

深度強化式學習 Ch3

Deep Reinforcement Learning

Deep Q-Network



Outline



3-1 狀態價值函數與動作價值函數

3-2 利用Q-learning進行探索

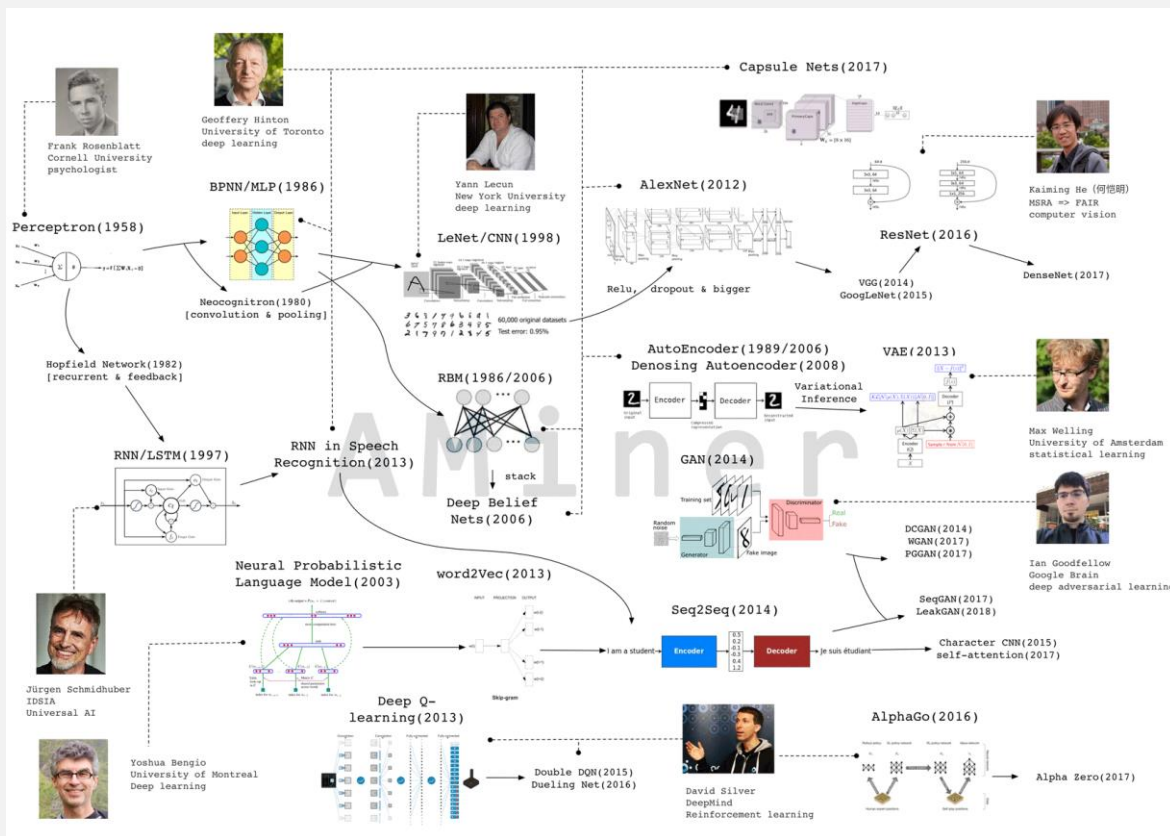
3.3 Q-Learning 與 Deep Learning

3-4 使用目標網路來提升學習穩定性

3-5 回顧

Deep Q-Network

- 深度強化式學習(Deep reinforcement learning)演變的起點：Deep Q-Network

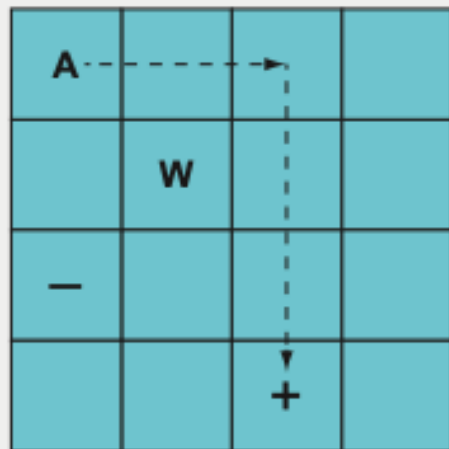


Grid-world



- 什麼是Grid-world ?

- 遊戲包含：玩家(代理人) 及一塊方格板
- 目標：玩家可向上、下、左或右移動到達目標板塊，且必須沿著最短路徑，移動到終點已獲得正回饋值。



A : Agent
W : Wall
- : Pit
+ : Goal

玩家(代理人) : A

牆壁 : W

陷阱 : -

終點 : +

方格塊(Grid Board)

3-1 狀態價值函數與動作價值函數



State 狀態

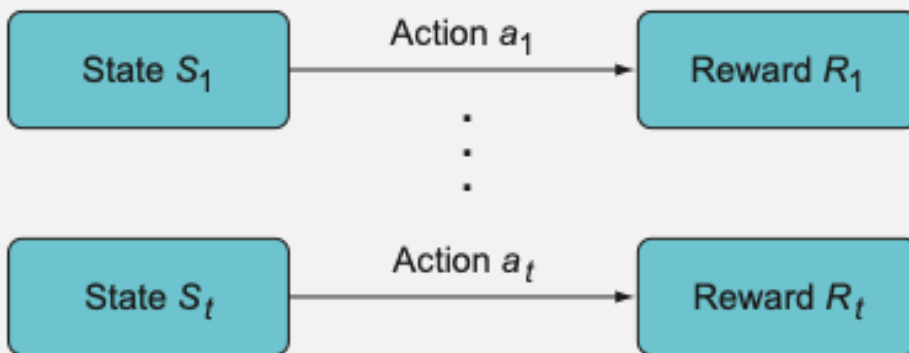
代理人用來決定**該採取什麼行動**的資訊(Grid-world中，為方格中所有物體位置的張量(tensor))

Policy 狀態 π

代理人得到某種狀態資訊所**採取的對策**

Reward 回饋值

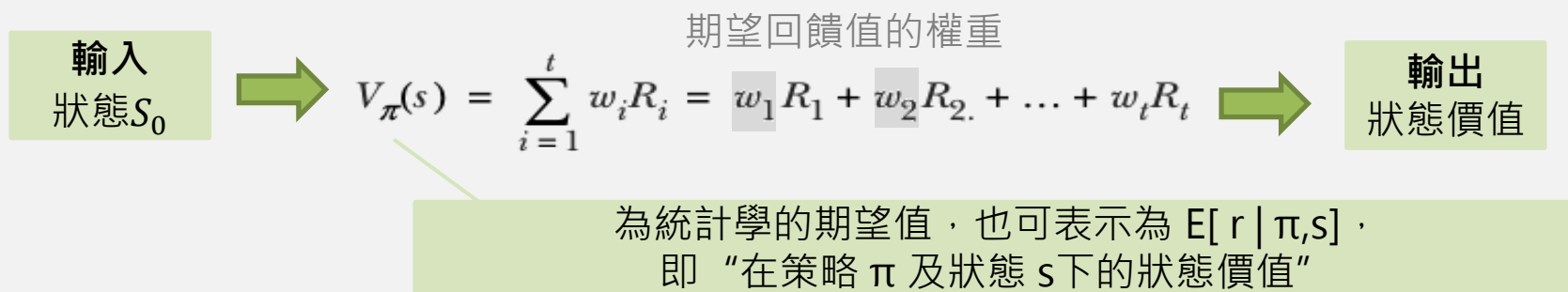
代理人執行完某個動作並使環境進入新狀態後，所得到的值



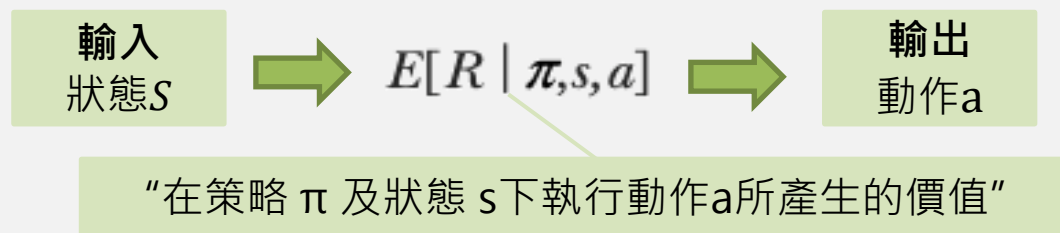
3-1 狀態價值函數與動作價值函數



- 狀態價值函數：從起始狀態 S_0 開始進行 t 輪後，每輪期望回饋值的加權總和



- 動作價值函數（Q函數）

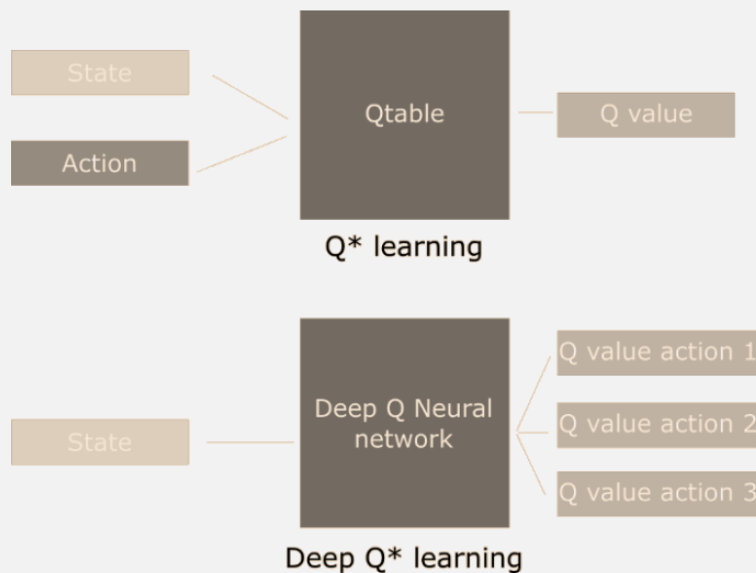


3-2 利用Q-learning進行探索



- 傳統Q-learning演算法的局限：

建立Q-table的 state 和 action 要於有限且不過多的情況下，索引表格才可能被建立

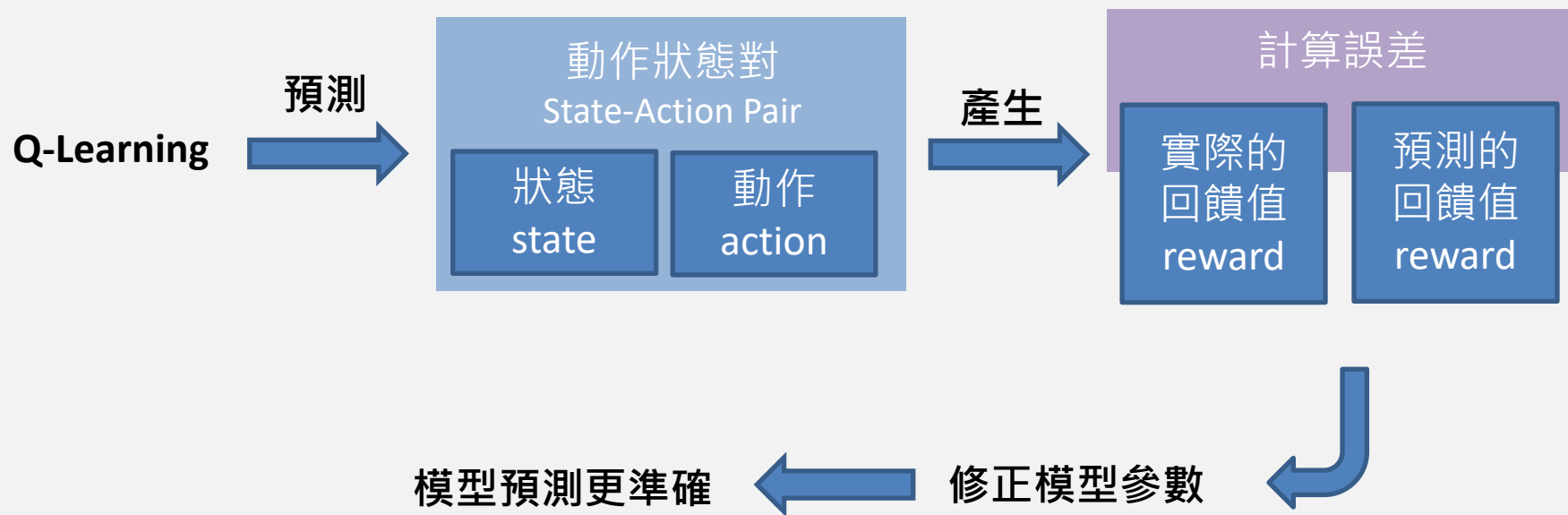


來源：[An introduction to Deep Q-Learning; let's play Doom](#)

3-2-1 Q-learning是什麼?



- **Q-Learning**：實現動作價值函數的一種演算法



3-2-1 Q-learning是什麼?



數學式

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

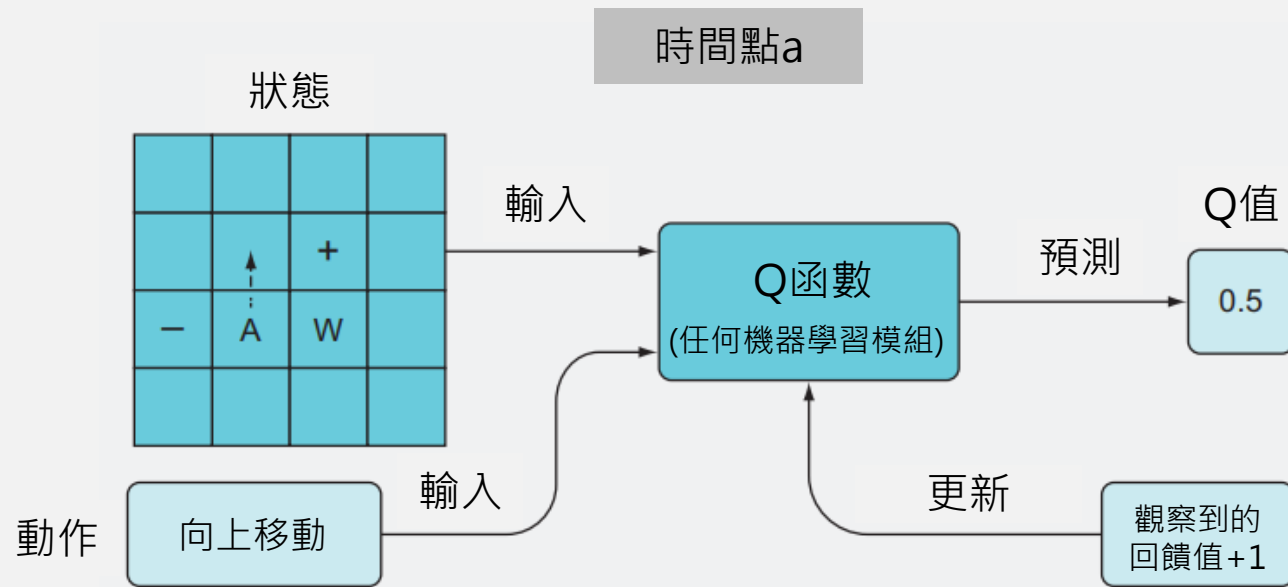
Diagram illustrating the Q-learning update formula with annotations:

- $Q(S_t, A_t)$ (left): 更新後的Q值 (Updated Q-value)
- $Q(S_t, A_t)$ (middle): 當前狀態的Q值 (Current state Q-value)
- α : 學習率 (Learning rate)
- R_{t+1} : 實際得到的回饋值 (Actual received reward)
- γ : 折扣係數 (Discount factor)
- $\max_a Q(S_{t+1}, a)$: 目標Q值 (Target Q-value)
- $Q(S_{t+1}, a)$: 下一個狀態的最大Q值 (Maximum Q-value of the next state)
- $Q(S_t, A_t)$ (right): 當前狀態的Q值 (Current state Q-value)

虛擬碼

```
def get_updated_q_value(old_q_value, reward, state, step_size, discount):  
    term2 = (reward + discount * max([Q(state, action) for action in actions]))  
    term2 = term2 - old_q_value  
    term2 = step_size * term2  
    return (old_q_value + term2)
```

3-2-2 征服Gridworld



狀態、動作 → Q函數 → Q值 → 修正參數 → 預測更準確

3-2-3 學習率與折扣係數



學習率 α

參數更新的幅度

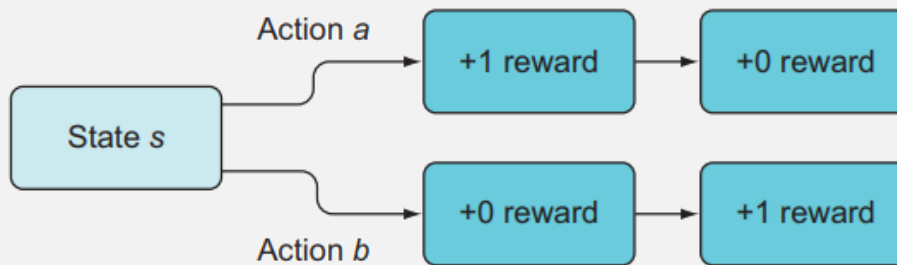
α 小 \rightarrow 參數調整幅度小

α 大 \rightarrow 參數調整幅度大

折扣係數 γ

未來回饋值的影響程度

介於0~1之間
(不可以等於一)



總回饋值相同
價值不同

↓
價值取決於
折扣係數 γ

未來回饋值會比當前的還要小

時間 t 上，折扣係數等於 γ^t

3-2-3 稀疏回饋值問題



輸
掉入陷阱(-10)

贏
抵達終點(+10)

不會直接產生輸贏
(-1)

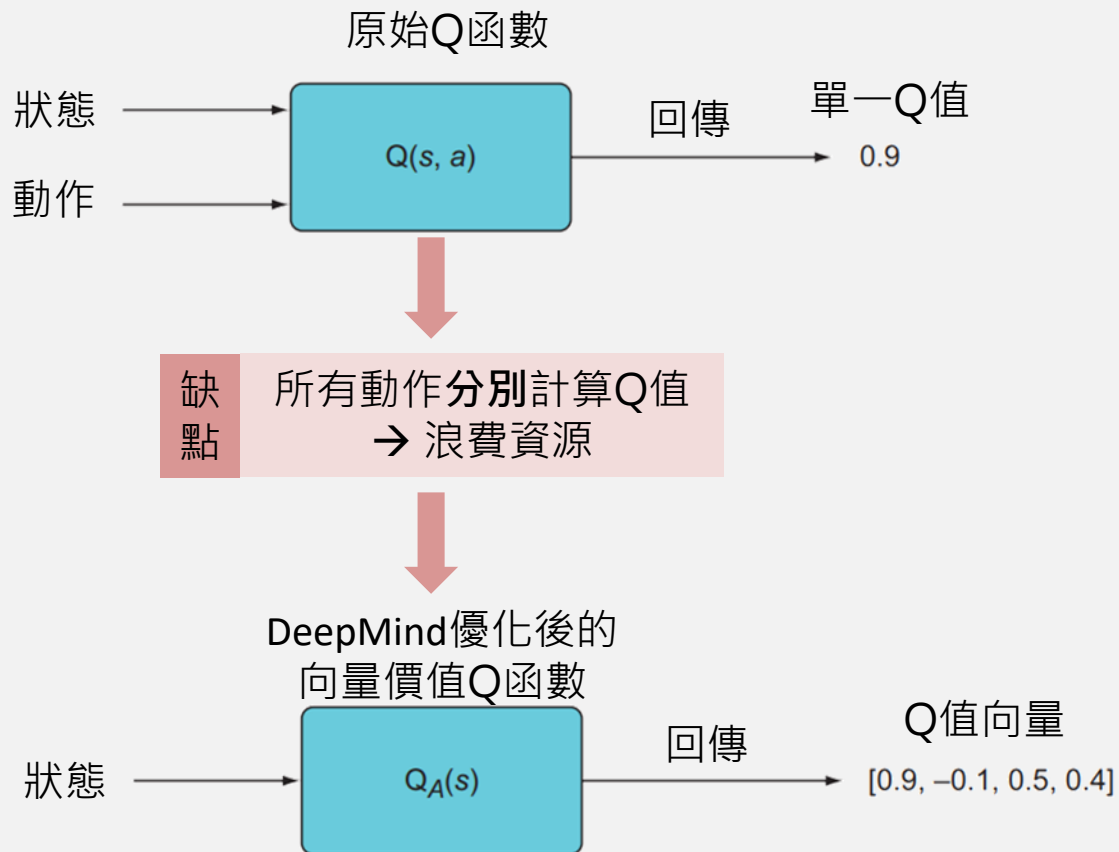
靠近終點, 遠離終點
回饋值都相同

演算法無從得知
此步走得好不好

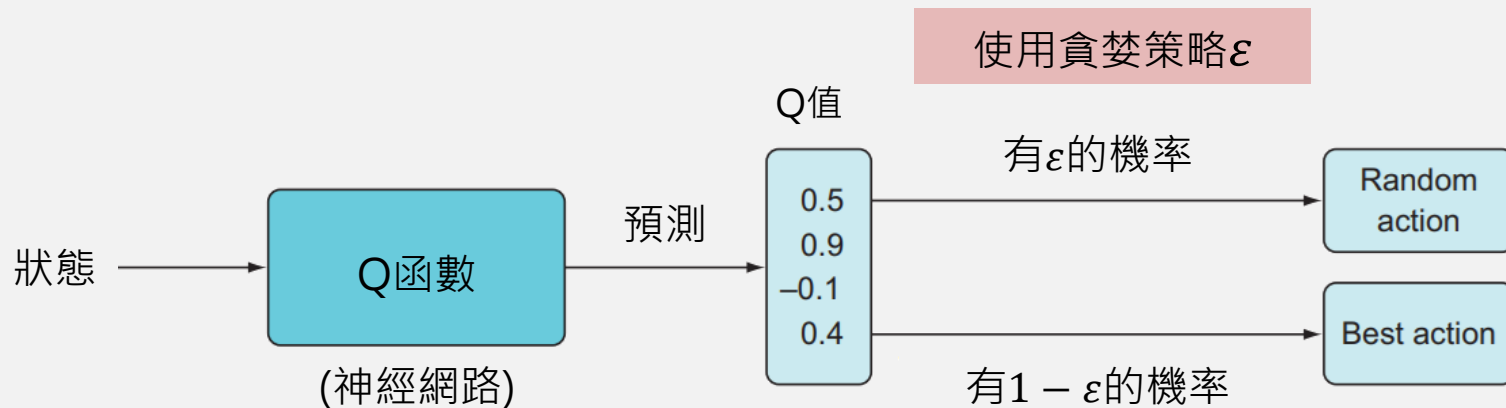


稀疏回饋值問題
(Sparse reward problem)

3-2-4 建構神經網路



3-2-2 征服Gridworld




ϵ 選擇要精準才不會造成過度探索或探索不足的問題

3-2-6 如何建立Gridworld遊戲?



```
from Gridworld import Gridworld
Game = Gridworld(size=4, mode='static') # size : 方格板的邊長, static : 靜態模式
```



static 靜態模式	player 玩家模式	random 隨機模式
所有物體的初始位置 都是固定的	只有玩家的初始模式是 隨機設定	所有物體的初始位置 都隨機設定

3-2-6 如何建立Gridworld遊戲?



```
game.display()
```

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

+ 終點
- 陷阱
W 牆壁
P 玩家(代理人)

```
game.makeMove('d')  
game.display()
```

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

u 向上 l 向左
d 向下 r 向右

```
game.reward() # 輸出該動作所產生的回饋值
```

```
-1
```


3-2-6 如何建立Gridworld遊戲?



`game.board.render_np()` # 顯示遊戲的當前狀態

```
array([[0, 0, 0, 0],
       [0, 0, 0, 1],
       [0, 0, 0, 0],
       [0, 0, 0, 0]],

       [[1, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]],

       [[0, 1, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]],

       [[0, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]]], dtype=uint8)
```

代表 玩家 的向量

代表 終點 的向量

代表 陷阱 的向量

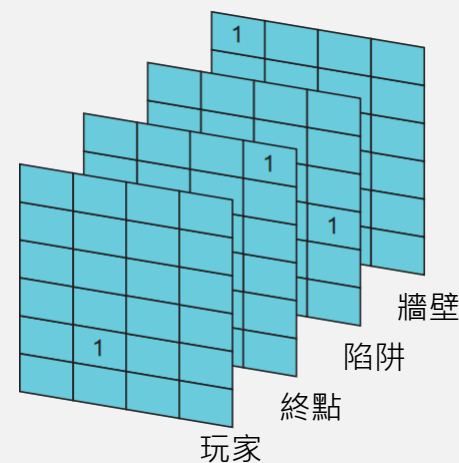
代表 牆壁 的向量

`game.board.render_np().shape` # 輸出狀態張量的shape

`(4, 4, 4)`

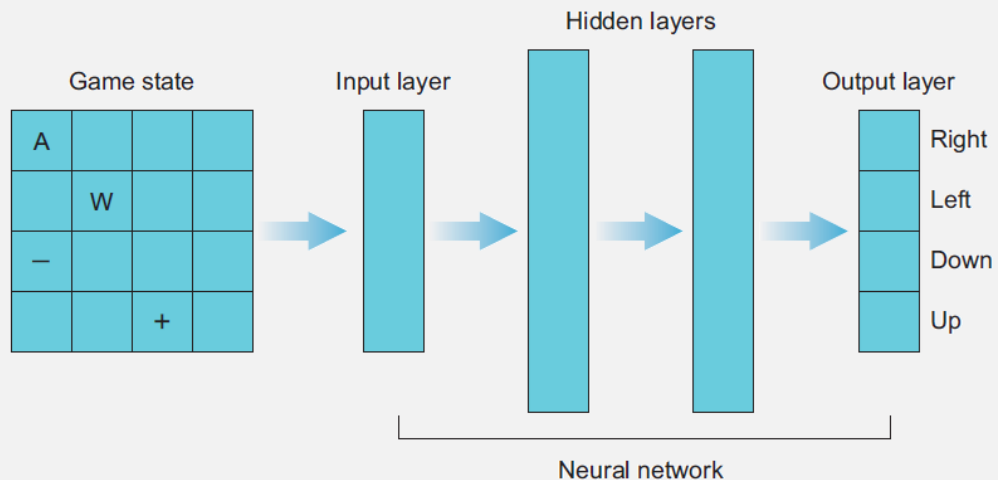
幀	高	寬
frames	height	width

包含四個由方格構成的平面

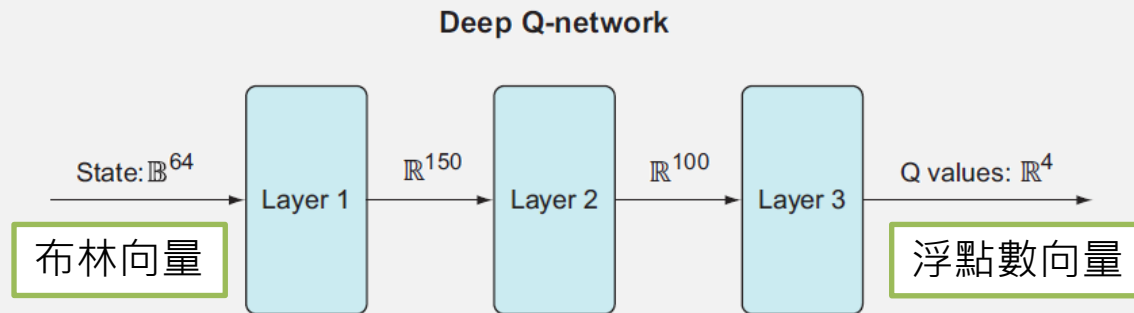


紀錄遊戲中物體的位置

3.2.6 利用神經網路扮演Q函數的角色



- 一個可以接受64維狀態向量的輸入層
- 兩個隱藏層
- 一個能產生4維向量的輸出層(代表4個不同的動作)



3.2.6 利用神經網路扮演Q函數的角色



● 建構神經網路模型

```
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 # 設定學習率為0.001
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
# 指定優化器為Adam，其中model.parameters會傳回所有要優化的權重參數
gamma = 0.9 # 折扣因子
epsilon = 1.0 #  $\epsilon$ -貪婪策略
```

```
# 需要import的模組
import numpy as np
import torch
from Gridworld import Gridworld
from IPython.display import clear_output
import random
from matplotlib import pylab as plt
```

主要訓練迴圈



```
epochs = 1000
losses = [] # 使用串列將每一次的loss記錄下來，方便之後將loss的變化趨勢畫成圖
for i in range(epochs):
    game = Gridworld(size=4, mode='static') # 建立遊戲，設定方格邊長為4，物體初始位置為是為static
    state_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
    # 將3階的狀態陣列 ( 4x4x4 ) 轉換成向量 ( 長度為64 )，並將每個值都加上一些雜訊 ( 很小的數值 )。
    state1 = torch.from_numpy(state_).float() # 將NumPy陣列轉換成PyTorch張量，並存於state1中
    status = 1 # 用來追蹤遊戲是否仍在繼續
    while(status == 1): # 『1』代表仍在繼續
        qval = model(state1) # 執行Q網路，取得4個動作的預測Q值
        qval_ = qval.data.numpy() # 將qval轉換成NumPy陣列
        # 依照ε-貪婪策略選擇動作
        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4個動作中Q值最大的 ( 探索 )
        action = action_set[action_] # 將代表某動作的數字對應到makeMove()的英文字母
        game.makeMove(action) # 執行之前ε-貪婪策略所選出的動作
        # 動作執行完畢，取得遊戲的新狀態並轉換成張量
        state2_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
        state2 = torch.from_numpy(state2_).float()
        reward = game.reward()
```

```
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』
}
```

主要訓練迴圈



```
with torch.no_grad(): # 指示pytorch不要生成運算圖
    newQ = model(state2.reshape(1,64)) # 訓練神經網路的學習目標
    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Q值就等於回饋值
        Y = reward
    Y = torch.Tensor([Y]).detach() # 把y節點從整張運算圖分離出來
    X = qval.squeeze()[action_]
    # 將演算法對執行的動作所預測的Q值存進x，並使用squeeze()將qval中維度為1的階去掉 (shape[1,4]會變成[4])
    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() # 將梯度設為零
    loss.backward() # 反向傳播，更新梯度
    optimizer.step() # 根據梯度更新參數
    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0
```

主要訓練迴圈 torch.no_grad()



```
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
```

```
tensor([9.], grad_fn=<AddBackward0>)
```

```
y.grad_fn
```

```
<AddBackward0 at 0x2ac03a9f148>
```

```
with torch.no_grad():
    y = linear_model(torch.Tensor([4.]),m,b)
y
```

```
tensor([9.])
```

```
y.grad_fn
```

None 不會有任何東西

AddBackward()是梯度函數的一種
儲存節點y對m和b參數的偏微分結果
也就是:

$$\frac{\partial y}{\partial m} = x \text{ 和 } \frac{\partial y}{\partial b} = 1$$

主要訓練迴圈



```
with torch.no_grad(): # 指示pytorch不要生成運算圖
    newQ = model(state2.reshape(1,64)) # 訓練神經網路的學習目標
    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Q值就等於回饋值
        Y = reward
    Y = torch.Tensor([Y]).detach() # 把y節點從整張運算圖分離出來
    X = qval.squeeze()[action_]
    # 將演算法對執行的動作所預測的Q值存進x，並使用squeeze()將qval中維度為1的階去掉 (shape[1,4]會變成[4])
    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() # 將梯度設為零
    loss.backward() # 反向傳播，更新梯度
    optimizer.step() # 根據梯度更新參數
    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0
```

主要訓練迴圈 backward()



```
y = linear_model(torch.Tensor([4.]),m,b)
y.backward()
m.grad
```

```
tensor([4.])
```

```
b.grad
```

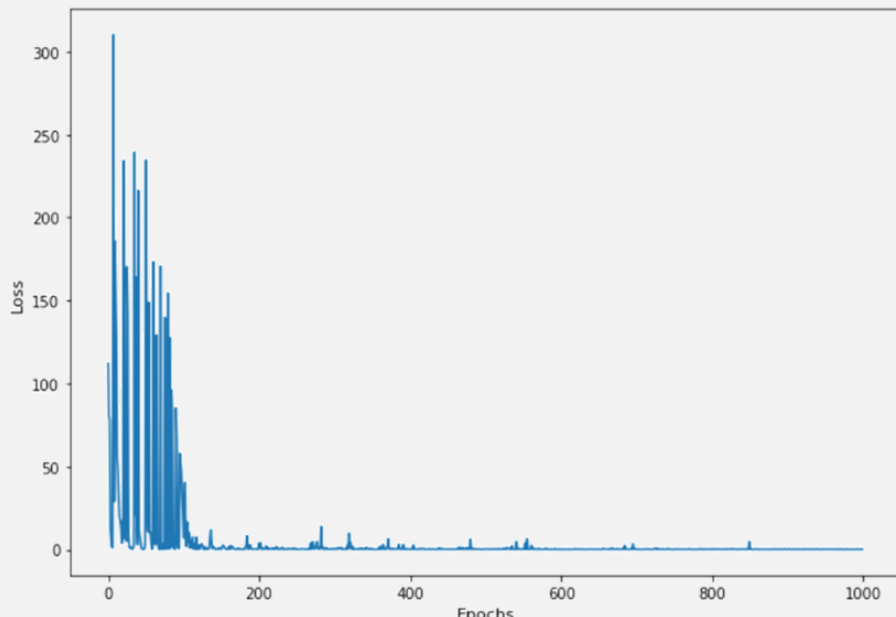
```
tensor([1.])
```


主要訓練迴圈



```
losses.append(loss.item())  
if epsilon > 0.1:  
    epsilon -= (1/epochs) # 讓 $\epsilon$ 的值隨著訓練的進行而慢慢下降，直到0.1 (還是要保留探索的動作)  
plt.figure(figsize=(10,7))  
plt.plot(losses)  
plt.xlabel("Epochs",fontsize=11)  
plt.ylabel("Loss",fontsize=11)
```

Output :





```
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)/10.0
    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(1,64)/10.0
        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
```

測試模型 - Static mode



```
test_model(model, 'static')
```

```
Initial State:
```

```
[[ '+' '-' ' ' ' ' 'P' ]  
[ ' ' 'W' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]
```

```
Move #: 0; Taking action: d
```

```
[[ '+' '-' ' ' ' ' ' ' ]  
[ ' ' 'W' ' ' 'P' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]
```

```
Move #: 1; Taking action: d
```

```
[[ '+' '-' ' ' ' ' ' ' ]  
[ ' ' 'W' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' 'P' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]
```

```
Move #: 2; Taking action: 1
```

```
[[ '+' '-' ' ' ' ' ' ' ]  
[ ' ' 'W' ' ' ' ' ' ' ]  
[ ' ' ' ' 'P' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]
```

```
Move #: 3; Taking action: 1
```

```
[[ '+' '-' ' ' ' ' ' ' ]  
[ ' ' 'W' ' ' ' ' ' ' ]  
[ ' ' 'P' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]
```

```
Move #: 4; Taking action: 1
```

```
[[ '+' '-' ' ' ' ' ' ' ]  
[ ' ' 'W' ' ' ' ' ' ' ]  
[ 'P' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]
```

```
Move #: 5; Taking action: u
```

```
[[ '+' '-' ' ' ' ' ' ' ]  
[ 'P' 'W' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]
```

```
Move #: 6; Taking action: u
```

```
[[ '+' '-' ' ' ' ' ' ' ]  
[ ' ' 'W' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]
```

```
Game won! Reward: 10
```

```
True
```

測試模型 - Random mode



Initial State:

```
[[ ' ' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ 'P' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 0; Taking action: u

```
[[ ' ' '+' ' ' ' ' ' ' ]  
[ 'P' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 1; Taking action: u

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 2; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 3; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 4; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 5; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 6; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 7; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 8; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 9; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 10; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 11; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 12; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 13; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 14; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

Move #: 15; Taking action: l

```
[[ 'P' '+' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' ' ' ' ' ' ' ' ' ]  
[ ' ' '-' 'W' ' ' ' ' ]]
```

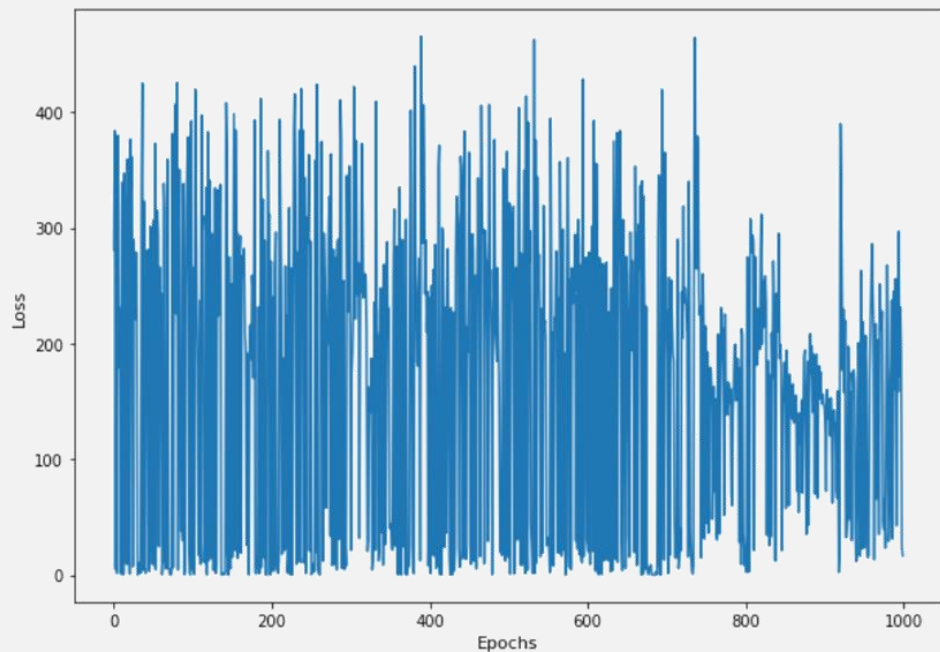
Game lost; too many moves.
False

test_model(model, 'random')

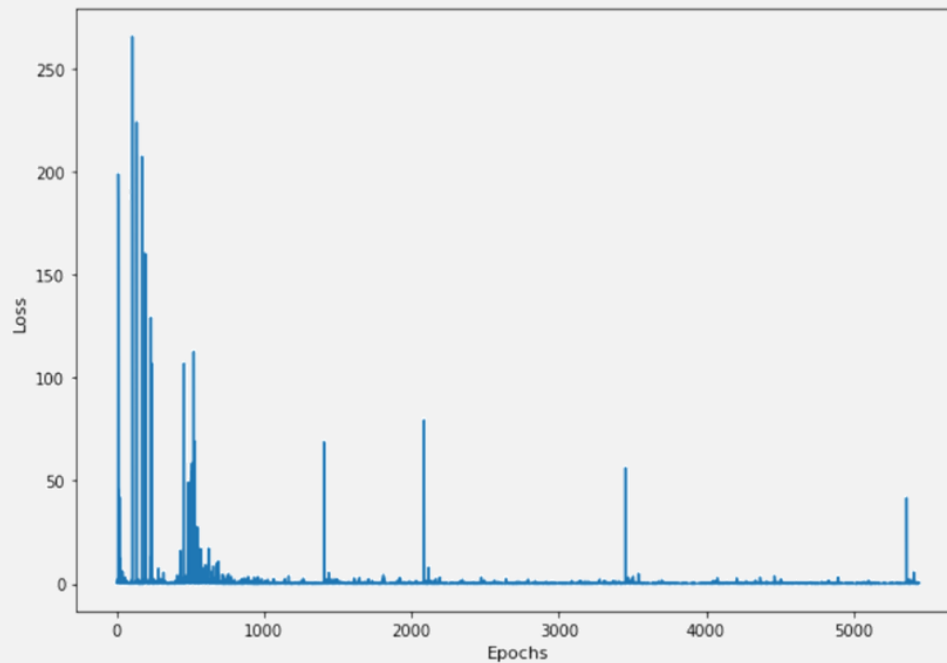
更改訓練方式 - Random/Player



遊戲生成模式: Random



遊戲生成模式: Player

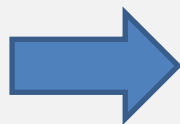


3-3 避免災難性失憶的發生：經驗回放



- 當環境發生改變
 - 過去的經驗造成很差的結果

-	A	+	
		W	



-		A	
		W	

Reward : +10

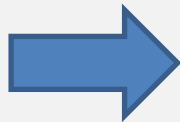
更新

Q函數

災難性失憶!

- 不同的環境:

+	A	-	
		W	



+		A	
		W	

Reward : -10

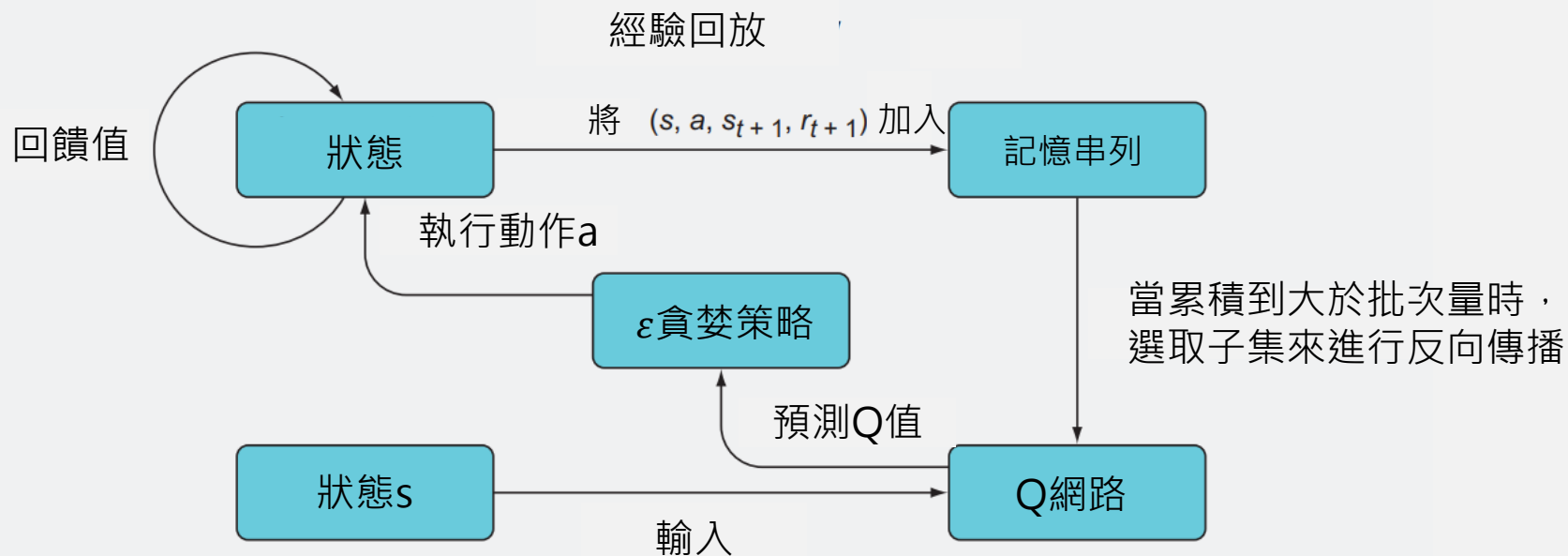
更新

Q函數

經驗回放流程



- ✓ 在即時學習中加入批次更新(Batch Updating)



實驗參數設定



```
from collections import deque
epochs = 5000 # 訓練5000次
losses = []
mem_size = 1000 # 設定記憶串列的大小
batch_size = 200 # 設定單一小批次(mini_batch)的大小
replay = deque(maxlen=mem_size) # 產生一個記憶串列(資料型別為deque)來儲存經驗回放的資料，並將其命名為
replay
max_moves = 50 # 設定每場遊戲最多可以走幾步
```

執行Q網路



```
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state1_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
    state1 = torch.from_numpy(state1_).float() # 現在的狀態 $S_t$ 
    status = 1
    mov = 0 # 記錄移動的步數，初始化為0
    while(status == 1):
        mov += 1
        qval = model(state1) # 輸出各動作的Q值
        qval_ = qval.data.numpy()
        if (random.random() < epsilon):
            action_ = np.random.randint(0,4)
        else:
            action_ = np.argmax(qval_) # 貪婪策略
        action = action_set[action_] # 透過現在的狀態所產生的Action
        game.makeMove(action)
        state2_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
        state2 = torch.from_numpy(state2_).float() # 下一個狀態 $S_t + 1$ 
```

進行經驗回放



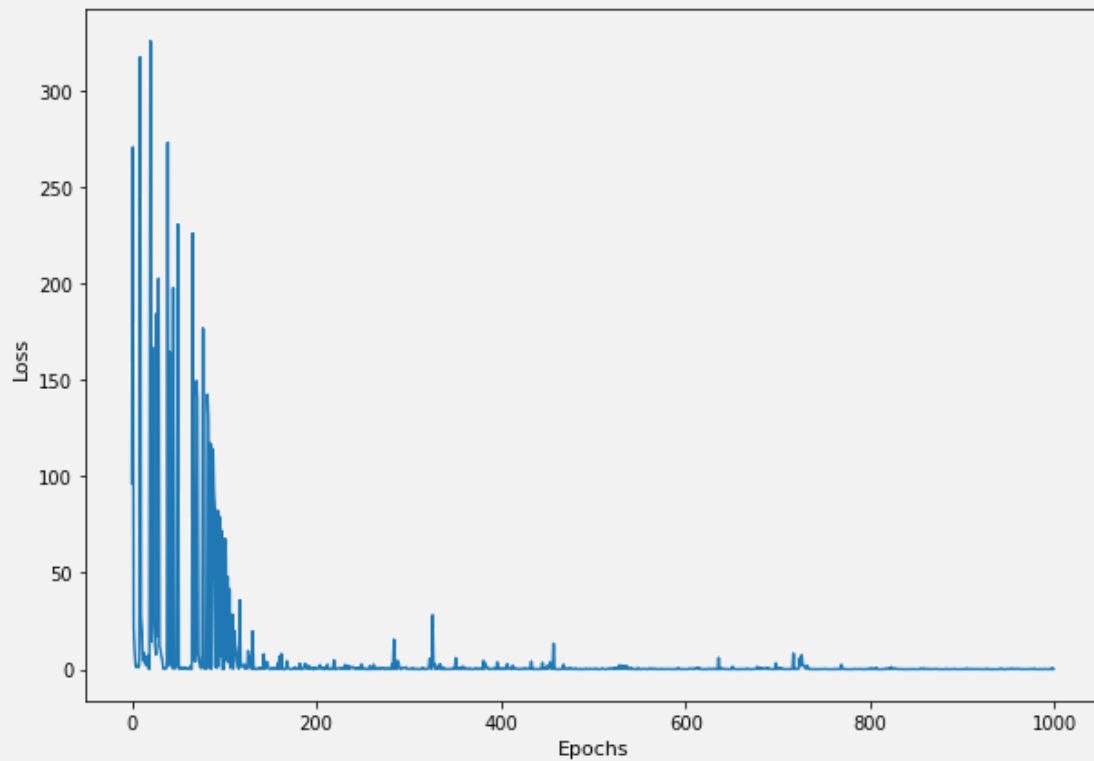
```
reward = game.reward() # 經過Action所產生的reward
done = True if reward != -1 else False
# 在reward不等於-1時設定done=True，代表遊戲已經結束了 (分出勝負時，reward會等於10或-10)
exp = (state1, action_, reward, state2, done)
# 產生一筆經驗，其中包含當前狀態、動作、新狀態、回饋值及done值
replay.append(exp) # 將該經驗加入名為replay的deque串列中
state1 = state2 # 產生的新狀態會變成下一次訓練時的輸入狀態
if len(replay) > batch_size: # 當replay的長度大於小批次量 (mini-batch size)時，啟動小批次訓練
    minibatch = random.sample(replay, batch_size)
    state1_batch = torch.cat([s1 for (s1,a,r,s2,d) in minibatch])
    # 將經驗中的不同元素分別儲存到對應的小批次張量中
    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])
    Q1 = model(state1_batch) # 利用小批次資料中的『目前狀態批次』來計算Q值
    with torch.no_grad():
        Q2 = model(state2_batch) # 利用小批次資料中的新狀態來計算Q值，但設定為不需要計算梯度
```

進行經驗回放



```
Y = reward_batch + gamma * ((1 - done_batch) * torch.max(Q2,dim=1)[0])
X = Q1.gather(dim=1,index=action_batch.long().unsqueeze(dim=1)).squeeze()
loss = loss_fn(X, Y.detach())
print(i, loss.item())
clear_output(wait=True)
optimizer.zero_grad()
loss.backward()
optimizer.step()
if abs(reward) == 10 or mov > max_moves:
    status = 0
    mov = 0 # 若遊戲結束，則重設status和mov變數的值
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 (還是要保留探索的動作)
losses = np.array(losses)
plt.figure(figsize=(10,7))
plt.plot(losses)
plt.xlabel("Epochs", fontsize = 11)
plt.ylabel("Loss", fontsize = 11)
```

顯示結果

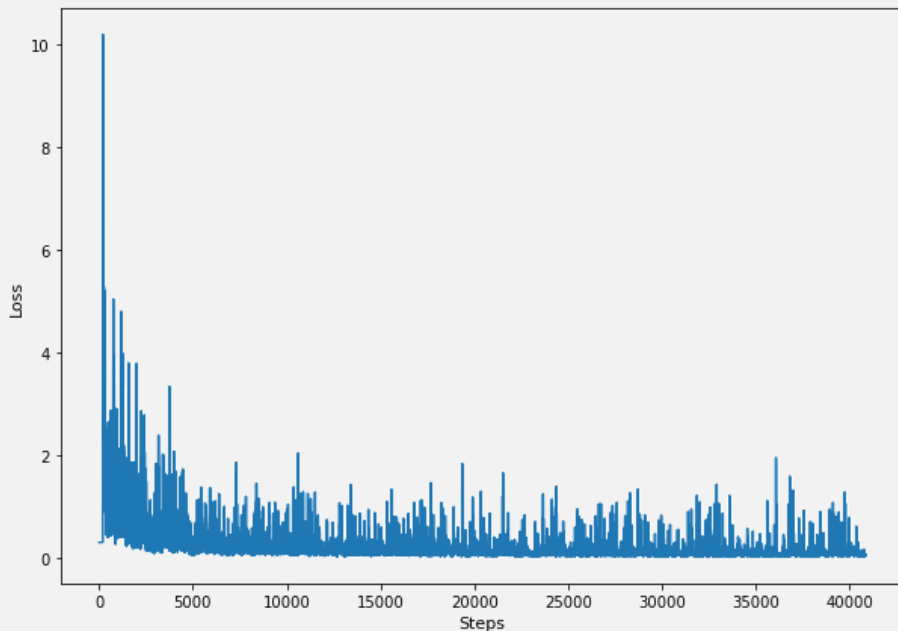


Loss 明顯下降

3-4 使用目標網路來提升學習穩定性



- 代理人能在
 - ① static模式(所有物體的初始位置固定)或
 - ② player模式(只有玩家位置隨機變化)掌握Gridworld遊戲
- 進階使用random模式也得到不錯的成效
- 瓶頸：損失圖的雜訊太多



→ 目標：
讓訓練的過程更加穩定

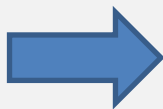
3.4.1 學習的不穩定性



- 每執行一個迴圈，就更新一次Q網路中的參數 → 學習的不穩定性(instability)將提高。

稀疏回饋值問題

代理人只有在獲勝或輸掉
遊戲時才提供顯著的回饋
(+10或-10)



解決方法

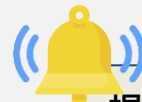
- 將Q網路複製兩份(兩模型參數互相獨立)。
- 一個稱為主要Q網路
另一個則稱為目標網路(target network)，記為 \hat{Q}
- 在未經任何訓練以前，目標網路和主要Q網路是一模一樣的

3.4.1 學習的不穩定性



- 目標網路在訓練過程中發揮了什麼作用(忽略經驗重播的細節)
 1. 初始化主要網路中的參數，這些參數(神經網路的權重)統稱為 θ_Q
 2. 將主要Q網路複製一份，產生一個參數為 θ_T 的目標網路並設定 $\theta_T = \theta_Q$
 3. 使用 ϵ -貪婪策略，參考主要Q網路所預測的Q值選擇動作a
 4. 觀察動作a所產生的回饋值 r_{t+1} 和新狀態 s_{t+1}
 5. 若動作a讓遊戲結束(分出勝負)，則將目標Q值設為 r_{t+1} ，否則就等於 $r_{t+1} + \gamma \max Q \theta_T(s_{t+1})$
 6. 利用目標網路所輸出的目標Q值對主要Q網路(注意!不是目標網路)進行反向傳播。
 7. 過了C次迴圈(C值可自己設定)後，重設 $\theta_T = \theta_Q$ (藉此更新目標網路內的參數)。

3.4.1 學習的不穩定性

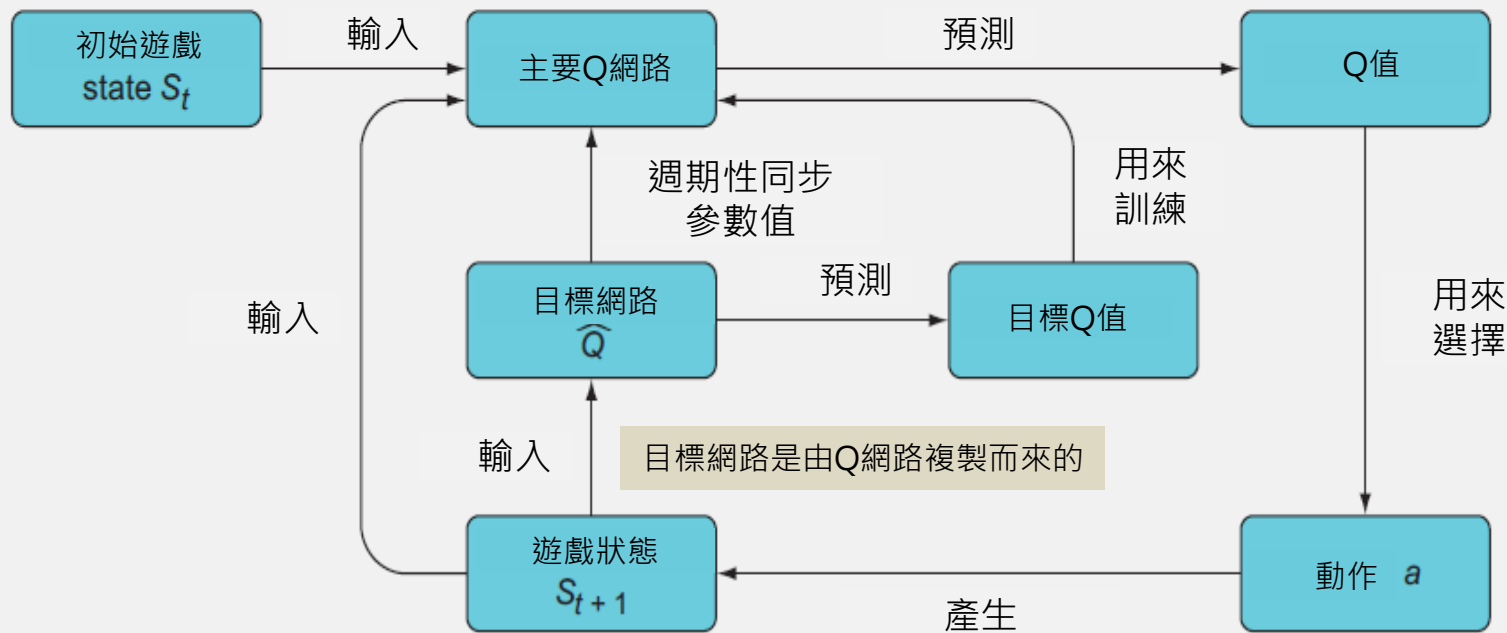


提高穩定性

用目標網路所產生的目標Q值來訓練Q網路，而不是主要Q網路本身的Q值

- ✓ \hat{Q} 網路的唯一功能：
 - 計算出Q網路進行網路進行反向傳播所需的目標Q值

在Q-learning演算法中加入目標網路



3.4.1 學習的不穩定性



```
import copy
```

```
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  
)
```

根據搭建模型的輸入、輸出和層次結構需求，中間需要經過三次線性變換，所以要使用三個線性層

3.4.1 學習的不穩定性



- 目標網路

```
model2 = copy.deepcopy(model) # 完整複製原始Q網路模型，產生目標網路模型
model2.load_state_dict(model.state_dict()) # 將原始Q網路中的參數複製給目標網路
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)
# 指定優化器為Adam，其中model.parameters會傳回所有要優化的權重參數

gamma = 0.9 # 折扣因子
epsilon = 1.0
```

補充：使用load_state_dict()來確保參數與原始的Q網路相同。

3.4.1 學習的不穩定性



- 利用經驗回放和目標網路訓練DQN

```
from collections import deque
epochs = 5000
losses = [] # 使用串列將每一次的loss記錄下來，方便之後將loss的變化趨勢畫成圖
mem_size = 1000 # 設定記憶串列的大小
batch_size = 200 # 設定批次大小
replay = deque(maxlen=mem_size) # 產生一個記憶串列（資料型別為deque）來儲存經驗回放的資料，並將其命名為replay
max_moves = 50
sync_freq = 500 # 設定Q網路和目標網路的參數同步頻率（每500步就同步一次參數）
j=0 # 記錄當前訓練次數
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state1_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0 # 將3階的狀態陣列（4x4x4）轉換成向量（長度為64），並將每個值都加上一些雜訊（很小的數值）
    state1 = torch.from_numpy(state1_).float() # 將NumPy陣列轉換成PyTorch張量，並存於state1中
    status = 1 # 用來追蹤遊戲是否仍在繼續（『1』代表仍在繼續）
    mov = 0 # 記錄移動的步數，初始化為0
```

3.4.1 學習的不穩定性



```
while(status == 1): # 遊戲仍在進行 和mov = 0同排
    j += 1 # 將訓練次數加1
    mov += 1 # 移動的步數+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()的英文字母
    game.makeMove(action) # 執行策略所選出的動作
    state2_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
    state2 = torch.from_numpy(state2_).float() # 動作執行完畢，取得遊戲的新狀態並轉換成張量
    reward = game.reward()
    done = True if reward != -1 else False # 在reward不等於-1時設定done=True，代表遊戲已經結束了（分出勝負時，reward會等於10或-10）
    exp = (state1, action_, reward, state2, done) # 產生一筆經驗，其中包含當前狀態、動作、新狀態、回饋值及done值
    replay.append(exp) # 將該經驗加入名為replay的deque串列中
    state1 = state2 # 產生的新狀態會變成下一次訓練時的輸入狀態
```

3.4.1 學習的不穩定性



if len(replay) > batch_size: 接續前面的程式，和state1 = state2同排

```
minibatch = random.sample(replay, batch_size) # 隨機選擇replay中的資料來組成子集
```

```
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])
```

```
Q1 = model(state1_batch) # 利用小批次資料中的目前狀態來計算Q值
```

```
with torch.no_grad(): # 用目標網路模型計算Q值, 但不要優化模型的參數
```

```
    Q2 = model2(state2_batch) # 利用小批次資料中的新狀態來計算Q值，但設定為不需要計算梯度
```

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

```
X = Q1.gather(dim=1,index=action_batch.long().unsqueeze(dim=1)).squeeze() #將4])
```

loss = loss_fn(X, Y.detach()) #計算目標Q值與預測Q值之間演算法對執行的動作所預測的Q值存進X，並使用
squeeze()將qval中維度為1的階去掉 (shape[1,4]會變成[的誤差

```
print(i, loss.item())
```

```
clear_output(wait=True)
```

```
optimizer.zero_grad()
```

```
loss.backward()
```

```
optimizer.step()
```

將經驗中的不同元素
分別儲存到對應的小
批次張量中

3.4.1 學習的不穩定性



```
if j % sync_freq == 0: #每500步，就將Q網路當前的參數複製一份給目標網路
    model2.load_state_dict(model.state_dict())
if reward != -1 or mov > max_moves:
    status = 0
    mov = 0
    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 (還是要保留探索的動作)
plt.figure(figsize=(10,7))
plt.plot(losses)
plt.xlabel("Steps",fontsize=11)
plt.ylabel("Loss",fontsize=11)
```

未分出勝負前
所有動作的
回饋值都是-1

代理人獲勝或輸掉遊戲時才提供顯著的回饋(+10或-10)

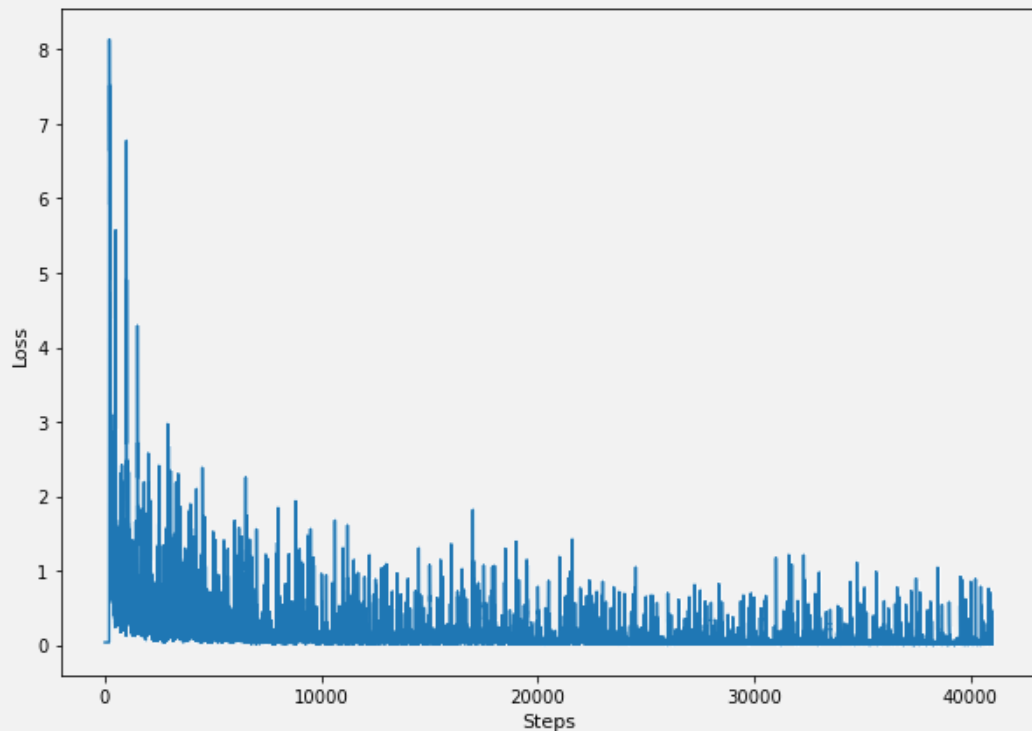
補充(3.5程式)：

```
if abs(reward) == 10 or mov > max_moves:
```

3.4.1 學習的不穩定性



● 損失圖



結果

雜訊仍大
但呈現明顯的下降趨勢
模型的表現和超參數息息相關
可以自行調參，觀察效果

3.4.1 學習的不穩定性



- 改良版 (加入『學習避免撞牆』機制)

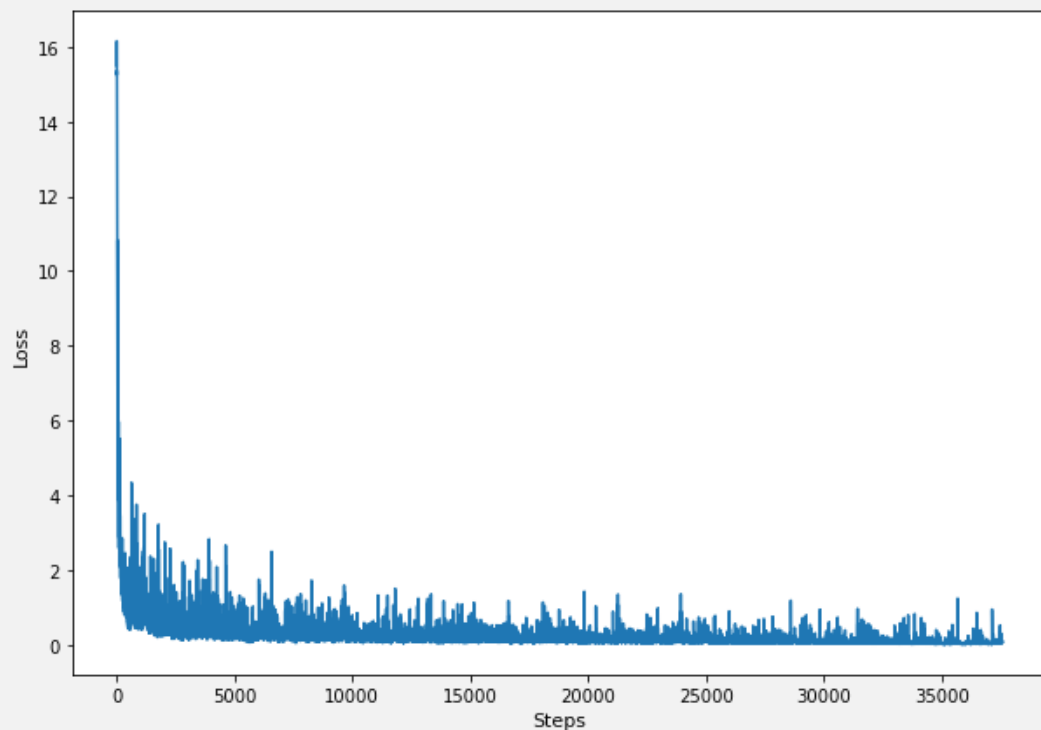
```
mem_size = 1000 # 設定記憶串列的大小
batch_size = 200 # 設定單一小批次(mini_batch)的大小
move_pos = [(-1,0),(1,0),(0,-1),(0,1)] # 移動方向 u,d,l,r 的實際移動向量

qval = model(state1) # 執行Q網路，輸出各動作的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)
    hit_wall = game.validateMove('Player', move_pos[action_]) == 1
    # 2.若有撞牆的動作，hit_wall就為True
    action = action_set[action_]
    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)/100.0
    state2 = torch.from_numpy(state2_).float()
    reward = -5 if hit_wall else game.reward() # 3.若撞牆回饋-5
    done = True if reward != -1 else False
```

3.4.1 學習的不穩定性



● 損失圖



結果

平均勝率高達97%
收斂過程變得穩定

3.5.1 Q-Learning 與 Deep Learning



Q-Learning的強化式學習演算法，這種算法只是一個數學式

在Q-Learning中，使用Q函數解決和控制任務(control task)有關的問題。

將狀態輸入到Q函數中，便能預測此狀態下所有動作能產生多少價值(預測價值, Q值)

Q函數能用不同的形式來呈現 ex.資料表或複雜的深度學習演算法

使用神經網路來扮演Q函數的角色，其變稱為Deep Q-Network(DQN)

『讓Q函數進行學習』對我們而言相當於『反向傳播訓練神經網路』

3.5.2 策略獨立與策略依賴



- Q-Learning是一種策略獨立(off-policy)演算法
- 策略：演算法為獲得最大回饋值所採用的對策。

以玩Gridworld 遊戲為例，其中一種策略是將所有可能路徑先畫出來，然後選擇最短的那一條；另一種策略則是隨機亂移動，直到剛好落在終點上為止。

策略獨立 Off-policy

模型是否可以準確預測Q值和
所採取的策略無關

策略依賴 On-policy

學習和策略密切相關
利用策略**選擇動作**並
依靠策略**收集學習**所需的資料

3.5.3 無模型與以模型為基礎的演算法



- 若模型已經**掌握了某個機制的運作方法**(包括：該機制由哪些元素構成以及這些元素之間的關聯性)，那麼該模型不僅能分析既有的資料，還能預測未來的結果。

無模型 model-free

未提供先驗(apriori)，即事先已知的相關知識)它是利用試誤(trial and error)的方式來學習

以模型為基礎 model-based

使用**領域知識**(domain knowledge)去建構模型

3.5.3 無模型與以模型為基礎的演算法

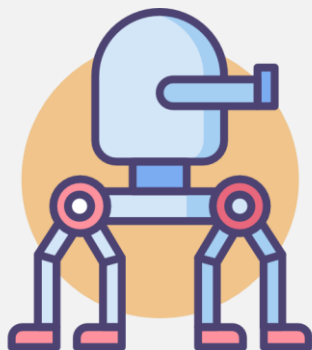


Ex. 機器人走路

無模型
model-free

先靠試誤來學習

掌握走路的技巧



以模型為基礎
model-based

針對環境建立模型

規劃從A到B的走法

