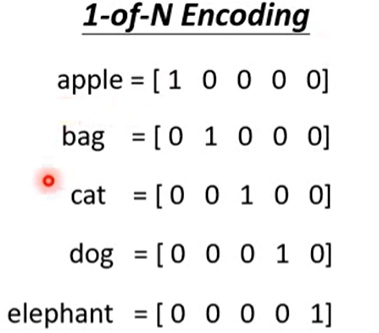
李弘毅 ELMO, BERT, GPT

1-of-N Encoding

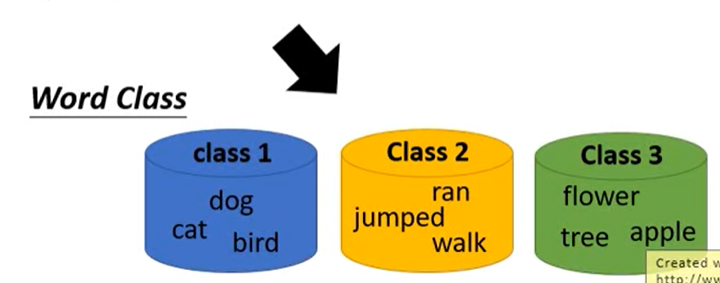


世界第一個讓機器讀文字的方式

缺點：難以讓電腦分辨出cat and dog are animals

Word Class

我們應該要分類，讓某些詞彙處於同一類別

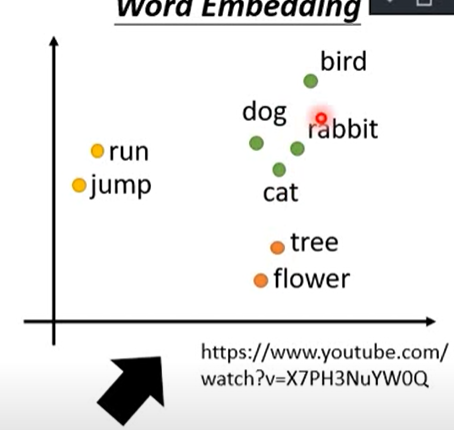


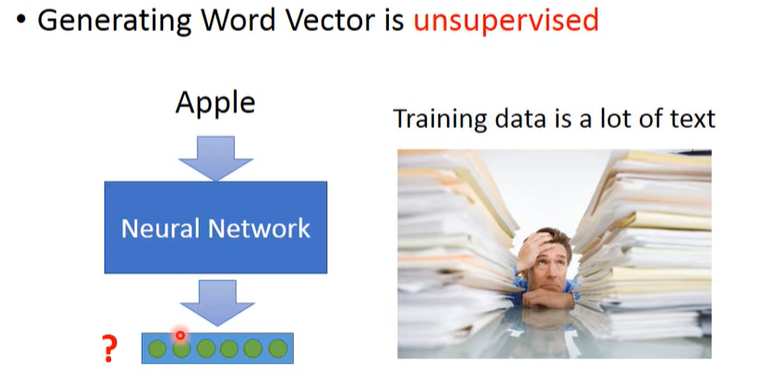
太粗糙，雖然都為動物，但哺乳類/鳥類之間的差別呢?

Word Embedding (50~100 dimension)

每一個維度都有其含義

unsupervised, machine read lots of context





將字彙用向量來表示，字彙相近的，向量在坐標軸上也比較相近

ex: dog, cat and rabbit -> 哺乳類；bird -> 鳥類

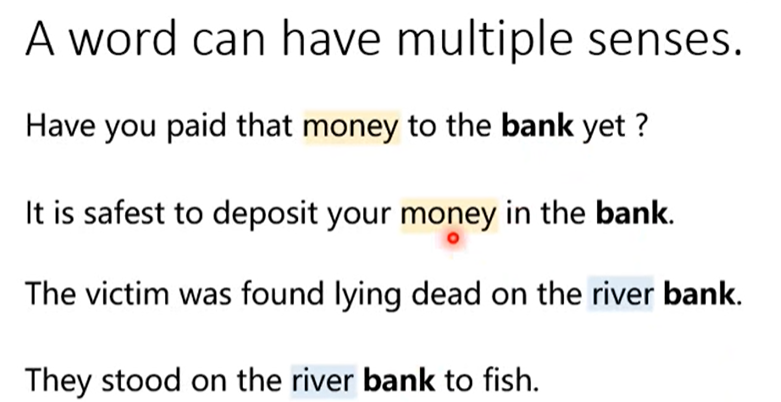
現在多為用word embedding表示一個詞彙



同一個詞彙可能有不同意思

ex: 4個bank是不同token，不同type，不同token有可能有不同的word embedding

1. 前2個句子的bank前面都有money這個詞彙 -> bank代表銀行
2. 後兩個前面有river -> bank表示河堤

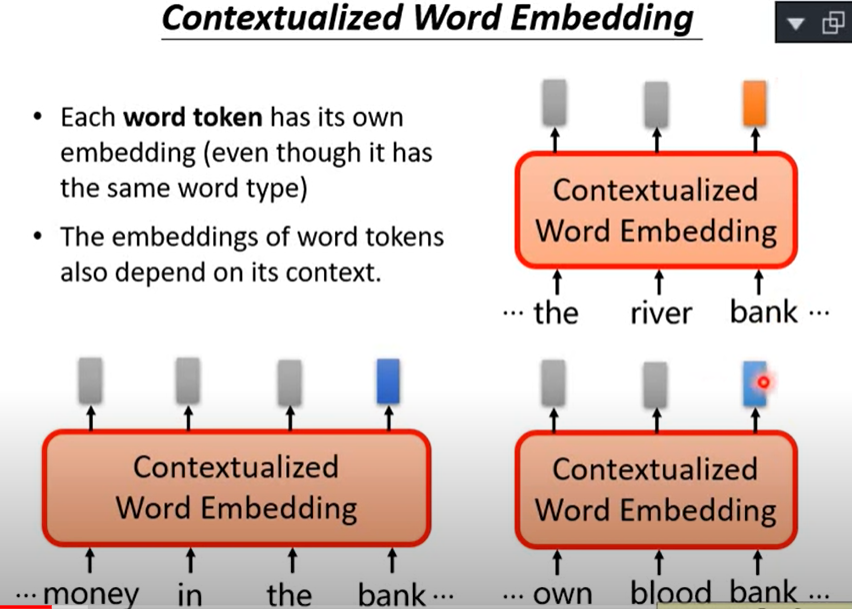




過去每一個type都有一個embedding，現在我們將不同token給不同embedding

過去是去查字典，bank這個type有2種意思，給2種embedding

-> 不夠，因為bank常有無限意思，語意太微妙



之前是一個type有一個or固定多個embedding

現在是每一個token都要有一個embedding

怎麼給呢? 看上下文，上下文愈接近的token

，有愈接近的embedding

each word token has its own embedding(even though it has

the same word type)

the embeddings of word tokens also depends on its context.

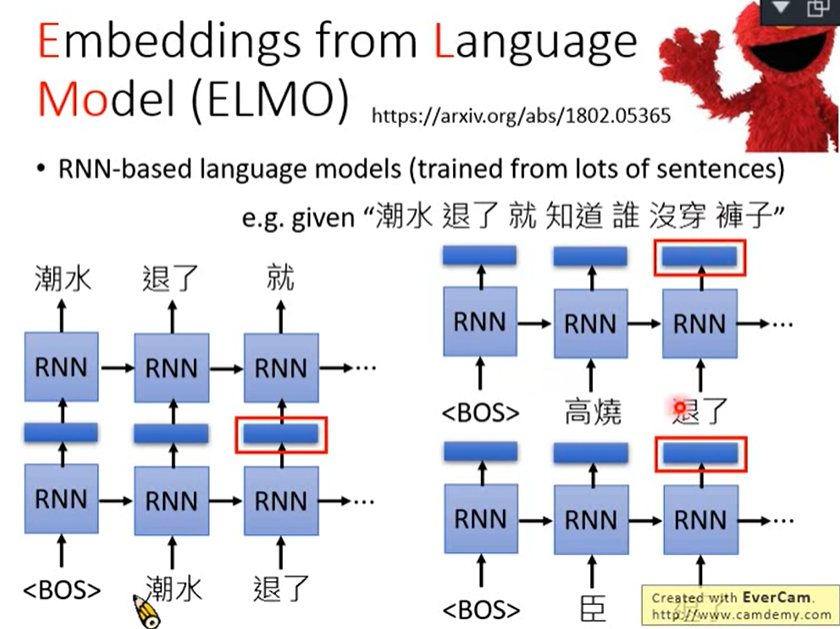
-> contextualized word embedding

下面2個bank可能有較接近的embedding，而上面那一個

bank的vector 可能離下面2個比較遠。

how to do it??? -> ELMO

RNN-based language model -> 給一堆句子，讓機器去學習如何預測下一個token會是什麼

ex: 潮水(begin with) -> 退了= 潮水退了 -> 就…

會得到embedding

ex: 讀了潮水退了 -> 得倒退了的embedding

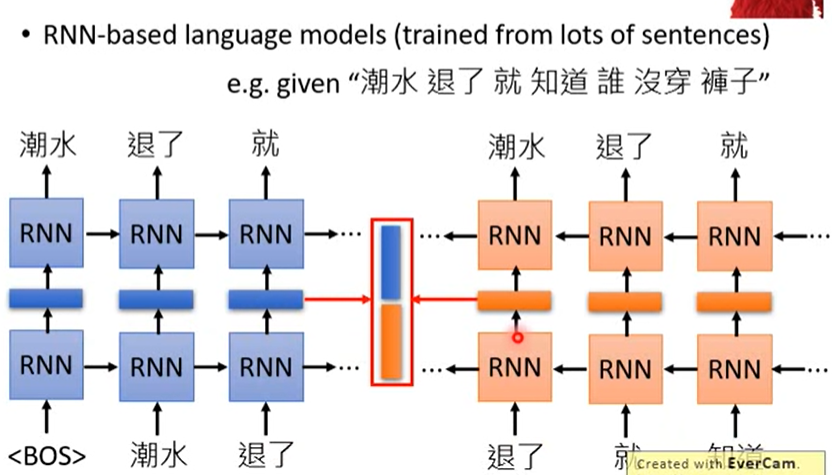
讀了 高燒退了-> 得到另一個token的embedding

讀了 臣退了 -> 又得到另一個token的embedding

好想只有考慮到前文??

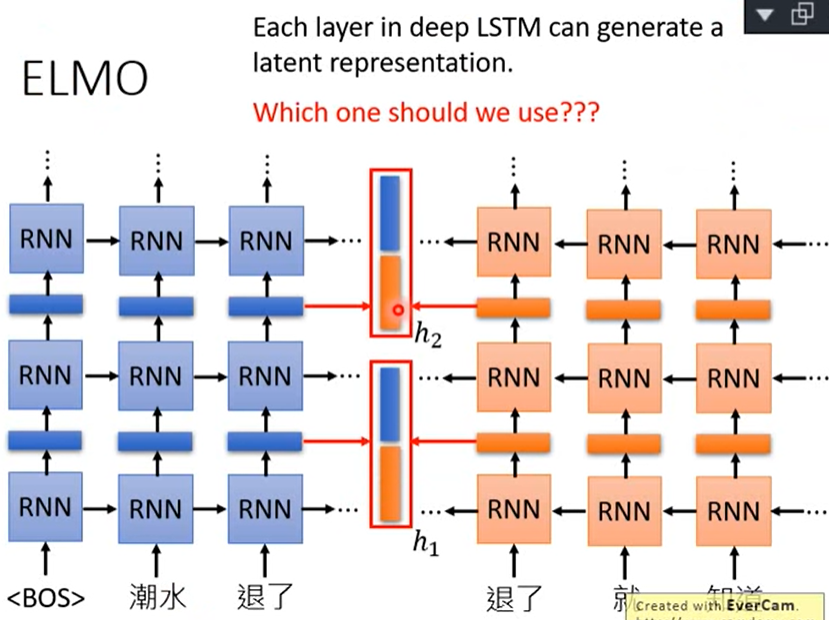
反向RNN，也考慮下文

將正向反向的embedding接起來

同一個詞彙的上下文不同，會有不同的embedding

Each layer in deep LSTM can generate a latent representation.

train deep，有很多層，同一個詞彙會有很多embedding(h1, h2, ……)，到底該用哪一層? -> 我全都要

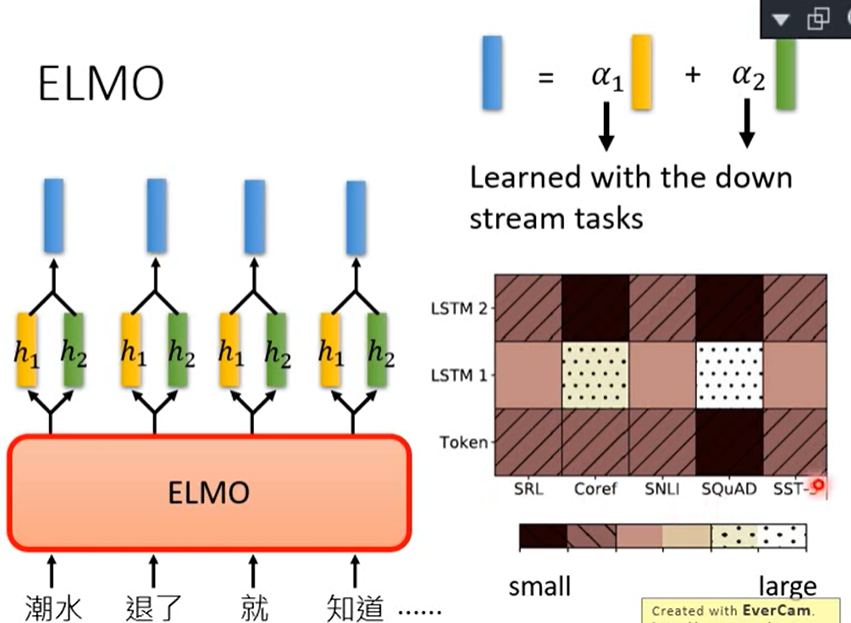


ex: RNN有2層 -> 吐出兩個embedding h1, h2 -> alpha1\*h1 + alpha2\*h2

用在接下來的down stream task的embedding有三個來源

token(原始沒有contextualized的embedding)、LSTM1(通過ELMO第一層的embedding)、LSTM2(第二層的embedding) -> 去做weighted sum 來得到藍色的那個embedding

以下圖例：不同的任務學出不同的weight



ex: Coref、SQuAD特別需要第一層的contextualized word embedding

BERT

先去看懂sequence2sequence, transformer等等

<https://leemeng.tw/neural-machine-translation-with-transformer-and-tensorflow2.html>

再看

<https://leemeng.tw/attack_on_bert_transfer_learning_in_nlp.html>

transformer的encoder其實就是BERT的network架構

原來要train一個transformer的時候，需要有一些task(ex: summarization, translation)，給transformer input並告訴他正確的output是甚麼

BERT只要train transformer裡面的encoder就好，且不需要有label資料，只要收集一大堆句子

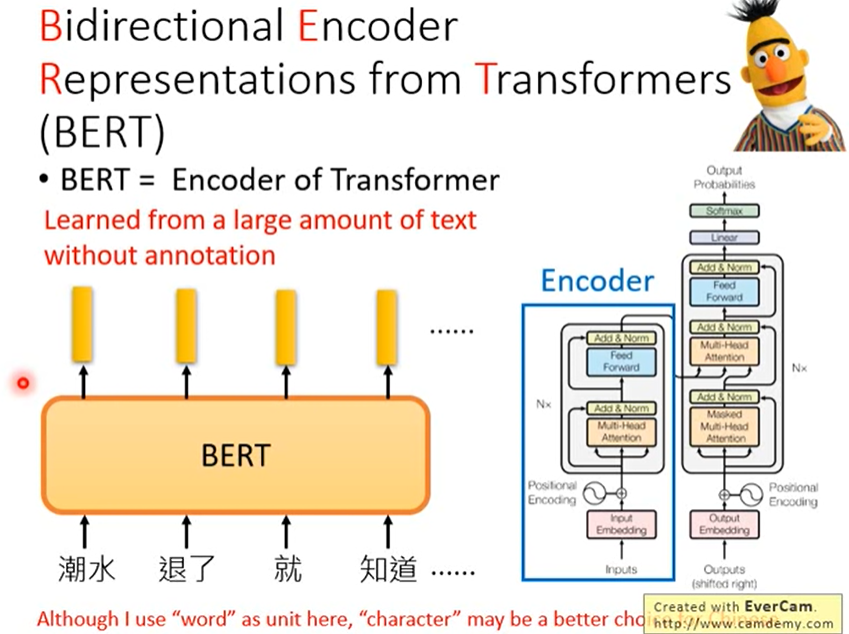
BERT是一個什麼東西?

給一個句子進去，每一個句子都會吐一個embedding給你

input word sequence output 一串embedding，某一個embedding對應到某一個input word

注意: 訓練中文詞彙，最好用字來當作單位

因為中文的詞是無限的，input one hot vector維度太大，而字的character是有限的，描述中文的character的one hot encoder的vector就不會太長



如何訓練BERT?????

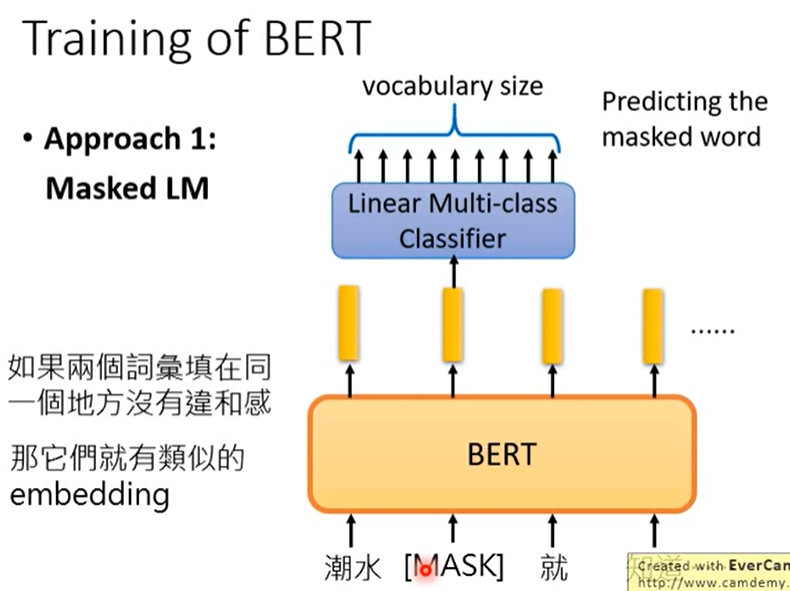
第一個訓練方法

Masked LM: 把所有的句子，15%的詞彙被置換成[MASK]

去猜測這些有蓋住的地方，到底是什麼詞彙(克漏字的意思)

將這些input token丟進去BERT -> 都會得到一個embedding

接下來將第二個(挖空的那個)embedding，丟到一個linear multi-class calssifier裡面，預測那個被MASK的詞彙是哪一個詞彙，因為是linear，預測能力非常弱，所以假如BERT有24或48層，那BERT這個model一定要剛好抽到一個非常好的representation(才預測的出來這個被MASK掉的辭彙是哪個)，可想而知，BERT到時候抽出來的representation(embedding)會是一個兩個詞彙填再同一個地方，沒有違和感的embedding(ex: 退了、落了)



第二個訓練方法

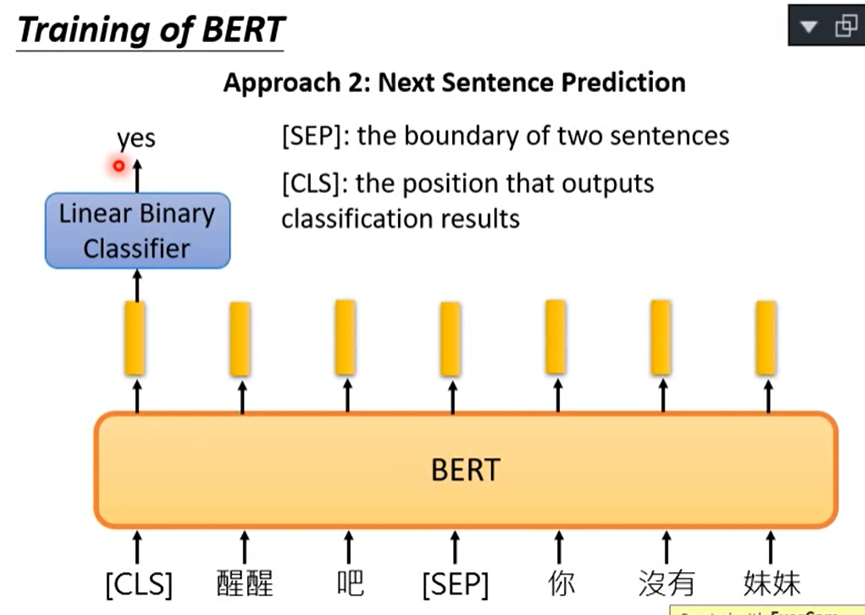
Next Sentence Prediction: 給他兩個句子，讓BERT去判斷這兩個句子，是接在一起的，還是不是接在一起的。

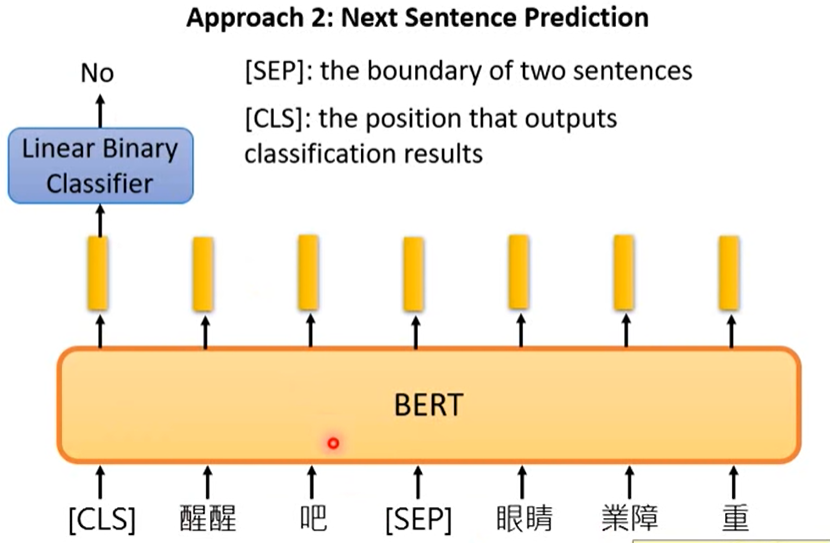
ex: 給BERT兩個句子：醒醒吧、你沒有妹妹，讓其判斷是否是接在一起的兩個句子。

[SEP] -> 告訴BERT兩個句子中間的交界在哪裡

句子的開頭放一個特殊的token: [CLS] -> 這個token輸出來的embedding丟到一個linear binary (yes/no) classification(是否接在一起)

為何在開頭? -> BERT內部是transformer，放在句首/尾沒差





方法一、方法二are used at the same time

現在每一個word(或character)都有他的contextualized word embedding了，那要如何用它呢??

例子一

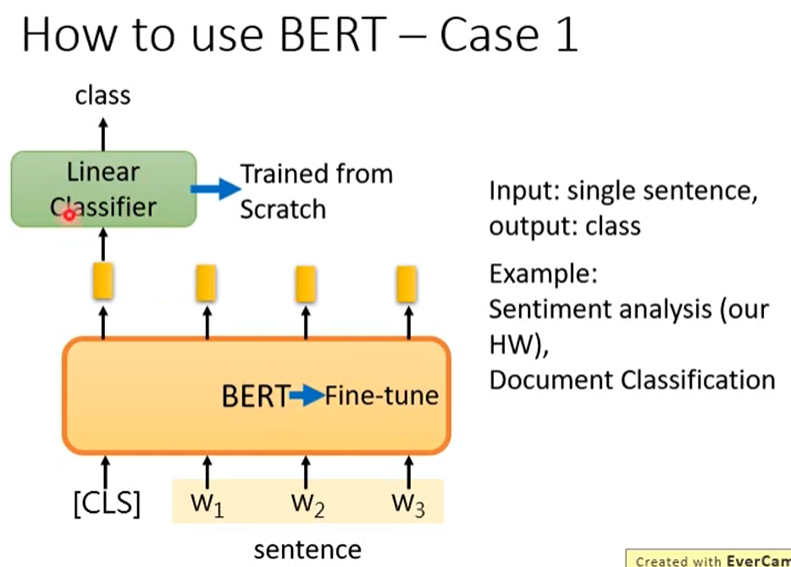
input sentence output classification

ex: 給一段句子，判斷他是正面還是反面

開頭給一個分類的token [CLS]、w1 w2 w3…..等sentence

CLS output的embedding丟到linear classifier，預測句子是正面還是反面

(需要從頭學的參數只有linear classifier，其他都已經train好了)

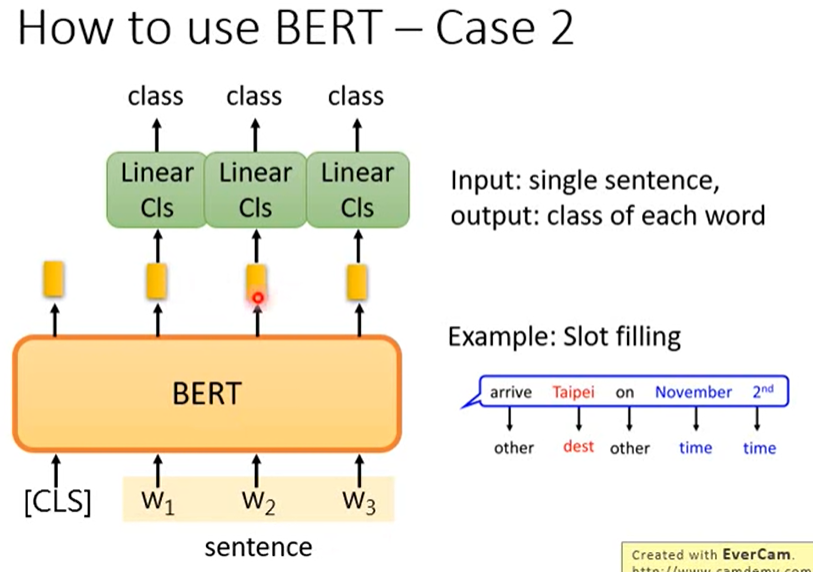


例子二

input single sentence output the class of each word

給定句子裡面的word要分類成甚麼output，讓各個linear classifier從頭開始學

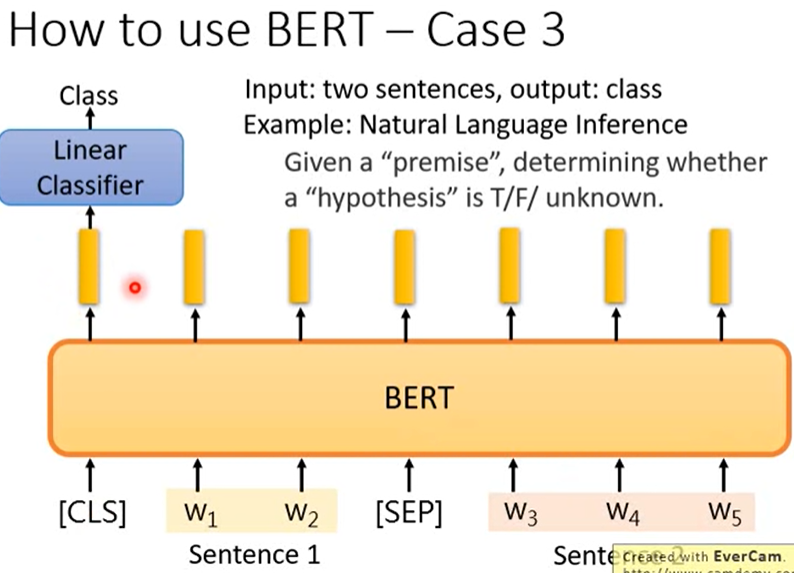
ex: arrive -> other；Taipei -> dest …….



例子三

input 2 sentences output class

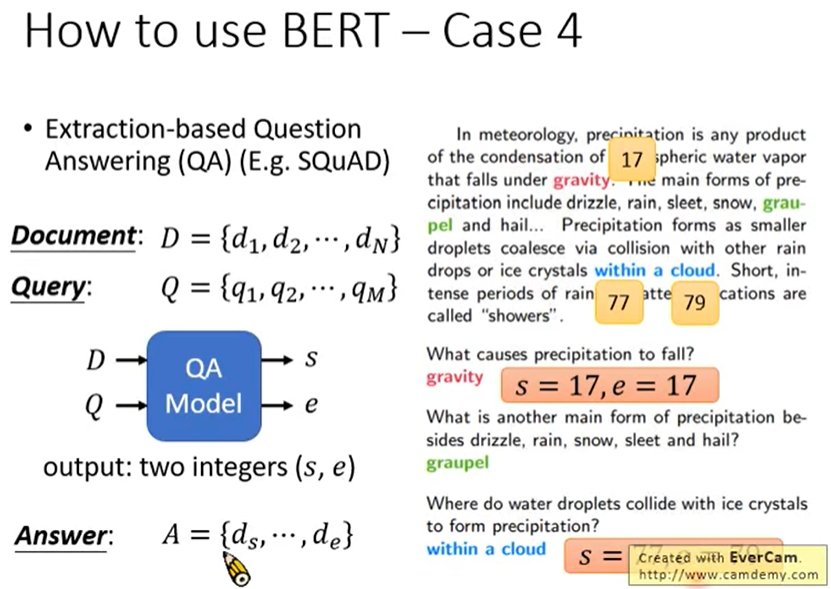
Natural Language Inference: 要機器學會透過這兩句話去推論。給機器一個前提(premise)，再給他一個假設(hypothesis)，問機器根據這個前提假設，到底是對/錯/不知道。



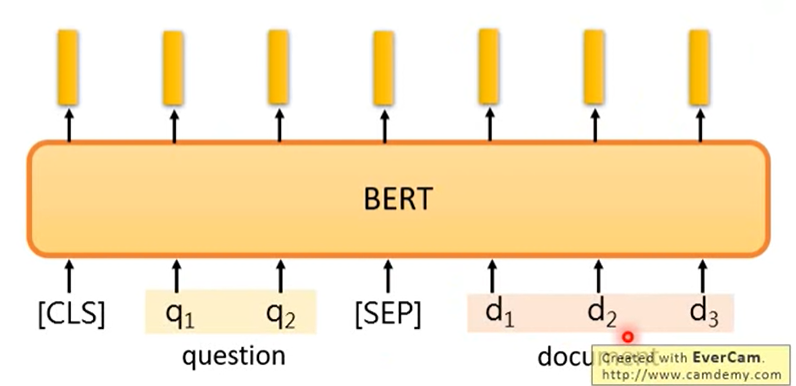
例子四

Extraction-based Question Answering

給BERT一篇文章，要BERT回答一段問題的答案，而這個答案一定出現在文章中。



question輸進去，給分隔符號[SEP]，再把文章輸進去

文章裡面每一個詞彙都會產生一個embedding

接下來讓machine去learn另外2個vector(紅藍)

這2個vector的維度跟黃色的vector一樣

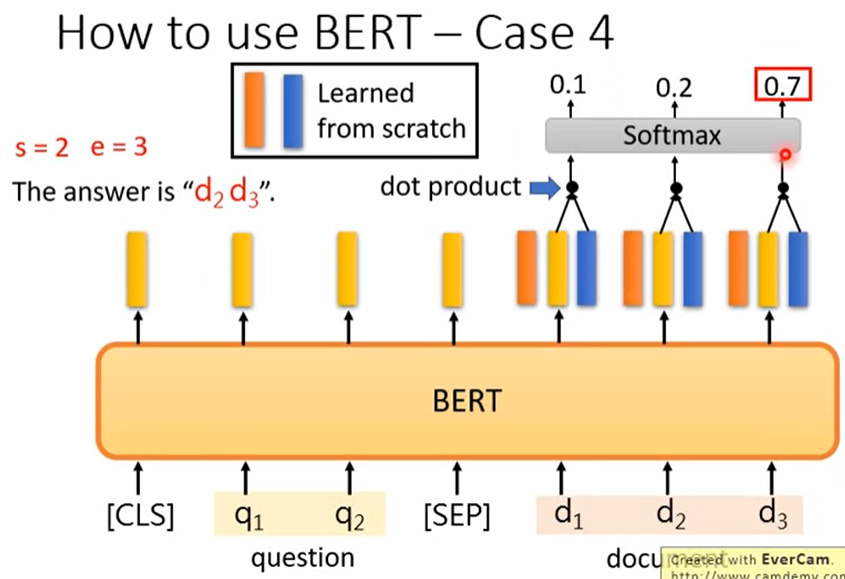
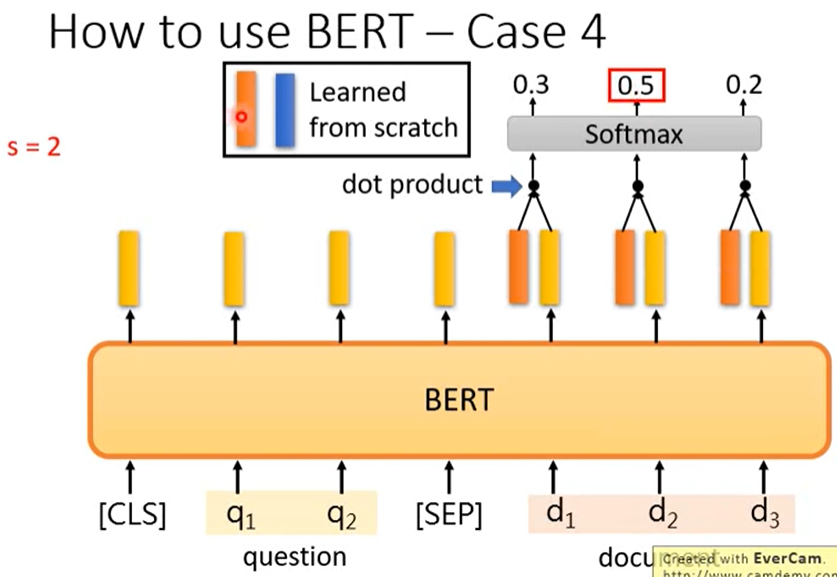
之後拿去跟每一個黃色的vector做dot product

都會算出一個scalar -> 算出softmax(類似分數)

取最高分的

紅色的vector決定s等於多少

藍色的vector決定e等於多少



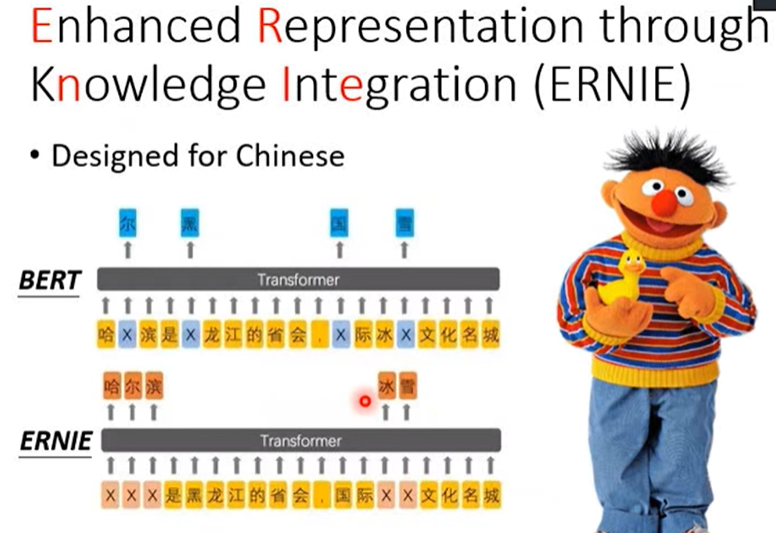
如果今天output的答案，e落在s前面(ex: s=3, e=2) -> 沒有答案(此題無解也是個答案)

注意: A = {ds, ……., de}，s應該落在e前面!!!

ERNIE

特別為了中文而設計的模型(BERT的好朋友)

BERT在做Masked LM時，如果是中文句子，只蓋掉一個字非常容易猜出來，因此改為蓋掉多個字。



BERT每一個層在做什麼?

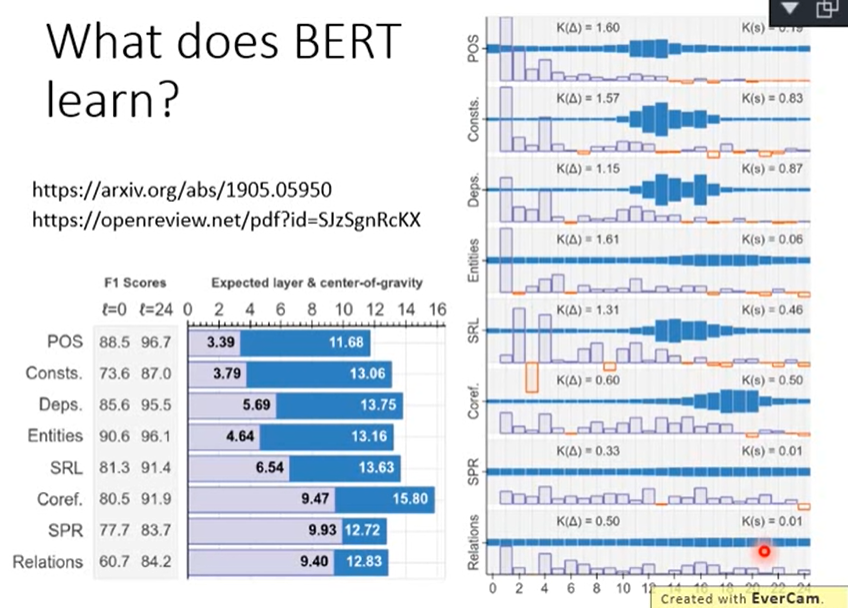
ex: 比較接近input的層，可能做一些簡單的文法的東西，接近output則比較複雜

把BERT裡面的24層的vector(24個)抽出來，去做weighed sum(like ELMO)

看每一個NLP的任務任的weight怎麼樣，就可以看出來這個任務特別需要BERT的那些層

任務舉例: POS packing 詞性標記，去標記每一個詞彙是甚麼詞性

最需要BERT第11~13層

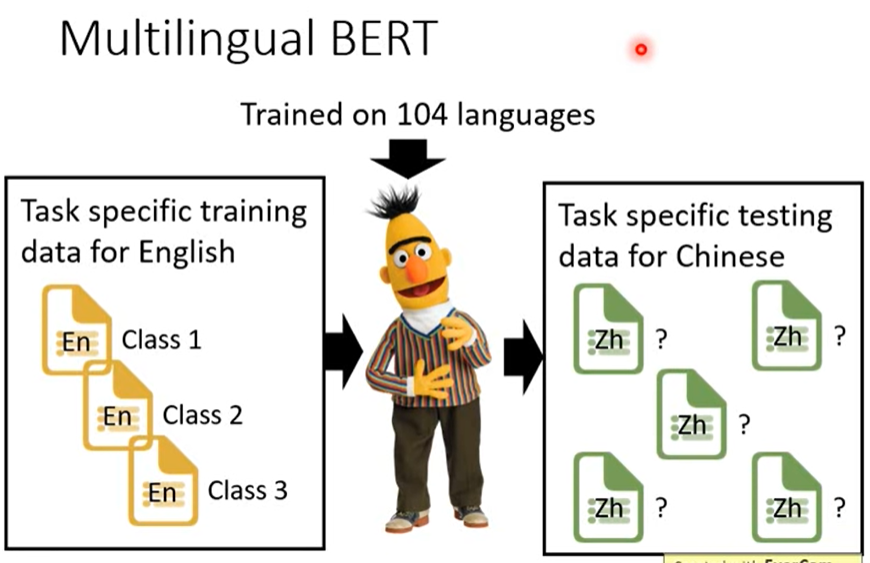


Multilingual BERT

trained on 104 language

讀過104種語言，自動學到了不同語言之間的對應關係

ex: 給BERT英文的文章分類，BERT也自動學會去做中文文章的分類!!!



<https://leemeng.tw/shortest-path-to-the-nlp-world-a-gentle-guide-of-natural-language-processing-and-deep-learning-for-everyone.html>

1. 進入NLP世界的最佳橋樑 入門指南

<https://leemeng.tw/neural-machine-translation-with-transformer-and-tensorflow2.html>

1. 淺談神經機器翻譯&用Transformer與TensorFlow2英翻中

<https://leemeng.tw/attack_on_bert_transfer_learning_in_nlp.html>

1. 進擊的BERT

入門指南

資料前處理

1. 文本分詞

文本分詞（Text Segmentation）是一個將一連串文字切割成多個有意義的單位的步驟。這單位可以是：**character, word, sentence**

Ex: 英文通常只要依照空白分割，就能得到一個有意義的詞彙列表了

中文無法這樣做。這時候我們將藉助 **Jieba** 這個中文斷詞工具，來為一連串的文字做有意義的切割Jieba 將我們的中文文本切成有意義的詞彙列表，並為每個詞彙附上對應的詞性（Flag）。

假設我們不需要標點符號，則只要將 flag == x 的詞彙去除即可。

另外值得一提的是，不管最後是使用哪種切法，切完之後的每個文字片段在 NLP 領域裡頭習慣上會被稱之為 Token。（如上例中的警方、GDP）

1. 建立字典並將文本轉成數字序列

當我們將完整的新聞標題切成一個個有意義的詞彙（Token）以後，下一步就是將這些詞彙轉換成一個數字序列，方便電腦處理。

這些數字是所謂的索引（Index），分別對應到特定的詞彙。

在這個簡單的狐狸例子裡頭，word\_index 就是我們的字典；我們利用該字典，**將 1 句話轉成包含多個數字的序列**，而每個數字實際上代表著一個 Token。

同理，我們可以分 4 個步驟將手上的新聞標題全部轉為數字序列：

將已被斷詞的新聞標題 A 以及新聞標題 B 全部倒在一起

建立一個空字典

查看所有新聞標題，**裏頭每出現一個字典裡頭沒有的詞彙，就為該詞彙指定一個字典裡頭還沒出現的索引數字，並將該詞彙放入字典**

利用建好的字典，將每個新聞標題裡頭包含的詞彙轉換成數字

有套件!!!!!

這種文字前處理因為出現頻率實在太過頻繁，Keras有專門的文字前處理模組來提升我們的效率

import keras

MAX\_NUM\_WORDS = 10000

tokenizer = keras \

.preprocessing \

.text \

.Tokenizer(num\_words=MAX\_NUM\_WORDS)

Tokenizer 顧名思義，即是將一段文字轉換成一系列的詞彙（Tokens），並為其建立字典。這邊的 num\_words=10000 代表我們限制字典只能包含 10,000 個詞彙，一旦字典達到這個大小以後，剩餘的新詞彙都會被視為 Unknown，以避免字典過於龐大。

詳見NLP\_intro.inpy檔

1. 序列的Zero Padding

我們已經將每個新聞標題的文本轉為一行行的數字序列，你會發現每篇標題的序列長度並不相同

而為了方便之後的 NLP 模型處理（見循環神經網路一章），一般會設定一個MAX\_SEQUENCE\_LENGTH 來讓所有序列的長度一致。

長度超過此數字的序列尾巴會被刪掉；而針對原來長度不足的序列，我們則會在詞彙前面補零。Keras 一樣有個方便函式 pad\_sequences 來幫助我們完成這件工作：

一般來說 MAX\_SEQUENCE\_LENGTH 可以設定成最長序列的長度（此例中的 61）。但這邊為了讓模型可以只看前 20 個詞彙就做出判斷以節省訓練時間，我們先暫時使用 20 這個數字。

1. 將正解做One-hot Encoding

到目前為止，我們已經將所有的新聞標題以數字型態表示，只剩分類欄位 label 要進行從文本到數字的轉換了

不過 label 的處理相對簡單。跟新聞標題相同，我們一樣需要一個字典將分類的文字轉換成索引

現在每個分類的文字標籤都已經被轉成對應的數字，接著讓我們利用 Keras 做 One Hot Encoding

用這樣的方式表達 label 的好處是我們可以把分類結果想成機率分佈。[1, 0, 0] 就代表一組新聞標題 A、B 為 unrelated 的機率等於 100 %。

在決定如何衡量模型的表現一節我們會看到，給定一組新聞標題 A、B，我們的模型會預測此成對標題屬於每個分類的機率值，比方說 [0.7, 0.2, 0.1]。而此預測結果代表模型認為這 2 個新聞標題的關係有 70 % 的機率為 unrelated、20 % 的機率是 agreed 而 10 % 為 disagreed。

因此，如果正解也事先用同樣的方式表達的話，會讓我們比較好計算以下兩者之間的差距：

正確的分類的機率分佈（[1, 0, 0]）

模型預測出的機率分佈（[0.7, 0.2, 0.1]）

在知道預測結果跟正確解答之間差距多少之後，深度學習模型就會自動修正學習方向，想盡辦法拉近這個差距。

切割訓練資料集 & 驗證資料集

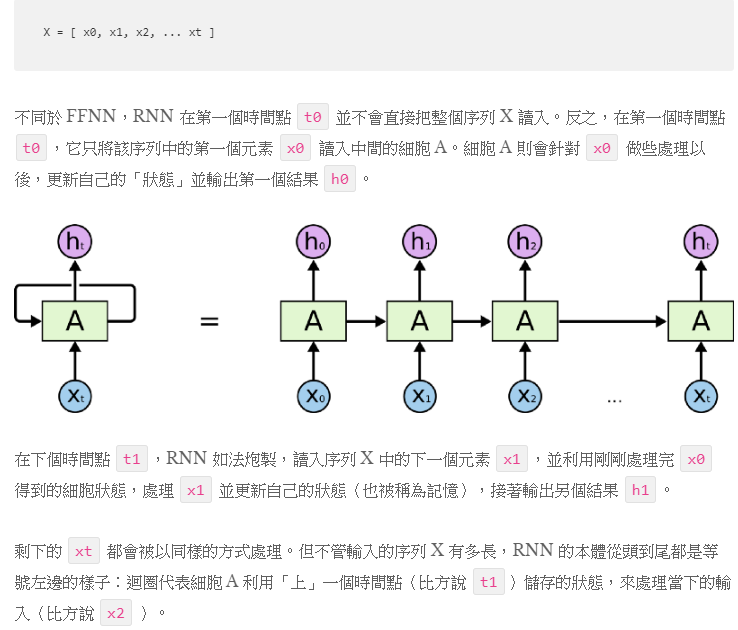
要切訓練資料集 / 驗證資料集，scikit-learn 中的 train\_test\_split 函式是一個不錯的選擇



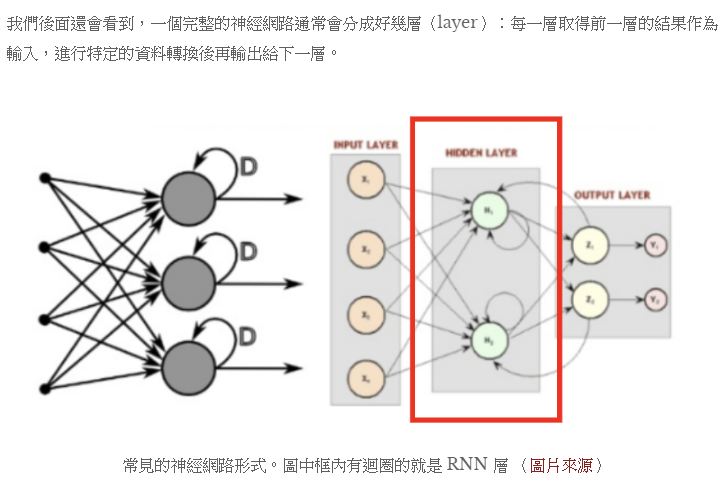
每當下個詞彙映入眼中，你腦中的處理都會跟以下兩者相關：

前面所有已讀的詞彙

目前腦中的記憶狀態

當然，實際人腦的閱讀機制更為複雜，但 RNN 抓到這個處理精髓，利用內在迴圈以及細胞內的「記憶狀態」來處理序列資料。





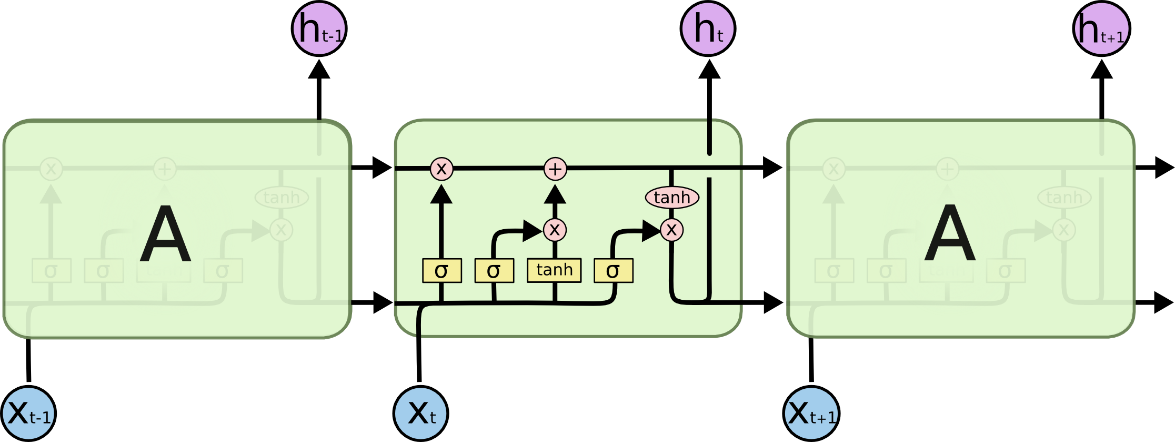
在了解 RNN 的基本運作方式以後，你會發現 RNN 真正的魔法，事實上藏在細胞 A 的 **f 函式**裏頭。

要如何將細胞 A 當下的記憶 state\_t 與輸入 input\_t 結合，才能產生最有意義的輸出 output\_t 呢？

在 SimpleRNN 的細胞 A 裡頭，這個 f 的實作很簡單。而這導致其記憶狀態 state\_t 沒辦法很好地「記住」前面處理過的序列元素，造成 RNN 在處理後來的元素時，就已經把前面重要的資訊給忘記了。

長短期記憶（Long Short-Term Memory, 後簡稱 LSTM）就是被設計來解決 RNN 的這個問題。

如下圖所示，你可以把 LSTM 想成是 RNN 中用來**實現細胞 A 內部處理邏輯**的一個特定方法：



Source: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>

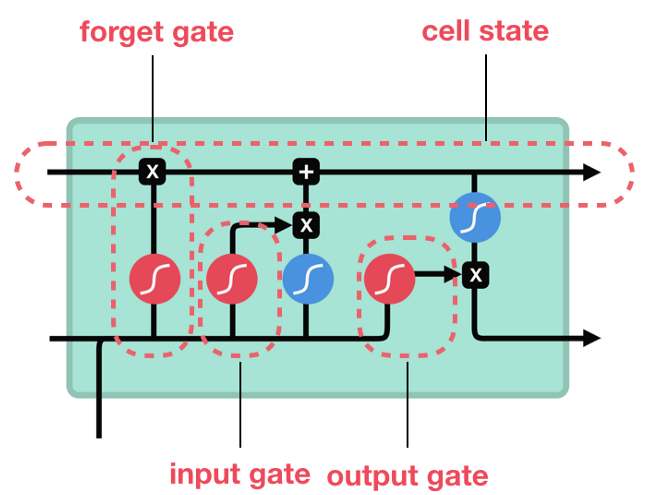
基本上一個 LSTM 細胞裡頭會有 3 個閘門（Gates）來控制細胞在不同時間點的記憶狀態：

**Forget Gate**：決定細胞是否要遺忘目前的記憶狀態

**Input Gate**：決定目前輸入有沒有重要到值得處理

**Output Gate**：決定更新後的記憶狀態有多少要輸出

透過這些閘門控管機制，LSTM 可以將很久以前的記憶狀態儲存下來，在需要的時候再次拿出來使用。值得一提的是，這些閘門的參數也都是神經網路自己訓練出來的。



Source: <https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21>

LSTM 細胞頂端那條 cell state 正代表著細胞記憶的轉換過程。

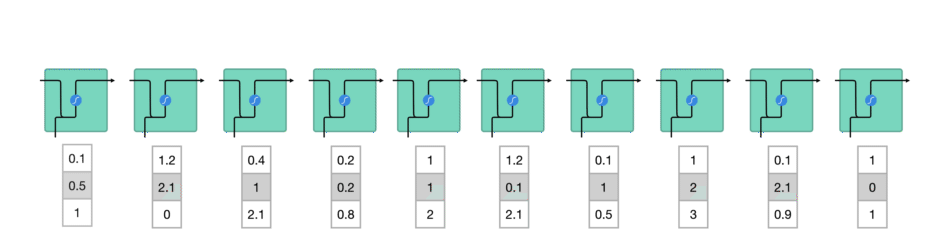
想像 LSTM 細胞裡頭的記憶狀態是一個包裹，上面那條直線就代表著一個輸送帶。

LSTM 可以把任意時間點的記憶狀態（包裹）放上該輸送帶，然後在未來的某個時間點將其原封不動地取下來使用。

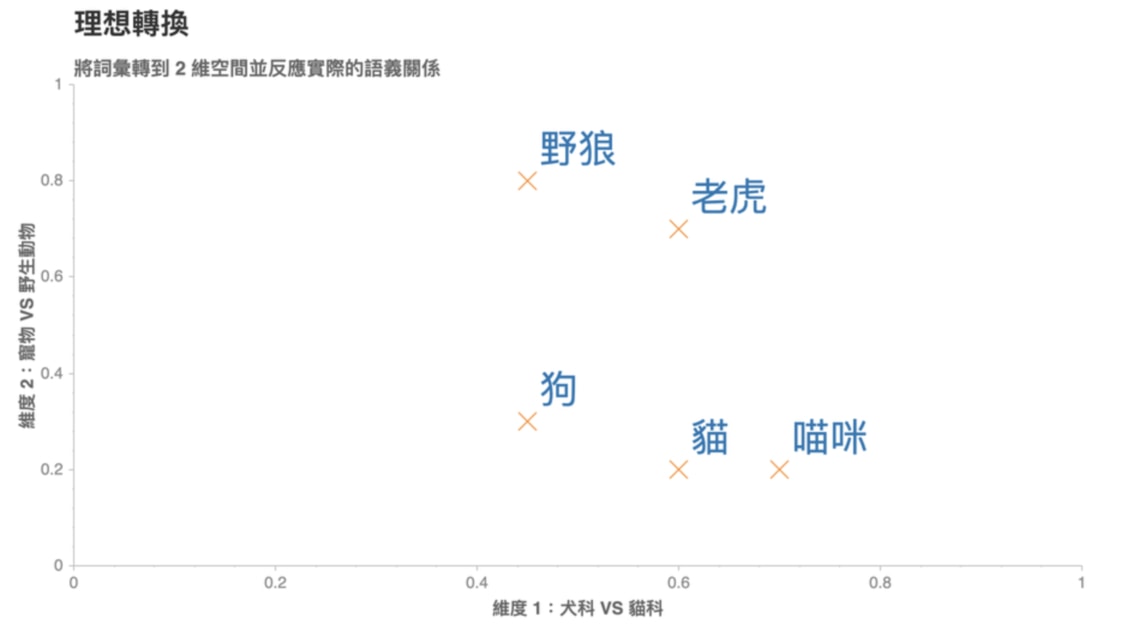
因為這樣的機制，讓 LSTM 即使面對很長的序列數據也能有效處理，不遺忘以前的記憶。

要讓神經網路能夠處理標題序列內的詞彙，我們要將它們表示成向量（更精準地說，是張量：Tensor），而不是一個單純數字。

如果我們能做到這件事情，則 RNN 就能用以下的方式讀入我們的資料：



注意：在每個時間點被塞入 RNN 的「詞彙」不再是 1 個數字，而是一個 N 維向量（圖中 N 為 3）



在這個 2 維空間裡頭，我們可以發現一個好的轉換有 2 個特性：

距離有意義：「喵咪」與意思相近的詞彙「貓」距離接近，而與較不相關的「狗」距離較遠

維度有意義：看看（狗, 貓）與（野狼, 老虎）這兩對組合，可以發現我們能將維度 1 解釋為貓科 VS 犬科；維度 2 解釋為寵物與野生動物

好消息是，大部分的情況我們並**不需要自己手動設定每個詞彙的詞向量**。我們可以**隨機初始化**所有詞向量（如前述的隨機轉換），並利用平常訓練神經網路的反向傳播算法（Backpropagation），讓神經網路自動學到一組適合當前 NLP 任務的詞向量

注意：反向傳播讓神經網路可以在訓練過程中修正參數，持續減少預測錯誤的可能性

在 NLP 裏頭，這種將一個詞彙或句子轉換成一個實數詞向量（Vectors of real numbers）的技術被稱之為詞嵌入（Word Embedding）。

而在 Keras 裡頭，我們可以使用 Embedding 層來幫我們做到這件事情：

from keras import layers

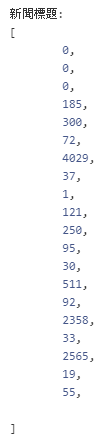
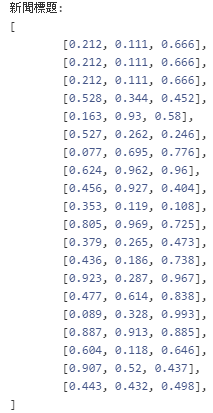
embedding\_layer = layers.Embedding(

MAX\_NUM\_WORDS, NUM\_EMBEDDING\_DIM)

**MAX\_NUM\_WORDS** 是我們的字典大小（10,000 個詞彙）、**NUM\_EMBEDDING\_DIM** 則是詞向量的維度。常見的詞向量維度有 128、256 或甚至 1,024。

Embedding 層一次接收 k 個長度任意的數字序列，並輸出 k 個長度相同的序列。輸出的序列中，每個元素不再是數字，而是一個 NUM\_EMBEDDING\_DIM 維的詞向量。

假如我們將第一筆（也就是 k = 1）假新聞標題 A 丟入 Embedding 層，並設定 NUM\_EMBEDDING\_DIM 為 3 的話，原來的標題 A：



序列裡頭的每個數字（即詞彙）都被轉換成一個 3 維的詞向量，而相同數字則當然都會對應到同一個詞向量（如前 3 個 0 所對應到的詞向量）。

一般來說，多數你見過的神經網路只會接受一個資料來源：

輸入一張圖片，判斷是狗還是貓

輸入一個音訊，將其轉成文字

輸入一篇新聞，判斷是娛樂還是運動新聞

一個能夠讀入成對新聞標題，並判斷兩者之間關係的神經網路架構：

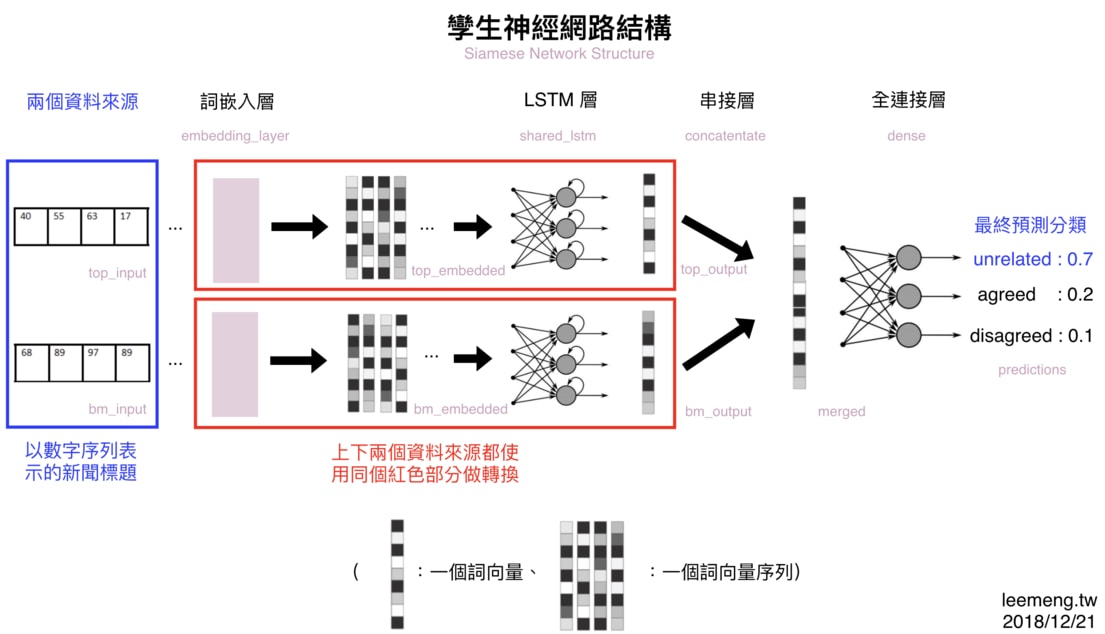
不相關（unrelated）

新聞 B 同意 A（agreed）

新聞 B 不同意 A（disagreed）

要怎麼做到這件事情呢？

我們可以使用孿生神經網路（Siamese Network）架構：



一部份的神經網路（紅框部分）被重複用來處理多個不同的資料來源（在本篇中為 2 篇不同的新聞標題）。

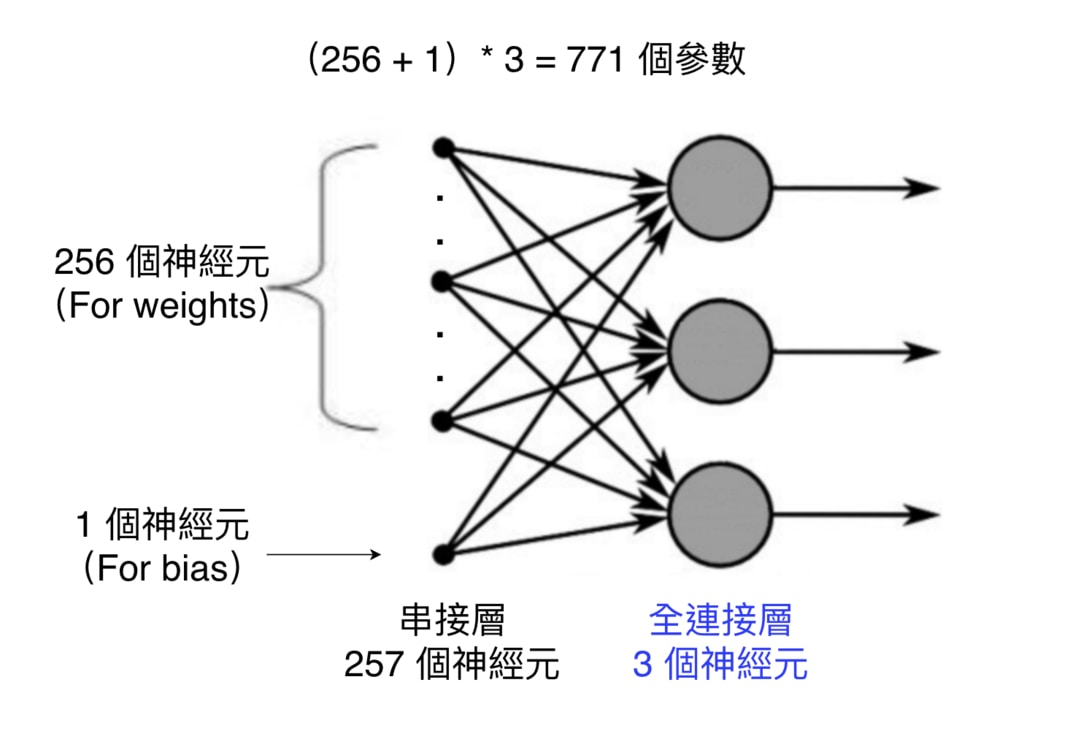
而會想這樣做，是因為不管標題內容是新聞 A 還是新聞 B，其標題本身的語法 & 語義結構大同小異。

神經網路說到底，就跟其他機器學習方法相同，都是對輸入進行一連串有意義的數據轉換步驟。神經網路將輸入的數據轉換成更適合解決當前任務的數據格式，並利用轉換後的數據進行預測。

以這樣的觀點來看的話，我們並不需要兩個不同的 LSTM 來分別將新聞 A 以及新聞 B 的詞向量做有意義的轉換，而是只需要讓標題 A 與標題 B 共享一個 LSTM 即可。畢竟，標題 A 跟標題 B 的數據結構很像。

唯一沒有在前面章節提到的是全連接層（Fully Connected Layer）以及其使用的 Softmax 函式。

全連接層顧名思義，代表該層的每個神經元（Neuron）都會跟前一層的所有神經元享有連結：



因為只需要預測 3 個分類，本文的全連接層只有 3 個神經元

全連接層在最下面。而因為其與前一層「緊密」連接的緣故，它在 Keras 裏頭被稱為 Dense 層

Param # 則紀錄了每一層所包含的模型參數（Parameters）。在機器學習的過程中，這些參數都會不斷地被調整，直到能讓模型能做出很好的預測。詞嵌入層有最多的參數，因為我們要為 字典裡頭的每個詞彙都建立一個 256 維度的詞向量，因此參數量為 10,000 \* 256。

這張表另外一個值得注意的地方是所有層的 Output Shape 的第一個維度都是 None。而 None 代表著可以是任意的數字。

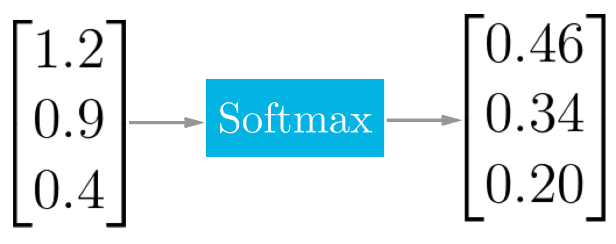
在 Keras 裡頭，第一個維度代表著樣本數（#Samples），比方說前 9,527 筆新聞標題 A 的數字序列的 shape 應該要是 （9527, 20）

而之所以每層的樣本數為 None 是因為 Keras 為了因應在不同場合會丟入不同數量的樣本需求。比方說，在訓練時你可能會一次丟 32 筆資料給模型訓練，但在預測的時候一次只丟 16 筆資料。

**Softmax**

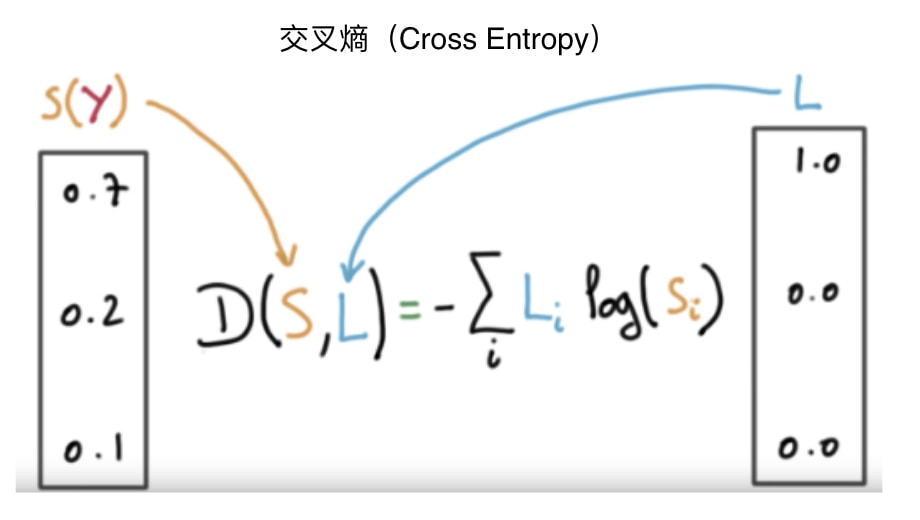
Softmax 函式一般都會被用在整個神經網路的**最後一層**上面，比方說我們這次的全連接層。

Softmax 函式能將某層中的所有神經元裡頭的數字作**正規化**（Normalization）：將它們全部壓縮到 0 到 1 之間的範圍，並**讓它們的和等於 1**。



Source: <https://towardsdatascience.com/deep-learning-concepts-part-1-ea0b14b234c8>

Loss function: Cross Entropy (交叉火商)



交叉熵能幫我們計算兩個機率分佈的差距，適合作為分類問題的損失函數

有了交叉熵來當作我們模型的損失函數，但是實際上模型要如何**更新裡頭的參數**呢？我們需要一個優化器（Optimizer）來做到這件事情。

雖然我們有很多種優化器，但它們基本上都是從梯度下降法（Gradient Descent）延伸而來。

不同優化器則有各自往下滑的秘方，比方說自動調整 Learning rate。

現在就先讓我們使用 RMSProp 優化器。而在有了損失函數以及優化器以後，我們就可以正式開始訓練模型了！

這邊特別值得拿出來提的是以下兩個參數：

BATCH\_SIZE

NUM\_EPOCHS

依照我們前面對損失函數（Loss Function）的說明，理論上模型是把訓練資料集裡頭的 32 萬筆資料全部看完一遍之後，再更新一次參數以降低損失函數。

但是這樣太曠日廢時，訓練可能要花很久才能完成。

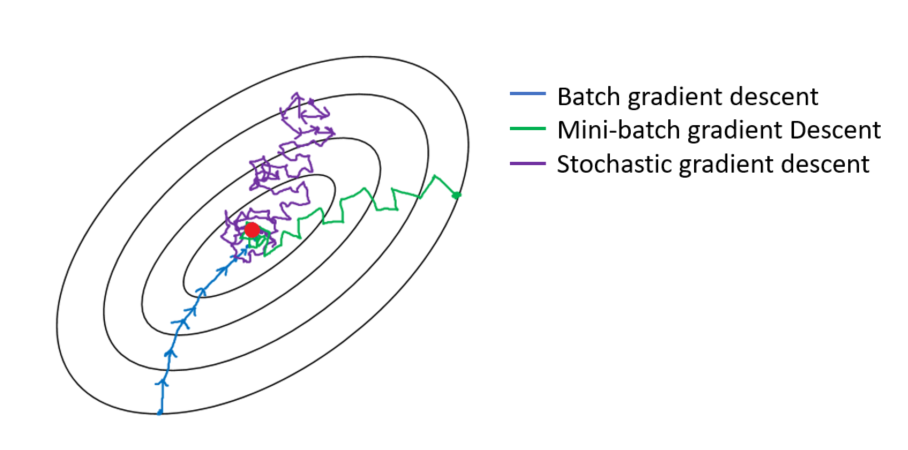
實務上都是每次只放入幾筆訓練數據，讓模型看完這些資料後就做一次參數的更新。而這個「幾筆」，就是 BATCH\_SIZE。

依照 BATCH\_SIZE 的大小，梯度下降（Gradient Descent, 後稱 GD）可以概括為 3 個類別：

GD（BATCH\_SIZE = 訓練資料集大小，且這時不稱為 batch）

Mini-batch GD（BATCH\_SIZE 通常為一個較小的 2 的倍數）

SGD（BATCH\_SIZE = 1）



Source: <https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3>

如上圖所示，下方的 GD 因為在每次更新參數前都會看完訓練資料集裡頭所有的數據，因此它更新參數的方向是最可靠的。但要往前走一步就就得看完 32 萬筆數據，未免成本也太大。

另一個極端是上方的 SGD：模型每看完 1 個訓練數據就嘗試更新權重，而因為單一一筆訓練數據並不能很好地代表整個訓練資料集，前進的方向非常不穩定。

因此我們常常採用的是中庸之道： Mini-batch GD 的方式來訓練模型，而這靠的是指定 model.fit 函式裡頭的 batch\_size。

NUM\_EPOCHS 則很容易理解：你希望模型不只將 32 萬筆的訓練數據都看過一遍，而是每一筆資料還要多看過好幾次，以讓模型確確實實地從它們身上學到東西。**NUM\_EPOCHS = 10 的意思就代表模型會重複看整個訓練資料集 10 次**。

Online course

台大電機系李宏毅教授的深度學習課程

奠定理論基礎

Coursera 的 Deep Learning 專項課程

理論 70 % + 實作 30 %

Deep Learning with Python

注重程式實作

<https://www.coursera.org/specializations/deep-learning>

<https://www.oreilly.com/library/view/deep-learning-with/9781617294433VE/>

<https://github.com/fchollet/deep-learning-with-python-notebooks>