深度學習

第 5 章 神經網路理論篇

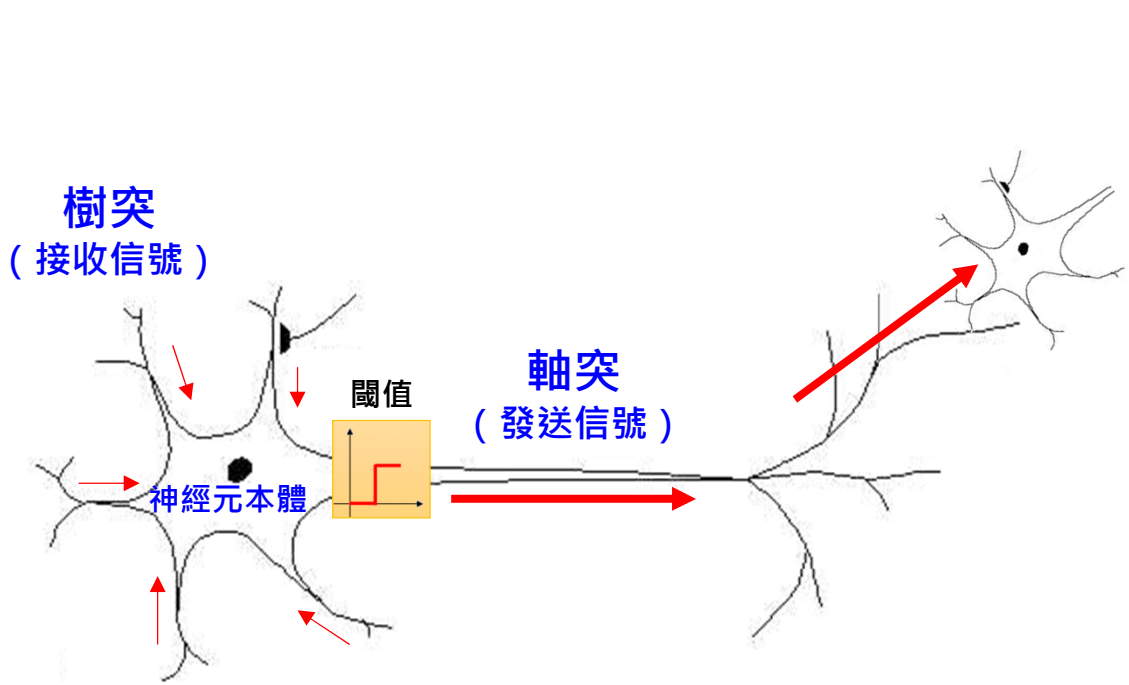
講師：紀俊男

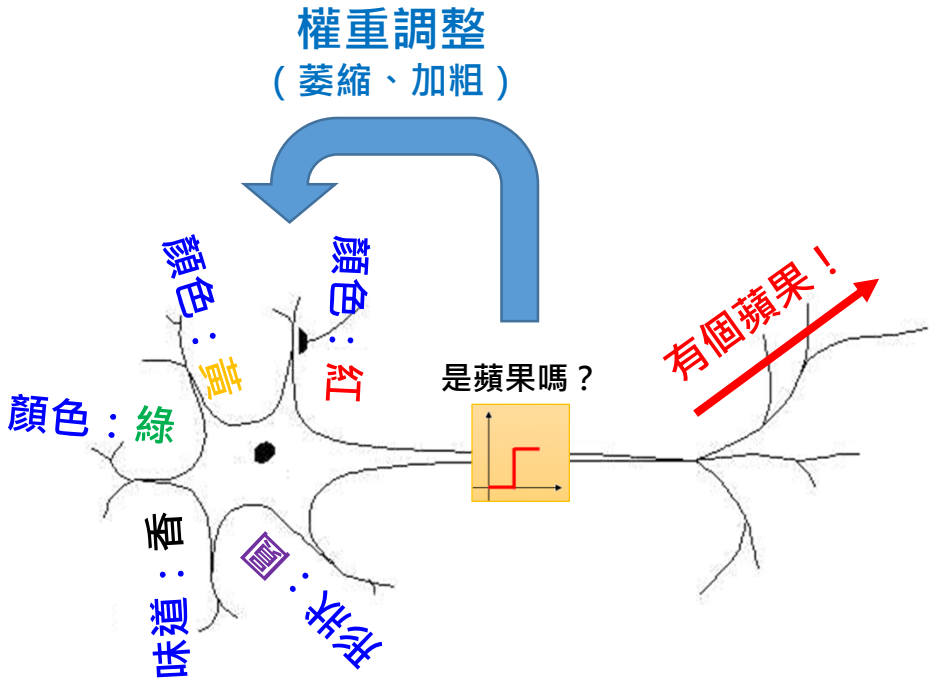
本章大綱

- **感知器原理**
- **多層感知器原理**
  - 原理概說
  - 五大元件

感知器原理
Principle of Perceptron

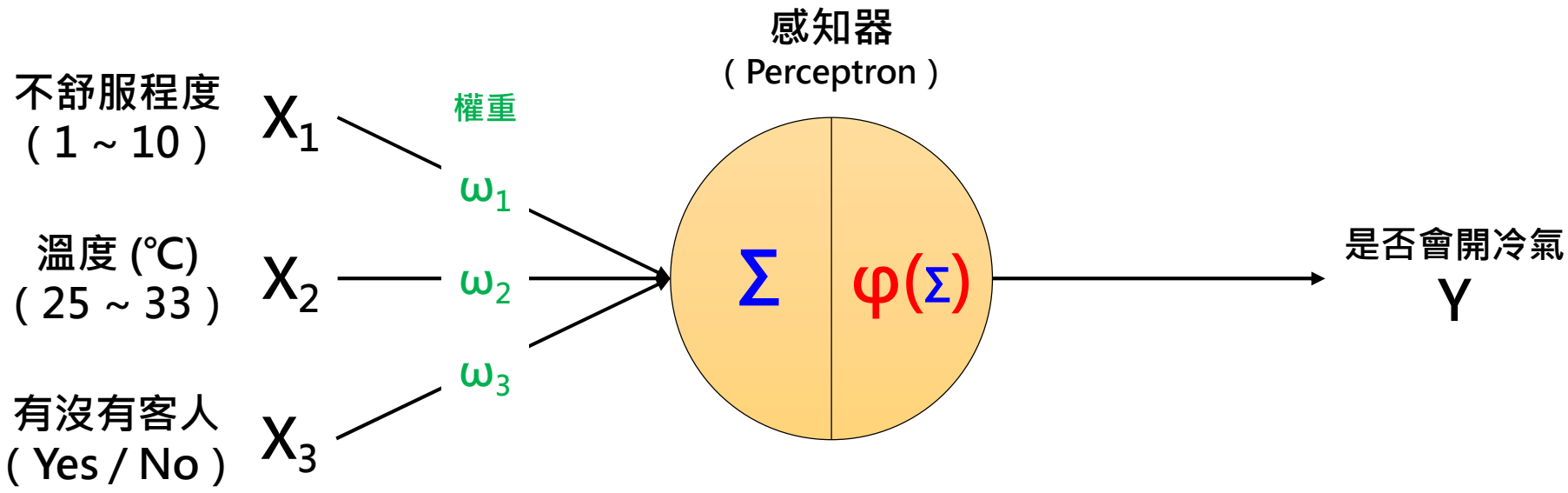
生物神經元（Neuron）





人工神經元：感知器（Perceptron）

- **弗蘭克·羅森布拉特**（Frank Rosenblatt），心理學家, 康乃爾航空實驗室, *1957*

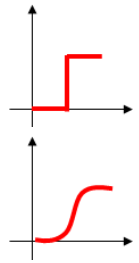


$$\Sigma = \omega_1 X_1 + \omega_2 X_2 + \omega_3 X_3$$

各成分加權計算

激活函數 φ(Σ)
（Activation Function）

- Step Function
- Sigmoid Function



= 線性邏輯迴歸分類器

感知器實例

平常需要做
「特徵縮放」

5	8	4
28	29	32
1	0	0

不舒服程度
(1 ~ 10) X_1

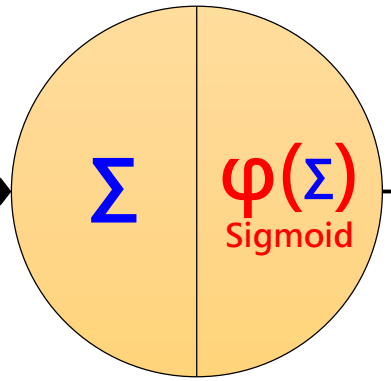
溫度 (℃)
(25 ~ 33) X_2

有沒有客人
(Yes / No) X_3

權重
init = "uniform"

0.1
0.1
0.1

感知器
(Perceptron)



是否會開冷氣
$\hat{Y}$

不舒服	溫度	客人	開冷氣
4	32	No	No
8	29	No	Yes
5	28	Yes	Yes

正向傳播 (計算損失函數)

損失函數
$C = \frac{1}{2}(\hat{Y} - Y)^2$

一樣本點修正一次：隨機梯度下降

K 樣本點修正一次：批次隨機梯度下降

全體樣本點修正一次：一般梯度下降

Σ=4*0.1+32*0.1+0*0.1=3.6 $\hat{Y} = \varphi(3.6) = \frac{1}{1+e^{-3.6}} = 0.9734 = Yes$ ½ (1-0)²=0.5

Σ=8*0.1+29*0.1+0*0.1=3.7 $\hat{Y} = \varphi(3.7) = \frac{1}{1+e^{-3.7}} = 0.9758 = Yes$ ½ (1-1)²=0.0

Σ=5*0.1+28*0.1+1*0.1=3.4 $\hat{Y} = \varphi(3.4) = \frac{1}{1+e^{-3.4}} = 0.9677 = Yes$ ½ (1-1)²=0.0

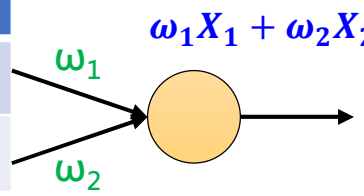
反向傳播 (修正權重，讓損失函數有最小 (偏微分))

全體樣本點訓練一次 = 一期 (Epoch)

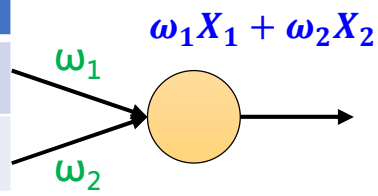
感知器的致命傷

- 只能用於「線性可分」的問題 -- Marvin Minsky, Seymour Papert; *1969*

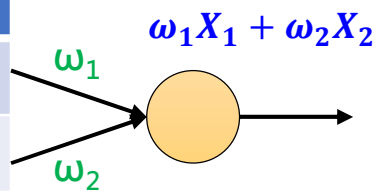
AND	0	1
0	0	0
1	0	1

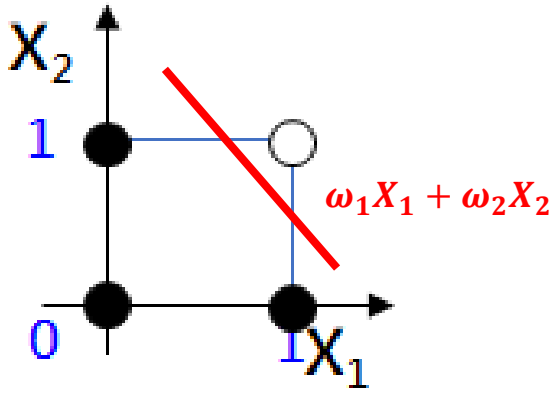


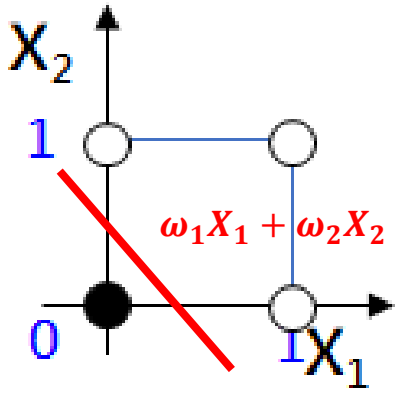
OR	0	1
0	0	1
1	1	1

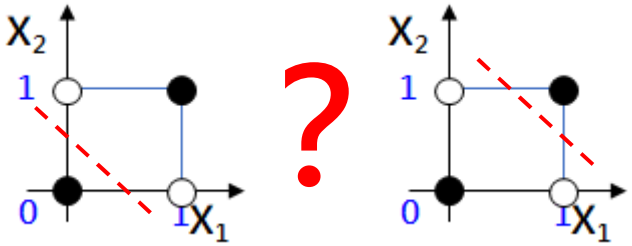


XOR	0	1
0	0	1
1	1	0





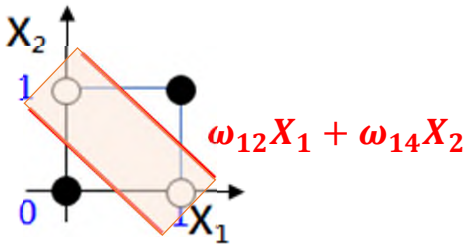


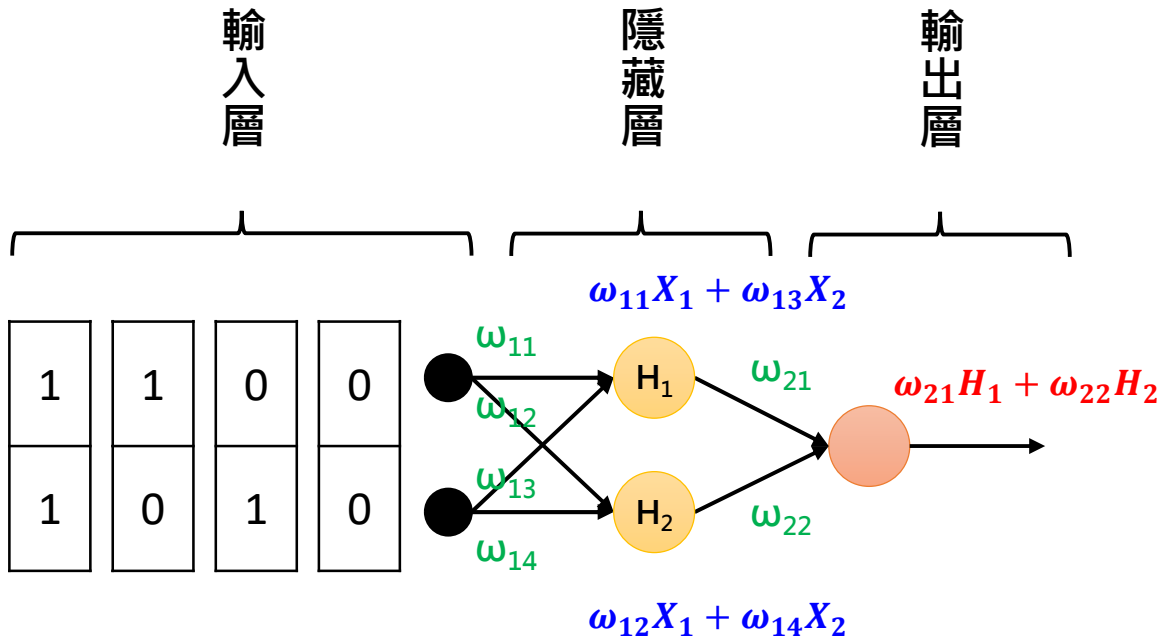


感知器致命傷的解法

- 多層感知器（Multi-Layers Perceptron）

XOR	0	1
0	0	1
1	1	0



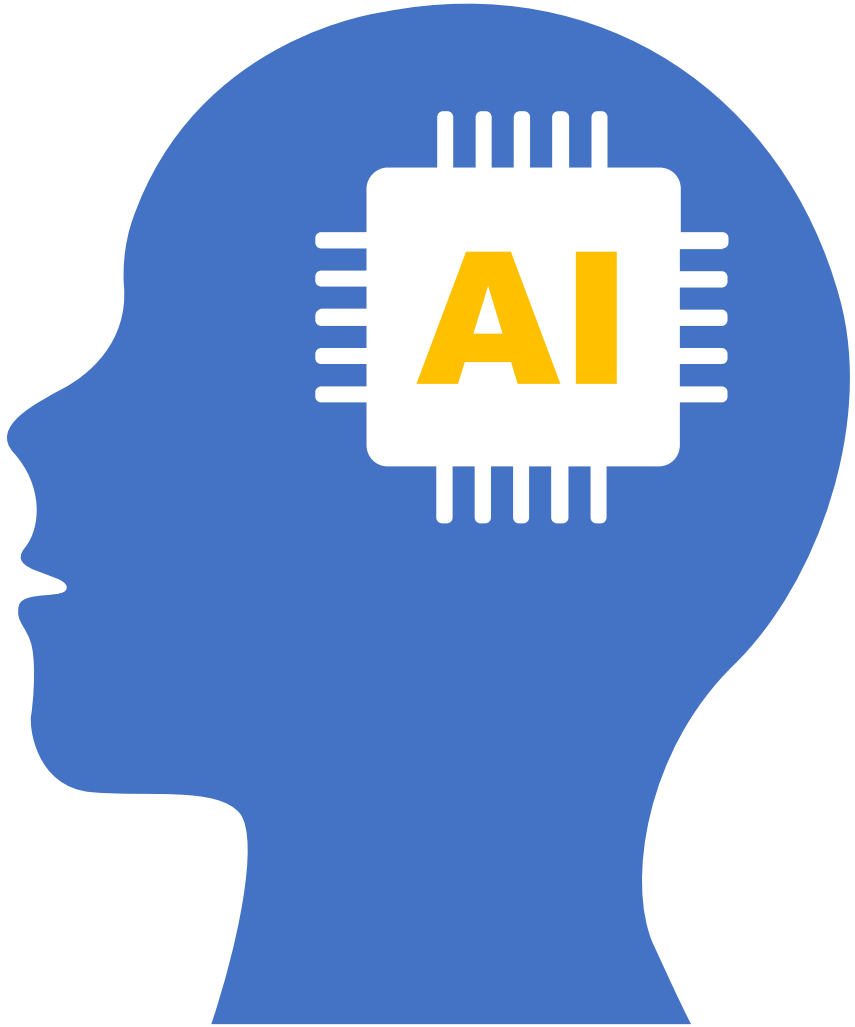


多層感知器原理

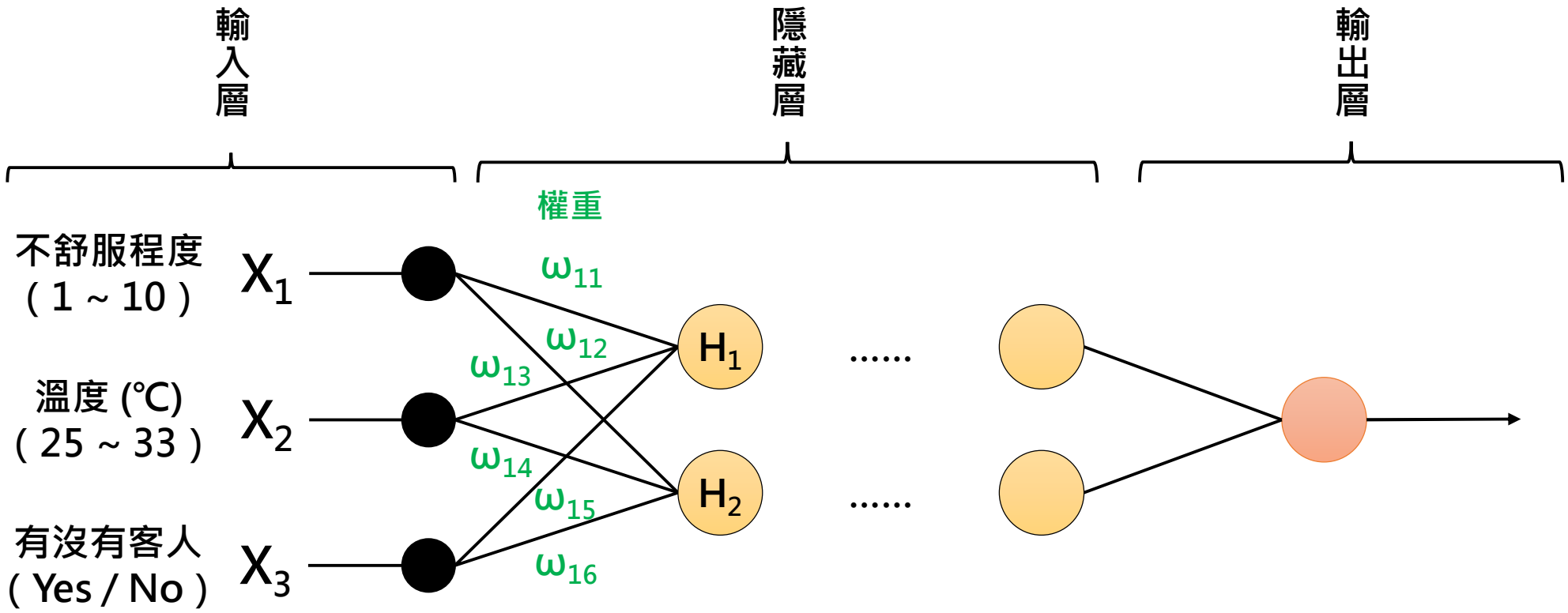
Principle of Multi-Layers Perceptron

多層感知器原理概說

- 多層感知器架構
- 隱藏層該有多少節點
- 隱藏層該有多少層
- 多層感知器的「學習方式」



多層感知器架構



隱藏層作用

- 增加抽象概念（H1：以不舒服程度為主。H2：以客人有無為主）
- 將模型提昇至「能解決線性不可分問題」的等級

隱藏層該有多少節點

- **沒有定論！常用的公式如下：**

公式一：算數平均數

$$\frac{\text{上一層節點數} + \text{下一層節點數}}{2}$$

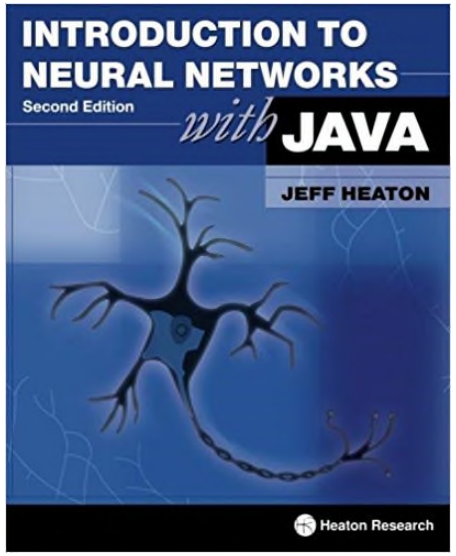
公式二：經驗法則公式

$$\frac{\text{樣本點個數}}{\alpha \times (\text{上一層節點數} + \text{下一層節點數})}$$

α=2~10（Scaling Factor）
（=2可防止過擬合，一般=5）

隱藏層該有多少層

- **沒有標準！但依據經驗法則，不需太多層就能解大部分的問題！**

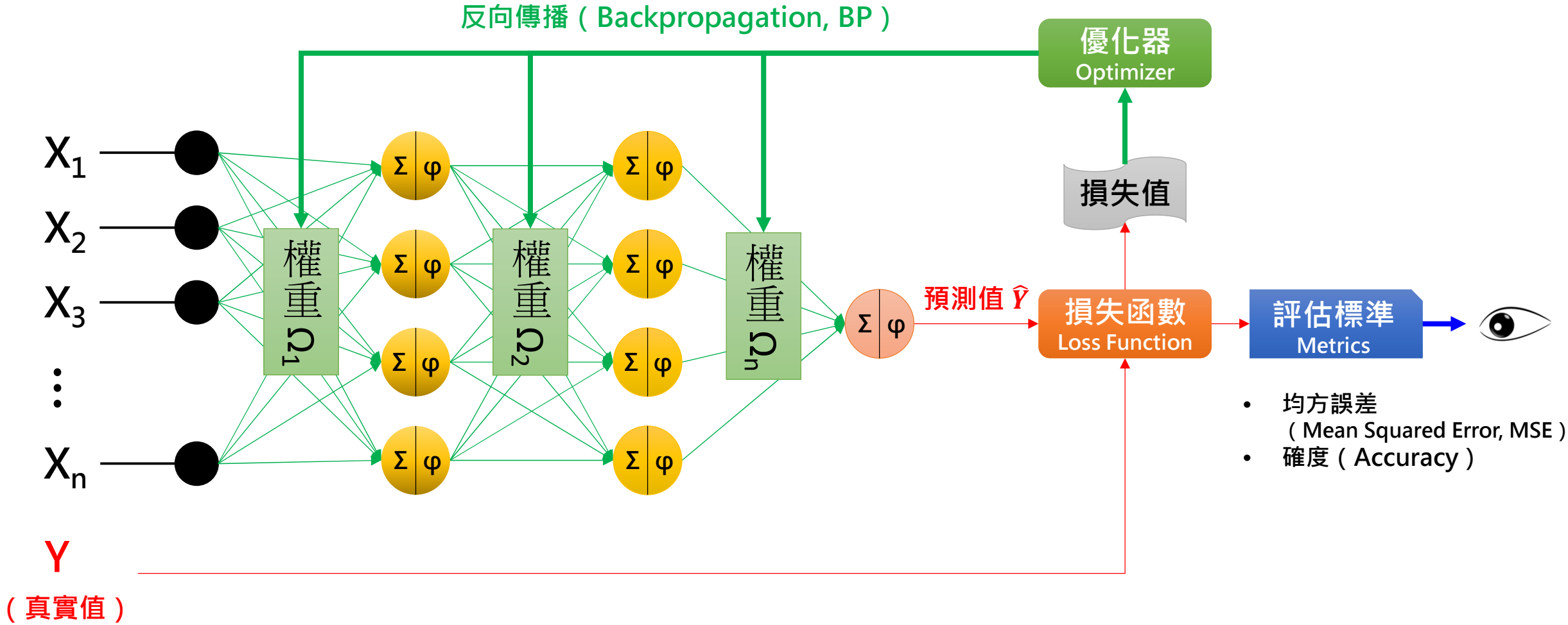


Introduction to Neural Networks with Java (2nd Ed.)

- **0 層**
  - 任何「線性可分」的題目都能解。
  - 等同「線性邏輯迴歸分類器」
- **1 層**
  - 任何「有限定義域」映射至「有限值域」的函數可分的都能解。
- **2 層**
  - 任何數學函數可分的都能解。

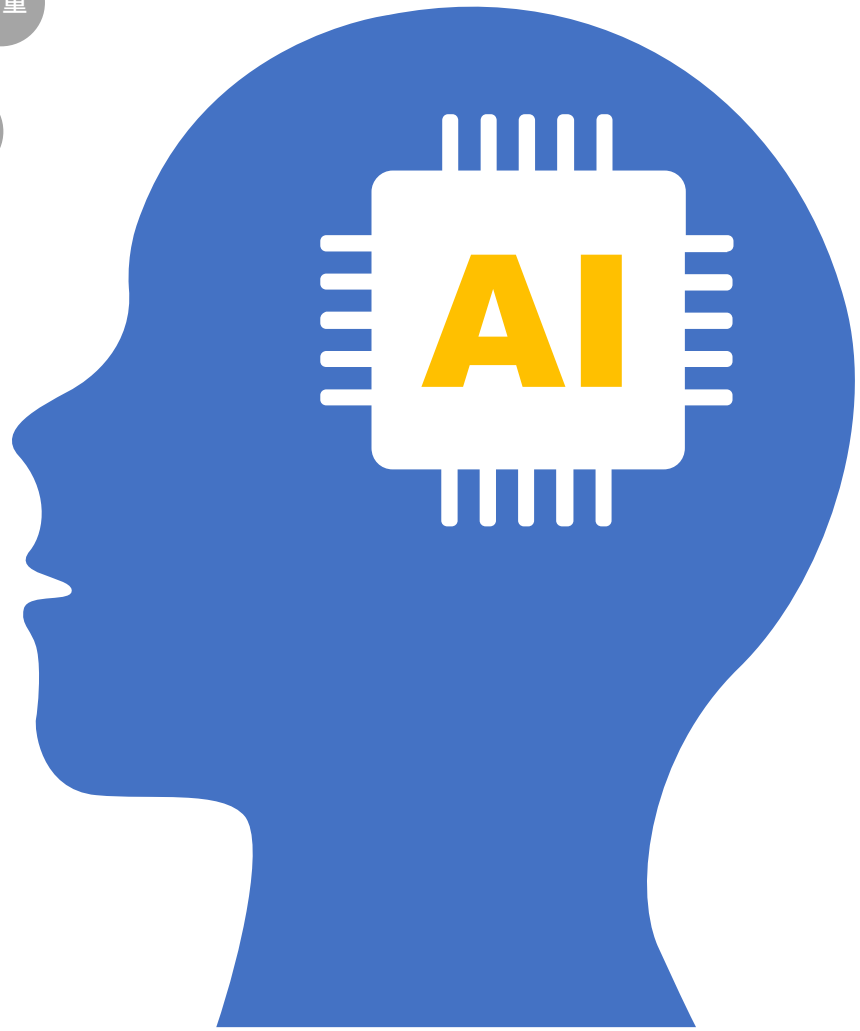
因此，在「淺層學習（Shallow Learning）」中，神經網路層數大多固定在 **2～3 層**。

多層感知器的「學習方式」


反向傳播（Backpropagation, BP）
優化器
Optimizer
損失值
X_1
X_2
X_3
X_n
權重 Ω_1
權重 Ω_2
權重 Ω_n
Σ φ
預測值 $\hat{Y}$
損失函數
Loss Function
評估標準
Metrics
均方誤差
（Mean Squared Error, MSE）
確度（Accuracy）
Y
（真實值）

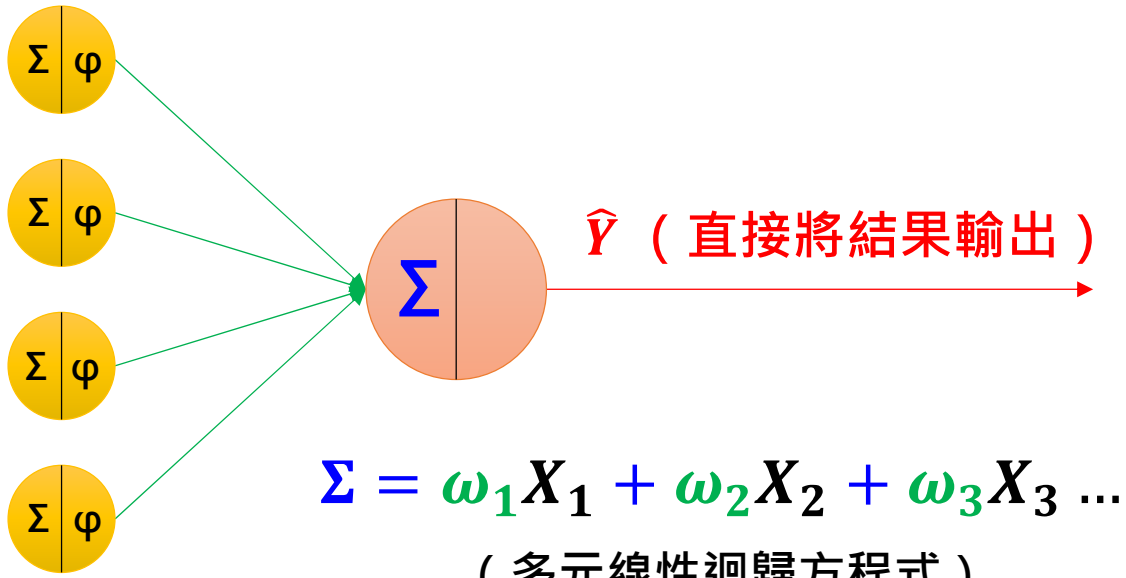
多層感知器五大元件

- 激活函數（Activation Functions）
- 權重初始器（Initializers）
- 損失函數（Loss Functions）
- 評估標準（Metrics）
- 優化器（Optimizers）



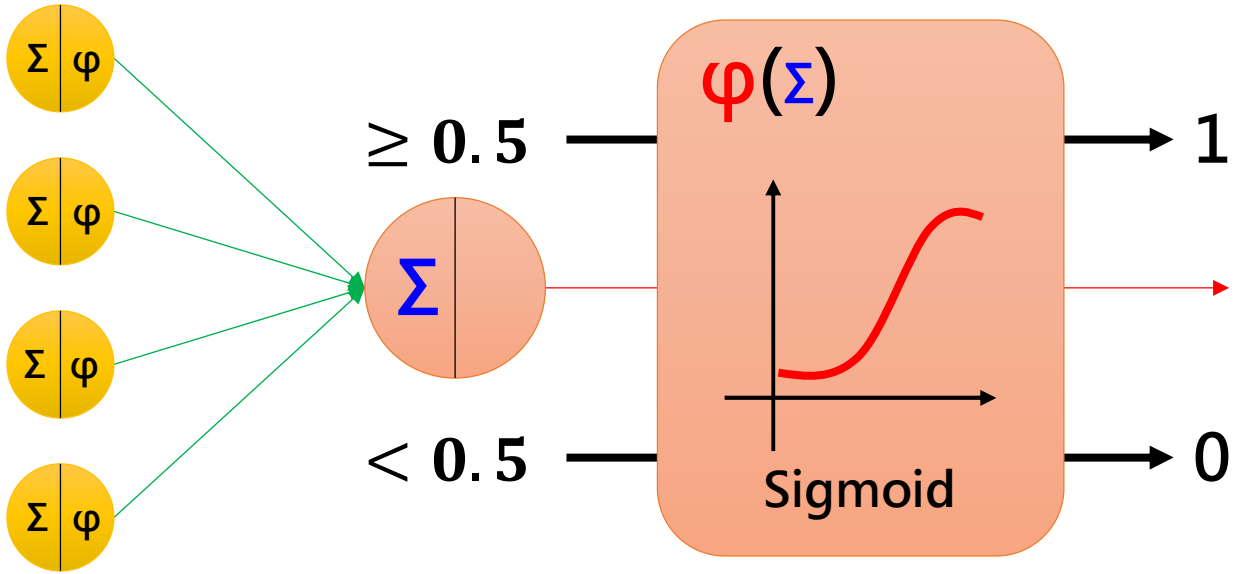
「激活函數」的選擇

- 「線性函數」（Linear）：迴歸問題使用。



「激活函數」的選擇

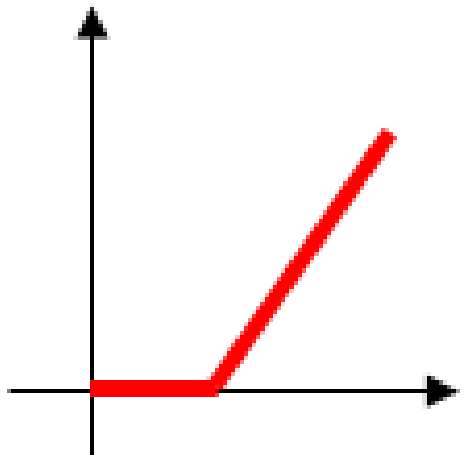
- Sigmoid 函數：輸出層一個節點、二選一時



「激活函數」的選擇

- 「線性整流函數」（Rectifier Linear Unit, ReLU）：隱藏層使用

$$ReLU = f(x) = \begin{cases} 0 & for\ x < 0 \\ x & for\ x \geq 0 \end{cases}$$



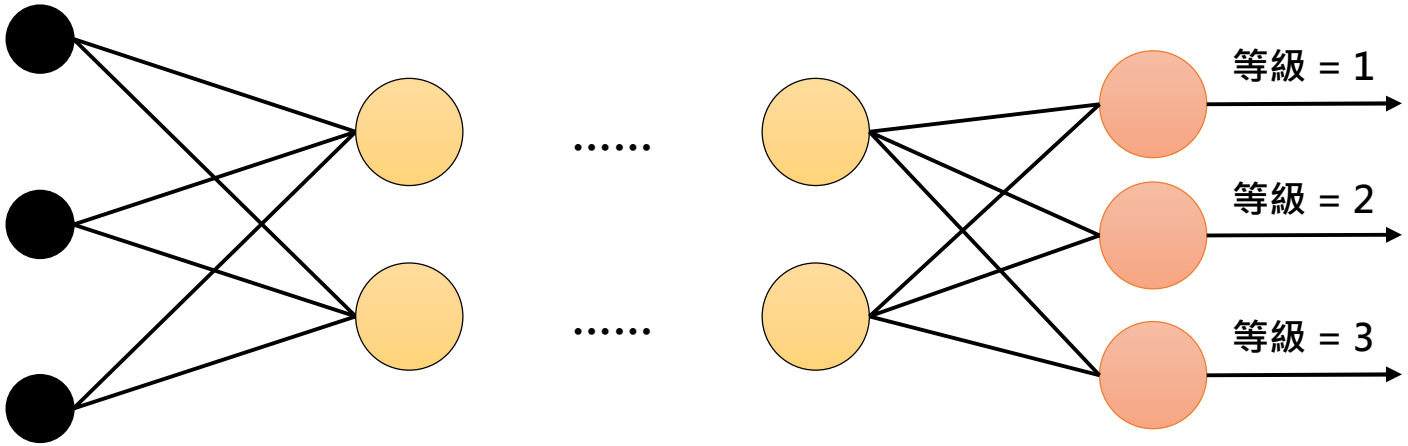
線性整流函數（ReLU）的好處：

- **解決「梯度消失問題」**
  - Sigmoid 的兩端是漸進線，會有「X 前進很多，Y 不太動」的問題。
  - 這會導致收斂末期，會收斂得很慢。稱為「梯度消失問題」。
- **直接切斷貢獻度小的神經元**
  - 一超過「閾值」，ReLU 會直接讓它變成 0。
  - Sigmoid 會「接近 0，但不等於 0」，仍殘留一點值，拖慢運算效能。
- **較貼近生物神經元「全有或全無」的特性**
  - 真正的生物神經元，低於「閾值」會直接不反應，ReLU 比較像。
  - Sigmoid 低於「閾值」，仍會保留很微弱的「殘值」。
- **節省計算量**
  - ReLU 的計算量比 Sigmoid 省，不必算 e^{-x}，效果又相近。

「激活函數」的選擇

- Softmax 函數：輸出層超過一個節點、多選一時

「紅酒評等」資料集



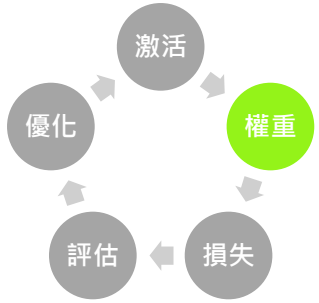
完整「激活函數」列表： https://bit.ly/2ZEHyDd

Softmax 函數

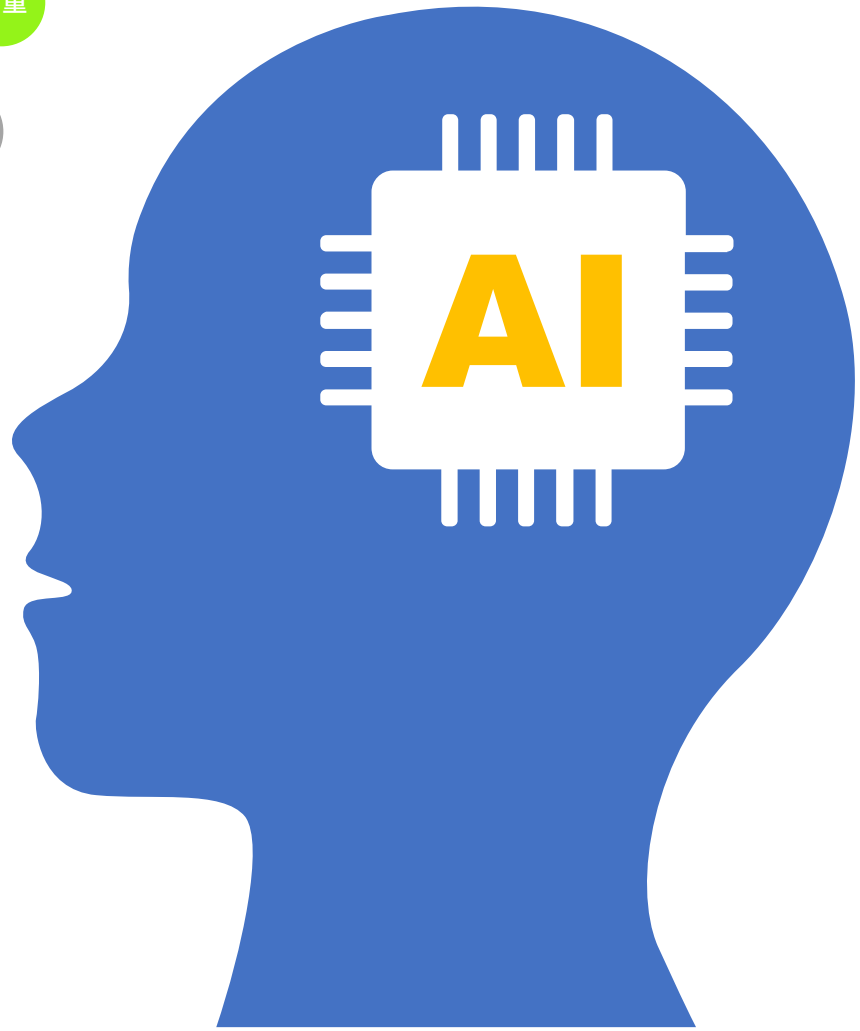
$$P(Y = j \mid X) = \frac{e^{X^T W_j}}{\sum_{k=1}^{n} e^{X^T W_k}}$$

- X: 特定自變數。如（35歲, 男性）
- P(Y=j|...): Y 分出來的答案是 1, 2, 3... 的機率
- $X^T W$: 所有自變數 x 所有權重
- $e^{X^T W}$：把 e^{-x} 從 $\frac{1}{1+e^{-x}}$ 簡化出來，用以代表 $X^T W_j$（j = 1, 2, 3...）發生之機率
- 假設 P(Y=1 | X)、P(Y=2 | X)、P(Y=3 | X)... P(Y=2 | X) 機率最高，則 Y=2 就該被激活。

多層感知器五大元件



- 激活函數（Activation Functions）
- **權重初始器（Initializers）**
- 損失函數（Loss Functions）
- 評估標準（Metrics）
- 優化器（Optimizers）



有哪些選擇？

常數型初始器

代表字串	函數	說明
“zeros”	Zeros()	全初始為 0
“ones”	Ones()	全初始為 1
"constant"	Constant(value=0)	全初始為特定常數

一般分佈型初始器

代表字串	函數	說明
"random_uniform"	RandomUniform(minval=-0.05, maxval=0.05)	從平均分佈 [minval, maxval) 抽樣
★ "random_normal"	RandomNormal(mean=0.0, stddev=0.05)	從常態分佈 (μ=mean, σ=stddev) 抽樣
★ "truncated_normal"	TruncatedNormal(mean=0.0, stddev=0.05)	同 “random_normal“。但 ±2σ（95%）以外的拋棄。

Glorot 分佈型初始器

代表字串	函數	說明
★ "glorot_uniform"	GlorotUniform()	從[-x, x] 抽，$x = \sqrt{6/(Fan_{in} + Fan_{out})}$
★ "glorot_normal"	GlorotNormal()	Truncated Normal(μ=0, $\sigma = \sqrt{2/(Fan_{in} + Fan_{out})}$
★ "variance_scaling"	VarianceScaling (distribution=“...", scale=..., mode=“...")	（後述）

★ ：常用

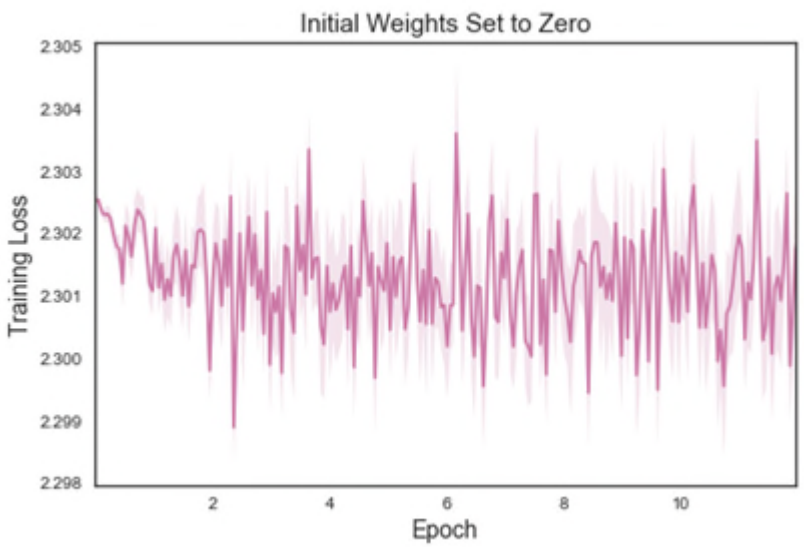
完整「權重初始函數（Intializer）」列表：
https://bit.ly/2OFuLdf

權重的初值，很重要嗎？

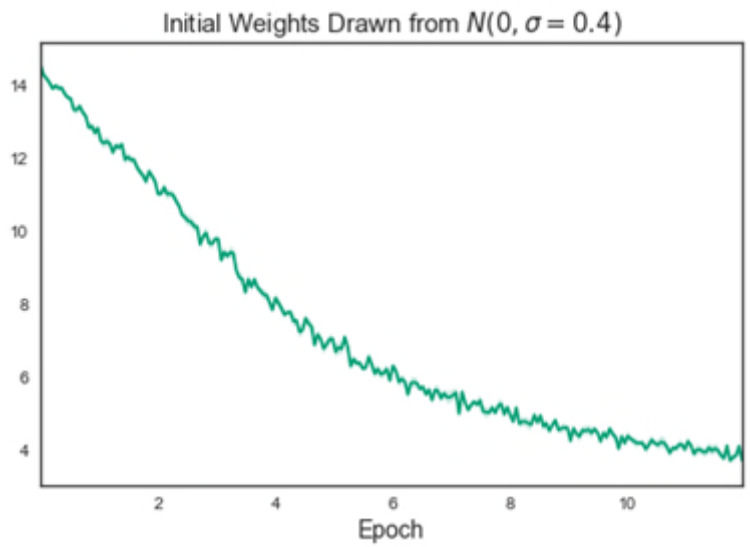
實驗方法說明：

MNIST 手寫數字資料集（60000 張圖片）、使用 CNN、
批次隨機梯度下降法（batch=128, Epochs=12）

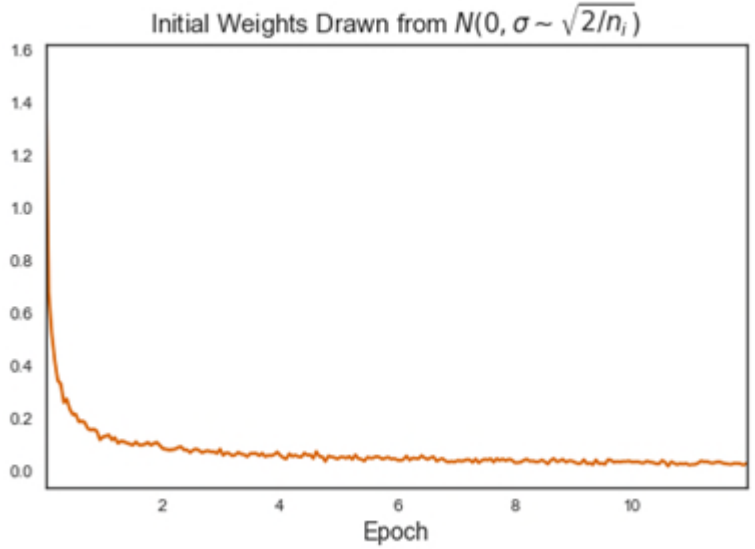
權重一樣 → 輸出類似
→ 梯度不明顯 → 收斂慢



權重初值 = Zeros



權重初值 = Random Normal
N（μ=0, σ=0.4）



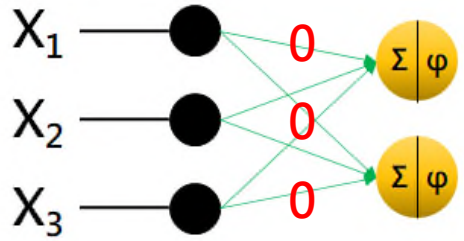
權重初值 = Glorot Normal
Truncated-N（μ=0, σ=$\sqrt{2/n}$）

「常數型」權重初始器

“zeros”

Zeros()

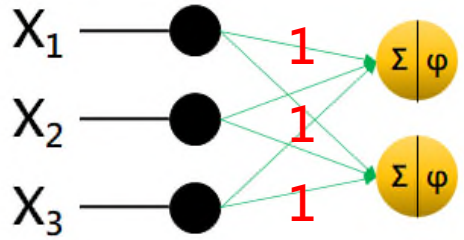
- 全初始為 0



“ones”

Ones()

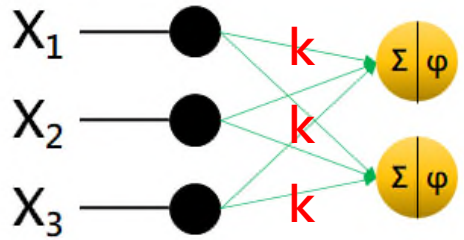
- 全初始為 1



“constant”

Constant(value=**k**)

- 全初始為常數 k

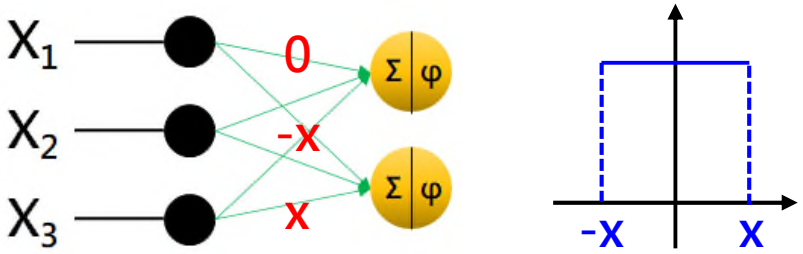


「一般分佈型」權重初始器

“random_uniform”

RandomUniform(minval=-x, maxval=x)

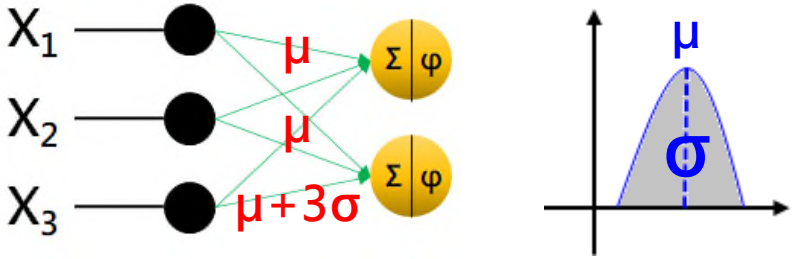
- 抽樣 ~ [-x, x]



“random_normal”

RandomNormal(mean= μ, stddev= σ)

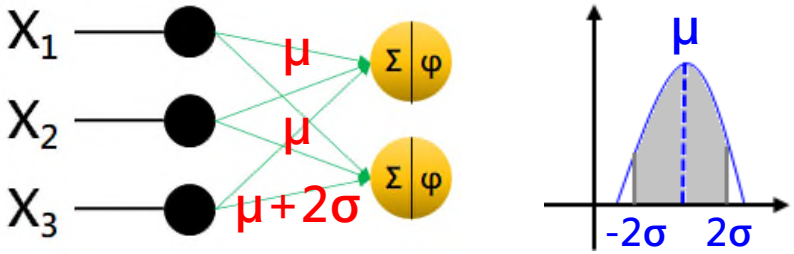
- 抽樣 ~ N(μ, σ)



“truncated_normal”

TruncatedNormal(mean= μ, stddev= σ)

- 抽樣 ~ N(μ, σ)
- ±2σ（95%）以外廢棄



「Glorot 分佈型」權重初始器

- 何謂「Glorot 分佈」？

Xavier Glorot
加拿大蒙特婁大學博士

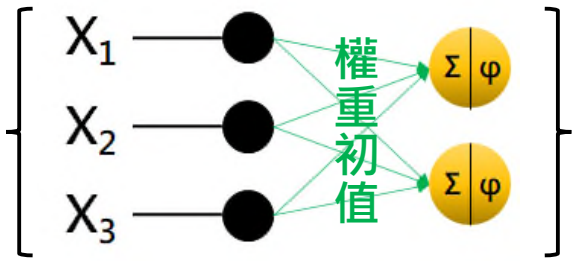
2010

Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot Yoshua Bengio
DIRO, Université de Montréal, Montréal, Québec, Canada

“最佳的權重初始值，與神經網路「輸入端點數」與「輸出端點數」有關！”

輸入端點數
（Fan-in）



輸出端點數
（Fan-out）

背後想法：

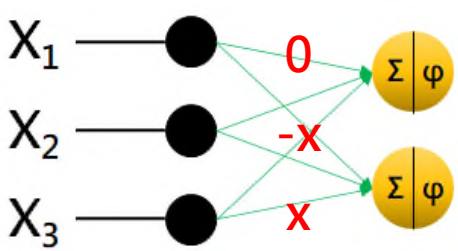
- 找到一種權重初值，能維持輸入端**權重分散程度** S_{in}，與輸出端**權重分散程度** S_{out} 類似。
- 只要**權重分散程度**不會越來**越小**（**集中**），就能維持**梯度**的**多樣性**，讓收斂**變快**。

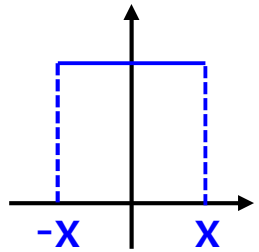
「Glorot 分佈型」權重初始器

“glorot_uniform”

GlorotUniform()

- 抽樣 ~ [-x, x]
- $x = \sqrt{6/(Fan_{in} + Fan_{out})}$

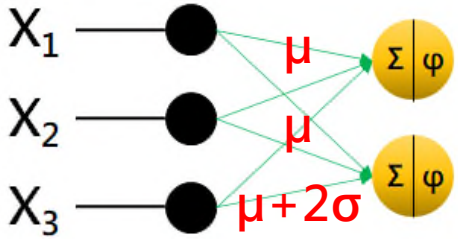


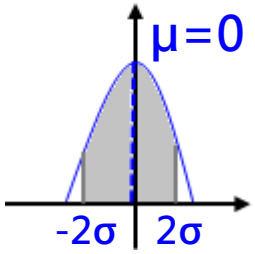


“glorot_normal”

GlorotNormal()

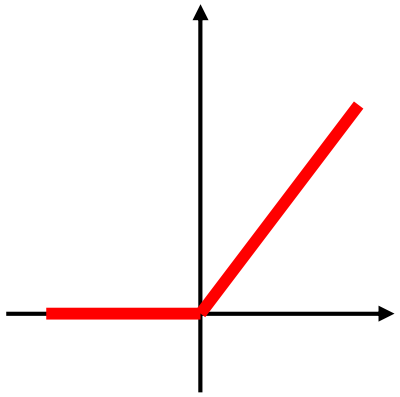
- 抽樣 ~ N(μ=0, $\sigma = \sqrt{2/(Fan_{in} + Fan_{out})}$)
- ±2σ（95%）以外廢棄





「Glorot 分佈型」權重初始器

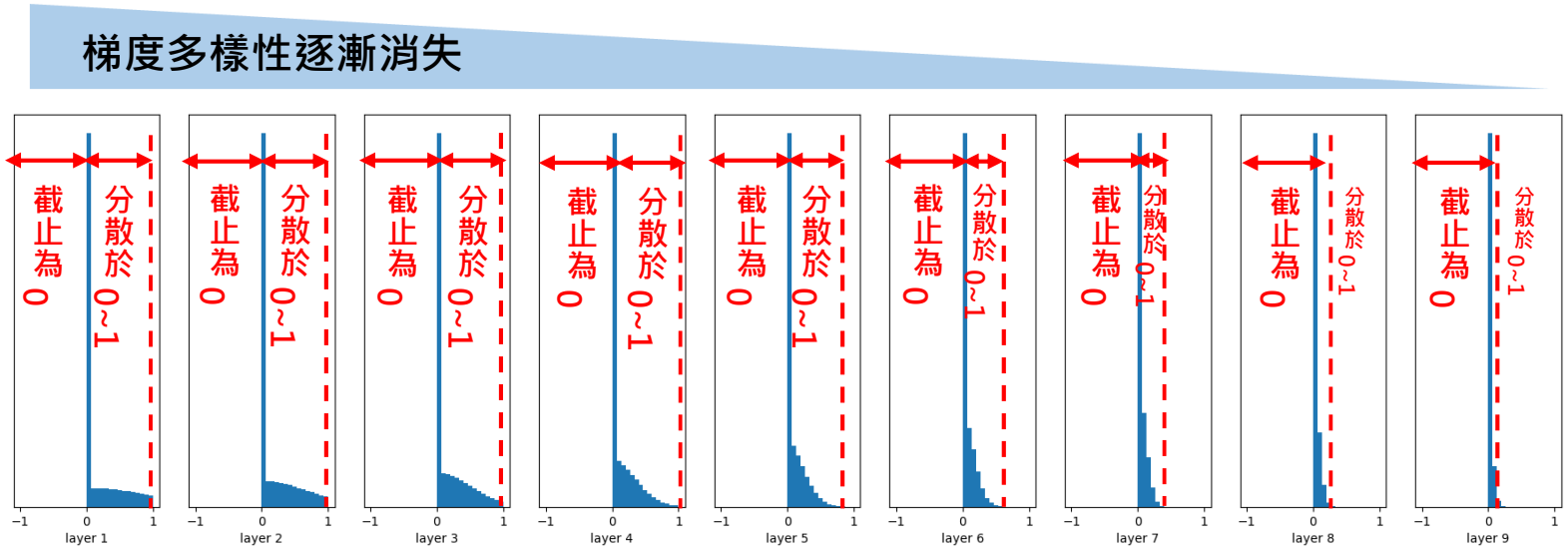
- **Glorot 分佈的缺點**
  - 在某些函數（如： ReLU）上適應不良！



- 每層平均有一半的梯度，會被截止為 0。
- 每層梯度的離散程度少一半
  → 下一層少一半的一半
  → 最後趨近於 0，多樣性消失

十層神經網路

採用 Glorot Normal + ReLU



「Glorot 分佈型」權重初始器

- 「何氏分佈」：改善 Glorot 會在 ReLU 梯度消失的問題

何愷明（Kaiming He）
Facebook / Microsoft AI Researcher

2015

Delving Deep into Rectifiers:
Surpassing Human-Level Performance on ImageNet Classification

Kaiming He　Xiangyu Zhang　Shaoqing Ren　Jian Sun
Microsoft Research

背後想法：

梯度少一半 → × 2 解決！

Glorot Normal

抽樣 ~ N(μ=0, $\sigma = \sqrt{\frac{2}{Fan_{in}+Fan_{out}}}$)

假設

Fan_{in} 與 Fan_{out} 的平均值是 n

$Fan_{in} + Fan_{out} = 2n$

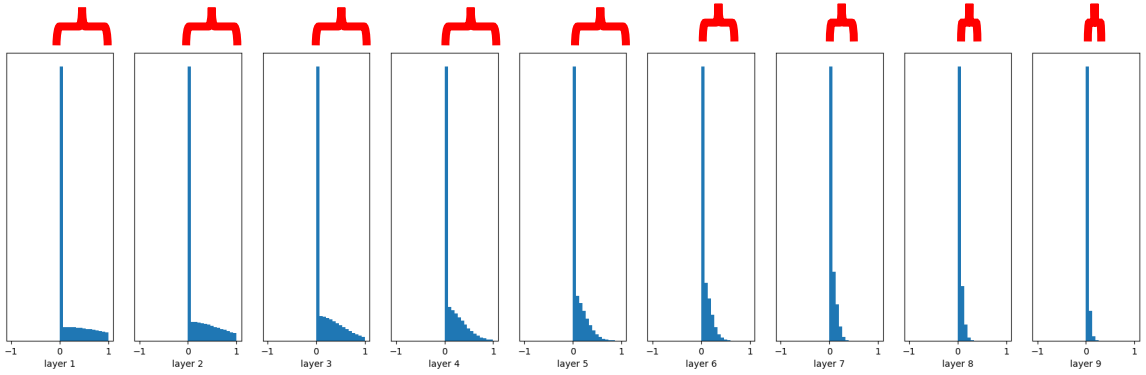
Glorot Normal (μ=0, $\sigma = \sqrt{\frac{2}{2n}} = \sqrt{\frac{1}{n}}$)

He Normal (μ=0, $\sigma = \sqrt{\frac{1\times2}{n}} = \sqrt{\frac{2}{n}}$)

「Glorot 分佈型」權重初始器

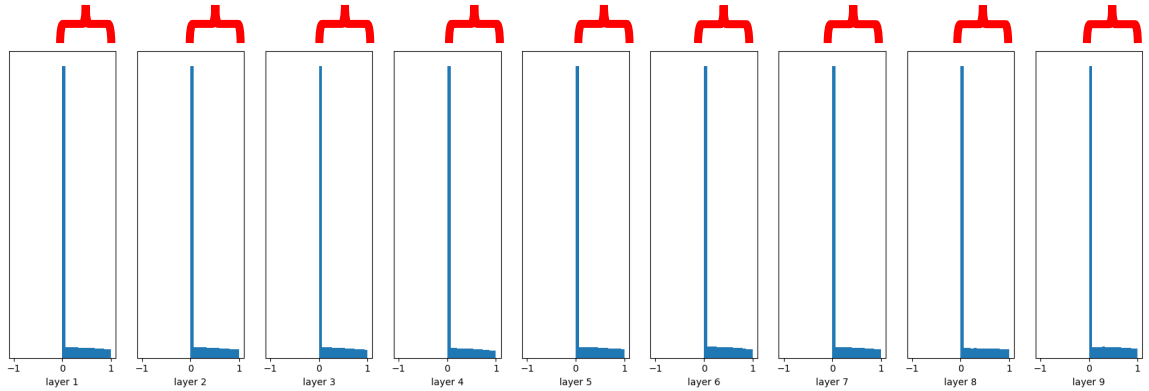
- 「Glorot Normal + ReLU」 vs. 「He Normal + ReLU」

梯度多樣性逐漸消失



Glorot Normal + ReLU

梯度多樣性保持豐富

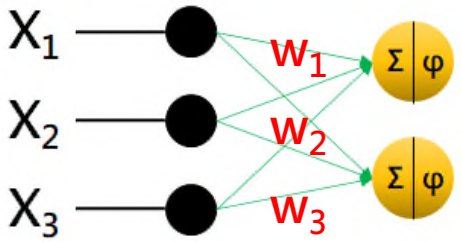


He Normal + ReLU

「Glorot 分佈型」權重初始器

“variance_scaling”

VarianceScaling (distribution=“...", scale=..., mode=“...")



$$\text{mode} = \begin{cases} "fan\_in" & n = n_{fan\_in} \\ "fan\_out" & n = n_{fan\_out} \end{cases}$$

$$\text{distribution} = \begin{cases} “uniform” & \text{抽樣} \sim [-x, x] \quad x = \sqrt{3 \times scale/n} \\ “untruncated\_normal” & \text{抽樣} \sim N(\mu = 0, \sigma = \sqrt{1 \times scale/n}) \\ “truncated\_normal” & \text{抽樣} \sim N\left(\mu = 0, \sigma = \sqrt{1 \times scale/n}\right) \end{cases}$$

±2σ（95%）以外廢棄

scale = 2，就是「**何氏分佈**」

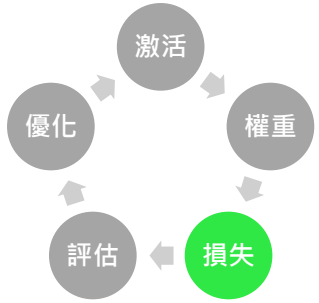
該如何挑選？

淺層學習
(Shallow Learning) → Random Normal、Truncated Normal

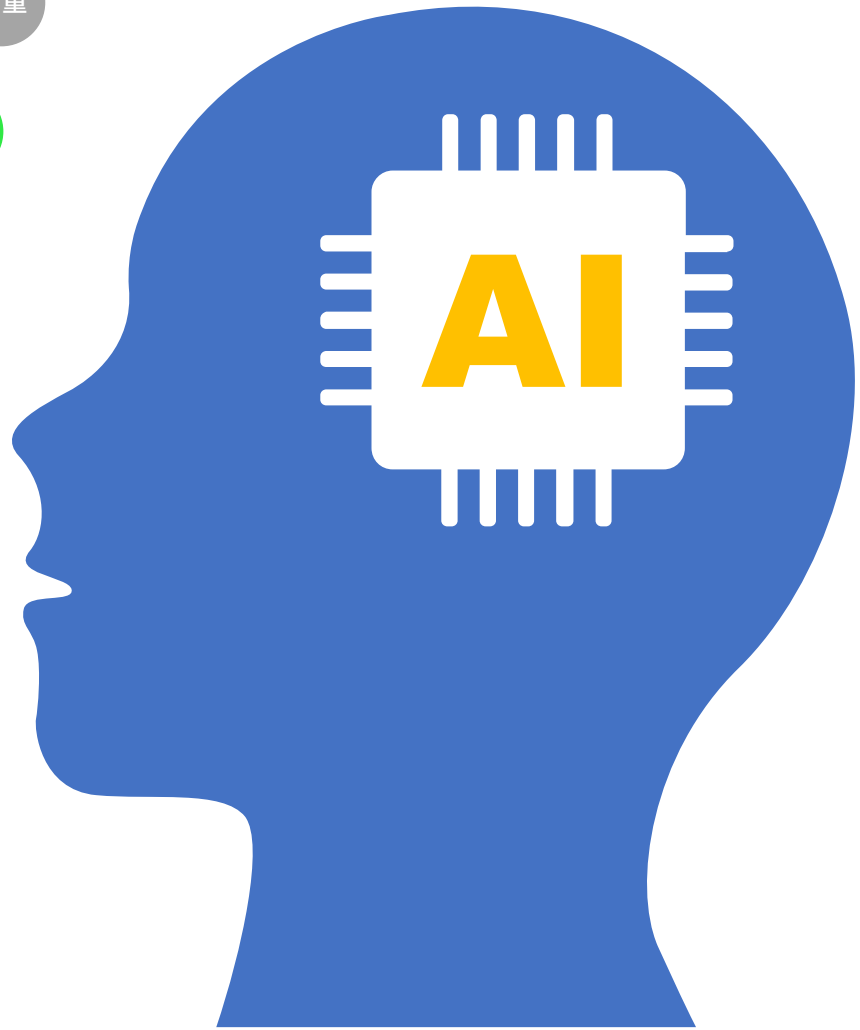
深度學習
(Deep Learning) → Glorot Normal

深度學習 + ReLU
(Deep Learning + ReLU) → Variance Scaling (scale = 2)

多層感知器五大元件



- 激活函數（Activation Functions）
- 權重初始器（Initializers）
- **損失函數（Loss Functions）**
- 評估標準（Metrics）
- 優化器（Optimizers）



「損失函數」的選擇

均方誤差
(Mean Squared Error, MSE)

$$MSE = \frac{\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}{n}$$

「迴歸問題」時使用

二元交叉熵
(Binary Cross Entropy)

$$H(p,q) = -p\log_2 q - (1-p)\log_2(1-q)$$

p: 事件真實發生機率　q: 事件預測發生機率

「二元分類問題」時使用

類別交叉熵
(Categorical Cross Entropy)

$$H(p,q) = -\sum_i p_i \log_2 q_i$$

p_i: 事件 i 真實發生機率　q_i: 事件 i 預測發生機率

「多元分類問題」時使用

完整「損失函數（Loss Functions）」列表：https://bit.ly/2OAk5g4

「損失函數」的選擇

- 何謂「交叉熵」（Cross Entropy）
  - 實際機率分布 vs. 預測機率分布的落差程度

Claude Shannon
1916-2001

資訊量
（資訊的稀有程度）

$$Info = -logP_i$$

$$\begin{cases} \text{東京下雪機率} = \dfrac{1}{10} \\ \text{台北下雪機率} = \dfrac{1}{1000} \end{cases}$$

$$\begin{cases} \text{東京下雪資訊量} = -log\dfrac{1}{10} = 1 \\ \text{台北下雪資訊量} = -log\dfrac{1}{1000} = 3 \end{cases}$$

資訊熵
（資訊亂度）
（資訊量的龐大程度）
（資訊量的期望值）

$$\text{資訊熵} = \sum_{i=1}^{n}(\text{資訊 } i \text{ 發生機率}) \times (\text{資訊 } i \text{ 資訊量})$$

$$= \sum_{i=1}^{n} Entropy_i = \sum_{i=1}^{n} P_i \times (-logP_i)$$

下雪的「資訊熵」

$$= Entropy(\text{東京}) + Entropy(\text{台北}) + \cdots$$

$$= \frac{1}{10} \times \left(-log\frac{1}{10}\right) + \frac{1}{1000} \times \left(-log\frac{1}{1000}\right) + \cdots$$

交叉熵
（實際 vs 預測的資訊量落差）

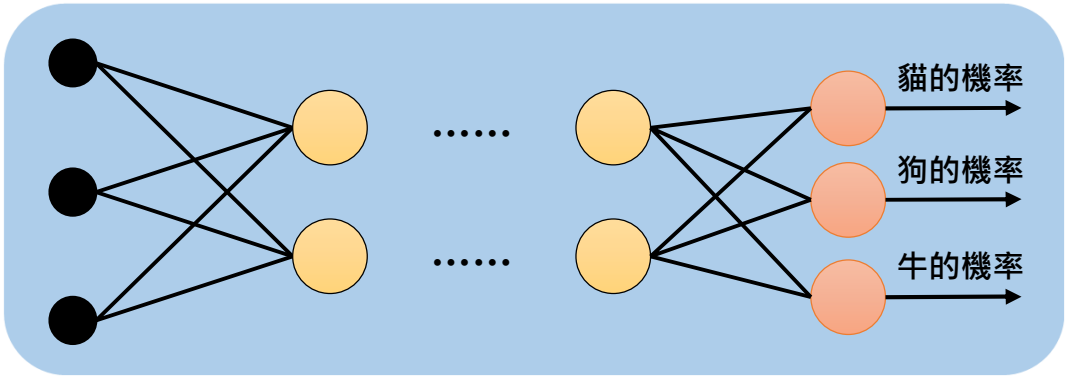
$$\text{交叉熵} = \sum_{i=1}^{n}(\text{資訊 } i \text{ 實際發生機率}) \times (\text{資訊 } i \text{ 預測資訊量}) = \sum_{i=1}^{n} P_i \times (-logQ_i)$$

「損失函數」的選擇

- 「交叉熵」的範例

種類	實際
貓	0
狗	1
牛	0

多選一神經網路分類器



種類	預測
貓	0.30
狗	0.45
牛	0.25

反向傳播/權重修正

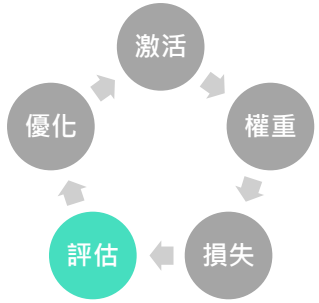
種類	實際	預測	交叉熵
貓	0	0.30	$0 \times (-\log 0.30) = 0$
狗	1	0.45	$1 \times (-\log 0.45) \cong 0.3469$
牛	0	0.25	$0 \times (-\log 0.25) = 0$
總交叉熵			0.3469

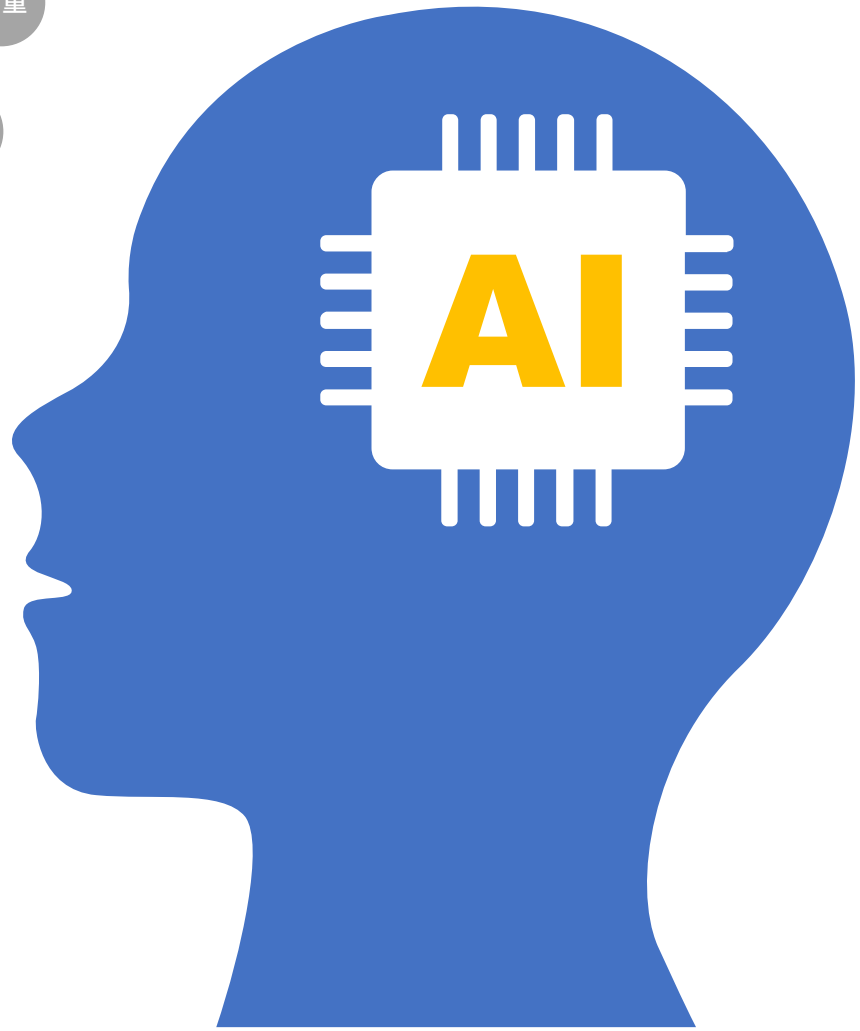
反向傳播/權重修正

種類	實際	預測	交叉熵
貓	0	0.20	$0 \times (-\log 0.20) = 0$
狗	1	0.75	$1 \times (-\log 0.75) \cong 0.1249$
牛	0	0.05	$0 \times (-\log 0.05) = 0$
總交叉熵			0.1249

種類	實際	預測	交叉熵
貓	0	0.00	$0 \times (-\log 0.00) = 0$
狗	1	1.00	$1 \times (-\log 1.00) \cong 0.00$
牛	0	0.00	$0 \times (-\log 0.00) = 0$
總交叉熵			0.0000

多層感知器五大元件





- 激活函數（Activation Functions）
- 權重初始器（Initializers）
- 損失函數（Loss Functions）
- **評估標準（Metrics）**
- 優化器（Optimizers）

「評估標準」的定義＆選擇

- **評估標準**
  - 讓人類得知，目前**這一期（Epoch）**的神經網路模型「**有多好**」

代表字串	底層類別	說明
“mean_squared_error” “mse”	MeanSquaredError()	均方誤差（Mean Squared Error）。 用於「迴歸問題」。
（無）	RootMeanSquaredError()	均方根誤差（Root Mean Squared Error）。 用於「迴歸問題」。
“accuracy” “acc”	Accuracy()	確度（Accuracy）。公式為 $(TN+TP)/ALL$。 用於「分類問題」。
（無）	Recall()	廣度（Recall）。公式為 $TP/(TP+FN)$。 用於「分類問題」。
（無）	Precision()	精度（Precision）。公式為 $TP/(TP+FP)$。 用於「分類問題」。

	$\hat{Y}=0$	$\hat{Y}=1$
$Y=0$	TN	FP
$Y=1$	FN	TP

完整「評估標準（Metrics）」列表： https://bit.ly/32zrdl4

為何沒有 F1-Score？

- 目前**有問題**，下架中
  - **問題點**：計算出來的 F1-Score 不正確！
  - 網址：https://bit.ly/32AQ73I

```
Epoch 98/100
610/610 [==============================] - 1s 1ms/step - loss: 2.1968e-10 - f1_score: 0.8371
Epoch 99/100
610/610 [==============================] - 1s 1ms/step - loss: 2.2058e-10 - f1_score: 0.8380
Epoch 100/100
610/610 [==============================] - 1s 969us/step - loss: 2.2145e-10 - f1_score: 0.8390
```

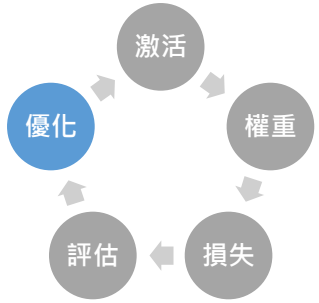
函式庫結果 vs. 手工計算

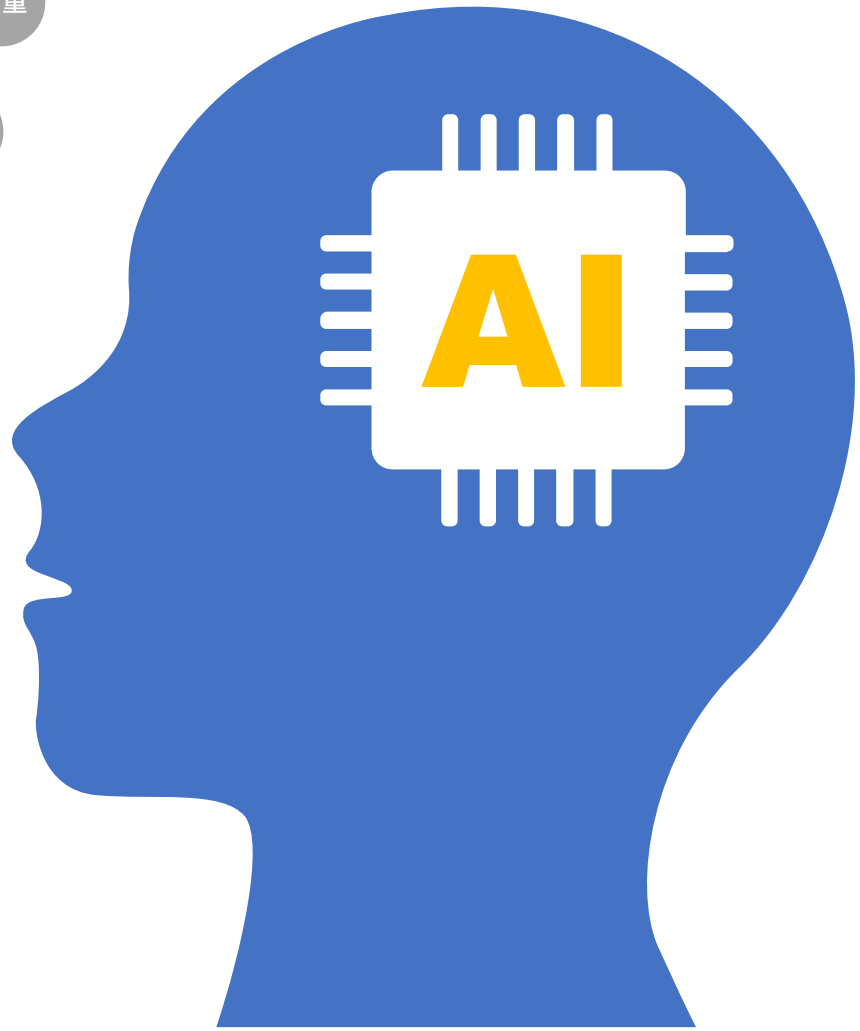
```
Confusion Matrix:
 [[1049    0]
 [   0  982]]
Accuracy: 100.00%
Recall: 100.00%
Precision: 100.00%
F1-score: 100.00%
```

- 有解決方案嗎？
  - 2020/02/17 已有人提出「Feture Request」
  - 目前因為提問人久未回應，暫時關閉中（Status: Open → Closed）
  - 網址： https://bit.ly/3jlKJr0

- 想試試看 F1-Score 到底有多不正確
  - 安裝： pip install tensorflow-addons
  - 引入： import tensorflow_addons as tfa
  - 使用：model.compile(... metrics=[tfa.metrics.F1Score(num_classes=2, average= “micro”)])
    - num_classes=2：代表最終答案有兩類（分類二選一問題）
    - average：求平均的方法（micro: 微觀平均、macro: 巨觀平均）

多層感知器五大元件



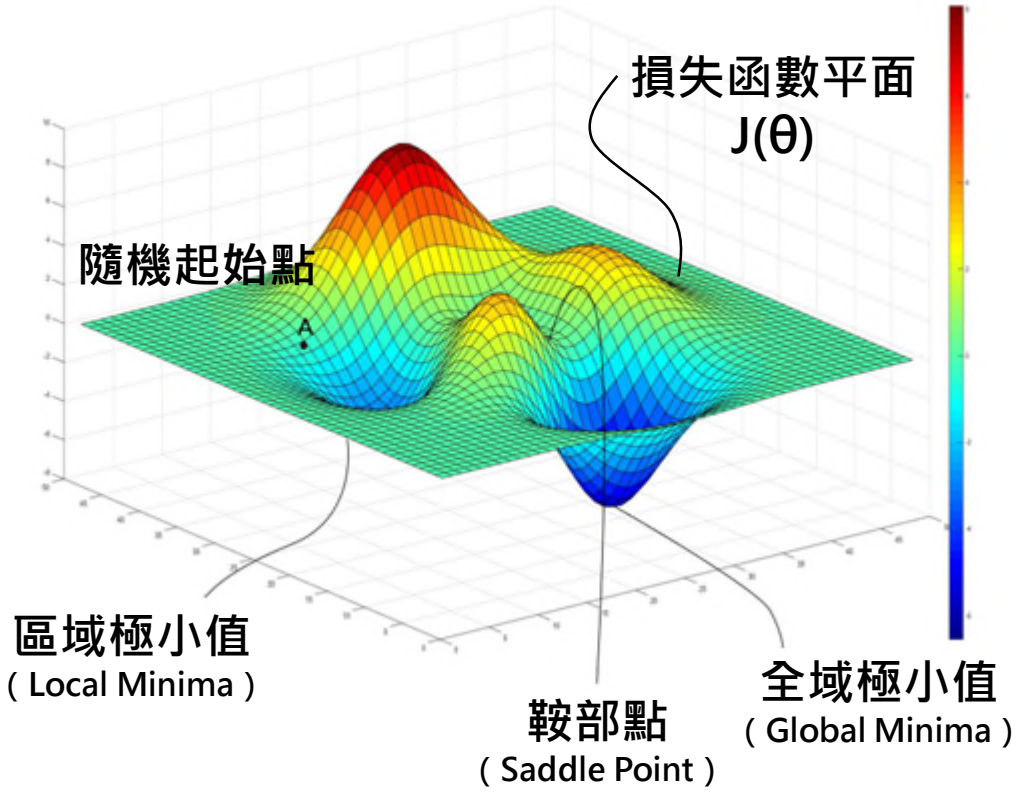


- 激活函數（Activation Functions）
- 權重初始器（Initializers）
- 損失函數（Loss Functions）
- 評估標準（Metrics）
- **優化器（Optimizers）**

「優化器」的選擇

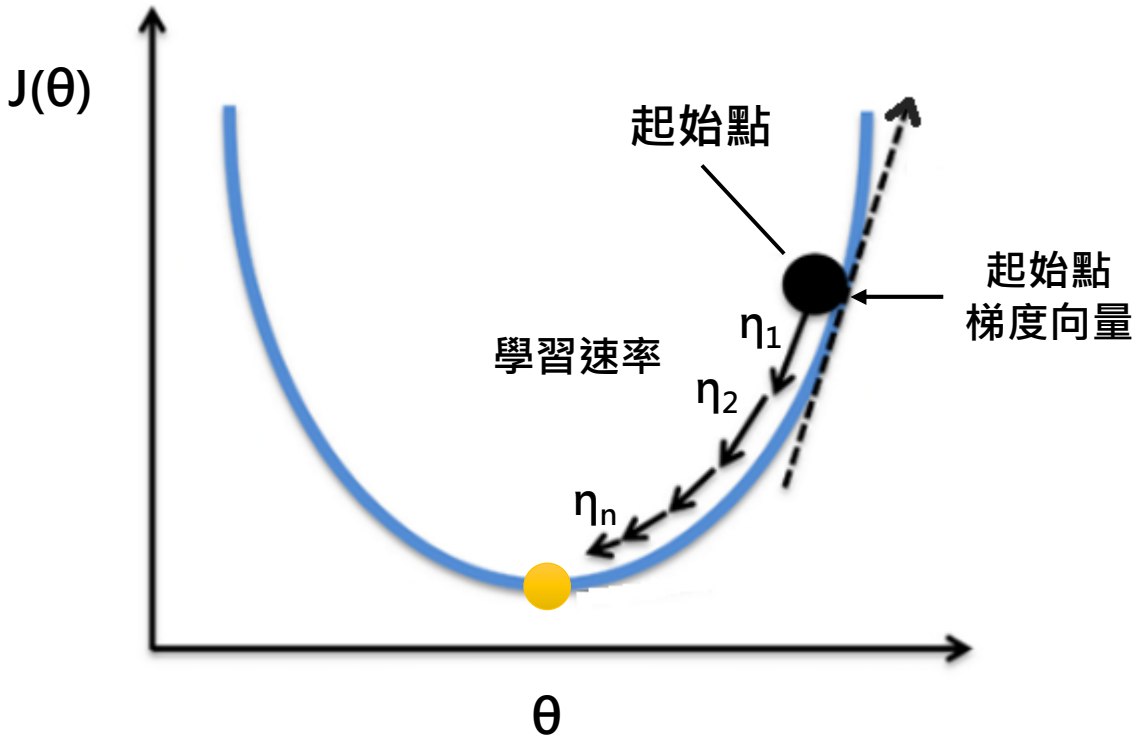
- 「優化器（Optimizers）」的任務
  - 找到「損失函數 J」的極小值
  - 更新「權重 θ」

- **全域極小值**
  - 我們要找的目標。
- **區域極小值**
  - 不小心找到的假目標。
- **鞍部點**
  - 可能會被困住的點。



「優化器」的選擇

- 「優化器」如何找極小值
  - 梯度下降法（Gradient Descent, GD）



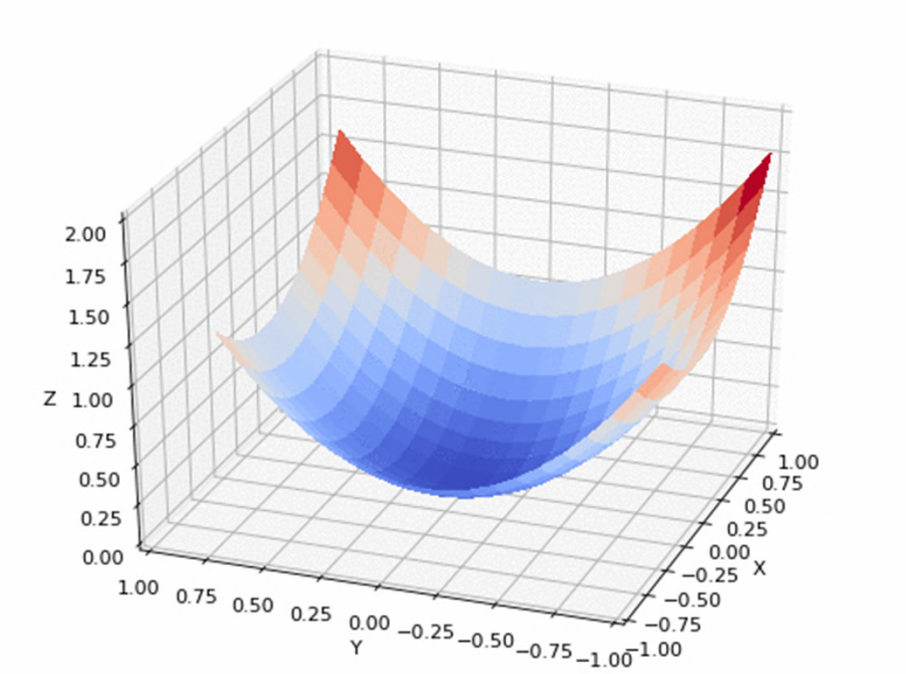
- **起始點**
  - 隨機任意選取的一個點。
- **梯度（Gradient）**
  - 總是指向「函數最大增加方向」的向量。
  - **梯度** = 函數 f 在各分量的偏微分。

$$\nabla f(x_1, x_2) = \left(\frac{\partial}{\partial x_1} f(x_1, x_2) \quad \frac{\partial}{\partial x_2} f(x_1, x_2) \right)$$

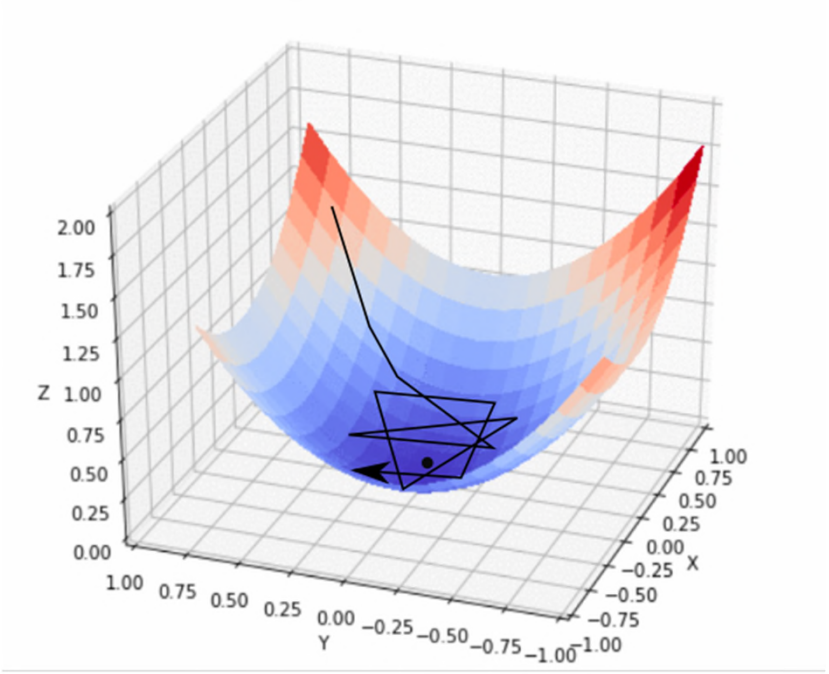
  - 概念類似「某一點的切線斜率（微分）」，只不過它是向量（除了大小，還有方向）。
  - 若梯度→0，代表接近水平，找到**終點**。
- **學習速率（Learning Rate, η (Eta)）**
  - 往目標走一步的距離。
  - 梯度越斜→學習速率應越大。反之越小。否則會出現在**終點**附近折返跑的現象。

「優化器」的選擇

- 「學習速率 η」與「收斂速度」的關係



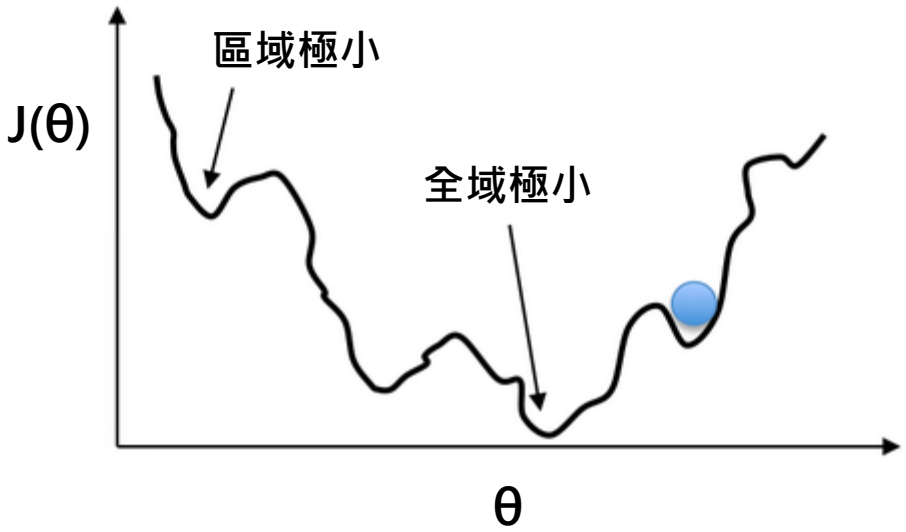
「學習速率」隨著梯度減小
（容易收斂）



「學習速率」一直固定不變
（不易收斂）

「優化器」的選擇

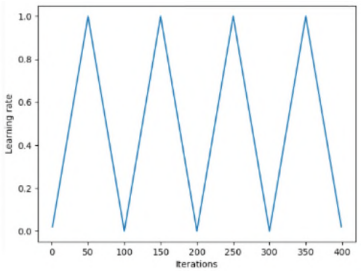
- 梯度下降法的問題（１）：區域極小問題（Local Minima）



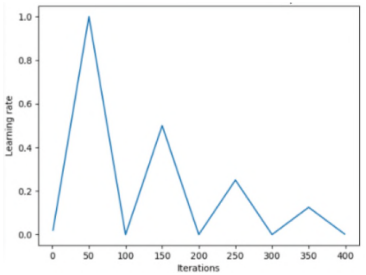
解決方法

讓「學習速率 η」定期循環跳動
（Cyclic Learning Rate Scheduling）

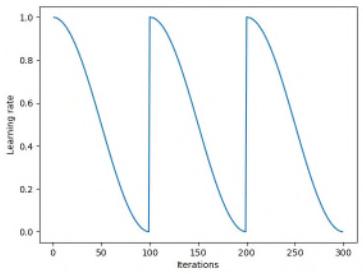
一稱「退火」（Annealing）



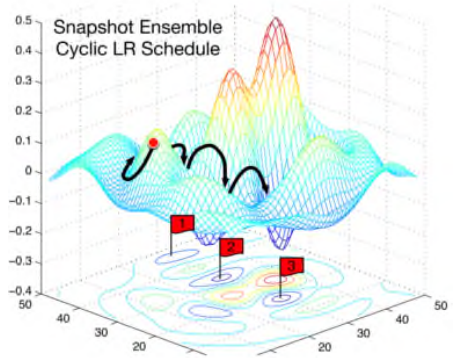
三角法



三角衰減法



餘弦衰減法

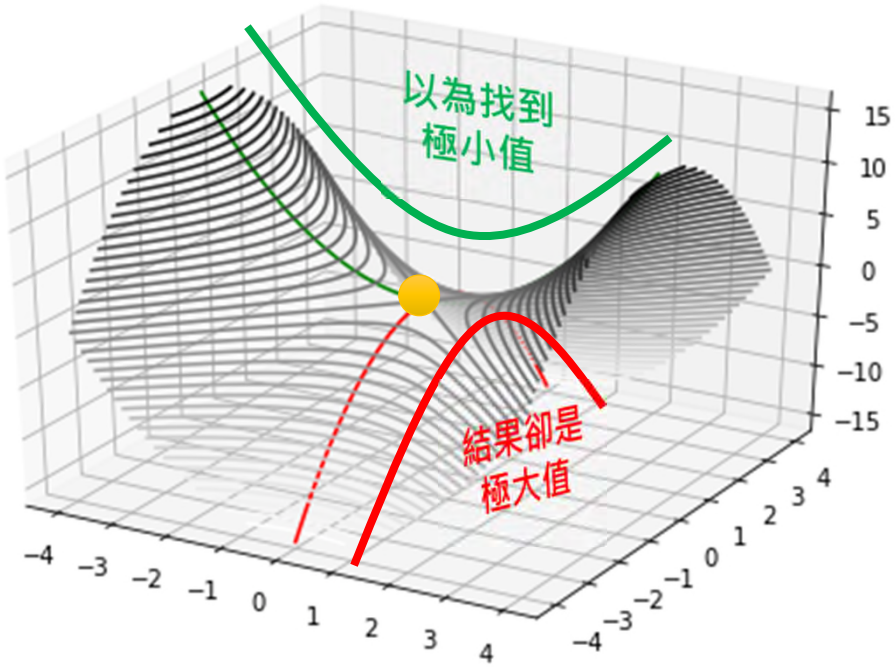


「學習速率 η」⇧

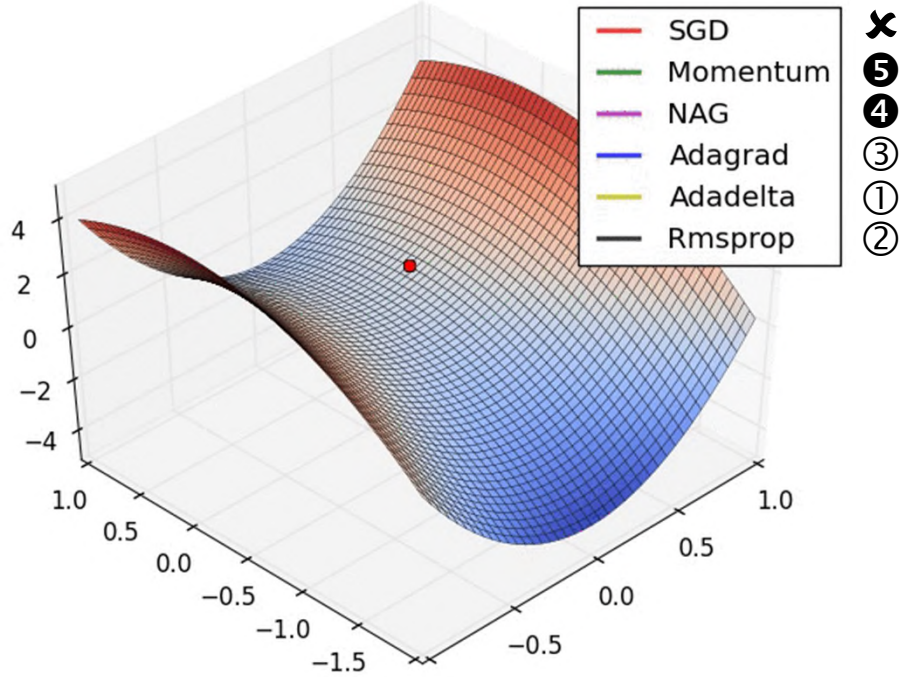
→ 容易跳出山谷
→ 找到多個極小值
→ 取最小的那一個

「優化器」的選擇

- 梯度下降法的問題（２）：鞍部點問題（Saddle Point）



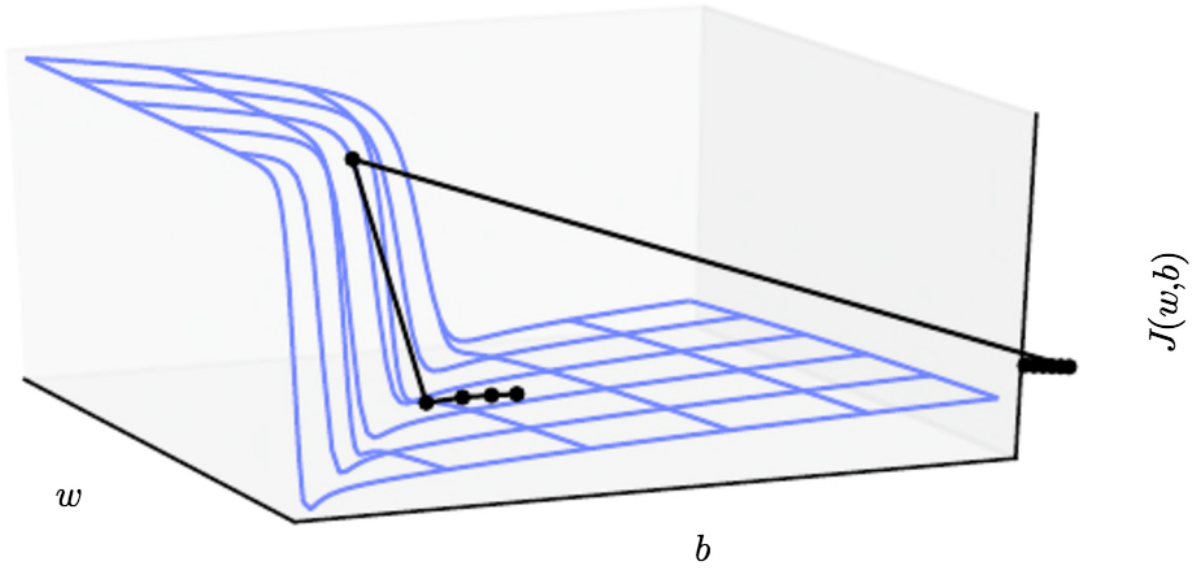
解決方法　使用一些改進過的梯度下降演算法



「優化器」的選擇

- **梯度下降法的問題（3）：梯度懸崖問題（Gradient Cliff）**

（一稱梯度爆炸（Gradient Exploding）問題）



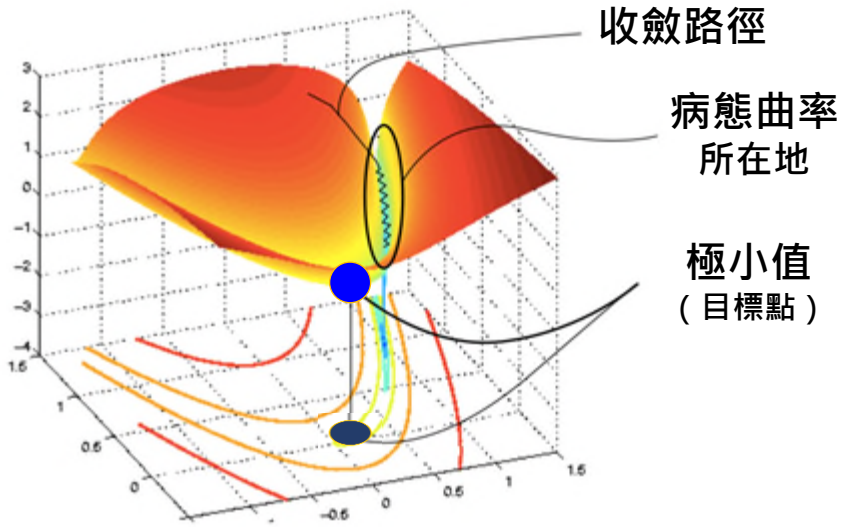
梯度在「懸崖邊」，突然一下子增大！
指向遠方，讓快要收斂的結果，功虧一簣！

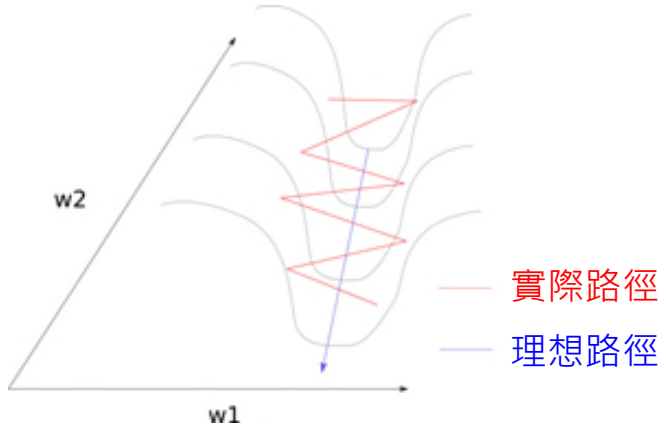
解決方法　使用「梯度截斷法」（Gradient Cut-Off）

梯度超過一定「閾值」
→ **拋棄不用！**

「優化器」的選擇

- 梯度下降法的問題（４）：病態曲率問題（Pathological Curvature）



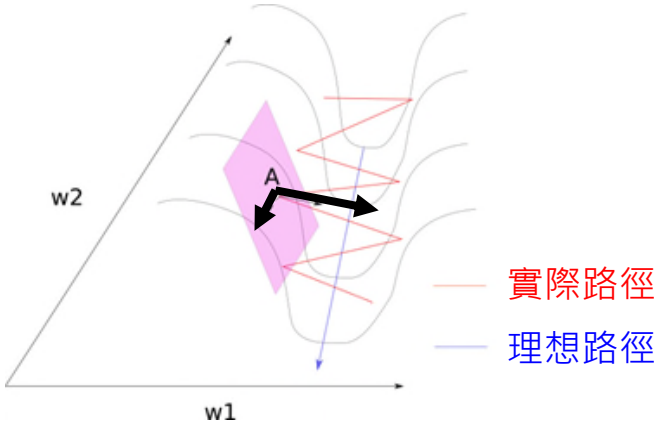


因為山谷太窄，就算 學習速率 η 已經縮到很小了，
仍然會產生「反覆橫跳」、浪費效能的現象。

「優化器」的選擇

- **梯度下降法的問題（4）：病態曲率問題（Pathological Curvature）**

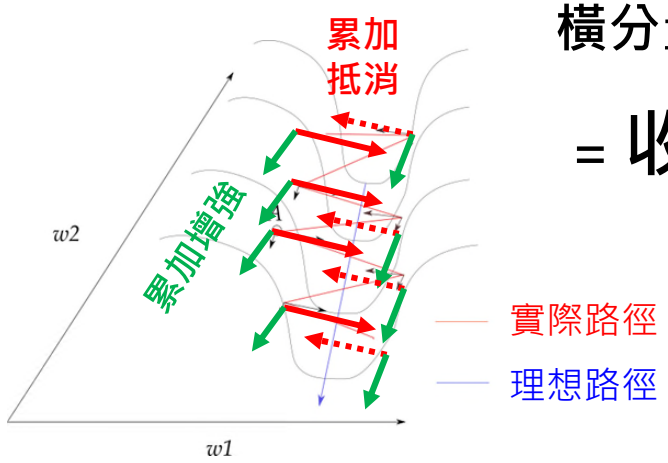
反覆橫跳的原因



梯度**橫方向**分量 > **縱方向**分量

解決方法　使用「**動量法**」（Momentum）

累加歷史上出現過的梯度**各分量**，
並將它用於**下一次**的梯度下降**距離**。



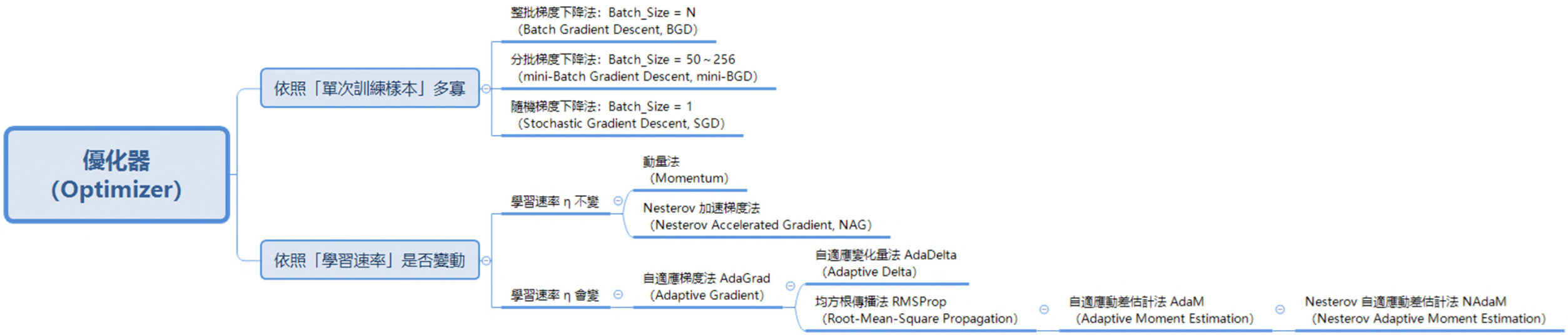
橫分量漸減 + 縱分量漸增

= 收斂更快！

「優化器」的選擇

- 有哪些「優化器」可用？

完整「優化器」列表： https://bit.ly/2CLkEkw



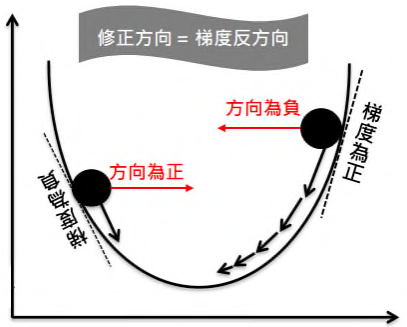
「優化器」的選擇

- **Keras 內常見的「優化器」（1）**
  - **隨機梯度下降法**（Stochastic Gradient Descent, **SGD**）

SGD
- AdaGrad
  - AdaDelta
    - RMSProp
      - AdaM
        - NAdaM

- 底層類別： "**sgd**" = **SGD**(learning_rate=0.01, momentum=0.0, nesterov=False)
- 學習速率： **固定**（手工輸入）。**預設值** = η = 0.01
- 注意事項： **Keras** 內的 **SGD** 事實上是使用 **mini-Batch Gradient Descent** 演算法。
- 數學原理： **下次權重** θ_{t+1} = **現在權重** θ_t + **學習速率** η × **現在梯度** ∇ 的反方向 (-1)

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t;\ X_{i\sim(i+n)}, Y_{i\sim(i+n)})$$ 常簡寫成 $$\Delta\theta = -\eta \nabla J(\theta)$$



現在梯度 $\nabla J()$ = { 現在權重 θ_t 與 自變數 X_i, 應變數 Y_i 開始的 n 個樣本點（小批次權重更新） } 取**梯度**

缺點：η 固定
- 易只找到局部極小。
- 終點前反覆橫跳。
- 易受騙困於鞍部點。
- 病態曲率時易震盪。

「優化器」的選擇

- Keras 內常見的「優化器」（2）
  - 隨機梯度下降 + 動量法（SGD + Momentum）　通常是 0.9
    - 底層類別：SGD(learning_rate=0.01, momentum=[0, 1) 之數, nesterov=False)
    - 學習速率：固定（手工輸入）。預設值 = η = 0.01
    - 解決問題：逃離區域極小、逃離鞍部點、減低病態曲率處反覆震盪。
    - 數學原理：$\theta_{t+1} = \theta_t + \boldsymbol{v_t}$　（下次權重 θ_{t+1} = 現在權重 θ_t + 現在動量 v_t）

$$v_t = \gamma \boldsymbol{v_{t-1}} - \eta \nabla J(\theta_t)$$

（現在動量 v_t = 上次動量 v_{t-1} × 衰減率 γ
+學習速率 η × 現在梯度 ∇ 的反方向 (-1)）

- SGD
  - AdaGrad
    - AdaDelta
      - RMSProp
        - AdaM
          - NAdaM

推導：

動量 P = 質量 m × 速度 v

設神經網路中，質量 m =1

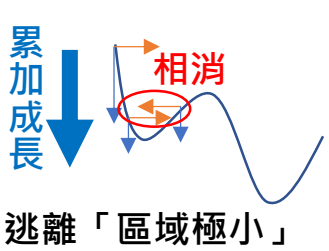
動量 P = 速度 v　再設「力」=修正方向（-學習速率×梯度）

動量 = 速度 v_t = 前速 v_{t-1} ×衰減率γ+作用力 $(-\eta \nabla J(\theta_t))$

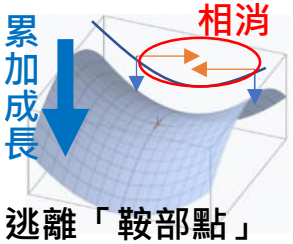
$v_t = \gamma \boldsymbol{v_{t-1}} - \eta \nabla J(\theta_t)$　γ=0.9, $G_0 = -\eta \nabla J(\theta_0)$

$v_1 = G_0$, $v_2 = 0.9G_0 + G_1$, $v_3 = 0.81G_0 + 0.9G_1 + G_2$, ...

v_i 有記憶效應！擺盪方向會累加相消！



逃離「區域極小」



逃離「鞍部點」

「優化器」的選擇

- Keras 內常見的「優化器」（3）
  - Nesterov 加速梯度法（SGD + Nesterov Accelerated Gradient, NAG）

SGD
- AdaGrad
  - AdaDelta
    - RMSProp
      - AdaM
        - NAdaM

- 底層類別：SGD(learning_rate=0.01, momentum=[0, 1), nesterov=True)
- 學習速率：固定（手工輸入）。預設值 = η = 0.01
- 解決問題：同動量法 + 收斂更快。
- 數學原理：$\theta_{t+1} = \theta_t + \boldsymbol{v_t}$　$\boldsymbol{v_t} = \gamma \boldsymbol{v_{t-1}} - \eta \nabla J(\theta_t)$

$\theta_{t+1} = \theta_t + \gamma \boldsymbol{v_{t-1}} - \eta \nabla J(\theta_t)$　無論如何 $\gamma \boldsymbol{v_{t-1}}$ 一定會加入到 θ_t 裡，成為 θ_{t+1} 的一部分

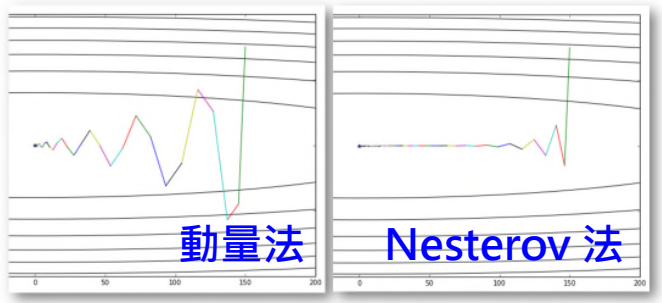
那就乾脆偷跑半步，求 $\nabla J(\theta_t + \gamma \boldsymbol{v_{t-1}})$ 不是更能預見未來、收斂更快嗎？

Yurii Nesterov
(1956-)

Nesterov 公式：

$$\theta_{t+1} = \theta_t + \boldsymbol{v_t}$$

$$\boldsymbol{v_t} = \gamma \boldsymbol{v_{t-1}} - \eta \nabla J(\theta_t + \gamma \boldsymbol{v_{t-1}})$$



「優化器」的選擇

- **Keras 內常見的「優化器」（4）**
  - **自適應梯度法（AdaGrad: Adaptive Gradient）**

SGD
AdaGrad
AdaDelta
RMSProp
AdaM
NAdaM

- 底層類別： "adagrad" = Adagrad(learning_rate=0.001, initial_accumulator_value=0.1, epsilon=1e-07)
- 學習速率： **變動**（梯度大→ η 大，梯度小→ η 小）。**預設值** = η = 0.001
- 解決問題： 讓**收斂**接近**終點**時，**學習速率**可以自動**縮小**。
- 數學原理：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\sum_{k=1}^{t}\left(\nabla J(\theta_t)\right)^2 + \varepsilon}}\nabla J(\theta_t)$$

效果
- 剛開始：梯度累積不多（小）→ η 大
- 結束前：梯度累積很多（大）→ η 小

$\sqrt{\sum_{k=1}^{t}\left(\nabla J(\theta_t)\right)^2}$：過往梯度**平方總和** + **開根號**

平方：不讓正負梯度互相影響
開根號：回復原先的數量級

ε：為了防止梯度=0 時分母=0。一般=10^{-7}左右。

缺點：
- 即將收斂結束時，會因為**梯度平方**永為**正**，累積太多，導致 η 過小，影響**收斂速度**。

「優化器」的選擇

- Keras 內常見的「優化器」（5）
  - AdaDelta

SGD
AdaGrad
AdaDelta
RMSProp
AdaM
NAdaM

- 底層類別： "adadelta" = Adadelta(learning_rate=0.001, rho=0.95, epsilon=1e-07)
- 學習速率： **變動**（梯度大→ η 大，梯度小→ η 小）。**預設值** = η = 0.001
- 解決問題： 修正 AdaGrad 中後段因為**學習速率**變小，而**收斂減緩**的問題。
- 解法巧思： (1) 累加「**過往梯度**平方」時，要讓歷史**越久遠**的梯度，影響力**越小**。

  (2) 引入**類似**「動量」概念，累加過往的**更新值**。再加**越久遠**、影響**越小**的效果
  （如此才能「**壞方向**累加**相消**、**好方向**累加**增進**」！）
- 數學公式： $\theta_{t+1} = \theta_t + \Delta X_t$ （ ΔX_t ：此次針對權重 θ_t 的更新值）

$$\Delta X_t = -\frac{\sqrt{\mathfrak{D}_{t-1} + \varepsilon}}{\sqrt{G_t + \varepsilon}} \nabla J(\theta_t)$$

$$\mathfrak{D}_{t-1} = \rho \mathfrak{D}_{t-2} + (1-\rho)\Delta X_{t-1}^{\ 2}$$ 歷史**更新值平方**累計+**衰減**

$$G_t = \rho G_{t-1} + (1-\rho)\nabla J(\theta_t)^2$$ 歷史**梯度平方**累計+**衰減**

「優化器」的選擇

- Keras 內常見的「優化器」（5）
  - AdaDelta

SGD
AdaGrad
AdaDelta
RMSProp
AdaM
NAdaM

- 公式說明：　$G_t = \rho G_{t-1} + (1-\rho)\nabla J(\theta_t)^2$　（假設 ρ = 0.9　$G_0 = 0$）

$$G_1 = 0.9 \times 0 + (1-0.9)\nabla J(\theta_1)^2 = 0.1\nabla J(\theta_1)^2$$

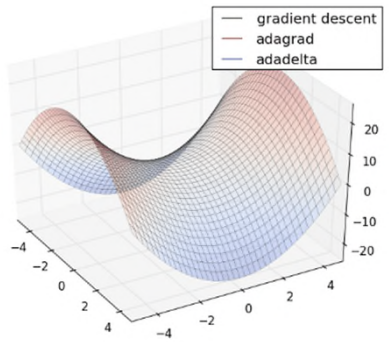
$$G_2 = 0.9 \times G_1 + (1-0.9)\nabla J(\theta_2)^2 = 0.9 \times (0.1\nabla J(\theta_1)^2) + (0.1\nabla J(\theta_2)^2)$$

$$G_3 = 0.9 \times G_2 + (1-0.9)\nabla J(\theta_3)^2$$

$$= \underline{0.81 \times (0.1\nabla J(\theta_1)^2)} + \underline{0.9 \times (0.1\nabla J(\theta_2)^2)} + \underline{(0.1\nabla J(\theta_3)^2)}$$

① 越古老、越衰減　② ρ 可以控制新加入項 $\nabla J(\theta_t)$ 的佔比 ③ $\mathfrak{D}_{t-1}$ 的累計原理類似

- 模型比較：



AdaGrad：收斂較慢

AdaDelta：收斂快，且完全模擬真實球體滾動軌跡

「優化器」的選擇

特別適用於 RNN

- Keras 內常見的「優化器」（6）
  - RMSProp（Root-Mean-Square Propagation）

SGD
- AdaGrad
  - AdaDelta
    - RMSProp
      - AdaM
        - NAdaM

- 底層類別： "rmsprop" = RMSprop(learning_rate=0.001, rho=0.95, momentum=0.0, epsilon=1e-07, centered=False)
- 學習速率： **變動**（梯度大→ η 大，梯度小→ η 小）。**預設值** = η = 0.001
- 解決問題： 修正 AdaGrad 中後段因為**學習速率**變小，而**收斂減緩**的問題。
- 解法巧思： 累加「**過往梯度**平方」時，要讓歷史**越久遠**的梯度，影響力**越小**。
- 數學原理：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \nabla J(\theta_t)$$

$$G_t = \rho G_{t-1} + (1-\rho)\nabla J(\theta_t)^2$$ 歷史**梯度平方**累計+**衰減**

「優化器」的選擇

- Keras 內常見的「優化器」（7）
  - AdaM（Adaptive Moment Estimation 自適應動差估計法）

SGD
AdaGrad
AdaDelta
RMSProp
AdaM
NAdaM

- 底層類別：“adam” = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07, amsgrad=False)
- 學習速率：**變動**（梯度大→ η 大，梯度小→ η 小）。**預設值** = η = 0.001
- 特色說明：(1) 結合「**RMSProp**」與「**動量法**」的優點。是目前用得**最廣泛**的演算法。
  (2) 有個**偏差校正式**，也是讓 AdaM 比別人優秀的另一個秘訣！
- 數學公式：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \varepsilon}}\hat{m}_t \qquad \hat{m}_t = \frac{m_t}{1-\beta_1{}^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2{}^t}$$

m_t, v_t 偏差校正式
- t=1 時，分母會變小
- 除以小分母，會使得初始動量變大

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla J(\theta_t)$$

歷史梯度累加+衰減 = 歷史梯度加權平均值
= 歷史梯度**一階動差估計式**（仿效動量法優點）

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla J(\theta_t)^2$$

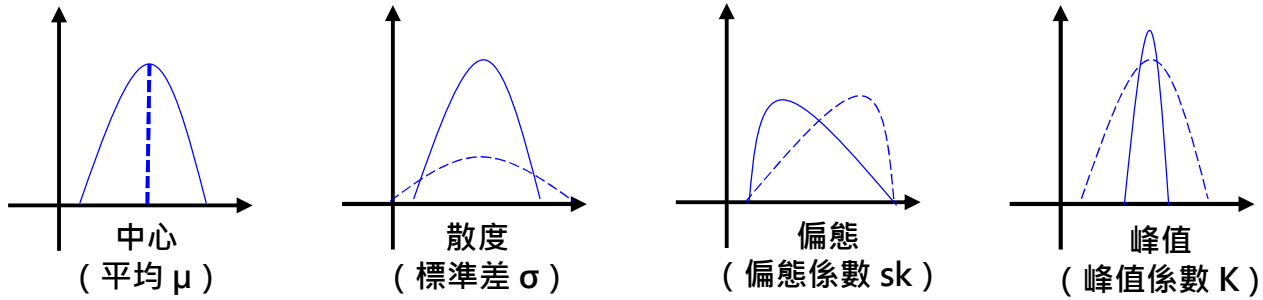
歷史梯度平方累加+衰減 = 歷史梯度變異數加權平均值
= 歷史梯度**二階動差估計式**（仿效 RMSProp 優點）

「優化器」的選擇

- Keras 內常見的「優化器」（7）
  - AdaM（Adaptive Moment Estimation 自適應動差估計法）

SGD
- AdaGrad
  - AdaDelta
    - RMSProp
      - **AdaM**
        - NAdaM

- 補充知識：何謂「**動差（Moment）**」？ → 一種能**快速計算**「**四大統計量**」的方法



- 動差定義：**樣本點 x_i** 與**特定數字 a** 距離的 **r 次方**和之**平均** = **r 階一般動差**
  - a=0：原點動差
  - a=μ：中心動差/主動差

$$\acute{m}_r = \frac{1}{n}\sum_{i=1}^{n}(x_i - a)^r$$

平均值 μ = 一階原點動差（r=1, a=0）= $\acute{m}_1 = \frac{1}{n}\sum_{i=1}^{n}(x_i - 0)^1 = \frac{1}{n}\sum_{i=1}^{n} x_i$

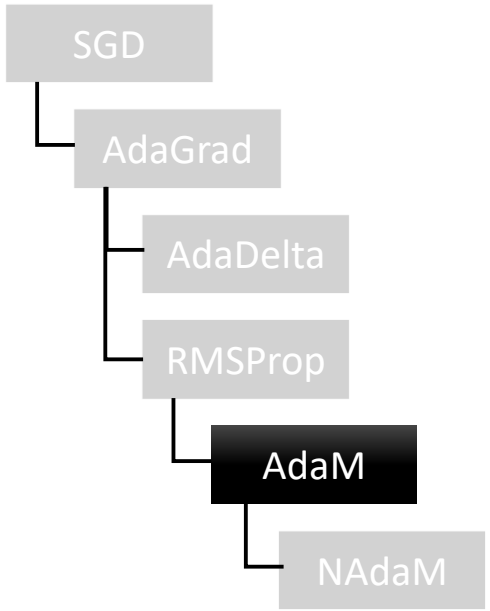
變異數 σ^2 = 二階主動差（r=2, a= μ）= $\acute{m}_2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$

偏態係數 sk = 三階主動差（r=3, a= μ）/ σ^3 = $\acute{m}_3 / \sigma^3$

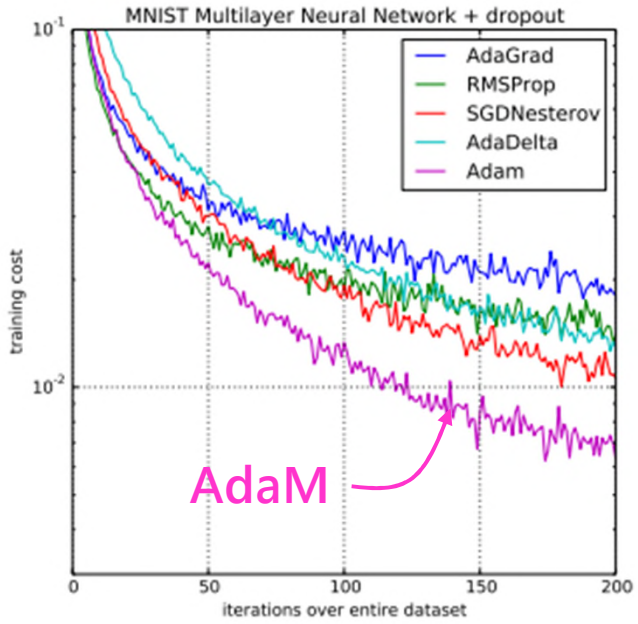
峰值係數 K = 四階主動差（r=4, a= μ）/ σ^4 = $\acute{m}_4 / \sigma^4$

「優化器」的選擇

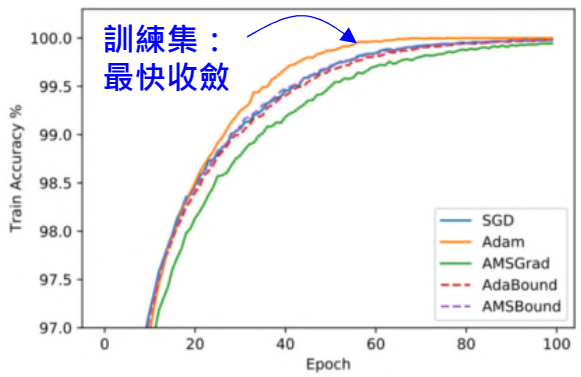
- Keras 內常見的「優化器」（7）
  - AdaM（Adaptive Moment Estimation 自適應動差估計法）



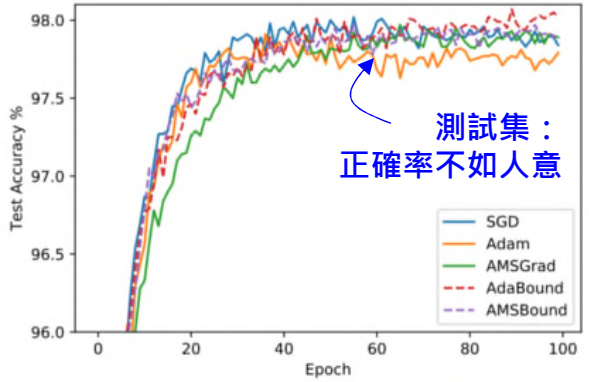
- Adam 的優點：收斂快



- Adam 的缺點： 複雜的 J(θ) 容易 Overfit，去找到那些擁有「病態曲率」的低點當極值



(a) Training Accuracy



(b) Test Accuracy

參考： https://is.gd/bIlyLl

「優化器」的選擇

- Keras 內常見的「優化器」（7）
  - AMSGrad：AdaM 的改進方法

SGD
AdaGrad
AdaDelta
RMSProp
AdaM
NAdaM

- 改進方法：AMSGrad

ON THE CONVERGENCE OF ADAM AND BEYOND

Sashank J. Reddi, Satyen Kale & Sanjiv Kumar
Google New York
New York, NY 10011, USA
{sashank,satyenkale,sanjivk}@google.com

2018

- 數學公式：

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

$$\hat{m}_t = \frac{m_t}{1-\beta_1{}^t} \qquad \hat{v}_t = \max\left(\frac{v_{t-1}}{1-\beta_2{}^t}, v_t\right)$$

只改了這裡

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla J(\theta_t)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla J(\theta_t)^2$$

"Adam 的問題在於，它太快收斂，
導致學習速率 η 縮小而「變慢」掉入深谷，
所以「過小的變化 v_t」就踢掉
衝快一點就能避免掉入山谷"

- 程式寫法：

```
Adam(learning_rate=0.001,
     beta_1=0.9, beta_2=0.999,
     epsilon=1e-07,
     amsgrad=True)
```

「優化器」的選擇

- Keras 內常見的「優化器」（8）
  - NAdaM（Nesterov-Accelerated Adaptive Moment Estimation）

SGD
AdaGrad
AdaDelta
RMSProp
AdaM
NAdaM

- 底層類別： "nadam" = Nadam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07)
- 學習速率：**變動**（梯度大→ η 大，梯度小→ η 小）。**預設值** = η = 0.001
- 特色說明：將 Adam 內的「動量」，換成「**Nesterov 動量**」（多偷看一步，**收斂快**）。
- 數學公式：

$$\boldsymbol{\theta_{t+1}} = \boldsymbol{\theta_t} - \frac{\eta}{\sqrt{\widehat{n_t}} + \varepsilon}\left(\beta_1 \widehat{m_t} + \frac{1-\beta_1}{1-{\beta_1}^t}\nabla J(\boldsymbol{\theta_t})\right)$$

$$\widehat{m_t} = \frac{m_t}{1-{\beta_1}^t} \qquad m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla J(\boldsymbol{\theta_t})$$

$$\widehat{n_t} = \frac{n_t}{1-{\beta_2}^t} \qquad n_t = \beta_2 n_{t-1} + (1-\beta_2)\nabla J(\boldsymbol{\theta_t})^2$$

「優化器」的選擇

- Keras 內常見的「優化器」（8）
  - NAdaM (Nesterov-Accelerated Adaptive Moment Estimation)

SGD
- AdaGrad
  - AdaDelta
    - RMSProp
      - AdaM
        - NAdaM

- 公式推導： 先看 Adam 公式

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

代入

$$\hat{m}_t = \frac{m_t}{1-\beta_1^{\,t}} \qquad m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla J(\theta_t)$$

代入

$$\hat{n}_t = \frac{n_t}{1-\beta_2^{\,t}} \qquad n_t = \beta_2 n_{t-1} + (1-\beta_2)\nabla J(\theta_t)^2$$

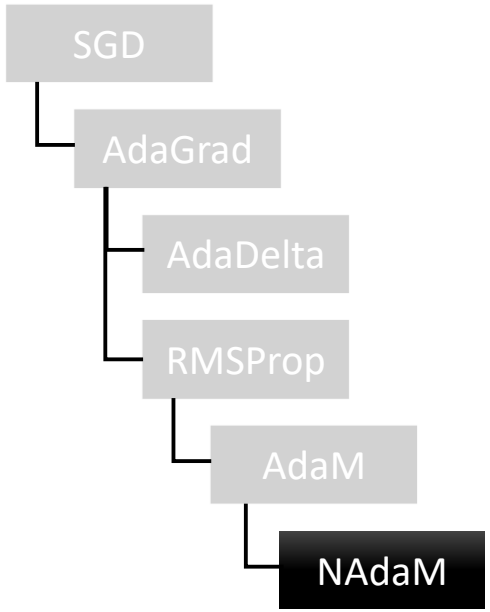
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{n}_t + \varepsilon}} \left(\frac{\beta_1 m_{t-1}}{1-\beta_1^{\,t}} + \frac{1-\beta_1}{1-\beta_1^{\,t}} \nabla J(\theta_t) \right)$$

若故意把過去動量 m_{t-1}
硬生生換成現在動量 m_t
（多看一步）$\rightarrow \frac{m_t}{1-\beta_1^{\,t}} = \hat{m}_t$

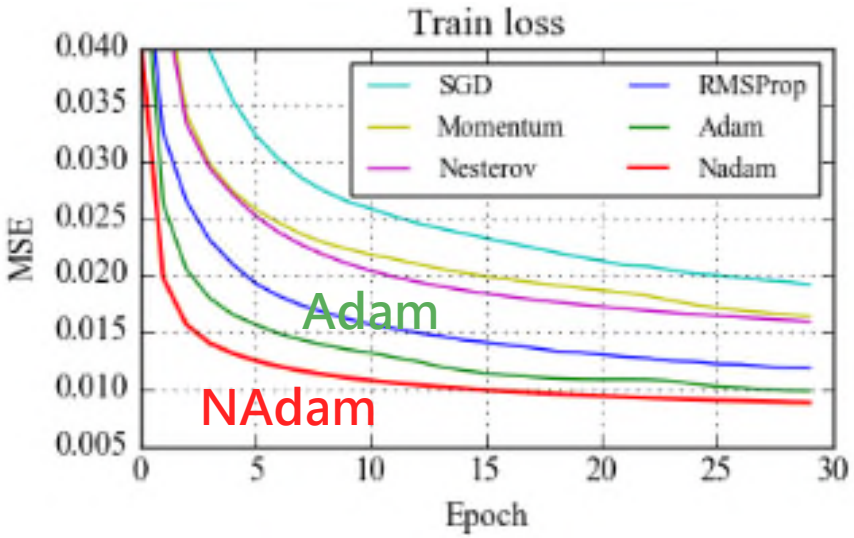
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{n}_t + \varepsilon}} \left(\beta_1 \hat{m}_t + \frac{1-\beta_1}{1-\beta_1^{\,t}} \nabla J(\theta_t) \right)$$

「優化器」的選擇

- Keras 內常見的「優化器」（8）
  - NAdaM（Nesterov-Accelerated Adaptive Moment Estimation）



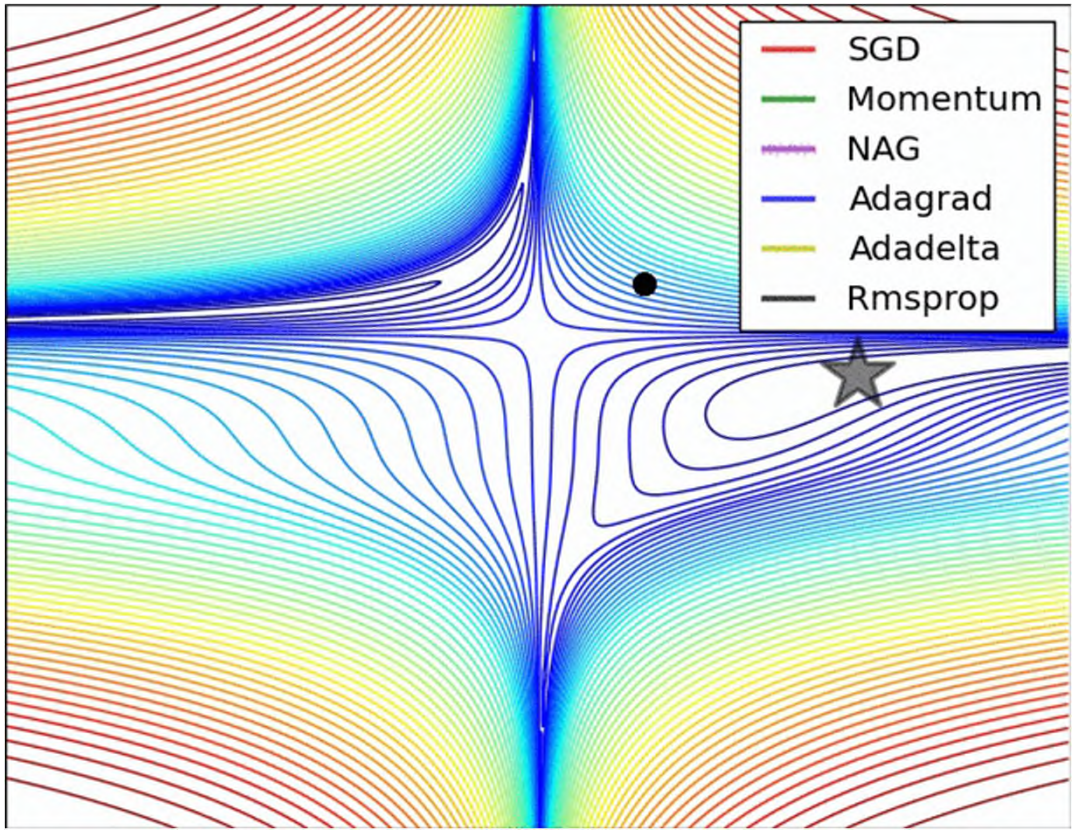
- NAdaM 與 AdaM 比較：

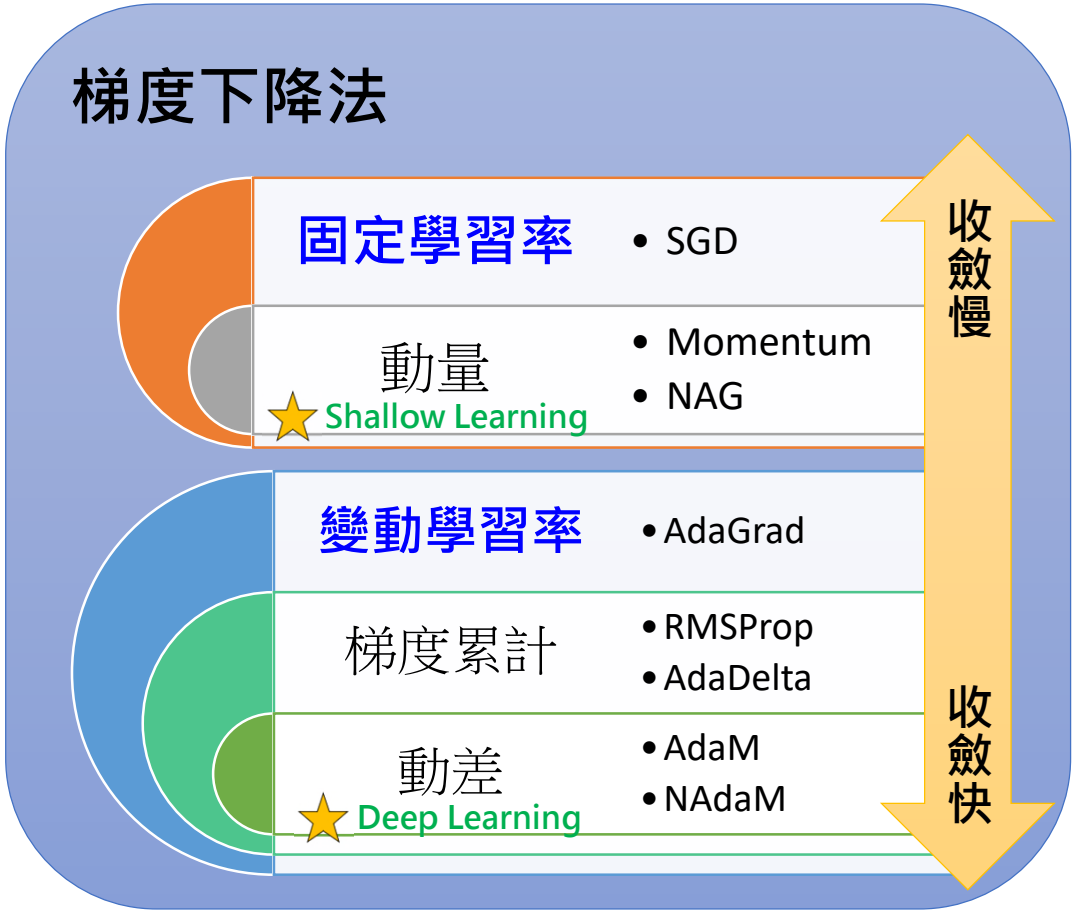


NAdam 收斂通常比 Adam 快！

「優化器」的選擇

- Step 1. 挑選「優化器模型」





「優化器」的選擇

- Step 2. 挑選「初始學習速率 η」

- SGD ≒ 0.1
- Momentum ≒ 0.1
- AdaGrad ≒ 0.1
- AdaDelta ≒ 10
- RMSProp ≒ 0.001
- Adam ≒ 0.001

比較保險的作法：超參數搜尋

- 網格搜尋法（Grid Search）
- Population Based Training (PBT)
  （一種基因演算法，Google 提出）

資料來源：*How to pick the best learning rate for your machine learning project*

本章總結

- **「多層感知器」架構**
  - 輸入層、隱藏層、輸出層

- **「隱藏層」節點個數**
  - 公式一：$\frac{\text{上一層節點數}+\text{下一層節點數}}{2}$
  - 公式二：$\frac{\text{樣本點個數}}{\alpha\times(\text{上一層節點數}+\text{下一層節點數})}$

- **「隱藏層」層數**
  - 淺層學習：2~3 層
  - 深度學習：數十~數百層

本章總結

- 「人工神經網路」學習架構