生成式 AI：文字與圖像生成的原理與實務

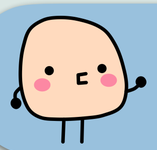
05.
RNN 及 transformers 的數學原理

蔡炎龍

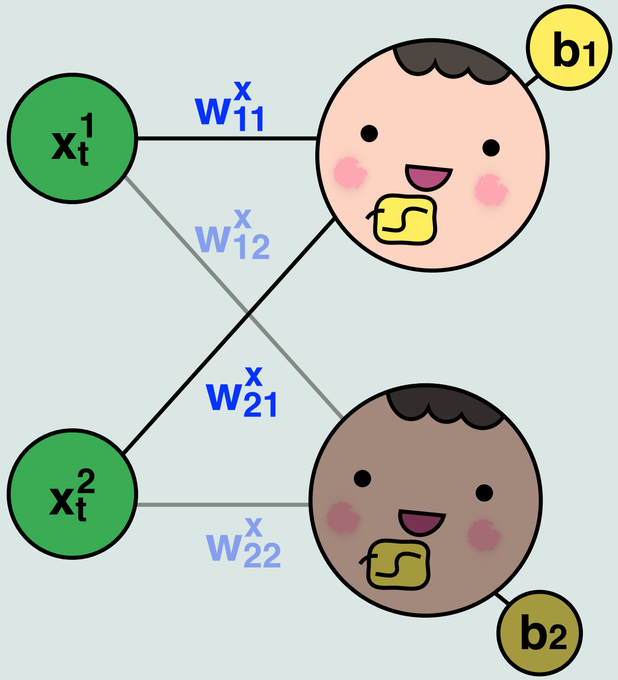
政治大學應用數學系

01.

RNN 的原理



遞歸層原理

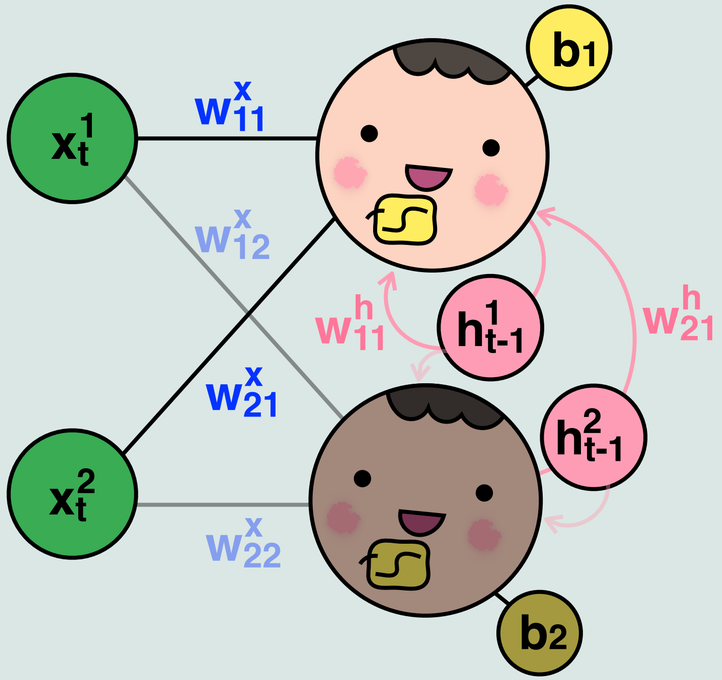


RNN 層

我們用有兩個輸入, 兩個 RNN 神經元的遞歸層來說明 (我們專注在第一個 RNN 神經元)。

看起來好像和以前全連結層一樣...

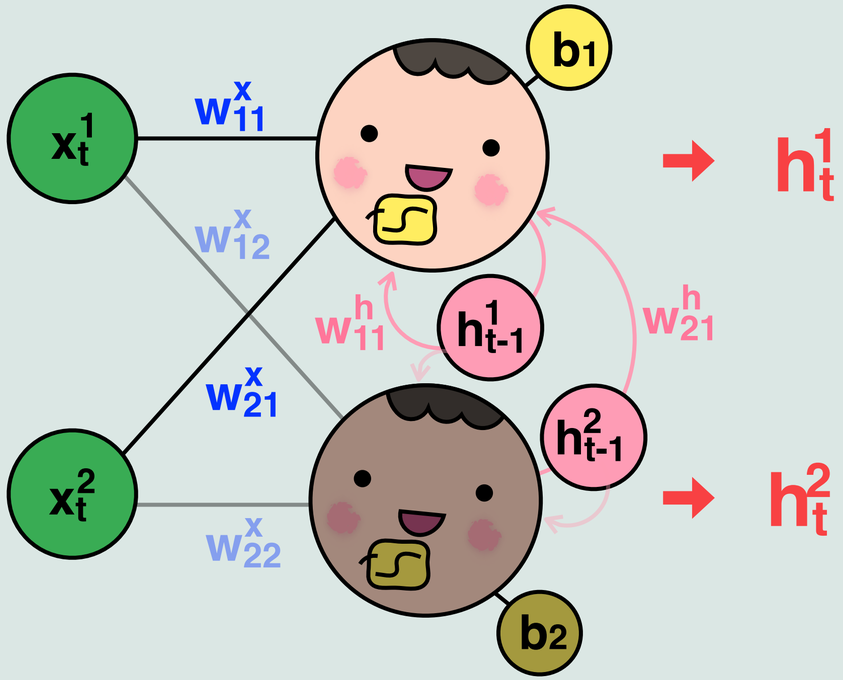
遞歸層原理



RNN 層

不一樣的是上次 RNN 層的輸出還會再傳回來。

遞歸層原理

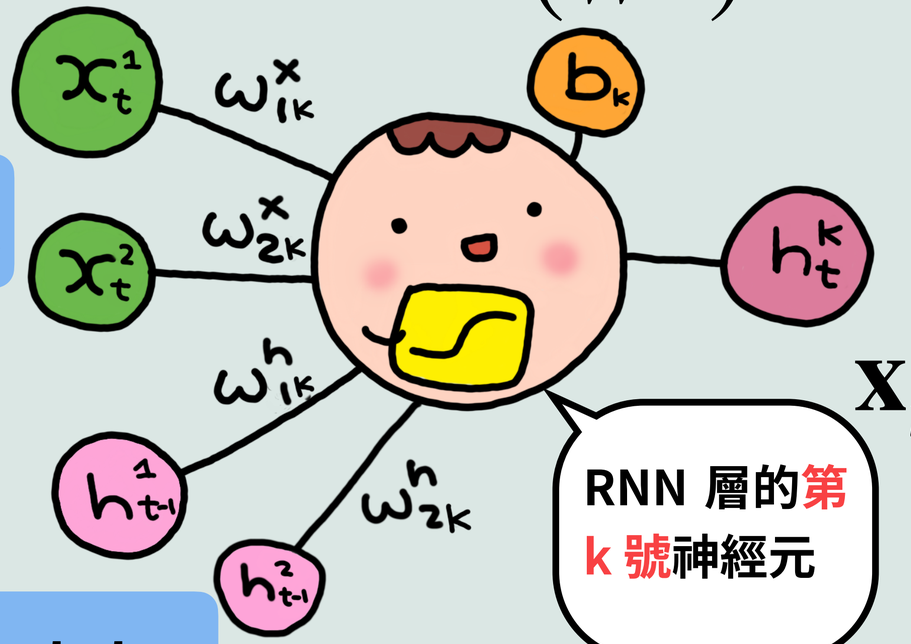


然後才決定這次的輸出。

$$h_t^1 = \sigma(w_{11}^x x_t^1 + w_{21}^x x_t^2 + w_{11}^h h_{t-1}^1 + w_{21}^h h_{t-1}^2 + b_1)$$

認真看 RNN 神經元就是一般的神經元 (只是有兩種輸入)

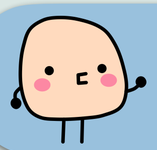
$$(W^X)^T = \begin{bmatrix} w_{11}^x & w_{21}^x \\ w_{12}^x & w_{22}^x \end{bmatrix}$$



$$\mathbf{x}_t = \begin{bmatrix} x_t^1 \\ x_t^2 \end{bmatrix}$$

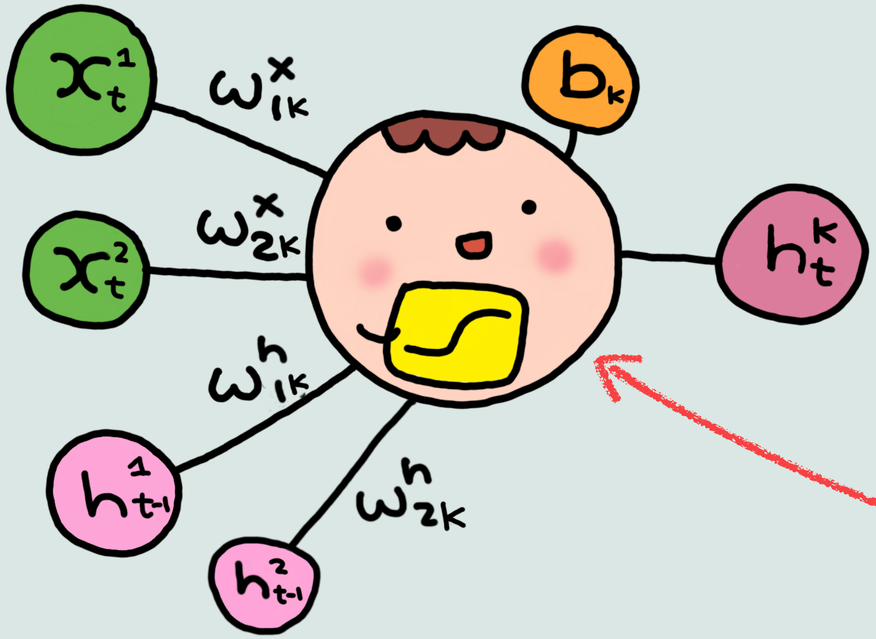
「正常」輸入

上次的 hidden states



認真看 RNN 神經元就是一般的神經元

$$(W^X)^T \cdot \mathbf{x}_t = \begin{bmatrix} w_{11}^x x_t^1 + w_{21}^x x_t^2 \\ w_{12}^x x_t^1 + w_{22}^x x_t^2 \end{bmatrix}$$

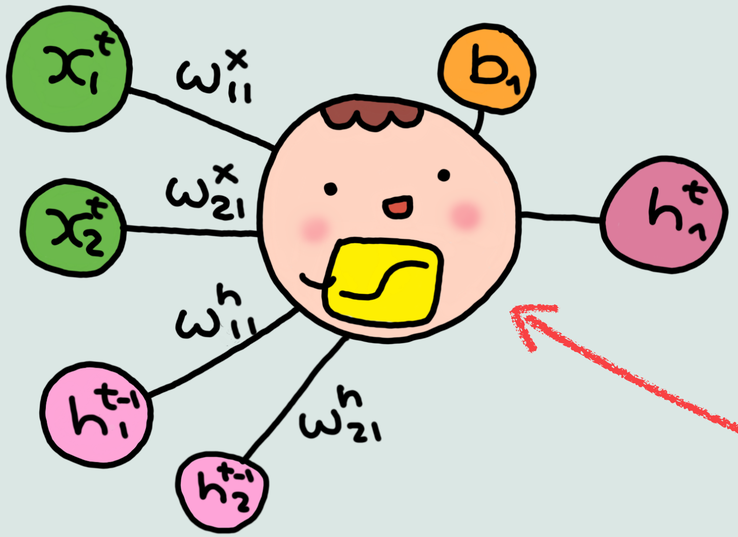


同時算了兩個神經元的!

1 號 RNN 神經元 ($k = 1$)

Hiden States 這邊基本上也是一樣的!

$$(W^H)^T \cdot \mathbf{h}_{t-1} = \begin{bmatrix} w_{11}^h h_{t-1}^1 + w_{21}^h h_{t-1}^2 \\ w_{12}^h h_{t-1}^1 + w_{22}^h h_{t-1}^2 \end{bmatrix}$$



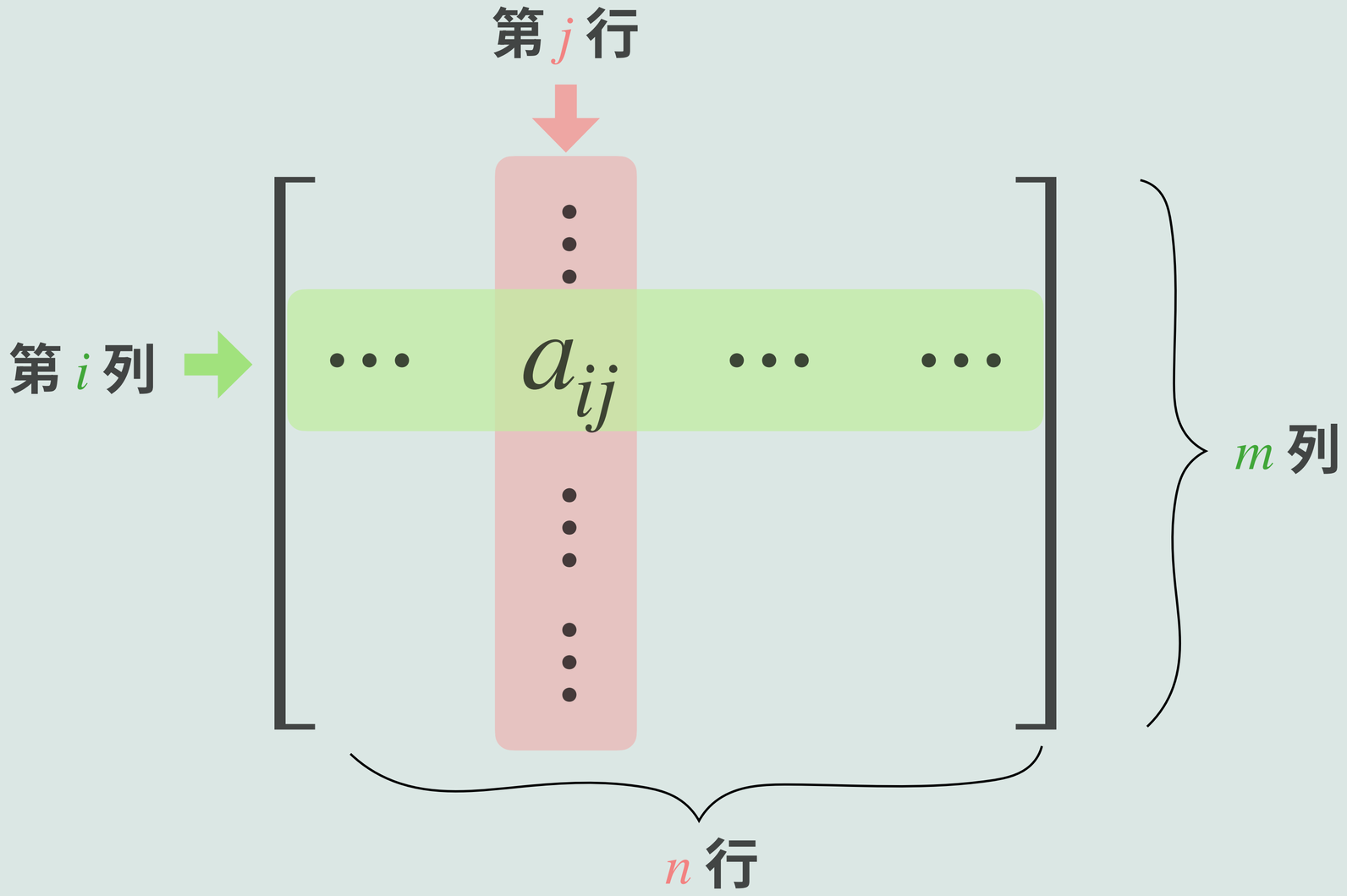
同時算了兩個神經元的!

1 號 RNN 神經元 ($k = 1$)

02.

線性代數 101

矩陣最大的關鍵就是先列後行!



a_{ij}

列 行

$m \times n$ 矩陣

列 行

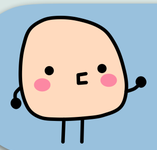
乘法也是先列後行

第 j 行

第 i 列

$$\begin{bmatrix} \vdots \\ \cdots\ a_{ik}\ \cdots\ \cdots \\ \vdots \end{bmatrix} \cdot \begin{bmatrix} \cdots\ \vdots\ \cdots\ \cdots \\ \cdots\ b_{kj}\ \cdots\ \cdots \\ \cdots\ \vdots\ \cdots\ \cdots \end{bmatrix}$$

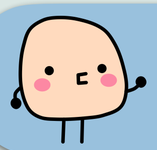
若 $C = AB$, c_{ij} 是 A 矩陣第 i 列和 B 矩陣第 j 行的內積。



內積 (嚴格說是 dot product) 是核心

設 $\mathbf{u} = [a_1, a_2, \ldots, a_p]$, $\mathbf{v} = [b_1, b_2, \ldots, b_p]$

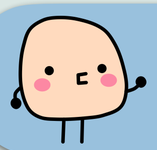
$$\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u}\mathbf{v}^T = \begin{bmatrix} a_1, a_2, \ldots, a_p \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{bmatrix}$$



列向量為中心的故事開展

一般我們說的線性轉換, 會寫成Ax。

但 Google 超喜歡列向量 (和內積), 於是引發了後面的故事...



矩陣乘法最重要技巧 (線性轉換)

$$\mathbf{x}\begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix} \quad A\begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_m \end{bmatrix}$$

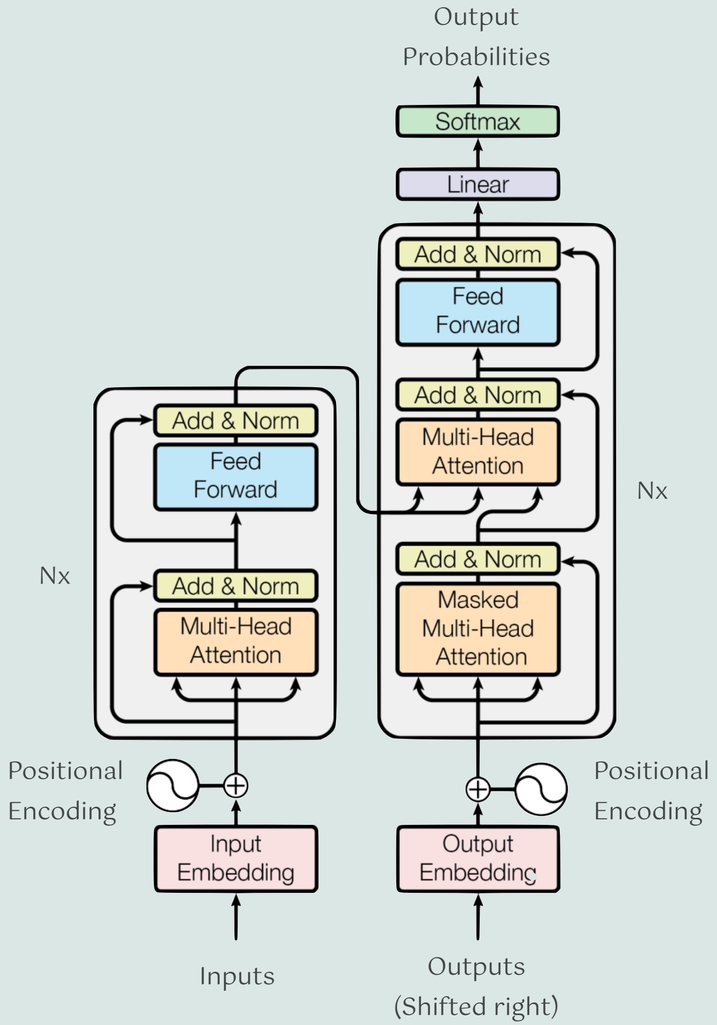
正好由 x 向量的分量,
對 A 矩陣的列向量做
線性組合!

$$= x_1 \cdot \mathbf{a}_1 + x_2 \cdot \mathbf{a}_2 + \cdots + x_m \cdot \mathbf{a}_m$$

03.

QKV 注意力機制

Transformers 的誕生



2017 年, Google 出了一篇有名的「Attention is All you need」, 介紹 transformers 的架構。最主要是想取代 RNN, 但從標題可知, (當時) Google 野心更大。

* "Attention is All you Need," Vaswani et. al., NIPS 2017

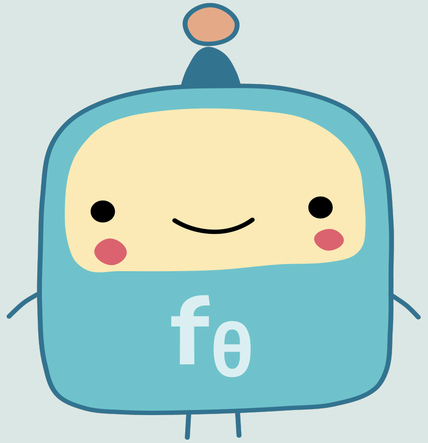
Attention 機制: 神秘的 QKV 概念

Q Query

K Key

V Value

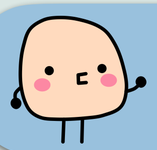
$\mathbf{q}$ (問題)



$\mathbf{h}$ (適合這問題的結果)

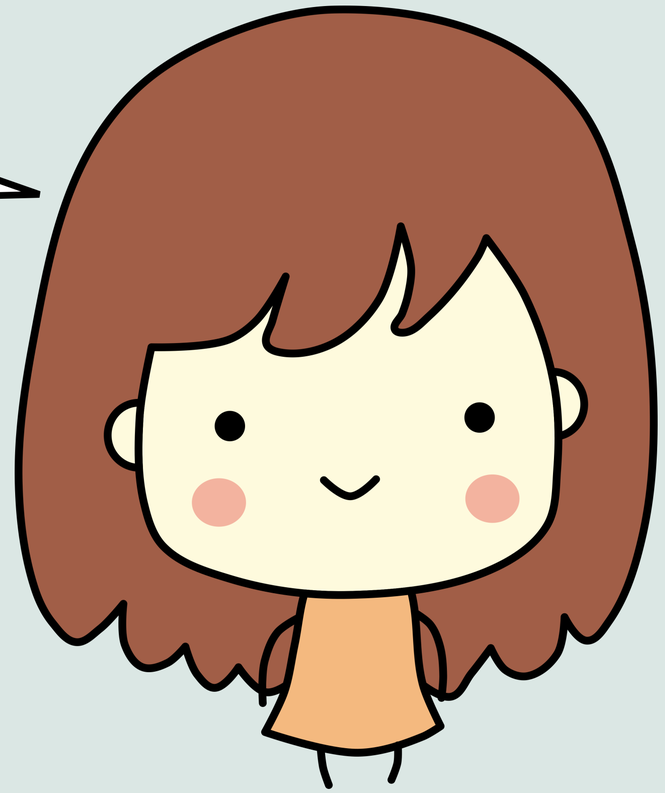
給定「先備知識」

$$\begin{array}{cccc} \mathbf{k}_1 & \mathbf{k}_2 & \cdots & \mathbf{k}_T \\ \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_T \\ \mathbf{x}_1 & \mathbf{x}_2 & & \mathbf{x}_T \end{array}$$

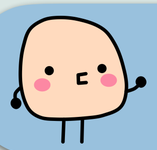


特徵代表向量

這裡的 $\mathbf{k}_t, \mathbf{v}_t$ 都是 $\mathbf{x}_t$ 的一個**特徵代表向量**。

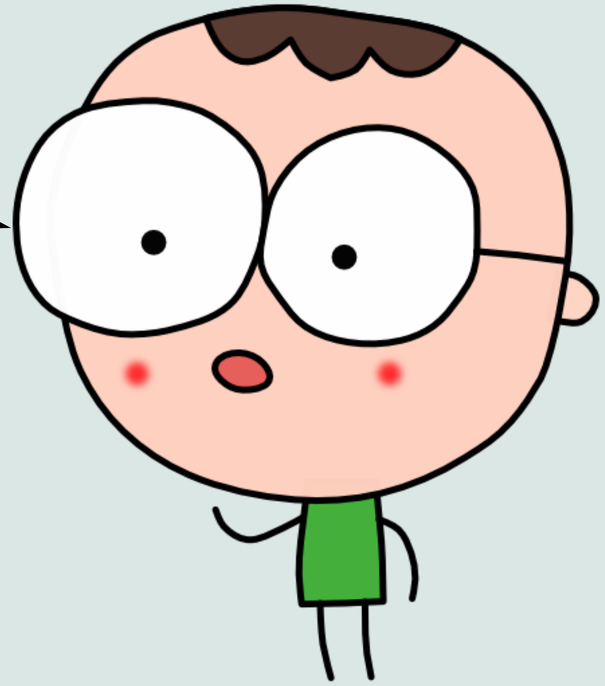


$$\mathbf{x}_t \rightarrow \mathbf{k}_t \quad \mathbf{v}_t$$

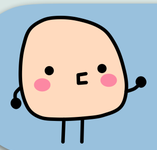


希望找到 Q 適合的代表向量

直覺的想法是有一段資訊, 比方說一段文字 $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T$, 現在有個 query $\mathbf{q}$ 進來, 想依前面的資訊, 找出最適合 $\mathbf{q}$ 的代表向量 $\mathbf{h}$。



$$\mathbf{q} \rightarrow \mathbf{h}$$



最後這個向量長這樣

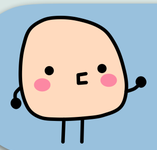
$$\mathbf{h} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \cdots + \alpha_T \mathbf{v}_T$$

$\mathbf{q}$ 和 $\mathbf{x}_1$ 的相關強度

$\mathbf{q}$ 和 $\mathbf{x}_2$ 的相關強度

$\mathbf{q}$ 和 $\mathbf{x}_T$ 的相關強度

所謂相關強度是 attention 的強度, 基本上愛怎麼合理的算都可以

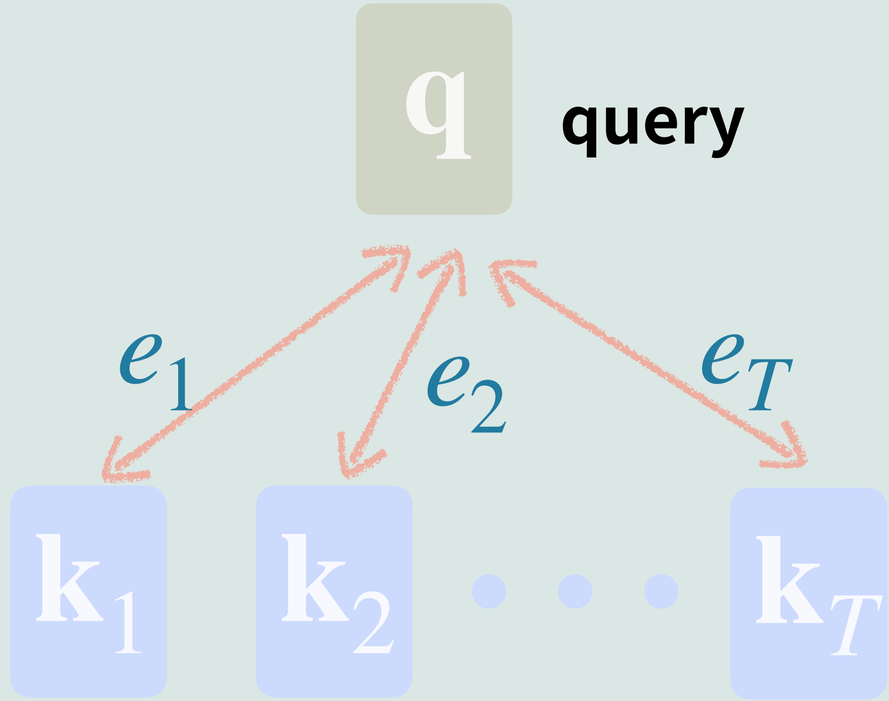


計算「注意力」的強度

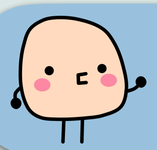
Attentation (alignment)

$$a(\mathbf{q}, \mathbf{k}_t) = \langle \mathbf{q}, \mathbf{k}_t \rangle$$

$$= \mathbf{q}\mathbf{k}_t^T = e_t$$



這是 Google 最愛的方法

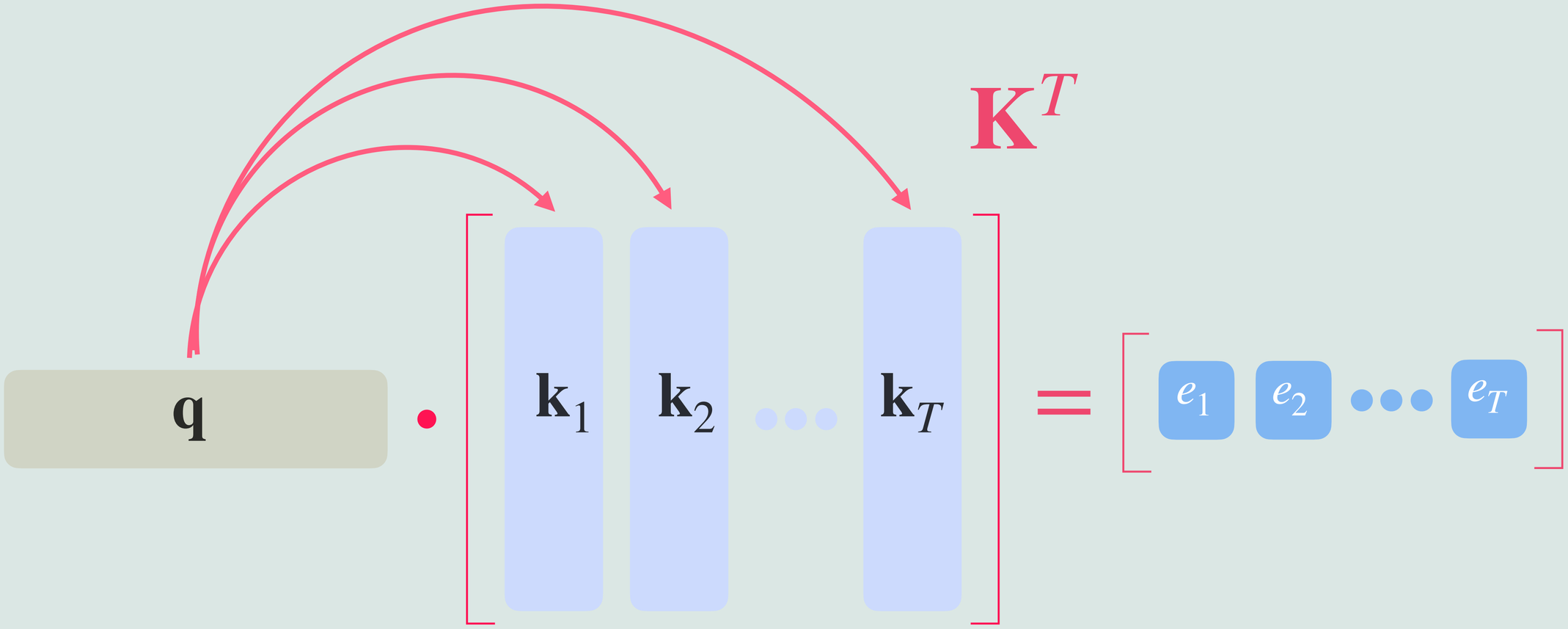


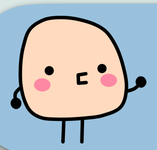
把所有的 $\mathbf{k}_i, \mathbf{v}_i$ 放入一個矩陣

$$\mathbf{K} = \begin{bmatrix} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \vdots \\ \mathbf{k}_T \end{bmatrix} \qquad \mathbf{V} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_T \end{bmatrix}$$

深愛列向量的 Google, 顯然會這樣寫

Attentation 強度只是簡單矩陣乘法一次算完





再經 softmax 就是最後權重係數

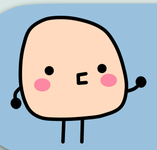
$$\mathbf{K}^T$$

$$\mathbf{q} \cdot \begin{bmatrix} \mathbf{k}_1 & \mathbf{k}_2 & \cdots & \mathbf{k}_T \end{bmatrix} = \begin{bmatrix} e_1 & e_2 & \cdots & e_T \end{bmatrix}$$

$$\downarrow \texttt{softmax}$$

$$\begin{bmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_T \end{bmatrix}$$

$$\mathbf{h} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \cdots + \alpha_T \mathbf{v}_T$$

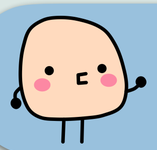


式子可以很美

$$\text{softmax}(\mathbf{q} \cdot K^T) = \begin{bmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_T \end{bmatrix}$$

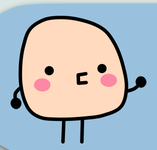
$$\begin{bmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_T \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_T \end{bmatrix} V = \sum_{t=1}^{T} \alpha_t \mathbf{v}_t$$

記得這乘法就是
列向量線性組合

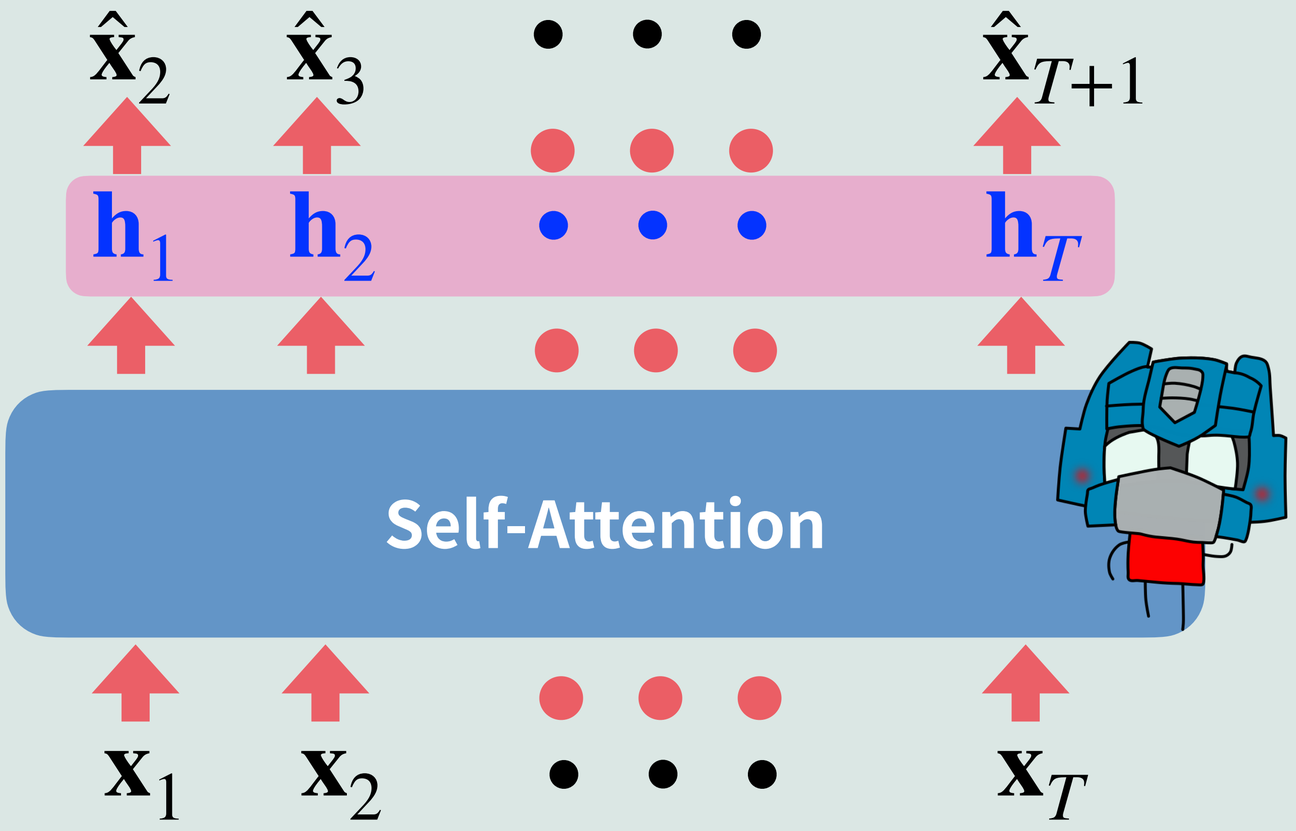


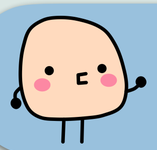
最終美美的 (?) 式子長這樣

$$\text{Attention}(\mathbf{q}, K, V) = \text{softmax}(\mathbf{q} \cdot K^T)V$$



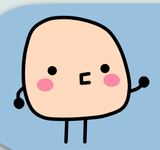
Self-Attention





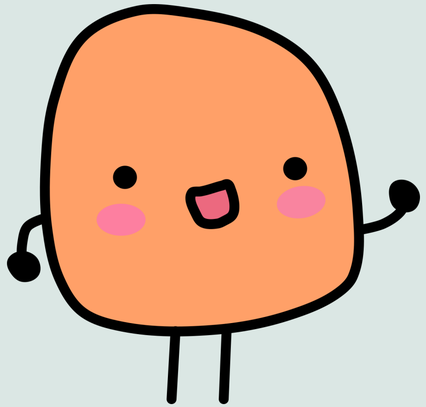
基本上是每個 x 再增加 q 特徵代表向量

$$\mathbf{x}_t \rightarrow \mathbf{q}_t \quad \mathbf{k}_t \quad \mathbf{v}_t$$



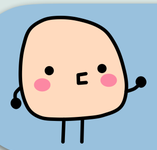
其實就是大家輪流當 q

就是標準
attention 計算



$$\mathbf{q}_j \rightarrow \mathbf{h}_j = \sum_{t=1}^{T} \alpha_t \mathbf{v}_t$$

$$\mathbf{x}_1 \quad \mathbf{x}_2 \quad \cdots \quad \mathbf{x}_j \quad \cdots \quad \mathbf{x}_T$$

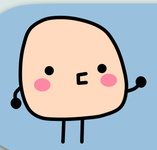


像 K, V 一樣, 把所有 $\mathbf{q}_i$ 放入一個矩陣

$$\mathbf{Q} = \begin{bmatrix} \mathbf{q}_1 \\ \mathbf{q}_2 \\ \vdots \\ \mathbf{q}_T \end{bmatrix}$$

就像前面說明的, 很容易可以導出 Google 引以為傲的公式。

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T)V$$



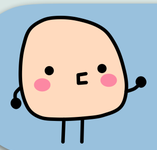
One More Thing

你可能會發現前面的公式和原始論文公式不同, 那是因為 Google 很尷尬的發現, 用內積算 attention 強度其實效果沒有人家 (比如訓練一個神經元去算) 好。主要原因是 softmax 是「贏者通吃」, 會讓權重高的比合理權重還要高很多。

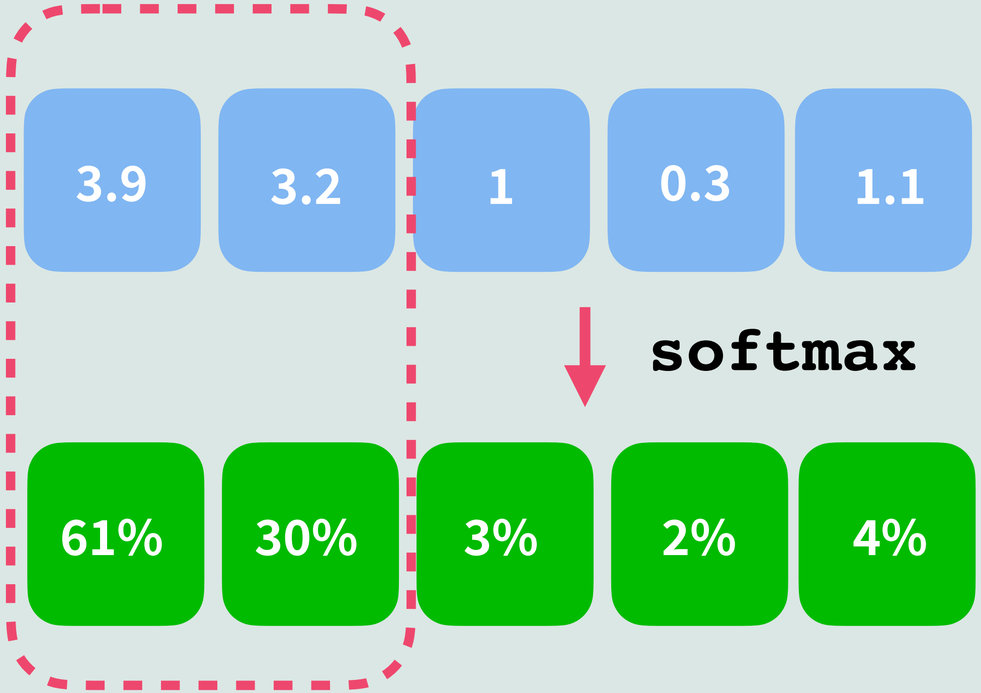
好在這不難救...

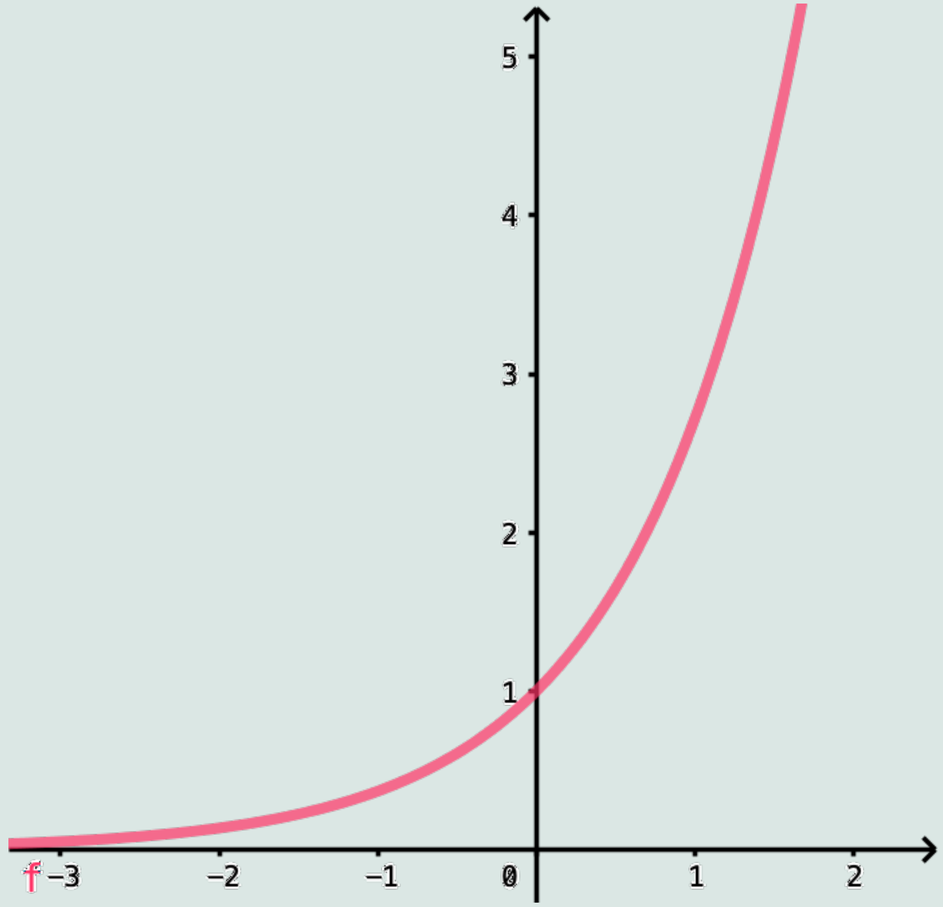
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$d_k = \dim(\mathbf{k}_i)$, 其實這裡放一個足夠大適當的數字就可以, 比如 $\sqrt{9487}$。Google 這樣寫基本上只是讓你覺得好有學問...

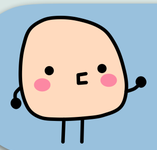


Softmax 贏者通吃





本來感覺只差一點啊...



把數字都拉到 0 的附近, 輕鬆解決問題

3.9	3.2	1	0.3	1.1

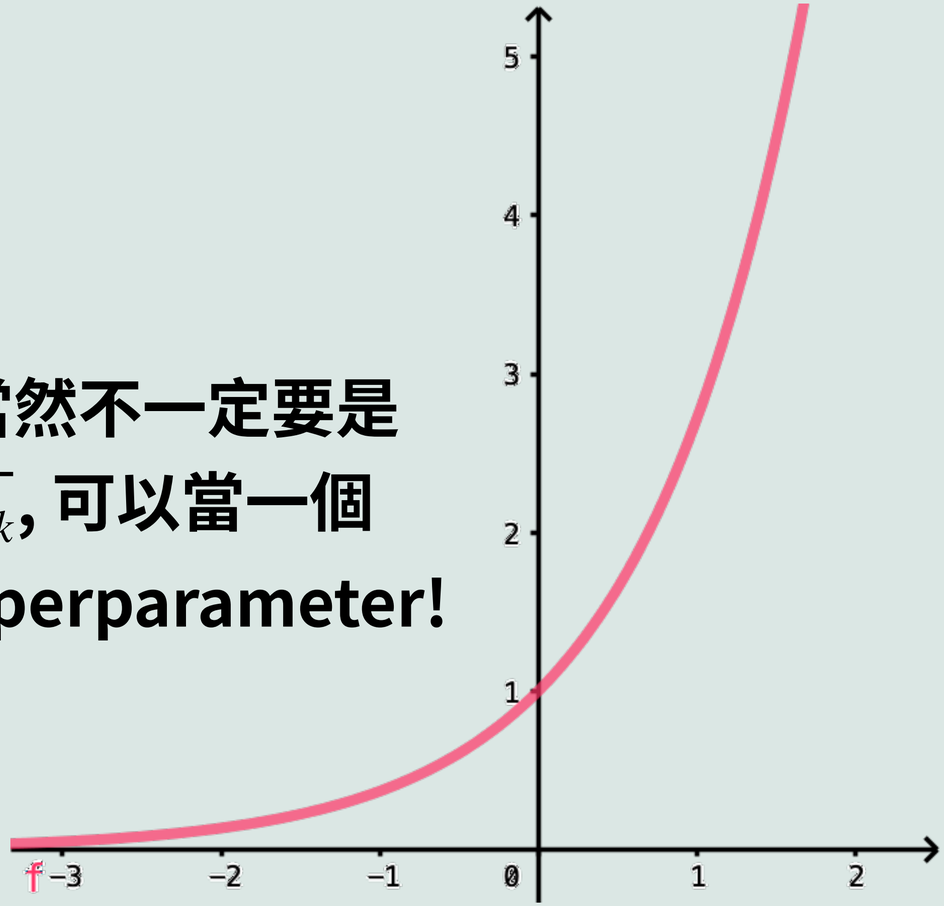
↓ 同除以 $\tau = \sqrt{d_k}$

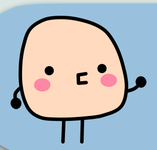
1.74	1.43	0.45	0.13	0.49

↓ `softmax`

40%	29%	11%	8%	11%

τ 當然不一定要是 $\sqrt{d_k}$, 可以當一個 hyperparameter!





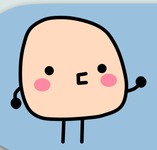
回過頭看各個向量是怎麼來的

$\mathbf{x}_1$ $\mathbf{x}_2$ $\cdots$ $\mathbf{x}_T$ 是 word embedding*

$$\mathbf{q}_t = \mathbf{x}_t W^Q, \mathbf{k}_t = \mathbf{x}_t W^K, \mathbf{v}_t = \mathbf{x}_t W^V$$

就是經過對 $\mathbf{x}_i$ 的線性轉換, 這些矩陣是學習來的

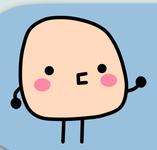
* 一般不用如 Word2Vec 等標準 word embedding, 而是直接加 embedding 層去學



Multihead Attention

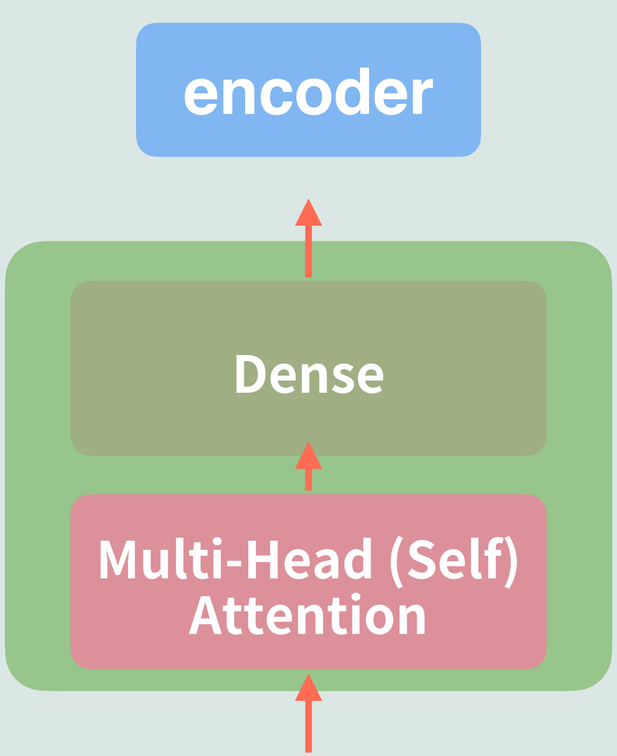
認真想想, attention 沒理由只有一種, 所以我們可以定義第 n 個 attention...

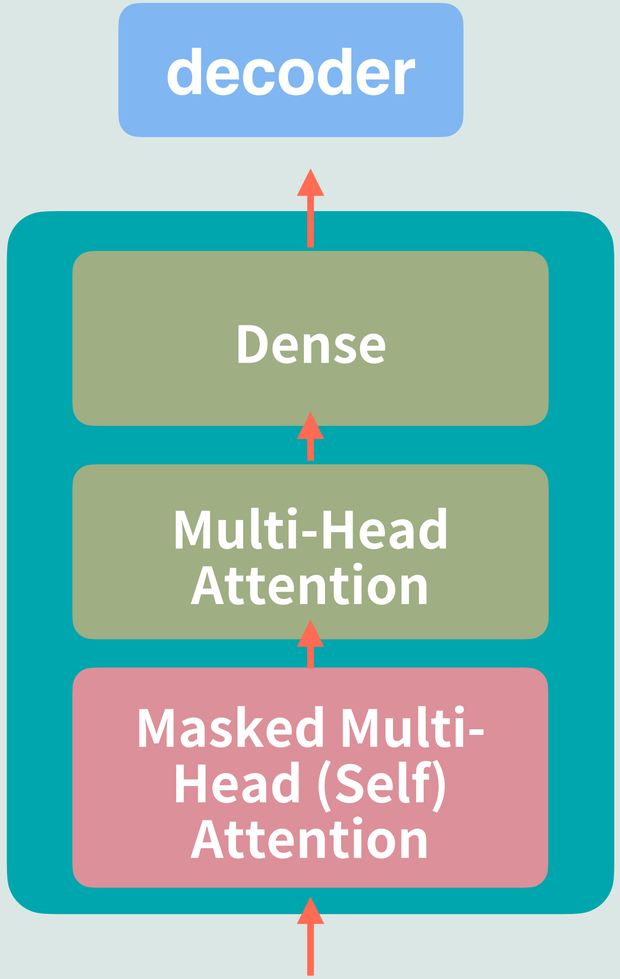
$$\text{Attention}(QW_n^Q, KW_n^K, VW_n^V)$$

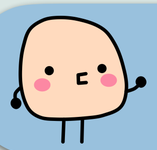


Transformer 層在 encoder 和 decoder 設計是不同的

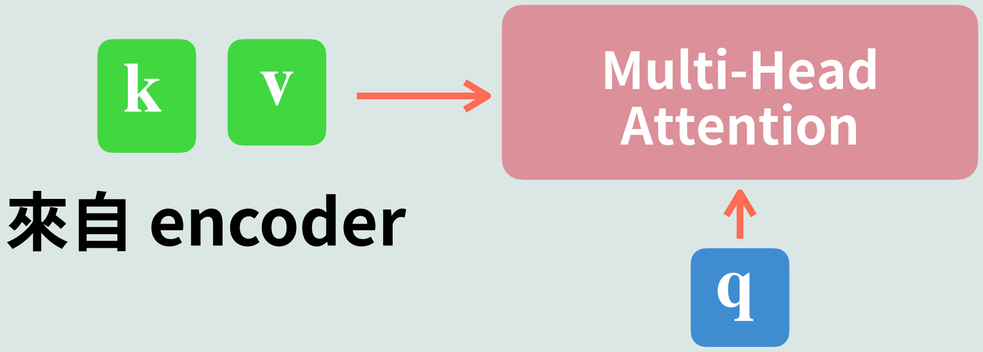
這兩處的 self-attenation 都是由輸入向量自家生產 $\mathbf{q}, \mathbf{k}, \mathbf{v}$





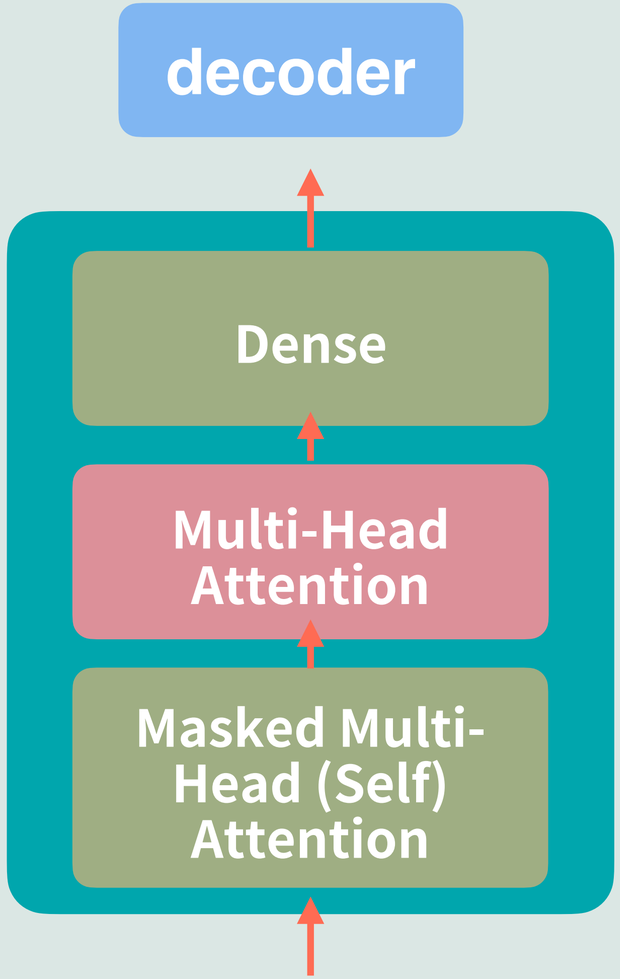


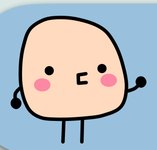
只有 decoder 中間那層不是 self-attention



來自 encoder

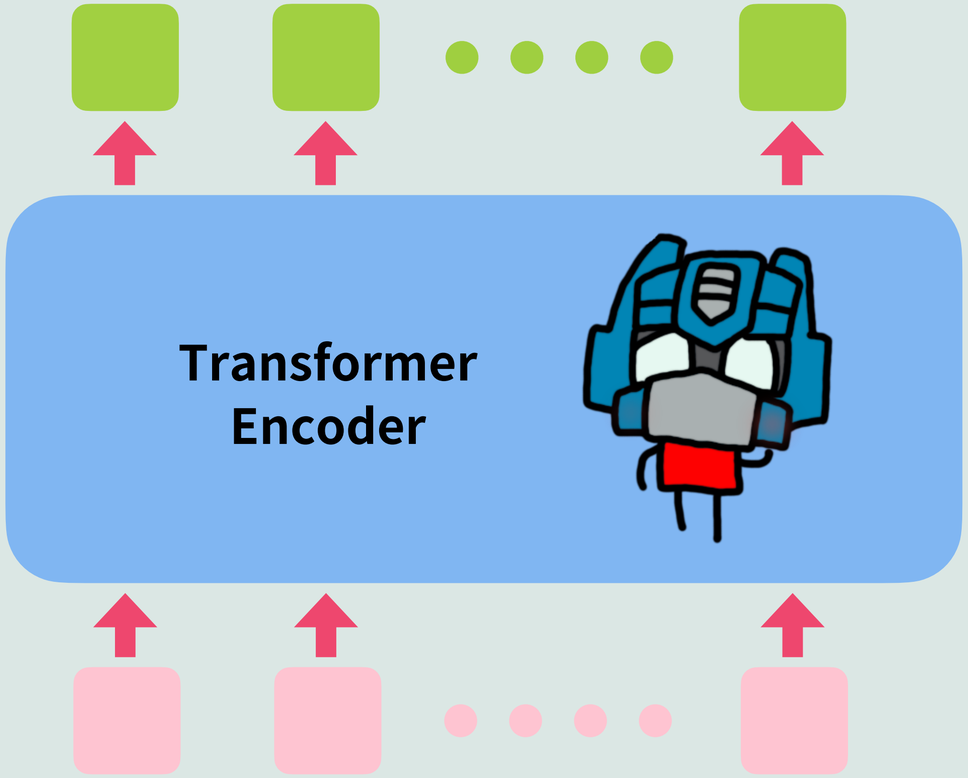
來自前一層的 output

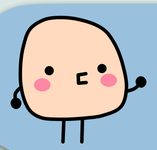




Masked Multihead Attention

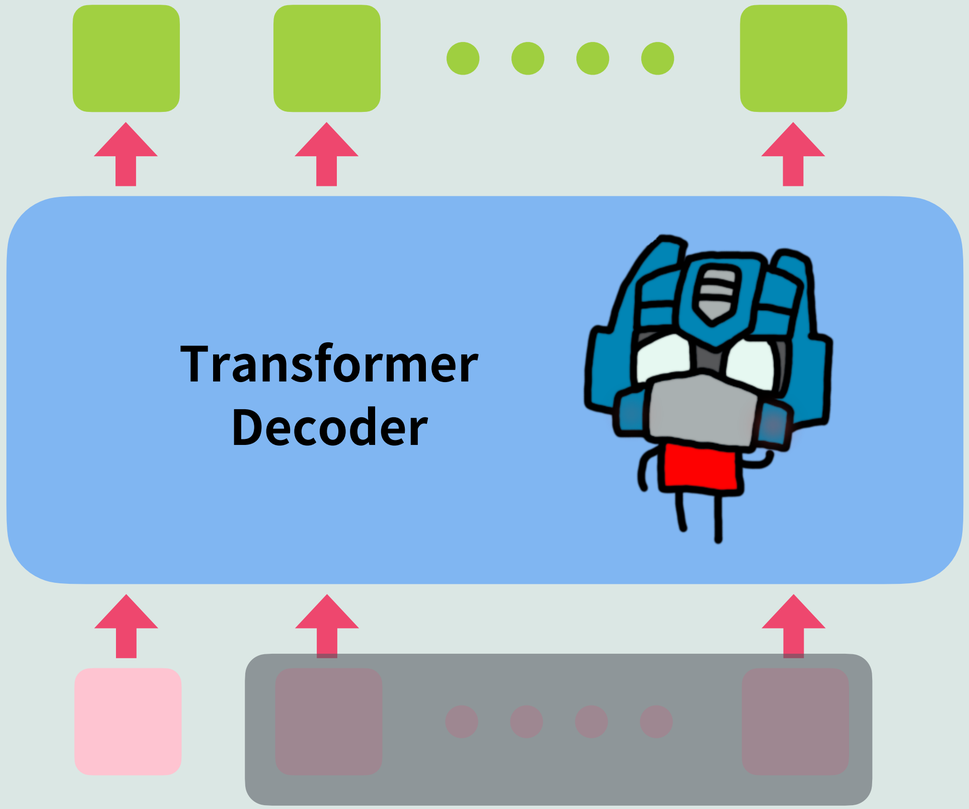
要注意的是, transformer 輸入有幾個 (字/詞向量), 輸出就有幾個。





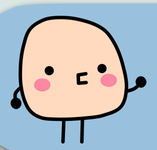
Masked Multihead Attention

包括 Decoder 也是這樣, 只是開始還沒有的輸入會被 mask 住。



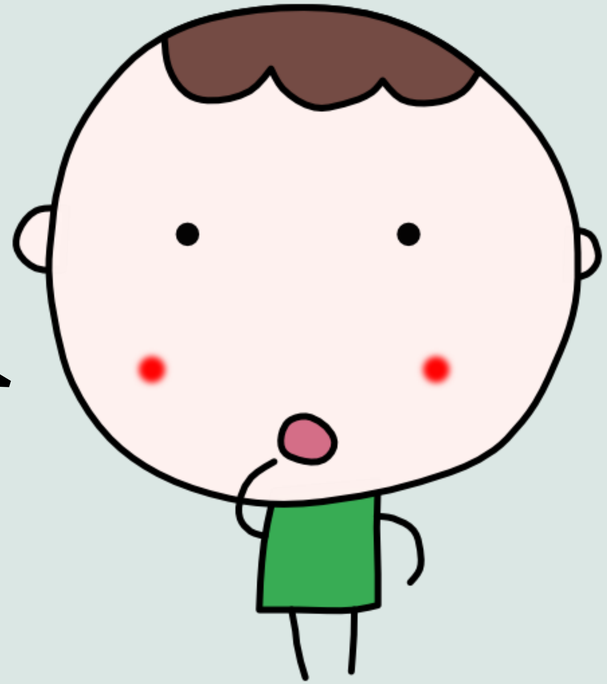
04.

Position Encoding



Position Encoding

注意 RNN 是真的一個字一個字讀, transformers 為了平行化, 一次做 self-attention, 每個字先後順序並沒有真的被考慮。所以我們應該要把位置資訊加入原來文字 embedding, 這叫 position encoding。



$$\mathbf{x}_t + \mathbf{p}_t$$

word embedding　　position encoding

觀察任何一個進位的數字系統

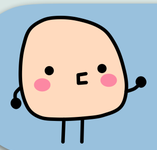
越高位數週期越長, 頻率越低。

比如我們熟悉的十進位數:

頻率 1/1000　　頻率 1/10　頻率 1

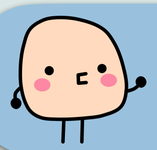
[9, 4, 8, 7]

頻率 1/100



為了通用, 我們想找個奇幻進位系統

- ☑ 連續函數
- ☑ 有週期性
- ☑ 數值不要太大 (以免影響原 embedding)



加上 position encoding

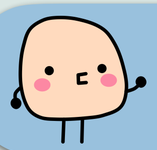
注意這是低位數!

這是高位數!

$$\mathbf{p}_t = \left[p_1, p_2, \cdots, p_d \right]$$

$$+$$

$$\mathbf{x}_t = \left[x_1, x_2, \cdots, x_d \right]$$



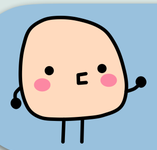
原論文兩兩一組同頻率

$$[\underbrace{p_0, p_1}_{\omega_0}, \cdots, \underbrace{p_{2k}, p_{2k+1}}_{\omega_k}, \cdots, p_{d-1}]$$

* 注意原論文足標通常是從 1 開始，但這裡應該要由 0 開始才符合後面的式子。

$$\omega_k = \frac{1}{10000^{2k/d}}$$

記得低位數在前面，後面的頻率要越來越慢，需要 $\omega_0 > \omega_1 > \cdots$



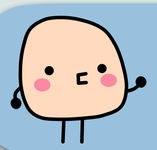
原論文的 position encoding 方式

$$\mathbf{p}_t = \left[p_0, p_1, \ldots, p_{d-1} \right]$$

$$p_i = \begin{cases} \sin(\omega_k \cdot t), & \text{當 } i = 2k \\ \cos(\omega_k \cdot t), & \text{當 } i = 2k+1 \end{cases}$$

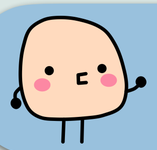
第 t 個字

位數越高頻率越低



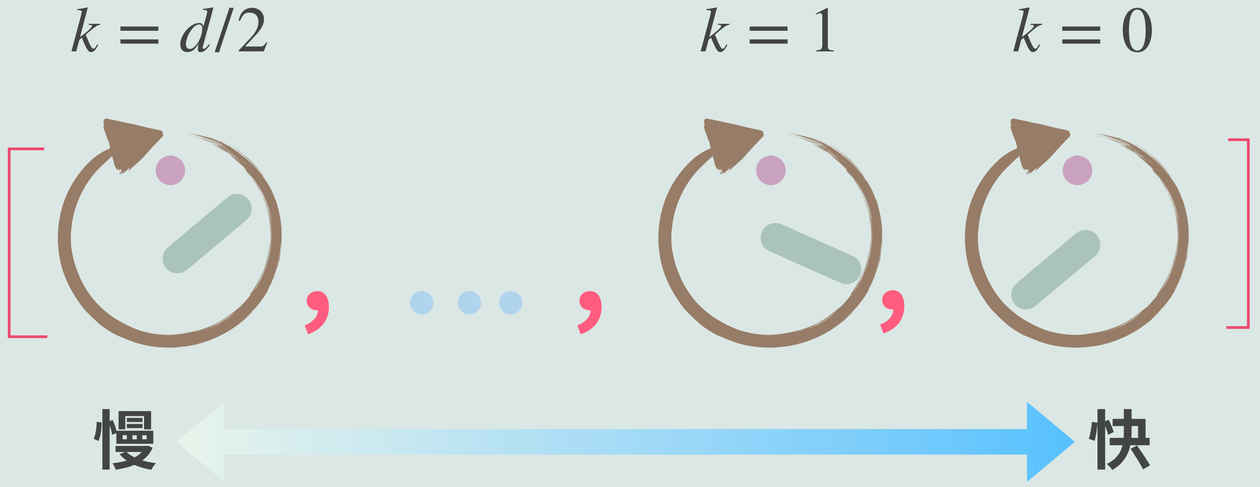
Position Encoding

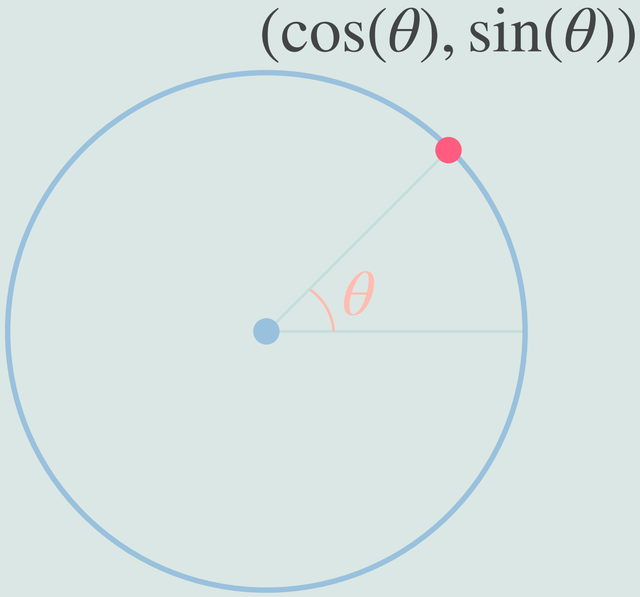
$$\mathbf{p}_t = [\sin(w_0)t, \cos(w_0)t,$$
$$\sin(w_1)t, \cos(w_1)t,$$
$$\vdots$$
$$\sin(w_{d/2})t, \cos(w_{d/2})t]$$

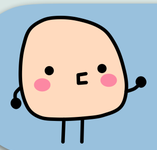


為什麼要加入 cos, 只有 sin 不行嗎?

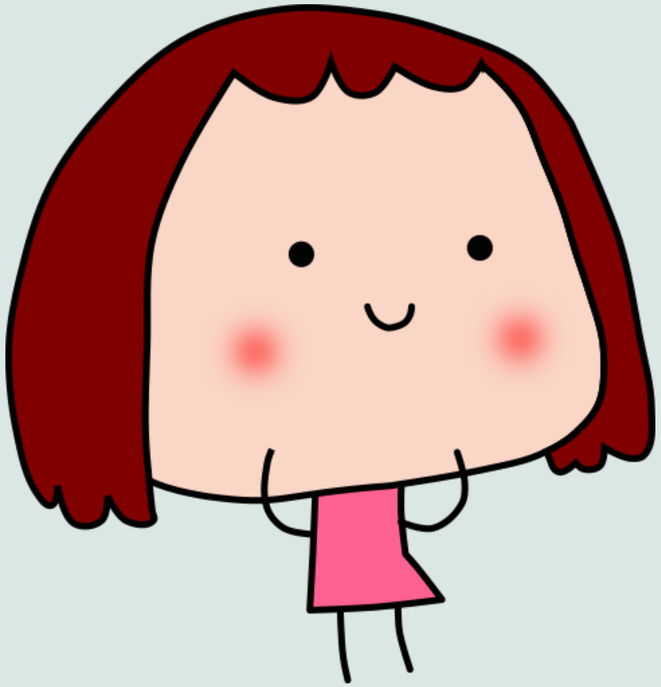
沒人規定一個位元只能用一個數字表示, $(\sin(\omega_t t), \cos(\omega_t t))$
剛好是單位圓上的一點, 我們可以想成時鐘上的一個點。
只是位元低的時鐘快, 位元高的時鐘慢。

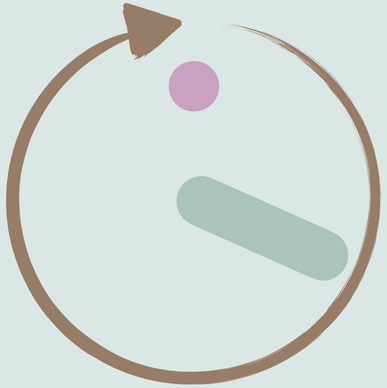






不要騙我不懂, 極座標明明是 cos 在前!

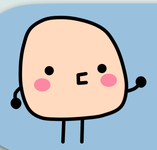




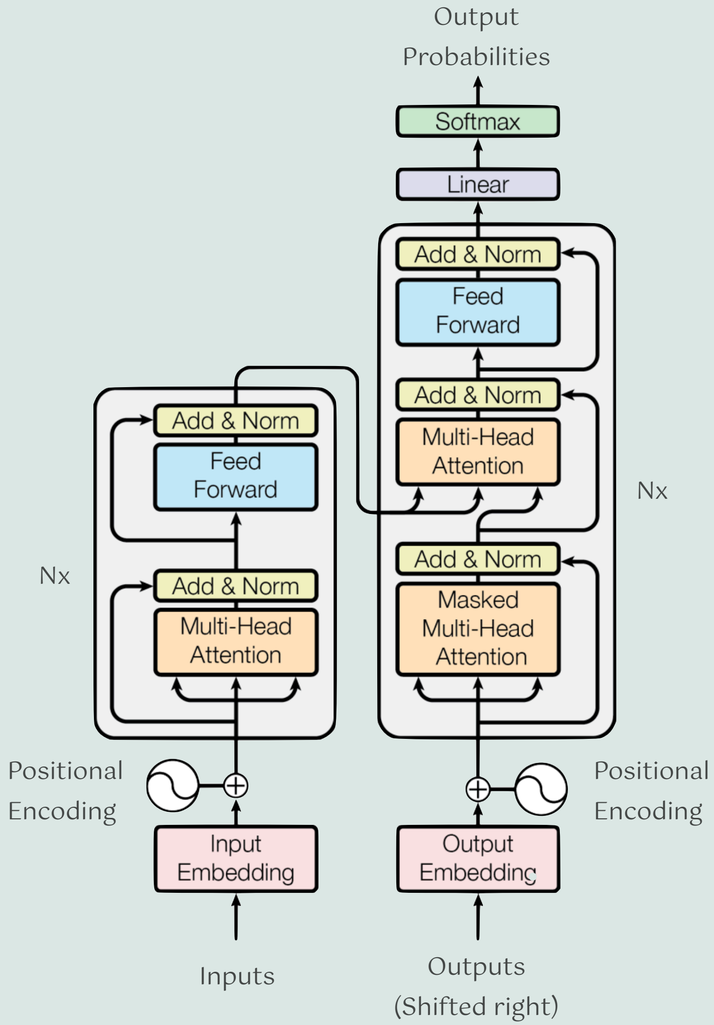
雖然不知 Google 的考量, 但 $(\sin(\theta), \cos(\theta))$ 繞一圈也是單位圓。而且 $\theta = 0$ 是指向一般時鐘 12 點方向, 並且走向會是順時鐘方向。

05.

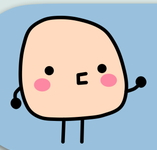
讓 transformers 更穩



原版的 Transformers 架構



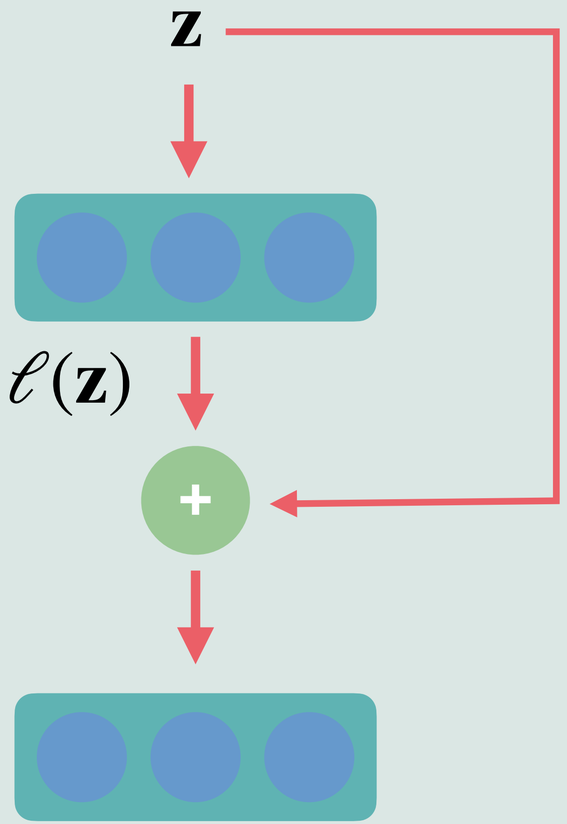
回頭看原本的 transformers 架構, 相信可以理解每一個細節了。我們沒有說的是 transformers 用了 residual 連結還有 layer normalization 的技術, 這些都是當年能讓神經網路更深、更穩定的最佳技術 (好在現在也差不多)。



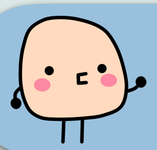
Transformers 中有個 ResNet 型的設計

我們看看 ResNet
是怎麼運作的?

ResNet 的設計方式



ResNet 的設計方式。



為什麼 ResNet 可以很深

這意思是本來某一層的輸出是

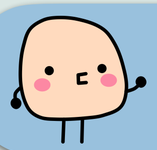
$$\ell(\mathbf{z})$$

現在變成

$$\mathbf{z} + \ell(\mathbf{z})$$

假設我們的目標 (**正確答案**) 是

$$f(\mathbf{z})$$



只需要學之前沒有學到的

也就是我們本來希望

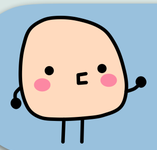
$$\ell(\mathbf{z}) \simeq f(\mathbf{z})$$

現在變成

$$\mathbf{z} + \ell(\mathbf{z}) \simeq f(\mathbf{z})$$

之前學到的

還沒學到的



會把重心放在前方

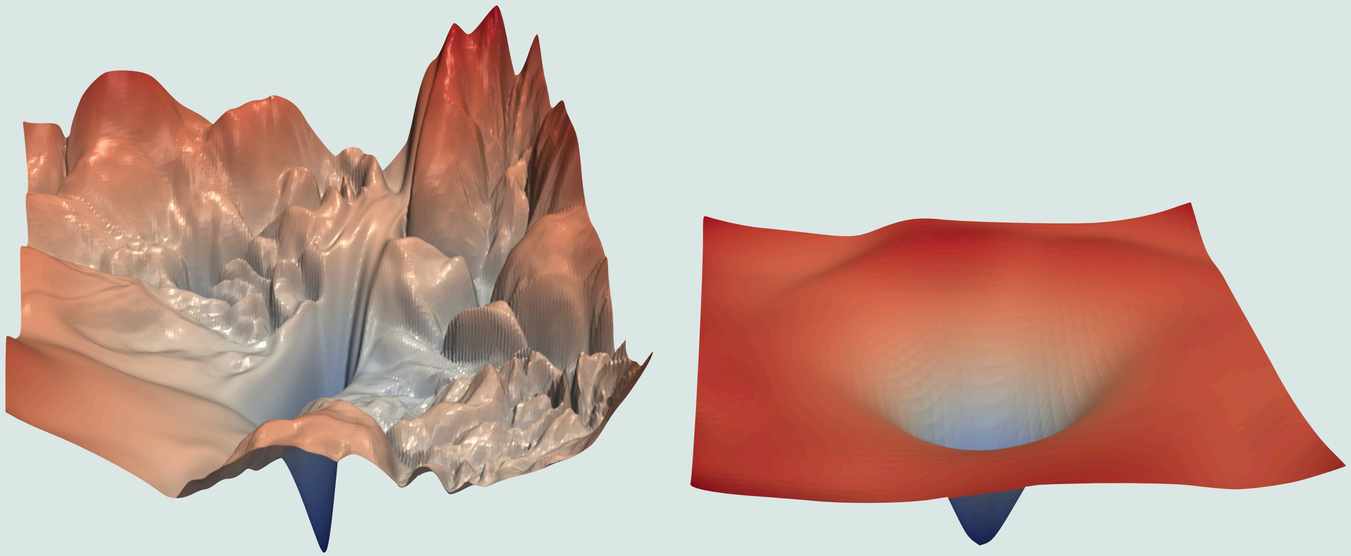
目標

$$\ell(\mathbf{z}) = f(\mathbf{z}) - \mathbf{z}$$

已經學到的

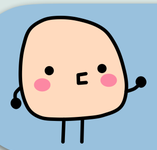
如果 $\mathbf{z}$ 已經很接近正確答案 $f(\mathbf{z})$, 那 $\ell(\mathbf{z})$ 就不太需要再學什麼...

平順的 loss function

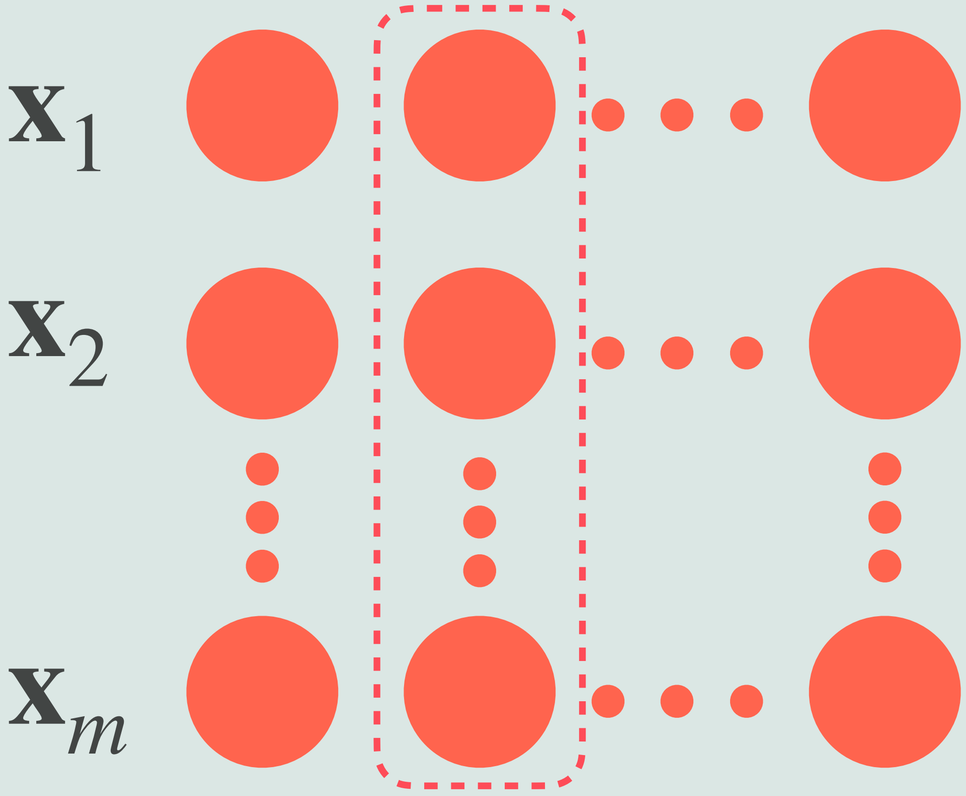


without skip **with skip**

Li,H., Xu, Z., Taylor, G., Studer C., Goldstein T. (NeuraIPS 2018). Visualizing the Loss Landscape of Neural Nets. .



Batch Normalization



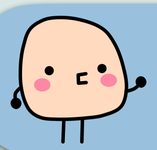
計算平均值、標準差

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

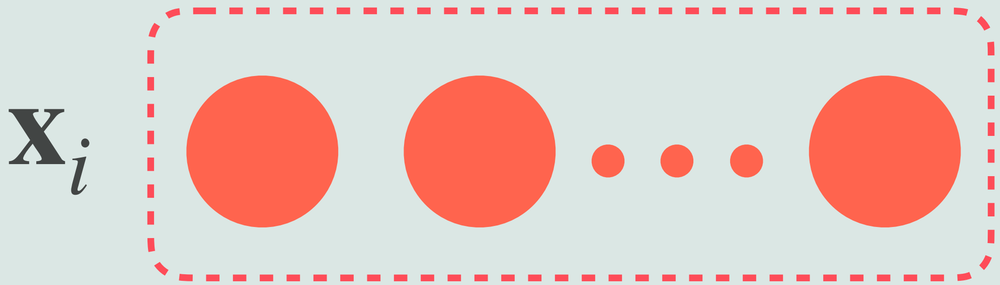
預防除以 0

$$\mathbf{y} = \gamma\hat{\mathbf{x}} + \beta$$

學習來的



Layer Normalization (原版 transformers 的選擇)



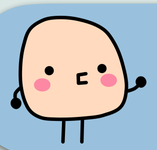
計算平均值、標準差

$$\hat{\mathbf{x}}_\mathbf{i} = \frac{\mathbf{x}_\mathbf{i} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

預防除以 0

$$\mathbf{y}_\mathbf{i} = \gamma \hat{\mathbf{x}}_\mathbf{i} + \beta$$

學習來的



複習一下這些計算

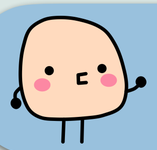
$\mathbf{x}$ x_1 x_2 $\cdots$ x_n

1 計算平均值

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

2 計算標準差

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (x_i - \mu)^2}$$



仔細看 Layer Normalization

$$\mathbf{x} \quad x_1 \quad x_2 \quad \cdots \quad x_n$$

這是向量

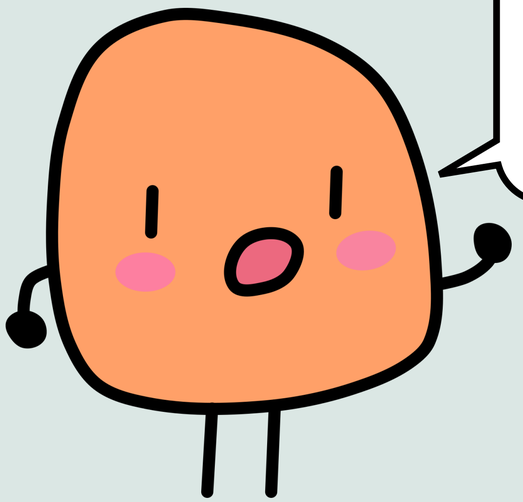
這是純量

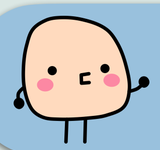
$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

都是向量

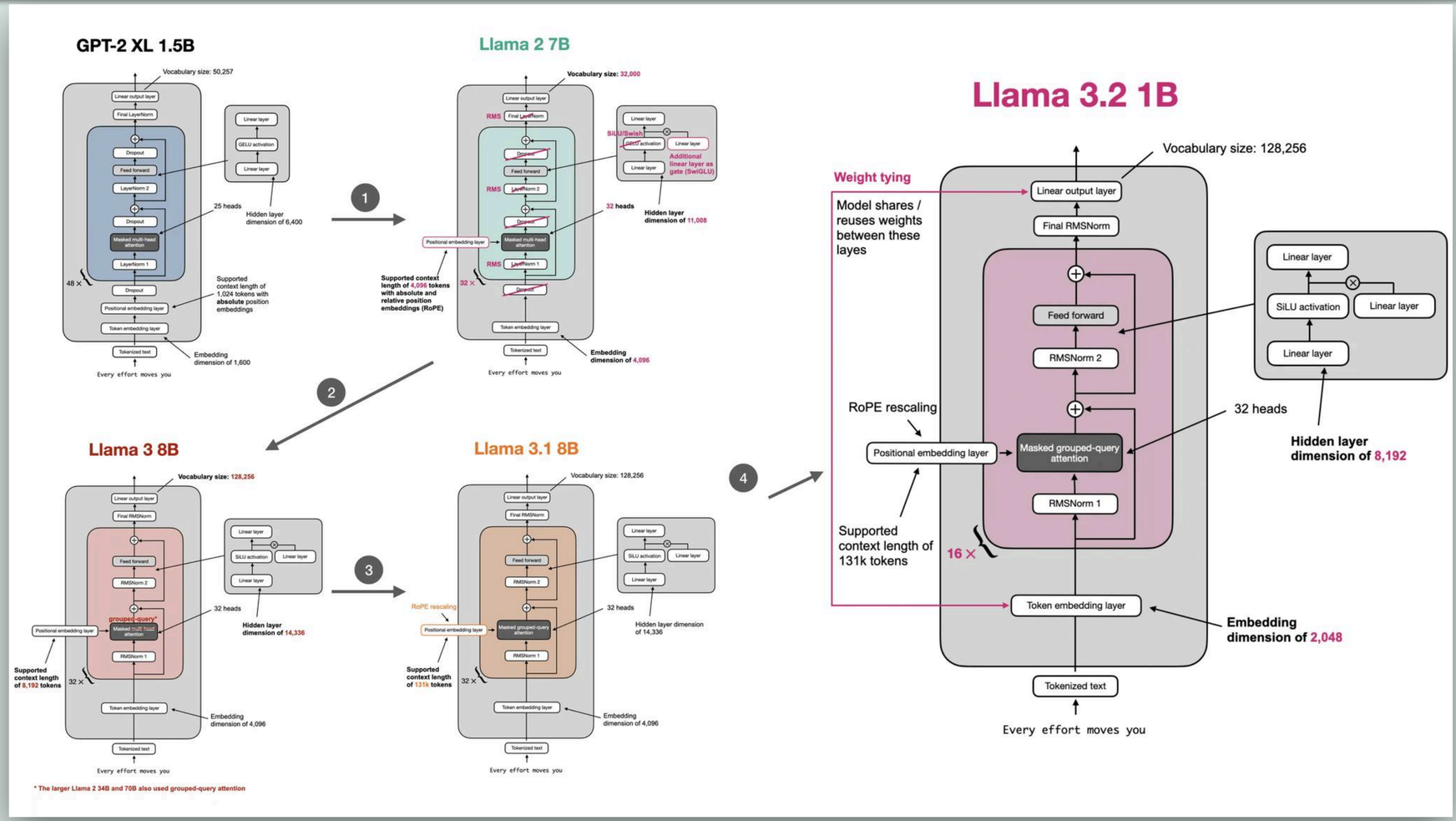
$$\mathbf{y}_i = \gamma \hat{\mathbf{x}} + \beta$$

基本上都是如 numpy 的 broadcasting 運算。

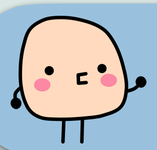




至此更新的模型也知道差異是什麼了



https://github.com/rasbt/LLMs-from-scratch



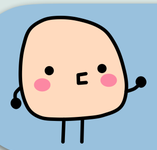
Llama 等 LLM 運用了 RMS Norm

$$\mathbf{x} \quad x_1 \quad x_2 \quad \cdots \quad x_n$$

$$\mathrm{RMS}(\mathbf{x}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x^2}$$

$$\mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\mathrm{RMS}(\mathbf{x})} \cdot \gamma$$

學來的向量



最近 Meta 說不用做 Normalization!

Transformers without Normalization

Jiachen Zhu[1,2], **Xinlei Chen**[1], **Kaiming He**[3], **Yann LeCun**[1,2], **Zhuang Liu**[1,4,†]

[1]FAIR, Meta, [2]New York University, [3]MIT, [4]Princeton University
[†]Project lead

Normalization layers are ubiquitous in modern neural networks and have long been considered essential. This work demonstrates that Transformers without normalization can achieve the same or better performance using a remarkably simple technique. We introduce Dynamic Tanh (DyT), an element-wise operation $\mathrm{DyT}(\boldsymbol{x}) = \tanh(\alpha \boldsymbol{x})$, as a drop-in replacement for normalization layers in Transformers. DyT is inspired by the observation that layer normalization in Transformers often produces tanh-like, S-shaped input-output mappings. By incorporating DyT, Transformers without normalization can match or exceed the performance of their normalized counterparts, mostly without hyperparameter tuning. We validate the effectiveness of Transformers with DyT across diverse settings, ranging from recognition to generation, supervised to self-supervised learning, and computer vision to language models. These findings challenge the conventional understanding that normalization layers are indispensable in modern neural networks, and offer new insights into their role in deep networks.

Date: March 14, 2025
Project page and code: jiachenzhu.github.io/DyT
Correspondence: jiachen.zhu@nyu.edu, zhuangl@princeton.edu

Meta

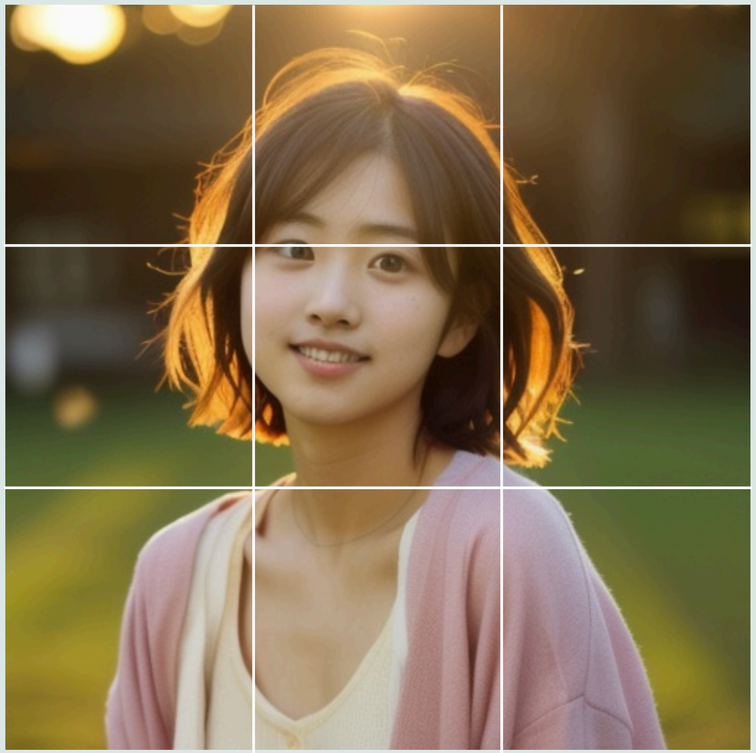
Dynamic Tanh

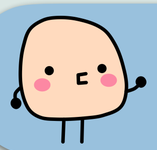
$$\mathrm{DyT}(\mathbf{x}) = \gamma \cdot \tanh(\alpha \mathbf{x}) + \beta$$

可設為 0.5

學習來的向量

不只是文字, 圖像 (聲音、時間序列數據) 也可以





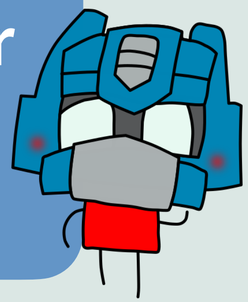
「混入」資訊也很好用

比如文字生圖的 AI, 是怎麼把文字的意思混進去的呢?

隨機產生「天馬行空」的想法。

Q

Transformer



Z

混入文字的意涵。

代表文字「意思」的矩陣。

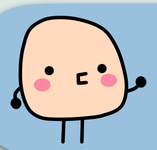
$X \rightarrow K, V$

【作業】

探索並應用一個提示工程技巧

探索並應用一個提示工程（Prompt Engineering）技巧

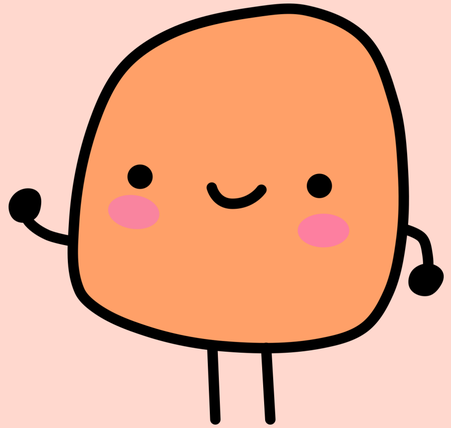
請上網搜尋，找到一個 LLM 提示工程的技巧，並且想一個有趣的、實用的例子。



記得我們說過 prompt 要做到的兩件事

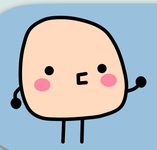
正確的資訊

要提供足夠多的正確資訊。



清楚的指引

要清楚的告訴 LLM 要做什麼。例如, 以上面的資訊, 用什麼樣的格式、風格, 來回答使用者的問題。

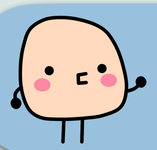


作業需要包括三個部份

① **技巧介紹：**簡單說明這個技巧是什麼，來源可以是文章、影片或論壇（請註明來源）。

② **兩件事連結：**解釋這個技巧如何幫助做到「提供足夠多正確資訊」及「清楚告訴 LLM 要做什麼」。

③ **應用範例：**設計一個有趣或實用的 prompt（可以和課業、日常生活、興趣相關），展示這個技巧的用法，並附上 LLM 的回覆範例。



Q & A