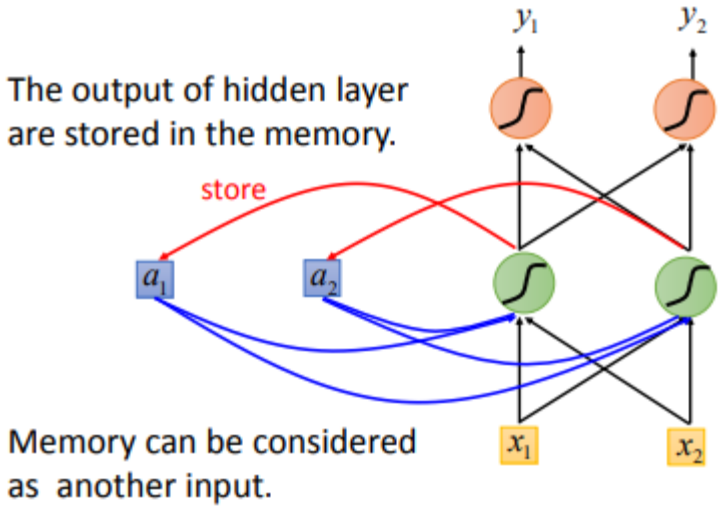
# INDEX

# Recurrent Neural Network

每一次 hidden layer 裡面的 neuron 產生 output 的時候，這個 output 都會被存到 memory 裡，如下圖藍色方塊。
下一次 input 進來的時候，這個 hidden layer 的 neuron 不是只會考慮 input $x_1$ 跟 $x_2$，它還會考慮存在這些 memory 裡面的值。意即對它來說除了 $x_1$ 跟 $x_2$ 以外，memory $a_1$ 與 $a_2$ 的值也會影響到它的 output。



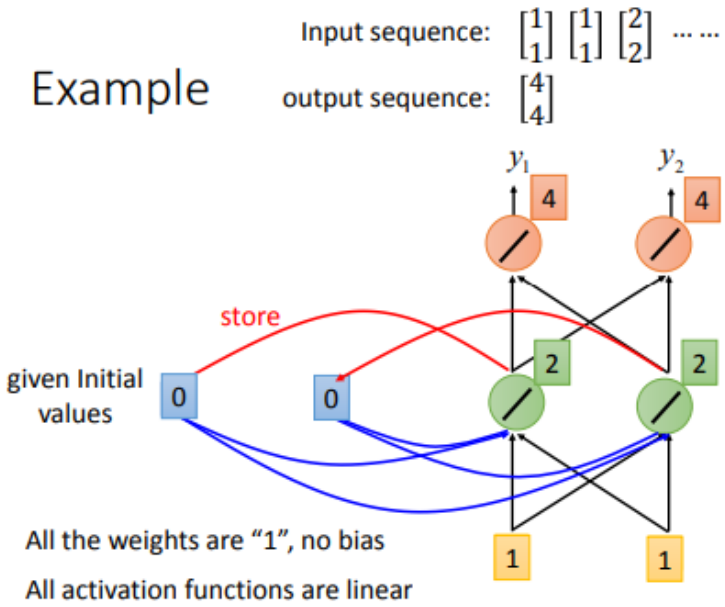

# Example

## 一、Toy Example

假設圖上的 network 所有的 weight 都是 1，且其內所有的 neuron 都沒有任何的 bias，然後假設 activation function 都是 linear。此處的 input 為[11] [11] [22]。

首先使用 RNN 前必須先給 memory 起始值，此處假設還沒有放任何東西之前，memory 裡面的值是 0。
輸入第一個 input [1 1]時，對綠色 neuron 來說，它除了接到 input [1 1]以外，它還接到 memory 的[0 0]，所以其 output 為 2。

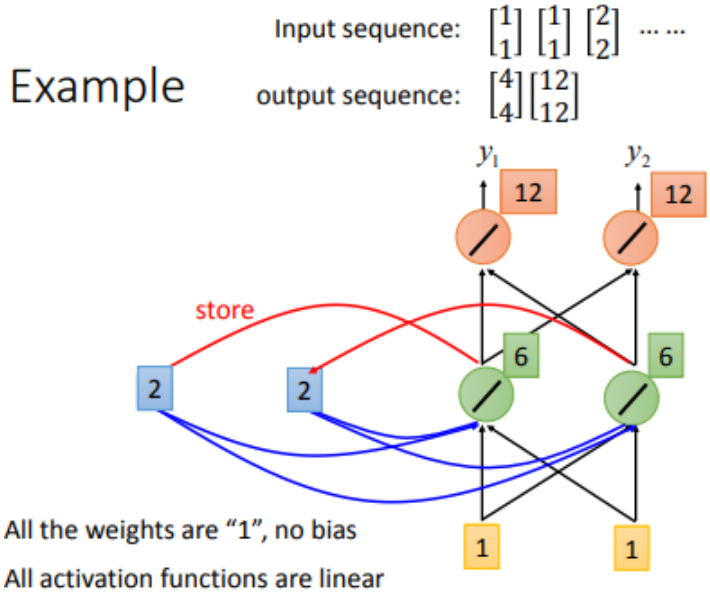
接下來因為所有的 weight 都是 1，所以紅色 neuron 它們的 output 就是 4。也就是說當 input 為[1 1] 的時候，它的 output 就是[4 4]。



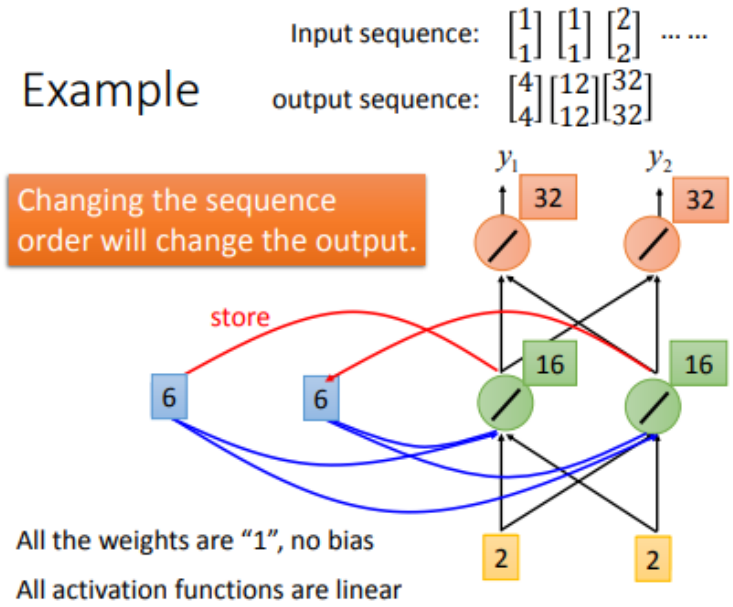

接著 RNN 會把這些綠色 neuron 的 output 存到 memory 裡面，所以 memory 裡面的值就被 update 成 2。

若接下來再輸入[1 1] 的時候，綠色 neuron 的輸入有 4 個，[1 1]跟[2 2]。然後得到的結果為 6。

那最後紅色 neuron 的輸出就是 6 + 6 為 12。因此第二次再輸入[1 1]時，輸出就是[12 12]。



接下來[6 6]又會被存到 memory 裡，而我們的 input 是[2 2]。
這邊每一個綠色的 neuron，它考慮的 4 個 input：[2 2]跟[6 6]，得到的值為 16。
那紅色 neuron 的 output 就是 32。




RNN 在考慮每個 input 時，並不是 independent 的，因此在做 RNN 的 input sequence 順序非常重要。

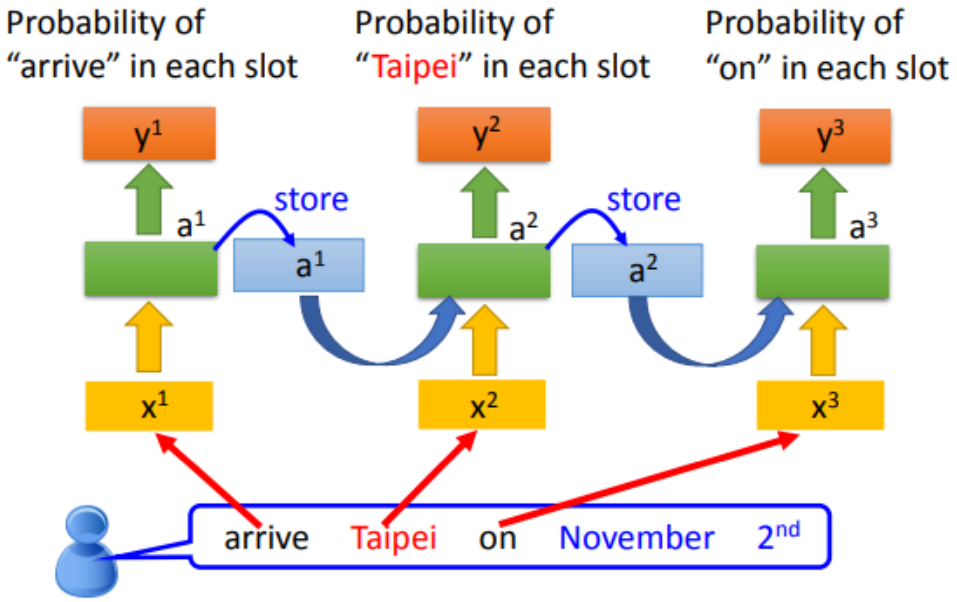
## 二、Slot Filling Problem

假設有一個使用者說 arrive Taipei on November 2nd，那 arrive 就變成一個 vector，丟到 neural network 裡面，其 output 為 $a^1$。

然後根據這個 $a^1$，我們產生 $y^1$，這個 $y^1$ 就是 arrive 屬於哪一個 slot 的機率。接下來 $a^1$ 會被存到 memory 裡。

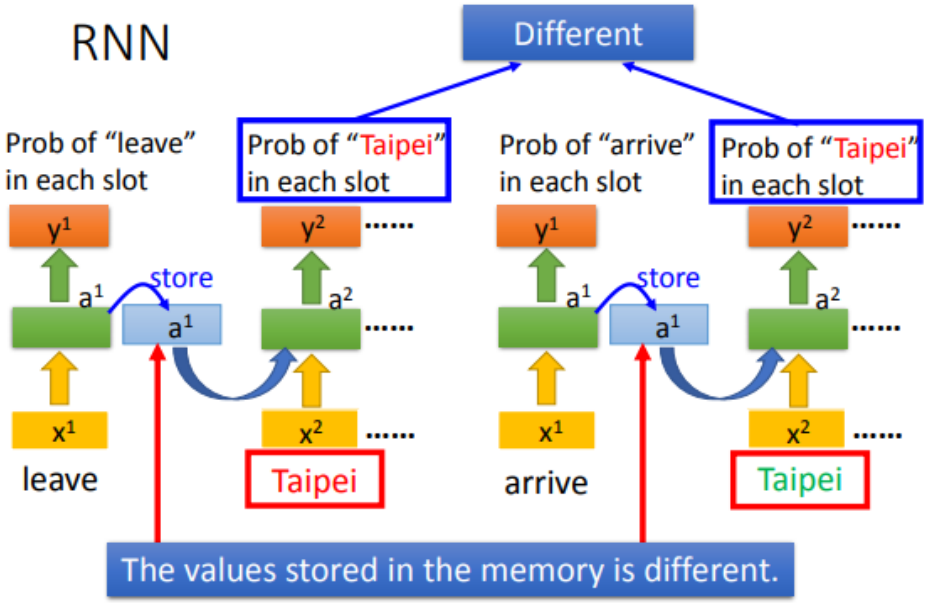
而當 Taipei 會變成 input 時，那這個 hidden layer 會同時考慮 Taipei 這個 input，跟存在 memory 裡面的 $a^1$，得到 $a^2$。

然後再根據 a2 產生 y2，代表 Taipei 屬於哪一個 slot 的機率。以此類推。



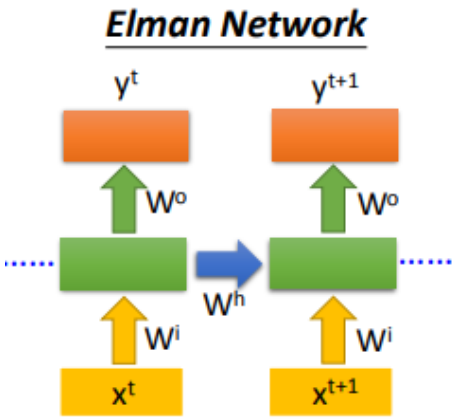

因此透過 RNN 可以做到輸入同一個詞彙時，區分其 output 數值的問題。

舉例而言，同樣是輸入 Taipei 這個詞彙，但是因為紅色 Taipei 前面接的是 leave；而綠色 Taipei 前面接的是 arrive。
因為 leave 跟 arrive 它們的 vector 不一樣，所以 hidden layer 的 output 不同；同理存在 memory 的值也會不同。



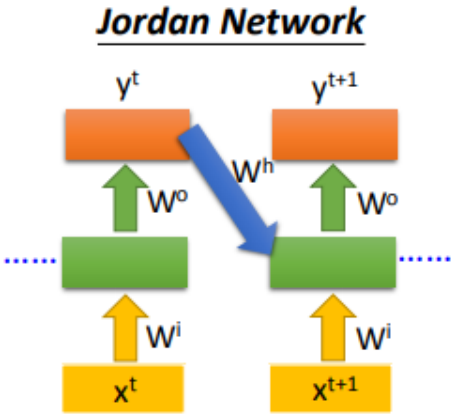
# Different Type of RNN

## 一、Elman Network




前述提出的 RNN 架構即是 Elman Network。也就是把 hidden layer 的值存進 memory，在下一個時間點再讀出來。

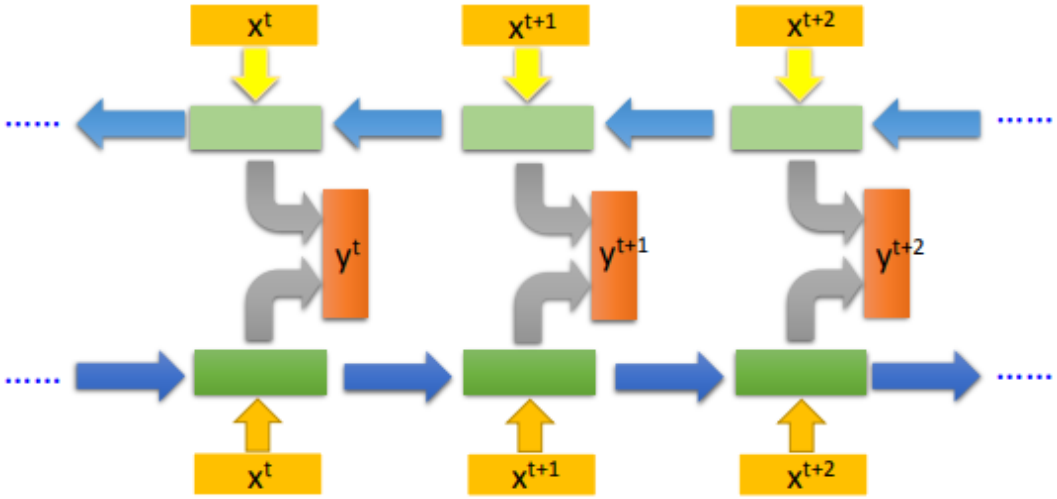
## 二、Jordan Network




該架構把整個 network output 的值存在 memory 裡面。然後它再把 output 的值，在下一個時間點讀進來。

由於 Elman Network 的 hidden layer 沒有 target，所以有點難控制它學到什麼樣的 hidden information，也就是不知道它學到把什麼東西放到 memory 裡面。
而 Jordan Network 是有 target 的，可以比較清楚我們放在 memory 裡面的，是什麼樣的東西，因此可以得到比較好的 performance。

## 三、Bidirectional RNN



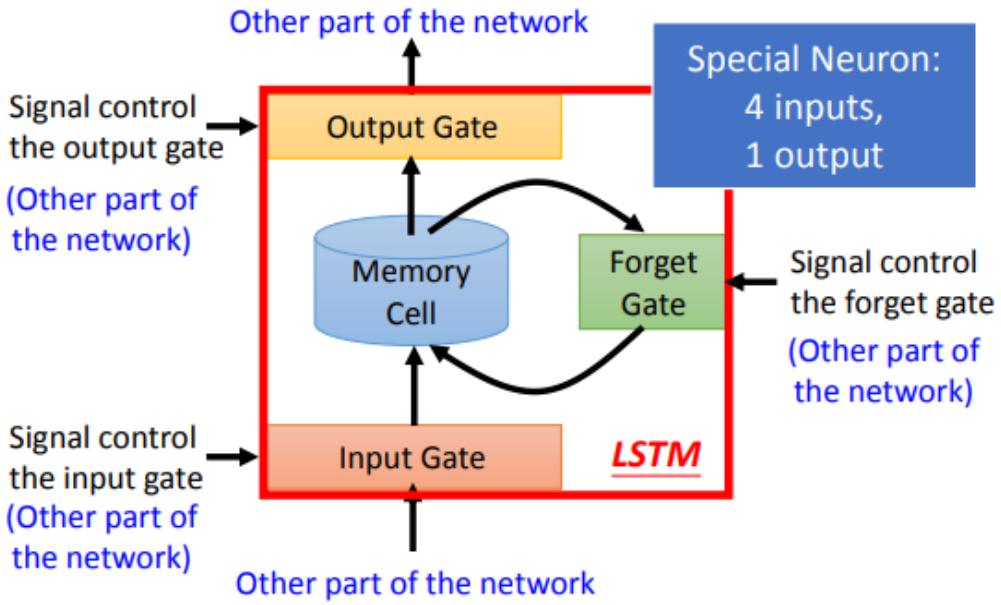

在前述 RNN 中 input 一個句子的話，它就是從句首一直讀到句尾。
假設句子裡面的每一個詞彙都用 $x^t$ 來表示的話，就是先讀 $x^t$ 再讀 $x^{t+1}$，最後再讀 $x^{t+2}$。

但其實它的讀取方向可以是反過來的，它可以先讀 $x^{t+2}$ 再讀 $x^{t+1}$，最後再讀 $x^t$。
所以可以同時 train 一個正向的 RNN 與一個逆向的 RNN，然後把這兩個 RNN 的 hidden layer 拿出來，都接給同一個 output layer，最後 $y^t$、$y^{t+1}$ 與 $y^{t+2}$。

用 Bidirectional RNN 的好處在於 network 在產生 output 的時候，它看的範圍是比較廣的。如果今天你只有正向的 network，在產生 $y^t$,跟 $y^{t+1}$ 的時候，你的 network 只看過 $x^1$ 一直到 $x^{t+1}$ 的部分。

但如果是 Bidirectional RNN，在產生 $y^{t+1}$ 的時候，你的 network 不只是看了 $x^1$ 一直到 $x^{t+1}$ 所有的 input，它也看了從句尾一直到 $x^{t+1}$ 的 input。
等同於 network 是看了整個 input 的 sequence，因此會比只看句子的一半，有更好的 performance。

## 四、Long Short-term Memory (LSTM)

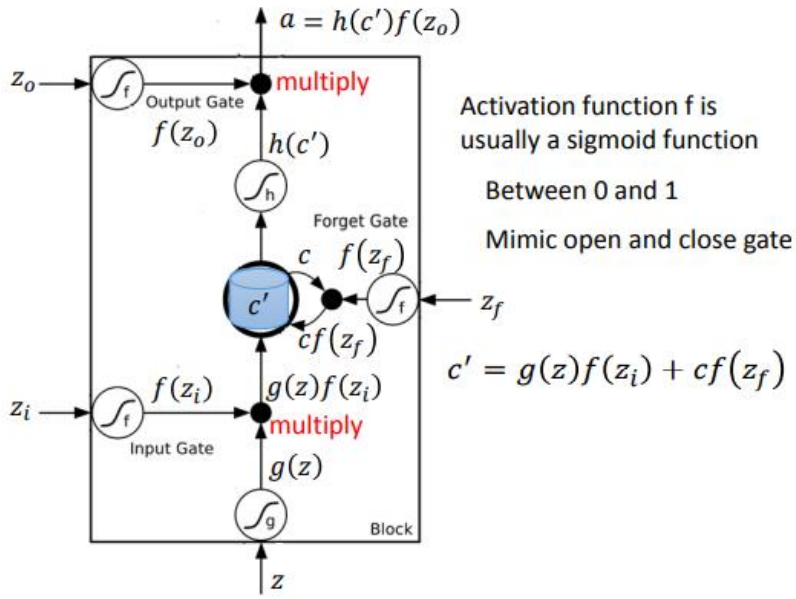



LSTM 有 3 個 gate。
當 neural network 的其他部分的 output 想要被寫到 memory cell 裡面的時候，它必須先通過一個 input gate。
當它要被打開的時候，才能夠把值寫到 memory cell 裡面。至於這個 input gate 打開還是關起來，是由 neural network 自己學的。

輸出的地方也有一個 output gate，決定其他的 neuron 可不可以從 memory 裡面把值讀出來。
當 output gate 為關閉的時候，就沒有辦法把值讀出來，同樣也是由 network 自己學習。

而第三個 gate 稱 forget gate。它決定什麼時候 memory 要把過去記得的東西 format 掉。

因此整個 LSTM 可以看成它有 4 個 input 與 1 個 output。
這 4 個 input 分別是：想要被存到 memory cell 裡面的值（仰賴 input gate 要不要讓這個 information 過去）、操控 input gate 的訊號、操控 output gate 的訊號以及操控 forget gate 的訊號。




更進一步來看它的架構的話，假設現在要被存到 cell 裡面的 input 叫做 z，操控 input gate 的 signal 叫做 $z_i$，操控 forget gate 的數值是 $z_f$，而 output gate 同樣有一個操控它的數值是 $z_o$。綜合這些東西以後，最後會得到一個 output，寫作 a。

接著假設現在 cell 裡面在有這 4 個輸入之前已經存了值 c。而 z 通過一個 activation function 得到 g(z)，然後把 $z_i$ 通過另外一個 activation function，得到 $f(z_i)$。

而這 3 個 $z_i$、$z_f$與 $z_o$他們通過的 activation function f，通常會選擇 sigmoid function，它的意義就是其值是介在 0~1 之間，代表了這個 gate 被打開的程度。如果這個 f 的 output 是 1，就代表這個 gate 是處於被打開的狀態。反之就是關閉的。

接下來就把 g(z)乘上 input gate 的值 $f(z_i)$，得到 $g(z) * f(z_i)$。那這個 forget gate 的 $z_f$通過這個 sigmoid activation function 亦得到 $f(z_f)$。
接著把存在 memory 裡面的值 c 乘上 $f(z_f)$，得到 c *　$f(z_f)$，再加上 $g(z) * f(z_i)$得到 c'，代表新的存在 memory 裡面的值。

根據到目前為止的運算可以發現，$f(z_i)$就是控制 g(z)可不可以輸入的一個關卡。因為假設 $f(z_i) = 0$，那 $g(z) * f(z_i)$就等於 0，就好像是沒有輸入一樣。
若 $f(z_i) = 1$，那就等於是直接把 g(z)當作輸入。

而同樣地，$f(z_f)$就是決定說要不要把存在 memory 裡面的值洗掉。
假設 $f(z_f) = 1$，也就是 forget gate 是被開啟時，這個時候 c 會直接通過，就等於是把之前存的值記憶下來。
那如果是 $f(z_f) = 0$，也就是 forget gate 被關閉的時候，0 乘上 c，過去存在 memory 裡面的值就會變成 0。

上述兩個值加起來，寫到 memory 裡面得到 c'，其通過 h 之後，得到 h(c')。

另一方面，output gate 受 $z_o$ 所操控，其通過 f 得到 $f(z_o)$。
若 $f(z_o) = 1$ 的話，我們會把 $f(z_o)$跟 h(c')乘起來，等同於是 h(c')通過 output gate。如果 $f(z_o) = 0$，則此處 output 會變成 0，代表存在 memory 的值沒有辦法通過 output gate 被讀取出來。

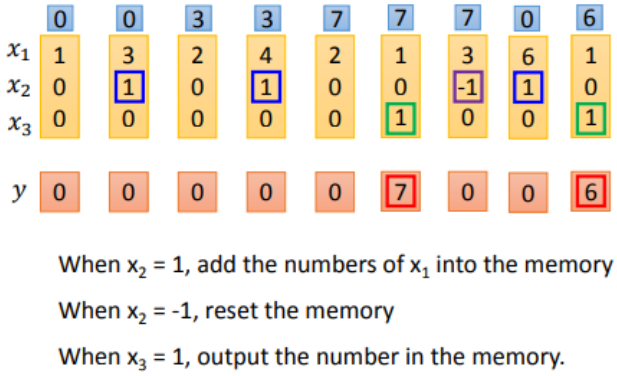
**1. Example**

假設在 network 裡面，只有一個 LSTM cell，且 input 都是三維的 vector；output 都是一維的 vector。

當第二個 dimension $x_2$ 的值是 1 的時候，$x_1$ 的值就會被寫到 memory 裡面（藍色方塊）。而 $x_2$ 的值是-1 的時候，memory 就會被 reset。
另一方面，假設 $x_3$ 等於 1 的時候，output gate 會打開，才能夠看到輸出。

所以假設原來存在 memory 裡面的值是 0，當 $x_2 = 1$ 的時候，3 會被存到 memory 裡面去。接著第四次 $x_2 = 1$ 時，$x_1 = 4$ 會被存到 memory 裡面，所以其值總合為 7。

第六次時，$x_3 = 1$，所以 7 會被輸出。
第七次的時候，$x_2 = -1$，則把 memory 裡面的值洗掉，因此看到-1 之後的下一個時間點，其 memory 的值就變成 0。

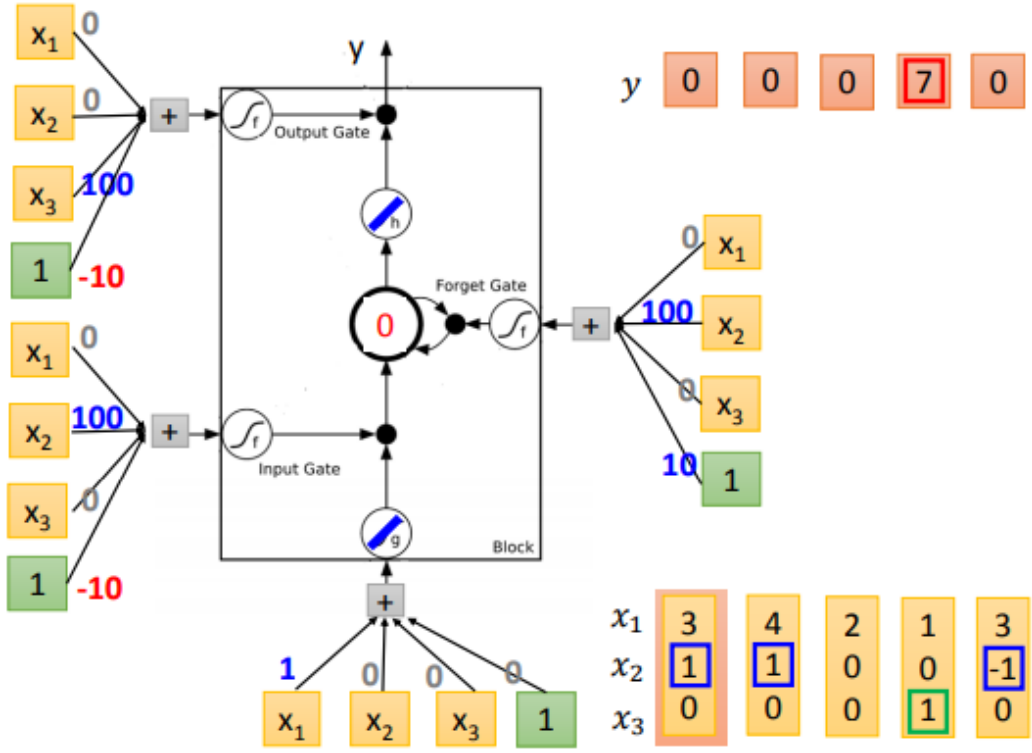



When $x_2 = 1$, add the numbers of $x_1$ into the memory

When $x_2 = -1$, reset the memory

When $x_3 = 1$, output the number in the memory.

我們知道 LSTM 的 memory cell 總共有 4 個 scalar input，為三維的 input vector 乘上 linear transform 以後得到的結果。也就是乘上三個 weight 再加上 bias 後，就得到它的 input。
而這些乘上的值與 bias，是透過 training data 去學到的。




舉上圖為例，在 network 的 input 中，$x_1$ 的 weight 為 1 其他都是 0，所以這邊直接把 $x_1$ 當作輸入。

在 input gate 的地方，它是 $x_2$ * 100 且 bias 為-10。也就是說，假設 $x_2$ 沒有值的時候，因為 bias 是-10，那通過 activation function 以後，它的值會接近 0，所以通常 input gate 是被關閉的。
只有在 $x_2$ 有值（假設為 1）的時候，它就會比 bias 的這個-10 還要大，此時 input 就會是很大的正值，代表 input gate 被打開。

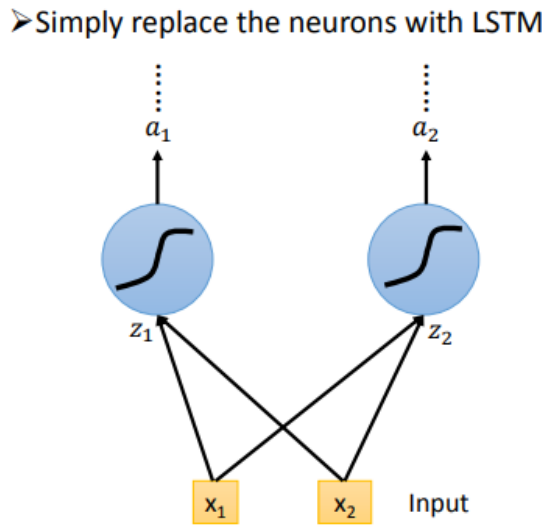
在 forget gate 部分，它平常都是被打開的（bias 為 10），所以會一直記得東西，只有在 $x_2$ 給它一個很大的負值的時候，會壓過這個 bias，才會把 forget gate 關起來。

output gate 平常也都是被關閉的，因為它的 bias 是很大的負值。但若 $x_3$ 有一個很大的正值的話，它就可以壓過 bias，把 output gate 打開。

（詳見講義 P.20 ~ P.25）

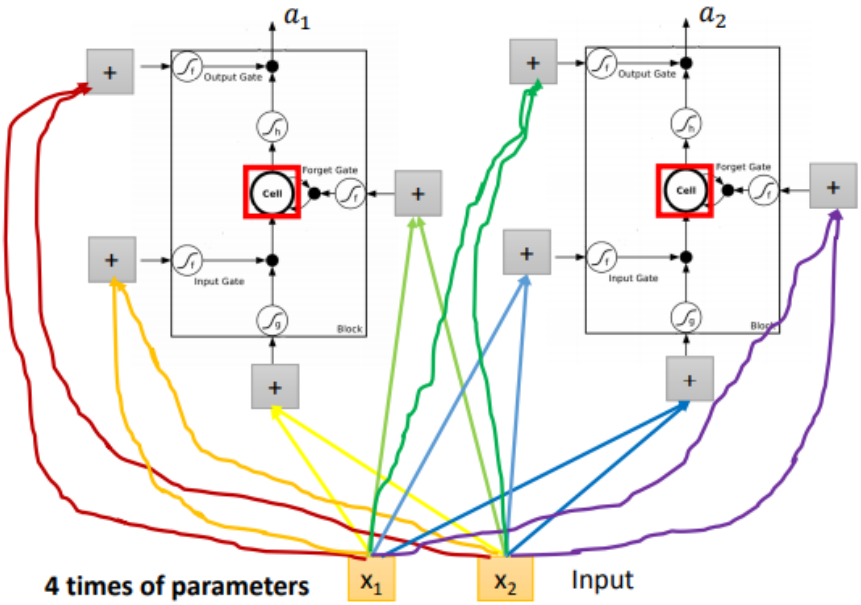
**2. Neural Network**

在原來的 neural network 中會有很多的 neuron，我們會把 input 乘上很多不同的 weight，當作是不同 neuron 的輸入。每一個 neuron 它都是一個 function，輸入一個 scalar，就會輸出另外一個 scalar。
對 LSTM 來說，可以把它的 memory cell 想成是一個 neuron。

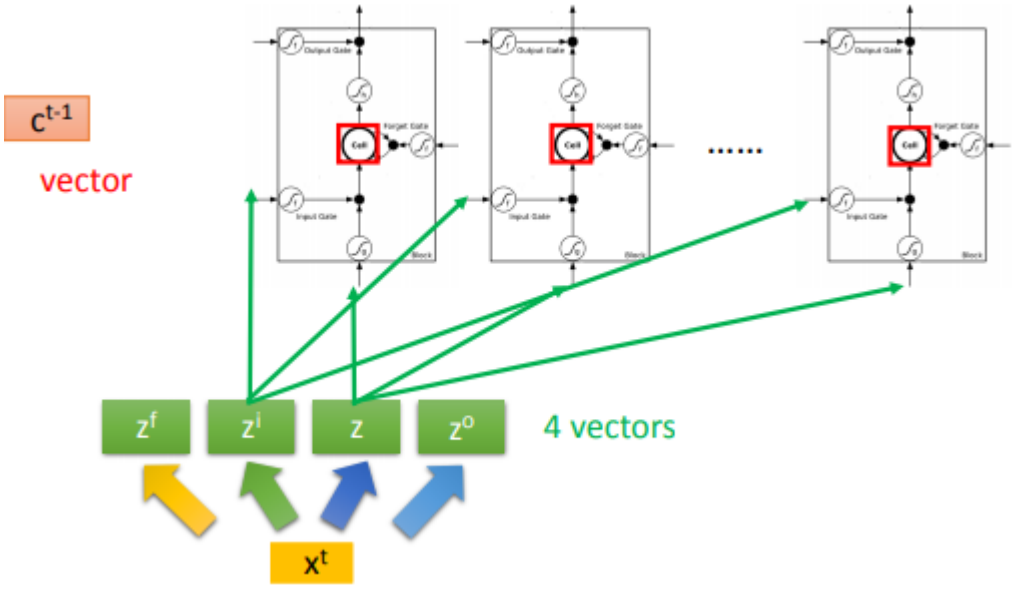



如果今天要用一個 LSTM 的 network，我們的事情就只是把原來一個簡單的 neuron，換成一個 LSTM 的 cell。而現在的 input $x_1$ 與 $x_2$ 它會乘上不同的 weight，當作 LSTM 不同的輸入。

假設只有兩 neuron，那 $x_1$ 與 $x_2$ 乘上某一組 weight 之後，會去操控第一個 LSTM 的 output gate；乘上另外一組 weight，操控第一個 LSTM 的 input gate。LSTM 的 input 與 forget gate 的 input 亦然。



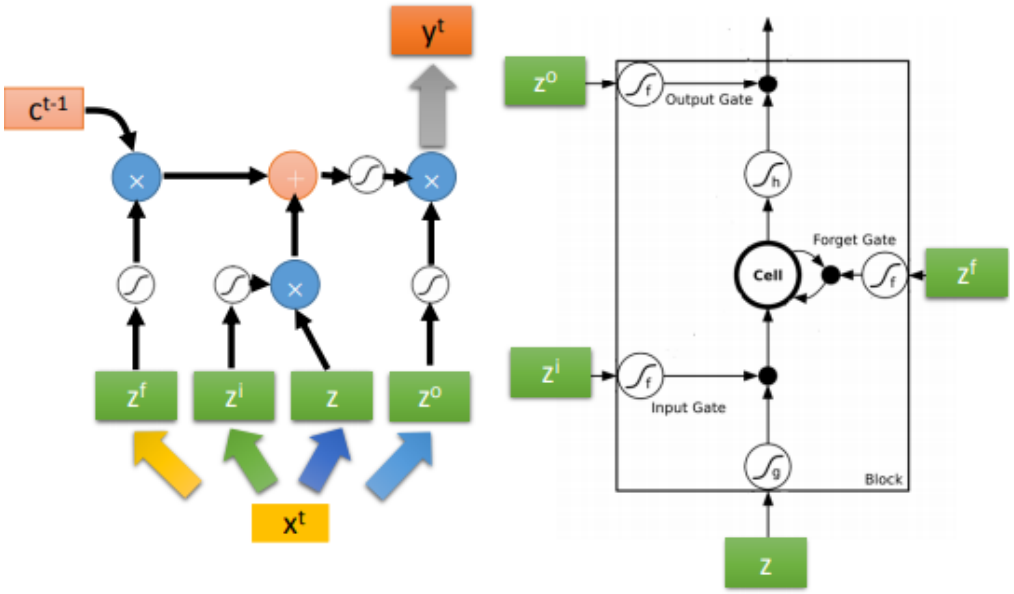
對一個 LSTM 來說，它有 4 個不一樣的 input。但在原來的 neural network 裡，一個 neuron 就是一個 input 以及一個 output；而在 LSTM 裡面它需要 4 個 input 才能產生一個 output。
就好比有一台機器，它只要插一個電源線就可以跑；那 LSTM 就要插 4 個電源線才能跑。




進一步說明之，假設現在有一整排的 LSTM，它們每一個人的 memory 都存了一個值。
若把所有的 scalar 接起來之後，就變成一個 vector，寫作 $c^{t-1}$。
也就是說，每一個 memory 裡面存的 scalar，就是代表這個 vector 裡面的一個 dimension。

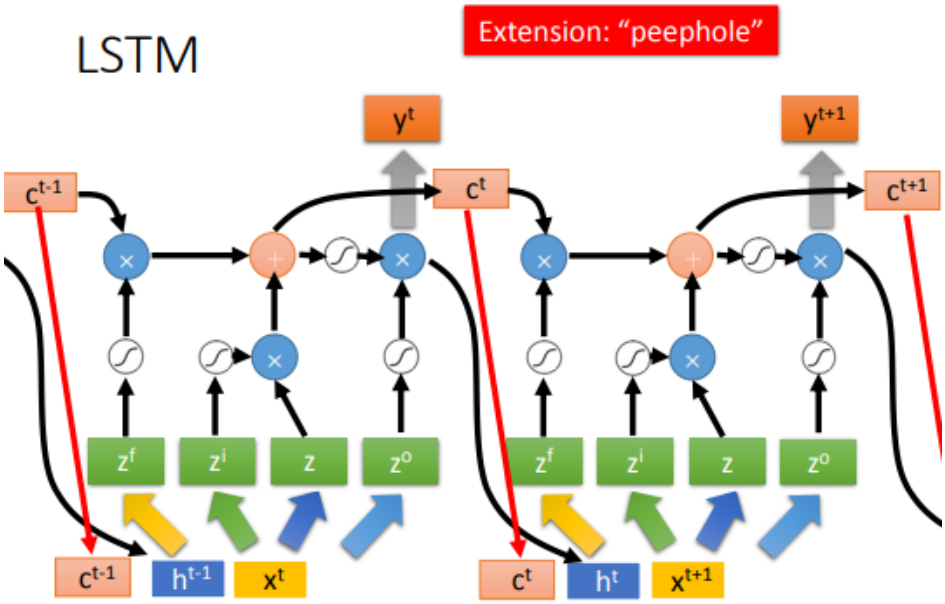
接著在時間點 t 輸入一個 vector $x^t$，這個 vector 會先進行 linear transform（乘上一個 matrix），最終得到另外一個 vector z。z 這個 vector 中每一個 dimension 就代表了操控每一個 LSTM 的 input，所以 z 的 dimension 就正好是 LSTM memory cell 的數目，。
因此 z 的第一維就丟給第一個 cell，第二維就丟給第二個 cell，以此類推。
$z^f$、$z^i$ 與 $z^o$ 同理。

結合 network 的觀念與 LSTM 的運算流程，可畫成下列架構：




而這些 memory cell 的運作是可以共同一起被運算的。
已知 input 的部分為，$z^i$ 通過 activation function 之後與 z 相乘（element-wise product）。同樣地，$z^f$ 通過 activation function 之後，跟已經存在於 cell 的值相乘。將上述兩個值加起來得到 $c^t$（$c^t = z * z^i + z^f * c^{t-1}$）。

而 output gate 部分，$z^o$ 通過 activation function 後，與 $c^t$ 通過 activation function 相乘得到 yt（$y^t = h(c^t) * f(z^o)$）。

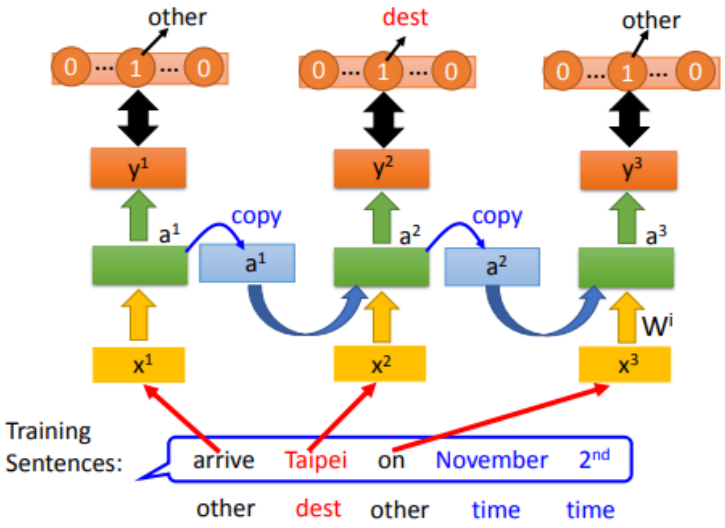



此外 LSTM 會把上一個時間點的 output 接進來，當作下一個時間點的 input。意即下一個時間點操控這些 gate 的值，不是只看那個時間點的 input x，也看前一個時間點的 output h。

而通常還會加上一個變數稱作「peephole」，用處在於把存在 memory cell 裡面的值也拉過來。也就是說，除了繼承給下一個時間點外，也作為其 input 考慮。所以在操縱 LSTM 的 4 個 gate 的時候，同時考慮了 x、h 與 c 這 3 個 vector。相乘後作為 linear transform 的 input，再去操控 LSTM。

## Learning

在做 learning 時，必須定一個 cost function 來評估 model 參數的好壞，意即選一個可以使 lost 最小的參數。



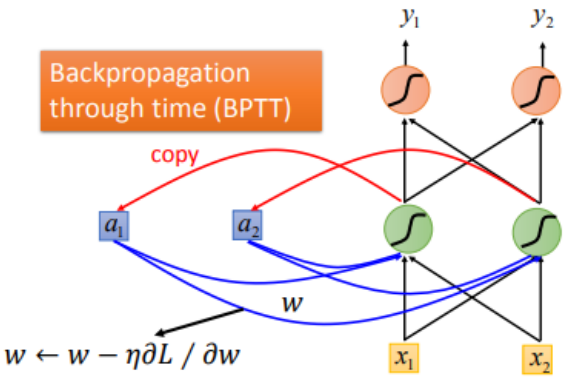

試舉例說明在 RNN 中如何定這個 lost。假設要處理的問題為 Slot Filling，那 training data 為 sentence，基於 sentence label 告訴 machine，「arrive」屬於「other」這個 slot；然後「Taipei」屬於「destination」這個 slot；而「November」與「$2^{nd}$」屬於「time」的 slot。

接下來我們會希望當「arrive」丟到 RNN 後得出的 output $y^1$，並對應到其中一項 reference vector 算它們的 cross entropy。

比如說該例有 40 個 slot，那這個 reference vector 的 dimension 就是 40。
而現在輸入的這個 word「arrive」應屬於「other」slot，那對應到「other」的那個 dimension 就是 1，其他則為 0。
因此當輸入 $x^1$（arrive）時，經過 RNN 後輸出的 $y^1$，它要跟 reference vector（other）距離越近越好。所以 RNN 的 output 跟 reference vector 的 cross entropy 和，就是需要 minimize 的對象。

但需要注意的是，你在丟 $x^2$ 之前，一定要先丟 $x^1$，否則無法知道存在 memory 裡面的值是多少。同樣地，在做 training 的時候，也不能把你 source 裡面的 word sequence 打散，而是要當作一個整體來看。

## 一、Training

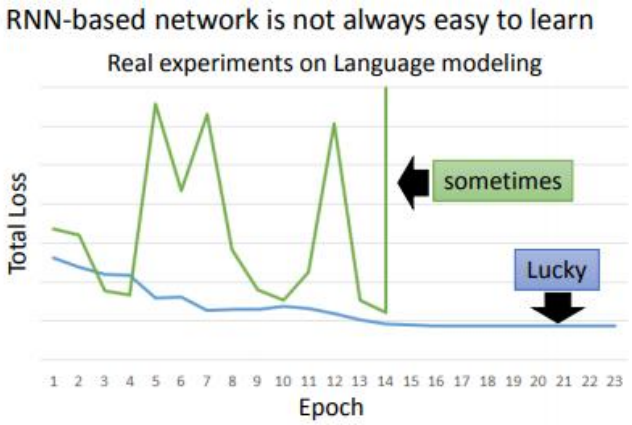



進行 training 時其實也是用 gradient decent，意即如果我們已經定出了 lost function L，當需要 update network 裡面的某一個參數 w 的時候，就去計算 w 對 L 的偏微分，把這個偏微分計算出來之後，就用 gradient decent 去 update 每一個參數。

在 feed forward network 使用 GD 的時候，要用一個比較有效的演算法稱 back propagation；而在 RNN 裡面，GD 的原理是一模一樣的，但是為了要計算方便，會使用 BPTT 演算法，意即在 GD 中考慮了時間的 information。

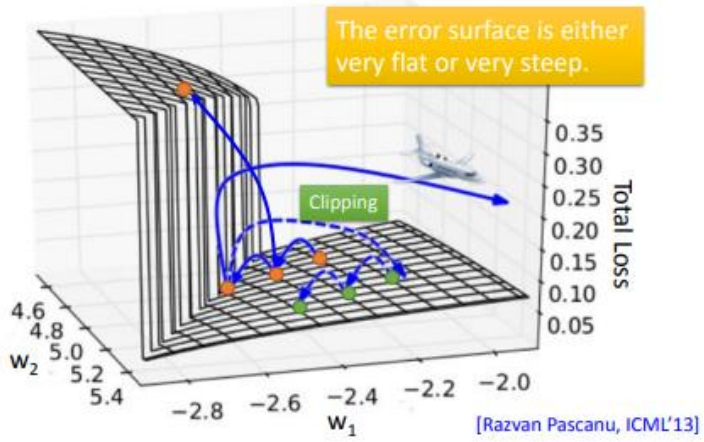
## 二、Difficulty

RNN 的 training 是比較困難的。一般而言，在做 training 時會希望 learning curve 為下圖藍線的趨勢。



這邊的縱軸為 total loss，橫軸則是 training epoch 的數目。一般情形下，希望說隨著 epoch 越來越多，參數不斷的被 update，loss 會慢慢下降最後趨向收斂。

然而在訓練 RNN 時，有時候會出現上圖綠線的趨勢。起因源自於 RNN 的 error surface（total loss 對參數的變化）。
RNN 的 error surface 非常崎嶇，意即該 error surface 有些地方非常平坦，但有一些地方，非常的陡峭。

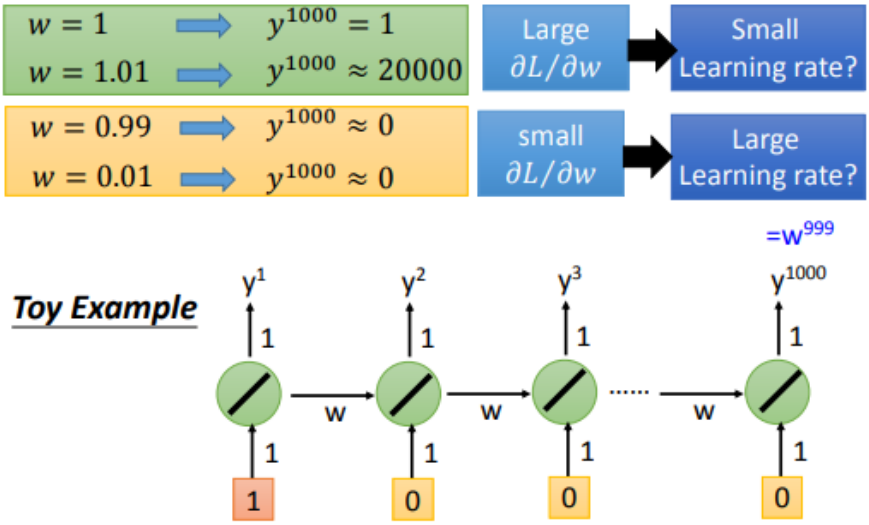



假設橙色的那個點為初始點，使用 GD 調整參數後會跳到下一個橙色的點。在平坦的地方時，可能會因為 gradient 都很小，所以 learning rate 調得比較大；但當位於懸崖邊界時，跳過一個懸崖之後 gradient 突然暴增，因此很大的 gradient 再乘上很大的 learning rate 使參數就 update 很多，有可能會得出 NaN 的結果。

## 三、Clipping

當 gradient 大於某一個 threshold 時候，就不要讓它超過那個 threshold。由於 gradient 不會太大，因此就算是踩在這個懸崖上參數就不會飛太遠，如此仍然可以繼續做 RNN 的 training。如上圖綠點所示。

而為何 RNN 會有如此特性，可以直觀的使用 gradient 的大小說明之。
也就是透過把某一個參數做小小的變化，看他對 network output 的變化有多大，並藉此測出這個參數的 gradient 大小。




舉一個簡單的 RNN 而例，假設該架構只有一個 linear neuron，它只有一個 input 沒有 bias 且 input 與 output weight 皆是 1；而 transition 部分的 weight 是 w，也就是從 memory 接到 neuron 的 input weight 是 w。

另外假設這個 network 的輸入是[1 0 0 0 0 0]，只有第一個時間點輸入 1，接下來都輸入 0。因此在最後一個時間點，第 1000 個時間點，的 output 值為 w 的 999 次方。
再假設 w 為我們要 learn 的參數，我們想要知道它 gradient 的大小，也就是當我們改變 w 的值的時候對 network output 有多大的影響。

w = 1 的時候，network 在最後時間點的 output ，$y^{1000}$，也是 1；
而 w = 1.01 的時候，$y^{1000}$ 是 1.01 的 999 次方，約等於 20000。
故知 w 有一點小小的變化時，對它的 output 影響是非常大的，所以 w 有很大的 gradient。此時會需要小一點的 learning rate。

而如果把 w 設成 0.99，$y^{1000}$ 就為 0；把 w 設為 0.01，那 $y^{1000}$ 還是等於 0。
這個時候又需要一個很大的 learning rate。

也就是說在 1 這個地方有很大的 gradient，但在 0.99 的地方 gradient 就突然變得非常小，使 error surface 變的很崎嶇（因為 gradient 是時大時小的）。

因此 RNN training 的問題，其實是來自於在 transition 的時候，它他把同樣的東西在時間和時間轉換的時候反覆使用。也就是從 memory 接到 neuron 的那一組 weight，在不同的時間點都是反覆被使用。因此這個 w 只要一有變化，它有可能完全沒有造成任何影響，也可能造成巨大的影響。

因應該問題現在廣泛被使用的技巧就是 LSTM，它可以讓 error surface 不要那麼崎嶇。它會把那些比較平坦的地方拿掉藉此解決 gradient vanishing，但無法解決 gradient explode。
也就是說有些地方仍然變化會是非常劇烈的，但是不會有特別平坦的地方，所以可以把 learning rate 設的小一點。

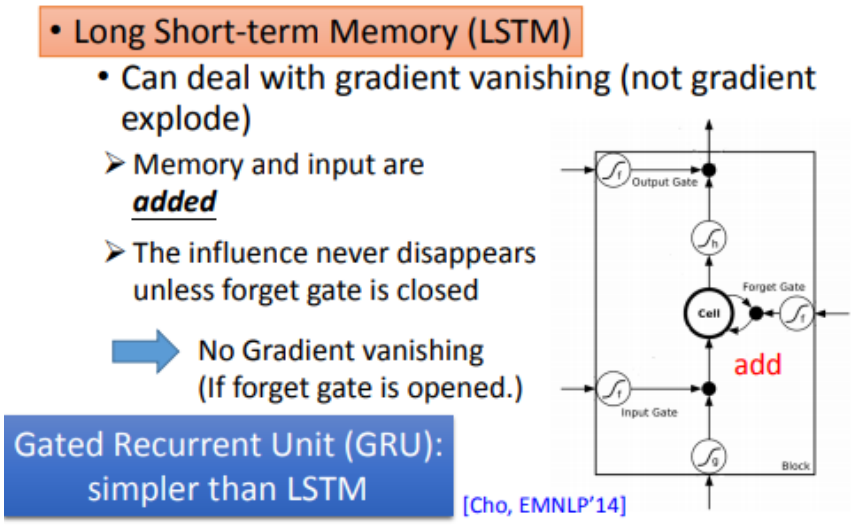
**1. LSTM VS. RNN**

至於 LSTM 為何可以做到解決 gradient vanish 的問題理由在於，RNN 跟 LSTM 它們在面對 memory 的時候，處理的方式不一樣。

RNN 在每一個時間點 memory 裡面的資訊都會被洗掉，neuron 的 output 都會被放到 memory 裡面去。

而在 LSTM 裡面不一樣，它是把原來的 memory 乘上一個值，再把 input 的值加起來放到 cell 裡面去，所以它的 memory 和 input 是相加的。

也就是說，再 LSTM 中，如果 weight 可以影響到 memory 的話，這個影響會永遠都存在，所以就不會有 gradient vanishing 的問題。
除非 forget gate 決定要把 memory 的值洗掉，然而一般在訓練 LSTM 時，不要給 forget gate 特別大的 bias，確保 forget gate 在多數的情況下是開啟的，只有少數情況會被 format 掉。



**2. Gated Recurrent Unit (GRU)**

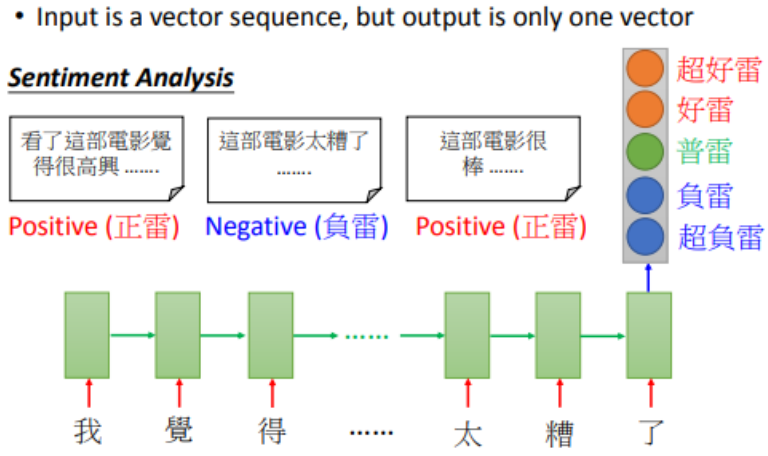
此外有另一個版本，使用 gate 操控 memory 的 cell，稱作 Gated Recurrent Unit。它只有兩個 gate，所以 GRU 相較於 LSTM 需要的參數量比較少。也因此 GRU 在 training 是比較 robust 的。因此在 train LSTM 的時候，發現 over fitting 的情況很嚴重，可以試用 GRU 改善之。

GRU 的精神簡而言之就是「舊的不去，新的不來」。它會把 input gate 跟 forget gate 連動起來，當 input gate 被打開的時候，forget gate 被打開，存在 memory 的值就會被洗掉；當 forget gate 沒有要 format 值時，input gate 就會被關起來。也就是你要把存在 memory 裡面的值清掉之後，才可以把新的值放進來。

# More Applications

## 一、Many to One: Sentiment Analysis

RNN 可以做到更複雜的事。例如它可以 input 一個 sequence，而 output 只是一個 vector。



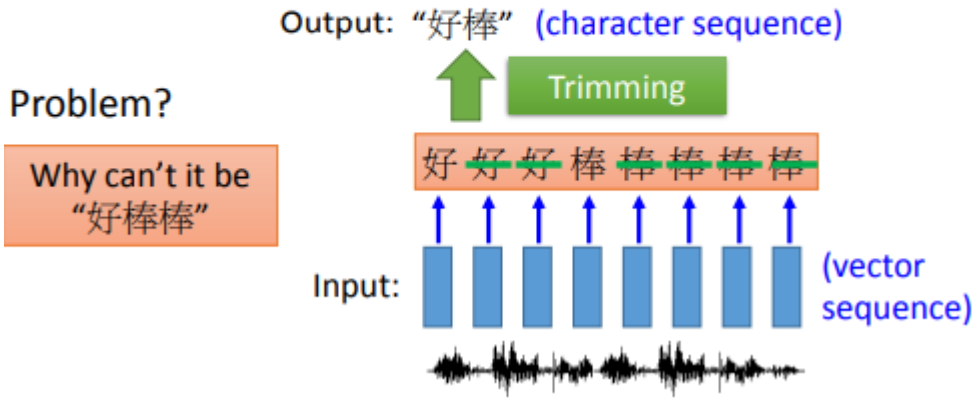

打個比方來說，某家公司想要知道他們的產品在網路上評價如何，他們可以寫一個爬蟲把它們產品有關係的網路文章都爬下來，接著使用 RNN 去分類這些文章哪是正向或是負向的。

它的 input 是一個 character sequence，然後在最後一個時間點把 hidden layer 拿出來，再通過幾個 transform 得到最後的 sentiment analysis 的 prediction。

## 二、Many to Many (Output is shorter)

該例的 input 與 output 都是 sequences，但 output sequence 比 input sequence 短。

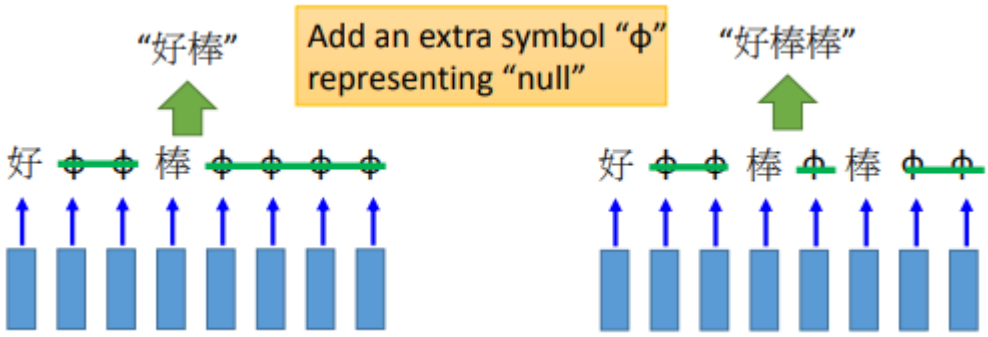
- Both input and output are both sequences, ***but the output is shorter.***
  - E.g. ***Speech Recognition***




以語音辨識為例，其 input 是一串 acoustic feature sequence。一般處理聲音訊號的方式為，在聲音訊號裡面每隔一小段時間，就把它用一個 vector 來表示，例如 0.01 秒。而它的 output 就是 character 的 sequence。

若是用原來的 RNN 把這一串 input 丟進去，它充其量只能做到每一個 vector 對應到哪一個 character。因此會發生 output 產生不必要的疊字（分割時間極短）。此處用 trimming 把重複的東西拿掉即可，但無法辨識實際上出現疊字的情形。

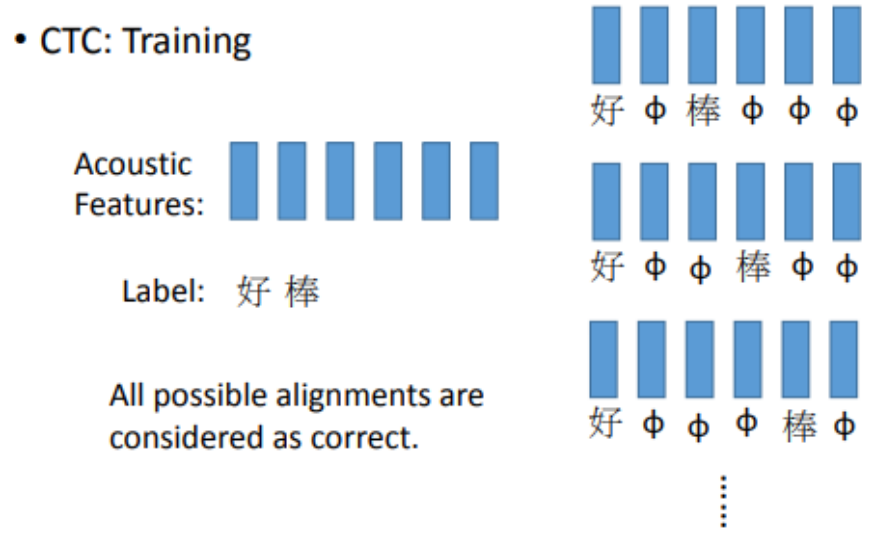
**Connectionist Temporal Classification (CTC)**




該方法可解決上述情形，在 output 的時候不只包含所有的 character，還多 output 一個符號，叫做「Null」。

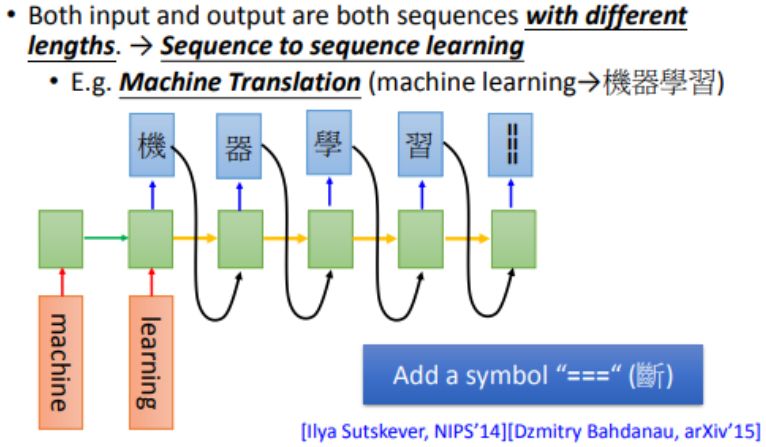
在做訓練的時候，training data 就會告訴你說，這一串 acoustic feature 會對應到某一串 character sequence。
但無法得知「好」是對應第幾個 frame 到第幾個 frame 或「棒」是對應到第幾個 frame。

解法為窮舉所有可能的 alignment。簡單來說假設所有的狀況都是可能的，在 training 的時候，就全部都當作正確的一起去 train，如下圖所示。




## 三、Many to Many (No Limitation)

該例子不確定 input 或 output 誰長誰短。比如說 machine translation，其 input 為英文的 word sequence，並要把它翻成中文的 character sequence。




現在假如 input 的是「machine learning」用 RNN 讀過去之後，在最後一個時間點 memory 就存了所有 input sequence 的 information，然後接下來讓 machine 吐一個 character。

如上圖所示，它吐的第一個 character 就是「機」，接著該 output 當作 input 把 memory 的值讀進來，它就會 output「器」。
如此循環下去後，若要阻止它繼續產生詞彙，就要多加一個 symbol 代表停止。

該方法的好處在於，直接 input 聲音訊號，然後 model 就得到辨識的結果，無須透過語音辨識，因此在 collect translation 的 training data 時會比較容易。

（實例繁多，另見講義 P.50 ~ P.79 與逐字稿）

## Deep & Structured

（補充單元，參見講義 P.80 ~ P.87 與逐字稿）