()
```

- data是長度為100的一維陣列，標準常態分佈(平均值0，標準差1)的隨機數
- mean_value計算data的平均值，用於畫一條參考線

再來畫出每個資料點的散佈圖




```
from Gridworld import Gridworld
game = Gridworld(size=4, mode='static')
game.display()
```

```
array([['+', '-', ' ', ' '],
       [' ', 'W', ' ', 'P'],
       [' ', ' ', ' ', ' '],
       [' ', ' ', ' ', ' ']], dtype='<U2')
```

這邊import gridworld這個python檔，並利用裡面寫好的function建立一個gridworld遊戲

GridWorld 是一個網格棋盤（預設大小是 4x4），上面有幾個元素：

- `Player (P)`：玩家，可上下左右移動
- `Goal (+)`：目標格子，到達得分

- `Pit (-)`：陷阱格子，掉進去失敗
- `Wall (W)`：牆壁，不能通過

玩家每走一步就會有一個 reward，走到 goal 得 +10，掉進 pit 扣 -10，其他步驟扣 -1

並且利用makemove這個function對玩家進行移動，u=往上走，d=往下走，l=往左走，r=往右走
若該動作導致撞牆/越界/掉坑，根據
`validateMove` 來判斷與執行

`reward(self)`

- 給出 reward：
  - 掉進坑：`10`
  - 到目標：`+10`
  - 其他移動：`1`

```
game.board.render_np()
game.board.render_np().shape
```

- game.board.render_up()主要功能是回傳棋盤的numpy陣列表示
- 並且利用.shape印出陣列維度(4,4,4)代表4x4的陣列，總共有4個

```
L1 = 64 #輸入層的寬度
L2 = 150 #第一隱藏層的寬度
L3 = 100 #第二隱藏層的寬度
L4 = 4 #輸出層的寬度

model = torch.nn.Sequential(
    torch.nn.Linear(L1, L2), #第一隱藏層的shape
    torch.nn.ReLU(),
    torch.nn.Linear(L2, L3), #第二隱藏層的shape
    torch.nn.ReLU(),
    torch.nn.Linear(L3,L4) #輸出層的shape
)
loss_fn = torch.nn.MSELoss() #指定損失函數為MSE（均方誤差）
learning_rate = 1e-3  #設定學習率
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) #指定優化

gamma = 0.9 #折扣因子
epsilon = 1.0
```

建立模型，分別定義神經穩路每一層的寬度，損失函數，優化器，折扣因子及探索率

- 折扣因子代表未來獎勵的重要程度
- 探索率設定1，根據 epsilon-greedy 策略，agent 在每一步都會選擇隨機動作，而不是利用已有的知識來選擇預期回報最高的動作。
  這有助於在訓練初期充分探索環境，防止過早陷入次優策略

```
action_set = {
   0: 'u', #『0』代表『向上』
   1: 'd', #『1』代表『向下』
   2: 'l', #『2』代表『向左』
   3: 'r' #『3』代表『向右』
}
```

模型輸出層（`Linear(L3, L4)`）會輸出一個 **長度為 4 的數值向量**，每個值對應一個動作的預測 Q 值

這代表：

- `output[0]` → Q 值 for 動作 `'u'`
- `output[1]` → Q 值 for 動作 `'d'`
- `output[2]` → Q 值 for 動作 `'l'`
- `output[3]` → Q 值 for 動作 `'r'`

這些 Q 值越高，代表模型越傾向選這個動作。

```
state_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
```

訓練迴圈，epochs設定1000，每輪都會建立一個新的 `Gridworld` 遊戲，並初始化狀態
取得 4×4×4 狀態轉成長度 64 的向量，加上小雜訊避免過度擬合

```
state1 = torch.from_numpy(state_).float()
```

轉成 PyTorch tensor

while(status == 1)代表持續進行動作直到遊戲結束(掉入坑或到達目標)

```
while(status == 1):
    qval = model(state1) #執行Q網路，取得所有動作的預測Q值
    qval_ = qval.data.numpy() #將qval轉換成NumPy陣列
    if (random.random() < epsilon):
      action_ = np.random.randint(0,4) #隨機選擇一個動作（探索）
    else:
      action_ = np.argmax(qval_) #選擇Q值最大的動作（探索）
    action = action_set[action_] #將代表某動作的數字對應到makeMove()的英文字
    game.makeMove(action) #執行之前ε—貪婪策略所選出的動作
    state2_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10
    state2 = torch.from_numpy(state2_).float().to(device) #動作執行完畢，取得遊
    reward = game.reward()
    with torch.no_grad():
      newQ = model(state2.reshape(1,64))
    maxQ = torch.max(newQ) #將新狀態下所輸出的Q值向量中的最大值給記錄下來
    if reward == -1:
      Y = reward + (gamma * maxQ)  #計算訓練所用的目標Q值
    else: #若reward不等於-1，代表遊戲已經結束，也就沒有下一個狀態了，因此目標
      Y = reward
    Y = torch.Tensor([Y]).detach()
    X = qval.squeeze()[action_].to(device) #將演算法對執行的動作所預測的Q值存
    loss = loss_fn(X, Y) #計算目標Q值與預測Q值之間的誤差
    if i%100 == 0:
      print(i, loss.item())
      clear_output(wait=True)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    state1 = state2
    if abs(reward) == 10:
      status = 0 # 若 reward 的絕對值為10，代表遊戲已經分出勝負，所以設status為
  losses.append(loss.item())
```

```
  if epsilon > 0.1:
    epsilon -= (1/epochs) #讓ε的值隨著訓練的進行而慢慢下降，直到0.1（還是要保
```

這段就是進行訓練迴圈
選擇動作>執行動作&取得新狀態>計算目標Q值Y>設定損失函數與反向傳播>收集損失，逐步降低 ε

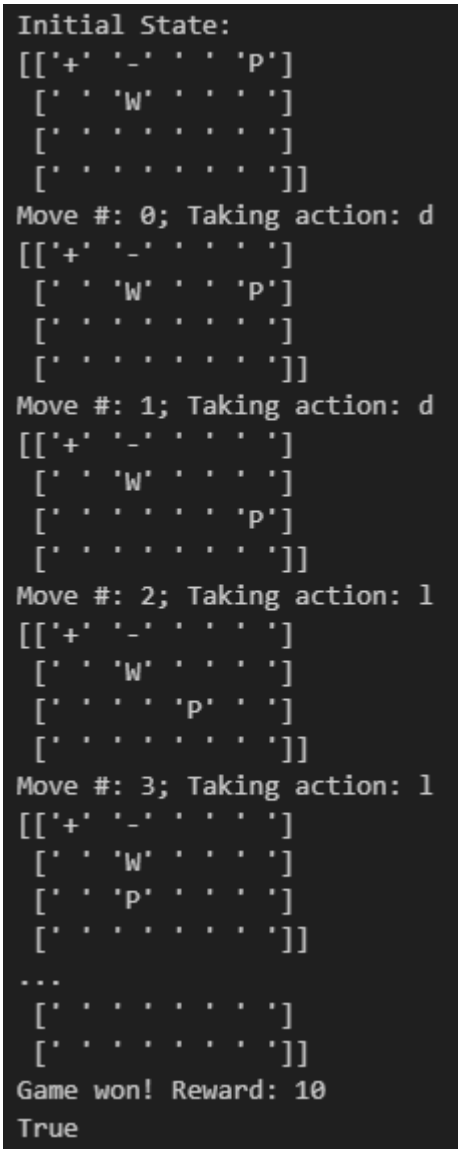
```
m = torch.Tensor([2.0])
m.requires_grad=True
b = torch.Tensor([1.0])
b.requires_grad=True
def linear_model(x,m,b):
  y = m*x + b
  return y
y = linear_model(torch.Tensor([4.]),m,b)
y
```

建立了一個簡單的
線性模型 `y = mx + b` 並使用 PyTorch 來自動計算梯度（gradient）

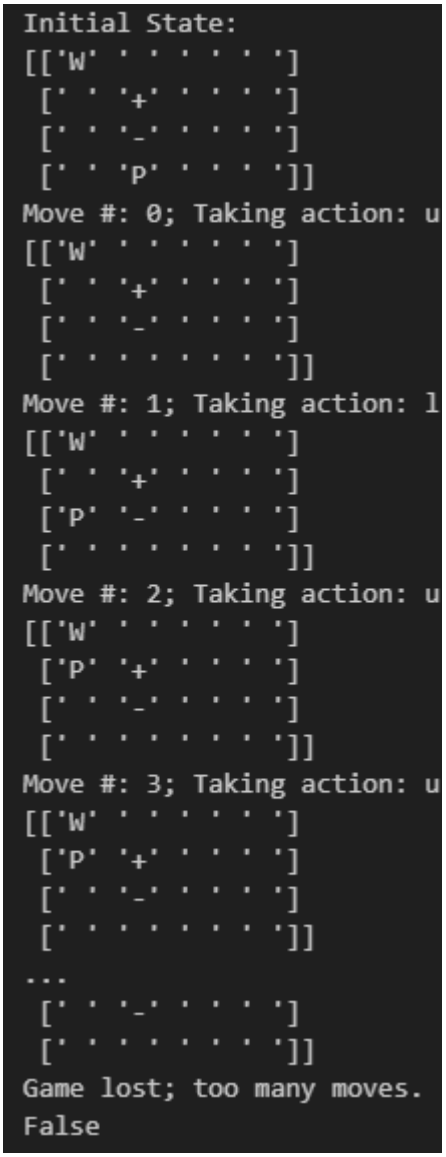
```
def test_model(model, mode='static', display=True):
    i = 0
    test_game = Gridworld(size=4, mode=mode) #產生一場測試遊戲
    state_ = test_game.board.render_np().reshape(1,64) + np.random.rand(1,64
    state = torch.from_numpy(state_).float()
    if display:
      print("Initial State:")
      print(test_game.display())
    status = 1
    while(status == 1): #遊戲仍在進行
      qval = model(state)
      qval_ = qval.data.numpy()
      action_ = np.argmax(qval_)
      action = action_set[action_]
      if display:
```

```
      print('Move #: %s; Taking action: %s' % (i, action))
    test_game.makeMove(action)
    state_ = test_game.board.render_np().reshape(1,64) + np.random.rand(1,6
    state = torch.from_numpy(state_).float()
    if display:
      print(test_game.display())
    reward = test_game.reward()
    if reward != -1: #代表勝利（抵達終點）或落敗（掉入陷阱）
      if reward > 0: #reward>0，代表成功抵達終點
        status = 2 #將狀態設為2，跳出迴圈
        if display:
          print("Game won! Reward: %s" %reward)
        else: #掉入陷阱
          status = 0 #將狀態設為0，跳出迴圈
          if display:
            print("Game LOST. Reward: %s" %reward)
    i += 1 #每移動一步，i就加1
    if (i > 15): #若移動了15步，仍未取出勝利，則一樣視為落敗
      if display:
        print("Game lost; too many moves.")
      break
  win = True if status == 2 else False
  print(win)
  return win
```

用來
測試訓練好的模型（`model`）是否能成功完成 Gridworld 任務（到達終點而不掉入陷阱）

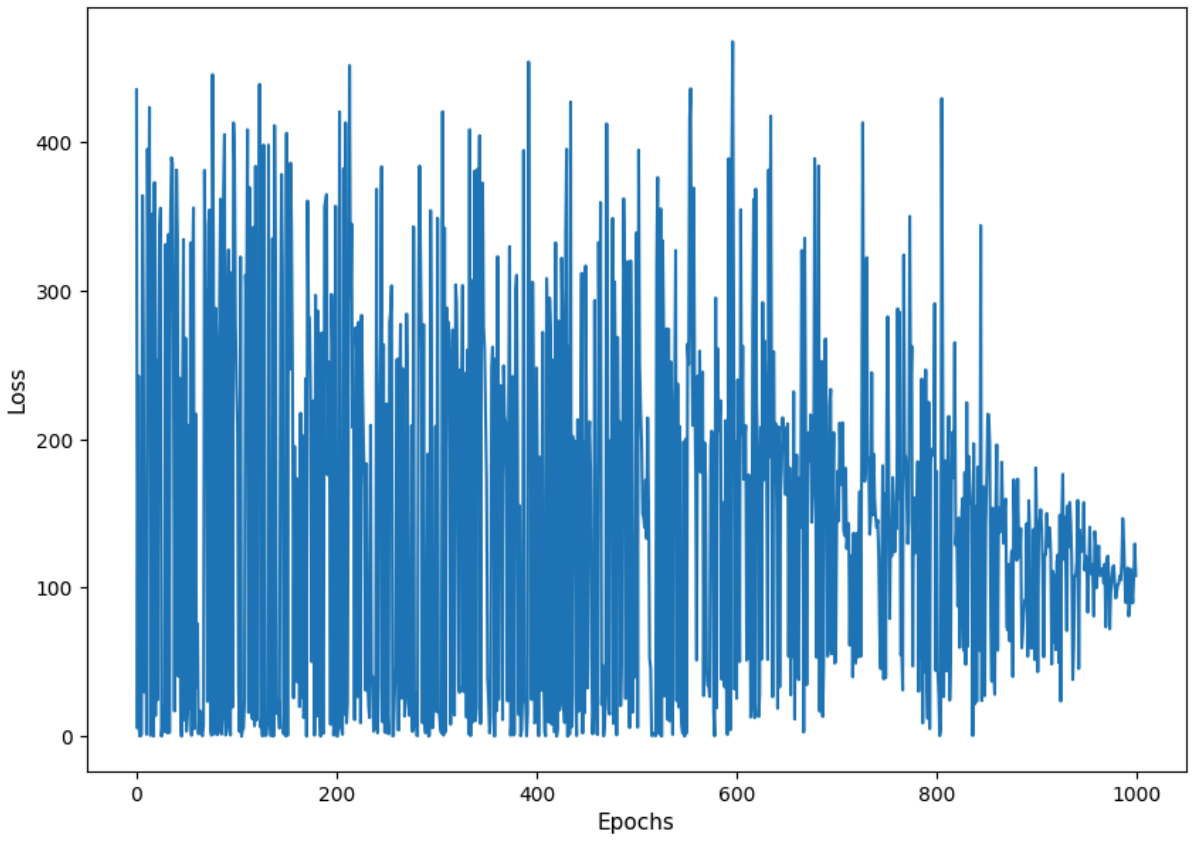


可以看到訓練效果，已經成功抵達而不掉入坑

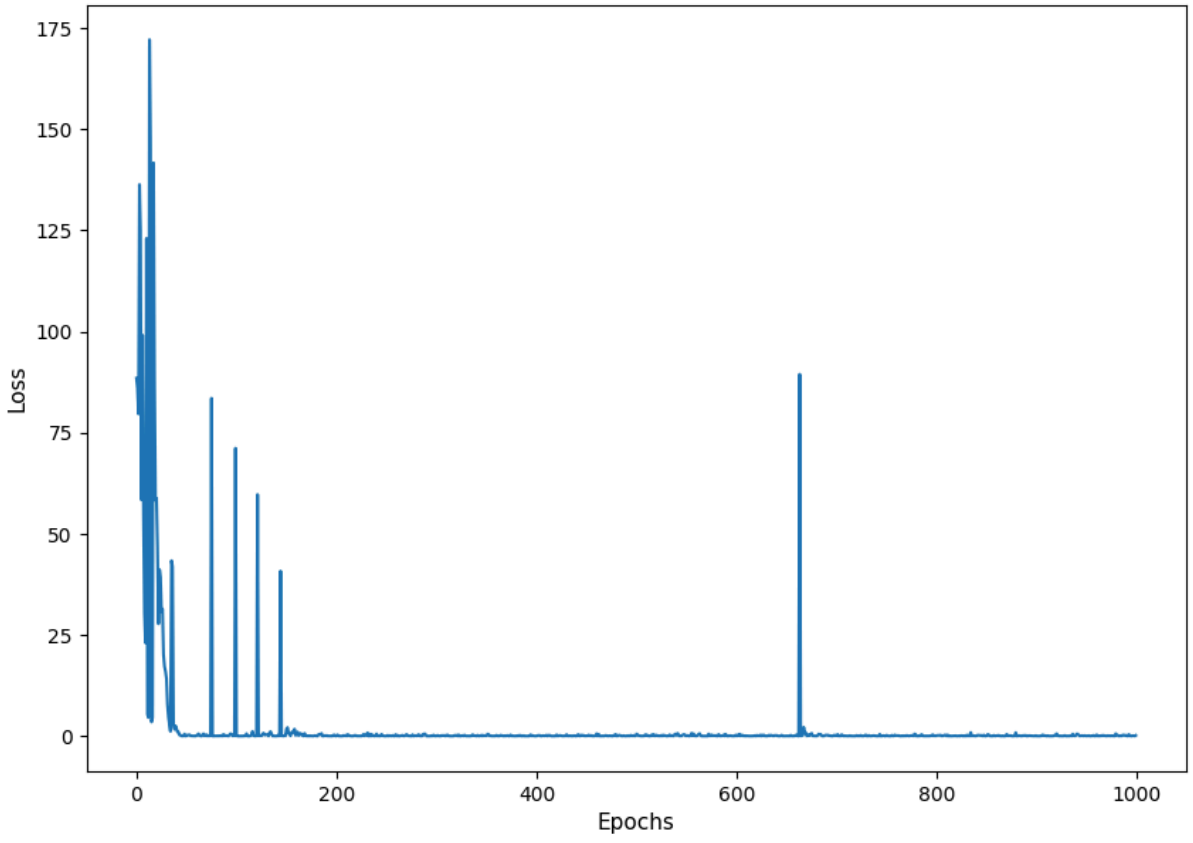


將模式改成random之後，就到不了終點需要重新訓練

- 地圖 **每場都不同**：陷阱和目標位置會變動。
- 模型學到的是**泛化策略**（例如避免陷阱、找到目標的普遍方法）。
- 難度大很多，訓練需要更久，也更考驗探索與策略的學習能力。



mode=random的效果



mode=player的效果，主要功能是「**讓玩家起始位置隨機**，但保留其他物件（牆、目標、陷阱）的位置不變」，也就是結合了 `static` 和 `random` 模式的部分特性

接下來為了讓DQN可以在random和player模式下有較好的表現，重建一個model，重新訓練並分別加入不同的優化機制

經驗回放:

```
from collections import deque

epochs = 5000
losses = []

mem_size = 1000      # 最大記憶庫容量
batch_size = 200     # 每次訓練使用 200 筆資料
replay = deque(maxlen=mem_size) # 雙端佇列記憶庫
max_moves = 50       # 每場最多移動步數
```

先用 `deque` 建立記憶庫，長度最多為 1000

一次訓練從中隨機抽出 200 筆經驗進行訓練（mini-batch)

```
while(status == 1):
  mov += 1
  qval = model(state1) # Q(state1)
  qval_ = qval.data.numpy()

  if (random.random() < epsilon):
    action_ = np.random.randint(0,4)
  else:
    action_ = np.argmax(qval_)

  action = action_set[action_]
  game.makeMove(action)
  ...
  reward = game.reward()
  done = True if reward != -1 else False
```

遊戲進行+儲存經驗

- 遵循 **ε-貪婪策略**，進行探索（隨機動作）或利用（最大Q）。
- 執行動作後取得新狀態 `state2`、回饋 `reward`。
- 若 `reward ≠ -1`，代表遊戲結束（成功或失敗）→ `done = True`

```
exp = (state1, action_, reward, state2, done)
replay.append(exp)
```

紀錄經驗

- 將當前的一筆完整經驗 `(s, a, r, s', done)` 加入 replay buffer。
- 超過 1000 筆時會自動移除舊的資料（因為 `deque` 設有 `maxlen`）。

```
if len(replay) > batch_size:
  minibatch = random.sample(replay, batch_size)
```

開始訓練（當 replay 有足夠資料）

- 當資料夠多，從中隨機抽取 `batch_size` 筆經驗組成 mini-batch。
- 這打破了資料時間上的相關性（decorrelate），增加訓練穩定性。

```
state1_batch = torch.cat([s1 for (s1,a,r,s2,d) in minibatch])
action_batch = torch.Tensor([a for (s1,a,r,s2,d) in minibatch])
reward_batch = torch.Tensor([r for (s1,a,r,s2,d) in minibatch])
state2_batch = torch.cat([s2 for (s1,a,r,s2,d) in minibatch])
done_batch = torch.Tensor([d for (s1,a,r,s2,d) in minibatch])
```

Mini-batch 資料整理

- 分別提取小批次中所有狀態、動作、回饋、新狀態、done 標記成張量。

```
Q1 = model(state1_batch)
with torch.no_grad():
  Q2 = model(state2_batch)

Y = reward_batch + gamma * ((1 - done_batch) * torch.max(Q2,dim=1)[0])
```

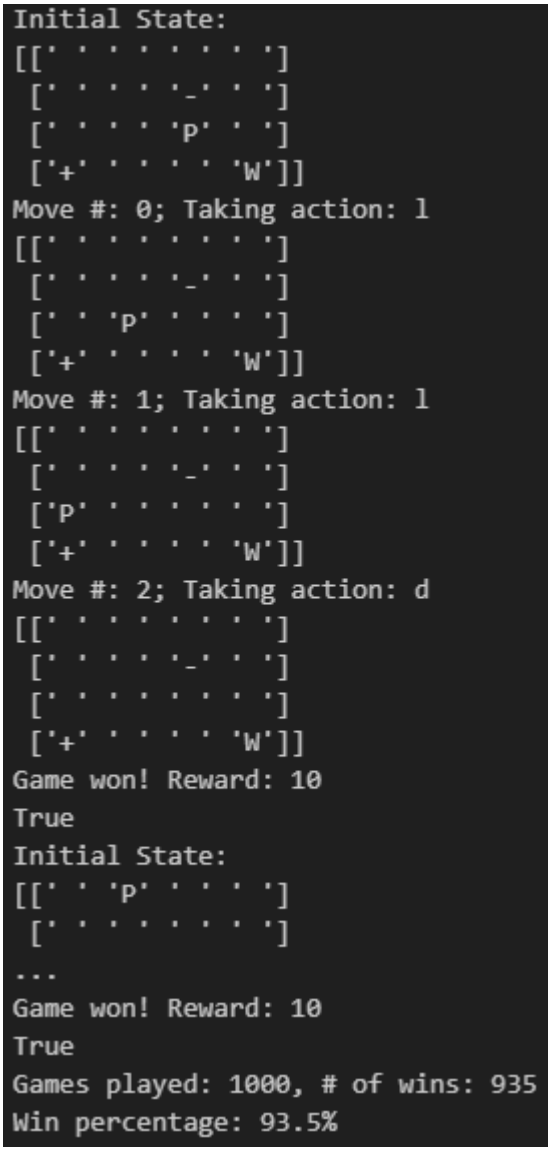
誤差計算與反向傳播

- `X`：實際選的動作對應的 Q 值。

- `loss`：目標 Q 與預測 Q 的差距。
- 執行梯度下降來更新網路參數。

總體來說，經驗回放有以下優點:

- **打破資料間的時間相關性**
  - 從記憶庫隨機抽樣，避免連續樣本導致模型過度擬合短期經驗。
- **提高樣本使用效率**
  - 同一筆經驗可被多次使用，而非只訓練一次即丟棄。
- **讓訓練更加穩定**
  - 資料分布較平均、變化較平緩，避免 Q 值震盪、學習不穩定。
- **能與 mini-batch 搭配使用**
  - 支援並行運算，提高 GPU 計算效率，加快訓練速度。
- **支援較複雜的策略學習**
  - 累積過去的經驗，有助於處理稀疏獎勵或長期依賴問題。



可以看到效果變好了

目標網路(解決高估Q值)+經驗回放

- DQN 中使用**最大化下一步 Q 值（**`max Q(s′, a′)`**）**作為目標值，但這樣會讓 Q 值不斷往上修正，可能造成**Q 值高估偏誤（overestimation bias）**
- 解決方法：引入**目標網路 model2**作為相對穩定的參考，來產生目標 Q 值。
  主網路
  `model` 負責學習和更新，目標網路 `model2` 負責計算 `maxQ`，**但不參與參數更新**。

```
with torch.no_grad():
    Q2 = model2(state2_batch)  # 用 target network 預測 Q 值，但不反向傳播
Y = reward_batch + gamma * ((1 - done_batch) * torch.max(Q2, dim=1)[0])
```

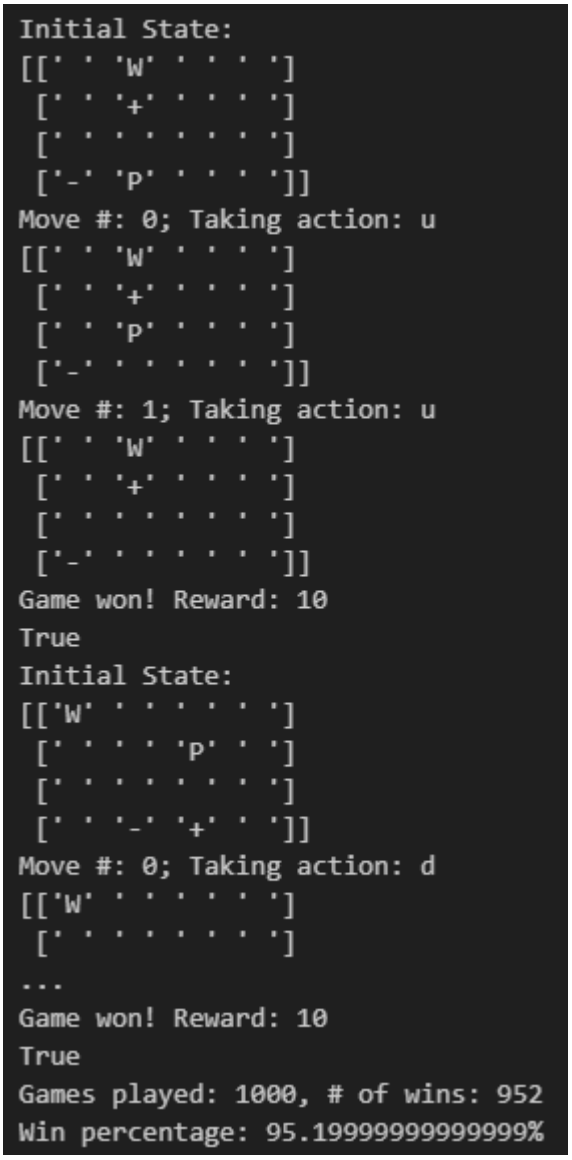
這段就是用
`model2` 產生 **下一步狀態的最大 Q 值**，並搭配回饋 reward 當作目標值 `Y`，而不是用 model 自己預測自己的目標，**降低高估的偏差**。

```
if j % sync_freq == 0:
    model2.load_state_dict(model.state_dict())
```

每隔
`sync_freq=500` 步，才會讓 `model2` 同步 `model` 的參數，讓目標網路的預測更穩定、不會即時波動。

```
replay = deque(maxlen=mem_size)  # 儲存經驗
exp = (state1, action_, reward, state2, done)
replay.append(exp)  # 存進去

# 當記憶體夠大後才開始訓練：
if len(replay) > batch_size:
    minibatch = random.sample(replay, batch_size)
    # 分批取出 s1, a, r, s2, d 進行訓練
```

這段就是經驗回放的核心機制：
資料儲存＋隨機取樣訓練



可以看到勝率又有明顯上升

學習避免撞牆機制

- 這段程式碼中加入了一個重要的新機制：**學習避免撞牆（hit wall penalty）**，目的在於讓 agent 不會傻傻地一直走向牆壁

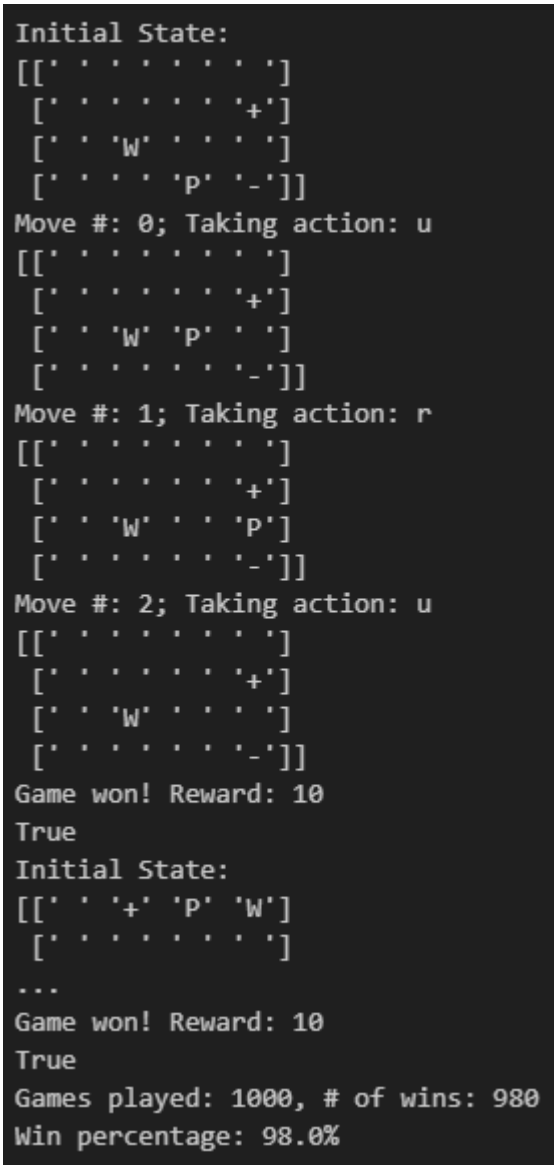
```
hit_wall = game.validateMove('Player', move_pos[action_]) == 1
```

這行使用
`validateMove` 函數檢查 agent 選擇的動作是否會撞到牆壁，如果會，`hit_wall` 為 `True`。

```
reward = -5 if hit_wall else game.reward()
```

- 如果撞牆，給一個**額外的懲罰** `5`。
- 否則就使用原本遊戲邏輯中的 reward（例如 +10 是贏，-10 是輸，-1 是繼續走）。

這樣做的目的是讓 agent 學會避開牆壁，因為每撞一次牆就會被扣 5 分，這對於追求最大總 reward 的 DQN 來說會成為一個負面學習訊號。



學習避免撞牆機制後，勝率略低於目標網路+經驗回放，可能有以下原因:

1. 撞牆懲罰太強，導致策略保守

- 原本一般行動的 reward 是 -1。

- 現在撞牆直接 -5，**懲罰強度高達 5 倍**。
- 可能導致 agent **過度害怕探索靠近牆邊的策略**，例如某些最佳路徑本來就得貼牆走。

2. 沒有目標網路導致 Q 值高估

在這段程式碼中，沒有使用目標網路（`model2`）來穩定訓練，Q 值是直接從主網路 `model` 推論未來回報，這會造成：

- Q 值不穩定，容易高估。
- 搭配撞牆懲罰時，模型可能快速更新錯誤策略。

3. 過度懲罰導致學習偏離真正目標

撞牆懲罰的本意是讓 agent 避免無效移動，但它不是遊戲的最終目標。

- 太強的撞牆懲罰可能讓 agent「為了不撞牆」而放棄追求勝利（例如遠離牆就比較安全，但也離目標遠）。
- 這是一種「**錯誤的目標導向偏差**」。

4. 訓練初期探索不足

在加強懲罰的環境中，如果 agent 在初期因為幾次撞牆受到重罰，可能就：

- 提早偏向保守策略。
- 導致學習不夠全面或陷入 local minimum。