銘　傳　大　學
資訊傳播工程學系碩士班

碩士學位論文計畫書

以深度網路建構自然語言處理模型

研究生：邱盛宇
指導教授：李明哲　博士

中華民國一〇五年十二月

目錄

表目錄

圖目錄

第一章　研究背景與動機

過往，機器學習與深度學習的架構受限於硬體的運算速度與演算法的發展，遲緩了近十年。演算法的發展直至 1998 年 Yann LeCun [1]等人發表了 LeNet [2]才大幅改變了演算法的發展。近年來，硬體設備的大幅邁進，電腦主機與 GPU [3] 顯示晶片等價錢日漸低廉，促使了整個人工智慧的發展與運算加速。層次可以到更深並且運算更加強大。人類離人工智慧的路程得以趨近。

近年來知名的成功案例是 Google [4] 公司旗下的 DeepMind [5] 開發團隊，在 2015 年 10 月利用深度網路開發的 AlphaGO [6] 系統成功打敗歐洲棋王，並於 2016 年 3 月成功打敗韓國最強旗手。開創了第一個人工智慧的先鋒。日本微軟在 2015 年 8 月開放了他們所開發了人工智能 AI「りんな」 [7]，並整合 Line [8] 線上社群平台為發佈空間，引發日本區的一陣熱潮與話題。深度網路的應用在近年來可見將漸漸深根在各行各業，幫助並加速整體經濟的成長。

由上述而知，構築在這麼多的成功案例，深度網路的發展已經日漸成型。近年的遞歸神經網路 Recurrent Neural Network(RNN) [9] 結合詞向量的運算在語意分析與應用上更顯卓越，由於其變化性與可塑性高，可應用在許多不同的實際問題上。將此架構結合 LSTM [10] 演算法幫助了整體的語意分析更加的穩定與準確。

英語系國家與部分亞太地區已漸漸地利用 Word2Vec [11] 架構演算能代表詞意的詞向量，並結合 RNN 結構建構出近人工智慧的架構，並開始推展各項相關可行性應用。因此，本研究以此為動機，將希望建構相關的自然語言處理模型以解決可套用的相關應用。

第二章　研究問題與目的

語意分析研究在過往以來一直是人類在人工智慧中挑戰的主要課題。若能做出好的語意分析，便能促進人機界面的發展，更能帶動機器人在各種應答上的躍進。

就語言架構本身，對於不同的閱讀者去理解同一句話就已有不同的意義。加之中文結構相對複雜，詞義的分析與詞距將影響整體的語言意義跟語言結構。並從而影響到使用者所反應出的情緒。因此，本研究就以此為研究問題，以大量的文本訓練為基礎，願能實作出具備情緒分析與解讀能力之演算法與機器。

受限於機器自身不具備語意中的情緒理解，基於此基礎上將會影響與阻礙語句應答。如此一來容易使得機器本身無法正確理解與應答，產生溝通障礙。有鑑於此，本研究希望基於此問題提供解決方法。藉由大量的資料分析出正確的情緒成為本研究目的。

第三章　文獻探討

第一節　　深度學習

深度學習是在機器學習中獨立出來的一個重要的分支。機器學習中目前常見的分類為兩類，一類是監督式學習(purely supervised learning algorithms)，另一類是非監督式學習(unsupervised and semi-supervised learning algorithms)，這是參照國外的深度學習的教學網站所做出的分類[12]。其中，監督式學習類有個可依循的學習基礎，依據羅吉斯回歸、多層類神經網路與深度卷積神經網路這個架構與方向學習。而非監督式網路傳統上則是依照研究領域所需項目分別研究與學習。

此時當深度學習的架構與概念呈現出來後，此舉打破傳統的二分觀念。由於深度學習是結合監督式學習與非監督式學習，因此深度學習的框架已大幅改善了以往監督式學習的學習效率與正確率。另外，有另一群研究者將此學習歸類於半監督式學習。目前此概念之架構底下衍伸出來已有數種知名的架構與應用。如深度全連結類神經網路、卷積式神經網路、深度信念網路與遞歸神經網路等等都已被應用在電腦視覺、語音識別、自然語言處理，音訊分析與生物基因演算等等領域並具有一定的效果 [13] [14] [15]。以下將針對深度神經網路、卷積神經網路與遞歸神經網路做更深入的探討。

第二節　　深度神經網路(Deep Neural Network)

第一個深度神經網路是由 Olexiy Hryhorovych (Alexey Grigoryevich) Ivakhnenko 等人於 1965 年所發表 [16] [17]，其概念是模擬人類神經元的運作並加以擴充，為最早期深度神經網路中的架構，如下圖所示。

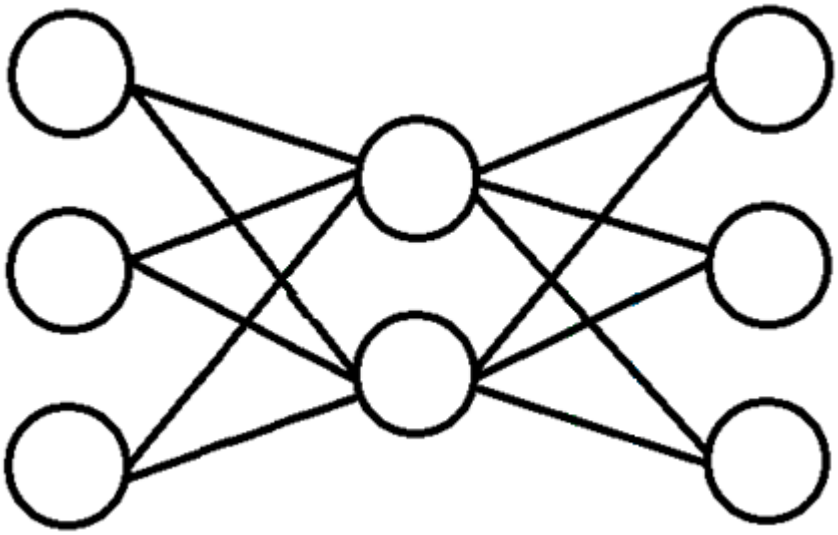


圖 三-I 深度神經網路示意圖

深度神經網路為一監督式網路，每顆神經元可接受一至多顆輸入，在每顆節點與節點間均有權重 W，每顆點上擁有 bias 權重。每顆節點將前一層的輸入乘以權重 W 並加上 bias 權重後，經過一個非線性化處理，將資料往下一層傳遞。不管層如何擴充都是以相同的做法計算，直到將結果算出來輸出至 output。

要讓深度神經網路能學習就必須將權重更新。要更新權重需要使用到梯度下降法(Gradient Descent)。如果將深度神經網路比喻成一張圖，則梯度下降法就是在計算斜率並朝正確的方向尋找圖形的最低點。整體的

深度神經網路倚賴著錯誤函式取得計算與實際值的誤差，使用梯度下降的概念，計算出函式的微分(即計算斜率) ，將錯誤下降至最小。而層跟層之間是擁有連結的，每層的關係將藉由數學計算取得相關的變數後進行倒傳遞，將更新的資訊網前一層資料傳遞。

訓練深度神經網路時需提供輸入與其對應的標籤。標籤將被視為該輸入的「標準答案」。由於深度神經網路的初始是亂數產生權重，因此需提供輸入與對應的標籤經不斷的訓練，以更新整體網路的權重，達成最佳化。

為了提升整體深度神經網路的準確率，已有實驗證實，擴大與擴深中間的層次與加入更多的訓練資料可達成上述結果。但過度的擴充架構將會使得原始架構中的梯度下降到某些特定層次後消失，無法繼續更新往後的權重；甚至因為訓練資料的不同使整體模型形狀特殊，而易落在區域最佳解 [18]。

為了解決梯度下降落在區域最佳解的問題，權重的初始化由原始的隨機產生改由先用非監督式學習法中的相關演算法先行計算後拿來使用 [19]。然而，此架構無法滿足廣泛且不同性質的資料，尤其在圖像資料、影像訊號與音訊、自然語言等連續性資料。因此近期慢慢地延展出了其他的演算法。

第三節　　卷積神經網路 (Convolutional Neural Network)

卷積神經網路的主要應用是由 Yann LeCun [1]等人於 1989 年為了解決圖像在深度神經網路的架構上因資料量太大而學習不易而提出的解決方法。作法是將圖像先經過圖學中的卷積(Convolutional)概念先進行卷積再接著使用圖學中的子採樣(Sub-Sampling)作法將圖片縮小至一定大小後再丟回深度神經網路進行運算。如此一來將利用較為不同的方式學習圖像中的特徵，並且能縮小進入深度神經網路的輸入，以解決區域最佳化之問題。其架構如下圖所示。

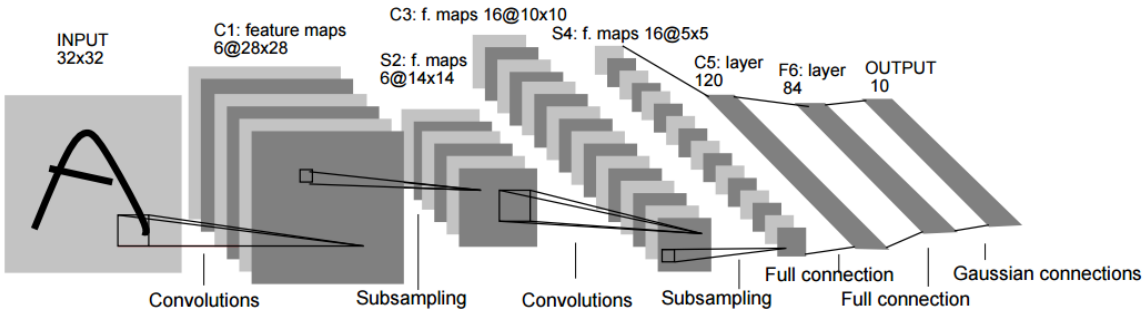


圖　三-II LeNet5 的架構圖

其中，最具知名的架構為修正後的第五版架構，被命名為 LeNet-5 [20]。此架構成功地應用在手寫文字辨識與機器文字辨識上。其輸入的訓練為手寫的圖像資料經由整體網路的不斷訓練後，可分析許多人手寫的數字，甚至連有些難以判別的數字都能被判別出來。

其中，深度網路中的卷積操作不同於圖學中的傳統卷積操作，採用

的是沒有擴充邊的作法。傳統圖學中的卷積操作是將圖擴充邊緣為零，從第一個像素開始卷積。如下圖所示。而卷積神經網路由於應用上的成功，並讓錯誤率降至極低(約 0.8%)，因此後期越來越多人將此網路架構做不同的變化與拓展應用到其他領域上。

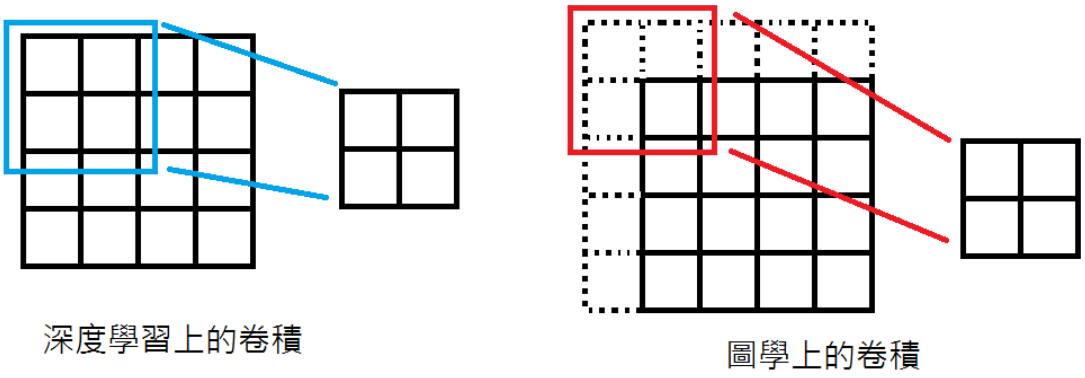


圖 三-III 卷積比較圖

第四節 遞歸神經網路(Recurrent Neural Network)

在深度網路中，資料的訓練與判別是獨立運作的。因此此技術相對不適合應用在音訊或自然語言這類的資料輸入。為了解決這類的序列性資料的訓練，就由以往的深度神經網路中做變化，有了新的網路架構為遞歸神經網路(Recurrent Neural Network) [21]。

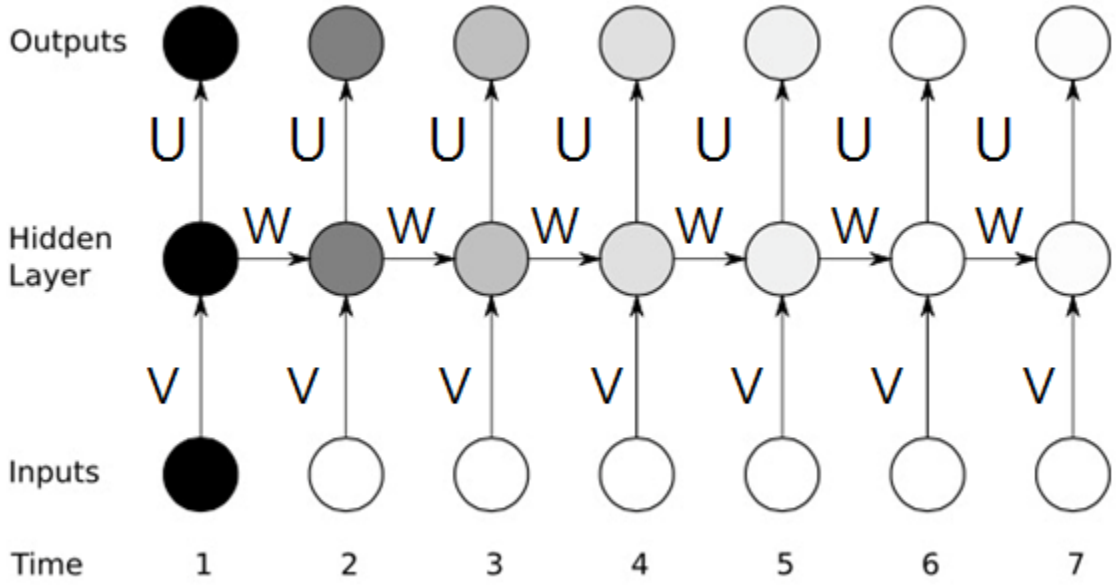


圖 三-IV 遞歸網路神經架構

遞歸神經網路的架構如上，其特殊點是在以往的深度神經網路中，新增了前後資料的隱藏層連結，讓隱藏層的資料不再如同以往只能接受當下輸入的資料，同時也能接收前一筆輸入的資料，同時在不同的層與層之間，使用了權重共享的架構。遞歸神經網路更利於有續性資料的分析。同時期應用也可以越加廣泛 [22]。

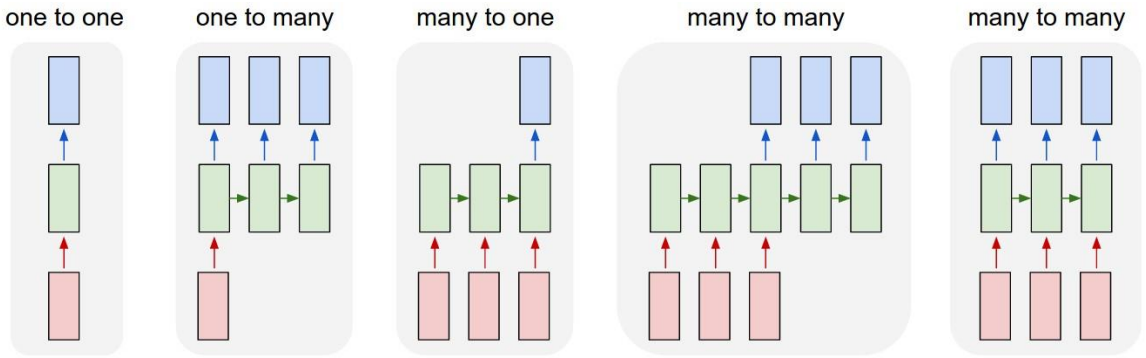


圖 三-V 遞歸神經網路的應用概念圖[21]

如此一來，應用在網路上的資訊可以因不同的輸入輸資料來加以修改整體的網路架構，如上圖所示。可以如同以往的一對一，訓練資料。

也可以一對多、多對一，甚至更能拓展成多對多與變相型的多對多。就以目前可知的例子即有英文字母的預測、文章產生器、維基百科格式建置演算法等等的各式應用。

在這個變化多端的演算法中，本研究也將希望藉此演算法應用在情緒分析上帶給此架構另一種不同的應用方法。

第四章　研究方法

第一節　　回歸(Regression)

一、線性回歸(Linear Regression)

是一種實數輸出的線性二元分類方式，利用線去將資料做分類。如下圖所示。

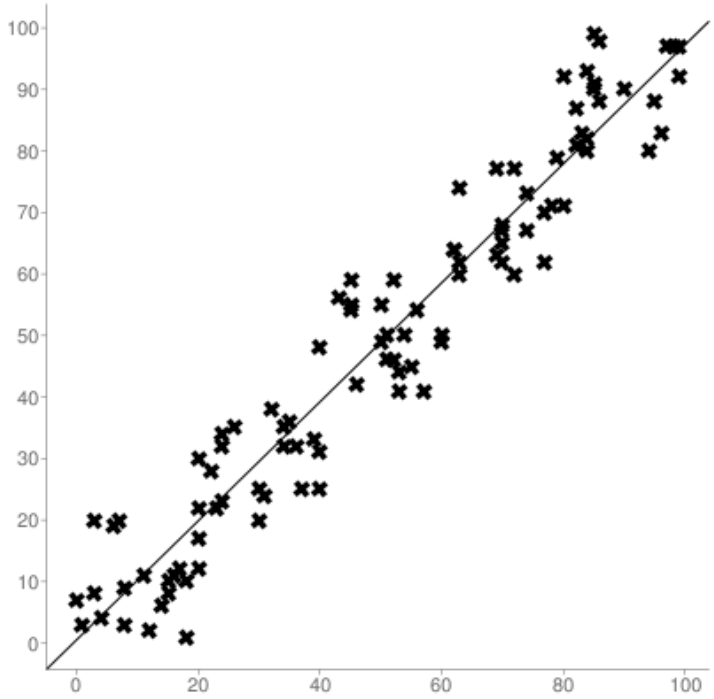


圖 四-I 圖形化線性回歸

二、非線性回歸(Logistic Regression)

屬多變量分析的其一分類模型，輸入資料後輸出預估值，而多數實作上會將輸出調節成 0/1 的二元輸出，其公式與變化如下。

$$logit(p_i) = ln\left(\frac{p_i}{1-p_i}\right) = \alpha + \beta_1 x_{1,i} + \cdots + \beta_k x_{k,i}, i = 1, \cdots, n$$

$$p_i = Pr(Y_i = 1|X) = \frac{1}{1 + e^{-(\alpha+\beta_1 x_{1,i}+\cdots+\beta_k x_{k,i})}}$$

第二節　　深度網路(deep learning)

深度網路的典型為全連結類神經網路，這是深度網路的最早的理論與應用。其目的是希望達到模仿人類神經元的架構並能產生出讓機器「學習」的功能。整體上的功能是提供該網路一組陣列形式的輸入宇標籤，藉由輸入計算出的值與提供之標籤比對並計算出錯誤，利用此錯誤做梯度下降與倒傳遞以修正其錯誤，進而達到學習的基礎。

其中，梯度下降的做法是針對計算錯誤率的公式進行微分，在此微分的動作中，我們可以取得該圖形當下的梯度值。當梯度值由原本的數值經過不斷的更新降至趨近於零時，我們便可使整個網路接近所需要建成的模型。在這個過程中，由於神經網路不只一個層次，因此我們需要利用到倒傳遞的技術將錯誤更新的資訊傳遞下去。以下便針對此架構的相關公式與細節進行分析與探討。

一、非線性化

- Sigmoid 函式，將輸出值限縮在 0 至 1 之間，使用公式如下:

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

- Hyperbolic Tangent 函式，將輸出值限縮至-1 至 1 之間，使用公式如下:

$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

二、錯誤評估(Error Function)

- 平方差公式(Mean squared error)，用來計算誤差的方法，雖然也可以使用絕對值做誤差計算，但使用絕對值來計算的話在後面的微分不方便實作。其函式如下：

$$Error = \frac{1}{n}\sum_{i=1}^{n}(Pred_i - True_i)^2$$

其中，n是總數Pred是算出來的預估值，True是實際希望得到的值。

三、單顆類神經架構

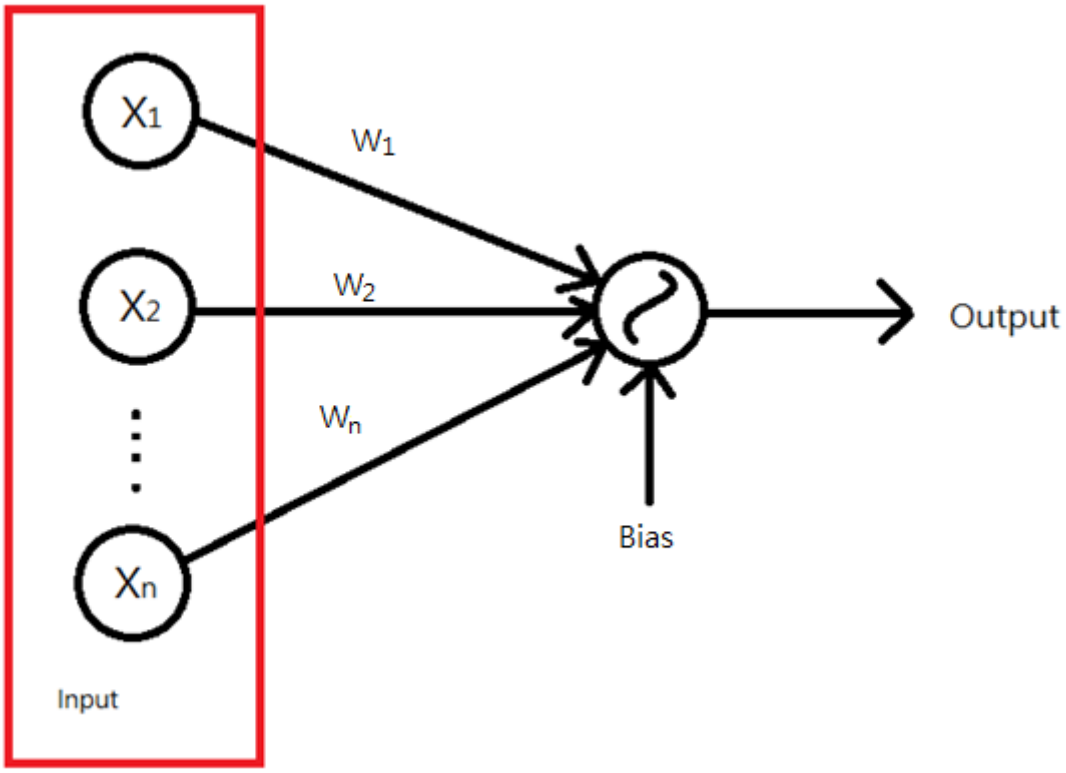


圖 四-II 單顆神經元架構

如上圖，輸入為X_1到X_n，權重為W_1到W_n，則饋式運算如下所示：

$$Output = X_1W_1 + X_2W_2 + \cdots + X_nW_n + b$$

但是這種做法卻有缺點，由於其可分類的變化度不夠，因此做了些

修改，套上 sigmoid function，如此一來函數就會修改成：

$$Output \quad = \quad \sigma(X_1W_1 + X_2W_2 + \cdots + X_nW_n + b)$$

四、全連結網路架構

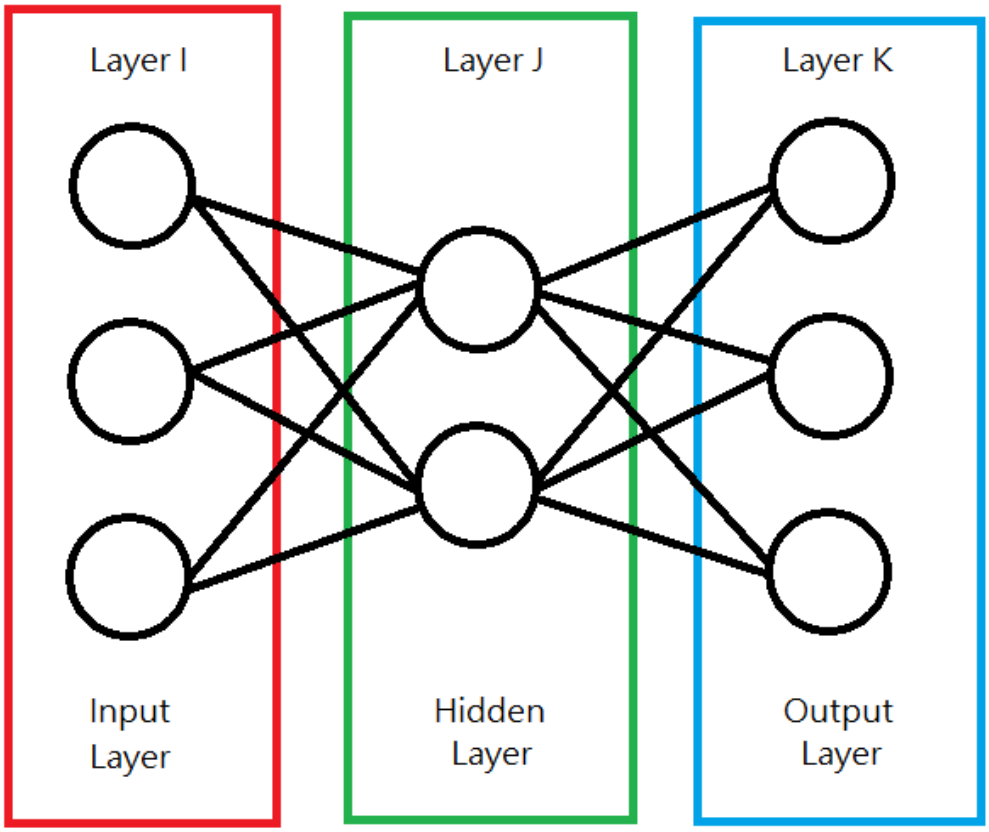


圖 四-III 全連結網路架構

- 前饋式運作流程：

輸出層(Output Layer)的任一節點，由隱藏層提供資訊並向右運算。而輸出層的任一節點運算如下所示：

$$Output_k = \sigma\left(\sum_{x=1}^{J} Output_x \times W_x^{JK} + bias\right)$$

隱藏層(Hidden Layer)的任一節點，由輸入層提供資訊並向右運算。而隱藏層的任一節點運算如下所示：

$$Output_J = \sigma\left(\sum_{x=1}^{I} Input_x \times W_x^{IJ} + bias\right)$$

- 倒傳遞運作流程：

Sigmoid 函式的倒傳遞推導：

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{(1+e^{-x})-1}{(1+e^{-x})^2}$$
$$= \frac{(1+e^{-x})}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} = \frac{(1+e^{-x})}{(1+e^{-x})^2} - \left(\frac{1}{1+e^{-x}}\right)^2$$
$$= \frac{1}{1+e^{-x}} - \left(\frac{1}{1+e^{-x}}\right)^2 = \sigma(x) - \sigma(x)^2$$

*注意，使用公式

$$\frac{\mathrm{d}}{\mathrm{dx}}\left[\frac{\mathrm{f(x)}}{\mathrm{g(x)}}\right] = \frac{\mathrm{f'(x)g(x) - f(x)g'(x)}}{\mathrm{g^2(x)}}$$

可得到：$\sigma' = \sigma(1-\sigma)$

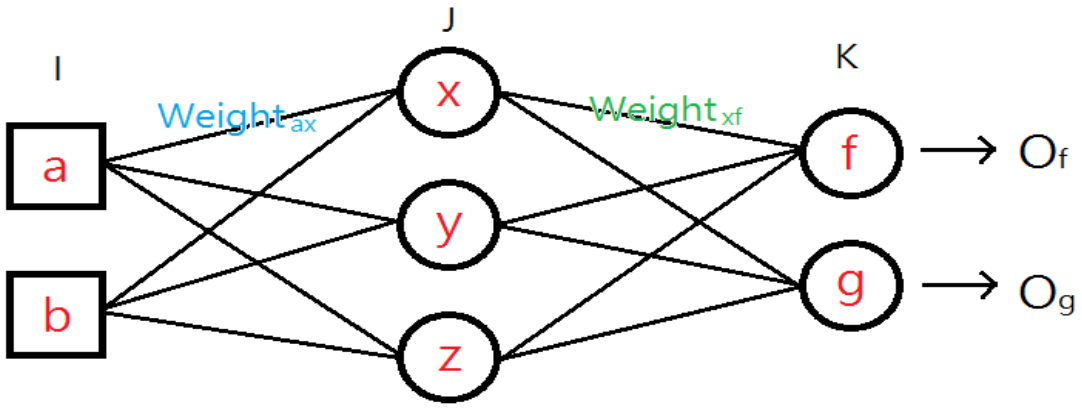


圖 四-IV 全連結網路示意圖 I

輸出層(Output Layer)的倒傳遞推導：

$$\frac{\partial Error}{\partial Weight_{xf}} = \frac{\partial}{\partial Weight_{xf}}\frac{1}{2}\sum_{K}(Output_K - Target_K)^2$$
$$= \frac{\partial}{\partial Weight_{xf}}\frac{1}{2}\left[\left(Output_f - Target_f\right)^2 + \left(Output_g - Target_g\right)^2\right]$$
$$= \frac{\partial}{\partial Weight_{xf}}\frac{1}{2}\left(Output_f - Target_f\right)^2$$
$$= \frac{\partial}{\partial Output_f}\frac{1}{2}\left(Output_f - Target_f\right)^2\frac{\partial Output_f}{\partial Weight_{xf}}$$

$$= (Output_f - Target_f)\frac{\partial Output_f}{\partial Weight_{xf}}$$
$$= (Output_f - Target_f)\frac{\partial Output_f}{\partial Input_f}\frac{\partial Input_f}{\partial Weight_{xf}}$$
$$= (Output_f - Target_f)\frac{\partial \sigma(Input_f)}{\partial Input_f}\frac{\partial Input_f}{\partial Weight_{xf}}$$
$$= (Output_f - Target_f)\sigma(Input_f)[1 - \sigma(Input_f)]\frac{\partial Input_f}{\partial Weight_{xf}}$$
$$= (Output_f - Target_f)Output_f(1 - Output_f)\frac{\partial Input_f}{\partial Weight_{xf}}$$
$$= (Output_f - Target_f)Output_f(1 - Output_f)\frac{\partial}{\partial Weight_{xf}}(Output_x Weight_{xf} + Output_y Weight_{yf} + Output_z Weight_{zf} + Bias_f)$$
$$= (Output_f - Target_f)Output_f(1 - Output_f)\frac{\partial}{\partial Weight_{xf}}(Output_x Weight_{xf})$$
$$= (Output_f - Target_f)Output_f(1 - Output_f)Output_x$$

$$\frac{\partial Error}{\partial Bias_f} = \frac{\partial}{\partial Bias_f}\frac{1}{2}\sum_K (Output_K - Target_K)^2$$
$$= \frac{\partial}{\partial Bias_f}\frac{1}{2}\left[(Output_f - Target_f)^2 + (Output_g - Target_g)^2\right]$$
$$= \frac{\partial}{\partial Bias_f}\frac{1}{2}(Output_f - Target_f)^2$$
$$= \frac{\partial}{\partial Output_f}\frac{1}{2}(Output_f - Target_f)^2\frac{\partial Output_f}{\partial Bias_f}$$
$$= (Output_f - Target_f)\frac{\partial Output_f}{\partial Bias_f}$$
$$= (Output_f - Target_f)\frac{\partial Output_f}{\partial Input_f}\frac{\partial Input_f}{\partial Bias_f}$$
$$= (Output_f - Target_f)\frac{\partial \sigma(Input_f)}{\partial Input_f}\frac{\partial Input_f}{\partial Bias_f}$$
$$= (Output_f - Target_f)\sigma(Input_f)[1 - \sigma(Input_f)]\frac{\partial Input_f}{\partial Bias_f}$$
$$= (Output_f - Target_f)Output_f(1 - Output_f)\frac{\partial Input_f}{\partial Bias_f}$$

$$= (Output_f - Target_f) Output_f (1 - Output_f) \frac{\partial}{\partial Bias_f} (Output_x Weight_{xf} + Output_y Weight_{yf} + Output_z Weight_{zf} + Bias_f)$$

$$= (Output_f - Target_f) Output_f (1 - Output_f) \frac{\partial}{\partial Bias_f} (Bias_f)$$

$$= (Output_f - Target_f) Output_f (1 - Output_f)$$

為了方便與簡化算式，我們將會把上述式子中取出一段以代數δ取代，其式子如下：

$$\delta_f = (Output_f - Target_f) Output_f (1 - Output_f)$$

而上述的推導公式就能簡化成如下所示：

$$\begin{cases} \frac{\partial Error}{\partial Weight_{xf}} = \delta_f Output_x \\ \frac{\partial Error}{\partial Bias_f} = \delta_f \end{cases}$$

隱藏層(Hidden Layer)的倒傳遞推導：

$$\frac{\partial Error}{\partial Weight_{ax}} = \frac{\partial}{\partial Weight_{ax}} \frac{1}{2} \sum_K (Output_K - Target_K)^2$$

$$= \frac{\partial}{\partial Weight_{ax}} \frac{1}{2} \left[(Output_f - Target_f)^2 + (Output_g - Target_g)^2 \right]$$

$$= \frac{\partial}{\partial Weight_{ax}} \frac{1}{2} (Output_f - Target_f)^2 + \frac{\partial}{\partial Weight_{ax}} \frac{1}{2} (Output_g - Target_g)^2$$

$$= (Output_f - Target_f) \frac{\partial Output_f}{\partial Weight_{ax}} + (Output_g - Target_g) \frac{\partial Output_g}{\partial Weight_{ax}}$$

$$= (Output_f - Target_f) \frac{\partial Output_f}{\partial Input_f} \frac{\partial Input_f}{\partial Weight_{ax}} + (Output_g - Target_g) \frac{\partial Output_g}{\partial Input_g} \frac{\partial Input_g}{\partial Weight_{ax}}$$

$$= (Output_f - Target_f) Output_f (1 - Output_f) \frac{\partial Input_f}{\partial Weight_{ax}} + (Output_g - Target_g) Output_g (1 - Output_g) \frac{\partial Input_g}{\partial Weight_{ax}}$$

$$
\begin{aligned}
&= (Output_f - Target_f) Output_f (1 \\
&\quad - Output_f) \frac{\partial}{\partial Weight_{ax}} (Output_x Weight_{xf} \\
&\quad + Output_y Weight_{yf} + Output_z Weight_{zf} + Bias_f) \\
&\quad + (Output_g - Target_g) Output_g (1 \\
&\quad - Output_g) \frac{\partial}{\partial Weight_{ax}} (Output_x Weight_{xg} \\
&\quad + Output_y Weight_{yg} + Output_z Weight_{zg} + Bias_g) \\
&= (Output_f - Target_f) Output_f (1 \\
&\quad - Output_f) \frac{\partial}{\partial Weight_{ax}} (Output_x Weight_{xf}) \\
&\quad + (Output_g - Target_g) Output_g (1 \\
&\quad - Output_g) \frac{\partial}{\partial Weight_{ax}} (Output_x Weight_{xg}) \\
&= (Output_f - Target_f) Output_f (1 \\
&\quad - Output_f) \frac{\partial (Output_x Weight_{xf})}{\partial Output_x} \frac{\partial Output_x}{\partial Weight_{ax}} \\
&\quad + (Output_g - Target_g) Output_g (1 \\
&\quad - Output_g) \frac{\partial (Output_x Weight_{xg})}{\partial Output_x} \frac{\partial Output_x}{\partial Weight_{ax}} \\
&= (Output_f - Target_f) Output_f (1 - Output_f) Weight_{xf} \frac{\partial Output_x}{\partial Weight_{ax}} \\
&\quad + (Output_g - Target_g) Output_g (1 \\
&\quad - Output_g) Weight_{xg} \frac{\partial Output_x}{\partial Weight_{ax}} \\
&= (Output_f - Target_f) Output_f (1 \\
&\quad - Output_f) Weight_{xf} \frac{\partial Output_x}{\partial Input_x} \frac{\partial Input_x}{\partial Weight_{ax}} \\
&\quad + (Output_g - Target_g) Output_g (1 \\
&\quad - Output_g) Weight_{xg} \frac{\partial Output_x}{\partial Input_x} \frac{\partial Input_x}{\partial Weight_{ax}} \\
&= (Output_f - Target_f) Output_f (1 - Output_f) Weight_{xf} Output_x (1 \\
&\quad - Output_x) \frac{\partial Input_x}{\partial Weight_{ax}} \\
&\quad + (Output_g - Target_g) Output_g (1 \\
&\quad - Output_g) Weight_{xg} Output_x (1 - Output_x) \frac{\partial Input_x}{\partial Weight_{ax}}
\end{aligned}
$$

$$= (Output_f - Target_f)Output_f(1 - Output_f)Weight_{xf}Output_x(1 - Output_x)\frac{\partial}{\partial Weight_{ax}}(aWeight_{ax} + bWeight_{bx} + Bias_x) + (Output_g - Target_g)Output_g(1 - Output_g)Weight_{xg}Output_x(1 - Output_x)\frac{\partial}{\partial Weight_{ax}}(aWeight_{ax} + bWeight_{bx} + Bias_x)$$

$$= (Output_f - Target_f)Output_f(1 - Output_f)Weight_{xf}Output_x(1 - Output_x)\frac{\partial}{\partial Weight_{ax}}(aWeight_{ax}) + (Output_g - Target_g)Output_g(1 - Output_g)Weight_{xg}Output_x(1 - Output_x)\frac{\partial}{\partial Weight_{ax}}(aWeight_{ax})$$

$$= (Output_f - Target_f)Output_f(1 - Output_f)Weight_{xf}Output_x(1 - Output_x)a + (Output_g - Target_g)Output_g(1 - Output_g)Weight_{xg}Output_x(1 - Output_x)a$$

經過整理後，可精簡程如下式子：

$$= Output_x(1 - Output_x)a\left[(Output_f - Target_f)Output_f(1 - Output_f)Weight_{xf} + (Output_g - Target_g)Output_g(1 - Output_g)Weight_{xg}\right]$$

$$= Output_x(1 - Output_x)a\sum_{x\epsilon K}\delta_k Weight_{JK}$$

$$\frac{\partial Error}{\partial Bias_x} = \frac{\partial}{\partial Bias_x}\frac{1}{2}\sum_K (Output_K - Target_K)^2$$

$$= \frac{\partial}{\partial Bias_x}\frac{1}{2}\left[(Output_f - Target_f)^2 + (Output_g - Target_g)^2\right]$$

$$= \frac{\partial}{\partial Bias_x}\frac{1}{2}(Output_f - Target_f)^2 + \frac{\partial}{\partial Bias_x}\frac{1}{2}(Output_g - Target_g)^2$$

$$= (Output_f - Target_f)\frac{\partial Output_f}{\partial Bias_x} + (Output_g - Target_g)\frac{\partial Output_g}{\partial Bias_x}$$

$$= \left(Output_f - Target_f\right)\frac{\partial Output_f}{\partial Input_f}\frac{\partial Input_f}{\partial Bias_x} + \left(Output_g - Target_g\right)\frac{\partial Output_g}{\partial Input_g}\frac{\partial Input_g}{\partial Bias_x}$$

$$= \left(Output_f - Target_f\right)Output_f\left(1 - Output_f\right)\frac{\partial Input_f}{\partial Bias_x} + \left(Output_g - Target_g\right)Output_g\left(1 - Output_g\right)\frac{\partial Input_g}{\partial Bias_x}$$

$$= \left(Output_f - Target_f\right)Output_f\left(1 - Output_f\right)\frac{\partial}{\partial Bias_x}\left(Output_x Weight_{xf} + Output_y Weight_{yf} + Output_z Weight_{zf} + Bias_f\right) + \left(Output_g - Target_g\right)Output_g\left(1 - Output_g\right)\frac{\partial}{\partial Bias_x}\left(Output_x Weight_{xg} + Output_y Weight_{yg} + Output_z Weight_{zg} + Bias_g\right)$$

$$= \left(Output_f - Target_f\right)Output_f\left(1 - Output_f\right)\frac{\partial}{\partial Bias_x}\left(Output_x Weight_{xf}\right) + \left(Output_g - Target_g\right)Output_g\left(1 - Output_g\right)\frac{\partial}{\partial Bias_x}\left(Output_x Weight_{xg}\right)$$

$$= \left(Output_f - Target_f\right)Output_f\left(1 - Output_f\right)\frac{\partial\left(Output_x Weight_{xf}\right)}{\partial Output_x}\frac{\partial Output_x}{\partial Bias_x} + \left(Output_g - Target_g\right)Output_g\left(1 - Output_g\right)\frac{\partial\left(Output_x Weight_{xg}\right)}{\partial Output_x}\frac{\partial Output_x}{\partial Bias_x}$$

$$= \left(Output_f - Target_f\right)Output_f\left(1 - Output_f\right)Weight_{xf}\frac{\partial Output_x}{\partial Bias_x} + \left(Output_g - Target_g\right)Output_g\left(1 - Output_g\right)Weight_{xg}\frac{\partial Output_x}{\partial Bias_x}$$

$$= (Output_f - Target_f)Output_f(1 - Output_f)Weight_{xf}\frac{\partial Output_x}{\partial Input_x}\frac{\partial Input_x}{\partial Bias_x} + (Output_g - Target_g)Output_g(1 - Output_g)Weight_{xg}\frac{\partial Output_x}{\partial Input_x}\frac{\partial Input_x}{\partial Bias_x}$$

$$= (Output_f - Target_f)Output_f(1 - Output_f)Weight_{xf}Output_x(1 - Output_x)\frac{\partial Input_x}{\partial Bias_x} + (Output_g - Target_g)Output_g(1 - Output_g)Weight_{xg}Output_x(1 - Output_x)\frac{\partial Input_x}{\partial Bias_x}$$

$$= (Output_f - Target_f)Output_f(1 - Output_f)Weight_{xf}Output_x(1 - Output_x)\frac{\partial}{\partial Bias_x}(aWeight_{ax} + bWeight_{bx} + Bias_x) + (Output_g - Target_g)Output_g(1 - Output_g)Weight_{xg}Output_x(1 - Output_x)\frac{\partial}{\partial Bias_x}(aWeight_{ax} + bWeight_{bx} + Bias_x)$$

$$= (Output_f - Target_f)Output_f(1 - Output_f)Weight_{xf}Output_x(1 - Output_x)\frac{\partial}{\partial Bias_x}(Bias_x) + (Output_g - Target_g)Output_g(1 - Output_g)Weight_{xg}Output_x(1 - Output_x)\frac{\partial}{\partial Bias_x}(Bias_x)$$

$$= (Output_f - Target_f)Output_f(1 - Output_f)Weight_{xf}Output_x(1 - Output_x) + (Output_g - Target_g)Output_g(1 - Output_g)Weight_{xg}Output_x(1 - Output_x)$$

經過整理後，可精簡程如下式子：

$$= Output_x(1 - Output_x)\left[(Output_f - Target_f)Output_f(1 - Output_f)Weight_{xf} + (Output_g - Target_g)Output_g(1 - Output_g)Weight_{xg}\right]$$

$$= Output_x(1 - Output_x)\sum_{x\epsilon K}\delta_k Weight_{JK}$$

- 梯度下降與權重的更新：

  輸出層(Output Layer)的權重更新：

$$\delta_k = Output_k(1 - Output_k)(Output_k - Target_k)$$

  隱藏層(Hidden Layer)的權重更新：

$$\delta_j = Output_j\left(1 - Output_j\right)\sum_{x\epsilon K}\delta_x Weight_{jx}$$

  權重更新的通式：

$$\begin{cases} Weight_{new} = Weight_{old} + \left(-\eta\delta_{Layer}Output_{Layer-1}\right) \\ Bias_{new} = Bias_{old} + \eta\delta_{Layer} \end{cases}, \eta\text{為學習率}$$

- 陣列角度全連結網路架構：

針對多類別的錯誤函示而言，當有 c 個類別與 N 個訓練樣本要計算與調整，使用以下之錯誤函數：

$$E^N = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{c}(t_k^n - y_k^n)^2$$

其中，t_k^n代表第 n 群訓練樣本中某 c 類別實際的標籤矩陣，而y_k^n代表著第 n 群訓練樣本中某 c 類別實際的輸出值。

- 前饋式運作流程：

$$Output^l = f(Weight^l Output^{l-1} + Bias^l)$$

其中，l是層，而$f(\cdot)$多數常見使用的函式是 sigmoid function 或是 hyperbolic tangent function。為了方便計算與一致性，此處將使用 sigmoid function。同時由此可知，此架構與前面的架構一樣，差別在於此處的每個變數都是以陣列形式表示。為了方便解釋，此處將使用前面的圖與變數命名規則方便理解：

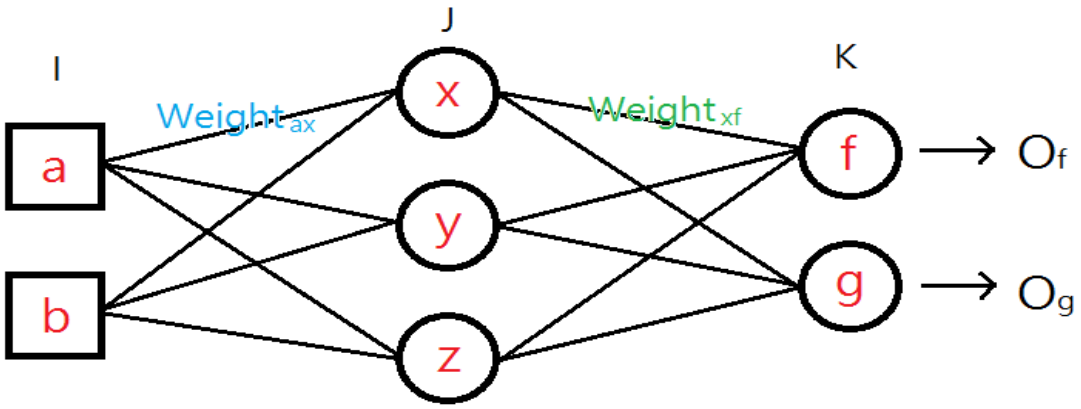


圖 四-V 全連結網路示意圖 II

由輸入層 I 至隱藏層 J 的角度出發，a 與 b 為輸入，輸出要為 x, y 與

z，則矩陣化的式子將會如下所示。

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \sigma\left(\begin{bmatrix} Weight_{ax}Weight_{bx} \\ Weight_{ay}Weight_{by} \\ Weight_{az}Weight_{bz} \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix} + \begin{bmatrix} Bias_x \\ Bias_y \\ Bias_z \end{bmatrix}\right)$$

同樣的，隱藏層 J 至輸出層 K 將會如同

$$\begin{bmatrix} f \\ g \end{bmatrix} = \sigma\left(\begin{bmatrix} Weight_{xf}Weight_{yf}Weight_{zf} \\ Weight_{xg}Weight_{yg}Weight_{zg} \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} Bias_f \\ Bias_g \end{bmatrix}\right)$$

- 倒傳遞運作流程：

概念如同上面所示，是由單顆節點的延伸，因此倒傳遞的運作方式是一樣的。純粹把單顆節點擴充成矩陣節點。為了方便表示，我們將增加代數表示：

$$u^l = Weight^l Output^{l-1} + Bias^l$$

輸出層的倒傳遞：

$$\frac{dError}{dWeight} = \frac{dError}{d\sigma(Input)}\frac{d\sigma(Input)}{dInput}\frac{dInput}{dWeight}$$
$$= Output^{l-1}\left(\sigma'(u^l)\circ(y^n - t^n)\right)^T$$
$$\frac{dError}{dBias} = \frac{dError}{d\sigma(Input)}\frac{d\sigma(Input)}{dInput}\frac{dInput}{dBias} = \sigma'(u^l)\circ(y^n - t^n)$$

此處可對照著前面(單顆節點)所推出的兩個公式與前頁的矩陣表示式對應理解：

$$\begin{cases} \dfrac{\partial Error}{\partial Weight_{xf}} = \delta_f Output_x \\ \delta_f = \left(Output_f - Target_f\right)Output_f\left(1 - Output_f\right) \end{cases}$$

其中的 $\sigma'(u^l)\circ(y^n - t^n)$ 為 $\left(Output_f - Target_f\right)Output_f\left(1 - Output_f\right)$的陣列形式轉換。$Output_f\left(1 - Output_f\right)$對應到的是$\sigma'(u^l)$，而$\left(Output_f - Target_f\right)$對應到的是$(y^n - t^n)$。$\sigma'(u^l)$與$(y^n - t^n)$皆為對應的大小，如下所示：

$$\sigma'(u^l)=\begin{bmatrix}f'\\g'\end{bmatrix}=\sigma'\left(\begin{bmatrix}Weight_{xf}Weight_{yf}Weight_{zf}\\Weight_{xg}Weight_{yg}Weight_{zg}\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}+\begin{bmatrix}Bias_f\\Bias_g\end{bmatrix}\right),$$

$$(y^n-t^n)=\begin{bmatrix}O_f-T_f\\O_g-T_g\end{bmatrix}$$

接著，$Output^{l-1}$可對應到$Output_x$，這時我們將會注意到這兩個矩陣接會長的相似：

$$Output^{l-1}\approx\sigma'(u^l)\circ(y^n-t^n)=\delta^L\approx\begin{bmatrix}\cdot\\\cdot\end{bmatrix}$$

因此在運算時我們必須將$\sigma'(u^l)\circ(y^n-t^n)=\delta^L$轉置$[\ ]^{\mathrm{T}}$才有辦法做矩陣相乘。

- 隱藏層的倒傳遞：

$$\frac{dError}{dWeight}=\frac{dError}{d\sigma(Input)}\frac{d\sigma(Input)}{dInput}\frac{dInput}{dWeight}$$

$$=Output^{l-1}\left((Weight^{l+1})^T\delta^{l+1}\circ\sigma'(u^l)\right)^T$$

$$\frac{dError}{dBias}=\frac{dError}{d\sigma(Input)}\frac{d\sigma(Input)}{dInput}\frac{dInput}{dBias}=(Weight^{l+1})^T\delta^{l+1}\circ\sigma'(u^l)$$

上述兩式可對照以下幾組公式理解。

$$\begin{cases}\dfrac{\partial Error}{\partial Weight_{xf}}=Output_x(1-Output_x)a\displaystyle\sum_{x\epsilon K}\delta_k Weight_{JK}\\[2ex]\delta_j=Output_j\left(1-Output_j\right)\displaystyle\sum_{x\epsilon K}\delta_x Weight_{jx}\\[2ex]\begin{bmatrix}x\\y\\z\end{bmatrix}=\sigma\left(\begin{bmatrix}Weight_{ax}Weight_{bx}\\Weight_{ay}Weight_{by}\\Weight_{az}Weight_{bz}\end{bmatrix}\begin{bmatrix}a\\b\end{bmatrix}+\begin{bmatrix}Bias_x\\Bias_y\\Bias_z\end{bmatrix}\right)\end{cases}$$

其中，$Output_j\left(1-Output_j\right)$對應到$\sigma'(u^l)$而$(Weight^{l+1})^T\delta^{l+1}$對應到$\sum_{x\epsilon K}\delta_x Weight_{jx}$。如果對應到圖的話可把$(Weight^{l+1})^T\delta^{l+1}$寫成以下形式來理解。

$$(Weight^{l+1})^T\delta^{l+1} = \begin{bmatrix} Weight_{ax}Weight_{bx} \\ Weight_{ay}Weight_{by} \\ Weight_{az}Weight_{bz} \end{bmatrix}^T \begin{bmatrix} \delta_x \\ \delta_y \\ \delta_z \end{bmatrix}$$

如此就能理解到為何矩陣運算時必須將 Weight 權重轉置後再相乘。

而最後，a 對應到的為$Output^{l-1}$。若上面的邏輯繼續推論，可知道$(Weight^{l+1})^T\delta^{l+1}\circ\sigma'(u^l)$的形式與$Output^{l-1}$的形式相近，如下：

$$(Weight^{l+1})^T\delta^{l+1}\circ\sigma'(u^l) = \delta^l \approx Output^{l-1} \approx \begin{bmatrix} \vdots \\ \cdot \\ \cdot \end{bmatrix}$$

如此一來就可以解釋為何上述兩項相乘時，後者必須轉置。

- 梯度下降與權重的更新：

  輸出層(Output Layer)的權重更新：

  $$\delta^L = \sigma(u^l)\circ(y^n - t^n)$$

  隱藏層(Hidden Layer)的權重更新：

  $$\delta^l = (Weight^{l+1})^T\delta^{l+1}\circ\sigma(u^l)$$

  權重更新的通式：

  $$\begin{cases} Weight_{new} = Weight_{old} + \left(-\eta Output_{Layer-1}(\delta^*)^T\right) \\ Bias_{new} = Bias_{old} + \eta\delta^* \end{cases}$$ *,η為學習率,*

  $*$ 因層的不同而選擇L或l

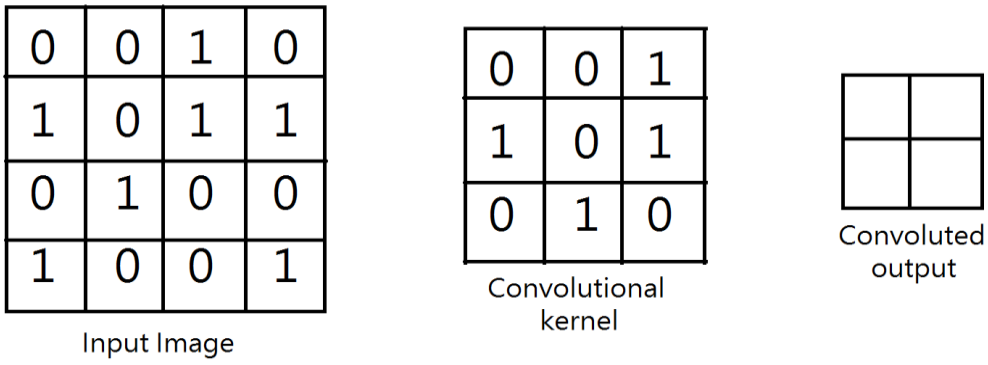
第三節　卷積神經網路（Convolutional Neural Network）

全連結類神經網路雖然已提供了各種計算與應用，但卻在隱藏層擴大之後會導致整體架構肥大增加計算量外，在倒傳遞上會有近乎無法更新的問題存在。在此問題中，唯有發展新的演算法補足來解決此問題。

此時，卷積神經網路即為解決方案。為了讓圖像與影像此類資訊量極大的資料也可以快速與方便的成為輸入資料進行運算與辨識，借用了圖學中的卷積概念進行修改與在設計，成為卷積神經網路的基礎。以下將針對卷積神經網路中的公式與做法做進一步的介紹與探討。

一、前饋式運作流程

- 卷積層(Convolutional Layer)：

基礎架構

卷積步驟

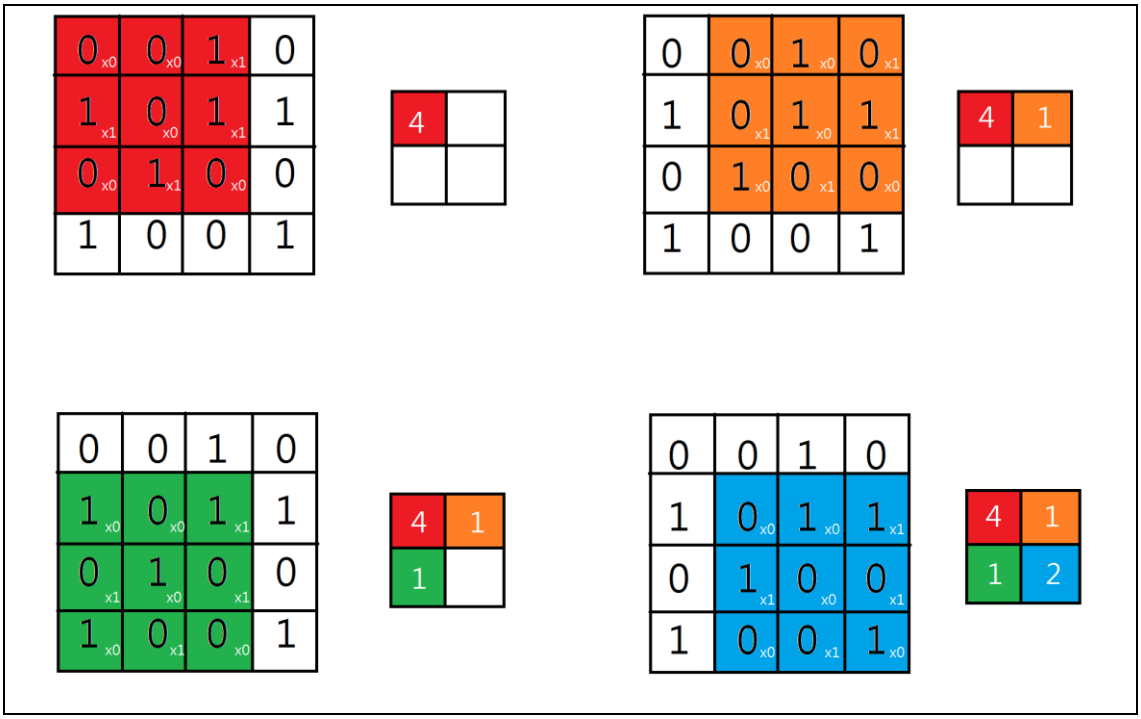


圖 四-VI 卷積層架構表

在卷積神經網路中，捲基層把前一層算出的特徵圖(feature maps)利用可學習的卷積核(learnable kernels)來捲，並將產生出來的圖經過活化函數(activation function)後產生輸出的特徵圖(feature maps)。而每個輸出圖結合了許多張輸入的圖。常態來說，我們有如下公式：

$$Output\_map_j^l = \sigma\left(\sum_{i \epsilon M_j} Output\_map_i^{l-1} * Kernel_{ij}^l + Bias_j^l\right)$$

其中M_j表示一系列的輸入圖。而每個輸出圖會對應到一個加的位移量(Bias)，而對於同一個特定的輸出圖，會使用同一份卷積核(kernel)來卷這些輸入的圖。意思就是，如果輸出圖 J 與輸出圖 K 都是經由 I 圖輸入而得到的，J 與 K 其實使用的為不同的卷積核(kernel)來卷的。在卷積神經網路中，我們一樣透過 sigmoid function(σ)當作非線性化轉換的基礎。

- 取樣層(Pooling Layer):

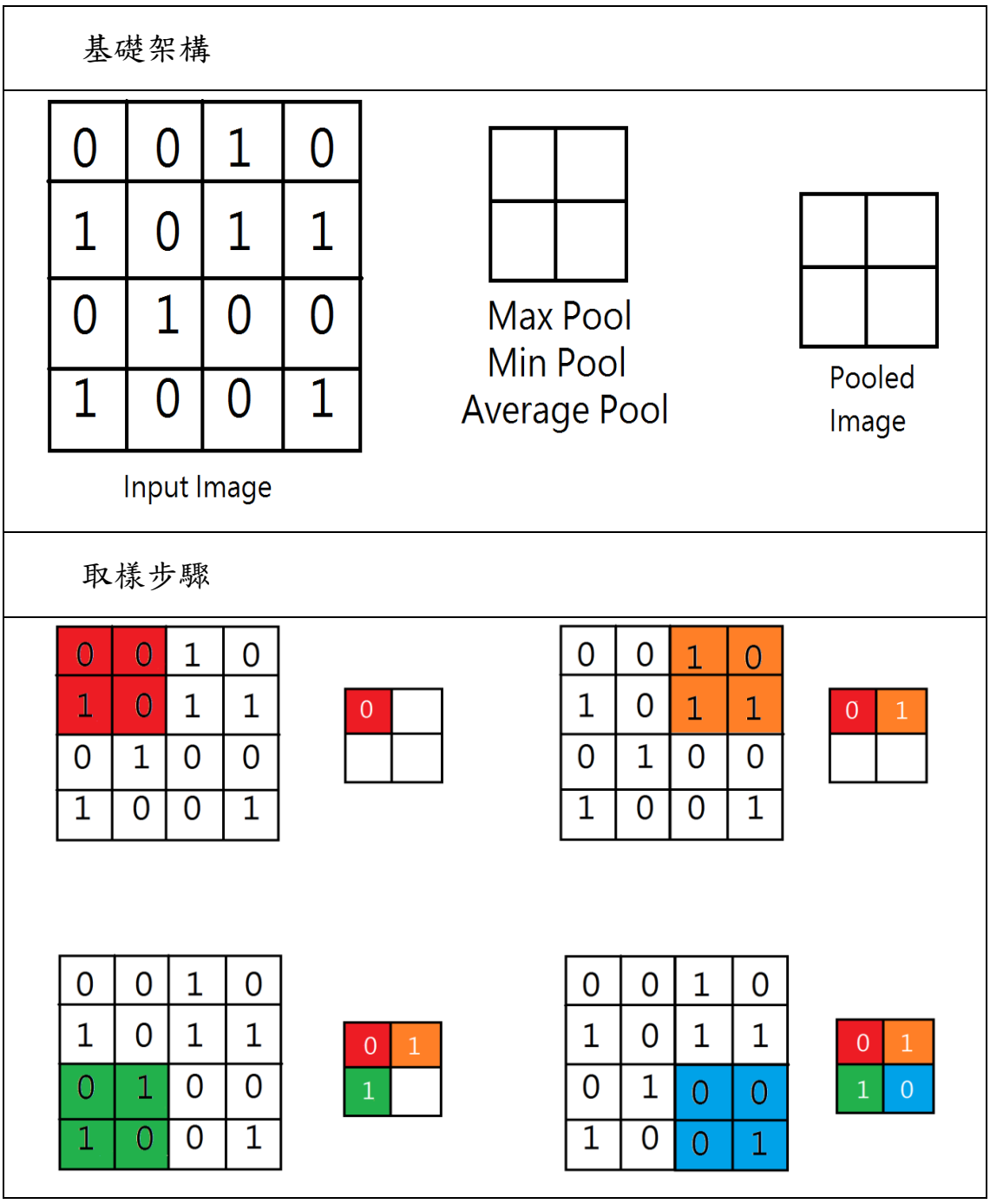


圖 四-VII 子採樣層架構表

子採樣層(subsampling layer)針對輸入圖提供了許多向下採樣(downsampled)的方法。如果此時有 N 張輸入圖，那同時也會產生出恰巧 N 張的輸出圖，差別在輸出的圖會遠小於輸入的圖。如果我們將上述動

作規格化表示，就會如下所示：

$$Output\_map_j^l = \sigma\left(\beta_j^l down\left(Output\_map_j^{l-1}\right) + Bias_j^l\right)$$

其中$down(\cdot)$為任一種子採樣的函式，相對於資料輸入的屬性與所對應的標籤不同，在取樣層將採行不同的數據壓縮做法，這會在後面的章節提及。常見來說這個函式會將 NxN 個輸入圖的區塊做計算，因此輸出圖將會是輸入圖每邊 N 倍小。根據需求不同也會提供先加後乘的 β 與 Bias 權重後再做非線性轉換，產生出輸出的圖。另外其實我們也可以不含任何β與 Bias 權重，而直接將圖經過簡單的子採樣後向下傳遞。

若我們將上述的兩個層以圖像化表示，將會如下圖所示：

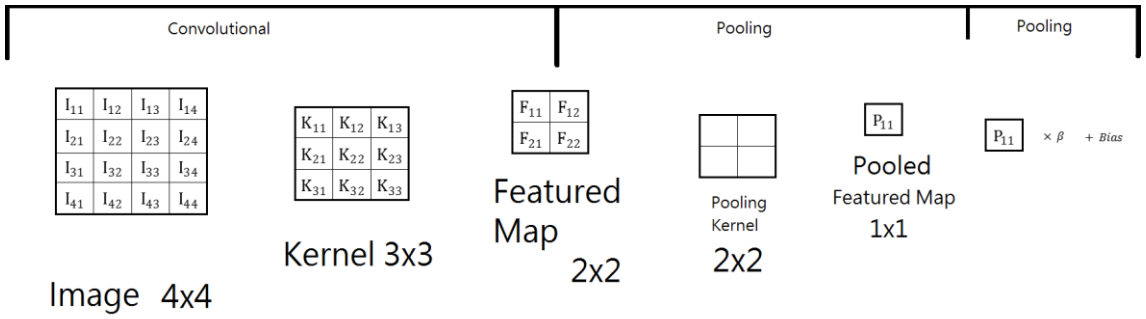


圖 四-VIII 完整卷積神經網路圖 I

$$\begin{cases} Convolutional \begin{cases} F_{11} = \sigma\left(\begin{matrix} I_{11}K_{11} + I_{12}K_{12} + I_{13}K_{13} + I_{21}K_{21} + I_{22}K_{22} \\ +I_{23}K_{23} + I_{31}K_{31} + I_{32}K_{32} + I_{33}K_{33} + Bias_{Kernel} \end{matrix}\right) \\ F_{12} = \sigma\left(\begin{matrix} I_{12}K_{11} + I_{13}K_{12} + I_{14}K_{13} + I_{22}K_{21} + I_{23}K_{22} \\ +I_{24}K_{23} + I_{32}K_{31} + I_{33}K_{32} + I_{34}K_{33} + Bias_{Kernel} \end{matrix}\right) \\ F_{21} = \sigma\left(\begin{matrix} I_{21}K_{11} + I_{22}K_{12} + I_{23}K_{13} + I_{31}K_{21} + I_{32}K_{22} \\ +I_{33}K_{23} + I_{41}K_{31} + I_{42}K_{32} + I_{43}K_{33} + Bias_{Kernel} \end{matrix}\right) \\ F_{22} = \sigma\left(\begin{matrix} I_{22}K_{11} + I_{23}K_{12} + I_{24}K_{13} + I_{32}K_{21} + I_{33}K_{22} \\ +I_{34}K_{23} + I_{42}K_{31} + I_{43}K_{32} + I_{44}K_{33} + Bias_{Kernel} \end{matrix}\right) \end{cases} \\ Pooling \begin{cases} Method1: down(F_{11}, F_{12}, F_{21}, F_{22}) \\ Method2: \beta_j^l down(F_{11}, F_{12}, F_{21}, F_{22}) + Bias_{Pool} \end{cases} \end{cases}$$

二、倒傳遞運作流程

卷積網路的架構如下圖所示，卷積層與取樣層是一組。每層卷積層後面都會跟隨著一層子採樣層。

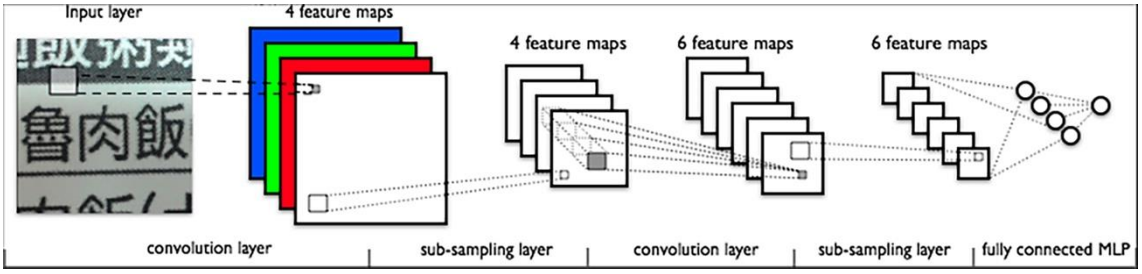


圖 四-IX 完整卷積神經網路圖 I

- 卷積層(Convolutional Layer)：

由倒傳遞的定義中可知，我們必須先加總跟本層有關與對應到下一層的節點變數，並乘上下(l+1)一層所定義的權重(weights)。接著還要乘以由這層向下傳遞的方程式之微分，才有辦法計算與更新本層的權重變數。

以現在的架構中，δ 同樣為卷積輸出特徵圖所對應到子採樣層的依據，而每個特徵圖的節點與子採樣層是一對一對應關係。因此，我們需要先將子採樣層出來的圖還原回與卷積輸出特徵圖一樣的大小，並且對應更新子採樣層公式中使用到的β如下公式，即可算出本層的δ。

$$\delta_j^l = \beta_j^{l+1}\left(\sigma'\left(u_j^l\right)\circ up\left(\delta_j^{l+1}\right)\right)$$

其中 up 公式對應到的是子採樣層縮小圖所用到的 down 公式之微分還原式。由於子採樣層會將圖面的每個邊縮小 n 倍，所以比較可行的作法是利用克羅內克積(Kronecker product)來將縮小的圖還原回原始大小。

我們會將此函式定義如下。

$$up(x) \equiv x \otimes 1_{n \times n}$$

這時，由於取得了δ的算法，因此我們就可以快速的運算 Bias 的微分。只需要將所有的δ做加總。

$$\frac{dError}{dBias_j} = \sum_{u,v} \left(\delta_j^l\right)_{uv}$$

最後我們需要利用倒傳遞計算卷積核中的權重。由於卷積核中的權重是共享的，因此我們的做法將會類似 Bias 的算法，針對對應的權重去加總所有相關的δ。

$$\frac{dError}{dKernel_{ij}^l} = \sum_{u,v} \left(\delta_j^l\right)_{uv} \left(Patch_j^{l-1}\right)_{uv}$$

其中 Patch 是將前一層的特徵圖利用同一個卷積核使用 element-wise product 的形式來進行卷積。

- 取樣層(Pooling Layer):

對應到子採樣層的定義，由於架構中包含的可更新權重只有β跟 Bias，所以在本層只需針對這兩個變數更新。若是在子採樣層所使用的公事並不包含上述兩個權重，基本上在這層是無任何資訊需要更新的，只需要考慮到如何把δ傳回上一層即可。

這時若要計算倒傳遞，對應到架構來講，我們必須將下一層(卷積層)的δ傳回上一層。這時必須尋找一個方法方便我們傳遞這些變數。不過由於在前面推導全連結陣列形式時就已經得到了類似的結論。因此我們

現在即可簡單的沿用前面的概念到這邊來做更有效率的傳遞。

$$\delta_j^l = f'(u_j^l) \circ conv2(\delta_j^{l+1}, rot180(k_j^{l+1}), 'full')$$

這時我們就會使用以上公式來傳遞 δ。再推導中我們會發現利用核的反轉來對應 δ 的卷積可以確實地把下一層的 δ 正確運算與回傳。因此使用上述的公式來方便解釋。

這時我們就可以來計算 β 與 Bias 的倒傳遞。加的 Bias 為同樣的概念，只需簡單的把所有的 δ 加總即可，如下。

$$\frac{dError}{dBias_j} = \sum_{u,v} (\delta_j^l)_{uv}$$

而乘的 β 同樣的會受到原本現在的子採樣層(Pooling Layer)的影響。因此，為了這個需求我們在實際程式撰寫時會建議將這些子採樣過的圖在前饋作業時就先存起來，以方便此處的運算。我們會將此公式定義如下。

$$d_j^l = down(x_i^{l-1})$$

這時我們在計算 β 時就會如下面的公式所式。其中 δ 與 d 中間為陣列的對應位置做相乘，為 Hadamard product。

$$\frac{dError}{d\beta_j} = \sum_{u,v} (\delta_j^l \circ d_j^l)_{uv}$$

額外補充$down(\cdot)$的常見的三種作法將在下面定義。

$$down(x) = \begin{cases} \max(x), \text{取出 pooling kernel 中的最大值來 pass} \\ \min(x), \text{取出 pooling kernel 中的最小值來 pass} \\ average(x), \text{針對 pooling kernel 算出平均值來 pass} \end{cases}$$

第四節　　文字向量(Word2Vec)

Word2Vec [11]利用文章來計算詞的向量，進而能計算詞與詞之間的距離，而用這些距離即可表達。目前常用與常見的 API 為 tensorflow [23]與 gensim [24]。

無論是使用哪一個 API，其所接受的輸入為文字文本資料。由於兩者皆採空白符號作為斷詞依據，因此若要將此應用套用在中文文本內須考量到中文詞彙的結構，而還需斷詞的前處理技術。目前中文文本常見的使用為中央研究院的斷詞系統 [25]與 github 上提供開源方案的結巴斷詞 [26]，這部分偏向於中文文學研究領域為主，因此本研究將視目前所需狀況進行相關調整。

在這個階段目前使用了 tensorflow 官方提供的 word2vec_basic，word2vec 與 word2vec_optimized 三份程式計算詞向量。並且參照另一份線上筆記使用 gensim 計算詞向量並做簡單的詞距比較，關係比較等等。在 word2vec 與 word2vec_basic 中，提供了詞向量的運算與以向量作圖的方法。並運算出以 python binary 格式的詞向量文件。而 word2vec_optimized 為前兩者的優化版本，並可支援 CPU 與 GPU 兩種運算方法。在 gensim 裡提供的運算也是同樣的文字輸入，同時具備支援 bz2 與 gz 壓縮過的文字檔，輸出的詞向量可選擇文字或是二進檔。無論是使用哪一種 API 均有提供簡易的詞距分析等功能。

第五節　　Recurrent Neural Network(RNN)

遞歸神經網路 Recurrent Neural Network (RNN)，是基於全連結網路的變形與延伸，在架構上增加與加強了不同輸入資料的關係。其架構如下圖所示。

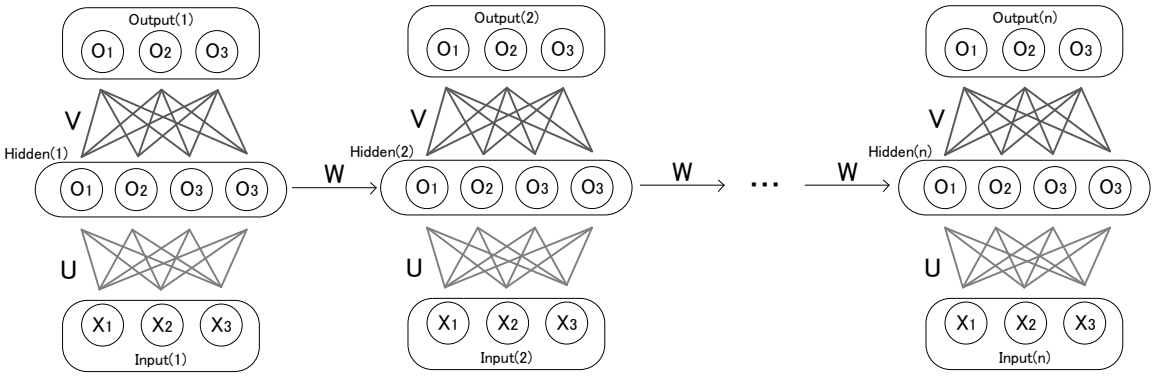


圖 四-X 遞歸神經網路架構圖

RNN 遞歸神經網路屬於全連結類神經網路的其中一種變形，其目的是為了解決有續性資料的分析，好比自然語言中的音訊資料與文本資料。在這兩類的資料當中，前文與後文的資料是有相對應的關係。若利用以往的全連結網路將會切斷這層關係以致資料分析上的緩慢與資源浪費。因此將以往的全連結類神經網路架構中增加了隱藏層之間的資訊傳遞，以保留前後向輸入資料的關係。

其中，在上述架構中的 U，V 與 W 為共享的權重。在 U 至隱藏層中間會經過一個非線性轉換，通常使用 Hyperbolic Tangent Function(tanh)。另外，在 V 至輸出前也會經過一個轉換，使用 Soft-max Function。而本架構中所使用的錯誤函式為 Cross Entropy。

一、**相關公式與推導**

- Hyperbolic Tangent 函式，將輸出值限縮至-1 至 1 之間，使用公式如下

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- Soft-max 函式，將輸出值轉換成機率形式，使用公式如下

$$Softmax(x) = \frac{e^x}{\sum_i e^{x_i}}$$

- Cross Entropy 函式，用來計算與修正錯誤使用，公式如下

$$E(y, \hat{y}) = -\sum_t y_t \, log \, \hat{y}_t$$

二、**前饋運算(Feed Forward)**

本演算法架構中，輸出層是經由隱藏層結合分享權重 V 做計算，而隱藏層是經由輸入結合分享權重 U，若有前一層，參照前一層的隱藏層利用分享權重 W 來合併計算。其公式定義如下：

$$Output_t = Softmax(VH_t)$$

$$Hidden_t = \begin{cases} tanh(UX_t + WH_{t-1}) \ , if \ \ have \ \ above \ \ layer \\ tanh(UX_t) \ , \quad if \ \ no \ \ more \ \ layer \end{cases}$$

其中 X 為輸入 t 表示層次。

三、**倒傳遞(Back Propagate)**

對應架構中可知，有三個變數需要做更新，U、V 與 W。則須將錯誤函式針對此三者變數進行倒傳遞以做權重更新的基礎。

$$\frac{\partial E}{\partial V}=\frac{\partial E}{\partial Softmax(O)}\frac{\partial Softmax(O)}{\partial O}\frac{\partial O}{\partial V}$$
$$=-\sum_t T_t\frac{1}{O_t}\left(O_t-{O_t}^2\right)\otimes H-\sum_t T_{t'}\frac{1}{O_{t'}}(-O_tO_{t'})\otimes H$$
$$=-\sum_t T_t(1-O_t)\otimes H-\sum_t T_{t'}(-O_t)\otimes H$$
$$=\sum_t T_t(O_t-1)\otimes H+\sum_t T_{t'}(O_t)\otimes H$$
$$=\sum_t (T_tO_t-T_t)\otimes H+\sum_t (T_{t'}O_t)\otimes H$$
$$=\sum_t (T_tO_t-T_t+T_{t'}O_t)\otimes H$$
$$=\sum_t (O_t-T_t)\otimes H$$
$$\frac{\partial E}{\partial W}=\frac{\partial E}{\partial Softmax(O)}\frac{\partial Softmax(O)}{\partial O}\frac{\partial O}{\partial tanh(H)}\frac{\partial tanh(H)}{\partial H}\frac{\partial H}{\partial W}$$
$$=\sum_t (O_t-T_t)\otimes V[1-tanh(H)^2]\otimes H^{UpperLayer}$$
$$\frac{\partial E}{\partial U}=\frac{\partial E}{\partial Softmax(O)}\frac{\partial Softmax(O)}{\partial O}\frac{\partial O}{\partial tanh(H)}\frac{\partial tanh(H)}{\partial H}\frac{\partial H}{\partial U}$$
$$=\sum_t (O_t-T_t)\otimes V[1-tanh(H)^2]\otimes\frac{\partial H}{\partial U}$$
$$=\sum_t (O_t-T_t)\otimes V[1-tanh(H)^2]\otimes\frac{\partial}{\partial U}[UX+tanh(H^{UpperLayer})\,W]$$
$$=\sum_t (O_t-T_t)\otimes V[1-tanh(H)^2]$$
$$\otimes\left[X+\frac{\partial\,tanh(H^{UpperLayer})\,W}{\partial\,tanh(H^{UpperLayer})}\frac{\partial\,tanh(H^{UpperLayer})}{\partial H^{UpperLayer}}\frac{\partial H^{UpperLayer}}{\partial U}\right]$$
$$=\sum_t (O_t-T_t)\otimes V[1-tanh(H)^2]$$
$$\otimes[X+W[1-tanh(H^{UpperLayer})^2]\otimes X^{UpperLayer}]$$

其中，$[X+W[1-tanh(H^{UpperLayer})^2]\otimes X^{UpperLayer}]$*成為多層向後推展遞延式*(delta)。

四、**權重更新**

U、V 與 W 的更新方式如下。

$$U_{new} = U_{old} + \left(-\eta \frac{\partial E}{\partial U}\right)$$
$$V_{new} = V_{old} + \left(-\eta \frac{\partial E}{\partial V}\right)$$
$$W_{new} = V_{old} + \left(-\eta \frac{\partial E}{\partial W}\right)$$

第五章　初步研究成果

第一節　　API 比較

	Theano	Tensor Flow	Keras	DIGITS
使用語言	python	python	python	python
API 類型	低階	低階	中階	高階
運算支援度	CPU/GPU	CPU/GPU	CPU/GPU	CPU/GPU
系統支援度	Windows/ Linux/ MacOS	Linux/ MacOS	對應到使用的底層	Linux Ubuntu 14.04
界接 API	無	無	Theano(預設)/ Tensor Flow	Torch/ Caffe

表 五-i API 比較表

第二節　　環境建置

一、使用 theano

Windows 環境下可直接使用 WinPython 套件包，將 theano 引用到標頭後即可使用 CPU 運算。

Windows 環境下若要使用 GPU 運算，詳細的安裝步驟如下。

1. 確認手邊的顯卡支援 CUDA 運算並為 NVIDIA 顯卡。
2. 安裝 Visual Studio，並依照 CUDA 文件安裝對應的版本
3. 安裝 CUDA 7.5，由於目前最新版本的 theano(0.8.2)主要支援最穩定的版本為此版。
4. 下載 cuDNN 函式庫對應到 CUDA 版本並複製至 CUDA 目錄下。

5. 更新顯卡驅動。

6. 此時依照 Nvidia 官網提供的文件所述，找到 simple code 中的 bandwidthTest 與 deviceQuery 兩個範例程式編譯並執行。確認 CUDA 有安裝正確與成功。

7. 接著下載並解壓縮 WinPython，建議版本為 2.7.10.3

8. 在 WinPython 中的 settings 底下新增.theanorc.txt，內容如下

其中，cuda, nvcc,與 winpython 需對應到所安裝的實際版本與路徑做對應的變更。

.theanorc
[global] device = gpu floatx = float32 [lib] cumem = 0.87 [cuda] root = C:¥Program Files¥NVIDIA GPU Computing Toolkit¥CUDA¥v7.5 [nvcc] flags=-LC:¥WinPython-64bit-2.7.10.3¥python-2.7.10.amd64¥libs compiler_bindir=C:¥Program Files (x86)¥Microsoft Visual Studio 12.0¥VC¥bin

9. 最後執行官方提供的範例測試，確定可使用 GPU 運算。

Linux 的 theano GPU 環境建置。我們這裡所建置的是使用 Ubuntu 14.04 LTS 的 64 位元英文版作為建置系統。

1. 安裝 Ubuntu 14.04 x64 LTS 版並更新到最新的套件

2. 利用已下指令安裝 python 與 pip

```
sudo apt-get install python-pip python-dev build-essential
```

3. 為了方便之後的程式撰寫，利用 apt-get install spyder 安裝 spyder

4. 至 NVIDIA 的官方網站安裝 CUDA 7.5，並使用已下指令來正確安裝對應的版本

```
dpkg -i cuda-repo-ubuntu1404-7-5-local_7.5-18_amd64.deb
apt-get update
apt-get install cuda-tookit-7-5
```

5. 接著至 NVIDIA 官方網站下載 cuDNN 5.0，並注意到要找是支援 CUDA 7.5 版的，下載後利用以下指令安裝

```
tar xvzf cudnn-7.5-linux-x64-v5.0-ga.tgz
sudo cp cuda/include/cudnn.h /usr/local/cuda/include
sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
sudo chmod a+r /usr/local/cuda/include/cudnn.h
/usr/local/cuda/lib64/libcudnn*
```

6. 這時再來使用 Ubuntu 內建的圖形化介面安裝已測試過的顯卡驅動並重新開機

7. 重開機後，輸入 nvidia-smi 指令確認 cuda 有正確安裝並正常運作

8. 利用 sudo pip install theano 安裝完 theano 套件

9. 接著到家目錄底下新增.theanorc 的設定檔

.theanorc

```
 [global]
device = gpu
floatx = float32

[lib]
cumem = 0.87

[cuda]
root = /usr/local/cuda-7-5
```

最後用以下指令安裝完缺少的 blas 套件

```
sudo apt-get install libblas-dev liblapack-dev
sudo apt-get install libatlas-base-dev
```

利用 theano 官方提供之 GPU 測試檔確認可使用 GPU 運算

二、使用 tensorflow

目前 tensor flow 所支援的作業系統只有 Linux 與 macOS，因此這邊我們就以 Linux ubuntu 的環境建置為例。CPU 建置上使用 Ubuntu 14.04/16.-04 LTS x64，並使用 sudo apt-get install python-dev python-pip 安裝 python。接著使用 sudo apt-get install spyder 來安裝 spyder IDE 編譯器。接著依照 tensor flow 官方網站使用以下指令做完 CPU 建置。

```
# Ubuntu/Linux 64-bit, CPU only, Python 2.7
(tensorflow)$ export
TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu/t
ensorflow-0.11.0rc2-cp27-none-linux_x86_64.whl
# Python 2
(tensorflow)$ pip install --upgrade $TF_BINARY_URL
```

GPU 建置,同樣的到 NVIDIA 官方網頁下載 CUDA8.0 與 cuDNN5.1 安裝，接著依照 tensor flow 官方網站使用已下指令即可完成建置。

```
# Ubuntu/Linux 64-bit, GPU enabled, Python 2.7
# Requires CUDA toolkit 8.0 and CuDNN v5. For other versions, see
"Install from sources" below.
```

```
$ export
TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu/t
ensorflow-0.11.0rc2-cp27-none-linux_x86_64.whl
# Python 2
$ sudo pip install --upgrade $TF_BINARY_URL
```

三、使用 keras

當前置安裝完 theano 或 tensor flow 作為 backend 後，即可直接使用”pip install keras”來安裝即可使用。

四、Nvidia DIGITS

1. 安裝 Ubuntu 14.04 x64 LTS 英文版

2. 利用以下指令安裝相依套件

```
# Access to CUDA packages
CUDA_REPO_PKG=cuda-repo-ubuntu1404_7.5-18_amd64.deb
wget
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1404/x8
6_64/${CUDA_REPO_PKG} -O /tmp/${CUDA_REPO_PKG}
sudo dpkg -i /tmp/${CUDA_REPO_PKG}
rm -f /tmp/${CUDA_REPO_PKG}

# Access to Machine Learning packages
ML_REPO_PKG=nvidia-machine-learning-repo-ubuntu1404_4.0-
2_amd64.deb
wget http://developer.download.nvidia.com/compute/machine-
learning/repos/ubuntu1404/x86_64/${ML_REPO_PKG} -O
/tmp/${ML_REPO_PKG}
sudo dpkg -i /tmp/${ML_REPO_PKG}
rm -f /tmp/${ML_REPO_PKG}

# Download new list of packages
sudo apt-get update
```

3. 使用 apt-get install digits 完成安裝建置

4. (選用 GPU 建置) 依照 Nvidia 官方網頁安裝 CUDA 8.0 與 cuDNN 5.1

5. (選用 GPU 建置) 使用 sudo apt-get install DIGITS 更新 gpu 支援套件

第三節　Tensor Flow 運作流程

引用官方文件的說法，tensor flow 重新定義與詮釋了程式碼執行運算的概念與流程。整體城市的運算與執行被稱之為作圖(graphs)，使用與定義一系列的流程(Sessions)來做圖。將資料流以 tensors 來表示，並使用自定義的 Variables 變數來管理與儲存資料狀態。

基本上 tensor flow 的程式執行前，將會先利用 Variables 定義完所需使用的變數。接著使用函數來定義所有需要執行的不同 tensors 並將完整的程式包入 Session 裡，因此對於深度網路這類需要常常將資料的質與狀態經常性的存入與讀出的架構就可以使用 Sessions 的概念輕易達成。接著，程式執行的整體流程定義在 Sessions 裡，並使用 run 指令來告訴整體的 Sessions 在哪些地方必須做哪些運算。

第四節　基於線上 CNN 的菜單辨識翻譯 APP 應用

本應用為卷積網路中手寫辨識的延伸。本應用簡單的修改了原始的 LeNet-5 並新增了一至兩個層次，並使用台灣常見的傳統菜單小吃做應用。主旨在建置 APP 與雲端的 CNN 卷積網路辨識菜名並提供對應的菜名翻譯，提供給外國人作為一個便捷觀光的應用。

一、資料訓練

本應用使用的硬體規格為 i7-4790 (3.6 GHz)型主機，具備 20GB 的

記憶體。初期使用 GPU 運算訓練資料。輸入資料由於硬體受限之因，使用的是等比縮小成 512 像素成以 488 像素之樣本來訓練。訓練資料中，為了讓錯誤率最小化，本應用產生了四組不同的輸入資料做實際訓練。A 樣本中的資料內容為真實拍攝菜單之圖像，其中包含旋轉，跟不同縮放比的資料。B 樣本為攝取 A 樣本後經由影像處理軟體進行去被與填入背景色的方式。C 樣本是把 A 與 B 樣本混合在一起當作另一份新的樣本。而樣本 D 則是使用不同的電腦字型產生出的訓練圖。

其中 A 與 B 樣本的截圖如下：

圖 五-I A 與 B 樣本比較圖

而這四組資料經過 1000 個訓練單位後所降低的錯誤率如下表所示。

Categories	Method	Error(%)
A	Pictures with rotation and resize	82.0
B	Pictures with gray noise in background	82.0
C	Mixed Pictures A+B	38.7
D	Picture with gray background, diverse fonts and without rotation and resize	22

表 五-ii 訓練資料比較表

架構中我們所使用的是經過層次修改的網路(橘)，相較於原始的卷積神經網路(藍)，其錯誤率下降狀況如下圖所示。

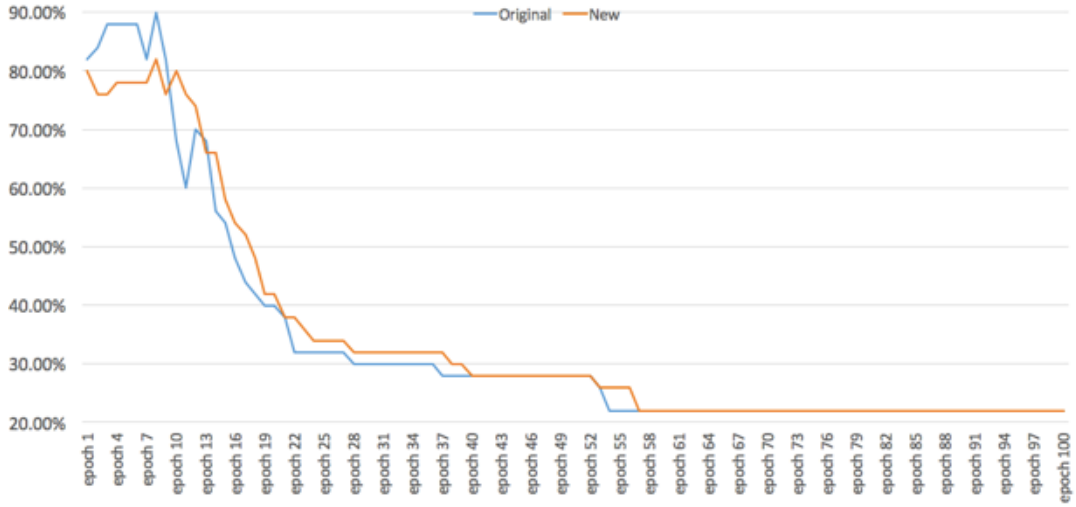


圖 五-II 錯誤率下降圖

二、網頁上線服務

本應用將前面訓練好的模式與架構，藉由 python flask framework 建置網頁服務，如此一來可以讓手機 APP 端上傳預測圖片並取得預測後的標籤。伺服器截圖如下。

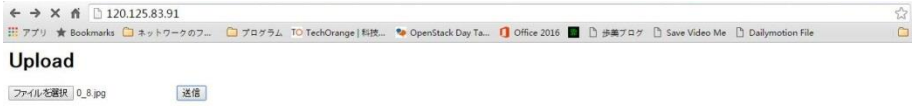


圖 五-III 網頁上傳圖

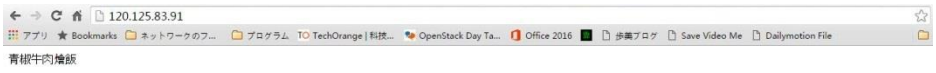


圖 五-IV 辨識後的網頁回傳圖

三、手機 APP

在手機上的應用主要是使用 Android Studio 建置手機 APP。主要提供前面章節敘述之功能，同時希望提供語言界面、查詢歷史與相關設定等功能。介面如下圖所示。

圖 五-V 手機介面示意圖

第六章　預期成果

由於前期「基於線上 CNN 的菜單辨識翻譯 APP 應用」為學習深度學習前期的基礎練習。接下來的研究將不繼續在此應用上著墨。未來的應用將會著重在使用自然語言之文本訓練出可判別字詞語語句所表達之情緒之系統。

有鑑於文字本身是無法使用電腦直接進行計算與判別，因此本研究將採行使用 Word2Vec 演算法先將文本丟入進行訓練，進而計算出詞向量。其中，計算詞向量之 Word2Vec 的常見使用之 API 有 tensorflow 與 gensim 兩種，且分為使用 CPU 硬體做運算與可混合使用 GPU 硬體來加速運算的不同做法。因此本研究希望未來可以在這部分比較並找出最適合的配套作法，讓即使在大量文本的輸入狀況下依然可兼具良好的運算效能與速度的方法出來。

基於 Word2Vec 演算法本身所提供的架構，我們必須將非英語系文字進行判別處理，此部份將借助目前常用的 API 來解決。目前常見的中文字的分詞處理為結巴斷詞與史丹佛大學的中文斷詞系統，本研究將在未來針對此兩項做比較與選擇，比較並找出針對不同輸入文本之適合的方法。

接著，運算出的詞向量將成為遞歸神經網路 Recurrent Neural Network(RNN)的輸入與輸出。本研究在這邊將會遇到較大的困難，對於

文本的挑選與情緒判別的標籤製作上，在中文方面目前較無看到相關資料。本研究希望在未來可多搜集資料並建置相關的情緒判別標籤，以幫助未來其他研究需要在中文文本上的分析。

参考文献

[1] Y. LeCun, "Yann LeCun's Home Page," Director of AI Research, Facebook, 2004. [Online]. Available: Yann LeCun. [Accessed 2015].

[2] L. B. Y. B. a. P. H. Yann LeCunm, "Gradient-Based Lerning Applied to Document Recognition," IEEE, 1998.

[3] Nvidia, “ディープラーニングテクノロジ | NVIDIA,” Nvidia, 2016. [線上]. Available: http://www.nvidia.co.jp/object/deep-learning-jp.html. [存取日期: 8 2016].

[4] Google, “会社情報 – Google,” Google, 19 8 2004. [線上]. Available: https://www.google.com/about/company/. [存取日期: 3 2016].

[5] Google Engineer, “DeepMind,” Google, 2016. [線上]. Available: https://deepmind.com/. [存取日期: 3 2016].

[6] Google Engineer, “Mastering the game of Go with deep neural networks and tree search,” *Nature,* 第 冊 529, 編號 28 January 2016, p. 484 – 489, 2016.

[7] マイクロソフト AI, “りんな,” マイクロソフト, 8 2015. [オンライン]. Available: http://rinna.jp/. [アクセス日: 1 2016].

[8] LINE Corporation, “コミュニケーションアプリ LINE（ライン）,” LINE（ライン）, 4 2013. [オンライン]. Available: https://line.me/ja/. [アクセス日: 11 2015].

[9] Z. C. L. Zachary C. Lipton, "A Critical Review of Recurrent Neural

Networks," Cornell University, Ithaca, New York, U.S., 2015.

[10] I. S. O. V. Wojciech Zaremba, “Recurrent Neural Network Regularization,” Cornell University, Ithaca, New York, U.S., 2015.

[11] I. S. K. C. G. C. a. J. D. Tomas Mikolov, “Distributed Representations of Words and Phrases and their Compositionality,” Neural Information Processing Systems Conference, 2013.

[12] Theano Development Team, “Deep Learning Tutorials — DeepLearning 0.1 documentation,” Theano Development Team, 2013. [線上]. Available: http://deeplearning.net/tutorial/. [存取日期: 9 2015].

[13] デ. 兼. C. 村. 真奈, “ディープラーニング最新動向と技術情報,” 22 7 2016. [オンライン]. Available: https://images.nvidia.com/content/APAC/events/deep-learning-day-2016-jp/NVIDIA-DeepLearning-Intro.pdf. [アクセス日: 10 2016].

[14] Wikipedia Group, "Deep learning - Wikipedia," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Deep_learning. [Accessed 2016].

[15] 維基百科使用者, “深度学习 - 维基百科，自由的百科全书,” 2016. [線上]. Available: https://zh.wikipedia.org/wiki/%E6%B7%B1%E5%BA%A6%E5%AD%A6%E4%B9%A0. [存取日期: 2016].

[16] Wiki Group, "Deep learning History," 22 10 2016. [Online]. Available: https://en.wikipedia.org/wiki/Deep_learning#History.

[Accessed 12 2016].

[17] S. Juergen, “Connectionists: First Deep Learning Networks in 1965,” 7 2014. [線上]. Available: http://mailman.srv.cs.cmu.edu/pipermail/connectionists/2014-July/027158.html. [存取日期: 11 2016].

[18] Wiki Community, “Vanishing gradient problem - Wikipedia,” 29 10 2016. [線上]. Available: https://en.wikipedia.org/wiki/Vanishing_gradient_problem. [存取日期: 11 2016].

[19] S. Hochreiter, “Untersuchungen zu dynamischen neuronalen Netzen,” Technical University Munich, Institute of Computer Science, Arcisstraße 21, 80333 München, 15 Jun 1991.

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "LeNet-5, convolutional neural networks," 1998. [Online]. Available: http://yann.lecun.com/exdb/lenet/. [Accessed 11 2015].

[21] Wikipedia Community, "Recurrent neural network - Wikipedia," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Recurrent_neural_network. [Accessed 10 2016].

[22] A. Karpathy, “The Unreasonable Effectiveness of Recurrent Neural Networks,” 21 5 2015. [線上]. Available: http://karpathy.github.io/2015/05/21/rnn-effectiveness/. [存取日期: 10 2016].

[23] Google, “TensorFlow — an Open Source Software Library for Machine Intelligence,” Google, 9 10 2015. [線上]. Available:

https://www.tensorflow.org. [存取日期: 3 2016].

[24] R. Řehůřek, “gensim: Topic modelling for humans,” various, 2008. [線上]. Available: https://radimrehurek.com/gensim/. [存取日期: 9 2016].

[25] 中央研究院, “中文斷詞系統,” 中央研究院, [線上]. Available: http://ckipsvr.iis.sinica.edu.tw.

[26] MIT, “fxsjy/jieba: 结巴中文分词,” MIT, [線上]. Available: https://github.com/fxsjy/jieba.