專案主題:
Introduction of Feed Forward NN & CNN

學生: 陳冠維 國立清華大學 計財所 碩一

大綱

- 一般NN模型介紹
- 模型訓練
- NN實作
- CNN
- CNN實作

1

一般NN模型介紹

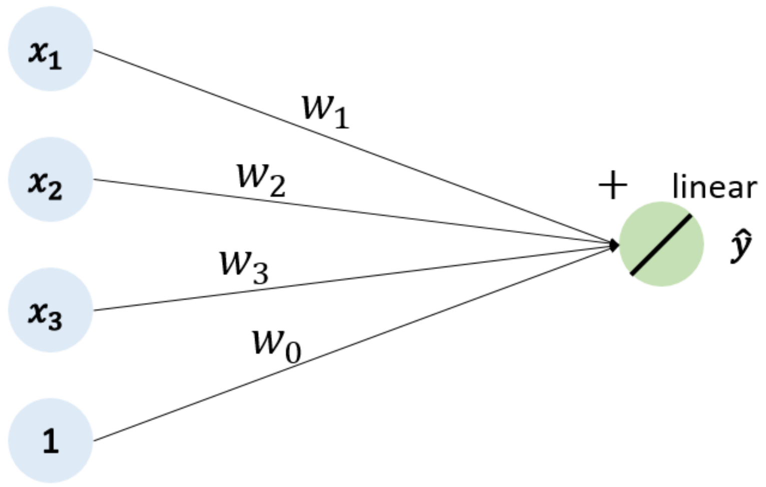
Neural Network / Deep Learning

- 「類神經網路」發跡於1980年代，由日本科學家福島邦彥提出
  - 1990年代，NN近乎消聲匿跡，原因是計算代價過於龐大，耗時過久
- 直至2010年代，NN再次受到重視，賦名為「深度學習」
  - 歸功於電腦計算能力的提升，人們得以透過NN解決許多難題
  - 例如：圖像辨識、影音辨識、NLP、AlphaGo

Model

- Regression model可視為neural network的一個特例

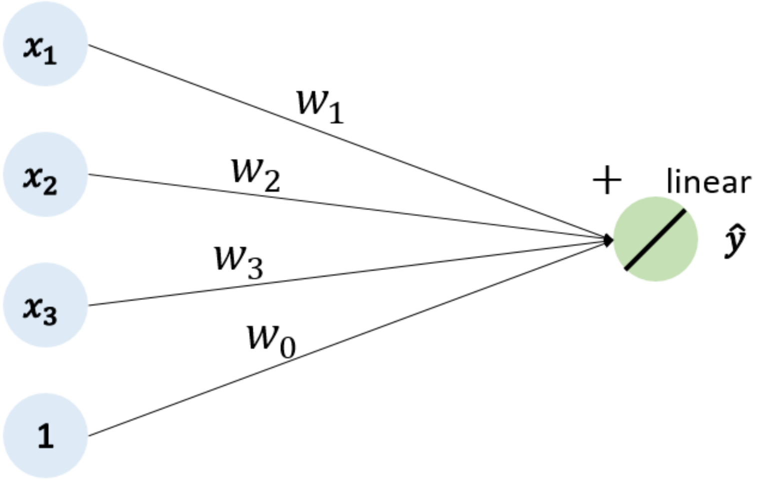
$$\boldsymbol{y} = w_0 + w_1\boldsymbol{x_1} + w_2\boldsymbol{x_2} + w_3\boldsymbol{x_3} + \boldsymbol{\varepsilon}$$



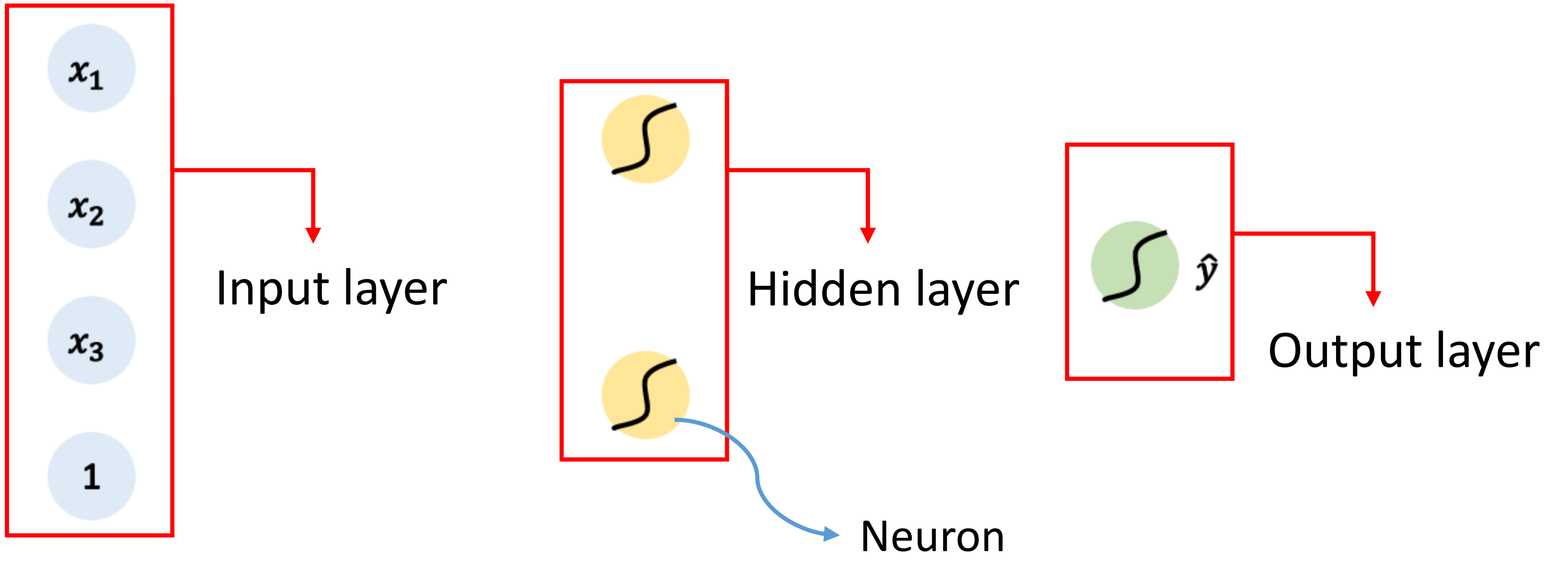
Model

- Regression model可視為neural network的一個特例

$$\boldsymbol{y} = w_0 + w_1\boldsymbol{x_1} + w_2\boldsymbol{x_2} + w_3\boldsymbol{x_3} + \boldsymbol{\varepsilon}$$



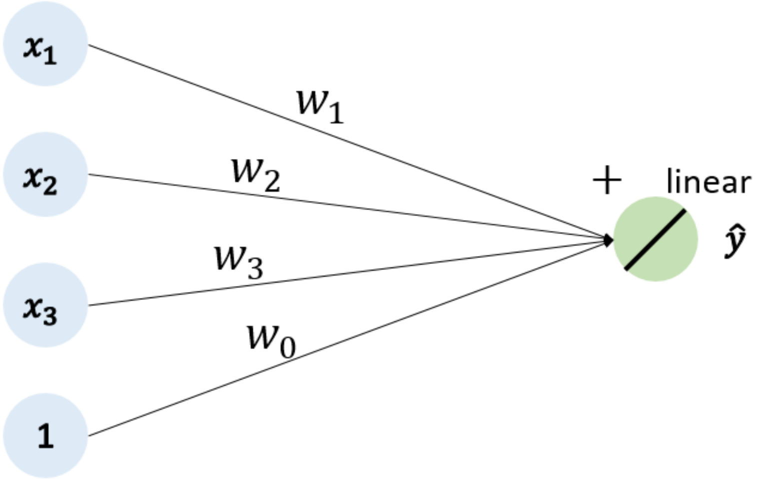
- Feed-forward and fully-connected NN



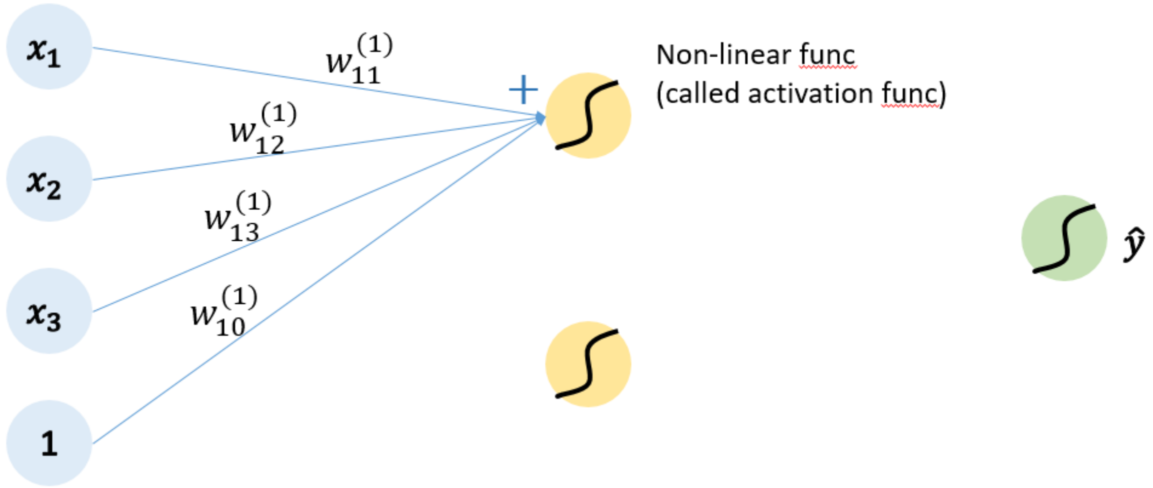
Model

- Regression model可視為neural network的一個特例

$$\boldsymbol{y} = w_0 + w_1\boldsymbol{x}_1 + w_2\boldsymbol{x}_2 + w_3\boldsymbol{x}_3 + \boldsymbol{\varepsilon}$$



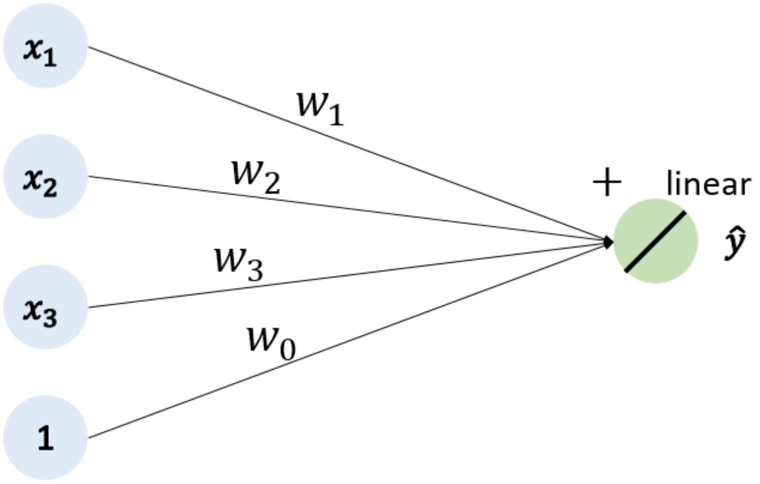
- Feed-forward and fully-connected NN



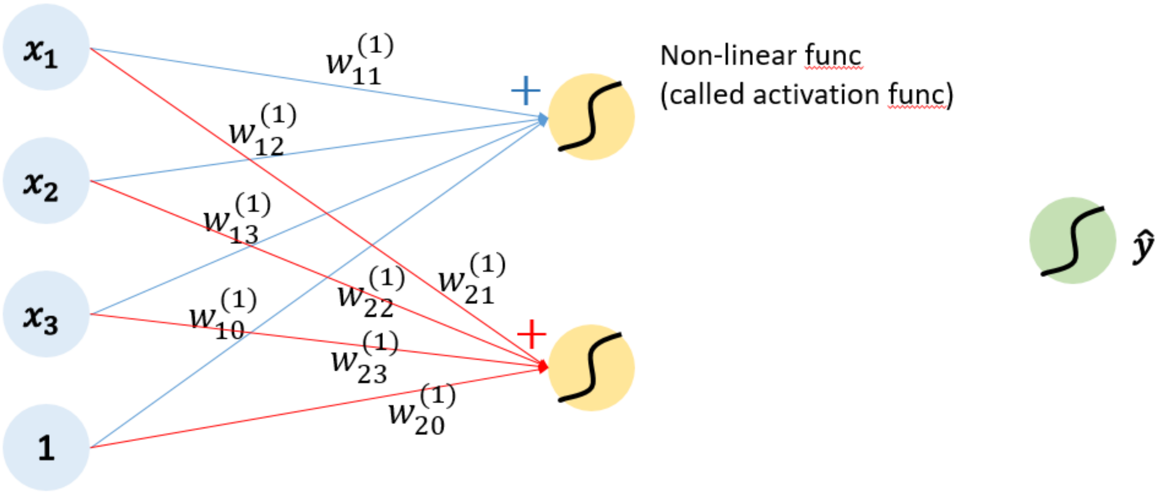
Model

- Regression model可視為neural network的一個特例

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + \varepsilon$$



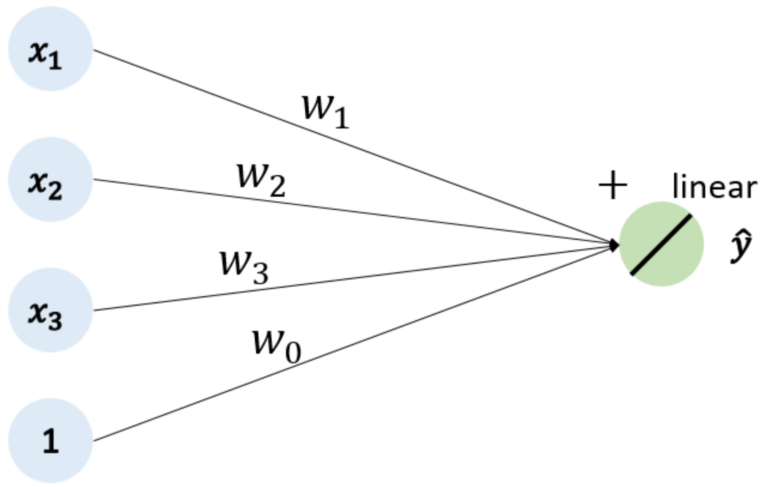
- Feed-forward and fully-connected NN



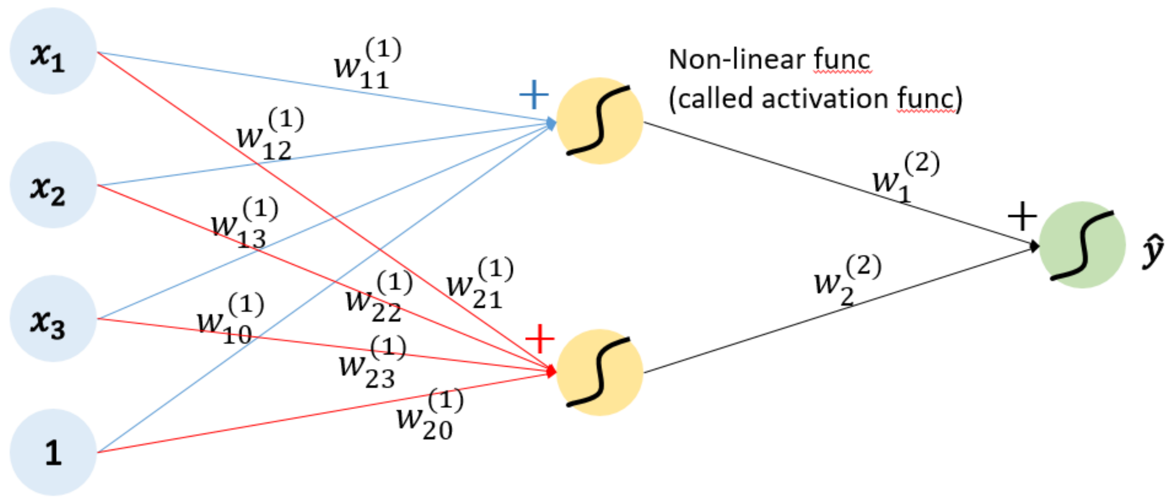
Model

- Regression model可視為neural network的一個特例

$$\boldsymbol{y} = w_0 + w_1\boldsymbol{x_1} + w_2\boldsymbol{x_2} + w_3\boldsymbol{x_3} + \boldsymbol{\varepsilon}$$



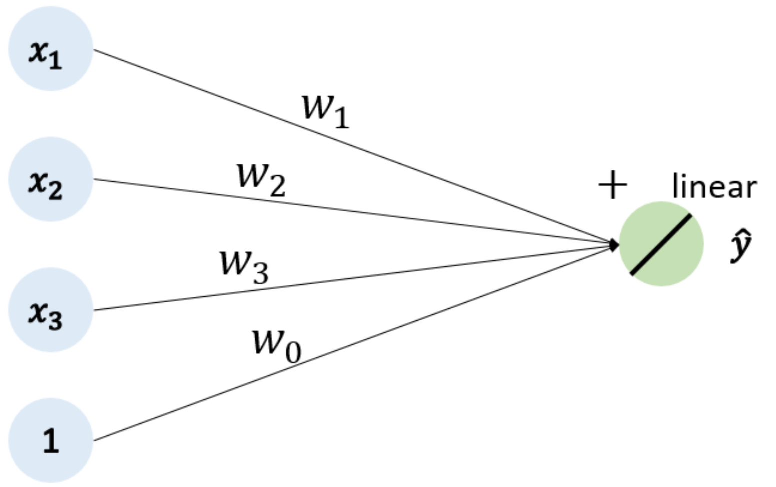
- Feed-forward and fully-connected NN



Model

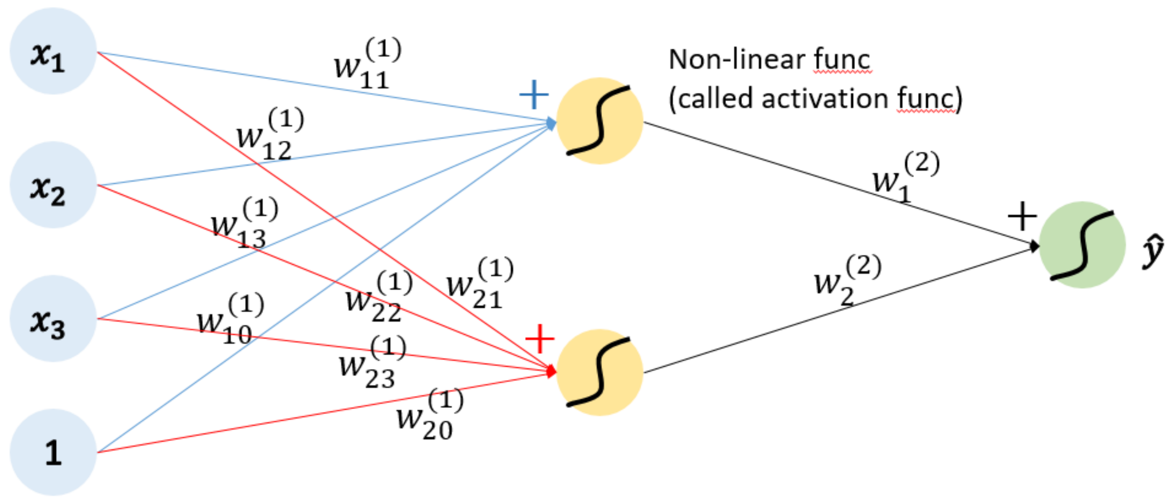
- Regression model可視為neural network的一個特例

$$\boldsymbol{y} = w_0 + w_1\boldsymbol{x_1} + w_2\boldsymbol{x_2} + w_3\boldsymbol{x_3} + \boldsymbol{\varepsilon}$$



- 訓練參數數量較少
- input與權重做線性組合後的結果就是output

- Feed-forward and fully-connected NN

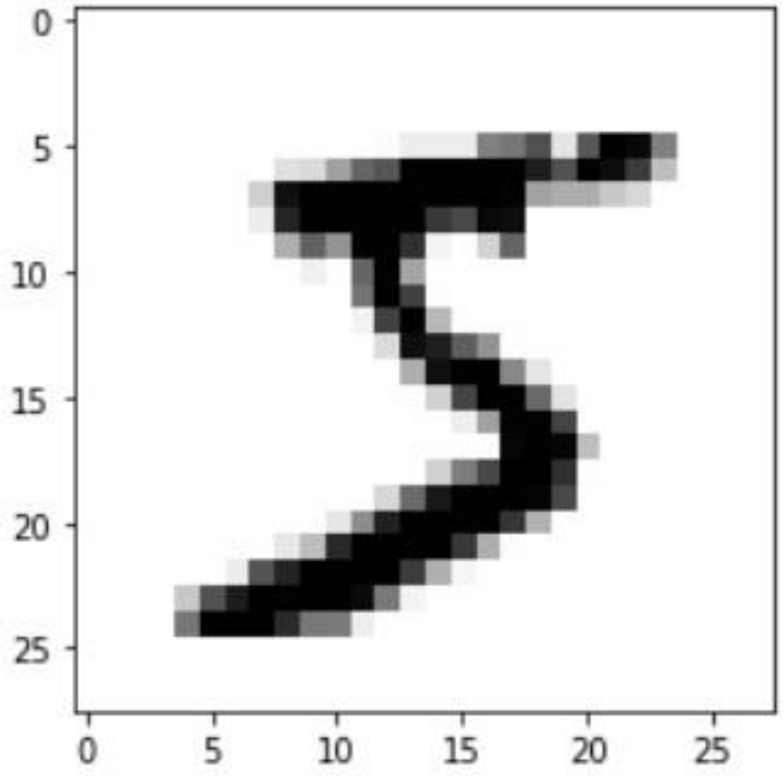


- 訓練參數數量較多
- input與權重做線性組合後，會經過一個非線性的激勵函數，再產生output

Example: Image

每一個pixel的值介於 0~255
值越大的地方代表顏色越深

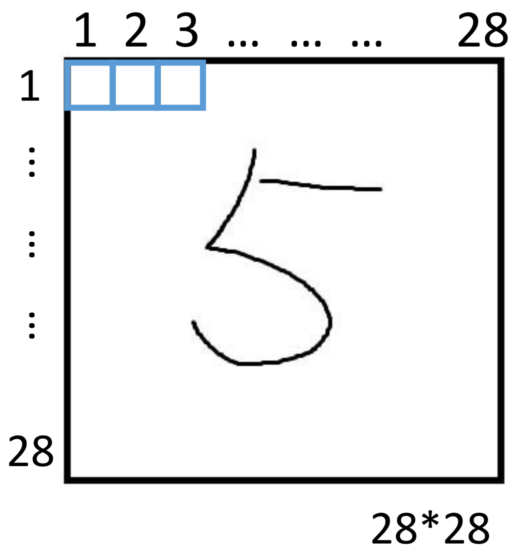
Input 為一個 28*28 的灰階圖像



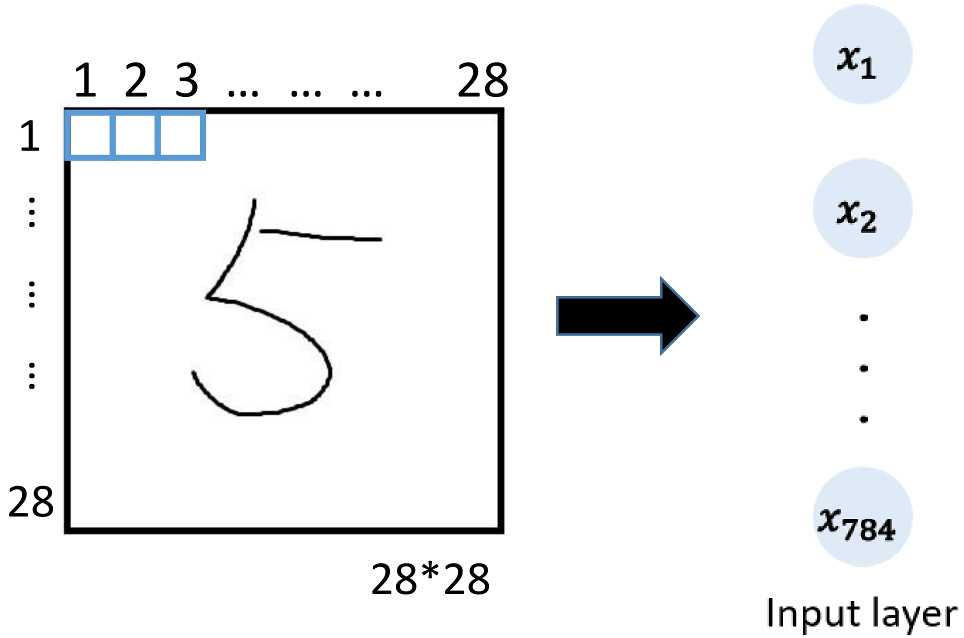
透過matrix
將圖像轉換成數值

	0	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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	3	18	18	18	126	136	175	26	166	255	247	127	0	0	0	0
6	0	0	0	0	0	0	0	0	30	36	94	154	170	253	253	253	253	253	225	172	253	242	195	64	0	0	0	0
7	0	0	0	0	0	0	0	49	238	253	253	253	253	253	253	253	253	251	93	82	82	56	39	0	0	0	0	0
8	0	0	0	0	0	0	0	18	219	253	253	253	253	253	198	182	247	241	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	80	156	107	253	253	205	11	0	43	154	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	14	1	154	253	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	139	253	190	2	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	11	190	253	70	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	35	241	225	160	108	1	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	81	240	253	253	119	25	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	186	253	253	150	27	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	93	252	253	187	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	249	253	249	64	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	46	130	183	253	253	207	2	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	39	148	229	253	253	253	250	182	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	24	114	221	253	253	253	253	201	78	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	23	66	213	253	253	253	253	198	81	2	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	18	171	219	253	253	253	253	195	80	9	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	55	172	226	253	253	253	253	244	133	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	136	253	253	253	212	135	132	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Example: Image

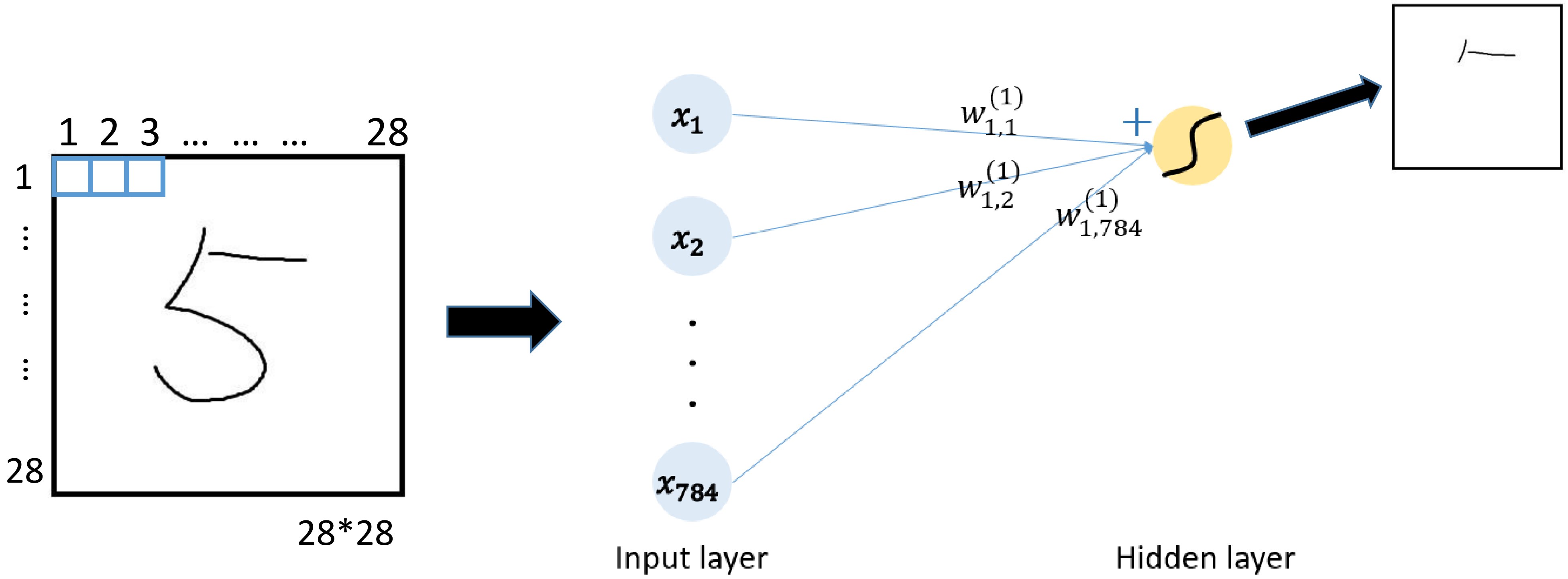


Example: Image



將這個 28*28 的圖像拉成一條向量，每個 1*1 的像素代表一個 factor (x_i)

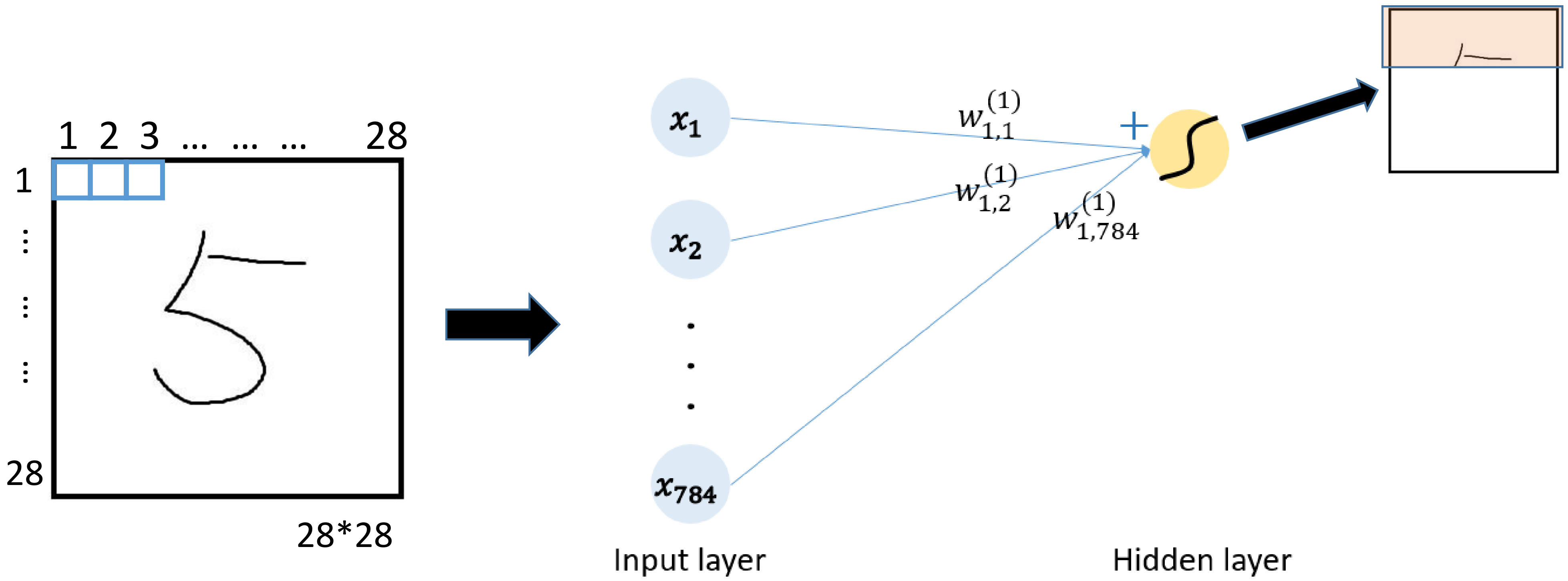
Example: Image



將這個 28*28 的圖像拉成一條向量，每個 1*1 的像素代表一個 factor (x_i)

作特徵擷取

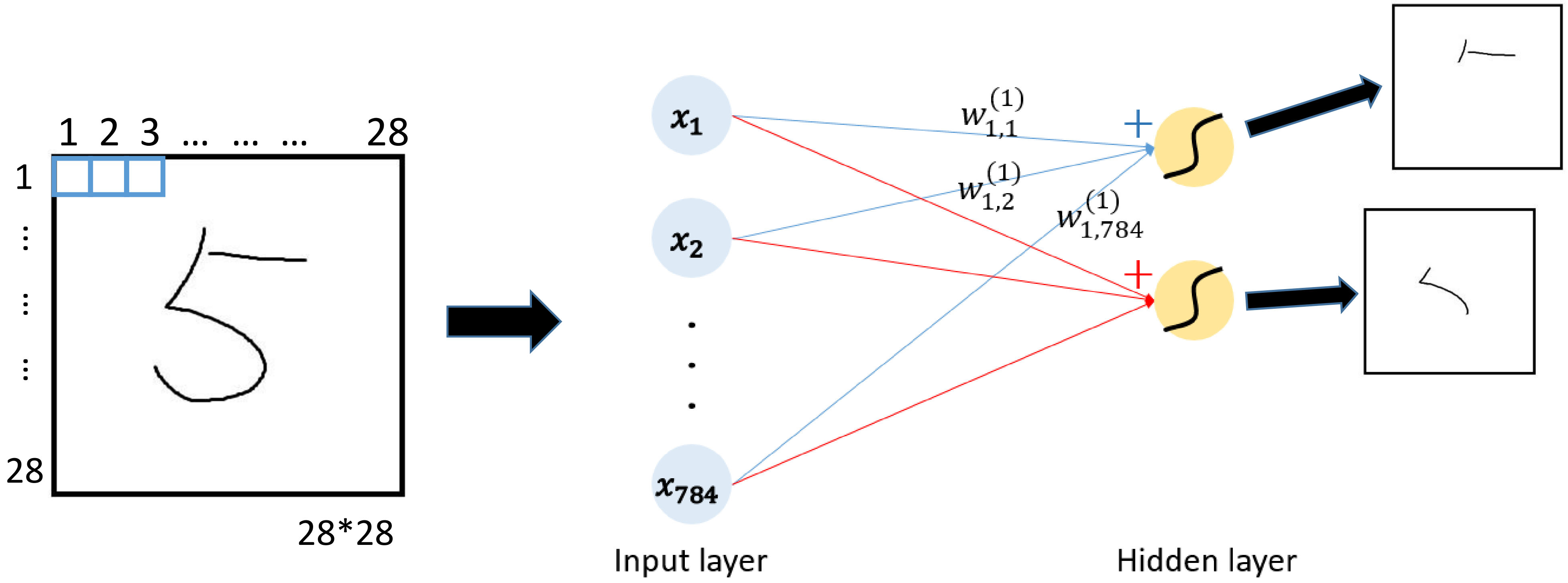
Example: Image



將這個 28*28 的圖像拉成一條向量，每個 1*1 的像素代表一個 factor (x_i)

作特徵擷取

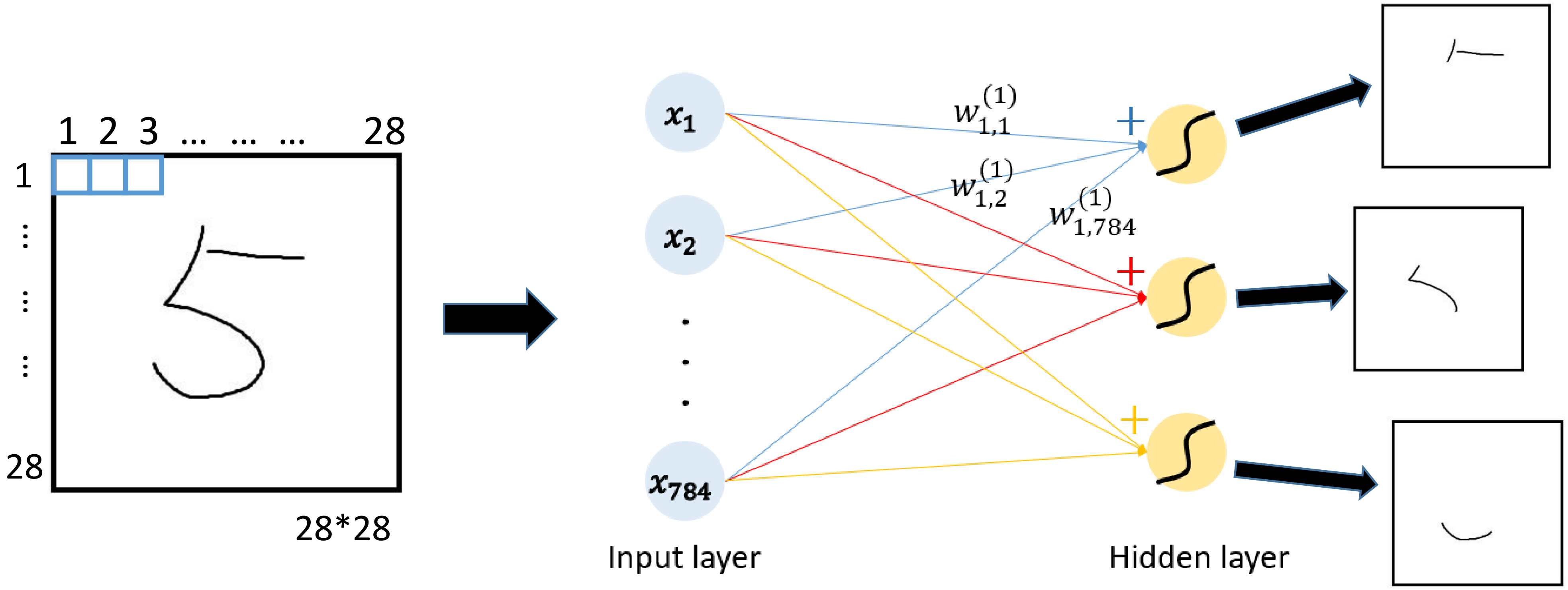
Example: Image



將這個 28*28 的圖像拉成一條向量，每個 1*1 的像素代表一個 factor (x_i)

作特徵擷取

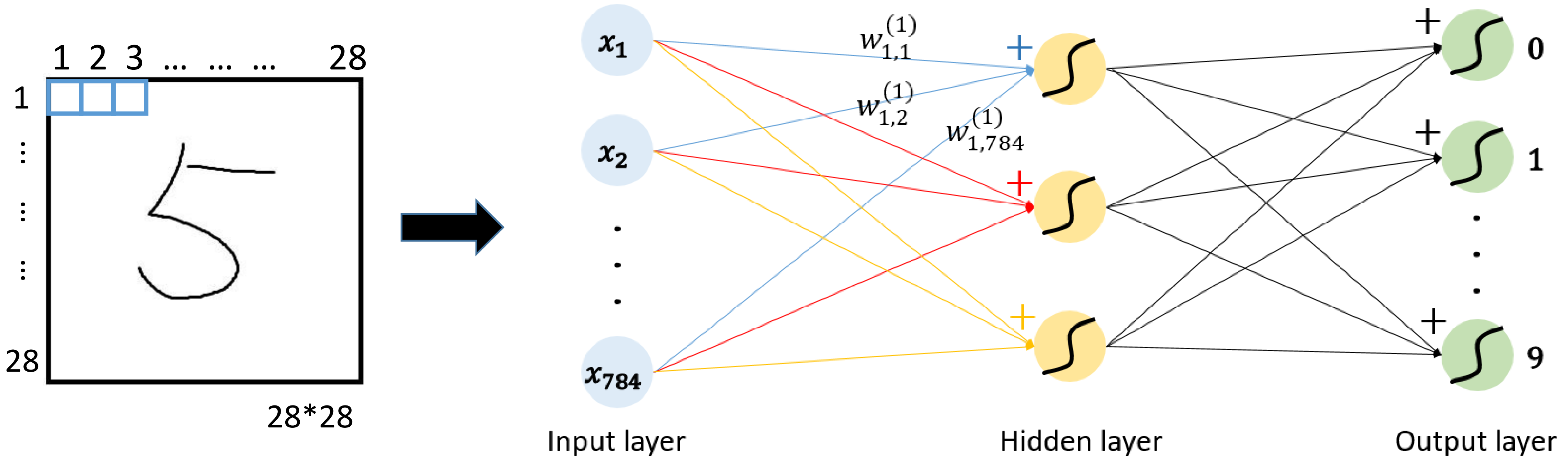
Example: Image



將這個 28*28 的圖像拉成一條向量，每個 1*1 的像素代表一個 factor (x_i)

作特徵擷取

Example: Image



將這個 28*28 的圖像拉成一條向量，每個 1*1 的像素代表一個 factor (x_i)

作特徵擷取

透過Hidden layer擷取出的這些特徵，去預測這個圖像是哪個類別

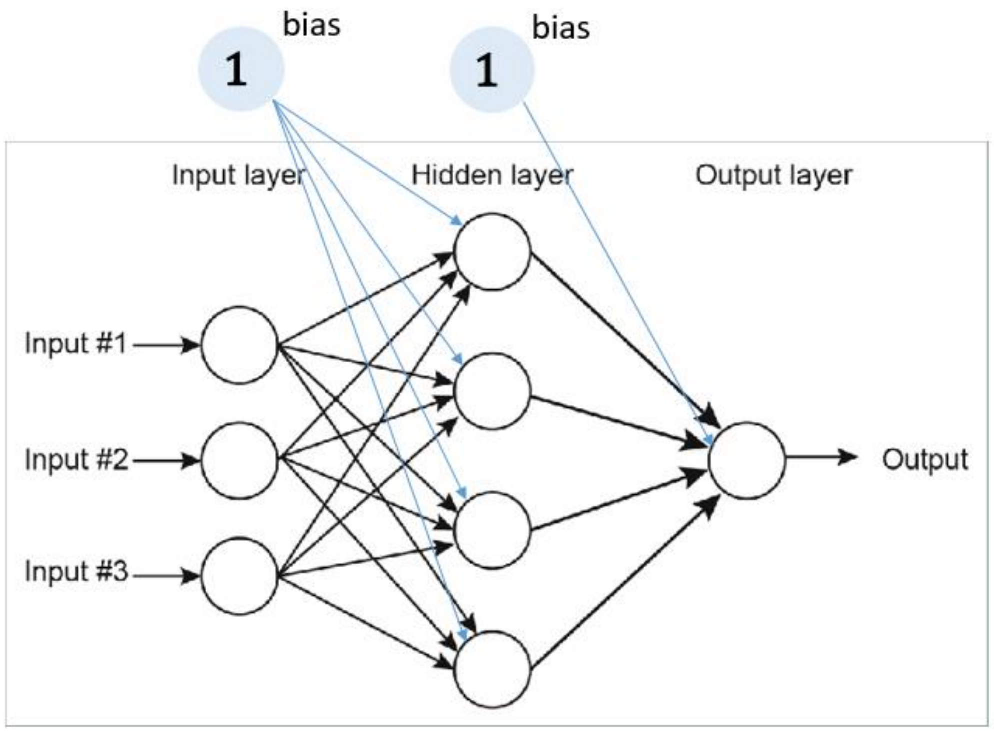
Feed-forward neural network

➢ 3-layer:

$$z_\ell^{(2)} = w_{\ell 0}^{(1)} + \sum_{j=1}^{p} w_{\ell j}^{(1)} x_j\,, \qquad \ell = 1, 2, \dots, p_2$$

$$a_\ell^{(2)} = g^{(2)}\left(z_\ell^{(2)}\right), \qquad \ell = 1, 2, \dots, p_2$$

$$z^{(3)} = w_0^{(2)} + \sum_{\ell=1}^{p_2} w_\ell^{(2)} a_\ell^{(2)}\,, \qquad o = z^{(3)}$$



Note:

$z_\ell^{(2)}$ is a linear transformation of all inputs.

$g^{(2)}(\cdot)$ is ***activation function*** (usually nonlinear).

$w_{\ell j}^{(1)}$ and $w_\ell^{(2)}$ are ***weights***.

$w_{\ell 0}^{(1)}$ and $w_0^{(2)}$ are ***bias parameters***.

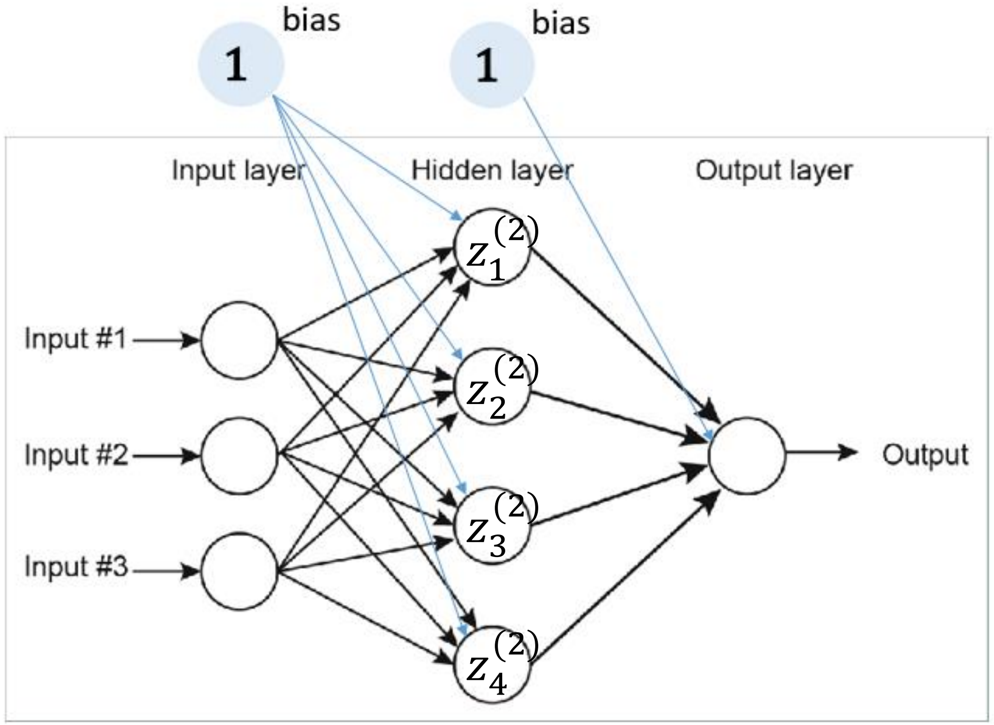
Feed-forward neural network

➢ 3-layer:

$$z_\ell^{(2)} = w_{\ell 0}^{(1)} + \sum_{j=1}^{p} w_{\ell j}^{(1)} x_j\,, \qquad \ell = 1, 2, \dots, p_2$$

$$a_\ell^{(2)} = g^{(2)}\left(z_\ell^{(2)}\right), \qquad \ell = 1, 2, \dots, p_2$$

$$z^{(3)} = w_0^{(2)} + \sum_{\ell=1}^{p_2} w_\ell^{(2)} a_\ell^{(2)}\,, \qquad o = z^{(3)}$$



Note:

$z_\ell^{(2)}$ is a linear transformation of all inputs.

$g^{(2)}(\cdot)$ is ***activation function*** (usually nonlinear).

$w_{\ell j}^{(1)}$ and $w_\ell^{(2)}$ are ***weights***.

$w_{\ell 0}^{(1)}$ and $w_0^{(2)}$ are ***bias parameters***.

Feed-forward neural network

- 3-layer:

$$z_\ell^{(2)} = w_{\ell 0}^{(1)} + \sum_{j=1}^{p} w_{\ell j}^{(1)} x_j\,, \qquad \ell = 1, 2, \dots, p_2$$

$$a_\ell^{(2)} = g^{(2)}\left(z_\ell^{(2)}\right), \qquad \ell = 1, 2, \dots, p_2$$

$$z^{(3)} = w_0^{(2)} + \sum_{\ell=1}^{p_2} w_\ell^{(2)} a_\ell^{(2)}\,, \qquad o = z^{(3)}$$

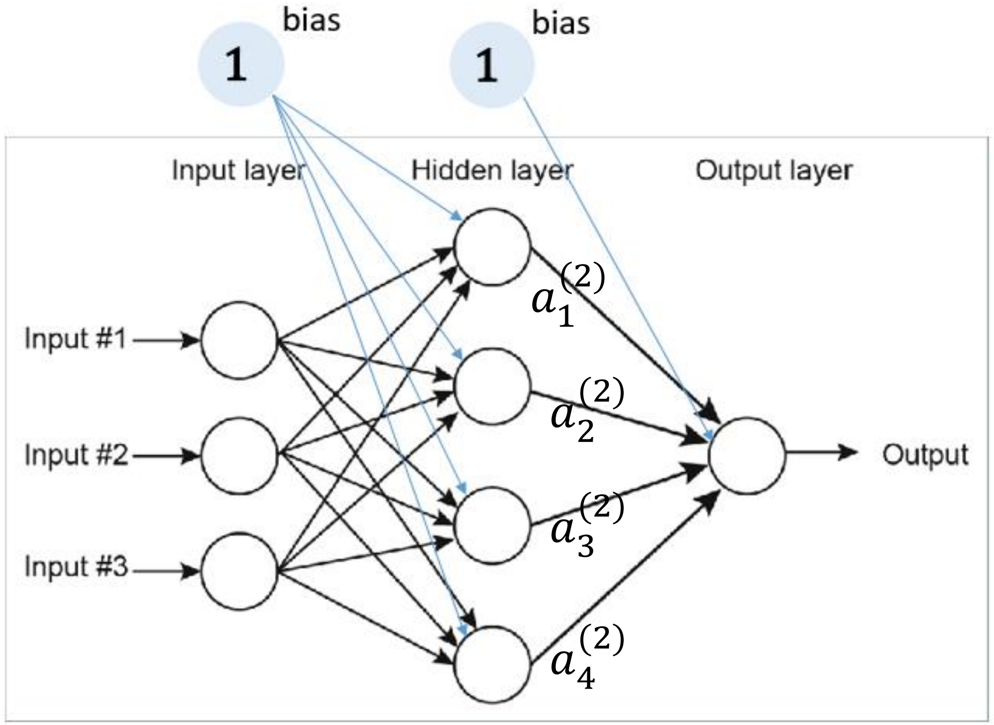
Note:

$z_\ell^{(2)}$ is a linear transformation of all inputs.

$g^{(2)}(\cdot)$ is ***activation function*** (usually nonlinear).

$w_{\ell j}^{(1)}$ and $w_\ell^{(2)}$ are ***weights***.

$w_{\ell 0}^{(1)}$ and $w_0^{(2)}$ are ***bias parameters***.



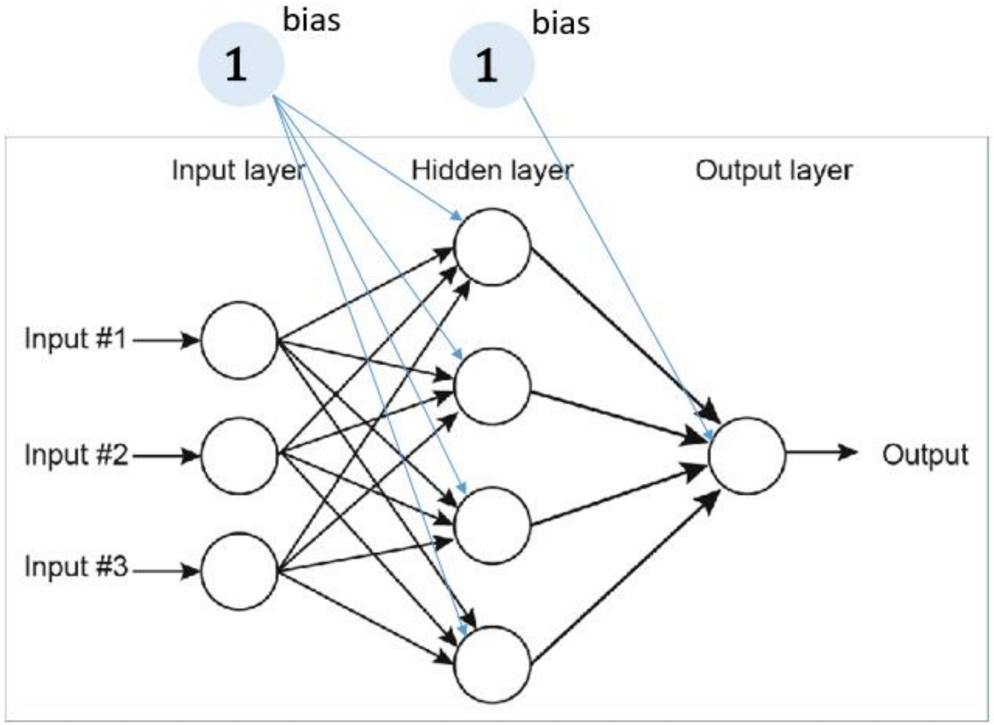
Feed-forward neural network

➢ 3-layer:

$$z_\ell^{(2)} = w_{\ell 0}^{(1)} + \sum_{j=1}^{p} w_{\ell j}^{(1)} x_j\,, \qquad \ell = 1, 2, \dots, p_2$$

$$a_\ell^{(2)} = g^{(2)}\left(z_\ell^{(2)}\right), \qquad \ell = 1, 2, \dots, p_2$$

$$z^{(3)} = w_0^{(2)} + \sum_{\ell=1}^{p_2} w_\ell^{(2)} a_\ell^{(2)}\,, \qquad o = z^{(3)}$$



➢ K-layer:

$$z_\ell^{(k)} = w_{\ell 0}^{(k-1)} + \sum_{j=1}^{p_{k-1}} w_{\ell j}^{(k-1)} a_j^{(k-1)}\,, \qquad a_\ell^{(k)} = g^{(k)}\left(z_\ell^{(k)}\right),$$

$$for\ \ell = 1, 2, \dots, p_k, \qquad where\ a_j^{(1)} = x_j\ \ and\ \ p_1 = p$$

What does Activation Function do?

- A K-layer feed-forward NN

$$f(\boldsymbol{x}, \boldsymbol{w}) = w_0^{(k-1)} + \sum_{\ell=1}^{p_{k-1}} w_\ell^{(k-1)} g^{(k-1)}\left(w_{\ell 0}^{(k-2)} + \sum_{j=1}^{p_{k-2}} w_{\ell j}^{(k-2)} x_j \right)$$

- Note, Activation Function: $g^{(k-1)}(\cdot)$

What does Activation Function do?

- A K-layer feed-forward NN

$$f(\boldsymbol{x}, \boldsymbol{w}) = w_0^{(k-1)} + \sum_{\ell=1}^{p_{k-1}} w_\ell^{(k-1)} g^{(k-1)}\left(w_{\ell 0}^{(k-2)} + \sum_{j=1}^{p_{k-2}} w_{\ell j}^{(k-2)} x_j \right)$$

  - Note, Activation Function: $g^{(k-1)}(\cdot)$

- **Basis Expansion**

$$f(X) = \sum_{m=1}^{M} \beta_m h_m(X)$$

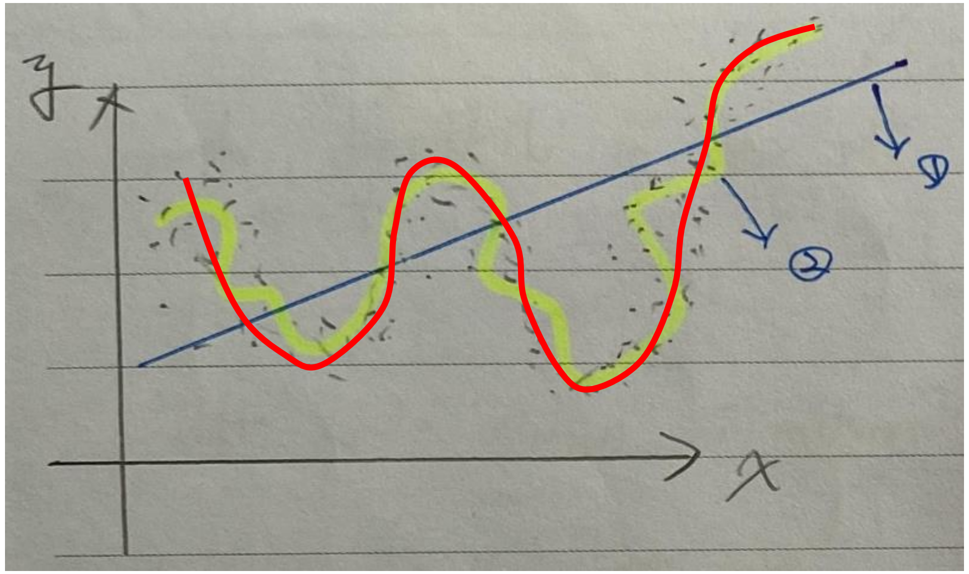
Basis Expansion

透過函數向量$\{h_1(X),\ h_2(X),\ \dots,\ h_M(X)\}$將X轉換到M維空間

$$f(X) = \sum_{m=1}^{M} \beta_m h_m(X)$$

$h_m(X)$ example:

$$h_1(X) = x_i \quad ; \quad h_2(X) = x_i^2 \quad ; \quad h_3(X) = \log(x_i) \dots$$



True function（紅線）： $y_i = g(x_i) + \varepsilon_i$

Model ①（藍線）：簡單的線性模型

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

Model ②（黃線）：透過一些polynomial term捕捉True function非線性的部分，我們可以**更好的去逼近True function**

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_k x_i^k + \varepsilon_i$$

Activation Function VS Basis Expansion

- A K-layer feed-forward NN

$$f(\boldsymbol{x},\boldsymbol{w}) = w_0^{(k-1)} + \sum_{\ell=1}^{p_{k-1}} w_\ell^{(k-1)} g^{(k-1)}\left(w_{\ell 0}^{(k-2)} + \sum_{j=1}^{p_{k-2}} w_{\ell j}^{(k-2)} x_j \right)$$

- **Basis Expansion**

$$f(X) = \sum_{m=1}^{M} \beta_m h_m(X)$$

- Activation Function的特點
  - 激勵函數其實就像Basis Expansion一樣，將輸入值做一些非線性的轉換，使我們的模型可以更好去逼近任意函數
  - 不過可以發現激勵函數有別於一般的Basis Expansion，其輸入值的權重是學出來的（$w_{\ell j}^{(k-2)}$），而Basis Expansion的輸入值則是給定的X。這就造就了我們的激勵函數的待估參數有無限多種可能，有別於一般的Basis Expansion只有一種可能。因此激勵函數要來的更加有彈性，可以更好的去捕捉各種pattern。

Activation Function

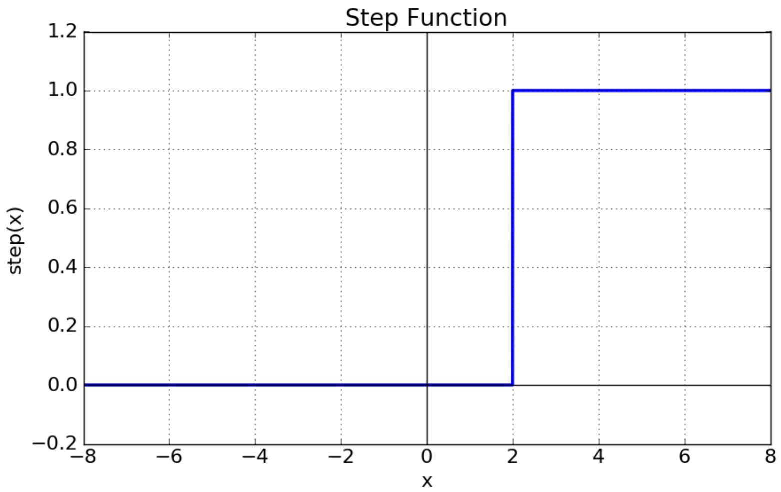
- ➢ $g^{(k)}(\cdot)$ is known as the ***activation function*** (usually nonlinear).
  - $g^{(k)}(\cdot)$ at the inner layers can be the same or different.

- ➢ When it comes to human brain:

  Each neuron in the network would be a simple binary on/off.

- ➢ For example, Heaviside (or Step) function:

$$f(x) = \begin{cases} 0 & : x_i \leq T \\ 1 & : x_i > T \end{cases}$$



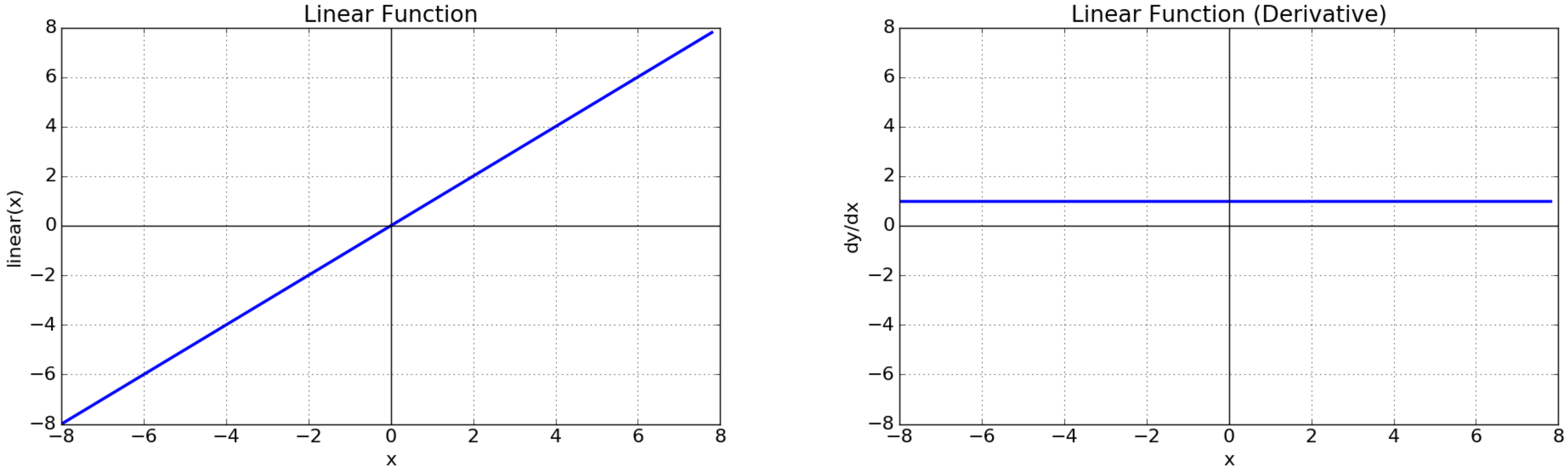
- ➢ In practice:

  People usually consider differentiable activation function.

  (Reason: We need to do gradient descent with backpropagation)

Activation Function — Identity (or Linear) Function

- Formula: $f(x_i) = x_i$



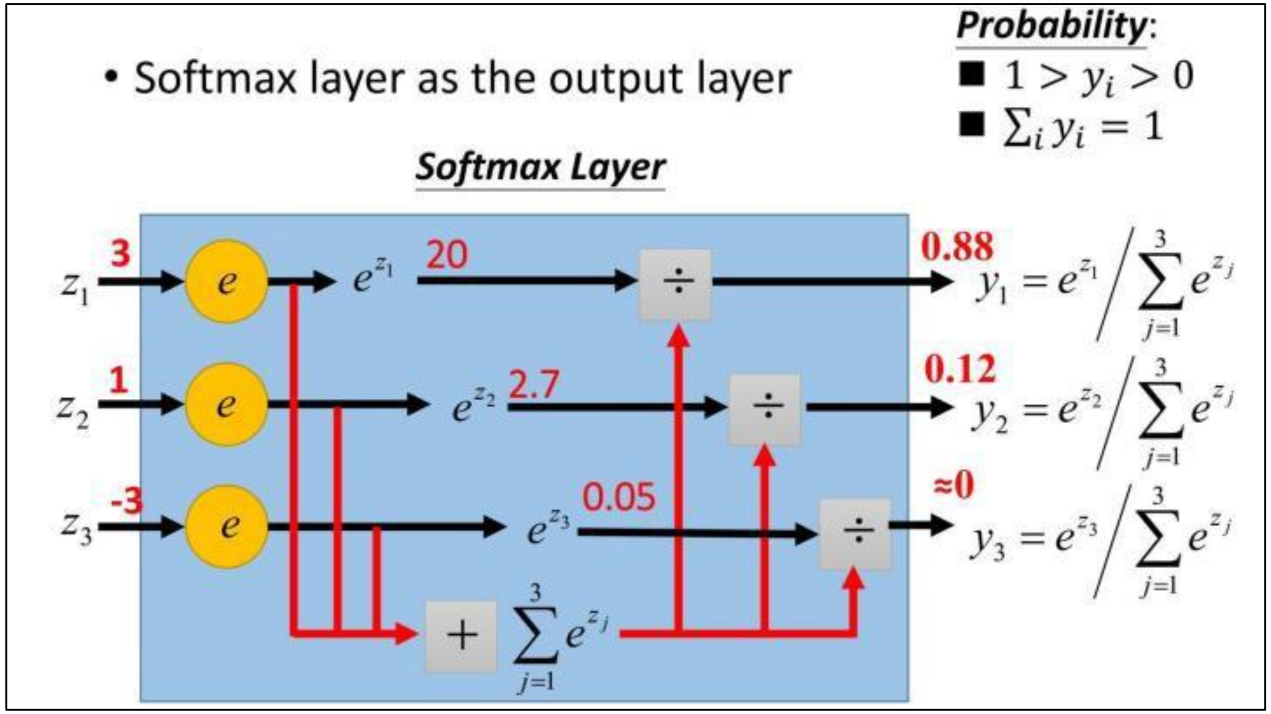
- **The linear function is not used in the hidden layers.**
  We must use non-linear transfer functions in the hidden layer nodes or else the output will only ever end up being a linearly separable solution.

Activation Function — Softmax

- For M-class classification problem, the number of output units is usually M, and the final activation function is usually the **softmax function**

- Formula:

$$g_m^{(K)}(z_m, \mathbf{z}) = \frac{e^{z_m}}{\sum_{\ell=1}^{M} e^{z_\ell}}$$



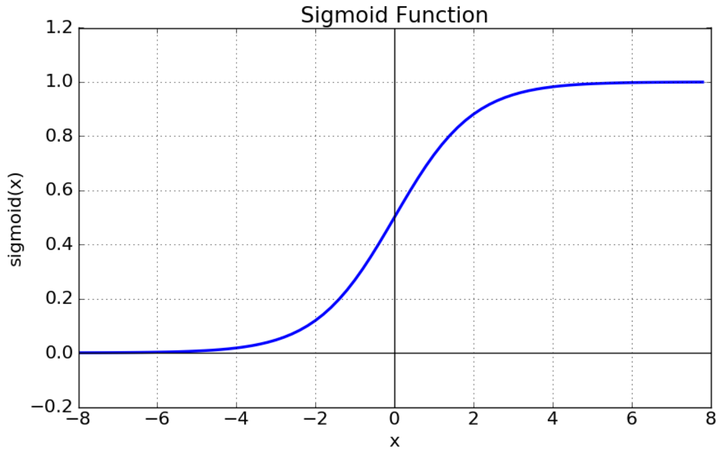
Softmax 示意圖

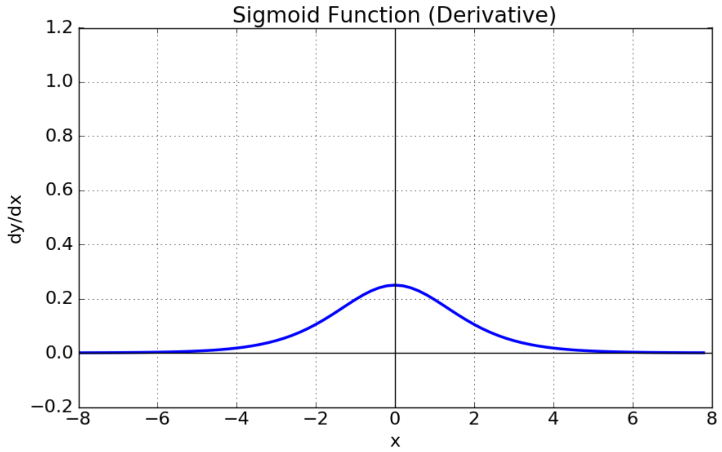
- Property:
  - Range: [0, 1]
  - The total sum of all outputs is 1.

- The output of softmax is a probability distribution.

Activation Function — Sigmoid (or Logistic) Function

- Formula: $f(x_i) = \frac{1}{1+e^{-x_i}}$ $f'(x_i) = f(x_i)(1 - f(x_i))$

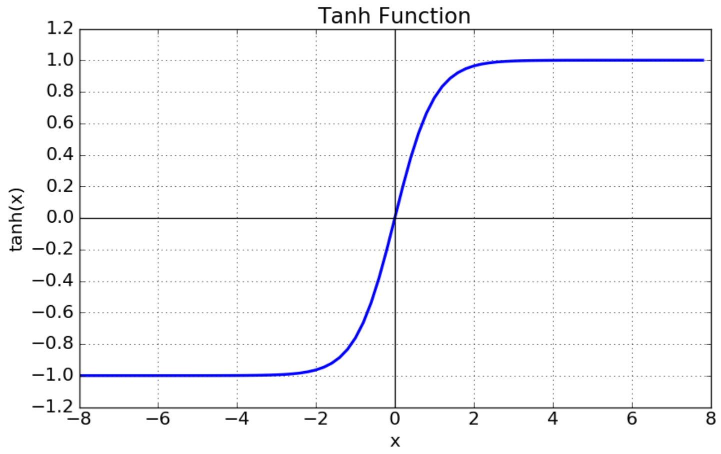


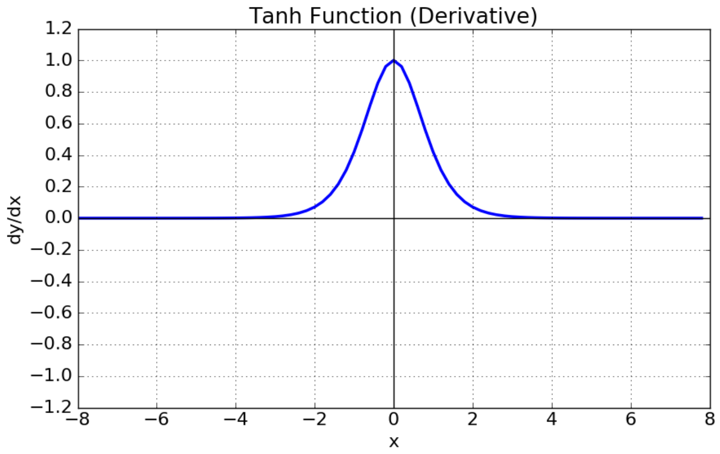


- Property:
  - Range: (0, 1)
  - 一般用於二分類神經網絡（輸出為機率值）
- Issue:
  - 易產生梯度消失，增大NN訓練難度、影響性能
  - exp 指數運算成本高

Activation Function — Hyperbolic Tangent Function (tanh(x))

- Formula: $f(x_i) = \tanh(x_i) = \dfrac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}}$ $\quad f'(x_i) = 1 - \tanh(x_i)^2$

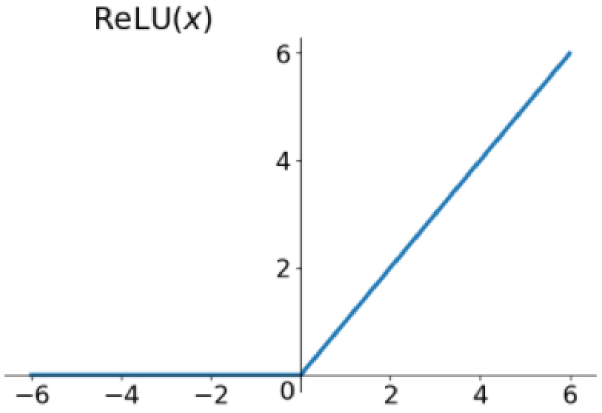


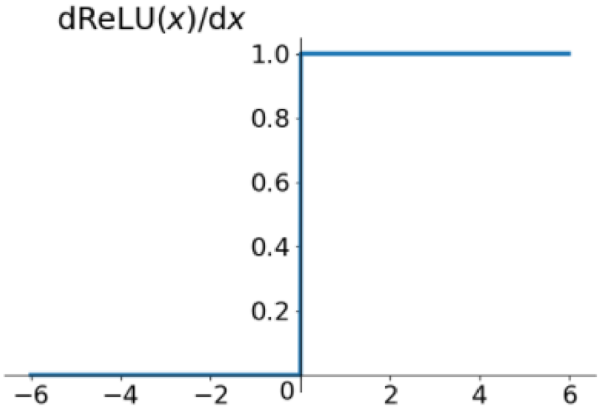


- Property:
  - Range: (-1,1)
- Advantage:
  - 收斂速度比 Sigmoid 快
- Issue:
  - 同樣存在梯度消失問題

Activation Function — Rectifying Linear Unit (ReLU)

- Formula: $\mathrm{ReLU} = \max(0, x)$



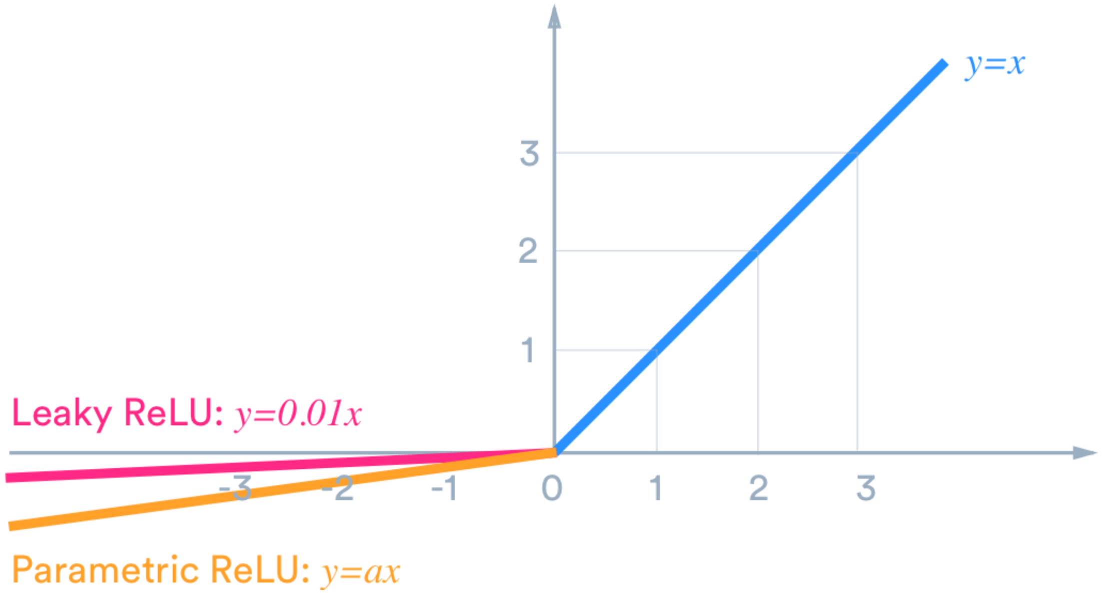


- Property:
  - Range: [0, infinity)
- Advantage:
  - 在正區間解決了梯度消失的問題
  - 計算速度較快，只需要判斷input是否>0
  - 收斂速度遠快於 sigmoid 和 tanh
- Issue:
  - Dying ReLU problem

Activation Function — Parametric ReLU

- Formula:

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



- Property:
  - When $\alpha = 0.01$, it's called Leaky ReLU.
  - Range: $(-\infty, \infty)$
  - Avoid flat spots and accompanying zero gradients, solving the Dying ReLU problem.

2

模型訓練

Fitting a Neural Network

➢ 根據前面的推導，我們的neural network模型可以表示成:

$$f(\boldsymbol{x}_i; W) = w_{\ell 0}^{(k-1)} + \sum_{j=1}^{p_{k-1}} w_{\ell j}^{(k-1)} a_j^{(k-1)}$$

➢ In order to fit our model $f(\boldsymbol{x}_i; W)$, we need to create a loss function（研究者設計的）

- ➢ E.g. $L(y, f(x)) = \frac{1}{2}\,(y - f(x))^2$

➢ After creating our loss function, we might seek to solve the above optimization problem:

$$\min_{\mathcal{W}} \frac{1}{n} \sum_{i=1}^{n} L\left(y_i, f\left(\mathbf{x}_i; \mathcal{W}\right)\right)$$

Fitting a Neural Network

- Problems we might face

Example: OLS

OLS的$loss\ function$ $L(y_i, f(\boldsymbol{x}_i))$是Convex，可用FOC解出最適解

Loss Function of NN

$L(y_i, f(\boldsymbol{x}_i; W))$ 顯然不是Convex

Loss Function of NN

我們無法去使用簡單的FOC解出最適解

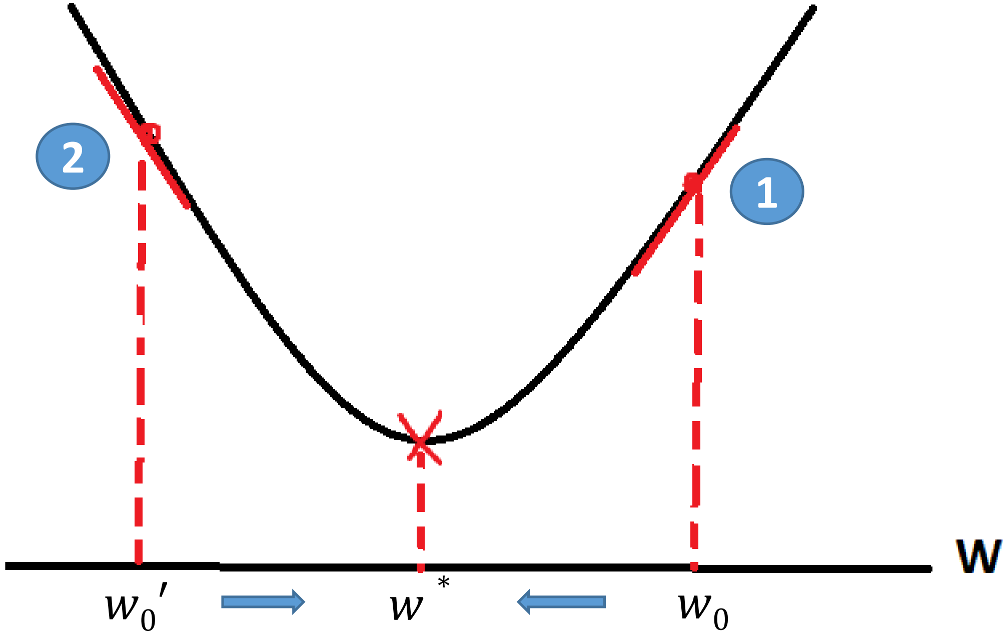
Gradient Descent

因此我們需要採用一些algorithm的解法。這裡我們採用梯度下降法去配適出我們模型的參數。

$$\min_{\mathcal{W}} \frac{1}{n} \sum_{i=1}^{n} L\left(y_i, f\left(\mathbf{x}_i; \mathcal{W}\right)\right)$$

The intuition of Gradient Descent

$$J(w) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, f(x_i; w))$$



- First Step: 給定一個起始值w_0
- Second Step: 計算出$J(w)$的gradient: $\frac{\partial J(w)}{\partial w}$，並將資料$(x_i, y_i)$代入
- Condition1: 當gradient > 0的時候，可以發現我們的weight太大了，因此我們減掉一些
- Condition2: 當gradient < 0的時候，可以發現我們的weight太小了，因此我們加上一些
- 重複這個動作，直到我們的gradient = 0，也就達到我們想要的optimal solution

Gradient Descent

- 將前頁一維的情形推廣至多維可得:

  - $mimize\ \frac{1}{n}\sum_{i=1}^{n} L(y_i, f(x_i; \boldsymbol{W}))$
  - Gradient: $\Delta w_{lj}{}^{(k)} = \frac{1}{n}\sum_{i=1}^{n} \partial L\big(y_i, f(x_i; \boldsymbol{W})\big)/\partial w_{lj}{}^{(k)}$, $k = 1, 2, \ldots, K-1$ $(number\ of\ tatal\ layers: K)$
  - 註1: 因為先累加再取偏微分，與先取偏微分，再累加的結果一樣。

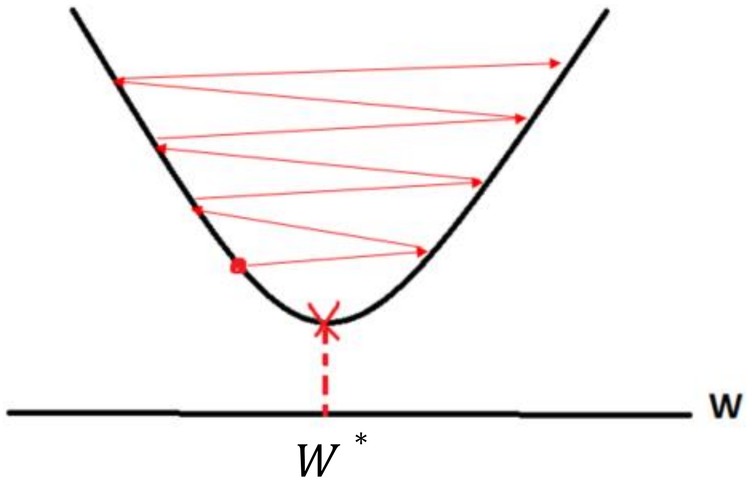
- Given a set of starting values $(w_0{}^{(k)})$ for all the weights, a gradient descent update is:

  - $w_{lj}^{(k)} \leftarrow w_{lj}^{(k)} - \alpha * \Delta w_{lj}{}^{(k)}, k = 1, 2, \ldots, K-1$

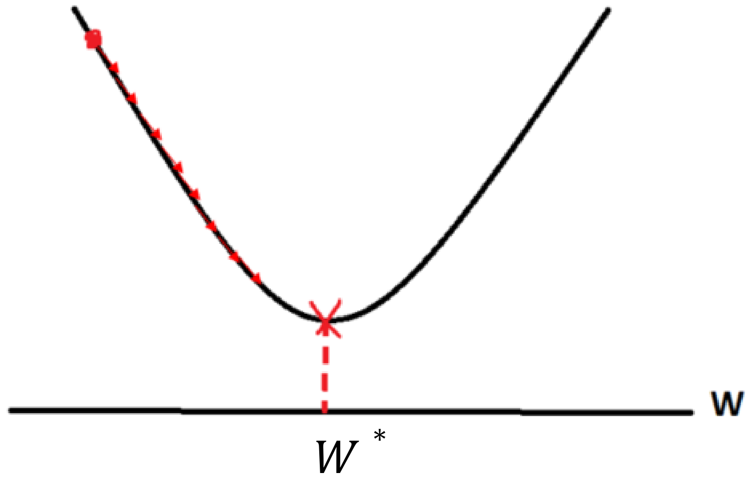
  - $\alpha > 0$ is the learning rate（表示gradient的步長，研究者自行給定）

The size of learning rate

- Condition 1:
  - When α is too big, gradient may overshoot the minimum. It may diverge.



- Condition 2:
  - When α is too small, the converge speed may be too slow.



- 因此我們需要謹慎挑選 α
  - 一般取值介於0、1之間
  - 可以用cross validation去挑選 α

Problems we met in Gradient Descent

The gradient is hard to calculate

$$\Delta w_{lj}{}^{(k)} = \frac{1}{n}\sum_{i=1}^{n} \partial L(y_i, f(x_i; \boldsymbol{W}))/\partial w_{lj}{}^{(k)}$$

Using Chain Rule

$$\frac{\partial L(y, f(x; \boldsymbol{W}))}{\partial w_{lj}{}^{(k)}} = \frac{\partial L}{\partial Z_l{}^{(k+1)}} * \frac{\partial Z_l{}^{(k+1)}}{\partial w_{lj}{}^{(k)}}$$

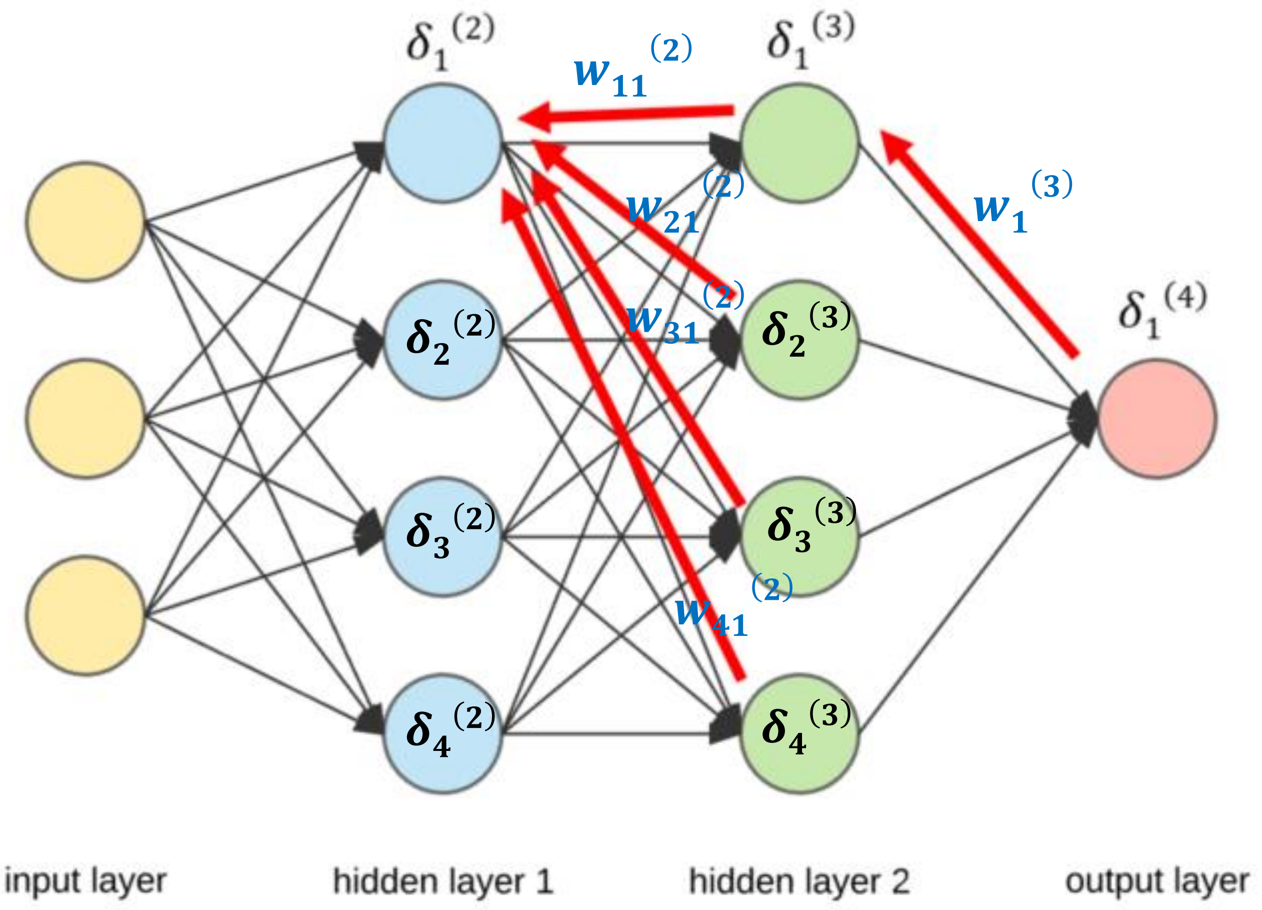
- Since $Z_l{}^{(k+1)}$ is a linear transformation of $a_l{}^{(k)}$ and $w_{lj}{}^{(k)}$, the second part is easy to calculate.
- $Z_l{}^{(k+1)} = w_{l0}{}^{(k)} + \sum_{j=1}^{P_k} w_{lj}{}^{(k)} * a_j{}^{(k)}$
- $\frac{\partial Z_l{}^{(k+1)}}{\partial w_{lj}{}^{(k)}} = a_j{}^{(k)}$

Backpropagation

- In order to solve $\frac{\partial L}{\partial Z_l{}^{(k+1)}}$
- We find a way called backpropagation

Intuition of Backpropagation

- 首先，定義$\frac{\partial L}{\partial Z_l^{(k)}}$ 為error term $\delta_l^{(k)}$
  - 這個error term代表了每個節點（node）的值的變化，會對誤差造成多少影響



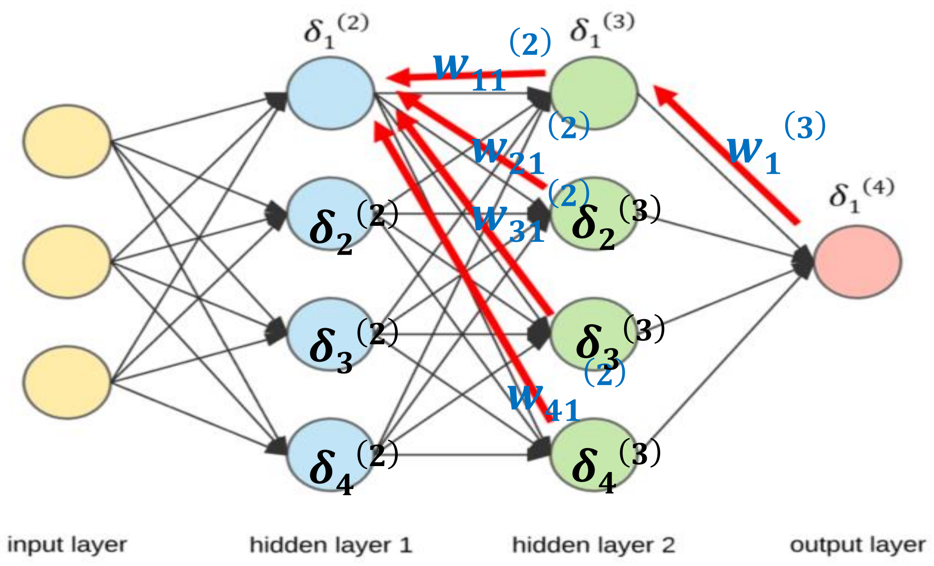
- 先求出output layer的error term $\delta_1^{(4)}$
- 接著依照目前的權重以及$\delta_1^{(4)}$，推出第二層hidden layer的error term $\delta^{(3)}$
- 再依據目前的權重以及$\delta^{(3)}$ ，推出第一層hidden layer的error term $\delta^{(2)}$

Backpropagation 數學推導

- For each output unit l in the output layer（$Layer_K$）:

$$\delta_l^{(K)} = \frac{\partial L}{\partial Z_l^{(K)}} = \frac{\partial L}{\partial a_l^{(K)}} * \frac{\partial a_l^{(K)}}{\partial Z_l^{(K)}} = \frac{\partial L}{\partial a_l^{(K)}} * g'^{(K)}(z_l^{(K)})$$

  - 註1: $a_l^{(K)} = g^{(K)}(z_l^{(K)})$
  - 註2: $g'^{(K)}(z_l^{(K)})$是activation function的一階微分



- Example: If the loss function is: $L(y, f(x) = a_l^{(K)}) = \frac{1}{2} * (y - a_l^{(K)})^2$
  - Then $\delta_l^{(K)} = \frac{1}{2} * 2(y - a_l^{(K)}) * (-1) * g'^{(K)}(z_l^{(K)})$
  - $y - a_l^{(K)}$ is the residual which we can calculate after getting $a_l^{(K)}$
  - $g'^{(K)}(z_l^{(K)})$ 則是activation function的一階微分，代入 $z_l^{(K)}$即可求出

Backpropagation 數學推導

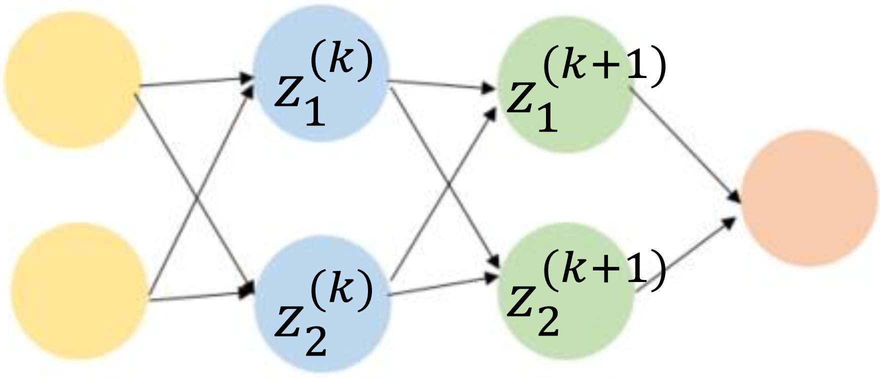
- For hidden layers $Layer_{k}$, $k = K-1, K-2, \dots\dots 2$ and each node l :

$$\delta_l^{(k)} = \frac{\partial L}{\partial Z_l^{(k)}} \; and \; \delta_l^{(k+1)} = \frac{\partial L}{\partial Z_l^{(k+1)}}$$

- Since $L(y, f(x; \boldsymbol{W}))$ is a function of $Z_j^{(k+1)}$ for $j = 1,2, \dots, P_{k+1}$, we can rewrite $\delta_l^{(k)}$ by Chain Rule:

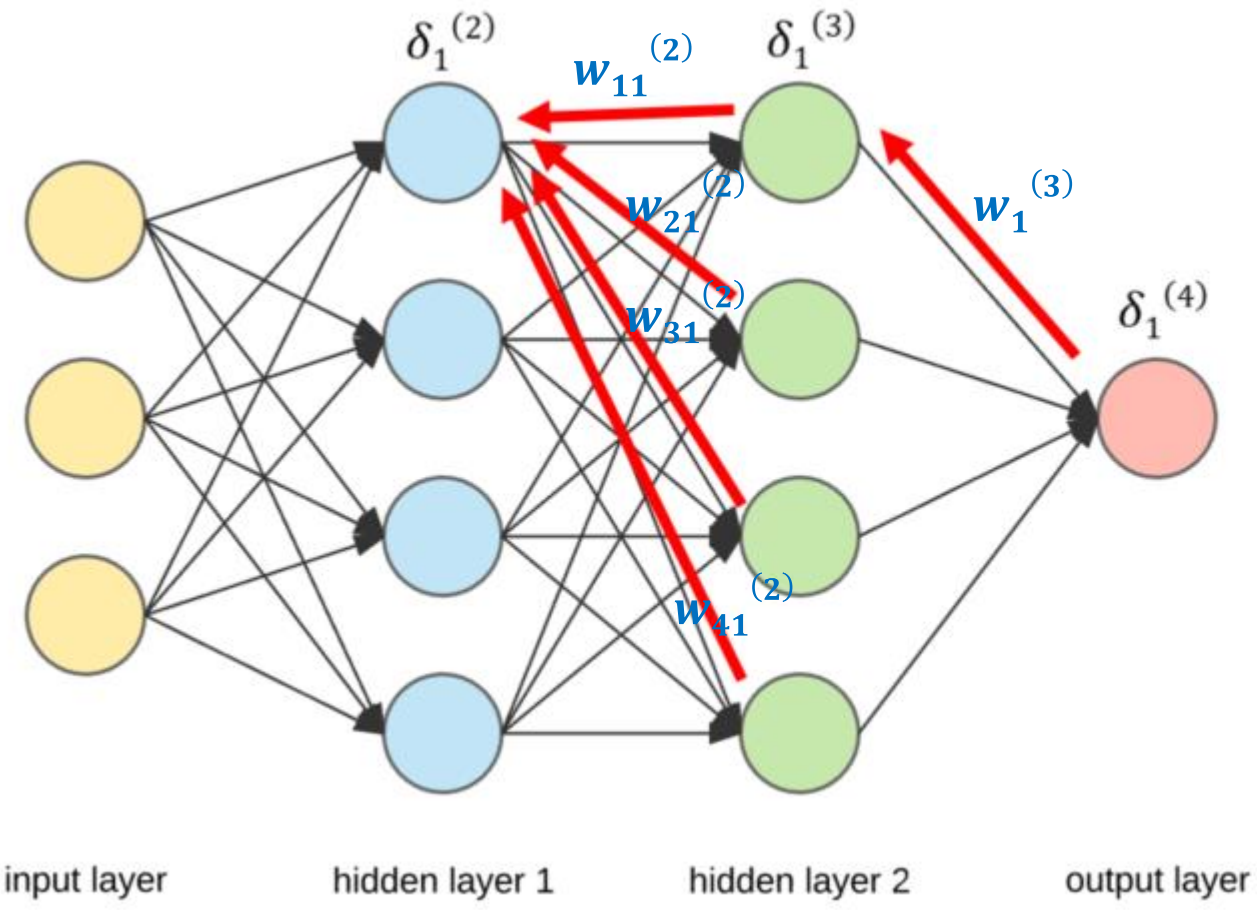
$$\delta_l^{(k)} = \sum_{j=1}^{P_{k+1}} \frac{\partial L}{\partial Z_j^{(k+1)}} * \frac{\partial Z_j^{(k+1)}}{\partial Z_l^{(k)}} = \sum_{j=1}^{P_{k+1}} \delta_j^{(k+1)} * \frac{\partial Z_j^{(k+1)}}{\partial Z_l^{(k)}}$$

Backpropagation 數學推導



- Next, we need to find out $\frac{\partial Z_j^{(k+1)}}{\partial Z_l^{(k)}}$
- $Z_j^{(k+1)} = w_{j0}^{(k)} + \sum_{l=0}^{P_k} w_{jl}^{(k)} * a_l^{(k)} = w_{j0}^{(k)} + \sum_{l=0}^{P_k} w_{jl}^{(k)} * g^{(k)}(Z_l^{(k)})$
- Example: $\frac{\partial Z_j^{(k+1)}}{\partial Z_1^{(k)}} = \frac{\partial(w_{j0}^{(k)} * g^{(k)}(Z_0^{(k)}) + w_{j1}^{(k)} * g^{(k)}(Z_1^{(k)}) + \cdots + w_{jP_k}^{(k)} * g^{(k)}(Z_{P_k}^{(k)}))}{\partial Z_1^{(k)}} = w_{j1}^{(k)} * g'^{(k)}(Z_1^{(k)})$
- Then we have $\frac{\partial Z_j^{(k+1)}}{\partial Z_l^{(k)}} = w_{jl}^{(k)} * g'^{(k)}(Z_l^{(k)})$ and $\delta_l^{(k)} = \sum_{j=1}^{P_{k+1}} \delta_j^{(k+1)} * \frac{\partial Z_j^{(k+1)}}{\partial Z_l^{(k)}}$
- Finally we got $\delta_l^{(k)} = \{\sum_{j=1}^{P_{k+1}} w_{jl}^{(k)} * \delta_j^{(k+1)}\} * g'^{(k)}(Z_l^{(k)})$

Backpropagation and Gradient Descent



- ***Feedforward pass:*** 將起始值$\boldsymbol{W}_0^{(k)}$以及每個training pair$(\boldsymbol{x}_i, y_i)$代入，求出每個Layer的output $a_l^{(k)}$, k=2,3,4

- ***BackPropagation:*** 先求出output layer的 $\delta_l^{(K)}$ ，再依序反向傳播推出hidden layer的 $\delta_l^{(k)}$

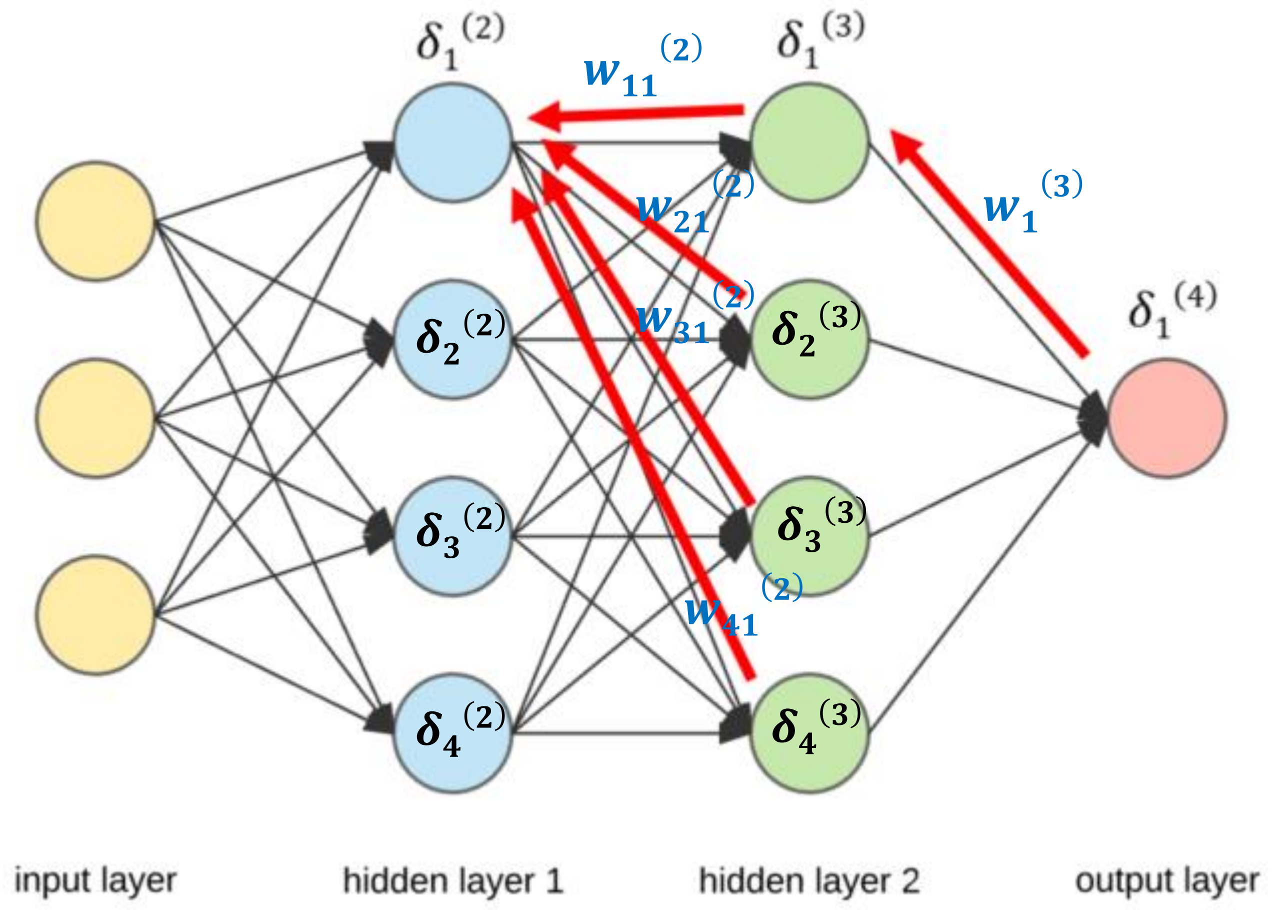
- $\delta_l^{(K)} = \frac{\partial L}{\partial a_l^{(K)}} * g'^{(K)}(z_l^{(K)})$

- $\delta_l^{(k)} = \{\sum_{j=1}^{P_{k+1}} w_{jl}^{(k)} * \delta_j^{(k+1)}\} * g'^{(k)}(z_l^{(k)})$

- $\frac{\partial L(y, f(x;\boldsymbol{W}))}{\partial w_{lj}^{(k)}} = \frac{\partial L}{\partial Z_l^{(k+1)}} * \frac{\partial Z_l^{(k+1)}}{\partial w_{lj}^{(k)}} = \delta_l^{(k+1)} * a_j^{(k)}$

- $w_{lj}^{(k)} \leftarrow w_{lj}^{(k)} - \alpha * \Delta w_{lj}^{(k)}$
  - $\Delta w_{lj}^{(k)} = \frac{1}{n} \sum_{i=1}^{n} \partial L(y_i, f(x_i; \boldsymbol{W}))/\partial w_{lj}^{(k)}$

Backpropagation and Gradient Descent



- ***Feedforward pass:*** 將起始值$\boldsymbol{W}_0^{(k)}$以及每個training pair$(\boldsymbol{x}_i, y_i)$代入，求出每個Layer的output $a_l^{(k)}$, k=2,3,4

- ***BackPropagation:*** 先求出output layer的 $\delta_l^{(K)}$ ，再依序反向傳播推出hidden layer的 $\delta_l^{(k)}$

- $\delta_l^{(K)} = \frac{\partial L}{\partial a_l^{(K)}} * g'^{(K)}(z_l^{(K)})$

- $\delta_l^{(k)} = \{\sum_{j=1}^{P_{k+1}} w_{jl}^{(k)} * \delta_j^{(k+1)}\} * g'^{(k)}(z_l^{(k)})$

- $\frac{\partial L(y, f(x; \boldsymbol{W}))}{\partial w_{lj}^{(k)}} = \frac{\partial L}{\partial Z_l^{(k+1)}} * \frac{\partial Z_l^{(k+1)}}{\partial w_{lj}^{(k)}} = \delta_l^{(k+1)} * a_j^{(k)}$

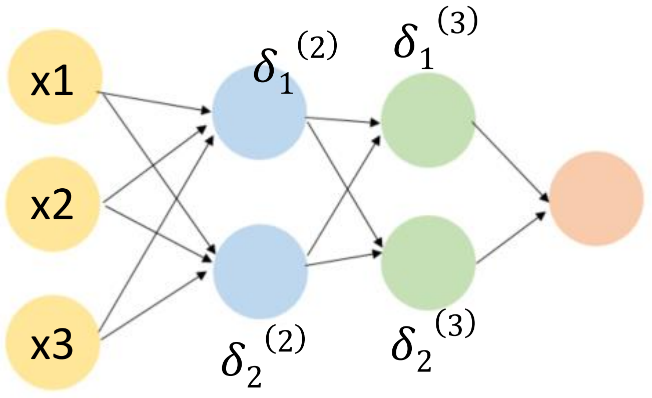
- $w_{lj}^{(k)} \leftarrow w_{lj}^{(k)} - \alpha * \Delta w_{lj}^{(k)}$
  - $\Delta w_{lj}^{(k)} = \frac{1}{n} \sum_{i=1}^{n} \partial L(y_i, f(x_i; \boldsymbol{W})) / \partial w_{lj}^{(k)}$

Gradient Descent: the initial weights

要做Gradient Descent時，一開始必須給定一組權重的起始值

- 如果**給定起始值皆為0**，將導致Gradient一開始就為0，因此權重永遠不會更新
  - Example: $\frac{\partial L(y,f(x;\boldsymbol{W}))}{\partial w_{lj}^{(k)}} = \delta_l^{(k+1)} * a_j^{(k)} = \{\sum_{j=1}^{P_{k+2}} w_{jl}^{(k+1)} * \delta_j^{(k+2)}\} * g'^{(k)}(z_l^{(k)}) * a_j^{(k)} = 0$

- 如果**給定起始值皆為同一數值**，將導致每個layer下的neurons的輸出都一樣（等同於多個neurons只學到一種pattern）。每個layer下各個權重的Gradient都一樣，最後每一個layer下各個權重更新後仍為同一數值
  - Example: if all weights set to w, we have:
    - $Z_1^{(2)} = w + w * x_1 + w * x_2 + w * x_3 = Z_2^{(2)} \rightarrow a_1^{(2)} = a_2^{(2)}$
    - $\delta_1^{(2)} = (w * \delta_1^{(3)} + w\delta_2^{(3)}) * g'^{(2)}(Z_1^{(2)})$
    - $\delta_2^{(2)} = (w * \delta_1^{(3)} + w\delta_2^{(3)}) * g'^{(2)}(Z_2^{(2)})$
    - $\delta_1^{(2)} = \delta_2^{(2)}$
    - $\frac{\partial L(y,f(x;\boldsymbol{W}))}{\partial w_{lj}^{(k)}} = \delta_l^{(k+1)} * a_j^{(k)}$

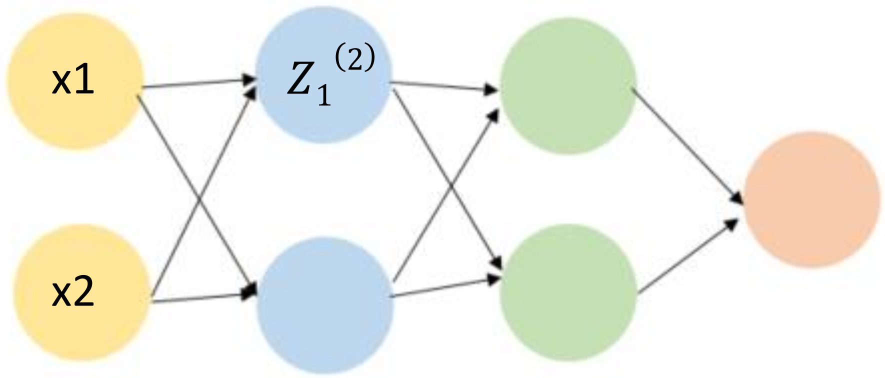


- In general, we would randomly select our initial weights from Uniform or Gaussian Distribution to break the symmetric.

Gradient Descent: feature scaling

做Gradient Descent時，如果沒有事先標準化每個features，則權重的起始值不同，會導致很不一樣的結果

- Example:
  X1: 年收入 (range from 0 to 5000000 dollars)
  X2: 平均每日工時 (range from 0 to 24 hours)
- $Z_1^{(2)} = w_{10}^{(1)} + w_{11}^{(1)} * x_1 + w_{12}^{(1)} * x_2$



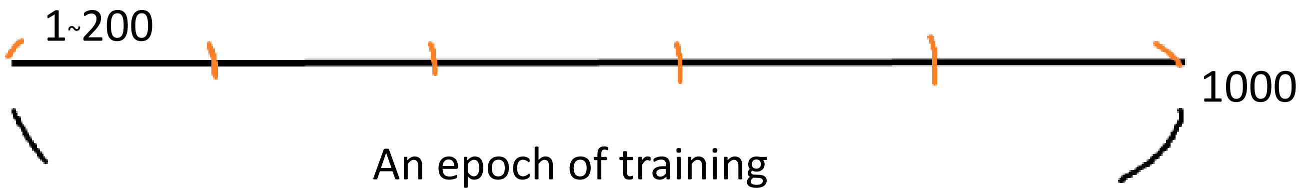
- For the first training pair $(x1 = 1000000, x2 = 8)$
  - if the initial weight is (0, 0.5, -0.5)
  - $Z_1^{(2)} = 0 + 0.5 * 1000000 - 0.5 * 8 = 499996$
  - If the initial weight is (0, -0.5, 0.5)
  - $Z_1^{(2)} = 0 - 0.5 * 1000000 + 0.5 * 8 = -499996$

- 可以發現 $Z_1^{(2)}$ 會因為起始值的不同而劇烈變動，同樣的$a_1^{(2)}$也會因起始值的不同而劇烈變動
- $\frac{\partial L(y, f(x;W))}{\partial w_{lj}^{(k)}} = \delta_l^{(k+1)} * a_j^{(k)}$ 也因起始值的不同而劇烈變動，這將導致每次隨著起始值的不同，Gradient Descent的結果可能會非常不一樣

Stochastic Gradient Descent

- 回顧一般的Gradient Descent
- $w_{lj}{}^{(k)} \leftarrow w_{lj}{}^{(k)} - \alpha * \Delta w_{lj}{}^{(k)}$
  - $\Delta w_{lj}{}^{(k)} = \frac{1}{N}\sum_{i=1}^{N} \partial L\left(y_i, f(x_i; \boldsymbol{W})\right)/\partial w_{lj}{}^{(k)}$
- 一般的Gradient Descent將全部的training pairs（$(\boldsymbol{x}_i, y_i), i = 1, \ldots, N$）代入後，只做一次的權重更新

- Stochastic Gradient Descent提出了一個更有效率的方法，一次只將**部分的training pairs**（$(\boldsymbol{x}_i, y_i), i = 1, \ldots, n$）代入後，就做一次的權重更新（將資料分批下去做訓練）
- 這個部分的n個training pairs 研究者可以自行給定，我們稱其為***Batch Size***
- 利用一個Batch Size的訓練集去做一次訓練，我們稱其為一個***iteration***。而完整的跑完整個training set我們稱為一個***epoch***
  - Example: 假設資料共有1000筆，我們設計一個Batch大小為200，則完成一個epoch的訓練需要5次iteration
  - Stochastic Gradient Descent在這個例子裡做了5次的權重更新，而一般的Gradient Descent只更新了一次



3

NN實作

手寫辨識（一）：以NN為例

```python
# Load mnist

from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) \
    = mnist.load_data()

# Take a look on mnist

train_labels[0:10]

import matplotlib.pyplot as plt

digit = train_images[9] # Try 2 and 9
plt.imshow(digit, cmap = plt.cm.binary)
plt.show()
```

從MINST(美國國家標準局)
資料庫導入範例所需資料

Train資料 ： 60000筆公務員手寫數字

train_images
60000個28*28的array

train_labels
為一60000*1的array
紀錄train_images對應之正確數字

Test資料 ： 10000筆高中生手寫數字

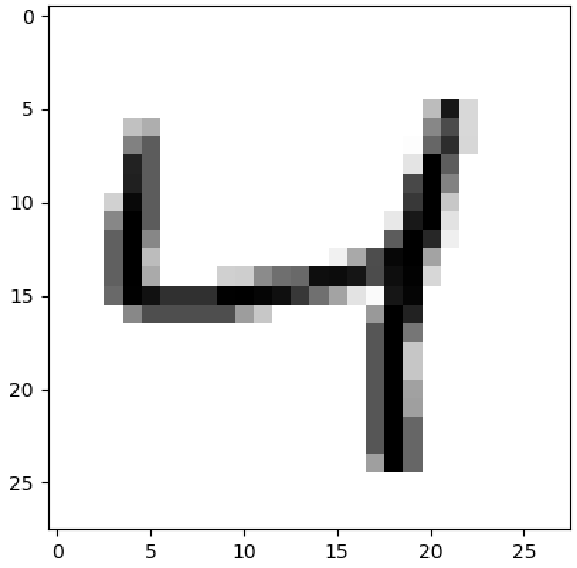
test_images
10000個28*28的array

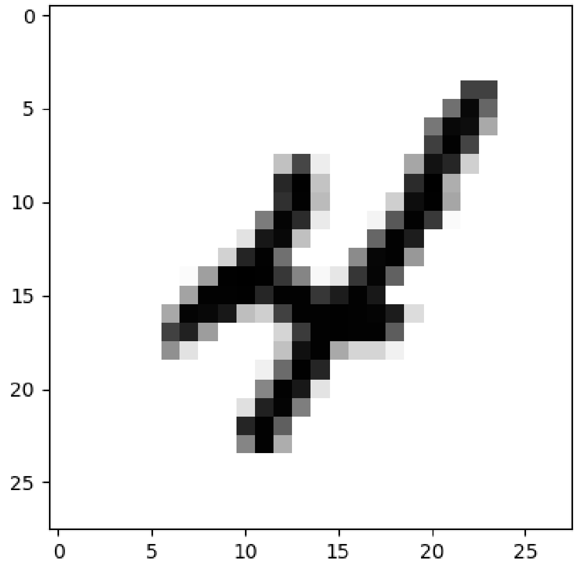
test_labels
為一10000*1的array
紀錄test_images對應之正確數字

```
In [30]: train_labels[0:10]
Out[30]: array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4], dtype=uint8)
```

```
import matplotlib.pyplot as plt

digit = train_images[2]
plt.imshow(digit, cmap = plt.cm.binary)
plt.show()
```

```
import matplotlib.pyplot as plt

digit = train_images[9]
plt.imshow(digit, cmap = plt.cm.binary)
plt.show()
```

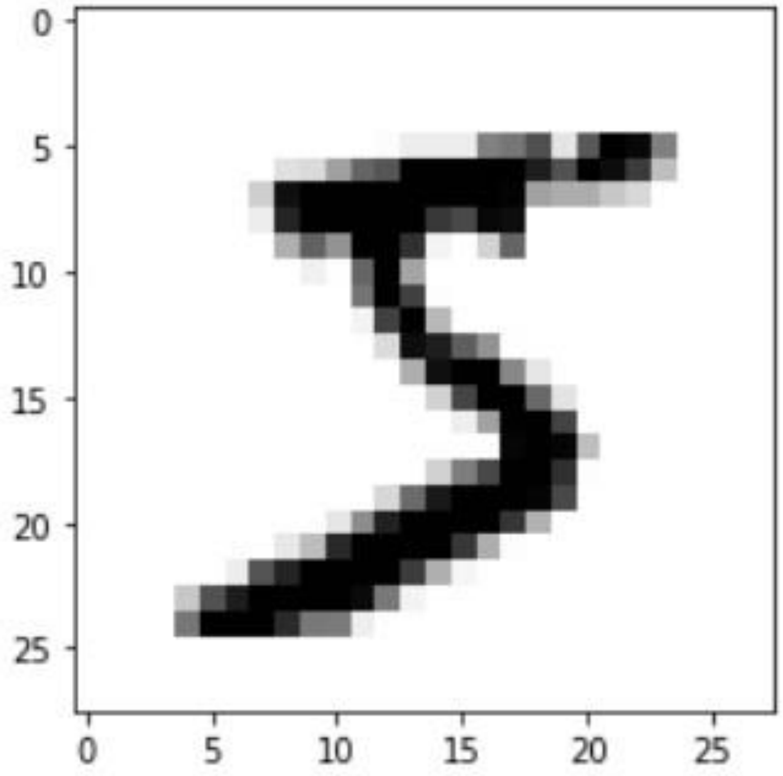




Example: Image

每一個pixel的值介於 0~255
值越大的地方代表顏色越深

Input 為一個 28*28 的灰階圖像



透過matrix
將圖像轉換成數值

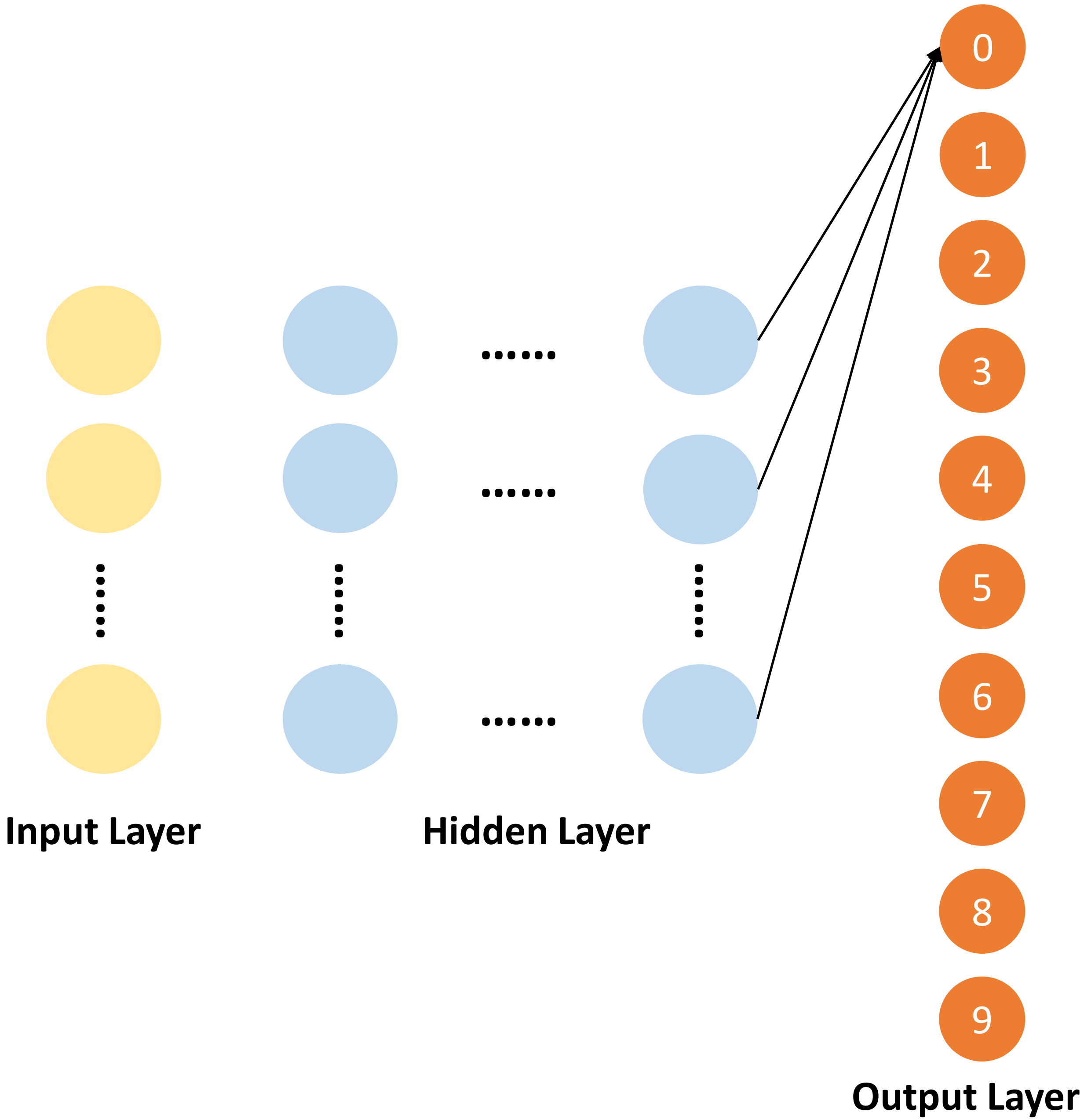
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	3	18	18	18	126	136	175	26	166	255	247	127	0	0	0	0
6	0	0	0	0	0	0	0	0	30	36	94	154	170	253	253	253	253	253	225	172	253	242	195	64	0	0	0	0
7	0	0	0	0	0	0	0	49	238	253	253	253	253	253	253	253	253	251	93	82	82	56	39	0	0	0	0	0
8	0	0	0	0	0	0	0	18	219	253	253	253	253	253	198	182	247	241	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	80	156	107	253	253	205	11	0	43	154	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	14	1	154	253	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	139	253	190	2	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	11	190	253	70	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	35	241	225	160	108	1	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	81	240	253	253	119	25	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	186	253	253	150	27	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	93	252	253	187	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	249	253	249	64	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	46	130	183	253	253	207	2	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	39	148	229	253	253	253	250	182	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	24	114	221	253	253	253	253	201	78	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	23	66	213	253	253	253	253	198	81	2	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	18	171	219	253	253	253	253	195	80	9	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	55	172	226	253	253	253	253	244	133	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	136	253	253	253	212	135	132	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

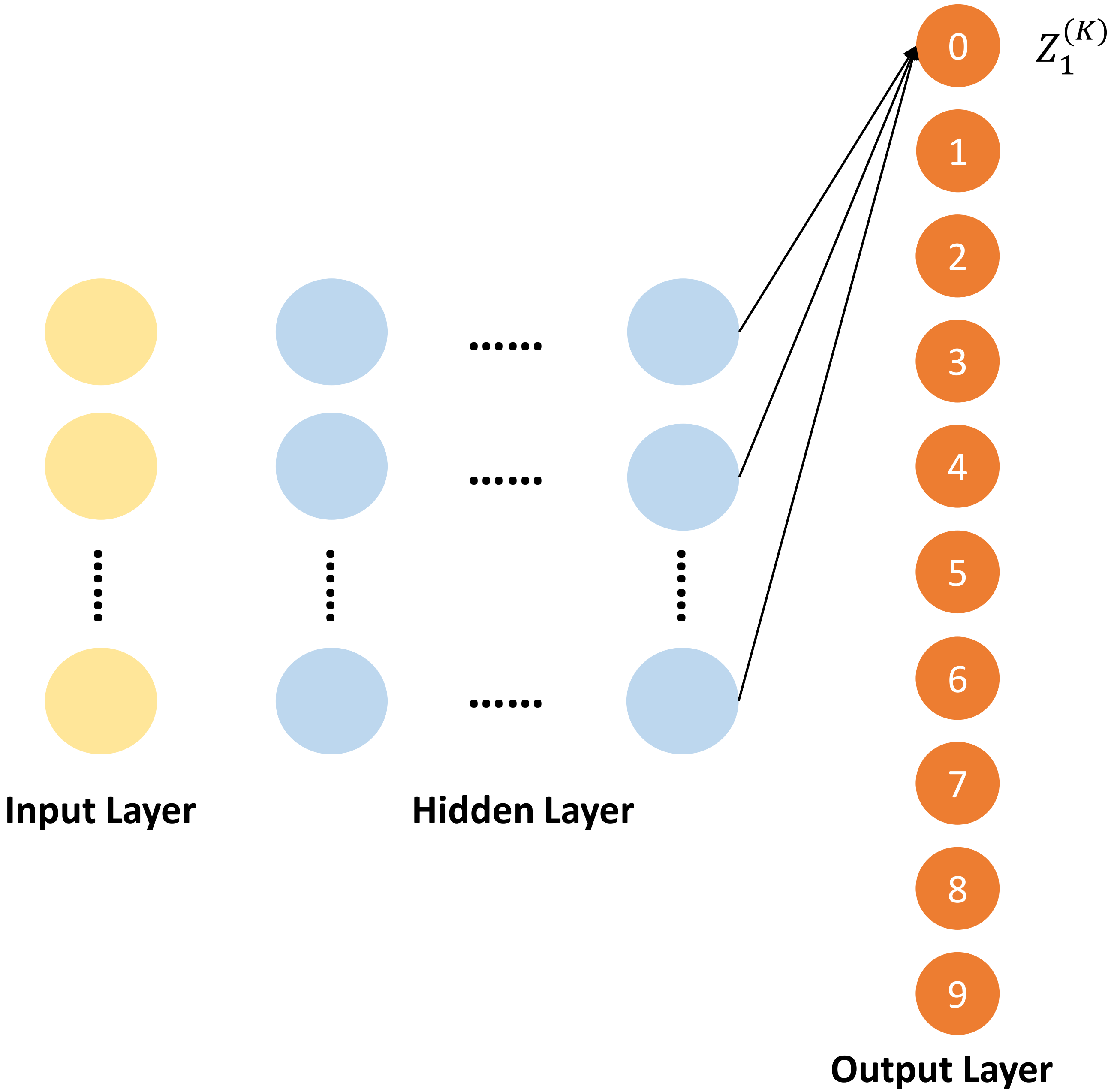
```python
# Scale and Labels

train_images_ = train_images.reshape((60000, 28 * 28))
train_images_ = train_images_.astype('float32') / 255

test_images_ = test_images.reshape((10000, 28 * 28))
test_images_ = test_images_.astype('float32') / 255

from keras.utils import to_categorical

train_labels_ = to_categorical(train_labels)
test_labels_ = to_categorical(test_labels)
```

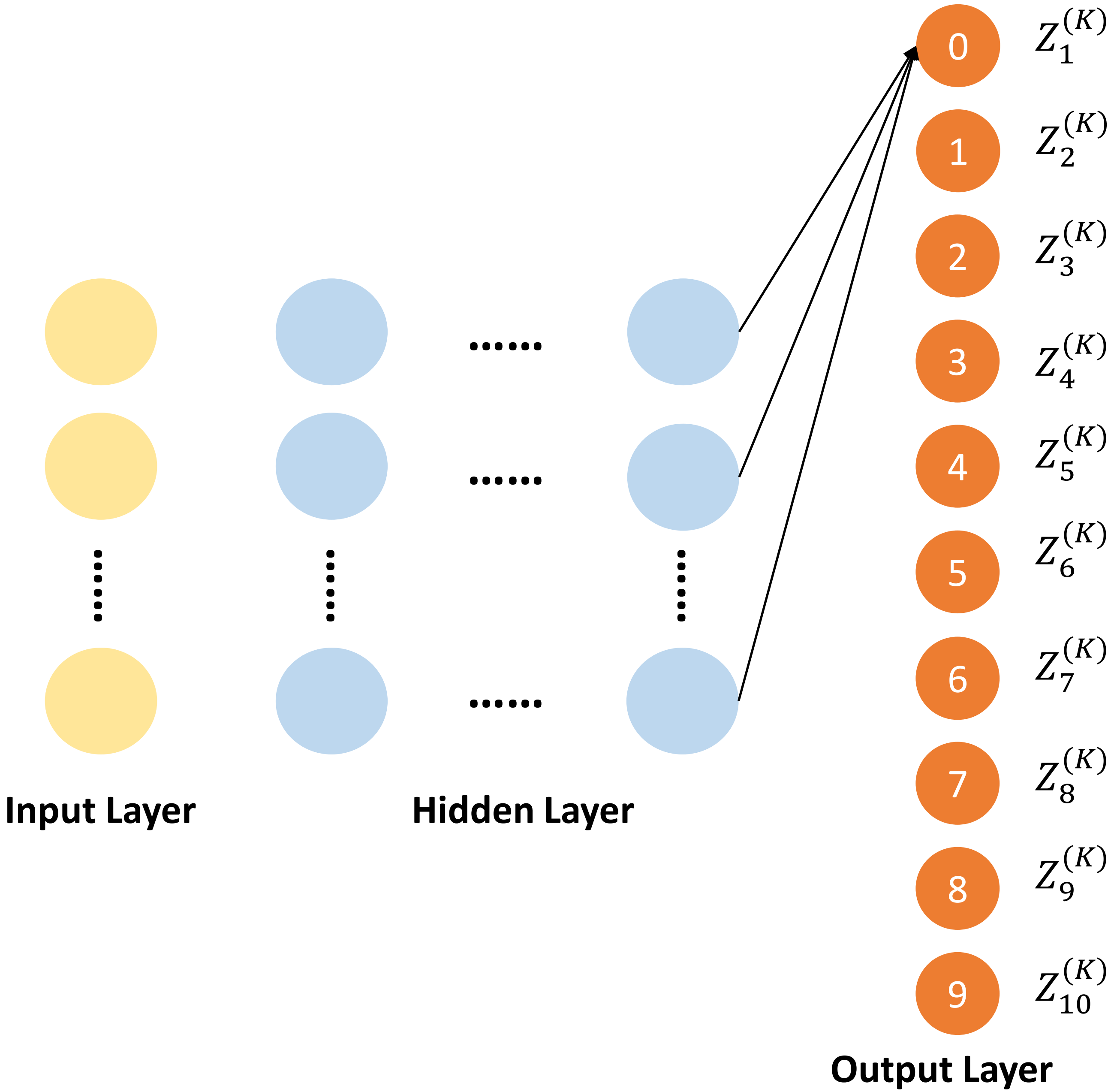
train_labels - NumPy array

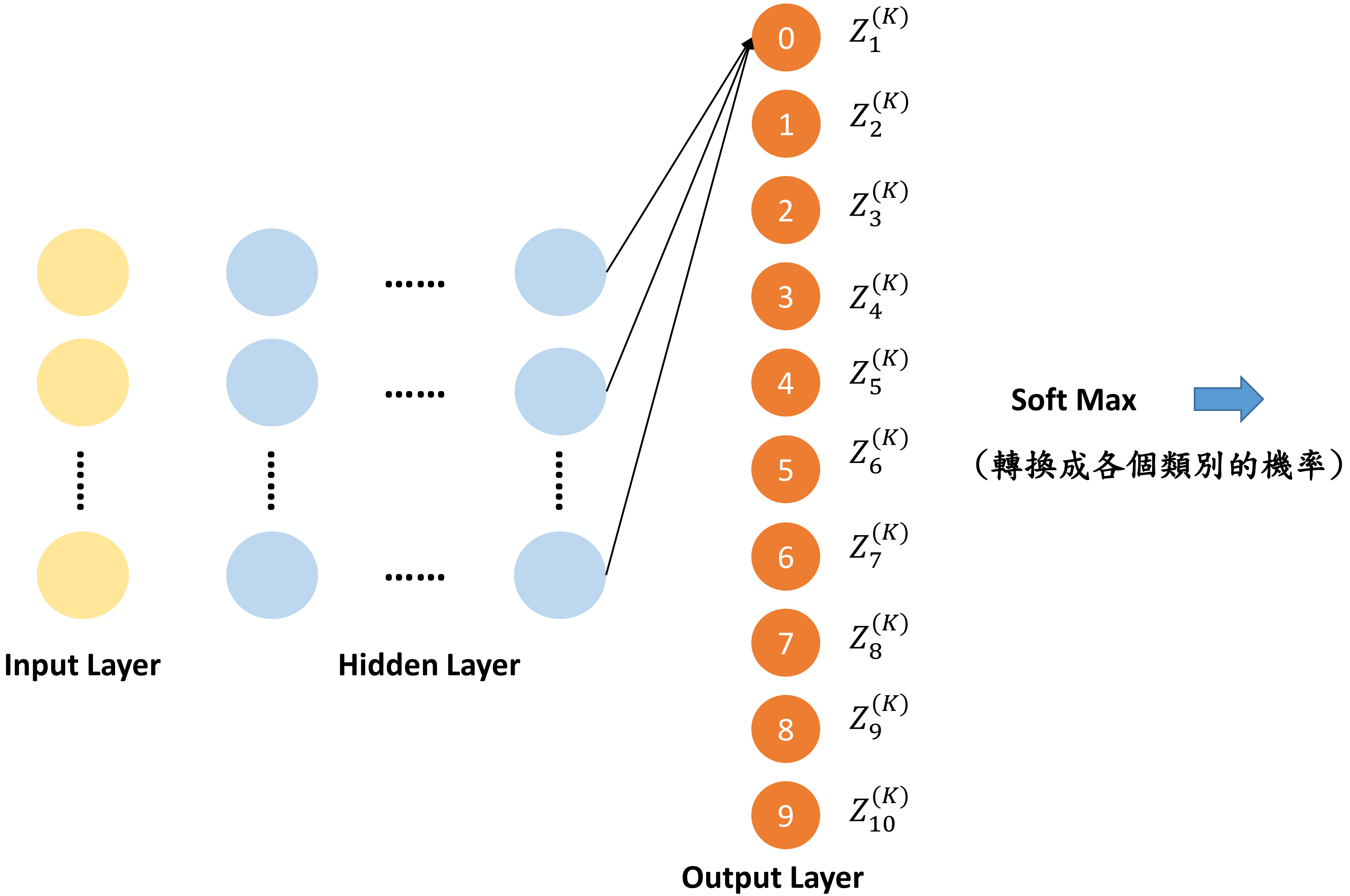
	0
0	5
1	0
2	4
3	1
4	9
5	2
6	1
7	3
8	1
9	4
10	3
11	5
12	3
13	6
14	1
15	7

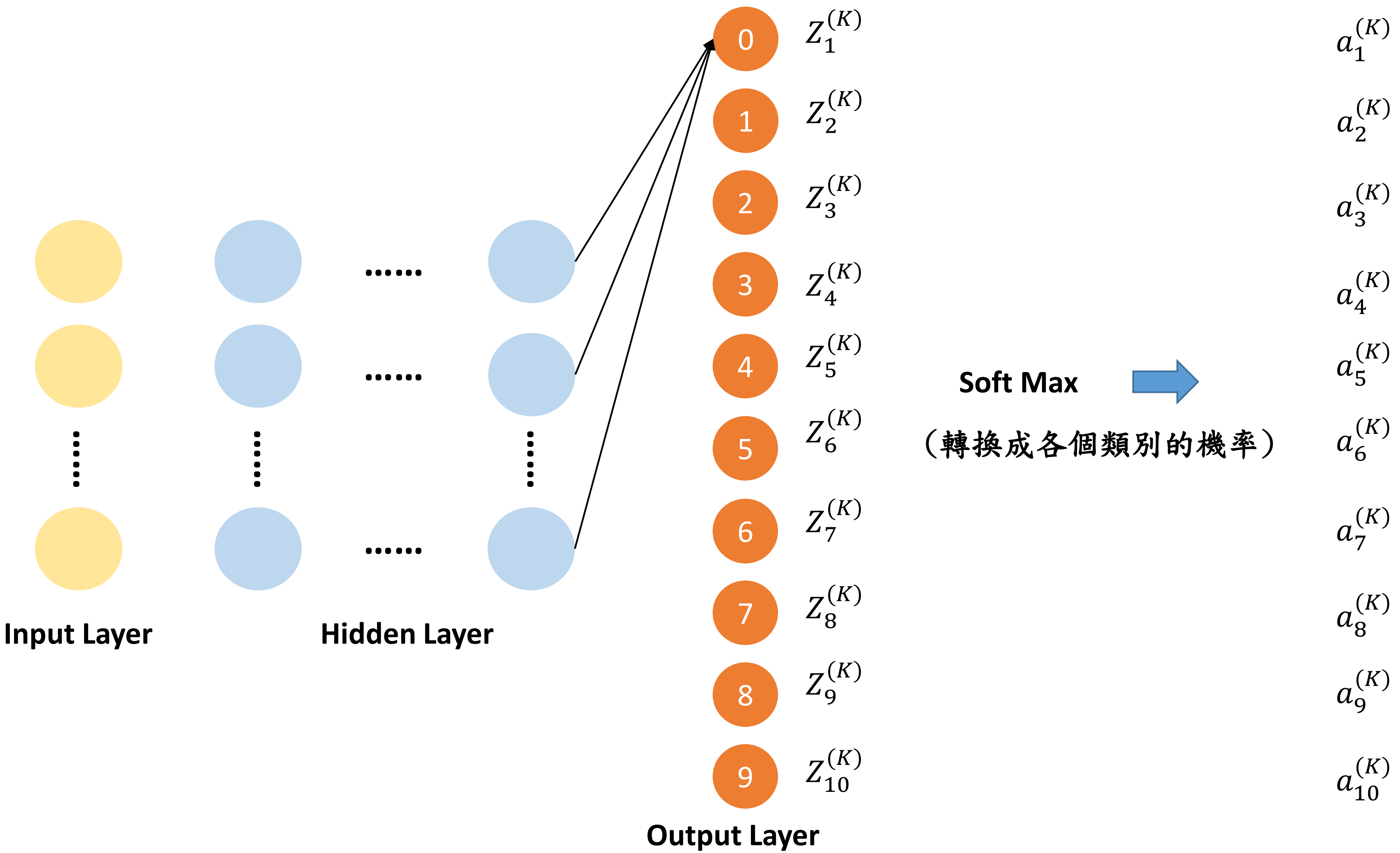
- 利用to_categorical函數轉換labels資料


0
1
2
3
4
5
6
7
8
9
Input Layer
Hidden Layer
Output Layer


$Z_1^{(K)}$
0
1
2
3
4
5
6
7
8
9
Input Layer
Hidden Layer
Output Layer


0
1
2
3
4
5
6
7
8
9
$Z_1^{(K)}$
$Z_2^{(K)}$
$Z_3^{(K)}$
$Z_4^{(K)}$
$Z_5^{(K)}$
$Z_6^{(K)}$
$Z_7^{(K)}$
$Z_8^{(K)}$
$Z_9^{(K)}$
$Z_{10}^{(K)}$
Input Layer
Hidden Layer
Output Layer


0
1
2
3
4
5
6
7
8
9
$Z_1^{(K)}$
$Z_2^{(K)}$
$Z_3^{(K)}$
$Z_4^{(K)}$
$Z_5^{(K)}$
$Z_6^{(K)}$
$Z_7^{(K)}$
$Z_8^{(K)}$
$Z_9^{(K)}$
$Z_{10}^{(K)}$
Input Layer
Hidden Layer
Output Layer
Soft Max
（轉換成各個類別的機率）



0
1
2
3
4
5
6
7
8
9
$Z_1^{(K)}$
$Z_2^{(K)}$
$Z_3^{(K)}$
$Z_4^{(K)}$
$Z_5^{(K)}$
$Z_6^{(K)}$
$Z_7^{(K)}$
$Z_8^{(K)}$
$Z_9^{(K)}$
$Z_{10}^{(K)}$
Input Layer
Hidden Layer
Output Layer
Soft Max
（轉換成各個類別的機率）
$a_1^{(K)}$
$a_2^{(K)}$
$a_3^{(K)}$
$a_4^{(K)}$
$a_5^{(K)}$
$a_6^{(K)}$
$a_7^{(K)}$
$a_8^{(K)}$
$a_9^{(K)}$
$a_{10}^{(K)}$

➢ 將輸入資料轉為one hot形式

Output Layer的Soft Max將數值轉換成各個類別的機率，形成一個10*1的Vector。為了計算Loss，我們將原始的labels（0~9）轉為one hot形式的類別資料，也形成一個10*1的Vector。

train_labels - NumPy array

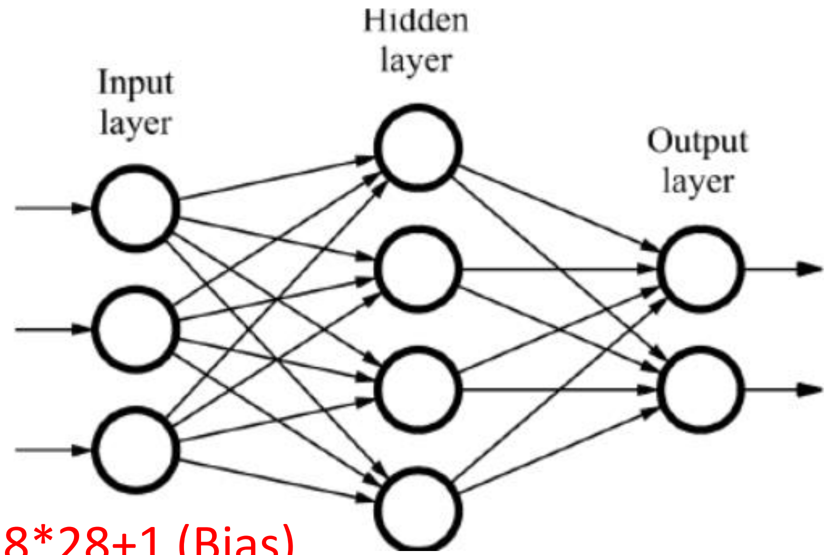
	0
0	5
1	0
2	4
3	1
4	9
5	2
6	1
7	3
8	1
9	4
10	3
11	5
12	3
13	6
14	1
15	7

train_labels_ - NumPy array

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0	0	0
3	0	1	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	1
5	0	0	1	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0	0	0
7	0	0	0	1	0	0	0	0	0	0
8	0	1	0	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0	0	0	0
10	0	0	0	1	0	0	0	0	0	0
11	0	0	0	0	0	1	0	0	0	0
12	0	0	0	1	0	0	0	0	0	0
13	0	0	0	0	0	0	1	0	0	0
14	0	1	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	1	0	0

```python
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation = 'relu',
                         input_shape = (28 * 28, )))
network.add(layers.Dense(10, activation = 'softmax'))

network.summary()
```

Sequential: Feed-forward NN



28*28+1 (Bias)

512

```
In [39]: network.summary()
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

= (28*28+1)*512

= (512+1)*10

```python
#以compile函數定義損失函數(loss)、優化函數(optimizer)及成效衡量指標(mertrics)
network.compile(optimizer = 'SGD',
                loss = 'categorical_crossentropy',
                metrics = ['accuracy'])

# Besides SGD, we may also try rmsprop, Adam, or others.

# Train

network.fit(train_images_, train_labels_,
            epochs = 5, batch_size = 128)
```

```
In [66]: network.fit(train_images_, train_labels_,
    ...:             epochs = 5, batch_size = 128)
Epoch 1/5
60000/60000 [==============================] - 4s 67us/step - loss: 0.3188 - accuracy: 0.9112
Epoch 2/5
60000/60000 [==============================] - 4s 67us/step - loss: 0.3040 - accuracy: 0.9152
Epoch 3/5
60000/60000 [==============================] - 4s 63us/step - loss: 0.2919 - accuracy: 0.9187
Epoch 4/5
60000/60000 [==============================] - 4s 64us/step - loss: 0.2812 - accuracy: 0.9218
Epoch 5/5
60000/60000 [==============================] - 4s 62us/step - loss: 0.2719 - accuracy: 0.9237
Out[66]: <keras.callbacks.callbacks.History at 0x1683cba4668>
```

Cross Entropy

$$-\sum_{i=1}^{n}\sum_{m=1}^{M} y_{im} * \ln(\hat{y}_{im})$$

- M is the total class of our labels
- n is the total numbers of training data
- $y_{im} = 1$ if sample i belongs to class m. Otherwise, $y_{im} = 0$
- $\hat{y}_{im}$ is the output probability that sample i belongs to class m

Cross Entropy

- 為何在分類問題要使用Cross Entropy Loss?
  - 以二元分類問題為例來解釋

- MSE Loss

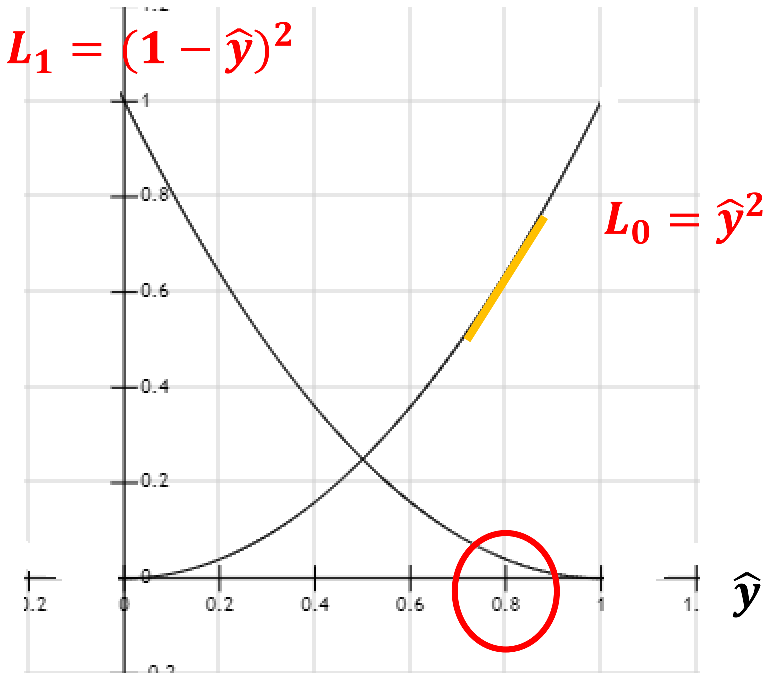
$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- Binary Cross Entropy Loss

$$-\sum_{i=1}^{n}[y_i * \ln(\hat{y}_i) + (1 - y_i) * \ln(1 - \hat{y}_i)]$$

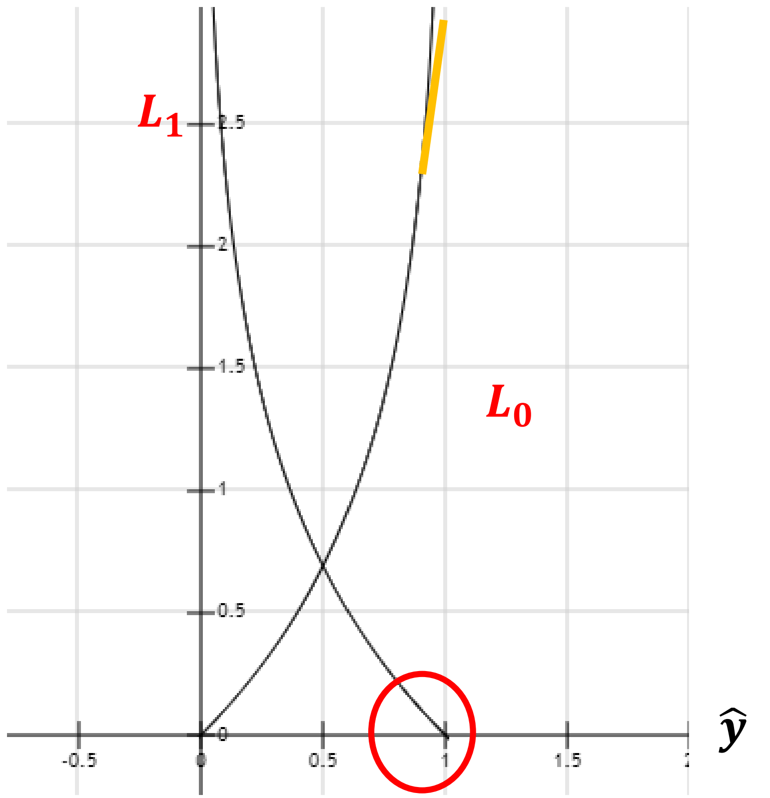
Cross Entropy

- MSE Loss
- $L = (y - \hat{y})^2$
  - $for\ y = 0, L_0 = \hat{y}^2$
  - $for\ y = 1, L_1 = (1 - \hat{y})^2$



- Binary Cross Entropy Loss
- $L = -y * \ln(\hat{y}) - (1 - y) * \ln(1 - \hat{y})$
  - $for\ y = 0, L_0 = -\ln(1 - \hat{y})$
  - $for\ y = 1, L_1 = -\ln(\hat{y})$

- 可以發現Cross Entropy在預測錯誤類別時給的Gradient較MSE大。
- 並且預測錯誤的越離譜，給予的懲罰成指數成長。這能使我們的模型在面對分類問題時，更好的去辨識類別。



```python
# Test

train_loss, train_acc = \
    network.evaluate(train_images_, train_labels_)
print('train_acc:', train_acc)

test_loss, test_acc = \
    network.evaluate(test_images_, test_labels_)
print('test_acc:', test_acc)
```

- .evaluate()函數回傳loss value 和 metrics values (此為accuracy)

print出loss value

```
In [3]: print('train_loss:', train_loss)
train_loss: 0.3326067911823591

In [4]: print('test_loss:', test_loss)
test_loss: 0.3167715199768543
```

print出預測準確率

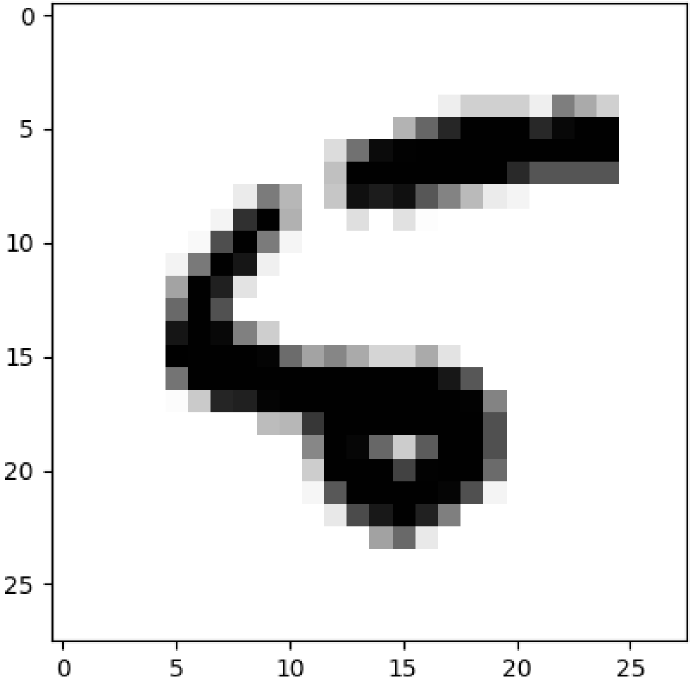
```
In [64]: print('train_acc:', train_acc)
train_acc: 0.9101166725158691

In [65]: print('test_acc:', test_acc)
test_acc: 0.9150999784469604
```

```
prediction_labels_ = network.predict(test_images_)
```

- prediction_labels_為一10000*10之array
- 列index ： 代表第幾張圖片
- 行index ： 代表該圖片為哪個數字
- 值 ： 表該圖片為行index數字之機率大小

	0	1	2	3	4	5	6	7	8	9
0	0.000302772	2.72351e-06	0.000140463	0.000724466	3.7201e-05	7.17846e-05	1.93222e-06	0.995531	7.55003e-05	0.00311227
1	0.0156435	0.000561828	0.896425	0.0145481	1.72796e-06	0.0126114	0.0534574	2.47404e-06	0.00674681	1.80917e-06
2	0.000252189	0.958659	0.0109619	0.00635125	0.00118656	0.0029894	0.00386011	0.00556357	0.00799581	0.00218005
3	0.996743	1.38839e-08	0.000255874	4.24796e-05	9.10588e-07	0.00188287	0.000666166	0.000251471	0.000107518	5.02274e-05
4	0.0040834	8.66717e-05	0.0145453	0.00102025	0.835824	0.00341786	0.0118257	0.0220998	0.0137318	0.0933656
5	2.26214e-05	0.983877	0.00249365	0.00349457	0.000201571	0.000460215	0.000182544	0.00409028	0.0043422	0.000835543
6	0.000157027	9.5832e-05	0.000187791	0.0032181	0.891031	0.0302201	0.000770723	0.00698703	0.0271017	0.0402307
7	0.000157233	0.00460278	0.00560672	0.0146375	0.136212	0.0876534	0.00783568	0.0133735	0.0719496	0.657972
8	0.0312687	0.000555947	0.189721	0.000136351	0.0550694	0.0306119	0.66966	0.000276212	0.0175523	0.00514877
9	0.000133228	1.40671e-06	2.17844e-05	5.75156e-05	0.0251209	0.000692835	8.46199e-05	0.0902726	0.00337519	0.88024
10	0.951081	1.289e-06	0.00280419	0.00221742	9.62963e-06	0.0387008	0.000270735	9.22378e-06	0.00490262	3.61688e-06
11	0.0353406	0.00234231	0.0785764	0.0144717	0.0152674	0.0301152	0.652974	0.000640833	0.168354	0.00191749
12	0.000218136	3.59033e-06	0.000219676	0.000387391	0.0602201	0.00136613	0.000236652	0.0379691	0.00182975	0.89755
13	0.982964	1.53531e-07	0.00027865	0.000141298	4.39752e-05	0.00947983	3.4453e-05	0.000191465	0.00557603	0.00129049

```python
digit = test_images[8]
plt.imshow(digit, cmap = plt.cm.binary)
plt.show()
```



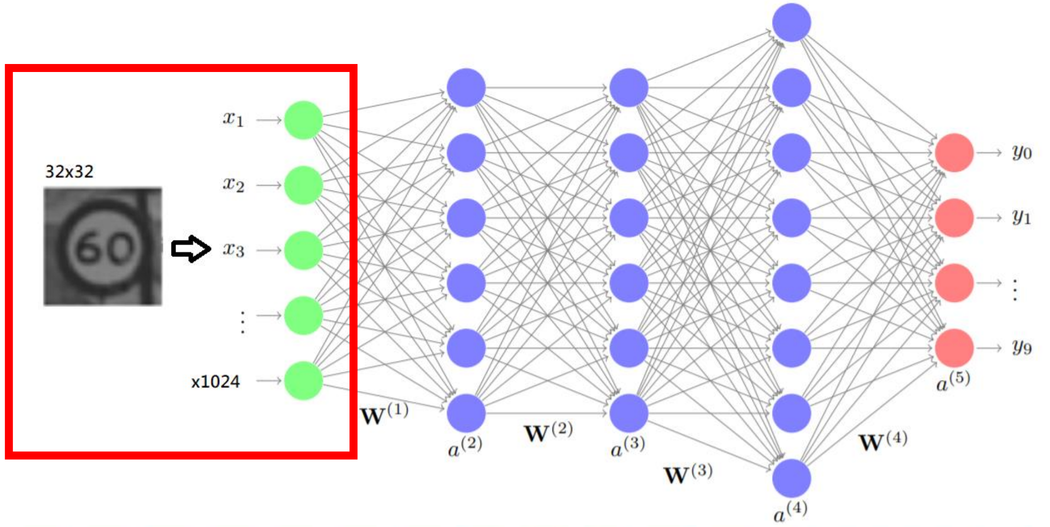

```
In [62]: test_labels[8]
Out[62]: 5

In [63]: prediction_labels[8]
Out[63]: 6
```

- 查看模型預測錯誤的圖像: 實際上這張圖片為數字5，但我們的模型預測成6。不過模型判斷錯誤情有可原，因為這張手寫圖本來就模稜兩可。

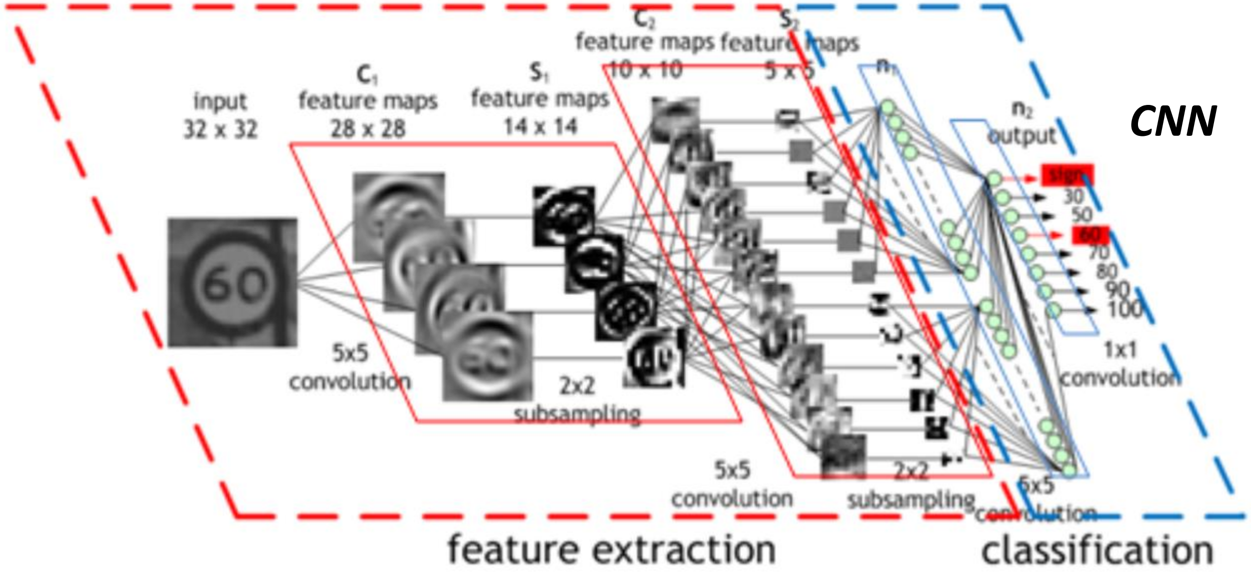
4

CNN

Difference between CNN & DNN



DNN



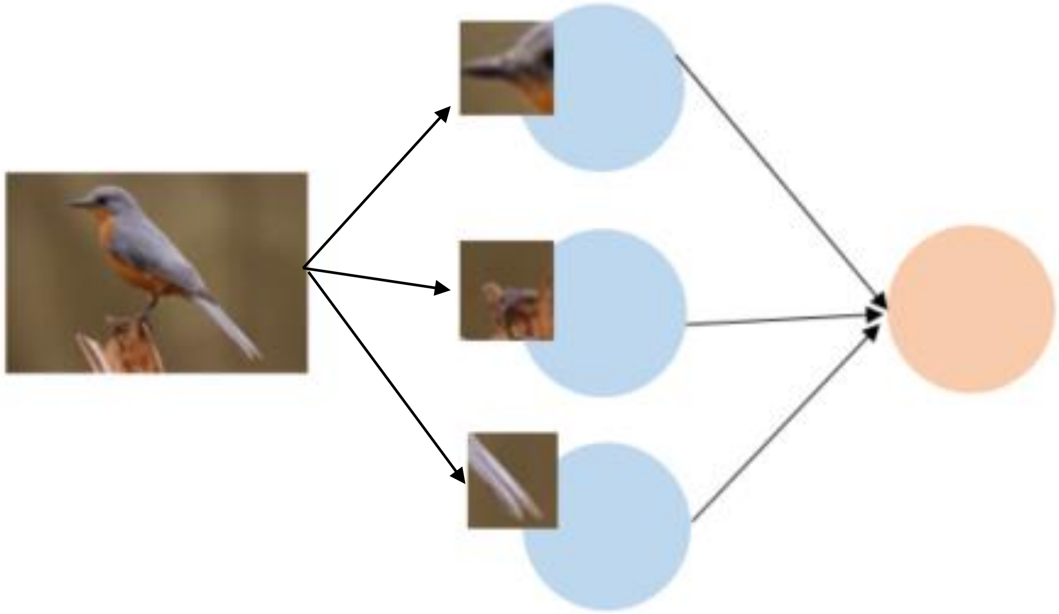
CNN

- 舉圖像辨識來說，一般的DNN會將原本32*32的灰階圖片先轉成一條vector，每一個1*1的pixel分別作為一個factor輸入神經網路，這將使這個圖像的輸入**失去了位置資訊**

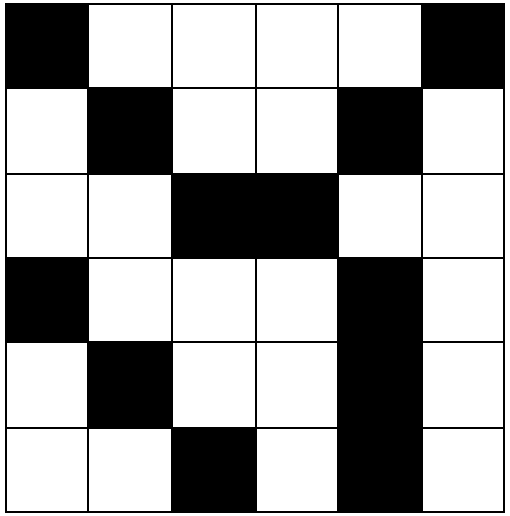
- CNN會將完整個32*32灰階圖片矩陣輸入，**保留了圖像的位置資訊**。並透過中間的Convolve Layer & pool Layer逐步擷取出圖像的特徵，再把這些細微的圖像特徵當作輸入，丟進原本的DNN中

Intuition1 of CNN

- 前面提到CNN可以擷取每個細小的圖像特徵。每個neuron要去觀察這些特徵有無出現時，實際上不需要看整張圖片，而**只需要看圖片的一小部分**，就可以決定這件事情

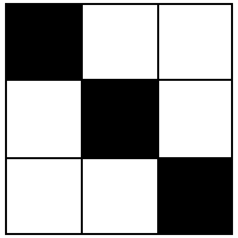


CNN 擷取特徵的方法



1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

- 假設今天我們要擷取下圖的這個3*3的斜條紋特徵



- 我們的filter會是下方這個3*3的矩陣

1	-1	-1
-1	1	-1
-1	-1	1

CNN 擷取特徵的方法

Our filter

1	-1	-1
-1	1	-1
-1	-1	1

⇨

Our image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

step1

將3*3的filter對6*6的image做rolling window的elementwise product的和

CNN 擷取特徵的方法

Our filter

1	-1	-1
-1	1	-1
-1	-1	1

Our image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

3

step1

將3*3的filter對6*6的image做rolling window的elementwise product的和

CNN 擷取特徵的方法

Our filter

1	-1	-1
-1	1	-1
-1	-1	1

Our image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

3 -1

step1

將3*3的filter對6*6的image做rolling window的elementwise product的和

CNN 擷取特徵的方法

Our filter

1	-1	-1
-1	1	-1
-1	-1	1

Our image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

3 -1 -3

step1

將3*3的filter對6*6的image做rolling window的elementwise product的和

CNN 擷取特徵的方法

Our filter

1	-1	-1
-1	1	-1
-1	-1	1

3x3

Our image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6x6

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

4x4

step1

將3*3的filter對6*6的image做rolling window的elementwise product的和

step2

可以發現我們rolling完整個matrix後，出現的是一個4*4的matrix

CNN 擷取特徵的方法

Our filter

1	-1	-1
-1	1	-1
-1	-1	1

3x3

Our image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6x6

Feature Map

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

4x4

step1

將3*3的filter對6*6的image做rolling window的elementwise product的和

step2

可以發現我們rolling完整個matrix後，出現的是一個4*4的matrix

step3

這個4*4的matrix我們稱為Feature Map，裡面值越大，代表**我們的filter要偵測的特徵出現在那個區域**

Intuition2 of CNN

- 同一個特徵，出現在一張圖裡多次，我們不需要用新的filter去偵測有無這個特徵

Our filter

1	-1	-1
-1	1	-1
-1	-1	1

3x3

Our image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6x6

Feature Map

$\boxed{3}$	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
$\boxed{3}$	-2	-2	-1

4x4

Intuition2 of CNN

➢ 同一個特徵，出現在一張圖裡多次，我們不需要用新的filter去偵測有無這個特徵（**權值共享**）

Our filter

1	-1	-1
-1	1	-1
-1	-1	1

3x3

Our image

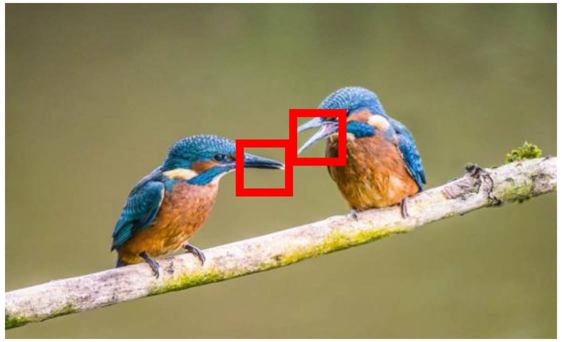
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6x6

Feature Map

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

4x4



➢ 事實上，filter裡的每個element就是之前DNN的每個權重。因此權值共享的這個特性，讓CNN有比DNN更少的參數，模型更為簡易。

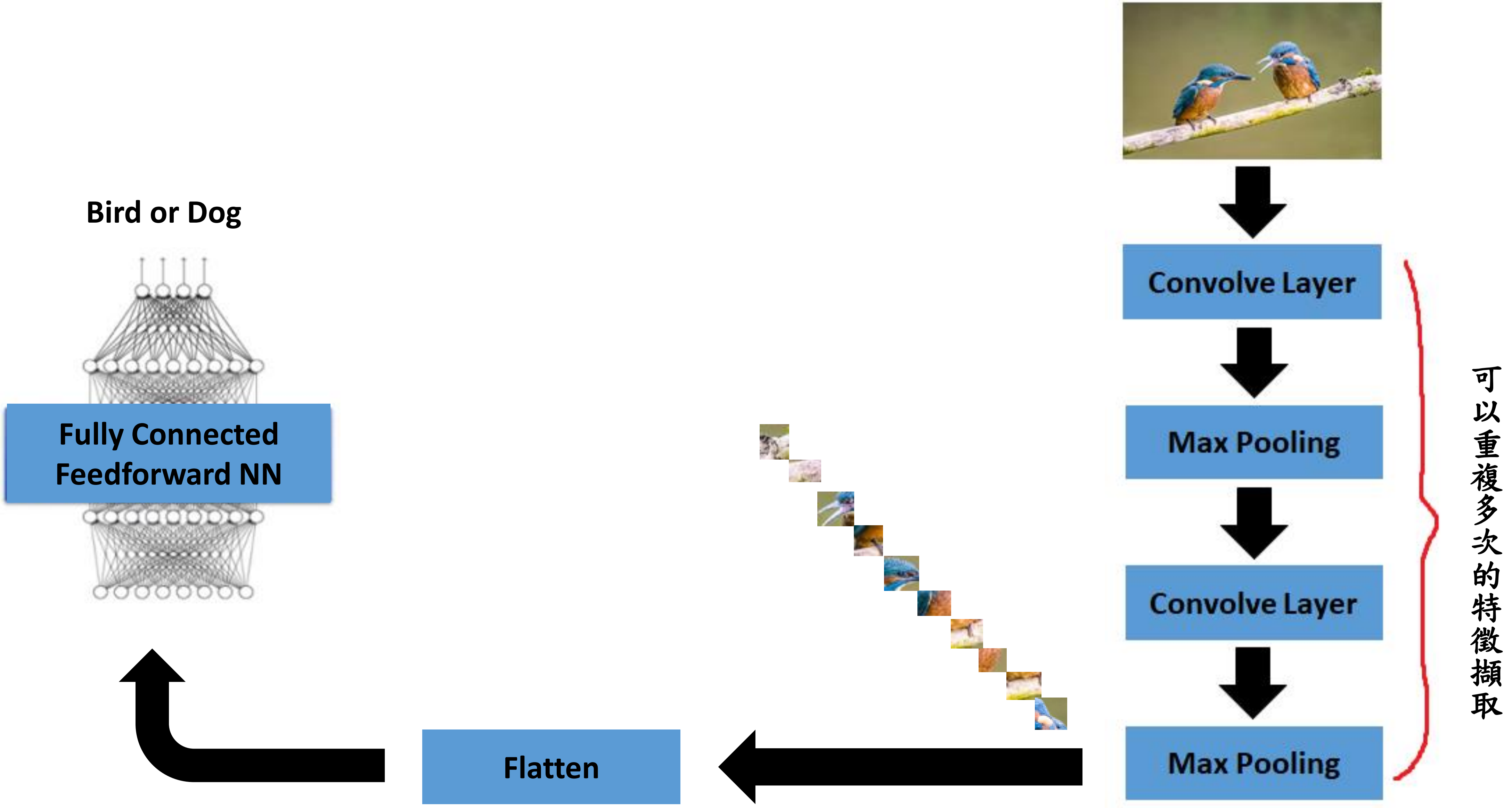
- ➢ 假設CNN我們設計有100個 filters，則這層layer總共會需要訓練$3 \ast 3 \ast 100 + 100 = 1000$ 個參數
- ➢ 假設DNN我們設計有100個 neurons，則這層layer總共會需要訓練$6 \ast 6 \ast 100 + 100 = 3700$ 個參數

Intuition3 of CNN

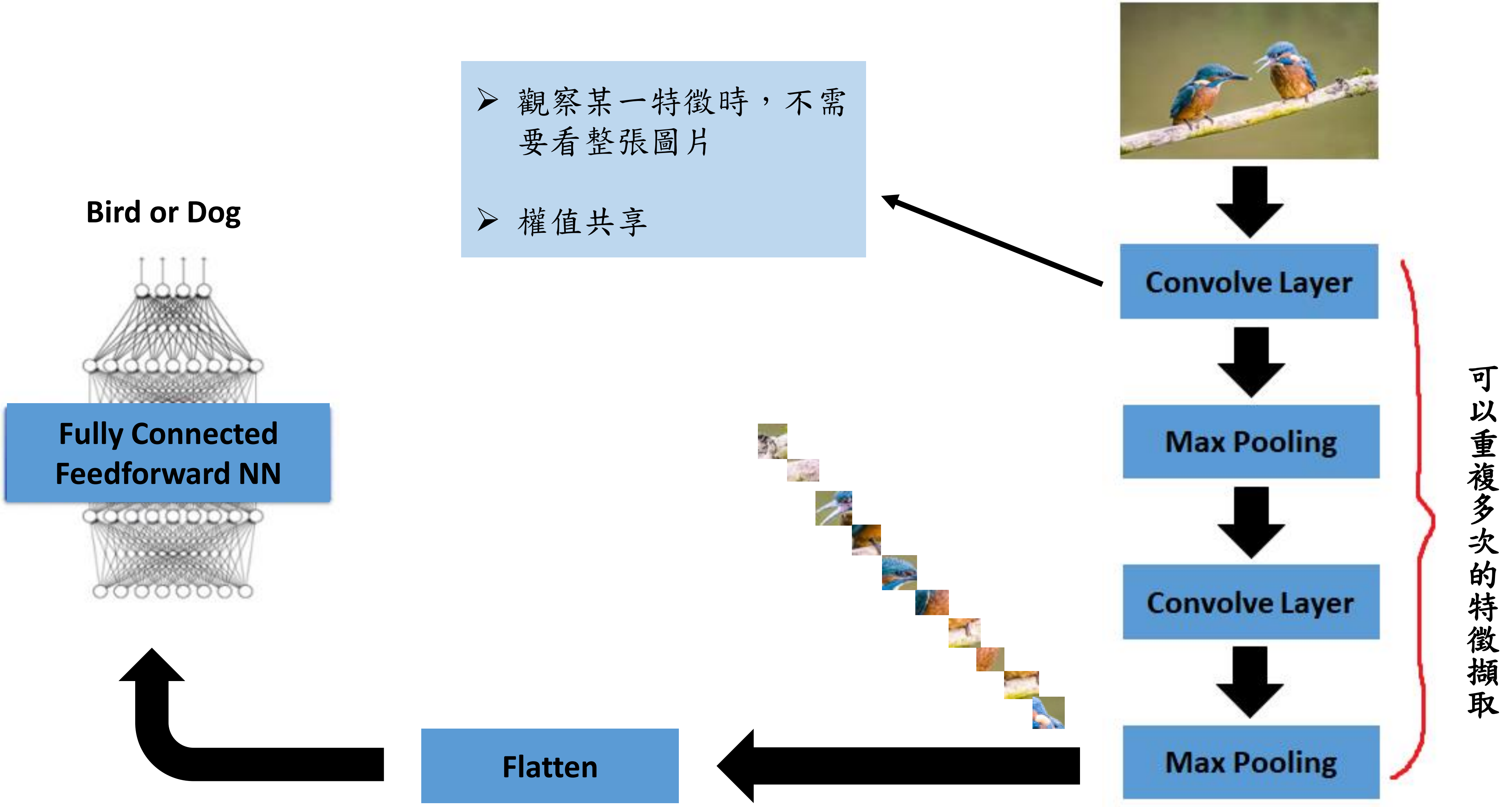
- 對原始的圖片做subsampling，並不會影響圖像的辨識
  - Example: 對這張圖抽掉奇數行、偶數列的pixel，圖縮小、畫值下降，但不影響圖像辨識

- 同樣地，做了subsampling 後，我們的圖縮小了。這也就表示我們可以用較少的參數去讓神經網路去學習這張圖像。

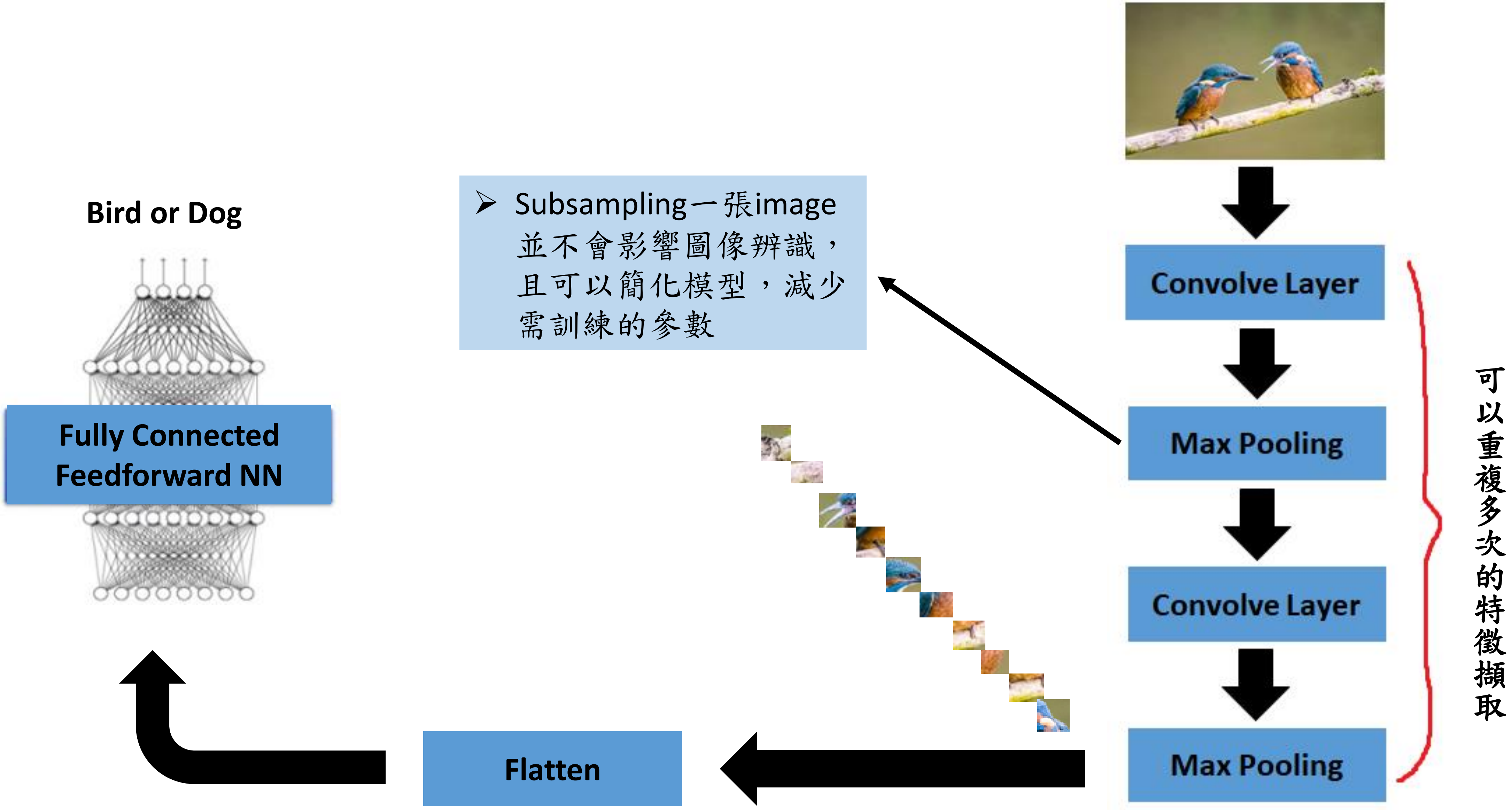
The Structure of CNN



The Structure of CNN



The Structure of CNN



Convolve Layer

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6x6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1 Matrix

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2 Matrix

⋮

- 這些Filter其實就是hidden layers的權重（待會解釋），是必須學出來的，不是研究者自行給定的
- Filter 1捕捉的是一個3*3的斜條紋特徵
- Filter 2捕捉的是一個3*3的直條紋特徵
- 研究者在每個Convolve Layer可以去決定要設定幾個Filter

Convolve Layer

Filter 1

1	-1	-1
-1	1	-1
-1	-1	1

Our image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

3

Convolve Layer

Filter 1

1	-1	-1
-1	1	-1
-1	-1	1

Our image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

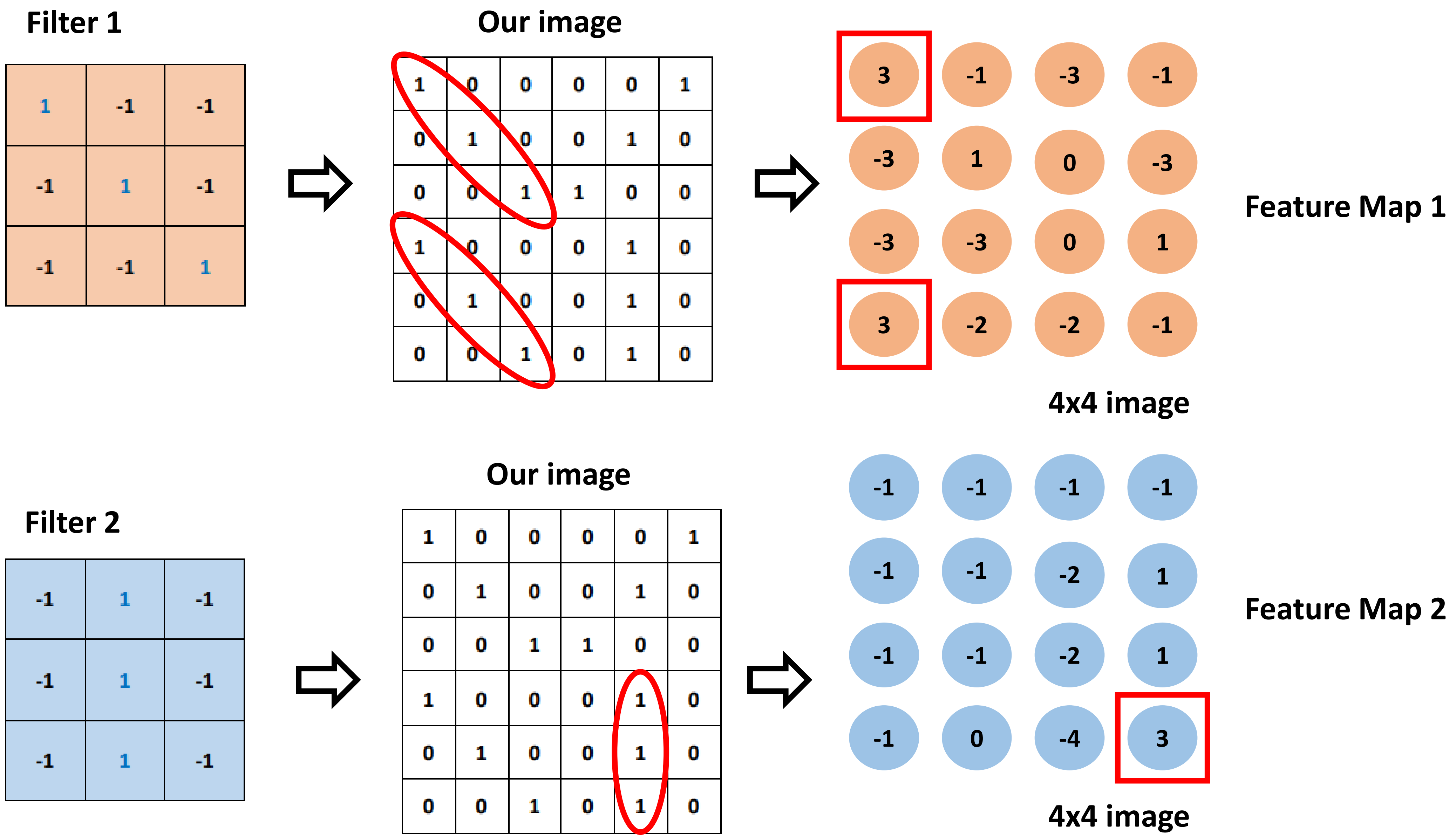
Feature Map 1

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

4x4 image

- 註:設定多少個Filter，就會得到多少個Feature Map

Convolve Layer



Convolve Layer & Fully Connected are Almost the same

Our image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

1	-1	-1
-1	1	-1
-1	-1	1

-1	1	-1
-1	1	-1
-1	1	-1

Convolution

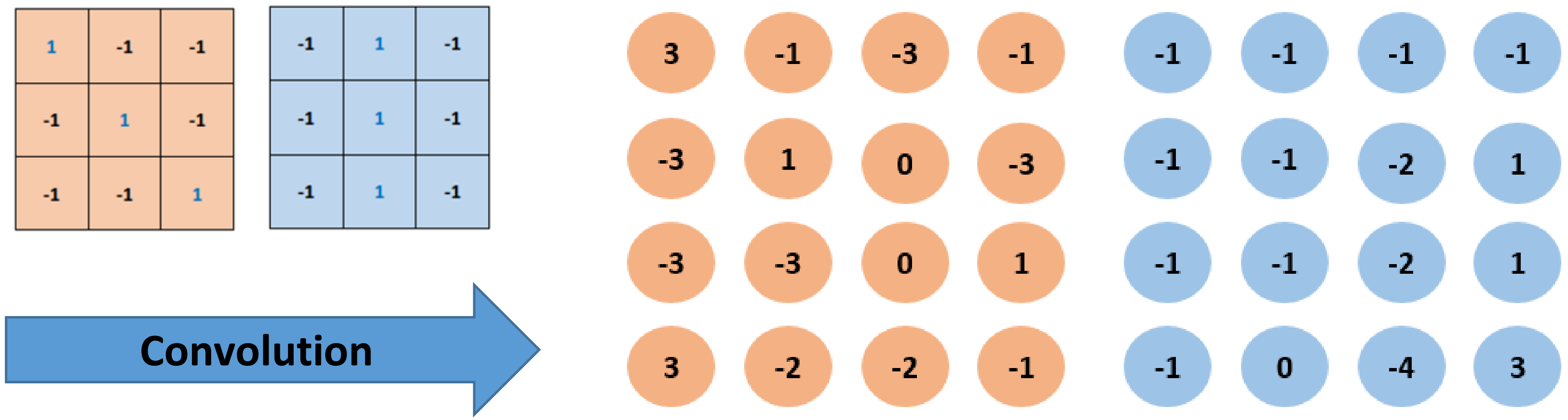
3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

-1	-1	-1	-1
-1	-1	-2	1
-1	-1	-2	1
-1	0	-4	3

Convolve Layer & Fully Connected are Almost the same

Our image

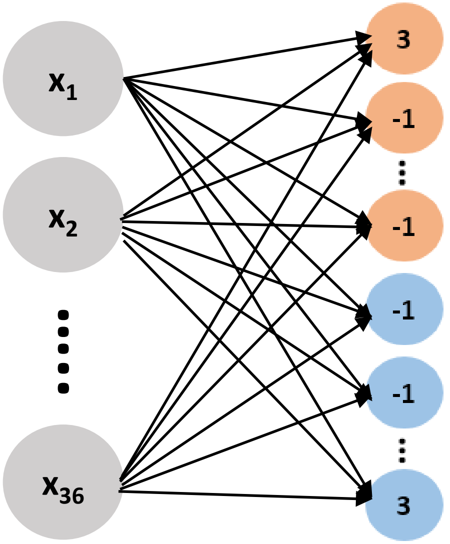
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0



Fully Connected Neural Network

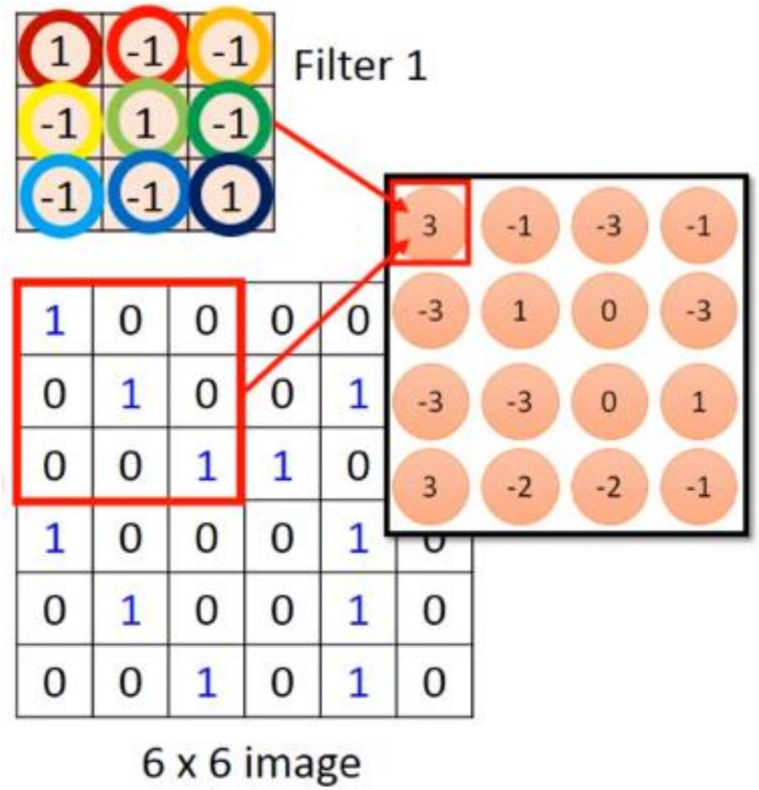
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6x6 image



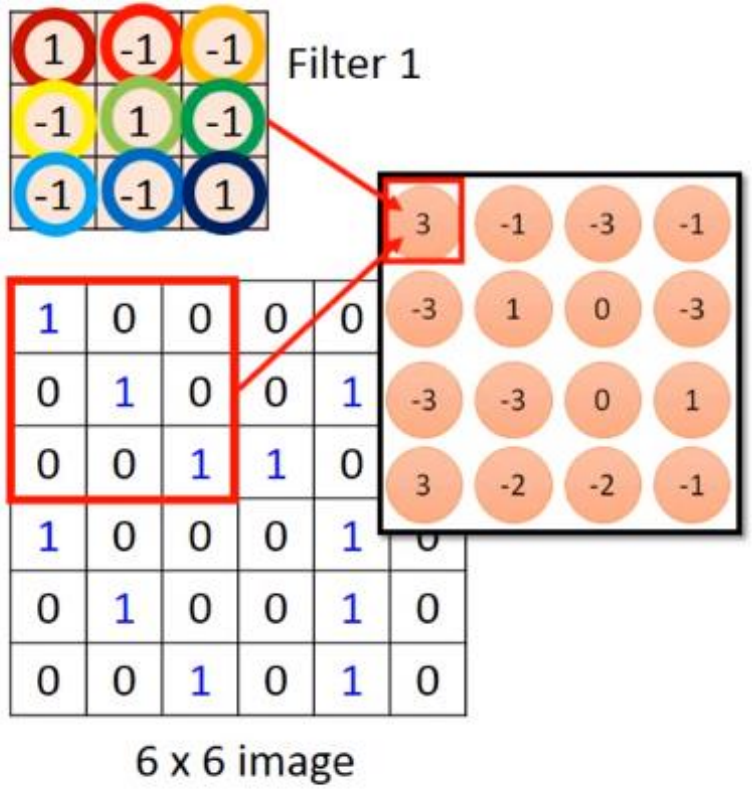
- Convolution Layer其實就像是 Fully Connected拿掉一些weight 的結果

Convolve Layer & Fully Connected are Almost the same



Convolve Layer & Fully Connected are Almost the same

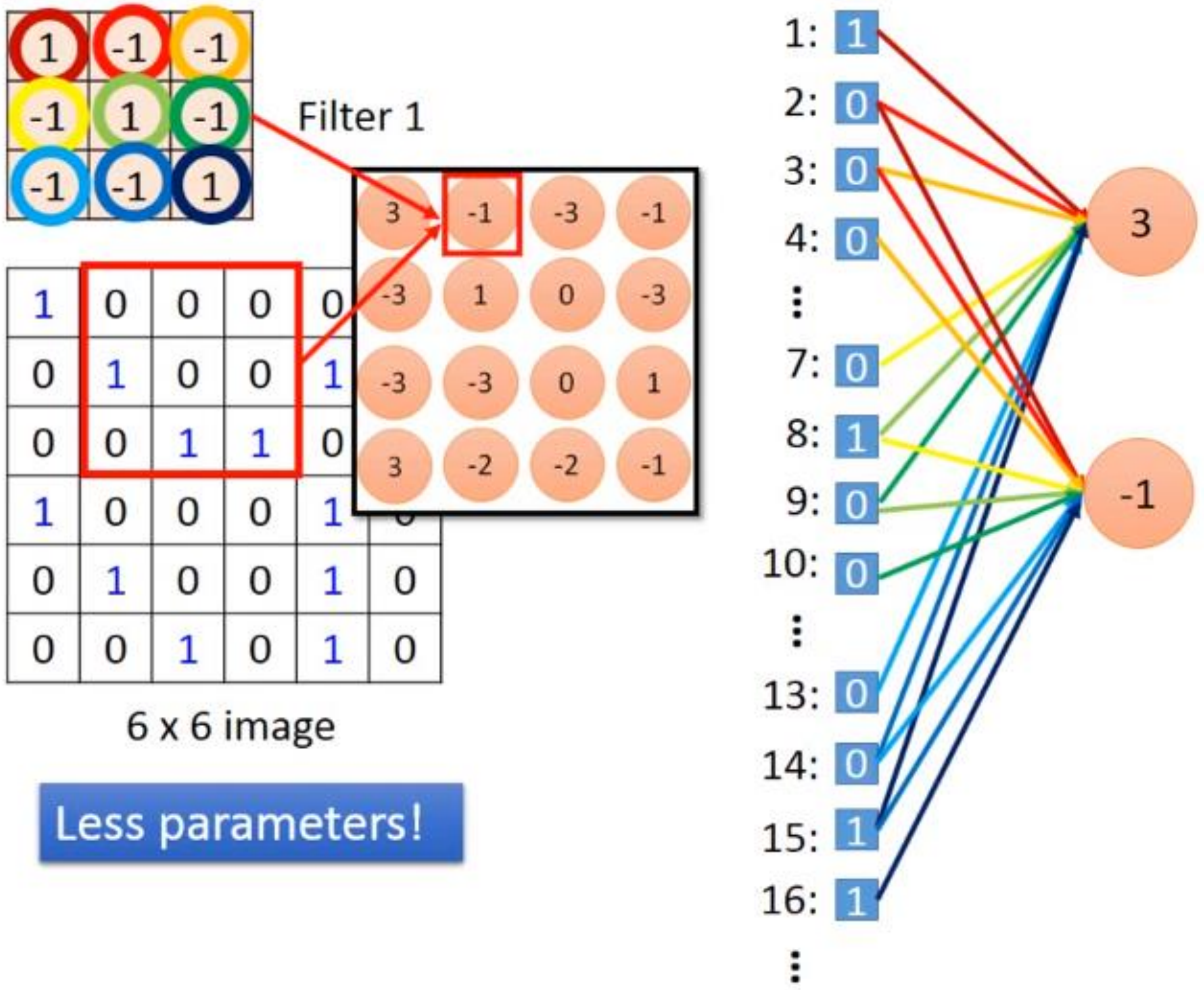
- 可以發現Feature Map 1的第一個element代表著這層Layer的一個Neuron。這個Neuron只連結了9個weights，也就是其他weights都為0



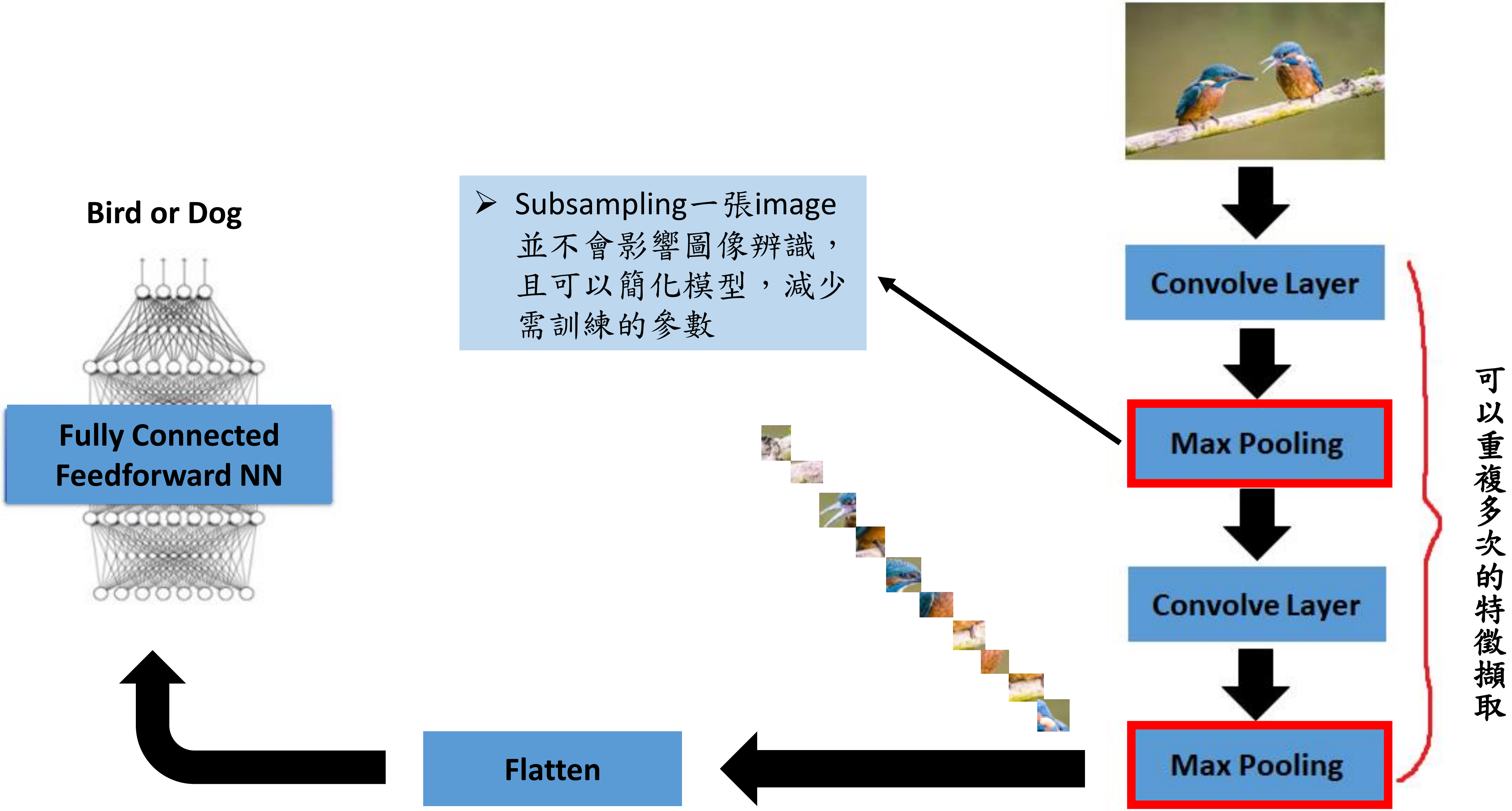
- 不是Fully Connected，因為只連接了9個inputs
- 從這裡也可以得知，當要觀察這張圖像有沒有某種特徵時，**我們不用看整張圖，我們只需要9個inputs即可**

Convolve Layer & Fully Connected are Almost the same

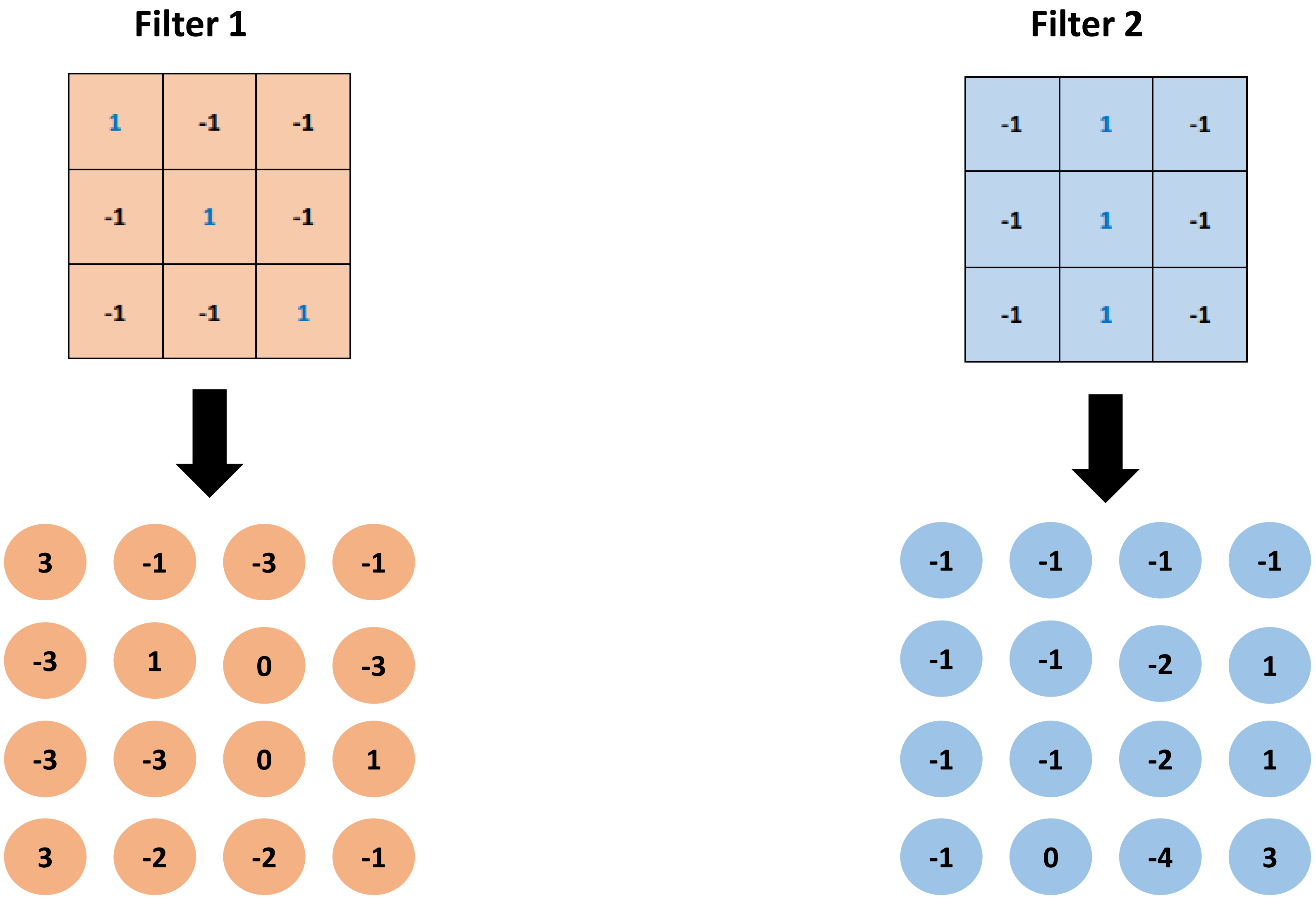
- 可以發現Feature Map 1的第一個element代表著這層Layer的一個Neuron。這個Neuron只連結了9個weights，也就是其他weights都為0
- 可以發現這裡3的Neuron和-1的Neuron**共享了權重**。有別於原本Fully Connected每個Neuron都有自己的一組weight



The Structure of CNN



Pool Layer (Subsampling)



Pool Layer (Subsampling)

Filter 1

1	-1	-1
-1	1	-1
-1	-1	1

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

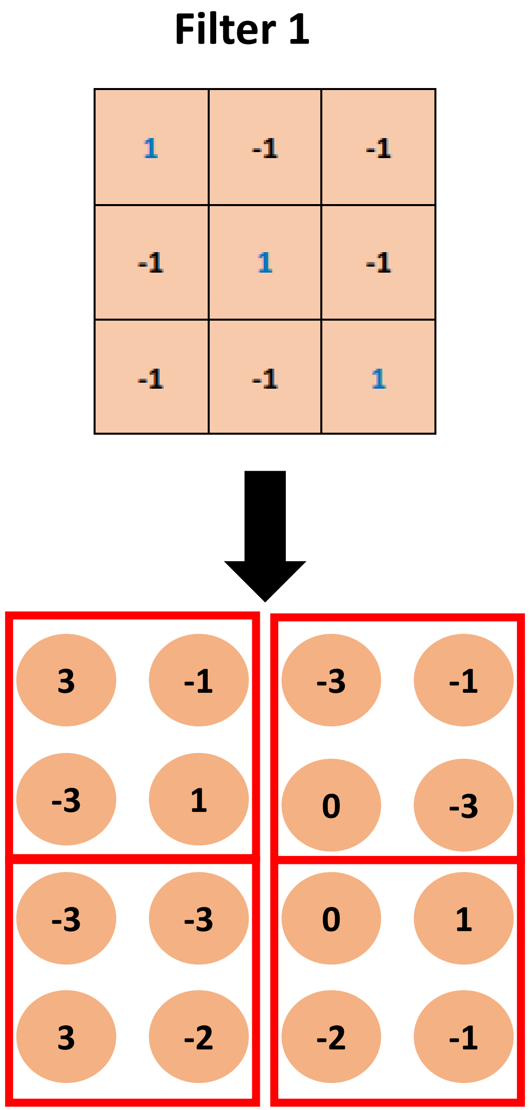
- 將這兩個Feature Maps切成2*2pixel為一組

Filter 2

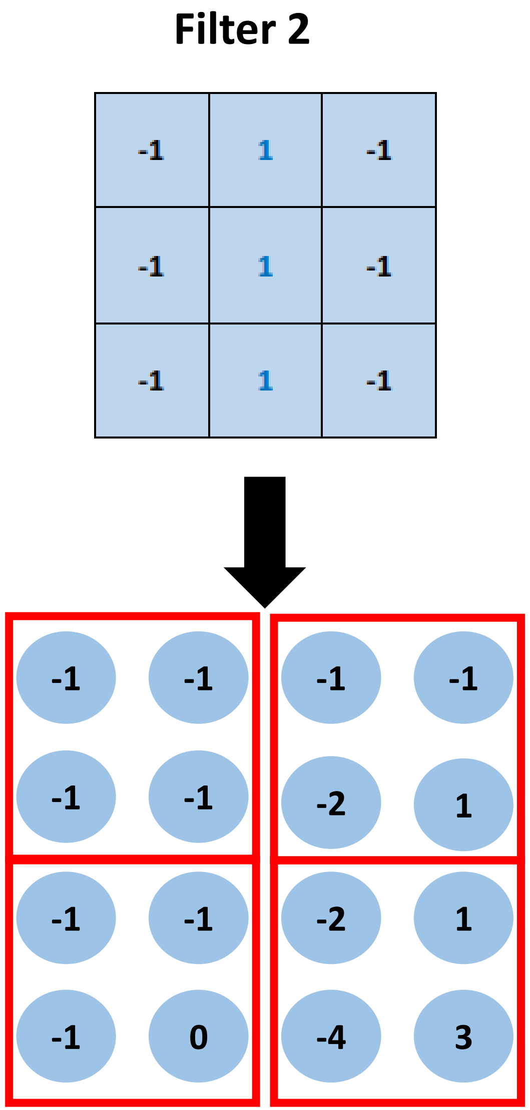
-1	1	-1
-1	1	-1
-1	1	-1

-1	-1	-1	-1
-1	-1	-2	1
-1	-1	-2	1
-1	0	-4	3

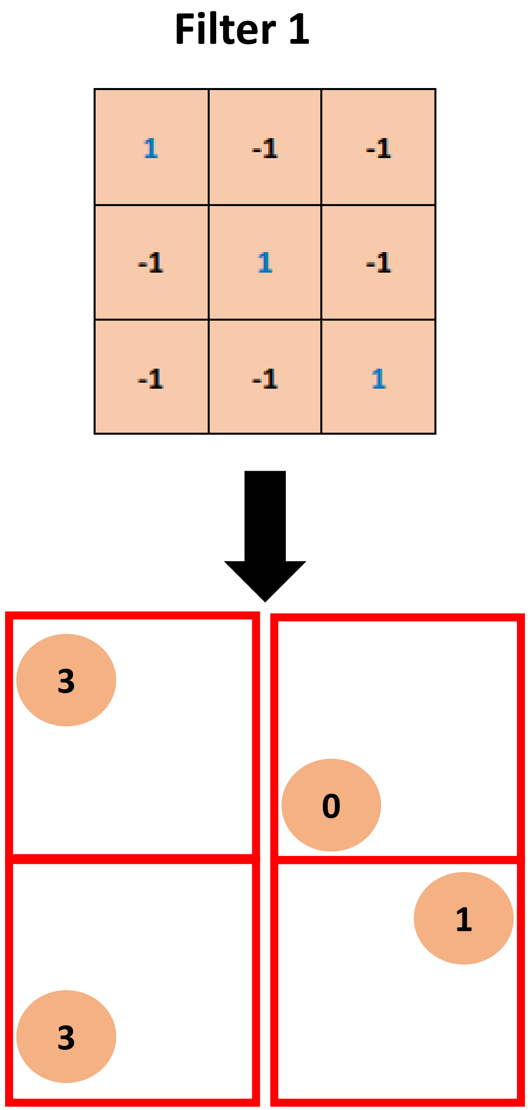
Pool Layer (Subsampling)



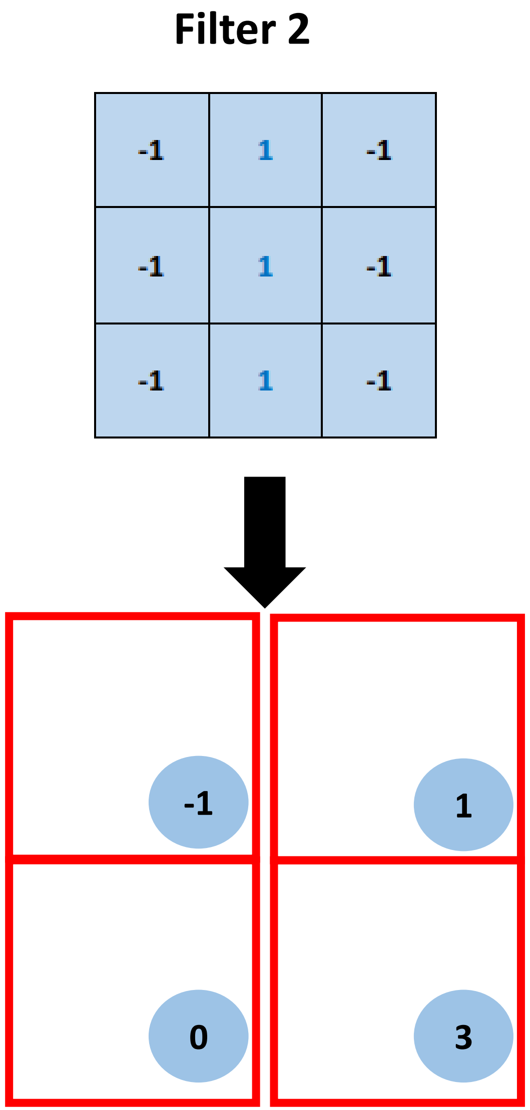
- ➢ 開始做subsampling，只取每個2*2一組的pixel中最大的值



Pool Layer (Subsampling)



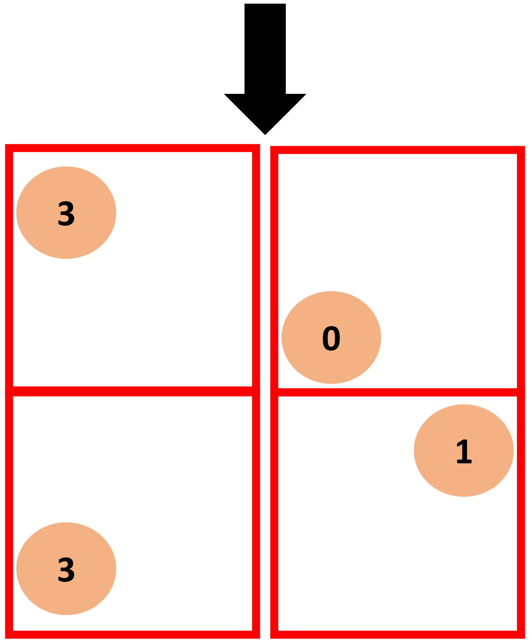
➢ 開始做subsampling，只取每個2*2一組的pixel中最大的值



Pool Layer (Subsampling)

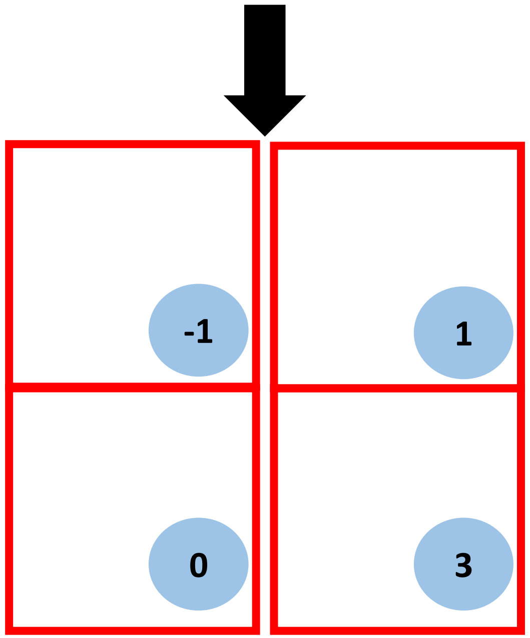
Filter 1

1	-1	-1
-1	1	-1
-1	-1	1



Filter 2

-1	1	-1
-1	1	-1
-1	1	-1

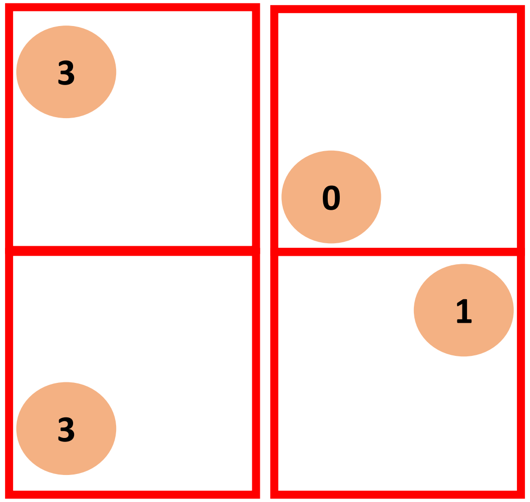


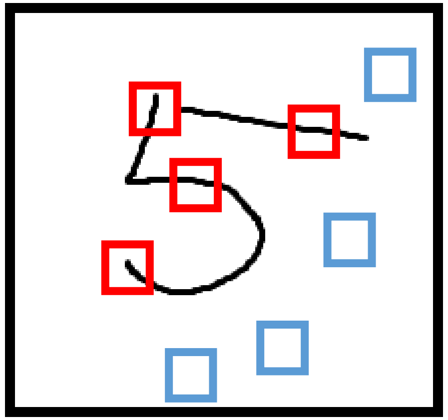
- 為何我們可以這樣做?只取maximum?

Pool Layer (Subsampling)

Filter 1

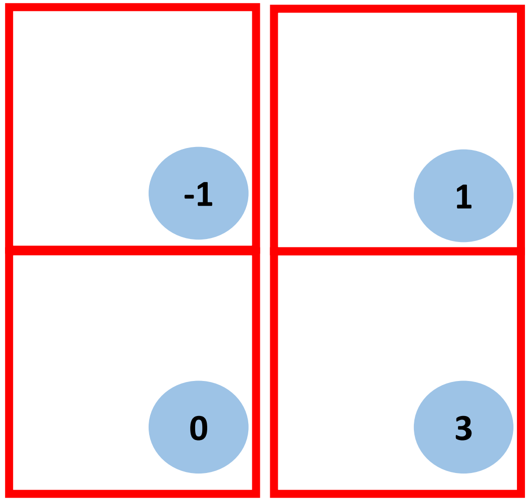
1	-1	-1
-1	1	-1
-1	-1	1





Filter 2

-1	1	-1
-1	1	-1
-1	1	-1



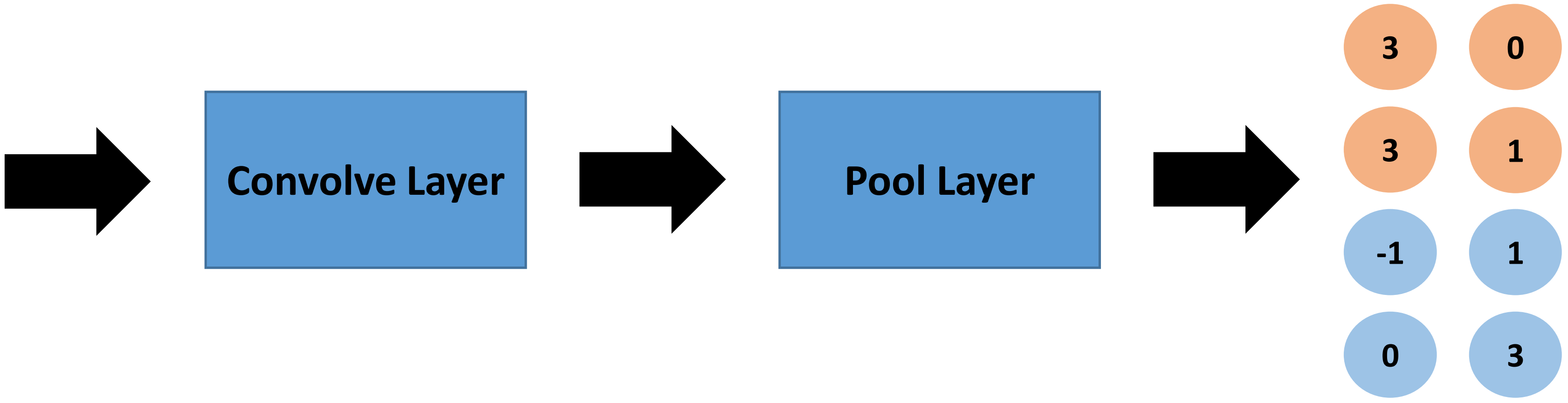
- 明顯的紅色框框為我們在意的區塊，藍色的框框我們不在意

Pool Layer (Subsampling)

Our Original image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6x6 image



Two different 2x2 image

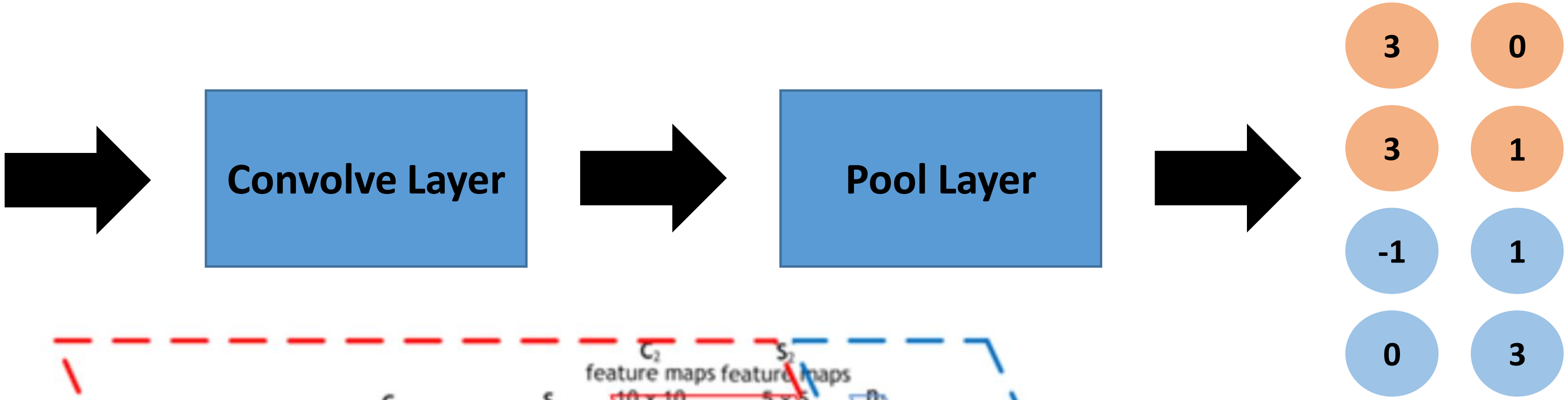
- Since we use 2 filter in this layer, we have 2 new but smaller image

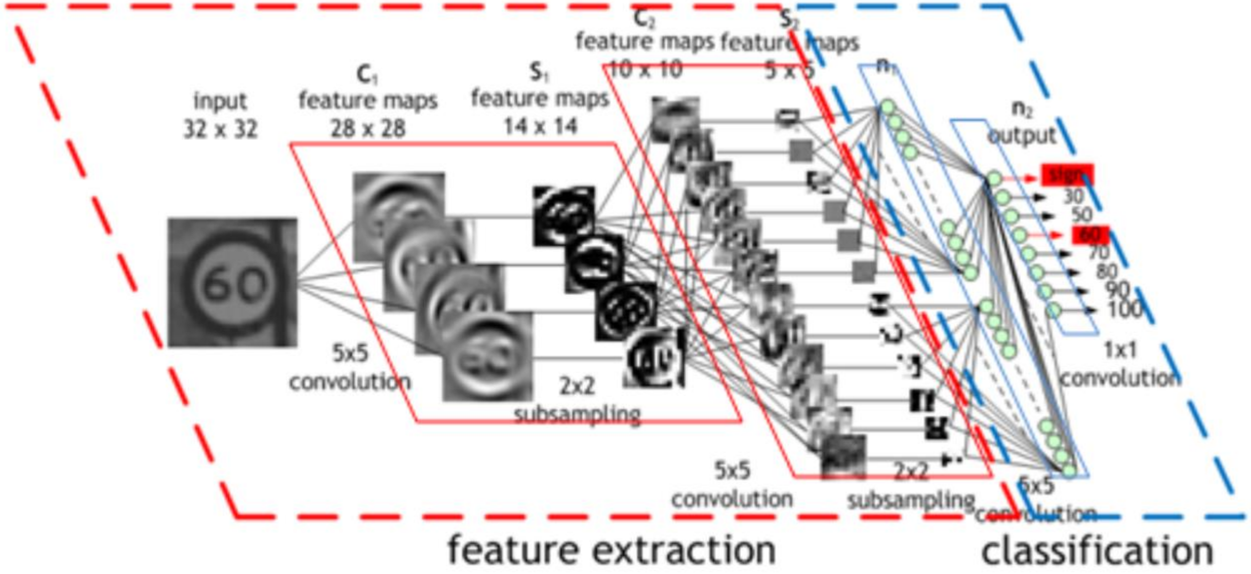
Pool Layer (Subsampling)

Our Original image

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6x6 image

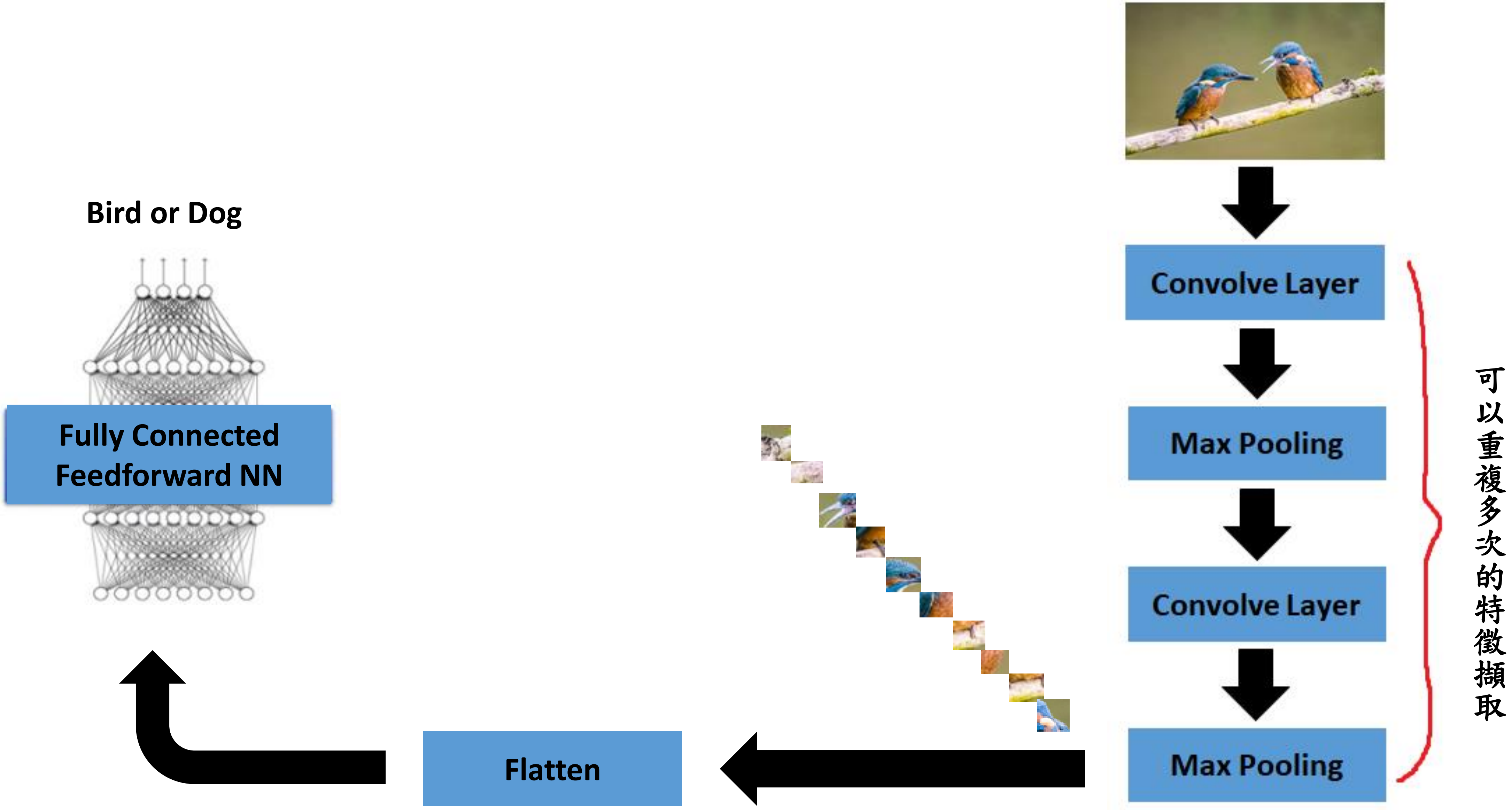




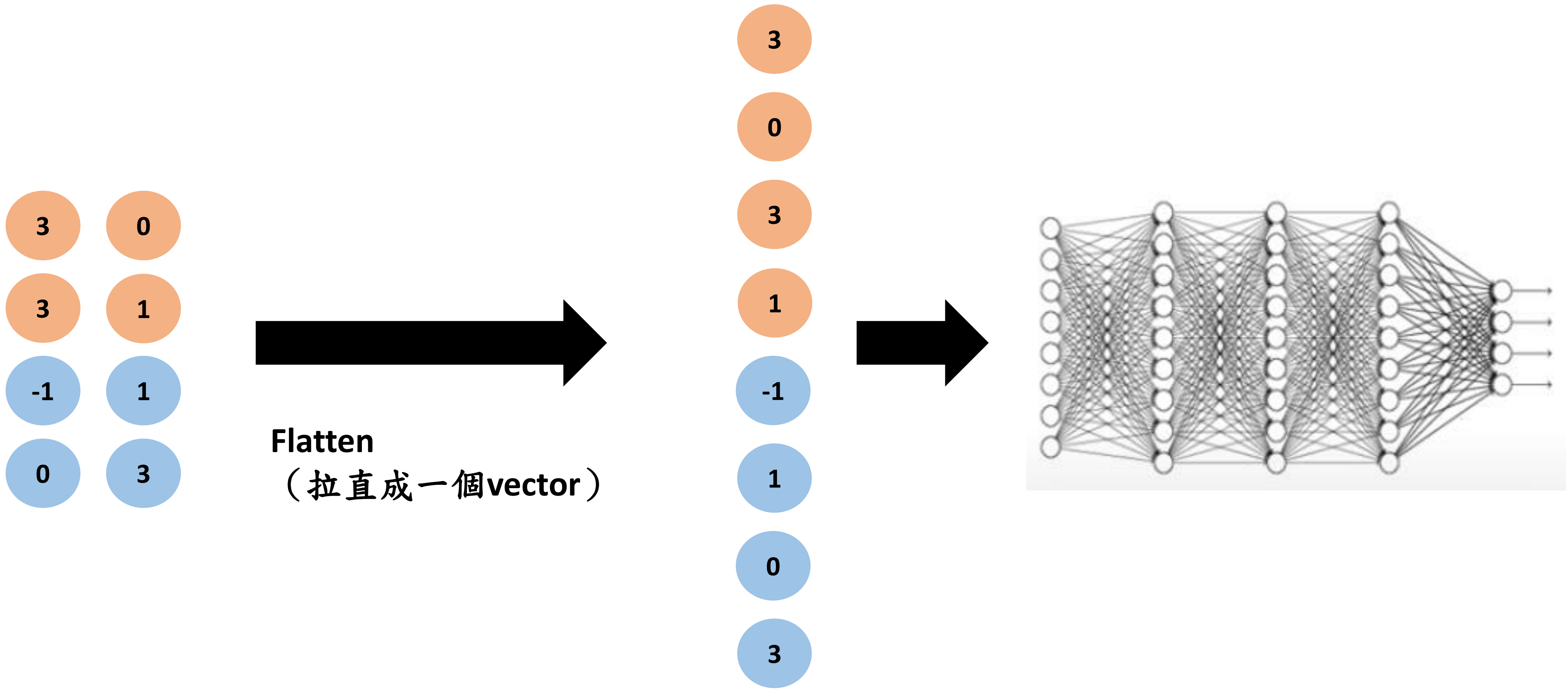
Two different 2x2 image

- Since we use 2 filter in this layer, we have 2 new but smaller image

The Structure of CNN



Flatten



5

CNN實作

手寫辨識（二）：以CNN為例

```
# Load mnist

from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) \
    = mnist.load_data()

train_images_ = train_images.reshape((60000, 28, 28, 1))
train_images_ = train_images_.astype('float32') / 255

test_images_ = test_images.reshape((10000, 28, 28, 1))
test_images_ = test_images_.astype('float32') / 255

from keras.utils import to_categorical

train_labels_ = to_categorical(train_labels)
test_labels_ = to_categorical(test_labels)
```

- **Reshape**

因為 CNN 將一張圖片作了特徵擷取，擷取成多張圖片，使這邊的輸入變成一個3維的 cubic。我們 reshape 一個新的維度去存取這個資料

```python
# Model

from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation = 'relu',
                        input_shape = (28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation = 'relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation = 'relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation = 'relu'))
model.add(layers.Dense(10, activation = 'softmax'))

model.summary()
```

```
In [45]: model.summary()
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_10 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_4 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_17 (Dense)             (None, 64)                36928
_________________________________________________________________
dense_18 (Dense)             (None, 10)                650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
```

```python
model.compile(optimizer = 'SGD',
              loss = 'categorical_crossentropy',
              metrics = ['accuracy'])

# Train

model.fit(train_images_, train_labels_,
          epochs = 5, batch_size = 64)
```

```
Epoch 1/5
60000/60000 [==============================] - 45s 754us/step - loss: 0.8008 - accuracy: 0.7595
Epoch 2/5
60000/60000 [==============================] - 48s 792us/step - loss: 0.2140 - accuracy: 0.9344
Epoch 3/5
60000/60000 [==============================] - 49s 815us/step - loss: 0.1421 - accuracy: 0.9571
Epoch 4/5
60000/60000 [==============================] - 44s 734us/step - loss: 0.1108 - accuracy:
0.966160000 [==============>...............] - ETA: 22s - loss: 0.1155 - accuracy: 0.9641
Epoch 5/5
60000/60000 [==============================] - 45s 753us/step - loss: 0.0939 - accuracy: 0.9705
Out[46]: <keras.callbacks.callbacks.History at 0x17d00da04e0>
```

```
In [48]: train_loss, train_acc = \
    ...:     model.evaluate(train_images_, train_labels_)
    ...: print('train_acc:', train_acc)
    ...: print('train_loss:', train_loss)
    ...:
    ...: test_loss, test_acc = \
    ...:     model.evaluate(test_images_, test_labels_)
    ...: print('test_acc:', test_acc)
    ...: print('test_loss:', test_loss)
    ...:
    ...: prediction_labels_ = model.predict(test_images_)
    ...: print(prediction_labels_)
60000/60000 [==============================] - 14s 241us/step
train_acc: 0.9754166603088379
train_loss: 0.07986300249285996
10000/10000 [==============================] - 2s 243us/step
test_acc: 0.9768999814987183
test_loss: 0.0713429974405095
```

```python
prediction_labels_ = model.predict(test_images_)
print(prediction_labels_)
```

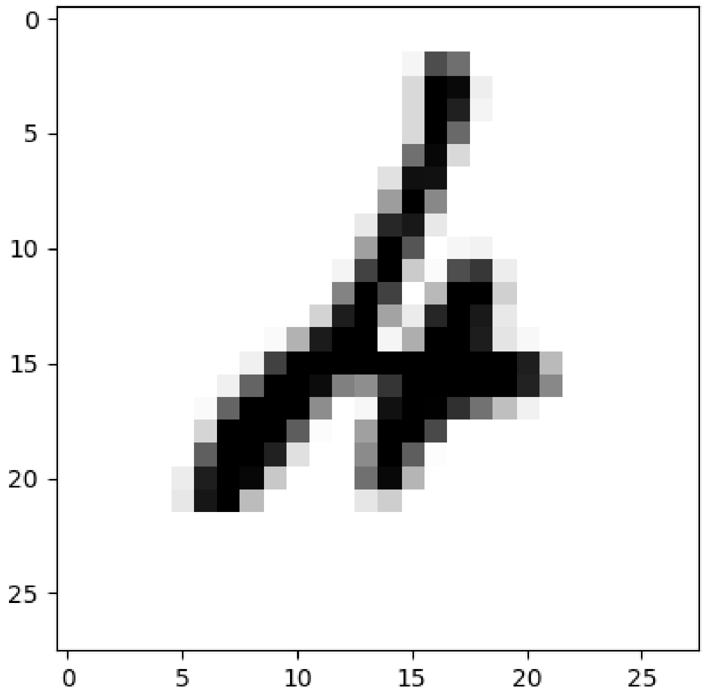
prediction_labels_ - NumPy array

	0	1	2	3	4	5	6	7	8	9
0	3.98347e-07	2.39098e-07	8.82927e-05	8.34233e-06	2.59351e-10	3.88126e-07	1.52368e-11	0.99989	3.52551e-07	1.19916e-05
1	0.000457783	0.00141693	0.997965	0.000118726	1.47063e-09	2.59835e-07	3.7427e-05	1.22674e-07	3.65812e-06	3.03868e-11
2	1.54006e-05	0.998305	0.000161518	5.23328e-05	0.000571158	3.6872e-06	3.67397e-05	0.000673382	0.000164858	1.56651e-05
3	0.99966	1.32143e-08	0.000122678	1.64148e-07	1.3685e-06	3.34461e-05	0.000122152	6.50997e-06	4.51145e-06	4.9159e-05
4	2.20174e-06	1.62384e-06	3.14915e-06	5.94696e-07	0.998494	1.05229e-07	4.97165e-06	1.46398e-05	3.18975e-06	0.00147552
5	7.11319e-07	0.998648	5.67097e-06	7.32383e-06	0.000127258	4.69029e-08	3.61701e-07	0.00116504	3.52738e-05	9.94527e-06
6	2.61395e-08	9.23076e-06	1.19052e-06	4.84983e-05	0.966106	0.00116555	4.09004e-06	0.00229129	0.00951138	0.0208627
7	1.41059e-06	0.00015442	0.00091158	0.00851292	0.00765205	0.00810194	5.81907e-07	0.00301966	0.00156175	0.970084
8	8.3535e-05	1.33264e-08	0.000356382	1.93962e-05	4.67693e-05	0.970318	0.0159721	5.07912e-06	0.0129097	0.000288762
9	1.8375e-06	1.11773e-07	3.95874e-05	0.000111064	0.000119689	9.14849e-05	1.25028e-07	0.0776257	0.00417967	0.917831
10	0.999134	1.16382e-07	0.000203231	4.3141e-07	1.06634e-06	0.000272205	0.000326756	1.20575e-06	2.27719e-05	3.87684e-05
11	0.000167322	2.69889e-08	4.6866e-06	1.88588e-07	6.08099e-07	0.000146328	0.997359	1.00986e-08	0.00232209	1.13795e-07
12	4.80919e-07	1.4498e-08	2.56503e-06	5.32362e-05	0.00058812	0.000128737	5.98134e-08	0.0022717	0.000111174	0.996844
13	0.999706	3.75238e-09	2.8017e-05	2.95162e-08	5.81228e-07	1.49476e-05	8.28073e-06	1.56893e-05	2.95746e-05	0.000197383
14	4.82769e-07	0.999587	3.54895e-05	0.000120722	8.09122e-06	3.92386e-06	2.37004e-06	3.77662e-05	0.000193004	1.15884e-05
15	1.0214e-06	2.12502e-08	6.23592e-05	0.00398774	3.21506e-08	0.995583	3.47299e-06	1.5802e-07	0.000352627	9.69322e-06
16	1.77657e-06	1.51161e-08	3.36053e-06	3.62013e-06	0.00145375	9.81736e-07	8.29354e-08	0.000403341	2.67995e-05	0.998106
17	1.75845e-06	9.74229e-08	0.000109827	2.62102e-05	1.27718e-11	4.35581e-07	1.94498e-12	0.999856	3.06508e-08	5.52277e-06
18	1.12931e-05	4.2977e-06	0.00562527	0.925307	9.1532e-07	0.0039657	1.64692e-05	4.79276e-06	0.065041	2.33458e-05
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Format Resize ☑ Background color

```python
digit = test_images[247]
plt.imshow(digit, cmap = plt.cm.binary)
plt.show()
```

查看讓模型預測錯誤的圖片




```
In [54]: test_labels[247]
Out[54]: 4

In [55]: prediction_labels[247]
Out[55]: 2
```

實際上這張圖片為數字4
但我們的模型預測成2

Reference

- 清大 楊睿中教授《機器學習與經濟計量》：Neural Networks

- Efron, B. and T. Hastie (2016), Computer Age Statistical Inference: Algorithms, Evidence and Data Science, Cambridge.

- 台大 李宏毅教授 ML Lecture 10: Convolutional Neural Network