

解密AI黑盒子



主題四：深度學習神經網路DNN

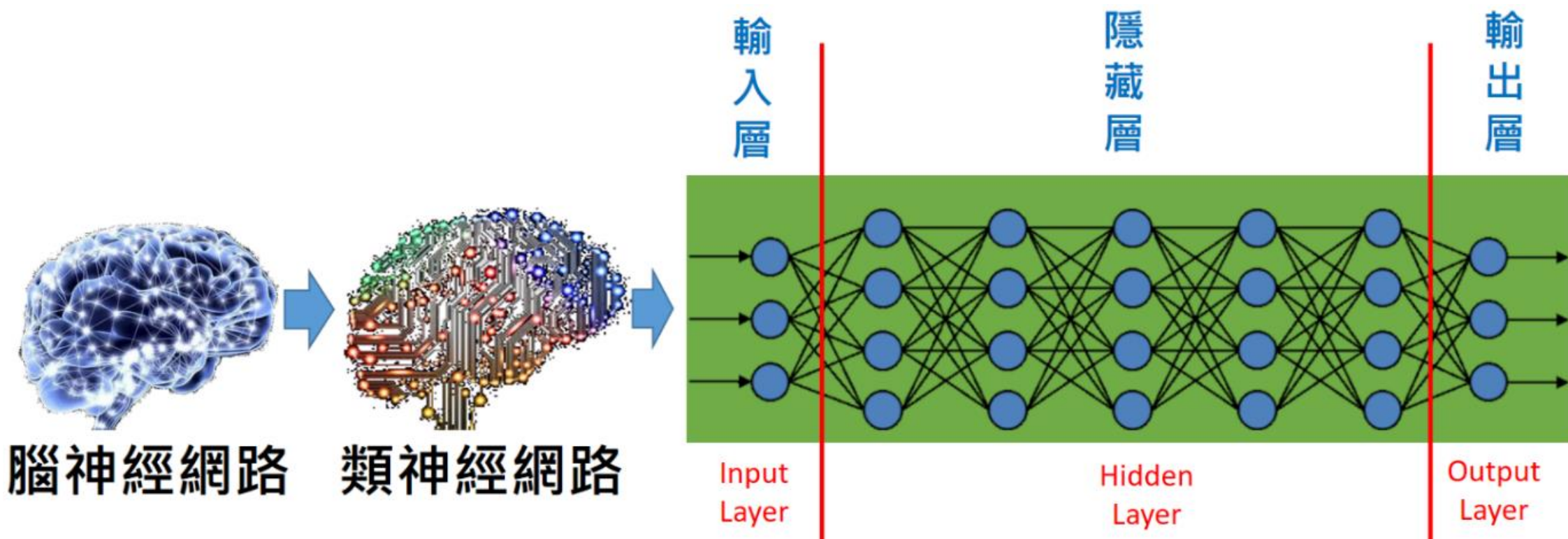
深度學習神經網路 DNN

單元1



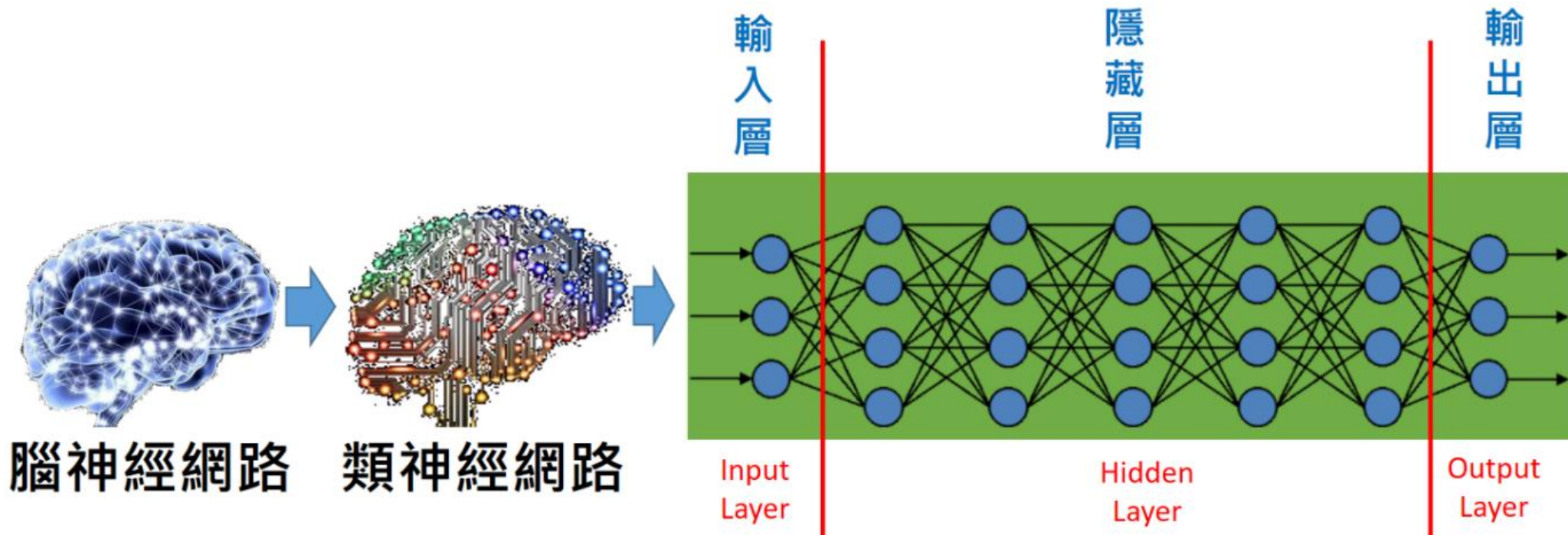
什麼是人工(類)神經網路？

- 人工(類)神經網路(Artificial Neural Network, ANN)：模仿人腦神經網路的連結和處理功能，透過學習後可以模仿人腦的某種功能，或建立輸入和輸出間的正確關係。



人工(類)神經網路的構造

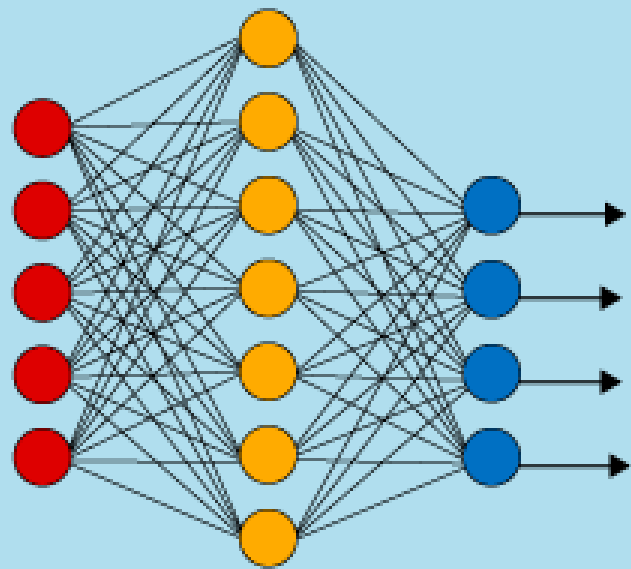
- 輸入層(1層)：樣本特徵由此輸入，其結點數量等於每個樣本的特徵數量。
- 隱藏層(可多層)：層數(決定神經網路的深度)和每一層的節點數(決定神經網路的寬度)可以自訂，隱藏層的功能是一種特徵工程，每一層都在找一種代表特徵。
- 輸出層(1層)：節點數量一用途(迴歸、二元分類、多元分類)而定。
- 類神經網路中每一層有多個節點，每個節點都具有感知器的功能。



多層感知器 和 深度學習神經網路

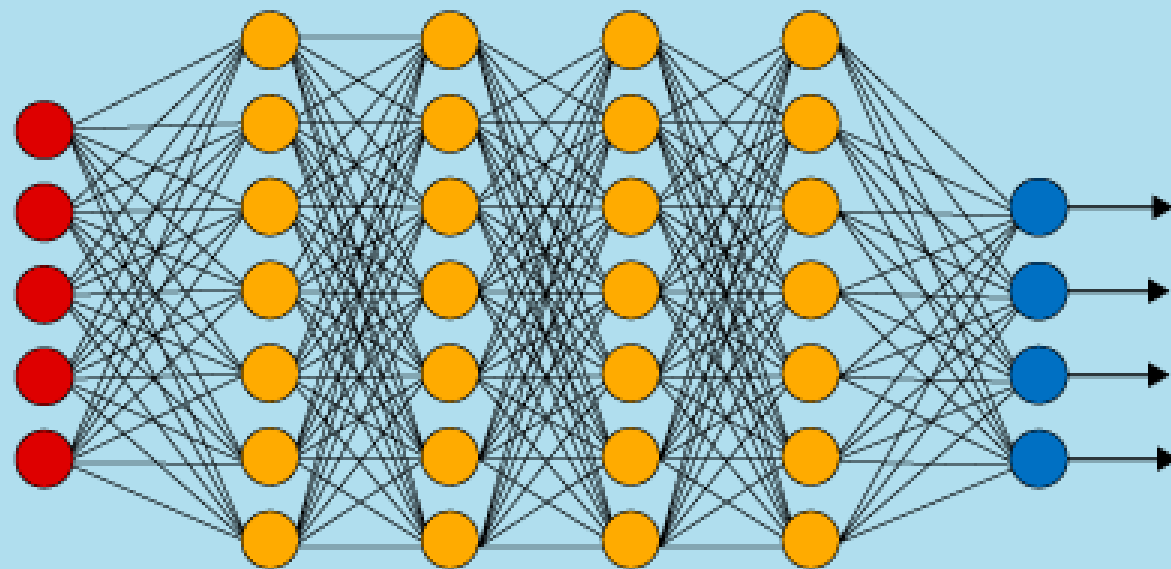
多層感知器MLP (Multi Layer perceptron)

只有一個隱藏層



深度神經網路DNN (Deep Neural Network)

必須有兩個以上的隱藏層



● 輸入層節點

● 隱藏層節點

● 輸出層節點

- 深度學習神經網路因為隱藏層的數量比較多，每一層都負責抽取專屬的特徵，且後一層通常能去抽取前一層更深度的特徵。

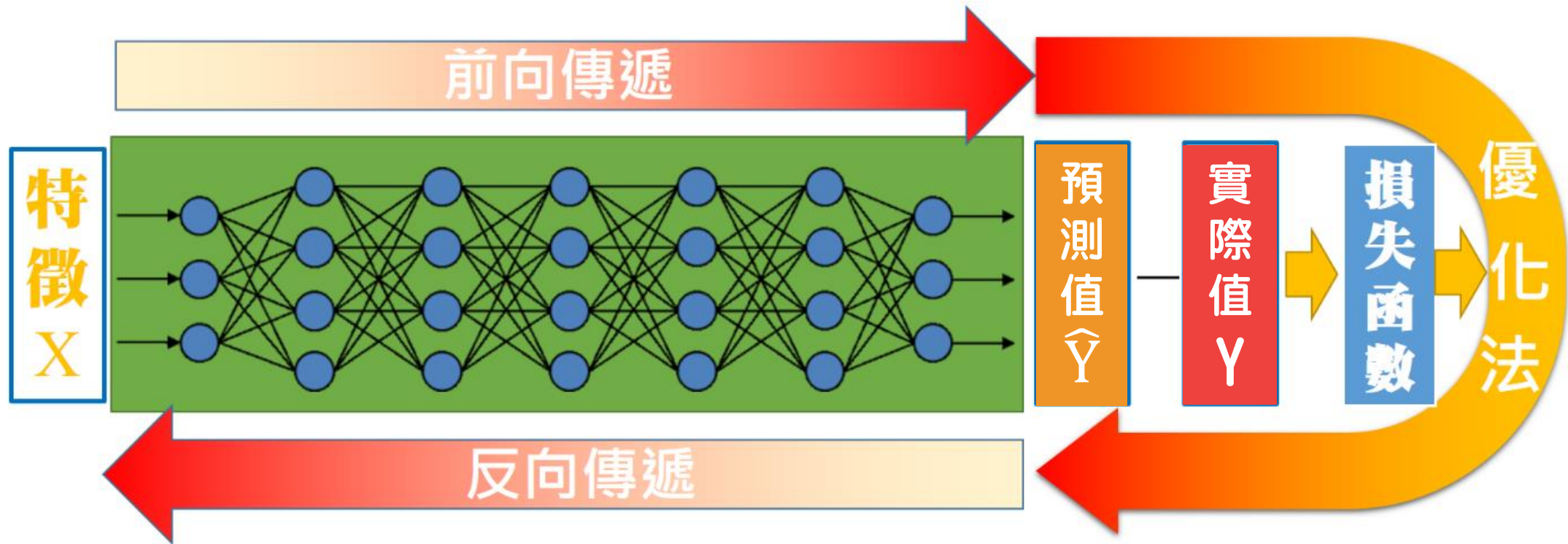
類神經網路的原理

單元2



神經網路的資料處理流程

- 前向傳遞(Forward propagation)
- 優化器處理(optimizer)
- 反向傳遞(Backward propagation)



前向傳遞(Forward propagation)

- 接收上一層多個節點輸出的資料，乘以對應的權重並做加總運算，再經過激活函數處理後，輸出到下一層多個節點。
- 輸入層到第1隱藏層：
$$Z^{(1)} = XW^{(1)} + B^{(1)}、H^{(1)} = G(Z^{(1)})$$
- 第n-1隱藏層到第n隱藏層：
$$Z^{(n)} = H^{(n-1)}W^{(n)} + B^{(n)}、H^{(n)} = G(Z^{(n)})$$
- 第N-1隱藏層到輸出層N：
$$Z^{(N)} = H^{(N-1)}W^{(N)} + B^{(N)}、H^{(N)} = G(Z^{(N)})$$

Pytorch中各層之間的線性運算：

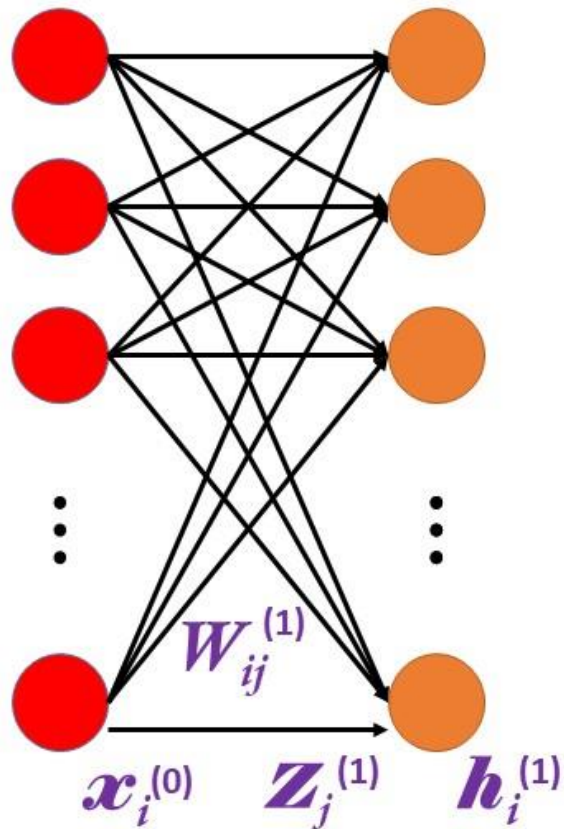
$$Z^{(n)} = H^{(n-1)}W^{(n)} + B^{(n)}$$

可以利用`torch.nn.Linear(n-1層節點數,n層節點數)`

神經網路的全連接架構(Fully connect)

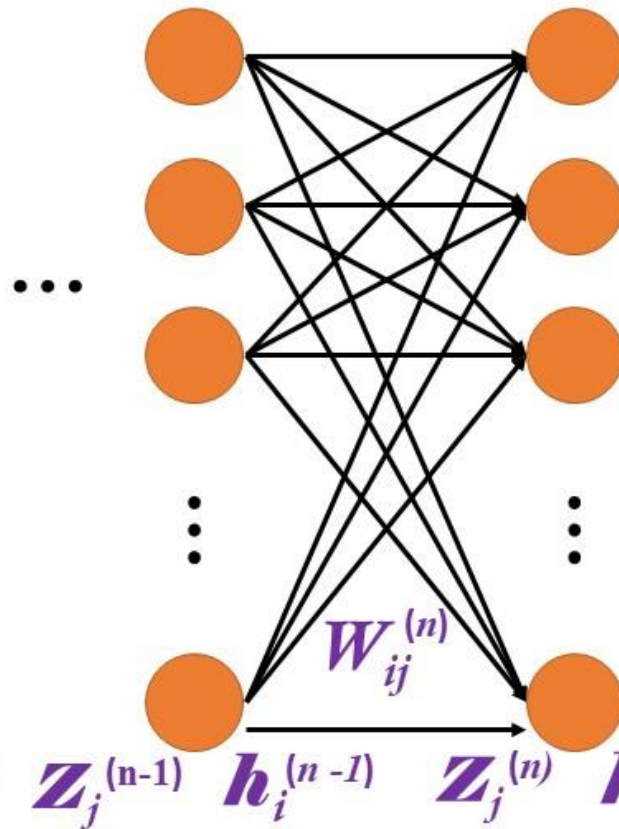
$$\mathbf{Z}^{(1)} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{B}^{(1)}$$

$$\mathbf{H}^{(1)} = \mathbf{G}(\mathbf{Z}^{(1)})$$



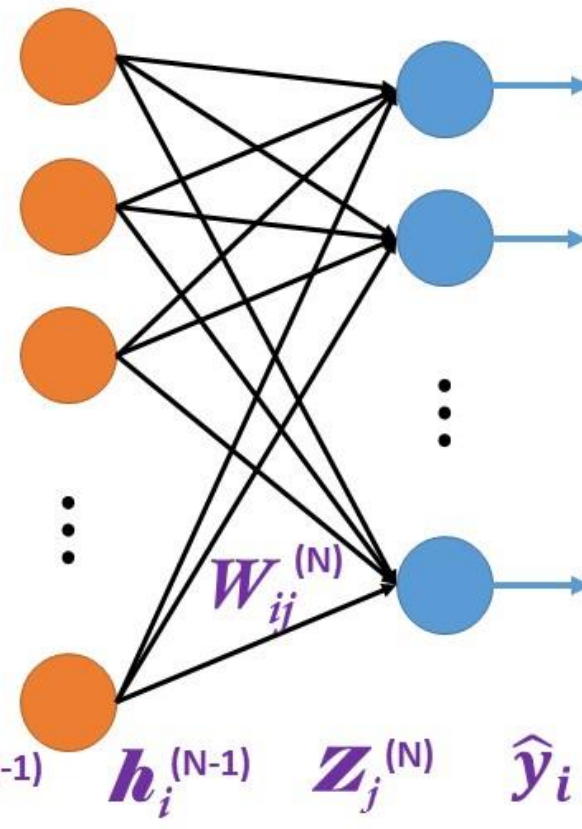
$$\mathbf{Z}^{(n)} = \mathbf{H}^{(n-1)}\mathbf{W}^{(n)} + \mathbf{B}^{(n)}$$

$$\mathbf{H}^{(n)} = \mathbf{G}(\mathbf{Z}^{(n)})$$



$$\mathbf{Z}^{(N)} = \mathbf{H}^{(N-1)}\mathbf{W}^{(N)} + \mathbf{B}^{(N)}$$

$$\mathbf{H}^{(N)} = \mathbf{G}(\mathbf{Z}^{(N)})$$



$$\frac{\partial L}{\partial \mathbf{Z}^{(1)}} \leftarrow \frac{\partial L}{\partial \mathbf{Z}^{(n-1)}} \leftarrow \frac{\partial L}{\partial \mathbf{Z}^{(n)}} \leftarrow \frac{\partial L}{\partial \mathbf{Z}^{(N-1)}} \leftarrow \frac{\partial L}{\partial \mathbf{Z}^{(N)}}$$

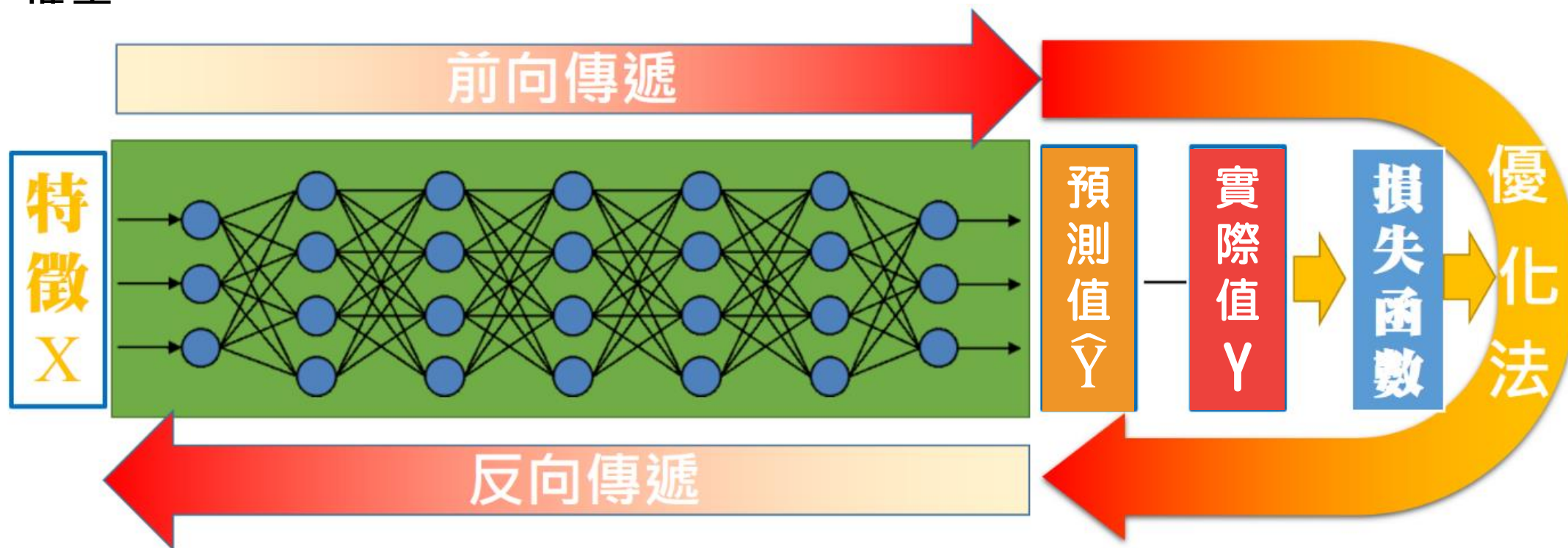
輸出層和損失函數

■ 輸出層根據用途(迴歸、二元分類、多元分類)使用不同激活函數 $G^{(N)}$ 和損失函數：

用途	迴歸	二元分類	多元分類
激活函數	沒有 $\hat{y} = z^{(N)}$	$\hat{y} = \text{Sigmoid}(Z_j^{(N)})$	$\hat{y} = \text{SoftMax}(Z_j^{(N)})$
損失函數	均方誤差 Mean square error	交叉熵 cross-entropy	交叉熵 cross-entropy
torch.nn.	MSELoss()	BCELoss() BCEWithLogitsLoss()	CrossEntropyLoss()

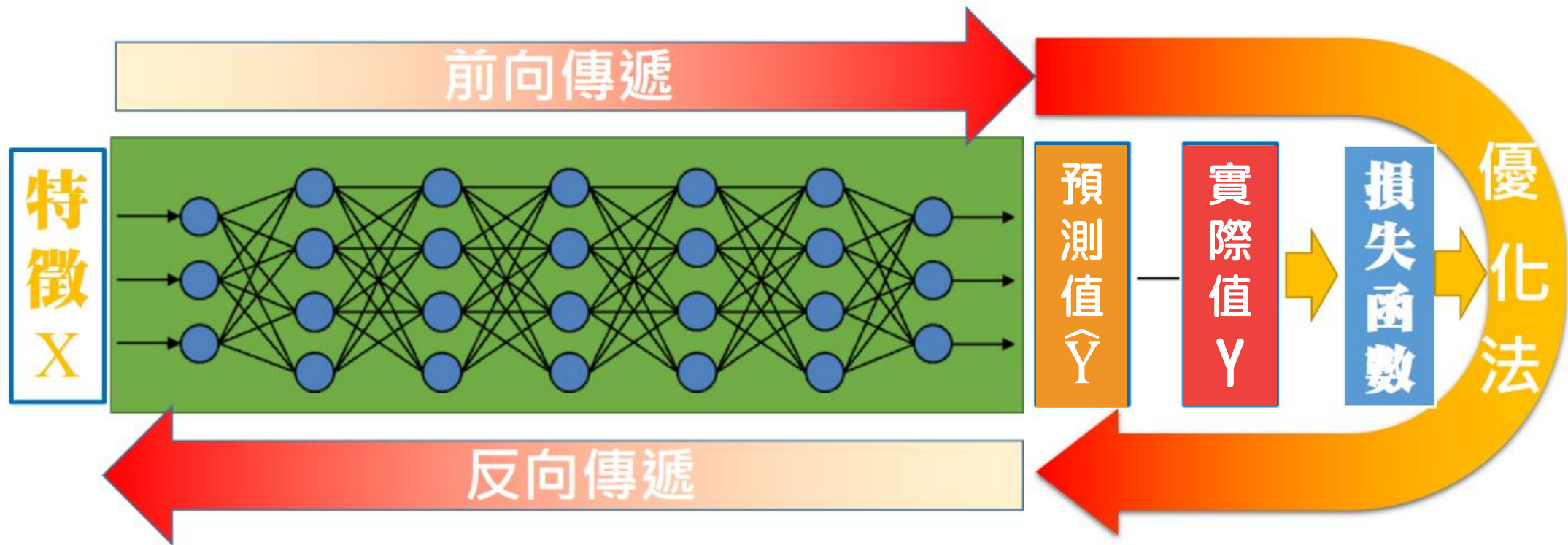
輸出層和損失函數

- 優化器(optimizer)處理：前向傳遞的終點即為輸出層，輸出層輸出的預測值 \hat{y} 和實際值 y 用來計算損失函數，用來評估預測值和實際值間的差距，判斷是否要結束訓練，若要繼續訓練則以一些優化方法(例如：梯度下降法)更新權重。



反向傳遞 (Backward propagation)

- 主要利用微分的連鎖法則，由輸出層的誤差開始，往輸入層逐一更新每一層的權重參數。

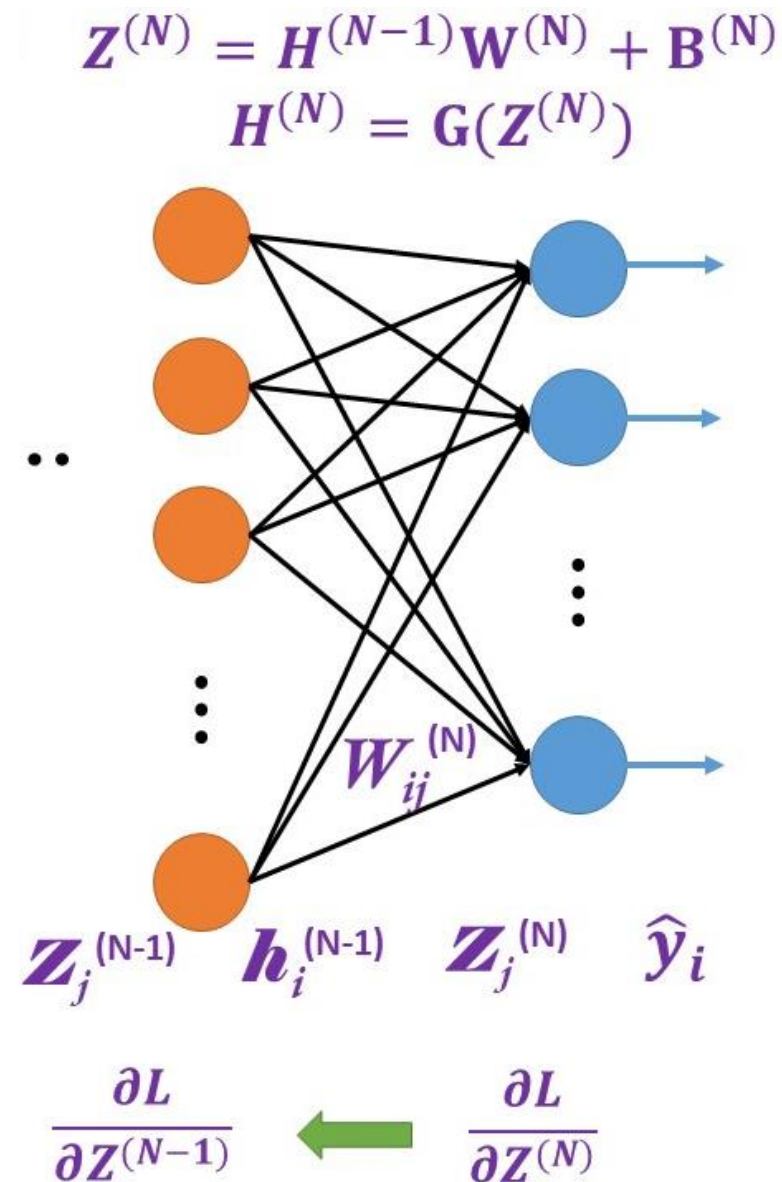


反向傳遞 (Backward propagation)

■ Loss值對輸出層加權輸入 $z_j^{(N)}$ 的偏微分為 $\frac{\partial L}{\partial z_j^{(N)}}$ ：

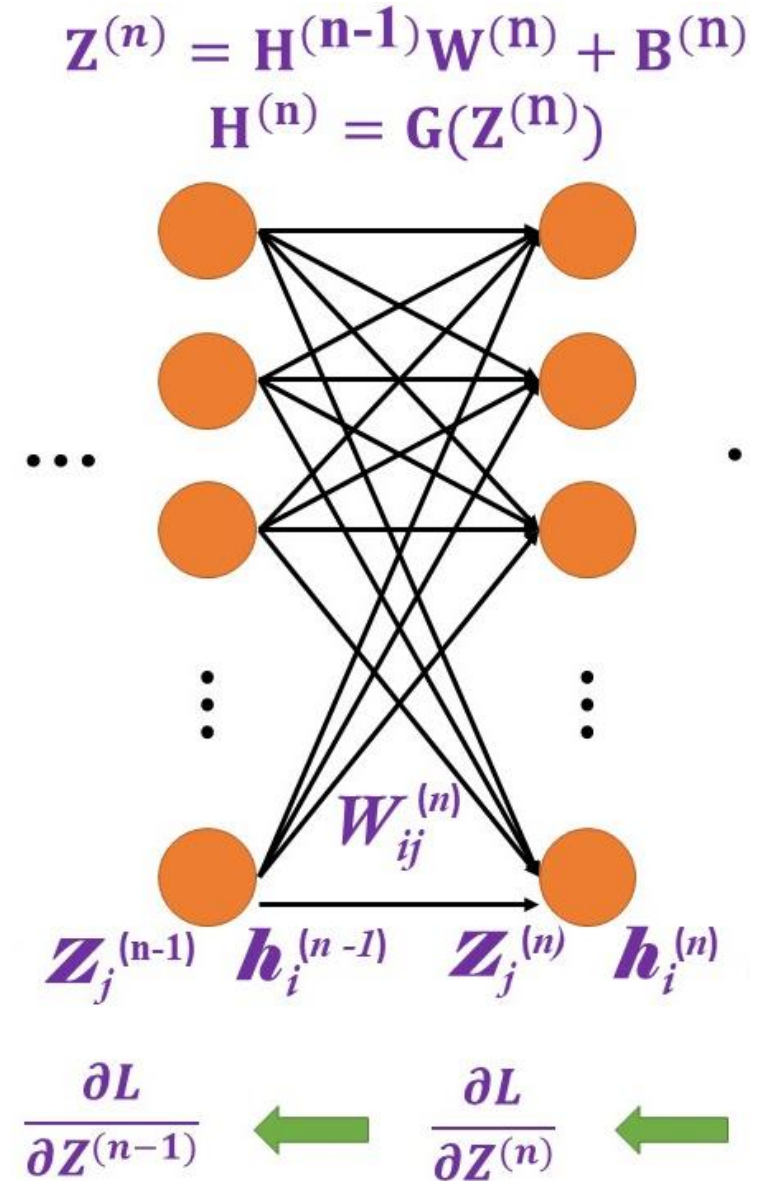
此值不論輸出為迴歸、二元分類、多元分類皆相同。

$$\frac{\partial L}{\partial \mathbf{Z}^{(N)}} = \sum_{k=1}^K (\hat{y}_k - y_k)$$



- 第n層隱藏層反向到第n-1隱藏層：

$$\frac{\partial L}{\partial \mathbf{Z}^{(n-1)}} = \frac{\partial L}{\partial \mathbf{Z}^{(n)}} \frac{\partial \mathbf{Z}^{(n)}}{\partial \mathbf{h}^{(n-1)}} \frac{\partial \mathbf{h}^{(n-1)}}{\partial \mathbf{Z}^{(n-1)}}$$

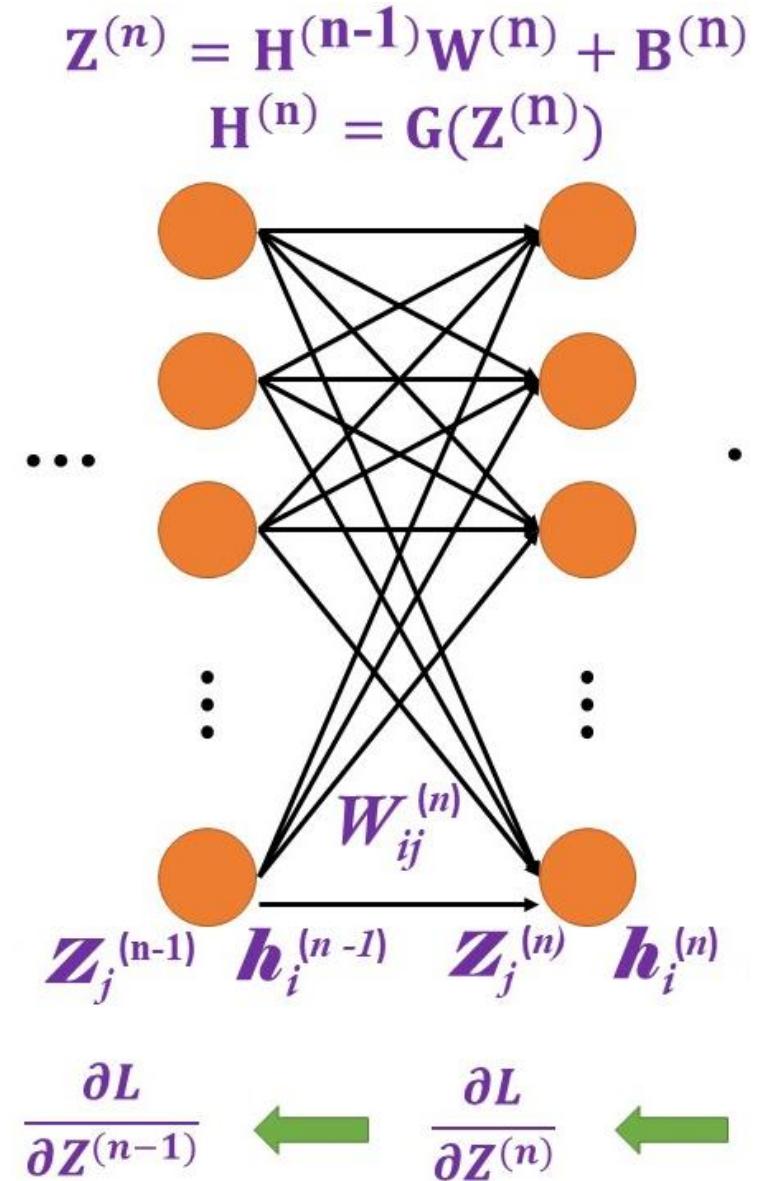


梯度的反向傳遞

- 若利用反向傳播求出損失函數L對第n隱藏層的加權值梯度 $\frac{\partial L}{\partial Z^{(n)}}$

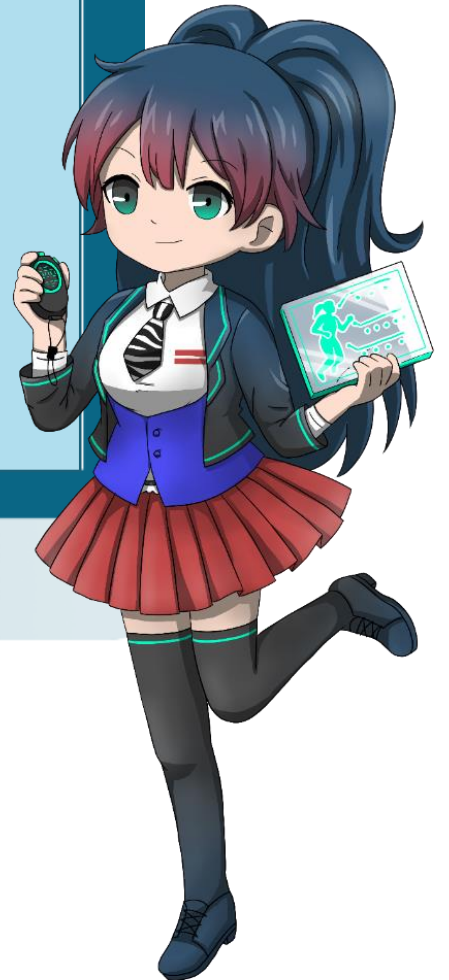
$$\frac{\partial L}{\partial w^{(n)}} = \frac{\partial L}{\partial Z^{(n)}} \frac{\partial Z^{(n)}}{\partial w^{(n)}} = \frac{\partial L}{\partial Z^{(n)}} H^{(n-1)}$$

$$\frac{\partial L}{\partial b^{(n)}} = \frac{\partial L}{\partial Z^{(n)}} \frac{\partial Z^{(n)}}{\partial b^{(n)}} = \frac{\partial L}{\partial Z^{(n)}}$$



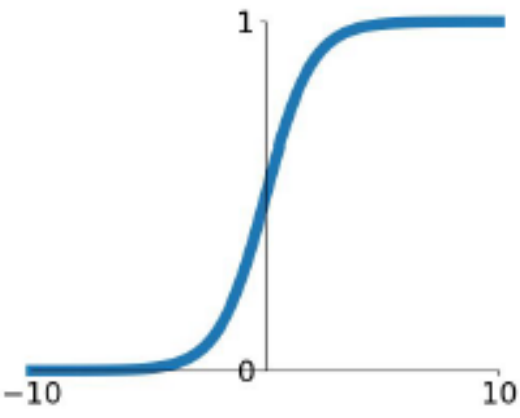
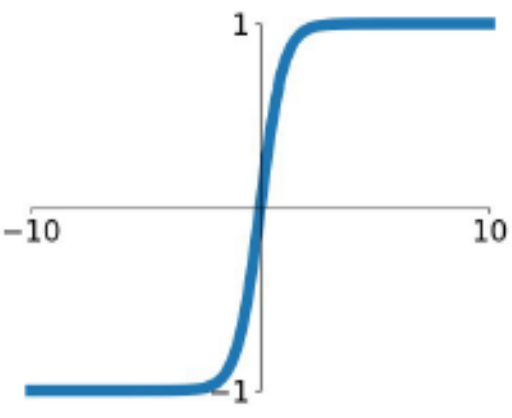
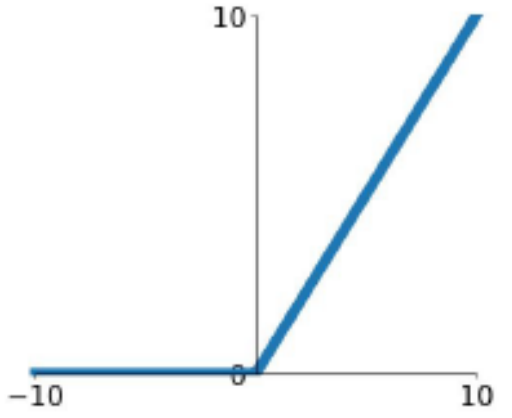
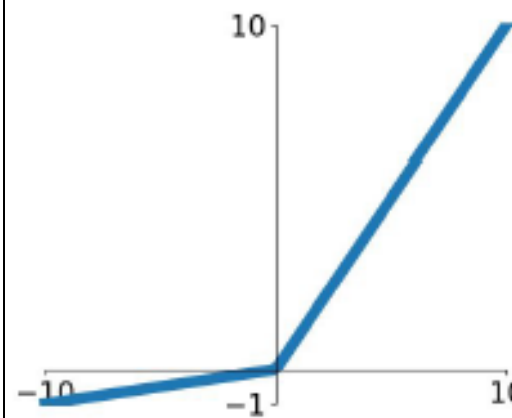
單元3

激活函數 Activation Function



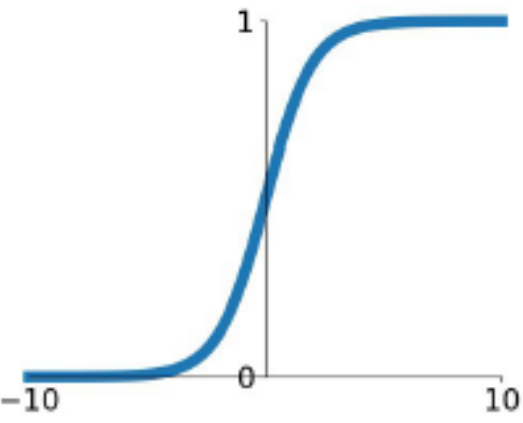
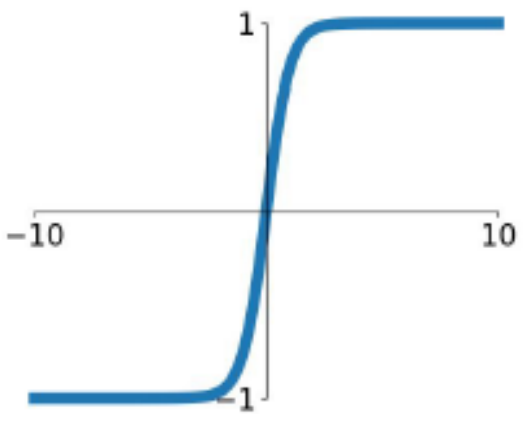
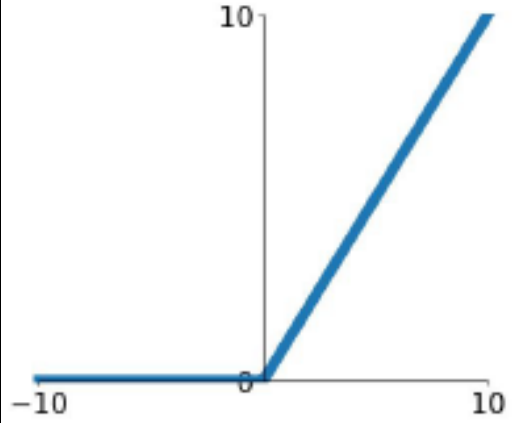
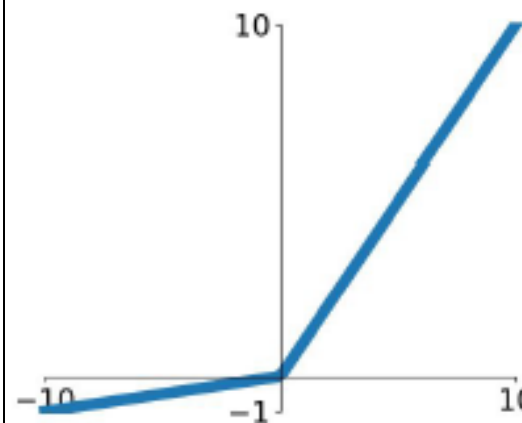
激活函數(Activation Function)

■ 各種激活函數的比較

名稱	Sigmoid	Tanh	ReLU	Leaky ReLU
函數圖				
方程式	$f(x) = \frac{1}{1 + e^{-x}}$	$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$	$f(x) = \begin{cases} x \geq 0 : x \\ x < 0 : 0 \end{cases}$	$f(x) = \begin{cases} x \geq 0 : x \\ x < 0 : \alpha x \end{cases}$
導函數	$\frac{df}{dx} = f(x)(1 - f(x))$	$\frac{df}{dx} = 1 - f(x)^2$	$\frac{df}{dx} = \begin{cases} x \geq 0 : 1 \\ x < 0 : 0 \end{cases}$	$\frac{df}{dx} = \begin{cases} x \geq 0 : 1 \\ x < 0 : \alpha \end{cases}$

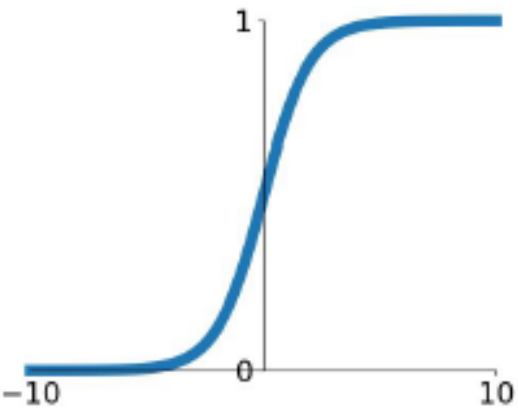
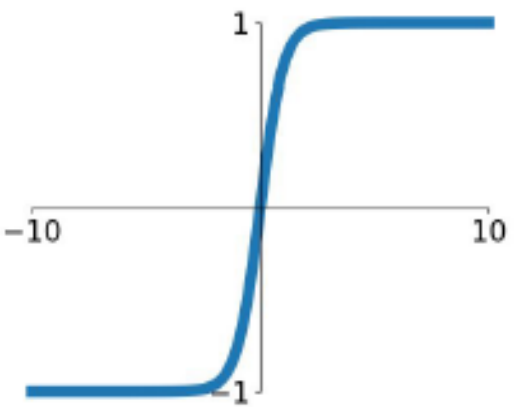
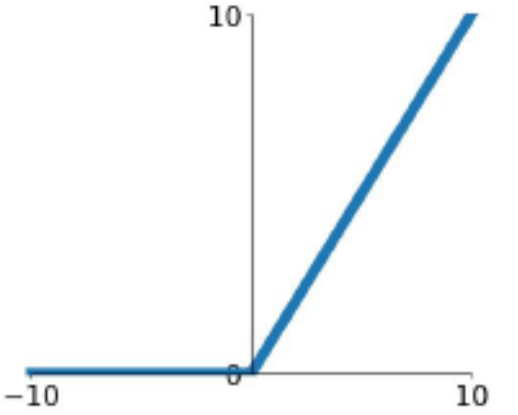
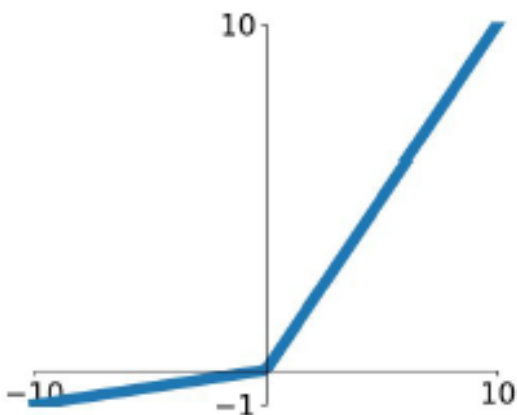
激活函數(Activation Function)

■ 各種激活函數的比較

名稱	Sigmoid	Tanh	ReLU	Leaky ReLU
函數圖				
優缺點	<ul style="list-style-type: none"> ● 計算相對複雜，耗時較長。 ● 梯度會衰減迅速，甚至於消失。 ● 最大梯度只有 0.25，造成訓練緩慢。 	<ul style="list-style-type: none"> ● 梯度比 Sigmoid 明顯上升。 ● 依然存在梯度消失問題。 	<ul style="list-style-type: none"> ● 可以避免梯度消失的情況發生。 ● 容易出現死亡神經元。 ● 計算較為簡單，處理速度快。 	<ul style="list-style-type: none"> ● 引入了一個小斜坡從而保持更新值具有活力。

激活函數(Activation Function)

■ 各種激活函數的比較

名稱	Sigmoid	Tanh	ReLU	Leaky ReLU
函數圖				
用途	用於輸出層作二元分類使用。	用於取代 Sigmoid，有較好的梯度下降。	只能用於隱藏層。	用於取代 ReLU，避免出現死亡神經元。

● 如何選擇合適的激活函數

- 在淺層神經網路中，選擇使用哪種激勵函數影響不大。
- 首選 ReLU，速度快，但是要注意學習速率的調整，如果 ReLU 效果欠佳,嘗試使用 Leaky ReLU。

激活函數(Activation Function)

- Pytorch 中的激活函式有很多, 包含 : Relu, sigmoid, tanh, softplus, leaky_relu
- 比較常用的激活函數relu、sigmoid、tanh, 直接使用torch.relu()、torch.sigmoid()、torch.tanh()。
- 比較少用的激活函數softplus、leaky_relu, 都在torch.nn.functional裡面。

```
1 import torch
2 import torch.nn.functional as F
3 data_x=torch.linspace(-10,10,1000)
4 sigmoid_y=torch.sigmoid(data_x)
5 tanh_y=torch.tanh(data_x)
6 relu_y=torch.relu(data_x)
7 softplus_y=F.softplus(data_x)
8 leakyrelu_y=F.leaky_relu(data_x,negative_slope=0.1)
```

建立神經網路模型 並 進行訓練

單元4



如何建立神經網路？

- 要使用PyTorch來建立神經網路需要引入一些函式庫。
 - `torch.nn`：創建神經網路層時需要用的到函式庫。
 - `torch.nn.functional`：提供許多神經網路會用的函式庫，如一些損失函數`loss function`等。
- 與`torch.nn`的差別是`functional`是只提供"純函式"，而`nn`則是包裝成整個`nn.module`，基本上兩者能互相轉換。

```
1 import torch.nn as nn
2 import torch.nn.functional as F
```

如何建立神經網路？

- 引入torch.nn後就可以來建立我們自己的神經網路了，整個神經網路模型(model)會包裝在一個class裡面，我們需要建立一個class並繼承torch.nn.model。
- 覆寫(override)__init__和forward()來自訂我們的神經網路。
 - __init__(self)：定義各種層網路，像是全連接層、卷積層、、、等等。
 - forward(self, x)：前向傳播(forward propagation)，讓訓練資料x通過整個神經網路的各層運算。

建立神經網路的程式模板

```
1 class NET(nn.Module):                                # Model 繼承 nn.Module
2
3     def __init__(self):                                # 覆寫建構子
4         super(NET, self).__init__()                  # 繼承原生class的建構子
5         self.fc1 = nn.Linear(4, 32)                   # 全連接層 1
6         self.fc2 = nn.Linear(32,10)                   # 全連階層2
7
8     def forward(self, x):                              # 覆寫前向傳播 · x是訓練資料
9         x=self.fc1(x)                                  #全連接1運算
10        x = F.relu(x)                                  # 經過激活函數relu()
11        x=self.fc2(x)                                  #全連接2運算
12        return x                                       #回傳神經網路的結果
```

特殊的神經網路層BatchNormal

- BatchNorm 最早在全連線網路中被提出，對每一層神經元的輸入做歸一化，其優點有下列幾個：
 - 防止過擬合：單個樣本的輸出依賴於整個 mini-batch，防止對某個樣本過擬合。
 - 加快收斂：梯度下降過程中，每一層的權重 W 和偏值都會不斷變化，導致輸出結果的分佈在不斷變化，後層網路就要不停地去適應這種分佈變化。用 BatchNorm 後，可以使每一層輸入的分佈近似不變。
 - 防止梯度彌散：forward 過程中，逐漸往非線性函式的取值區間的上下限兩端靠近，（以 Sigmoid 為例），此時後面層的梯度變得非常小，不利於訓練。
- Pytorch中如何使用：
 - `nn.BatchNorm1d(特徵數量)`
 - `nn.BatchNorm2d(特徵數量)`

利用dropout解決過擬合問題

- 每次反覆運算訓練時會隨機讓一定比例的神經元休息，不參與訓練，以避免過度擬合。
- Pytorch中如何使用：
 - `nn.Dropout`(神經元休息比例)

訓練結果的儲存

- 經過訓練和驗證的過程後，一個可靠的神經網路模型就產生，當我們要應用這個神經網路來做其它預測時，不用再重新訓練。
- 建好的神經網路模型本身的結構是固定的，隨時可以自己重新快速建立這個結構，因為我們儲存的目標其實是當前模型的參數(例：每一層的權重和偏值等)。
- Pytorch 提供了一個函式叫做 `state_dict()` 用來讀取模型的參數，其輸出為每個層映射到其參數張量的字典型態。
- 使用 `torch.save` 將模型參數儲存到檔案，注意儲存前如果模型是在GPU上，記得先轉到CPU。

```
1 print(model.state_dict())           # 查看模型model的參數
2 model.cpu()                         # 將模型轉到cpu
3 torch.save(model.state_dict(), "檔案名稱.pth") # 儲存模型參數
```

載入之前的訓練結果

- 我們只有儲存模型的參數，沒有儲存整個模型的結構，首先還是要宣告 model，但最好在CPU的運算模式下。
- 再改用儲存的 state_dict 參數代替原本初始的參數。
- 成功載入參數後，如果有必要再將模型改成GPU的運算模式。

```
1 model2.load_state_dict(torch.load( "檔案名稱.pth" ))  
2 model2.to(device)
```


神經網路模型的訓練流程

1

- 準備好訓練資料(Training data)。

2

- 預處理(Preprocessing)這些資料，建立DataSet、DataLoader等物件。

3

- 建好Neural Network架構(決定神經網路架構、loss function、Optimizer)

4

- 將訓練資料迭代丟進Neural Network運算(forward propagation)，得到預測結果。

5

- 預測結果以及真實結果來計算loss function值

6

- 對loss值用反向傳播法(backward propagation)算出每個神經網路中參數的梯度

7

- 使用Optimizer(SGD、Momentum、Adam...)和參數的梯度更新參數權重

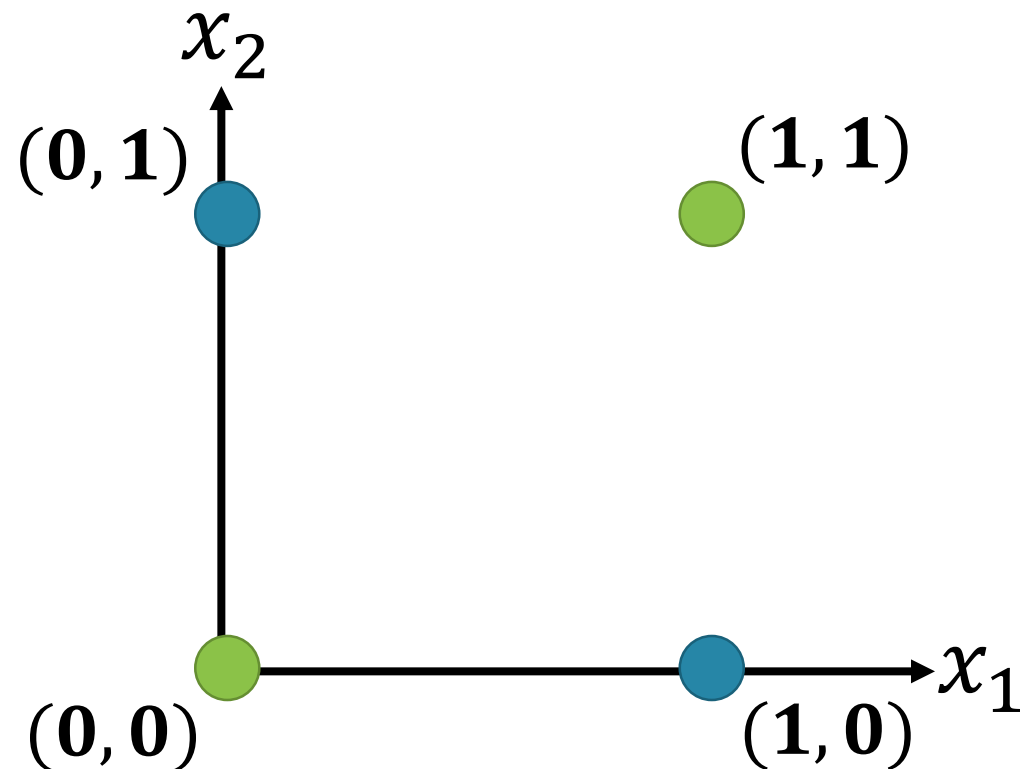
8

- 重複步驟4~7，持續到訓練結束(loss值小於定義的門檻值、執行N次訓練等)

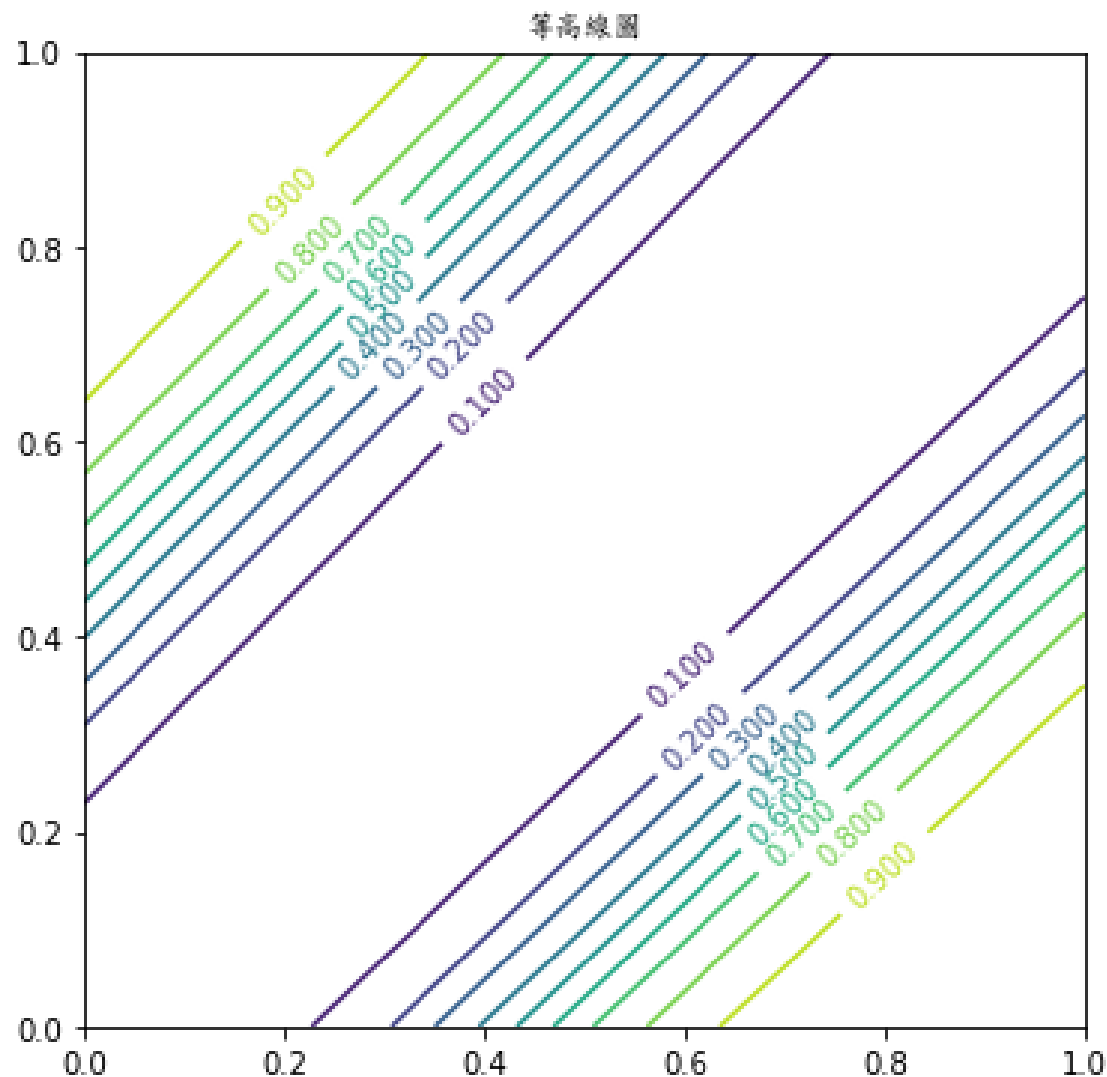
- 感知器被提出的初期，之所以沒被重視很重要的原因之一是連簡單的XOR問題都沒辦法處理。
- XOR問題是一個非線性分類問題，必須用多層感知器來處理。

XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



- 建立一個多層感知器(MLP)：輸入層有2個節點，隱藏層有3個節點，輸出層有1個節點。
- 隱藏層的激活函數使用Relu，輸出層的激活函數使用Sigmoid。
- 利用訓練完成的多層感知器(MLP)，畫出分類的決策邊界等高線。

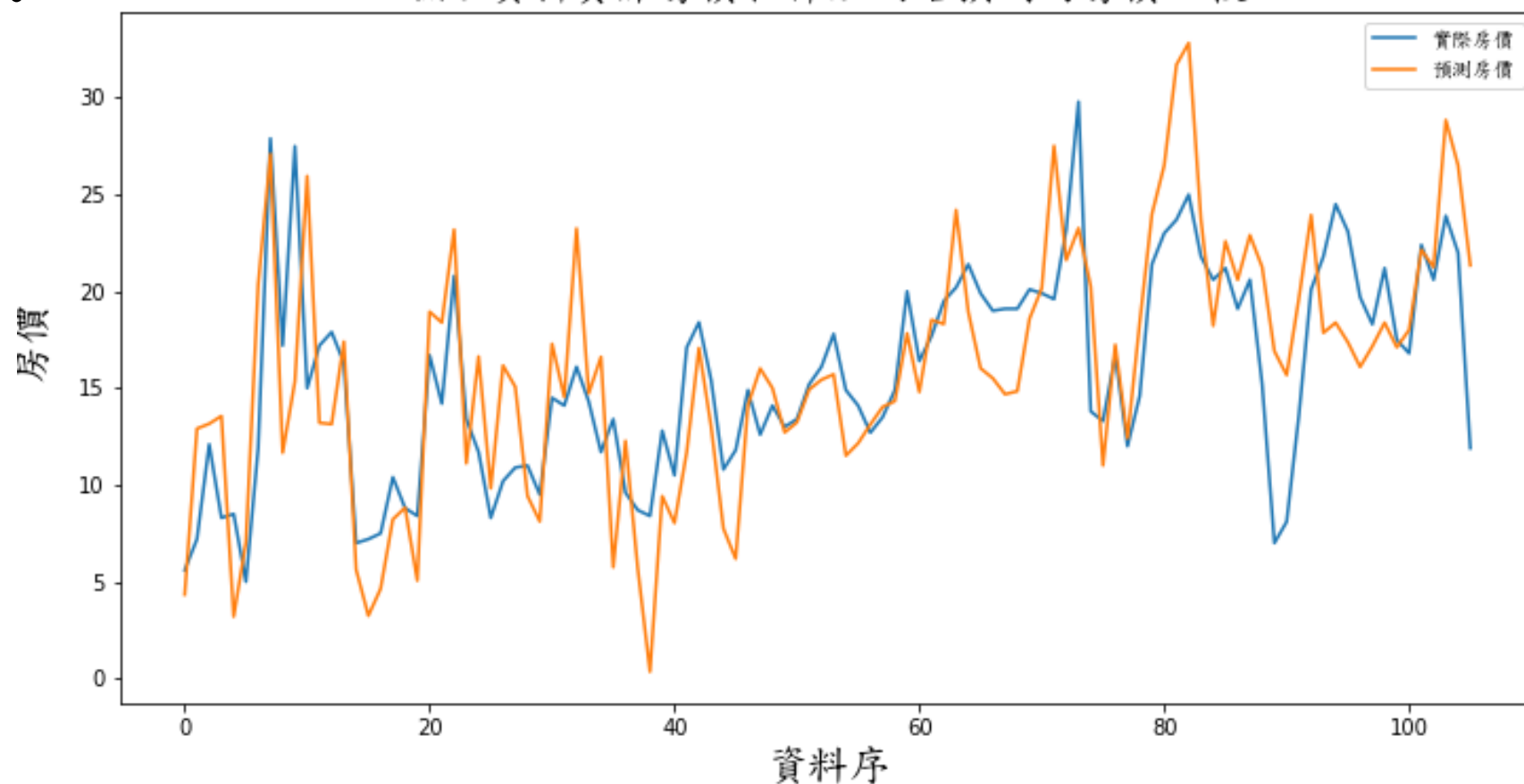


Boston Housing 數據集包含有關波士頓不同房屋的資料。該數據集中有 506 個樣本和 13 個特徵變量。

欄位	欄位代碼	說明	類別
第 1 欄	CRIM	城鎮人均犯罪率。	特徵資料
第 2 欄	ZN	住宅用地超過 25000 sq.ft. 的比例。	
第 3 欄	INDUS	城鎮非零售商用土地的比例。	
第 4 欄	CHAS	查理斯河空變數(如果邊界是河流，則為 1；否則為 0)。	
第 5 欄	NOX	一氧化氮濃度。	
第 6 欄	RM	住宅平均房間數。	
第 7 欄	AGE	1940 年之前建成的自用房屋比例。	
第 8 欄	DIS	到波士頓五個中心區域的加權距離。	
第 9 欄	RAD	輻射性公路的接近指數。	
第 10 欄	TAX	每 10000 美元的全值財產稅率。	
第 11 欄	PTRATIO	城鎮師生比例。	
第 12 欄	B	$1000(Bk-0.63)^2$ ，其中 Bk 指城鎮中黑人的比例。	
第 13 欄	LSTAT	人口中地位低下者的比例。	
第 14 欄	MEDV	自住房的平均房價，以千美元。	依變量

- 建立一個神經網路：輸入層有13個節點，隱藏層數、節點數自訂，輸出層有1個節點。
- 將資料集分成訓練資料集400筆，其它為驗證資料集。
- 利用訓練完成的神經網路，預測驗證資料集中房屋的價格，畫出和實際值的比較關係圖。

驗證資料實際房價和神經網路預測的房價比較



- 建立一個神經網路：輸入層有4個節點，隱藏層數、節點數自訂，輸出層有3個節點。
- 隱藏層的激活函數使用Relu
- 數據集分割為前70筆為訓練資料，後面剩下為驗證資料。
- 利用訓練完成的神經網路，以驗證資料集驗證其準確度。

- 灰階影像是由許多像素排列而成，每一個像素資料由8bits組成，能儲存0(黑色)~255(白色)256階的亮暗層次。
- 最基礎的影像分類方法是將圖片的像素由上而下，由左而右將像素資料拉平後，將每個位置的像素點視為資料特徵，輸入神經網路的輸入層，再經過隱藏層、輸出層逕行分類。
- 上述影像分類方法有所限制：
 - 影像必須位於圖片的中央位置。
 - 圖片尺寸不可過大，且僅限灰階，以免造成輸入節點過多，難以訓練。

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25
26	27	28	29	30
31	32	33	34	35

拉平



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

- MNIST 是一個手寫數字的圖像資料集，每個圖片大小(長x寬)是 28 x 28 像素。
- 檔案mnist_train.csv包含訓練資料(共60000筆)
檔案mnist_valid.csv包含驗證資料(共10000筆)。
- 檔案內一列一筆資料，每一筆資料逗號分開785欄位
 - 第1欄為0、1、2、3、4、5、6、7、8、9的數字標識
 - 第2欄~785欄為28X28=784個像素的灰階數字
 - 灰階數字由0~255依序代表黑到白的深淺程度
- 建立一個神經網路：輸入層有784個節點，
隱藏層個數、節點數自訂，輸出層有10個節點。
- 隱藏層的激活函數使用Relu
- 利用訓練完成的神經網路，以驗證資料集驗證其準確度。

