

國立虎尾科技大學  
機械設計工程系  
**cd2023 2a-pj1ag1 分組報告**

網際足球泡泡機器人場景設計

**Web-based bubbleRob Football  
Scene Design**

指導教授： 嚴 家 銘 老 師  
班 級： 四 設 二 甲  
學 生： 紀 閔 翔 (41023147)  
施 建 菖 (41023143)

中華民國 112 年 3 月

## 摘要

本課程將在設計簡單的移動機器人 bubbleRob 的同時，介紹相當多的 CoppeliaSim 功能。本教程相關的 CoppeliaSim 場景文件位於 `scenes/tutorials/BubbleRob`。

此專題是運用足球機器人，將其導入 CoppeliaSim 模擬環境並給予對應設置，將其機電系統簡化並運用 AI 進行訓練，找到適合此系統的演算法後，再到 CoppeliaSim 模擬環境中進行測試演算法在實際運用上的可行性。並嘗試透過架設伺服器將 CoppeliaSim 影像串流到網頁供使用者觀看或操控。

關鍵字: 類神經網路、強化學習、Chat GPT、CoppeliaSim、OpenAI

Gym

## Abstract

Due to the four major development trends of multidimensional arrays computing, automatic differentiation, open source development environment, and multi-core GPUs computing hardware. The rapid development of the AI field has been promoted. In view of this development, the physical mechatronic systems can gain machine learning efficiency through their simulated virtual system training process. And afterwards to apply the trained model into real mechatronic systems.

This project is to use the physical air hockey to play machine, introduce it into the CoppeliaSim simulation environment and give the corresponding settings, simplify its electromechanical system and use Open AI Gym for training, find an algorithm suitable for this system, and then perform it in the CoppeliaSim simulation environment Feasibility of testing algorithm in practical application. And try to stream CoppeliaSim images to web pages for users to watch or manipulate by setting up a server.

Keyword: nerual network 、 reinforcement learning 、 CoppeliaSim 、 OpenAI Gym

## 誌 謝

在此鄭重感謝製作以及協助本分組報告完成的所有人員，首先向大三學長致謝，他們不辭辛勞解決我們的提問，甚至從來沒有不耐煩，總是貼心為我們找出最佳解答。再來是我們的分組組長，他給了我們全方位的支援，提供我們解決問題的方向和建議，給予開始接觸網際對戰遊戲的我們有個學習的方向，開會時也時不時向我們提出建議以及未來走向，同時也給了我們能自由摸索的空間及時間，最後是由本分組組員同心協力才得以完成本報告，特此感謝。

## 目 錄

摘 要.....	i
Abstract .....	ii
誌 謝.....	iii
第一章 前言 .....	1
1.1 研究動機.....	1
1.2 製作過程.....	2
第二章 環境設定 .....	1
2.1 未來展望.....	1
2.2 規則說明.....	1
第三章 機器學習 .....	2
3.1 類神經網絡.....	2
3.1.1 啟動函數.....	4
3.1.2 損失函數.....	5
3.1.3 優化算法.....	9
3.2 強化學習.....	18
3.2.1 馬可夫決策 .....	21
3.3 Policy Gradient 理論 .....	25
3.3.1 Actor Critic .....	26

第四章 訓練環境 .....	28
4.1 OpenAI Gym.....	28
4.2 Pong.....	28
參考文獻 .....	29
附錄 .....	30

# 第一章 前言

## 1.1 研究動機

機器學習與各領域結合的應用越來越廣泛，在機電系統採用強化學習是為了讓機電系統的控制達到最佳化。本專題希望利用現代機器學習和模擬技術來建立一個虛擬的足球機系統作為訓練模型，將實體機器轉移到虛擬環境進行模擬，為了找到適合的訓練參數，因此將模型簡化後再進行測試各種參數的調整，透過不斷的訓練來得到一個優化過的對打系統，以下是成品圖。

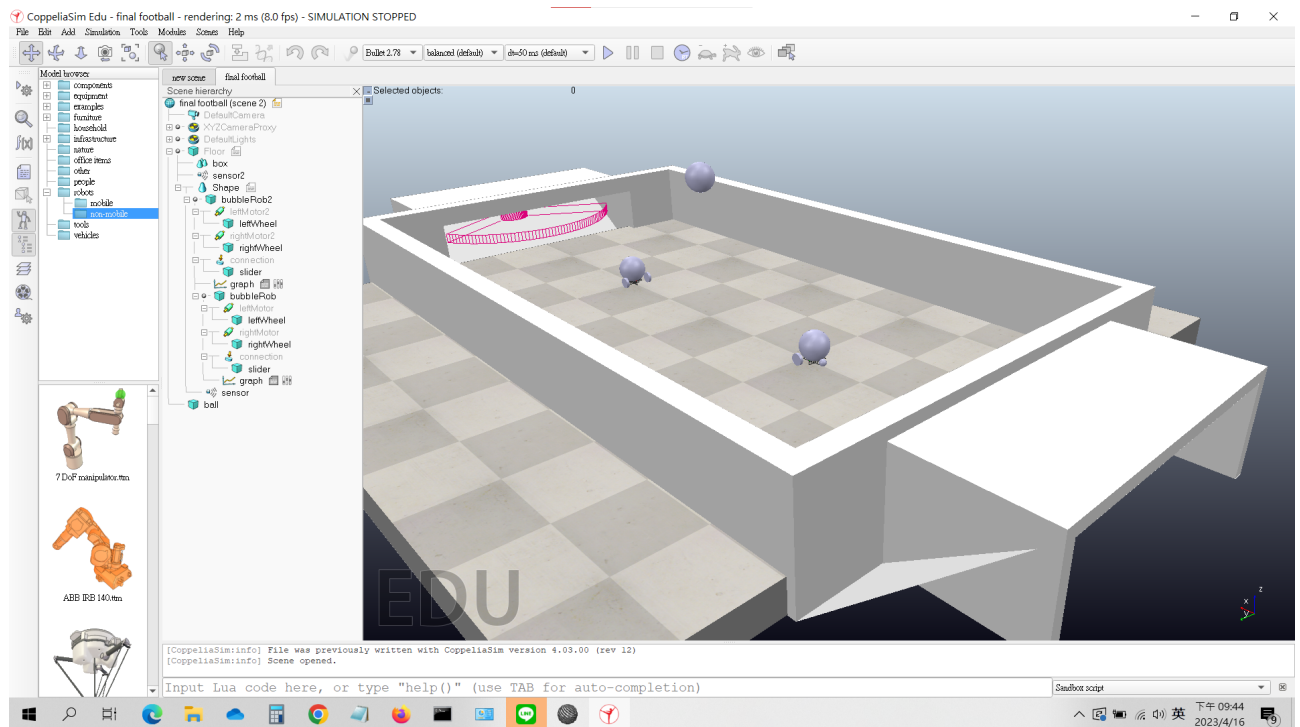


圖. 1.1: 模擬情形

## 1.2 製作過程

### 1. 先繪製球檯

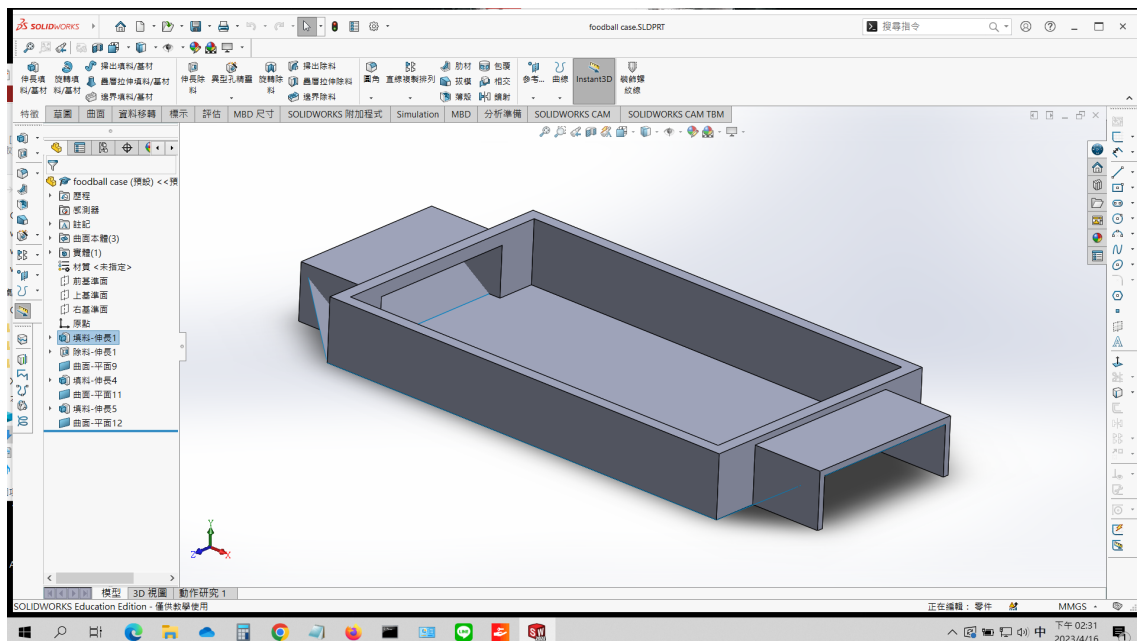


圖. 1.2: 球檯

2. 接著增加以 lua 腳本控制使 bubbleRob 可以前後左右移動  
sim.getObjectHandle 這個函式在 coppeliasim4.3.0 版本被淘汰了但還是可以使用，後來則改為 sim.getObject 去做使用在後面感測器腳本已做改善，並使復健名稱相互對應則可執行。



```

1 function sysCall_init()
2     score1 = 0
3
4     sensor = sim.getObject('/./sensor')
5     xml = [[
6         <ui closeable="false" resizable="false" title="Scoreboard">
7             <label id="10" style="*" {background-color: #808080; color: #000000; font-size: 40px; font-weight: bold; padding:
8             <label id="30" style="*" {background-color: #FFF; color: #000000; font-size: 40px; font-weight: bold; padding: 5;
9
10        </ui>
11    ]]
12    ui = simUI.create(xml)
13    simUI.setPosition(ui, 0,0, true)
14    bubbleRob = sim.getObject('/./bubbleRob')
15    ball = sim.getObject('/./ball')
16    bubbleRob2 = sim.getObject('/./bubbleRob2')
17    initialPosition = sim.getObjectPosition(bubbleRob, -1)
18    initialOrientation = sim.getObjectOrientation(bubbleRob, -1)
19    initialPosition2 = sim.getObjectPosition(bubbleRob2, -1)
20    initialOrientation2 = sim.getObjectOrientation(bubbleRob2, -1)
21    initialballPosition = sim.getObjectPosition(ball, -1)
22    initialballOrientation = sim.getObjectOrientation(ball, -1)
23
24 end
25
26 function sysCall_actuation()
27     --simUI.setLabelText(ui, 30, tostring(sim.getFloatSignal("myVariable")))
28     result=sim.readProximitySensor(sensor)
29     if(score1<5)then
30         if(result>0)then
31             score2 = score1+1
32             simUI.setLabelText(ui, 30, tostring(score2))
33
34             sim.setObjectPosition(bubbleRob, -1, initialPosition)
35             sim.setObjectOrientation(bubbleRob, -1, initialOrientation)
36             sim.setObjectPosition(bubbleRob2, -1, initialPosition2)
37             sim.setObjectOrientation(bubbleRob2, -1, initialOrientation2)
38             sim.setObjectPosition(ball, -1, initialballPosition)
39             sim.setObjectOrientation(ball, -1, initialballOrientation)
40             score1=score2
41         end
42     else
43         sim.pauseSimulation()
44     end
45 end
46 end
47 end

```

圖. 1.3: 感測器 Lua

### 3. 加入記分板並進行連線

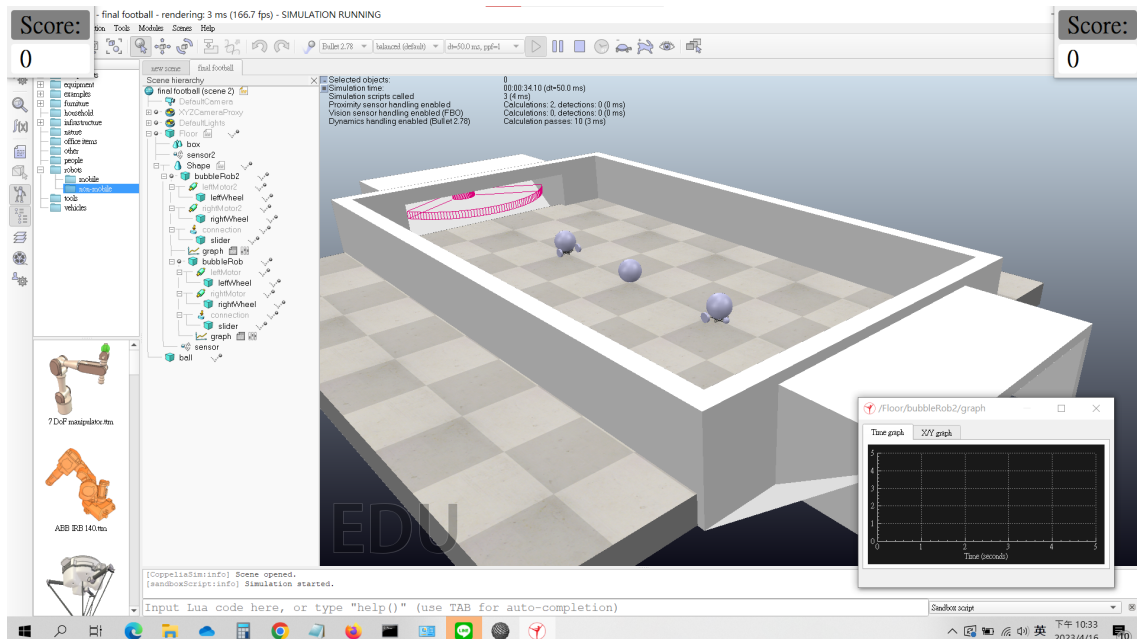


圖. 1.4: 模擬情形

## 第二章 環境設定

### 2.1 未來展望

此專題希望能利用現有完成的機械學習的算法，能發展成虛擬訓練，再將訓練完的機器學習應用到虛擬環境或是實體機電系統，並透過伺服器將影像串流提供玩家網頁介面進行遠端操控，同時提供多人觀看及時的比賽影像，將整個冰球機的控制和使用者的間有更完善串聯，機電系統的部分達到最優化控制和虛實整合的應用。

### 2.2 規則說明

Pong game 的遊戲規則簡單，透過擊錘將球打入對方球門即得一分，只要其中一方得 21 分就結束該局。擊錘只能沿單方向來回移動來進行防守和進攻。

遊戲規則如下：

1. 球打入敵方即得一分。
2. 擊錘只單一方向移動。
3. 最快贏得 21 分者獲勝，並結束該局遊戲。

## 第三章 機器學習

### 3.1 類神經網絡

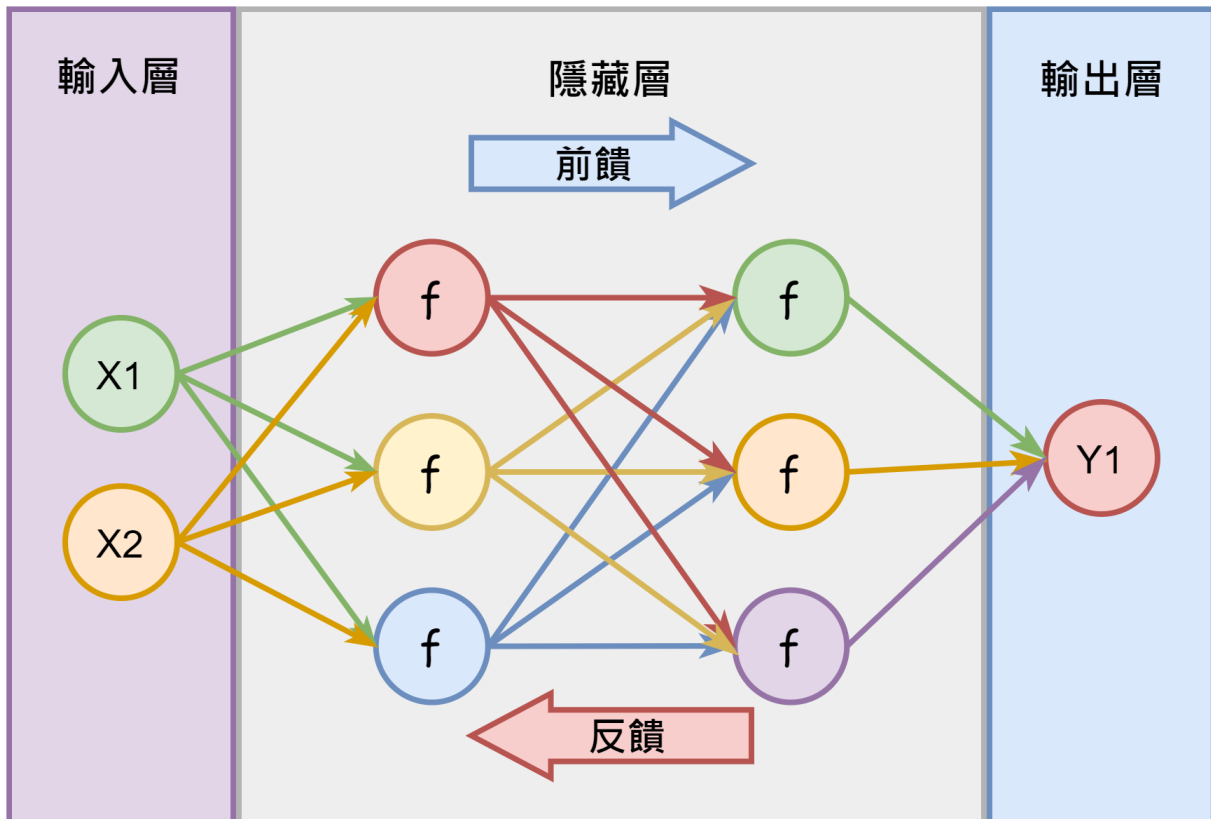


圖. 3.1: 類神經網路架構

典型的類神經網路架構，如 (圖.3.1) 所示每一層的每一個神經元都會連接到下一層全部的神經元，對於每個神經元輸出會有不同的權重。

神經元是 AI 系統中使用的數學模型，其行為與實際的大腦神經元運作方式相仿，模型以數字的方式表達，神經元間傳遞會有不同強度，而以數值的大小代表不同強度，這個數值我們稱為權重，對結果產生重要的影響。

(圖.3.1) 基礎的類神經網路架構主要由輸入層、隱藏層和輸出層這三部分組成，實際運用上還有更多樣更複雜的類神經網路架構，深度學

習則是有更多的隱藏層，從意義上來說就是增加了類神經網路的深度。

此外，如(圖.3.1)所示，資料由輸入層傳入，經過隱藏層運算和記憶，再由輸出層進行輸出，這種資料被傳遞的方式被稱為前饋輸入(feed-forward)。

類神經網路架構有了記憶，就能進一步讓網路學習。當類神經網路接收資料並猜測答案，如果答案與實際答案不符、有落差或有錯誤的情況，它會回授並修改對每個神經元權重和偏差修正的程度，並嘗試調配各項數值來修正輸出的結果，讓結果的正確性提高，這樣的修正行為就被稱為反向傳播(back-propagation)。透過迭代方法進行反複試驗，模擬人們學習的行為，而每一次的迭代被稱為 epoch，經過一定的迭帶次數後會透過反向傳播修正輸出的誤差，經過不斷執行的修正，最終類神經網路的學習會不斷進步並給出更好的答案，訓練時間長短取決於訓練項目的複雜程度。

可以看到該神經網路的輸出僅取決於互連的權重，還取決於神經元本身的偏差，雖然權重會影響啟動函數曲線的陡度，但是偏差會將發生變化的整個曲線，向右或向左，權重和偏差的選擇，決定了單個神經元的預測強度，而訓練類神經網路使用的輸入數據可以來微調權重和偏差。(圖.3.2)

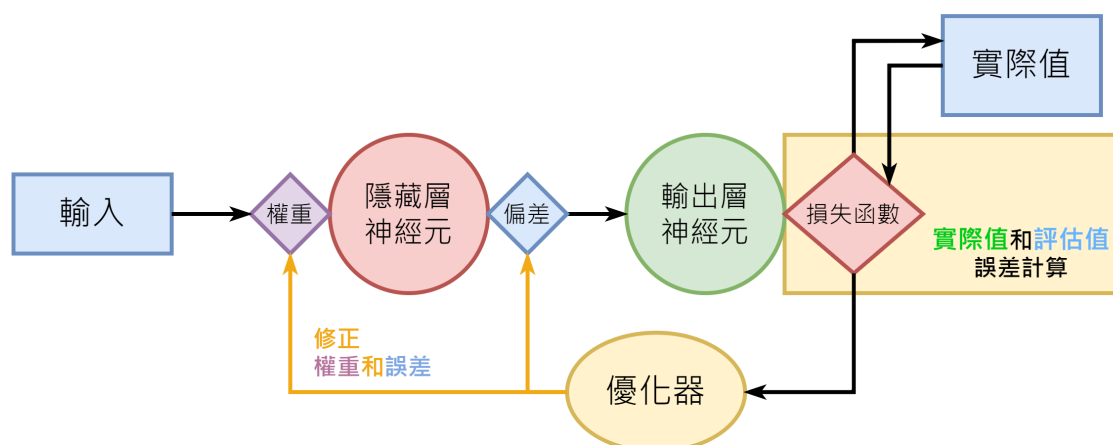


圖. 3.2: 類神經網路關係

### 3.1.1 啟動函數

啟動函數是設計類神經網路的關鍵部分，如果不使用啟動函數，神經元的計算只會有線性組合，這樣的類神經網路缺乏活性而且記憶性差；啟動函數能讓神經元計算呈現非線性，讓類神經網路因為計算的非線性而提高整個網路的活性和記憶性。

以下介紹幾種較為常見的啟動函數及其特性：

- Sigmoid Function(圖.3.3)：

輸出介於 0 到 1 之間，適用於二元分類，方程式具有非線性、可連續微分、且具有固定輸出範圍等特性，並可以讓類神經網路呈現非線性。

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

SigmoidPrime Function(圖.3.3，紅色的)是從 Sigmoid Function(圖.3.3，藍色)微分得來，以梯度運算的方式，可以減少梯度誤差，但也是造成梯度消失的主要原因，若要改善梯度消失需要搭配優化器使用，方程式如下：

$$\sigma'(x) = \sigma(x)[1 - \sigma(x)]$$

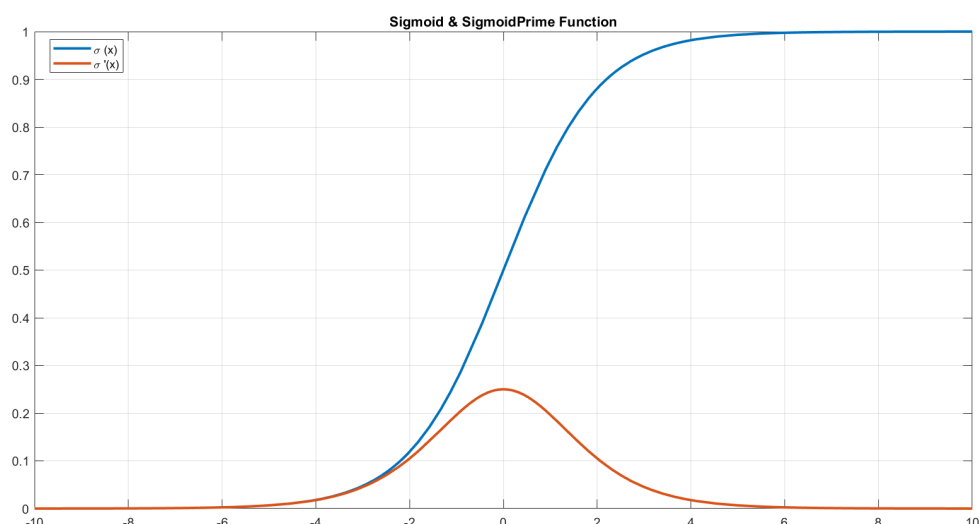


圖. 3.3: SigmoidFunction

- Softmax :

Softmax 會計算每個事件分布的機率，適用多項目分類、其機率總合為 1。以此專案為例，假設擊錘移動有向上移動、向下移動及不移動這三個決策選項，則這三個決策機率值總和為 1。

$$S(x) = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_i}}$$

- ReLU Function :

ReLU Function 方程式特性：若輸入值為負值，輸出值為 0；若輸入值為正值，輸出則維持該輸入數值。ReLU 計算方式簡單、收斂速度快，這是類神經網路最普遍拿來使用的啟動函數，因為可以解決梯度消散的問題，但須注意：起始值若設定到不易被激活範圍或是權重過渡所導致權重梯度為 0 就會造成神經元難以被激活。

$$f(x) = \max(0, x)$$

$$if, x < 0, f(x) = 0$$

$$else f(x) = x$$

### 3.1.2 損失函數

損失函數是類神經網路的另一個重要的部分，損失函數會將類神經網路的結果與期望結果進行比較，且必須重複估算模型當前狀態的誤差；該函數可用於估計模型的損失，以便可以更新權重減少下次評估時的損失，下一節會有詳細說明，以下簡述幾種優化的方法：

- Gradient Descent

利用梯度的方式尋找最小值的位置，其特色可找到凸面 error surface 的絕對最小值，在非凸面 error surface 上找到相對最小值。其缺點是在非凸面 error surface 要避免被困在次優的局部最小值。

- Batch gradient descent

用批次的方式計算訓練資料，整個資料集計算梯度只更新一次，因此計算和更新時會占用大量記憶體。整體效率較差、速度較緩慢。由 Gradient Descent 延伸出來的算法。其收斂行為與 Gradient Descent 相同。

- Stochastic gradient descent

每次執行時會更新並消除誤差，有頻繁更新和變化大的特性，較不容易困在特定區域。由 Gradient Descent 延伸出來的算法。其收斂行為與 Gradient Descent 相同。

- Mini-batch gradient descent

結合 Batch gradient descent 和 Stochastic gradient descent 的特點：批量計算和頻繁更新，所衍伸的算法。利用小批量的方式頻繁更新，並使收斂更穩定。其缺點：學習率挑選不易、預定義 threshold 無法適應數據集的特徵、對很少發生的特徵無法執行較大的更新、非凸面 error surface 要避免被困在次優的局部最小值等。

- Gradient descent optimization algorithms

為了改善前面幾種算法而發展出來的優化算法。以下將列出數種優化算法。

- Momentum

在梯度下降法加上動量的概念，會加速收斂到最小值並減少震盪。

- Nesterov accelerated gradient

NAG，有感知能力的 Momentum：在坡度變陡時減速，避免衝過最小值所造成的震盪（為了修正到最小值，來回修正而產生的震盪）。

- Adagrad

其學習率能適應參數：頻繁出現的特徵用較低的學習率，不經常出現的特徵則用較高的學習率，且無須手動調整學習率。其缺點是，學習率會急遽下降，最後會無限小，這算法就不再獲得知識。

- Adadelata

為 Adagrad 的延伸，下降激進程度，學習率從更新規則中淘汰，不需設定預設學習率。

- RMSprop

為了解決 Adagrad 學習率急劇下降的問題，學習率除以梯度平方的 RMS，解決學習率無限小的情形。

- Adam Function

結合了 Adagrad 和 RMSprop 的優勢，有論文表示，在訓練速度方面有巨大性的提升，但在某些情況下，Adam 實際上會找到比隨機梯度下降法更差的解決方法。以下是計算過程：

$$g_t = \delta_{\theta} f(\theta)$$

一次矩指數移動均線：

$$m_t = \beta(m_{t-1}) + (1 - \beta_1)(\nabla w_t)$$



$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

二次矩指數移動均線:

$$v_t = \beta_2(v_t - 1) + (1 - \beta_2)(\nabla w_t)^2$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

因此,Adam Function:

$$\omega_{t-1} = \omega_t - \frac{\eta}{\sqrt{\hat{v}_t - \epsilon}} \hat{m}_t$$

- AdaMax

與 Adam 相似，依靠  $(u_t)$  最大運算。

- Nadam

結合 Adam 和 NAG，應用先前參數執行兩次更新，一次更新參數一次更新梯度。

- AMSGrad

改善 Adam 算法所導致收斂較差的情況(用指數平均會減少其影響)，換用梯度平方最大值來做計算，並移除去偏差的步驟。是否有比 Adam 算法好仍有待觀察。

- Gradient noise

有助於訓練特別深且複雜的網絡，noise 可改善不良初始化的網路。

- Mean Squared Error

他能告訴你一組點與回歸線接近的程度，透過獲取點與回歸線之距離(這些距離就是誤差)並對它們進行平方來做到這點，而平方是為了消除所有負號，也能讓更大的差異賦予更大的權重。

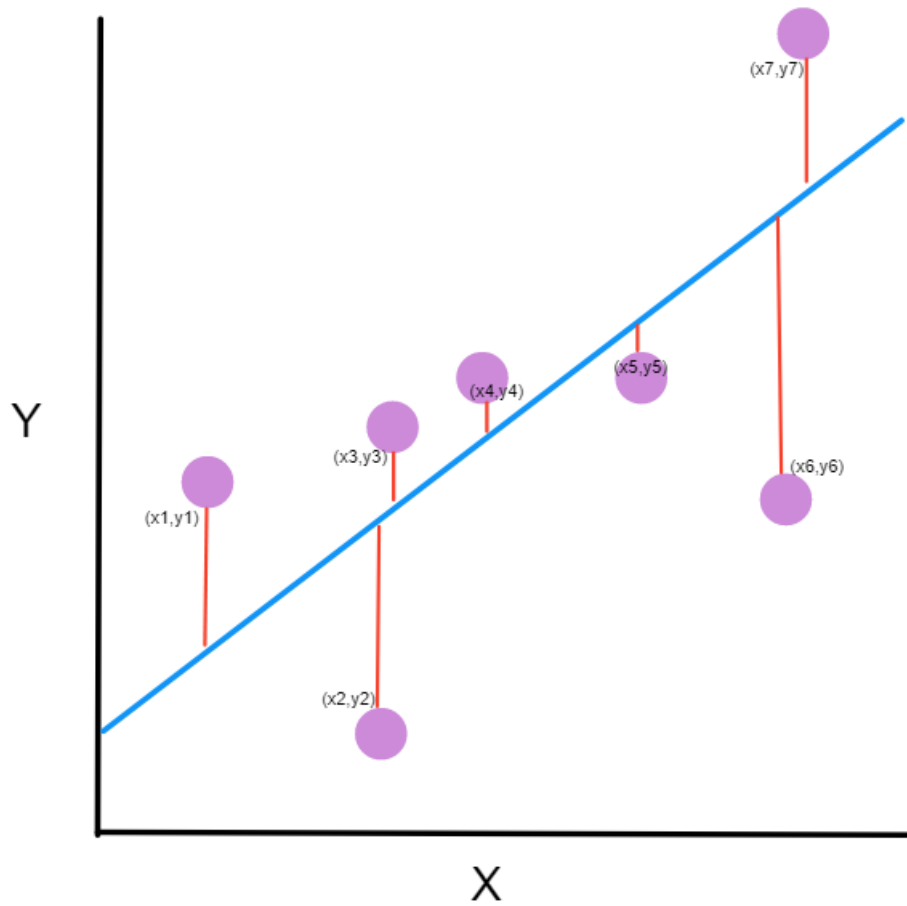


圖. 3.4: 線性回歸

回歸線: 數據點間最小距離的一條線。

$n$ : 數據點的數量

$y_i$ : 觀測值

$\hat{y}_i$ : 預測值

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y}_i)^2$$

### 3.1.3 優化算法

- Gradient Descent Optimizer[11]

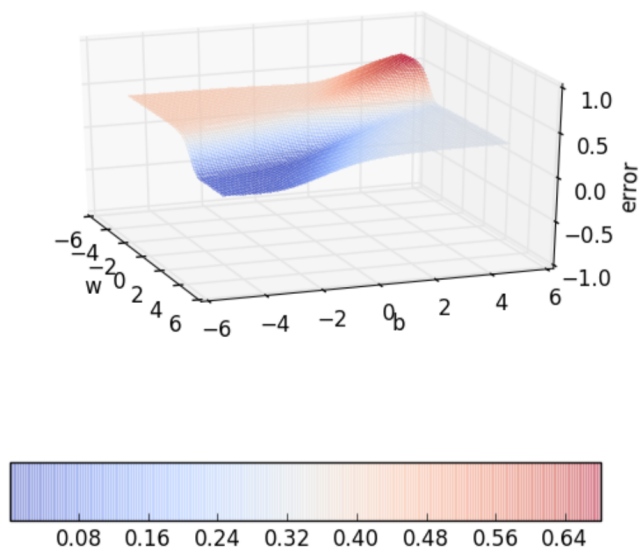


圖. 3.5: Error surface [11]

藉由梯度下降將目標函數值最小化，目標函數以 loss function  $L(\theta)$  為例， $\theta$  為 weight( $w$ ) 和 bias( $b$ ) 的向量函數，為了找到 (圖.3.5) 上的最小值，因此加上  $\Delta\theta$  將  $\theta$  的方向修正並引導到正確方向，避免每次修正的過多導致錯過最小值，利用係數  $\eta$ (學習率) 縮放  $\Delta\theta$  的修正量 (圖.3.6)，修正後方程式為：

$$\theta = \theta + \eta \cdot \Delta\theta$$

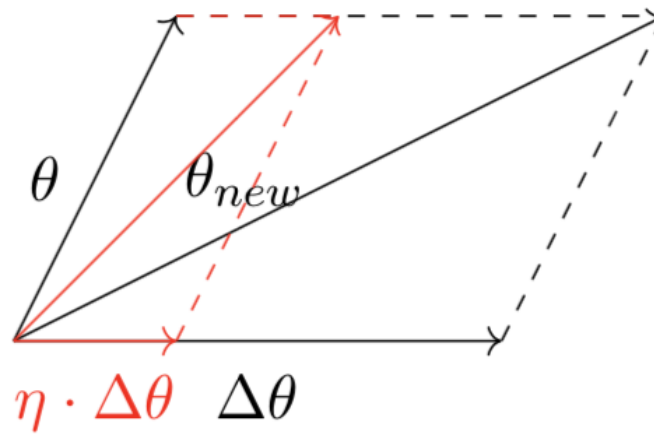


圖. 3.6: theta vector[11]

將  $\theta$  以泰勒展開式表示，假設並  $\Delta\theta$  為  $u$ :

$$L_{(\theta+\eta u)} = L_{(\theta)} + \eta u^T \cdot \nabla_{\theta} L_{(\theta)} + \frac{\eta^2}{2!} u^T \cdot \nabla^2 L_{(\theta)} u + \frac{\eta^3}{3!} \dots + \frac{\eta^4}{4!} \dots + \frac{\eta^n}{n!} \dots$$

以泰勒展開式的型式表示的好處是： $\theta$  些微的更動產生新值。 $\eta$  值通常小於一，當  $\eta^2 \ll 1$ ，因此可以忽略高階項

$$L_{(\theta+\eta u)} = L_{(\theta)} + \eta u^T \cdot \nabla_{\theta} L_{(\theta)} [\eta \text{ is typically small, so } \eta^2, \eta^3, \dots \rightarrow 0]$$

新的  $L(\theta + \eta u)$  輸出的值會小於  $L(\theta)$   $L(\theta + \eta u) - L(\theta) < 0$ ，同理可證  $u^T \cdot \nabla_{\theta} L(\theta)$ ，符合  $u$  這條件：當新的值小於舊的值， $u$  就是一個好的值。假設  $u$  和  $\nabla_{\theta} L(\theta)$  的夾角為  $\beta$

$$\cos(\beta) = \frac{u^T \cdot \nabla_{\theta} L_{(\theta)}}{|u^T| |\nabla_{\theta} L_{(\theta)}|}$$

因為  $\cos(\theta)$  的值介於 1 和 -1 之間

$$\begin{aligned} -1 < \cos(\beta) &= \frac{u^T \cdot \nabla_{\theta} L_{(\theta)}}{|u^T| |\nabla_{\theta} L_{(\theta)}|} \leq 1 \\ k &= |u^T| |\nabla_{\theta} L_{(\theta)}| \\ -k &\leq k \cos(\beta) = u^T \cdot \nabla_{\theta} L_{(\theta)} \leq k \end{aligned}$$

所以盡可能的讓新值小於舊值 ( $L(\theta + \eta u) - L(\theta) < 0$ )，loss 值就會減少得越多。因此  $u^T \cdot \nabla_{\theta} L(\theta)$  應該為負，在這情況下  $\cos(\beta)$  於  $-1$ ， $\beta$  的角度為  $180^\circ$  這就是  $\theta$  移動的方向與梯度方向相反的原因。梯度下降法告訴我們：當  $\theta$  在特定值，並想減少新的  $\theta$  值，使 loss 值逐漸減少就應該與梯度相反的方向找 (若梯度為正值，找最小值就需往負的方向找)：

$$\begin{aligned} w_{t+1} &= w_t - \eta \nabla w_t \\ b_{t+1} &= b_t - \eta \nabla b_t \\ \text{where at } w &= w_t, b = b_t \\ \left\{ \begin{aligned} \nabla w_t &= \frac{\partial L(\theta)}{\partial w} \\ \nabla b_t &= \frac{\partial L(\theta)}{\partial b} \end{aligned} \right. \end{aligned}$$

- Batch gradient descent [12]

Vanilla gradient descent 又稱 Batch gradient descent (批次梯度下降法)，計算目標函數的梯度，參數  $\theta$  對於整個訓練資料：

$$\theta = \theta - \eta \cdot \nabla_{\theta} L(\theta)$$

目標函數以為例 loss function  $L(\theta)$ ，參數  $\theta$  為 weight( $w$ ) 和 bias( $b$ ) 的函數， $\eta$  為學習率。由於計算整個資料集計算梯度只更新一次，Batch gradient descent 可能非常慢並且對於資料集無法符合及記憶體來說棘手 (一次需要儲存整個資料集的資料，當更新和計算時會占用大量記憶體)。

### 程式. 3.1: Batch gradient descent

---

```
for i in range(nb_epochs):
    params_grad = evaluate_gradient (loss_function, data, params)
    params = params - learning_rate * params_grad
```

---

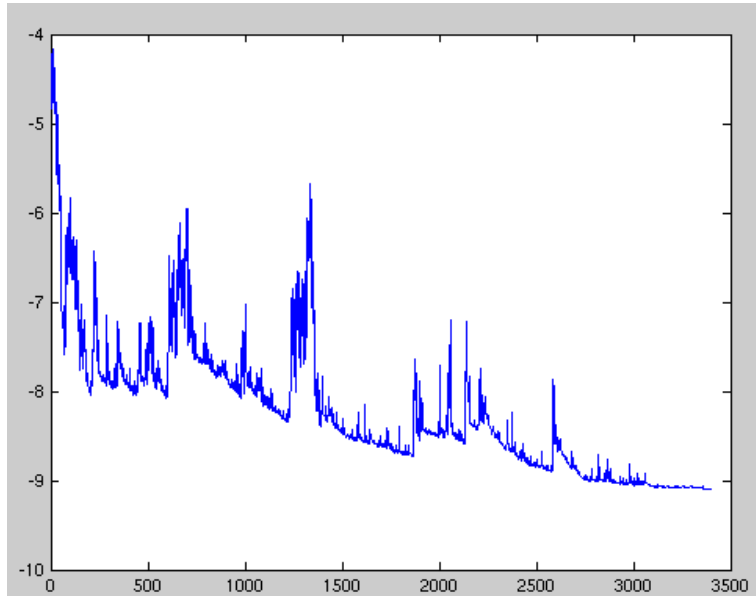


圖. 3.7: SGD fluctuation[12]

預定義每次 epoch，先計算 loss function 梯度向量對於整個資料集參數向量。如果梯度值來自於先前計算出的梯度值，就會檢查梯度，並以梯度相反的方向更新參數  $\theta$ ，學習率  $\eta$  決定多大的更新量。Batch gradient descent 對於凸面誤差可以保證收斂到廣域最小值，對於非面凸誤差可以收斂到局部最小值。

- Stochastic gradient descent(SGD)[12]

隨機梯度下降法，這裡的目標函數為  $J(\theta, x^i, y^i)$ (變數  $\theta$  為 w(weight) 和 b(bias) 的函數，也可以寫成  $J(w, b, \theta, x^i, y^i)$ )。

$$\theta = \theta - \eta \cdot \nabla_{\theta} J_{(\theta, x^i, y^i)}$$

批量梯度下降他會在每個參數更新前重新計算相似梯度。SGD 每次執行會更新來消除多餘(誤差)，因此通常速度很快。SGD 頻繁更新並變化很大，因為目標方程式波動很大(圖.3.7)。

SGD 的方程式一方面會跳到新的值和潛在局部最小值，另一方面 SGD 會持續超調(誤差超過預期)最後收斂到廣域最小值。無論如何

他被顯示當學習率下降緩慢，SGD 顯示與 Batch gradient descent 同樣收斂行為，幾乎可以肯定地，對於凸面或非凸面優化，會收斂到絕對或是局部最小值。這程式碼片段 [程式.2.2] 在訓練樣本上加入一個迴圈來對每個樣本評估梯度。每個 epoch(訓練循環) 會打亂訓練數據。

程式. 3.2: Stochastic gradient descent

---

```
for i in range(nb_epochs) :  
    np.random.shuffle(data)  
    for example in data :  
        params_grad = evaluate_gradient (loss_function , example , params)  
        params = params - learning_rate * params_grad
```

---

- Mini-batch gradient descent[12]

Mini-batch gradient descent(小批量梯度下降) 各取前兩者的優點，將資料集分割成小區塊，每個小區塊大小稱作 batch size，每次跑完 batch size 算迭代(iteration) 一次，算完一次資料集即完成一次 epoch。舉例: 資料集大小為 1000，若 batch size 為 50，iteration 為 datasets 的  $\text{batch size} = 1000 \div 50 = 20$ ，當 iteration 跑完 20 次算完成一次 epoch。這方式可以減少參數更新的方差，並且可以穩定收斂；可利用深度學習庫所共有的高度優化的矩陣優化，從而由一個小批量計算出梯度非常有效。通常 batch sizes 的範圍介於 50 256，會因為應用而有所差異。訓練神經網絡時，通常選擇 Mini-batch gradient descent 算法，而當使用這算法時，通常也用 SGD 稱呼。

$$\theta = \theta - \eta \cdot \nabla_{\theta} J_{(\theta, x^{(i:i+n)}, y^{(i:i+n)})}$$

下面 [程式.2.3] 為迭代範例，batch size 大小為 50：

程式. 3.3: Mini-batch gradient descent

---

```
for i in range(nb_epochs) :  
    np.random.shuffle(data)  
    for batch in get_batches (data , batch_size = 50):  
        params_grad = evaluate_gradient (loss_function , batch , params)  
        params = params - learning_rate * params_grad
```

---

Mini-batch gradient descent 無論如何還是無法確保收斂的很好，存在一些需要解決的挑戰：

- 1 選擇適當的學習率是有難度的。如果學習率太小會導致收斂困難或緩慢，學習率太大則會阻礙收斂導致 loss function 來回波動或發生偏離。
  - 2 學習率清單嘗試在訓練的時候調整學習率，即根據預定義清單或當目標下降於閾值 (threshold) 時降低學習率。但清單和閾值須預先定義，因此無法適應數據集的特徵。
  - 3 另外相同學習率適用全部參數更新。如果資料稀疏而且外型有很特別的頻率，我們可能不希望將所有特徵更新到相同的程度，而是對很少發生的特徵執行較大的更新。
  - 4 最小化神經網路常見的高度非凸面誤差方程式 (error function) 的另一關鍵挑戰則是要避免被困在大量次優的局部最小值區域中。認為困難實際上不是由局部最小值引起的，而是由鞍點引起的，即一維向上傾斜而另一維向下傾斜的點。這些鞍點通常被相同誤差的平穩段包圍，這使得 SGD 很難逃脫，因為在所有維度上梯度都接近於零。
- Gradient descent optimization algorithms[12]
- SGD 難以及在陡峭的往正確的方向，那就是說在一個維度上，曲面的彎曲比另一個維度要陡得多，這在局部最優情況下很常見。下圖 (圖.1) 的同心圓代表中心下凹的曲面。在這些情況下，SGD 會在陡峭的地方振盪，而僅沿著底部朝著局部最優方向猶豫前進，如 (圖.3.9) 所示。Momentum(動量) 是一個幫助加速 SGD 在正確方向和抑制震盪的方法，在 (圖.3.10)。

這麼做會增加一個係數  $\gamma$  來更新上次的向量到正確向量 (修正偏差)， $\gamma$  通常設為 0.9 左右。





圖. 3.9: SGD without momentum[12]

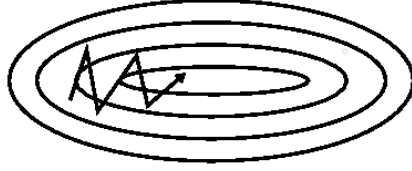


圖. 3.10: SGD with momentum[12]

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_{\theta} J_{(\theta)} \quad \theta = \theta - v_t$$

實際上，使用動量的時候，就像將球推下山坡。球在下坡時滾動時會累積動量，在途中速度會越來越快（如果存在空氣阻力，直到達到極限速度，也就是  $\gamma < 1$ ）參數更新也發生了同樣的事情：動量 (momentum) 對於梯度指向相同方向的維度增加，而對於梯度改變方向的維減少動量。結果，我們獲得了更快的收斂並減少了振盪。

Nesterov accelerated gradient (NAG) 是一種使動量具有一個去向的概念，以便在山坡再次變高之前知道它會減速。我們知道使用動量  $\gamma v_{t-1}$  來移動參數。計算  $\theta - \gamma v_{t-1}$  這樣就給了參數的下一個位置的近似值（完整更新缺少的梯度），這是參數將要存在的大致概念。現在，通過計算與當前參數無關的梯度來有效地看到目前的參數  $\theta$  將會移動到的位置：

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_{\theta} J_{(\theta - \gamma v_{t-1})} \quad \theta = \theta - v_t$$

同樣，我們設置動量  $\gamma$  約為 0.9。動量首先計算當前梯度（(圖.3.10) 中的藍色小向量），然後在更新的累積梯度（藍色向量）的方向上發生較大的跳躍，而 NAG 首先在先前的累積梯度的方向上進行較大的

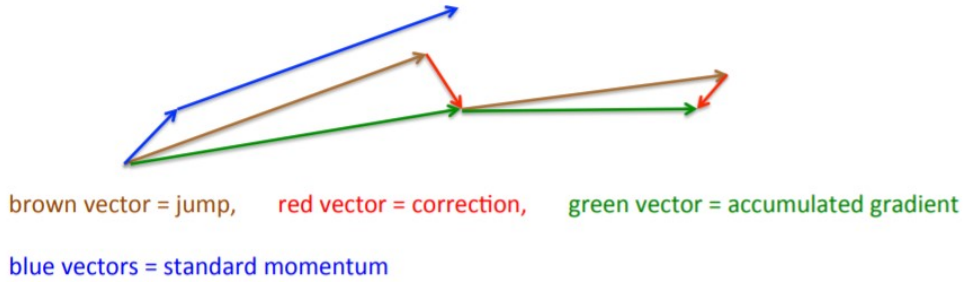


圖. 3.10: NAG[12]

跳躍（棕色向量），測量梯度，然後進行校正（紅色向量），從而完成 NAG 更新（綠色向量）。這種預期的更新可防止我們過快地進行，並導致響應速度增加，從而顯著提高了 RNN 在許多任務上的性能。

Adagrad[12]：

Adagrad 是一個梯度優化的算法，它可以做到：學習率適應參數，對於頻繁出現的特徵相關參數執行較小的更新（較低的學習率），以及對不經常出現的特徵相關參數進行較大更新（即學習率較高）。Adagrad 可以提高 SGD 的強度，用於訓練大型神經網絡。

先前，在同一次  $\theta$  參數（更新後就算另一次），每個  $\theta$  都使用相同的  $\eta$ （學習率）。Adagrad 則是對每個  $\theta$  參數使用不同的  $\eta$ ， $t$  代表 time step。先將 Adagrad 的更新參數向量化。用  $g_t$  表示目標函數（參數  $\theta$  在 time step  $t$ ）對參數做偏微分計算。

$$g_{t,i} = \nabla_{\theta} J_{(\theta_{t,i})}$$

當 SGD 更新每個參數  $\theta_i$ ，在每個 time step  $t$ ，因此變成：

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

更新規則，Adagrad 根據先前  $\theta_i$  計算的梯度，對每個參數  $\theta_i$  修改整個學習率  $\eta$  在每個 time step：

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}}$$

$G_t \in \mathbb{R}^{d \times d}$  這是一個對角矩陣每個對角元素  $i$ ， $i$  是關於  $\theta$  梯度平方和取決於 time step， $\epsilon$  是避免分母為 0（ $\epsilon$  通常為  $10^{-8}$ ），如果沒

有平方根運算，該算法的性能將大大降低。 $G_t$  包含了過去梯度平方根，由於全部  $\theta$  參數沿著對角線，通過向量的內積計算  $G_t$  和  $g_t$ :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$$

Adagrad 主要好處之一是，無需手動調整學習率。大多數實現使用預設值 0.01 並將其保留為預設值。Adagrad 主要弱點是會累積分母的平方梯度：由於每項都是正的，累積和會在訓練中不斷增長。反過來，學習率下降，並最終變得無限小，這算法就不再獲得知識。

Adadelat[12]：

Adadelat 是 Adagrad 的延伸，下降其激進的程度，單調的降低學習率。Adadelat 會限制過去累積的梯度，並將其限制在某個特定大小  $w$ ，並代替 Adagrad 過去累積的梯度平方，以梯度總和是遞迴定義為所有過去衰減梯度平方平均值。流動平均  $E[g^2]_t$  在 time step  $t$  然後取決於 (像 Momentum 的  $\gamma$ ) 先前平均和最近梯度：

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2$$

$\gamma$  值和 Momentum 的相似，約為 0.9，現在根據參數更新向量  $\Delta\theta_t$  來重寫 SGD：

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Adagrad 的參數更新向量替換成：對角矩陣  $G_t$  過去梯度平方的衰退平均  $E[g^2]_t$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$$

$$\text{replace } G_t \text{ with } E[g^2]_t \Rightarrow \Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$$

由於分母只是梯度的均方根 (RMS)，我們可以取代成縮寫：

$$\Delta\theta_t = -\frac{\eta}{\text{RMS}[g]_t} \cdot g_t$$

這個更新單位和 SGD、Momentum 以及 Adagrad 的單位不符合，因此更新需有相同的參數。為了實現這一點，首先定義另一個指數衰減平均值，這次不是梯度平方更新而是參數平方更新：

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma) \Delta\theta_t^2$$

RMS 參數更新：

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

$RMS[\Delta\theta]_t$  是未知的，更新參數的 RMS 取近似值到上個 time step。用  $RMS[\Delta\theta]_t$  取代學習率  $\eta$ ，最後產生新的規則：

$$\begin{aligned}\Delta\theta_t &= -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t\end{aligned}$$

使用 Adadelta，甚至不需要設定預設學習率，因為它已從更新規則淘汰。

RMSprop[12]：

RMSprop 是 Geoffrey Hinton 在他的課程中提出的未公開自適應學習率的方法。

RMSprop 和 Adadelta 都是為了解決 Adagrad 的學習率急劇下降的問題個別獨立開發出來的解決方式。RMSprop 實際上與 Adadelta 得出的第一個更新向量相同：

$$\begin{aligned}E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t\end{aligned}$$

RMSprop 也將學習率除以梯度平方的指數衰減平均值。Hinton 建議  $\gamma$  設為 0.9，好的預設學習率  $\eta$  數值為 0.001。

Adaptive Moment Estimation：[12]

Adaptive Moment Estimation 自適應矩評估 (Adam) 是另一種計算每個

評估學習率的方法。出了儲存過去梯度平方的指數衰減平均值  $v_t$ ，就像 Adadelta 和 RMSprop 一樣，Adam 還保留過去梯度的指數衰減平均值  $m_t$ ，類似動量 (Momentum)。如果 Momentum 被視為順著斜坡下滑的球，而 Adam 則是像一個帶有摩擦的沉重的球，因此更適合待在 error face 平坦的最小值區域。計算過去梯度平方的衰減平均值  $m_t$  和  $v_t$  分別如下：

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned}$$

$m_t$  和  $v_t$  分別是第一階矩平均估計值和第二階矩無中心方差估計值，因此是方法的名稱。像  $m_t$  和  $v_t$  被初始化為向量 0，Adam 的作者觀察到它們偏向零，特別是在初始 time step，尤其是在衰減率較小的時候 (也就是說  $\beta_1$  和  $\beta_2$  趨近於 1) 藉由計算校正偏差第一矩  $\hat{m}_t$  和第二矩  $\hat{v}_t$  抵消偏差：

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \end{aligned}$$

使用他們去更新參數，就像 Adadelta 和 RMSprop 中所看到的那樣，這將產生 Adam 更新規則：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

$\beta_1$  預設值建議為 0.9， $\beta_2$  預設值建議為 0.999， $\epsilon$  預設值建議為  $10^{-8}$ 。根據經驗證明 Adam 表現良好，並且與其他自適應學習算法相比具有優勢。在 Adam 更新規則中的  $v_t$  係數是與梯度成反比地縮放過去梯度的範數 (通過  $v_{t-1}$  項) 和當前梯度  $|g_t|^2$ ：

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) |g_t|^2$$

我們轉換這個更新到  $\ell_p$ 。注意  $\beta_2$  參數化為  $\beta_2^p$ ：

$$v_t = \beta_2^p v_{t-1} + (1 - \beta_2^p) |g_t|^p$$

大規範  $p$  值使數值上變得不穩定，這就是為什麼  $\ell_1$  和  $\ell_2$  規範在實踐中是最常見的。然而  $\ell_\infty$  通常也表現出穩定的行為。為了避免與 Adam 混用，所以使用  $u_t$  來表示無窮範數約束  $v_t$ ：

$$\begin{aligned} u_t &= \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty) |g_t|^\infty \\ &= \max(\beta_2 \cdot v_{t-1}, |g_t|) \end{aligned}$$

替換為 Adam 更新公式  $\sqrt{\hat{v}_t} + \epsilon$  和  $u_t$  得出 AdaMax 更新規則：

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t} \hat{m}_t$$

替換為 Adam 更新公式  $\sqrt{\hat{v}_t} + \epsilon$  和  $u_t$  得出 AdaMax 更新規則：

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t} \hat{m}_t$$

注意  $u_t$  依靠最大運算，不建議 Adam 中的  $m_t$  和  $v_t$  偏向零，這就是為什麼不需要針對  $u_t$  計算偏差。好的預設值  $\eta = 0.002$   $\beta_1 = 0.9$  和  $\beta_2 = 0.999$ 。

## 3.2 強化學習

強化學習 (Reinforcement Learning，簡稱為 RL) 是通過 agent(代理) 與已知或未知的環境持續互動，不斷適應與學習，會得到正向或負面的回饋，對應到獎賞 (reward) 和懲罰 (punishments)。考慮到 agent 與環境 (environment) 互動，進而決定要執行哪個動作，強化學習的學習模式是建立在獎賞與懲罰上。

強化學習與其他學習法不一樣的地方在於：不需要事先收集大量數據提供當作學習樣本，而是透過與環境互動，在環境下發生的狀態當作學習的資料來源，透過不斷嘗試使所得到的獎勵最大化。其他類型的機器學習大都需要給予特定資料且有明確的答案。

由於強化學習是建立在 agent 與環境互動上，因此許多參數進行運算，需要大量資訊來學習，並根據資訊採取行動。強化學習的環境可以是真實世界、2D 或 3D 模擬世界的場景。強化學習的範圍很廣，因為環境的規模可能很大，且在環境中有多相關因素，影響著彼此。強化學習以獎勵的方式，促使學習結果趨近或達到目標結果。

強化學習涵蓋範圍 (圖.3.11)：

強化學習可以運用在計算機科學、神經科學、心理學、經濟學、數學、工程等領域，涵蓋領域相當廣泛。

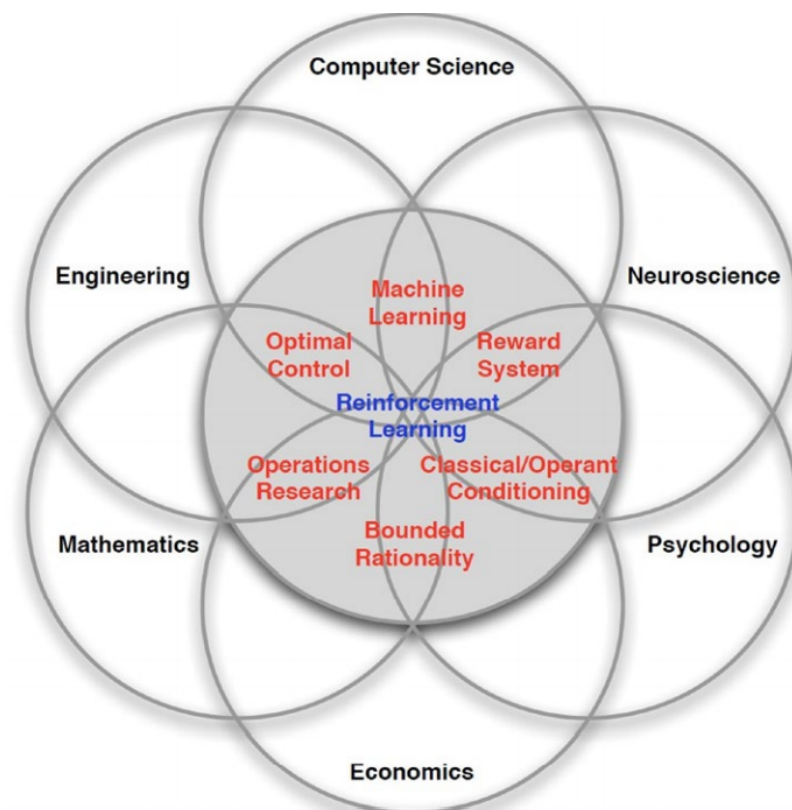


圖. 3.11: 各領域與機器學習應用範圍

強化學習的流程：

透過 agent 與環境間互動而產生狀態和獎勵，由於狀態的轉移，agent 會決定接下來執行動作 (圖.3.12)。

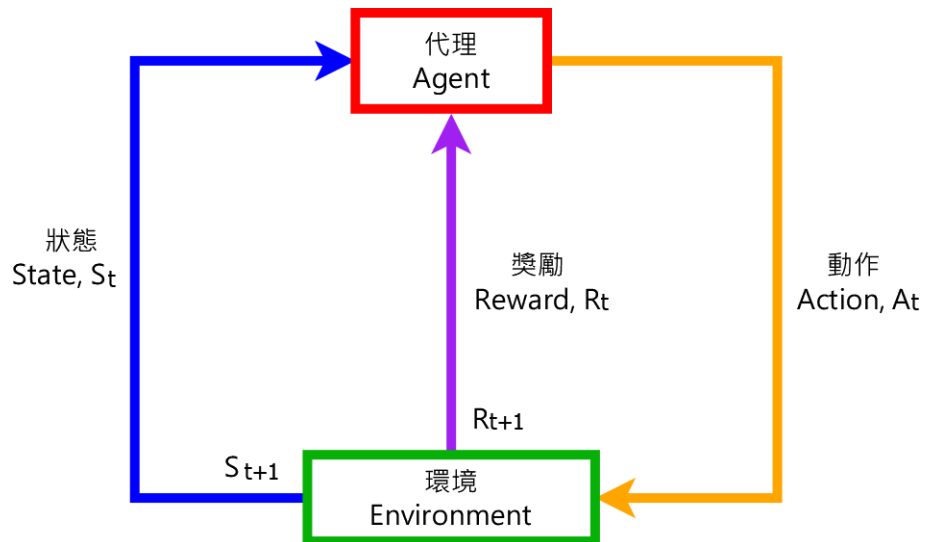


圖. 3.12: 強化學習架構

需要考慮的重點：

強化學習的狀態、獎勵和動作是互相關聯，agent 與環境之間存在著關聯，兩者都影響著狀態和動作並互相影響著彼此：機器人會因動作而造成狀態轉移，狀態的移轉也會影響機器人做出的決策。



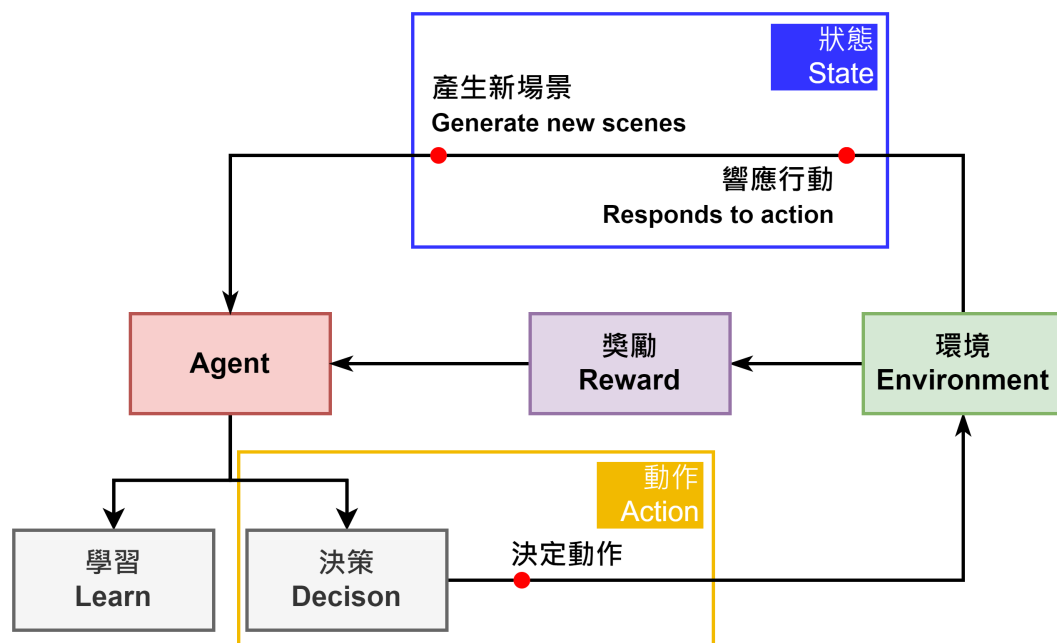


圖. 3.13: 整個互動過程

如(圖.3.13) agent 會透過輸入的狀態來決定採取何種行為(動作)，並試圖採取獲得最高獎勵的行動。當 agent 開始與環境互動時，agent 會透過當前狀態來決定將採取的行動，在 agent 採取行動後，環境的狀態也因此而改變，若 agent 採取行動後所到達我們所要的狀態就會得到獎勵，反之則會給予懲罰。在場景裡透過反覆的訓練，讓強化學習的行為漸漸趨近預期的目標。

強化學習中有兩個很重要的常數： $\gamma$  和  $\lambda$ 。

$\gamma$  會影響所獲得的獎勵。 $\gamma$  又稱為衰減因子，正常狀態下為小於 1 的常數用於每個狀態改變，當狀態改變時為時常數。 $\gamma$  允許使用者在每個狀態給予不同形式的獎勵(這種狀況下  $\gamma$  為 0)，如果著重在長期的決策時，獎勵就不受決策順序所影響(此時  $\gamma$  為 1)

$\lambda$  一般在我們處理時間差異問題時使用。這是涉及更多地連續狀態的預測。在每個狀態中  $\lambda$  值的增加代表演算法正在快速學習。

強化學習的互動是透過 agent 和環境之間的互動會產生獎勵，agent 採取行動，導致狀態改變是一種強化學習實現如何將情況映設為行動的方法，從而找到最大化獎勵的方法，機器或機器人不會像其他機器學習形式的機器人那樣被告知要採取哪些行動。

獎勵的目的與運作以獎勵的方式誘導機器採取我們所期望的動作，機器會採取最大化獎勵的方式，因此可將目的定為最大獎勵，以吸引機器執行期望做的行為。

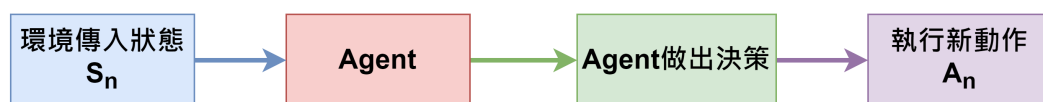


圖. 3.14: agent

強化學習的環境：

強化學習中的環境由某些因素組成，會對 agent 產生影響，agent 必須根據環境適應各種因素，並做出最佳決策，這些環境可以是各種形式，其中包括 2D、3D 或是真實世界。強化學習的環境具有確定性、可觀察性，可以是離散或是連續的狀態，則 agent 可以是單一或多個所組成。

### 3.2.1 馬可夫決策

- Markov Chain

馬可夫鏈 (Markov Chain) 主要是狀態變化的隨機過程 (stochastic process) 和馬可夫屬性 (Markov property) 結合。隨機過程 (stochastic

process) 狀態隨著時間變化，而狀態的變化存在著隨機性，並以數學模式表示。馬可夫屬性 (Markov property) 指在目前以及所有過去事件的條件下，任何未來事件發生的機率，和過去的事件不相關僅和目前狀態相關。當前決策只會影響下個狀態，當前狀態轉移 (action) 到其他狀態的機率會有所差異。

- Markov Reward Process

- action 到指定狀態會獲得獎勵。

$$R(s_t = s) = \mathbb{E}[r_t | s_t = s]$$

$$\gamma \in [0, 1]$$

- Horizon：在無限的狀態以有限的狀態表示。

- Return：越早做出正確決策獎勵越高。

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots + \gamma^{T-t-1} R_T$$

- State value function(決策價值)：

$$V_t(S) = \mathbb{E}[G_t | s_t = s]$$

$$P(s_{s+1} = s' | s_t = s, a_t = a)$$

- Discount Factor ( $\gamma$ ) 獎勵衰減有幾種作法：第一種，越早做出有獎勵的決策，獎勵越高；第二種，做出有價值的決策  $\gamma = 1$ ，不分決策順序先後；第三種，無用的決策  $\gamma = 0$ ，不會得到獎勵。

以 Bellman equation 的方式描述互動關係狀態：

$$V(s) = R(s) + \gamma \sum_{s' \in S} P(s' | s) V(s')$$

$R(s)$ : 立即獎勵

$\gamma \sum_{s' \in S} P(s'|s)V(s')$ ：未來獎勵衰減總和

Analytic solution(分析性解法)，MRP 的分析性解法：

$$V = (1 - \gamma P)^{-1} R$$

Bellman equation 及 Analytic solution 的方式只適合小的 MRP(個數比較少的)，矩陣複雜度為  $O(N^3)$ ， $N$  為狀態個數。若要計算大型的 MRP 會使用疊代法：動態規劃 (Dynamic programming)、Temporal-Difference learning 和 Monte-Carlo evaluation 以評估採樣的方式：

$$g = \sum_{i=t}^{H-1} \gamma^{1-i} r_i$$

$$G_t \leftarrow G_t + g, i \leftarrow i + 1$$

$$V_t(s) \leftarrow \frac{G_t}{N}$$

• Markov Decision Process 在 MRP 中加入決策 (decision) 和動作 (action)

- S：state 狀態
- A：action 動作
- P：狀態轉換  $P(s_{s+1} = s' | s_t = s, a_t = a)$
- R：獎勵，取決於當前狀態和動作會得到相對應的獎勵

$$R(s_t = s, a_t = a) = \mathbb{E}[r_t | s_t, a_t = a]$$

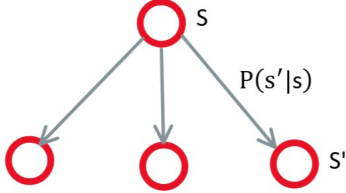
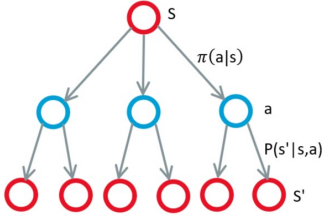
- D：折扣因子 (discount factor)

$$\gamma \in [0, 1]$$

policy(決策)：可以是一個決策行為的機率或確定執行的行為，若以數學方程式表示：

$$\pi(a|s) = P(a_t = a | s_t = s)$$

MRP 和 MDP 方程式互相轉換：

MRP	$\longleftrightarrow$	MDP
$P^\pi(s' s)$	$=$	$\sum_{a \in A} \pi(a s) P(s' s, a)$
$P^\pi(s)$	$=$	$\sum_{a \in A} \pi(a s) P(s, a)$
		

state value function(狀態值方程式) $v^\pi(s)$

$$\begin{aligned}
 v^\pi(s) &= \mathbb{E}[G_t | s_t = s] \\
 &= \mathbb{E}[R_{t+1} + \gamma v^\pi(s_{t+1}) | s_t = s] \\
 &= \sum_{a \in A} \pi(a|s) q^\pi(s, a)
 \end{aligned}$$

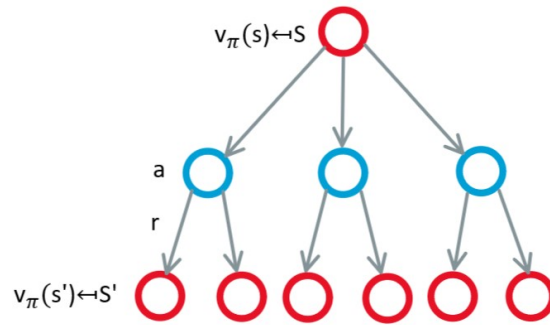


圖. 3.15:  $v^\pi$  程序圖

$$v^\pi(s) = \sum_{a \in A} \pi(a|s) (R(s, a) + \gamma \sum_{s' \in s} P(s'|s, a) v^\pi(s'))$$

state value function(狀態值方程式) $q^\pi(s)$

$$\begin{aligned} v^\pi(s) &= \mathbb{E}[G_t | s_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v^\pi(s_{t+1}) | s_t = s] \\ &= \sum_{a \in A} \pi(a|s) q^\pi(s, a) \end{aligned}$$

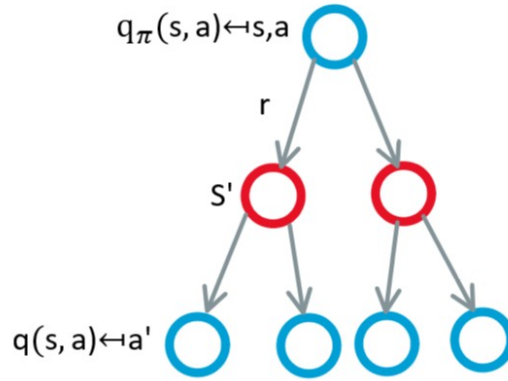


圖. 3.16:  $q^\pi$  程序圖

$$q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) \sum_{a' \in A} \pi(a'|s') q^\pi(s', a')$$

### 3.3 Policy Gradient 理論

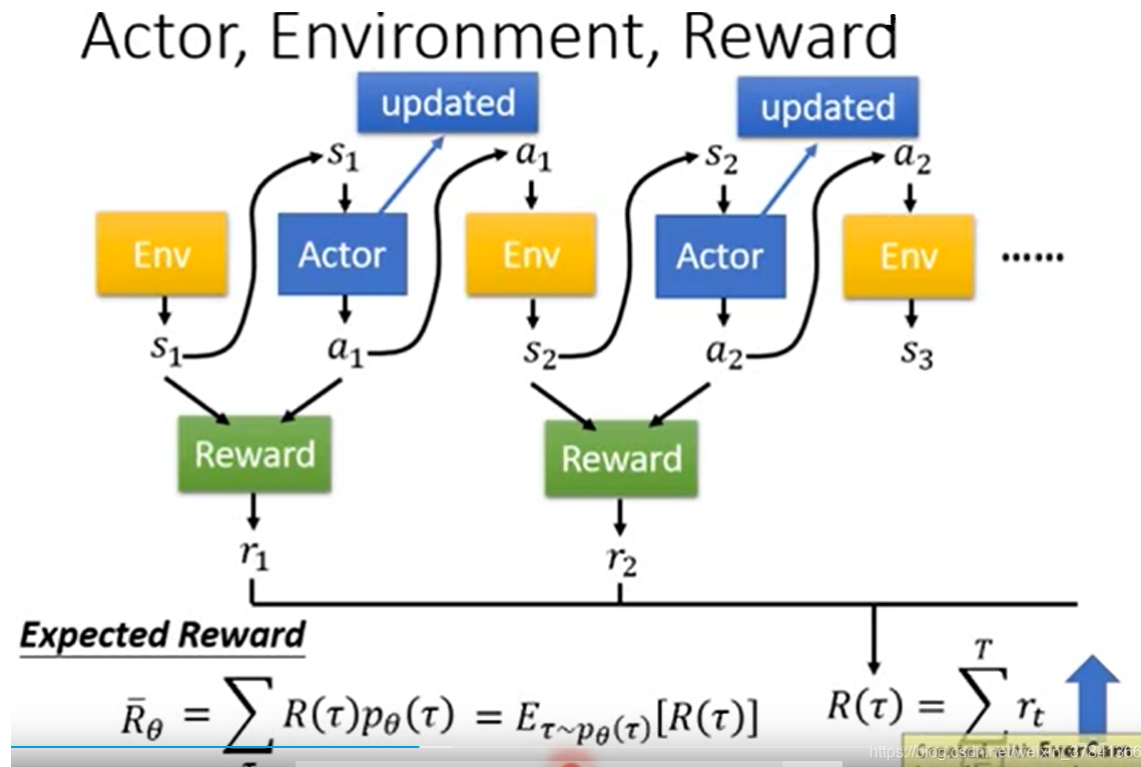


圖. 3.17: Policy Gradient 原理

Policy Gradient 主要目的是直接對決策進行建模與優化。該決策 (policy) 通常使用參數化函數建模，獎勵（目標）函數的值取決於此決策，可以應用各種算法來優化，以獲得最佳獎勵。(參數化：當軟體建置於一給定環境時，再依該環境的實際需求填選參數，即可成為適合該環境。)

參數介紹 [7]:

$\pi$  : policy

$s$  : 狀態 (States)。

$a$  : 動作 (Actions)。

$r$  : 獎勵 (Rewards)。

$S_t, A_t, R_t$  : 一個軌跡時間步長't' 的 State, Action and Reward。

$\gamma$  : 衰減因子 (Discount Factor); 懲罰未來的不確定獎勵 (reward)。

$G_t$  : 回傳衰減後的未來獎勵 (Discounted future reward)  $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$

$P(s', r|s, a)$ ：伴隨著現在狀態 (state) 的  $a$  和  $r$ ，前往下一個狀態  $s'$  的轉移機率矩陣 (單階)。

$\pi(a|s)$ ：隨機策略 (agent 的行為策略)。

$\mu(s)$ ：確定的策略；我們還使用不同的字母將其標記為  $\pi_s$ ，以提供更好的區分，以便我們可以輕鬆判斷策略是隨機的還是具有確定性的

$V(s)$ ：'狀態值函數' 測量狀態的預期收益 (報酬率)

$V^\pi(s)$ ：根據 policy 的狀態值函數  $V^\pi(s) = \mathbb{E}_{a \sim \pi}[G_t | S_t = s]$

$Q(s, a)$ ：行為值函數，評估一對狀態和動作的預期收益。

$Q^\pi(s, a)$ ：根據 policy 的行為值函數  $Q^\pi(s, a) = \mathbb{E}_{a \sim \pi}[G_t | S_t = s, A_t = a]$ 。

$A(s, a)$ ：Advantage Function， $A(s, a) = Q(s, a) - V(s)$ ：像是另一種版本的 Q-value，由狀態值為基準降低方差。

reward function 的值：取決於策略，可應用各種算法優化  $\theta$ ，獲得最佳獎勵。

$$J(\theta) = \sum_{s \in \mathcal{S}} d^\pi(s) V^\pi(s) = \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q^\pi(s, a)$$

Policy Gradient 通過反覆評估梯度來最大化預期的總獎勵 (reward)

$$g = \nabla_\theta \mathbb{E}[\sum_{t=0}^{\infty} r_t] ; g = \mathbb{E}[\sum_{t=0}^{\infty} \psi_t \nabla_\theta \log \pi_\theta(a_t | s_t)]$$

$\psi_t$  可能方法為下列：

- $\sum_{t=0}^{\infty} r$ ：決策軌跡的獎勵總和。
- $\sum_{t'=t}^{\infty} r'$ ：根據動作 (action) 的獎勵 (reward)  $a_t$ 。  
標準表示式： $\sum_{t'=t}^{\infty} r' - b(s_t)$
- $Q^\pi(s_t, a_t)$ ：state-action value function。
- $A^\pi(s_t, a_t)$ ：Advantage Function。
- $r_t + V^\pi(s_t + 1) - V^\pi(s_t)$ ：TD residual。



公式使用定義 [7]：

$$V^\pi(s_t) = \mathbb{E}_{s_t+1:\infty, a_t:\infty} [\sum_{l=0}^{\infty} r_t + l]$$

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s_t+1:\infty, a_t+1:\infty} [\sum_{l=0}^{\infty} r_t + l]$$

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) (\text{Advantage Function})$$

### 3.3.1 Actor Critic

原始的 policy gradient 沒有偏差，但方差大; 所以提出了許多以下算法來減少方差，同時保持偏差不變：

$$g = \mathbb{E} [\sum_{t=0}^{\infty} \psi_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)]$$

Actor-Critic：減少原始政策中的梯度方差包括兩個模型

Critic：更新值函數參數  $w$ ，根據算法，它可以是操作值  $Q_w(a|s)$  或狀態值  $V_w(s)$

Actor：按照 Critic 的建議，將策略參數  $\theta$  更新為  $\pi_{\theta}(a|s)$

它如何在簡單的行動價值參與者批評中發揮作用：

- 隨機的初始化  $s, \theta, w$ ; 取樣  $a \sim \pi_{\theta}(a|s)$
- For  $t = 1 \sim T$ :
  - 1 取樣 reward  $r_t \sim R(s, a)$  隨後下一階段  $s' \sim P(s'|s, a)$
  - 2 樣本的下一個動作  $a' \sim \pi_{\theta}(a'|s')$
  - 3 更新 policy 參數  $\theta$ :

$$\theta \leftarrow \theta + \alpha_{\theta} Q_w(s, a) \nabla_{\theta} \ln \pi_{\theta}(a|s)$$

- 4 計算校正 (TD error) 對於時間  $t$  的動作值:

$$\delta = r_t + \gamma Q_w(s', a') - Q_w(s, a)$$

並使用它來更新操作 action - value function:

$$w \leftarrow w + \alpha_w \delta \nabla_w Q_w(s, a)$$

5 更新  $a \leftarrow a'$  和  $s \leftarrow s'$ ; 學習率:  $\alpha_\theta$  和  $\alpha_w$ 。

## 第四章 訓練環境

### 4.1 OpenAI Gym

Gym 是用於開發和比較強化學習算法的工具包，他不對 agent 的結構做任何假設，並且與任何數據計算庫兼容，而可以用來制定強化學習的算法。這個環境具有共享的介面，使我們能用來編寫常規算法，也就能教導 agents 如何步行到玩遊戲。

### 4.2 Pong

取自 1977 年發行的一款家用遊戲機 ATARI 2600 中的遊戲，內建於 Gym，這是一個橫向的乒乓遊戲，左方是預設電腦玩家，右邊由使用者或是由訓練程式控制 (圖.4.1)。在強化學習範例中，Pong 與實體冰球機簡化後環境相似。

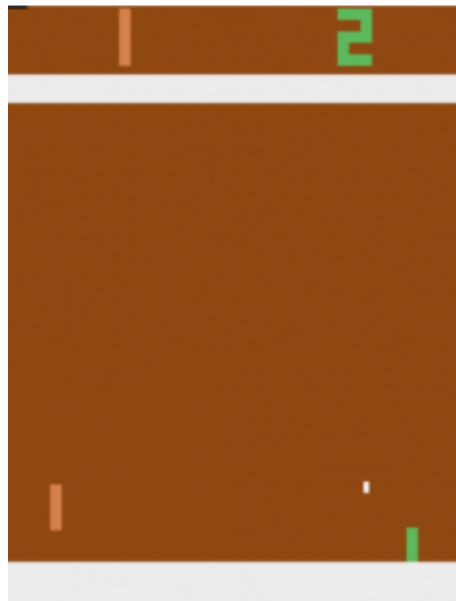


圖. 4.1: ATARI Pong

## 參考文獻

- [1] <https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c>
- [2] <https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e>
- [3] <http://www.incompleteideas.net/book/RLbook2020.pdf>
- [4] <https://medium.com/change-the-world-with-technology/policy-gradient-181d43a24cf5>
- [5] <https://livebook.manning.com/book/grokking-deep-reinforcement-learning/chapter-11/v-11/38>
- [6] <http://ukko.life.nctu.edu.tw/u0517047/usage.html>
- [7] <https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html>
- [8] <https://uupgrade.medium.com/python-那些年我們一起玩過的遊戲-三-打磚塊-d89b648896ca>
- [9] <https://cvfiasd.pixnet.net/blog/post/275774124-深度學習激勵函數介紹>
- [10] <https://www.coppeliarobotics.com/helpFiles/>
- [11] <https://hackernoon.com/the-reason-behind-moving-in-the-direction-opposite-to-the-gradient-f9566b95370b>
- [12] <https://ruder.io/optimizing-gradient-descent/>
- [13] <https://zh.wikipedia.org/wiki/HSL> 和 <https://zh.wikipedia.org/wiki/HSV> 色彩空間
- [14] <https://gist.github.com/karpathy/a4166c7fe253700972fcbc77e4ea32c5#file-pg-pong-py>

[15] [https://github.com/schinger/pong\\_actor-critic/blob/master/pg-pong-ac.py](https://github.com/schinger/pong_actor-critic/blob/master/pg-pong-ac.py)

[16] <https://gist.github.com/etienne87/6803a65653975114e6c6f08bb25e1522>

## 附錄

### LaTeX

LaTeX 為一種程式語言，支援標準庫 (Standard Libraries) 和外部程式庫 (External Libraries)，不過與一般程式語言不同的是，它可以直接表述 Tex 排版結構，類似於 PHP 之於 HTML 的概念。但是直接撰寫 LaTeX 仍較複雜，因此可以藉由 Markdown 這種輕量的標註式語言先行完成文章，再交由 LaTeX 排版。此專題報告採用編輯軟體為 LaTeX，綜合對比 Word 編輯方法，LaTeX 較為精準正確、更改、製作公式等，以便符合規範、製作。

表. 1: 文字編輯軟體比較表

	相容性	直觀性	文件排版	數學公式	微調細部
LaTeX	✓		✓	✓	✓
Word		✓			✓

- 特點:

1. 相容性：以 Word 為例會有版本差異，使用較高版本編輯的文件可能無法以較低的版本開啟，且不同作業系統也有些許差異；相比 LaTeX 可以利用不同編譯器進行編譯，且為免費軟體也可移植至可攜系統內，可以搭配 Github 協同編譯。
2. 文件排版：許多規範都會要求使用特定版型，使用文字編譯環境較能準確符合規定之版型，且能夠大範圍的自定義排定所需格式，並能不受之後更改而整體格式變形。
3. 數學公式呈現：LaTeX 可以直接利用本身多元的模組套件加入、編輯數學公式，在數學推導過程能夠快速的輸入自己需要的內容即可。
4. 細部調整：在大型論文、報告中有多項文字、圖片、表格，需要調整細部時，要在好幾頁中找尋，而 LaTeX 可以分段章節進行編譯，再進行合併處理大章節。

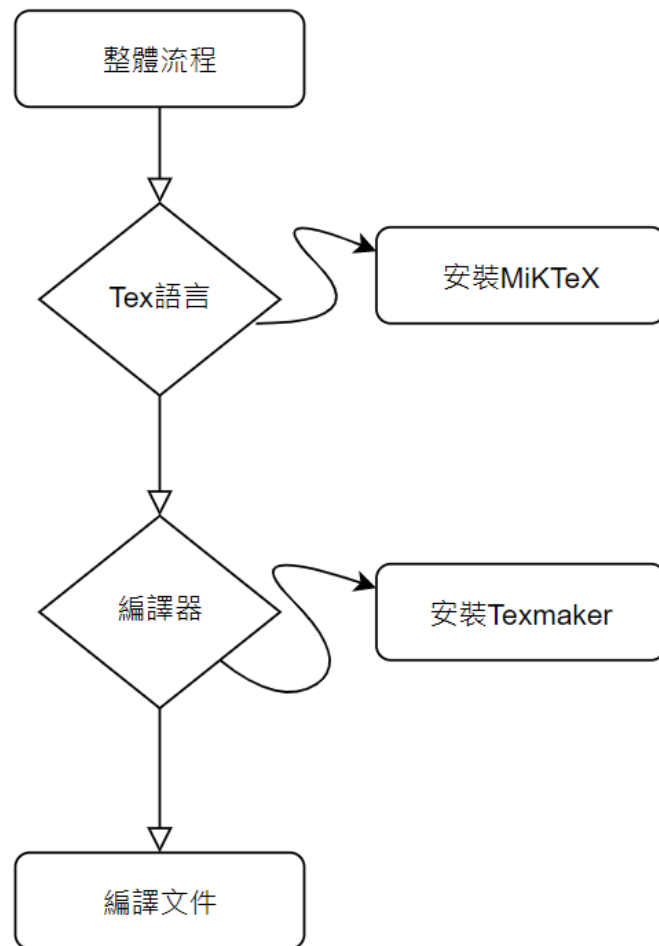


圖. 2: 編譯流程

## FFmpeg

FFmpeg 是一個開放原始碼的自由軟體，可以對音訊和視訊進行多種格式的錄影、轉檔、串流功能。在專題訓練過程中透過 FFmpeg 的視訊錄製的功能記錄對打影像來了解實際訓練狀況。