

看图学大模型

看图学

2024 年 6 月 23 日

当前版本为 v0.20240529, 还在更新中, 目前维持每周更新。

本电子书目前进度完成不到 10%, 可以关注下方公众号回复”看图学大模型”来获取最新版。



看图学

微信扫描二维码, 关注我的公众号

目前 PDF 排版等还略有问题，Latex 还需要略微调整。

目录

Chapter 1: 大模型发展史	3
Chapter 2: 主流大模型的模型架构和细节分析	3
Transformers 起源	3
Transformers 架构	17
Position Embedding	27
Transformers 面试八股	33
Chapter 3: 大语言模型 pipeline	53
Stage 1: Pretrain	53
Stage 2: SFT	53
Stage 3: Alignment	53
Stage 0: 数据处理	53
Chapter 4: 大语言模型训练	53
显卡和模型训练基础知识	53
多卡并行训练	70
当前流行训练框架	70
Chapter 5: 大语言模型推理	71
KV Cache	71
Chapter 6: MOE, 多模态等	75
Chapter 7: 大语言模型评估	75
Chapter 8: 大语言模型应用	75
Prompt Engineering	75
Agent	75
Chapter 1: 大模型发展史	75
AI 的起源	75
符号派、链接派，相爱又相杀	90
从 one-hot 到 ChatGPT	90

Chapter 1: 大模型发展史

这一章的内容以故事为主，挺长的，耽误大家学习。喜欢看故事的可以看最后一章。

Chapter 2: 主流大模型的模型架构和细节分析

Transformers 起源

要真正搞懂 Transformers 光看《Attention is All you Need》估计还远远不够，因为这篇已经是站在很多巨人的肩膀上了，其中的很多设计都有历史渊源。所以不要想着直接吃第 7 个馒头就饱了。

今天就来聊聊 Transformers 的进化史，相信你看完之后对现在 Transformers 的架构会有更深刻的理解。

Transformers 的进化史同样是神经网络机器翻译 (NMT) 的发展史。在攻克机器翻译这个难题的过程中，模型的框架经过了多次迭代变成了今天的 Transformers。所以先讲一下机器翻译的背景。

神经网络机器翻译

机器翻译干的事情，就是利用机器学习将 A 语言的一句话 $\mathbf{x} = \{x_1, x_2, \dots, x_S\}$ 翻译为 B 语言的一句话 $\mathbf{y} = \{y_1, y_2, \dots, y_T\}$

神经网络机器翻译 (NMT) 建模为：

$$P(\mathbf{y}|\mathbf{x}) = \prod_{t=1}^T P(y_t|y_0, y_1, \dots, y_{t-1}, \mathbf{x})$$

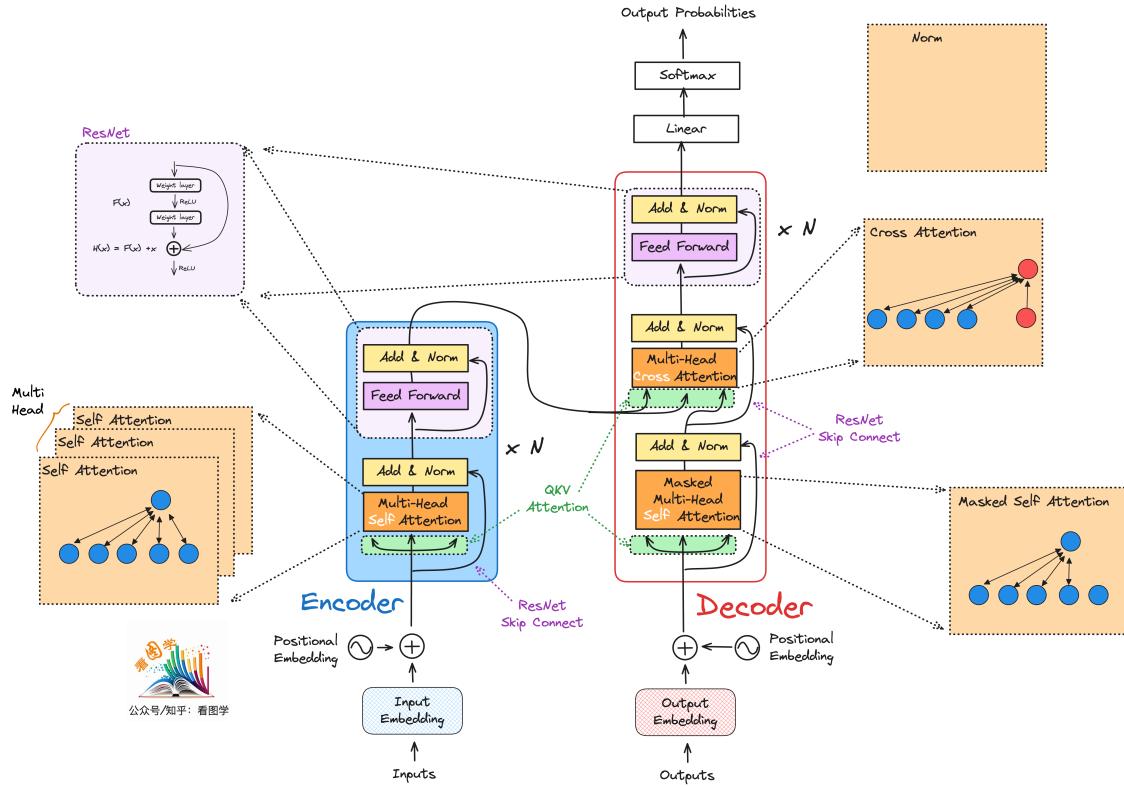
这个建模方式被提出来之后，基本上没什么变化，统计机器翻译 (SMT) 也是一样的，但是模型的架构演进了很多版本。

Transformers 的进化（或者 NMT 的进化）大概经历了如下几个重要的节点。

RCTM -> RNN Encoder-Decoder -> Bahdanau Attention -> Luong Attention -> Self Attention/Multihead Attention -> QKV Attention -> Transformers

当然还加入了已经成为神经网络基建的 ResNet。

当前架构每个地方设计的来源，见下图：



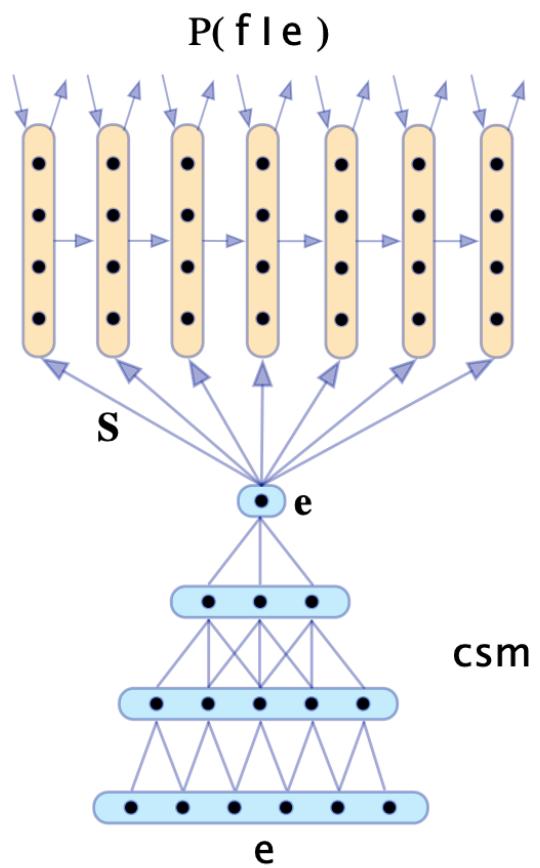
我们按时间顺序来看。

RCTM (Kalchbrenner and Blunsom, 2013)

这是牛津大学在 2013 年提出的一篇论文，这篇文章被认为是神经网络机器翻译 (NMT) 的开篇之作。

这篇论文已经有 Encoder-Decoder 的框架，但是并没有给这个框架命名。文中使用了 CNN 来将源文本“表示”为连续向量，现在通常被称作 Context，就是所谓的 Encoder。然后将连续向量输入给 RNN，RNN 作为 Decoder 去拟合目标句子。

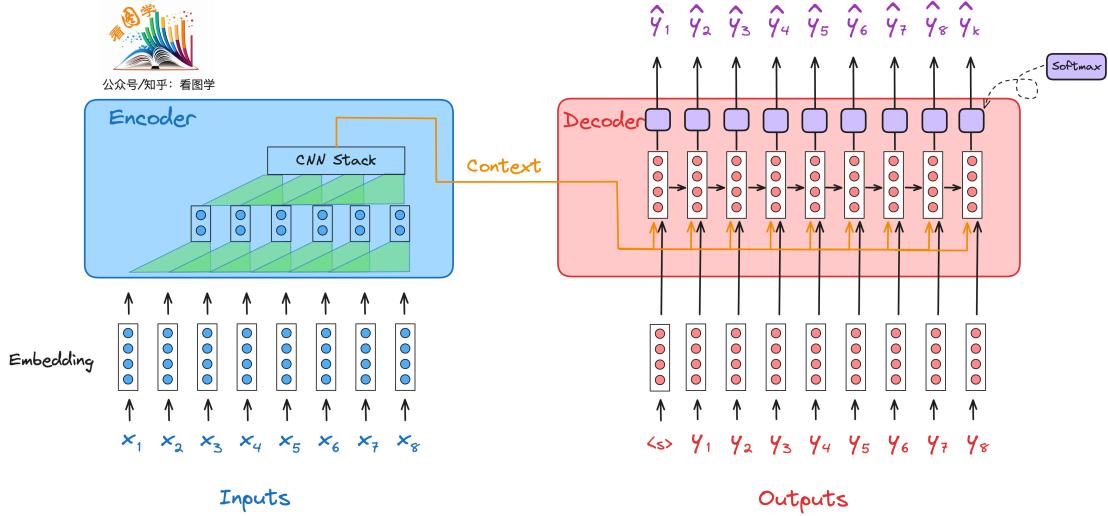
论文中的架构长这样：



RCTM I

转换成现在常见的样子是这样:

RCTM 架构

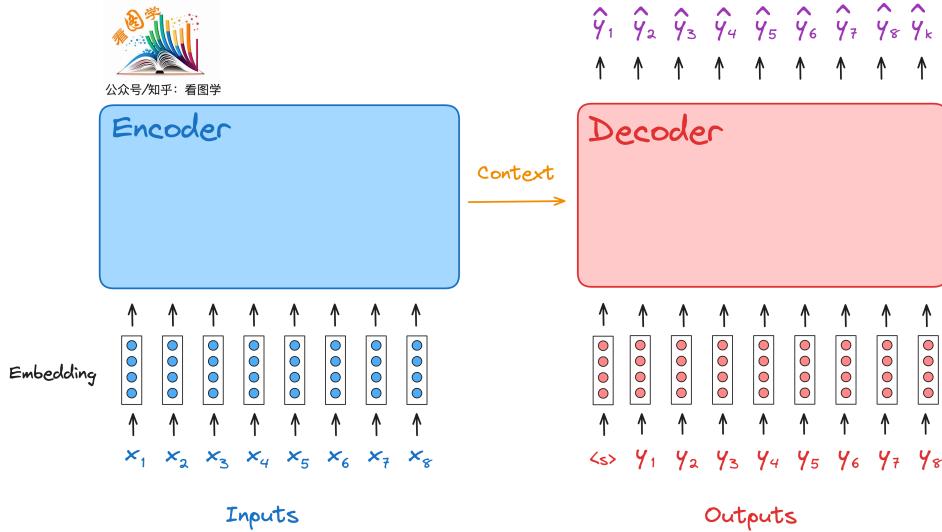


可以看出，这篇论文已经有 Encoder-Decoder 的样子了，只不过左边用了 CNN。之所以使用 CNN 是因为大家那个时候都是搞统计机器学习，n-gram 是无论如何都绕不开的，CNN 的 kernel 大小刚好跟 n-gram 有点像。估计作者这个时候也不敢步子迈的太大，所以采用了 CNN 作为 Encoder，RNN 作为 Decoder。

还有一点要注意的是 Context 参与了 Decoder 的每一步计算。早起大家都是这么干的，后面变成了 Context 只作为 Decoder 的初始状态向量。

该论文的顶层架构设计一直沿用至今，包括 Transformers。虽然内部可能略有差异，但是 框架一直是下面这个样子。为了表述的方便，后面 Encoder 的图示用蓝色表示，其内部的隐向量为蓝色的 h_j ，下标用 j 表示。Decoder 则用红色表示，其内部隐向量为红色的 s_i ，下标用 i 表示。

Encoder-Decoder 的通用架构



这篇神经网络机器翻译的开山之作当时取得了很好的效果，但依然存在一些问题。比如原生的 RNN 因

为梯度爆炸和消失对长句子无能为力。

1. 进化路径

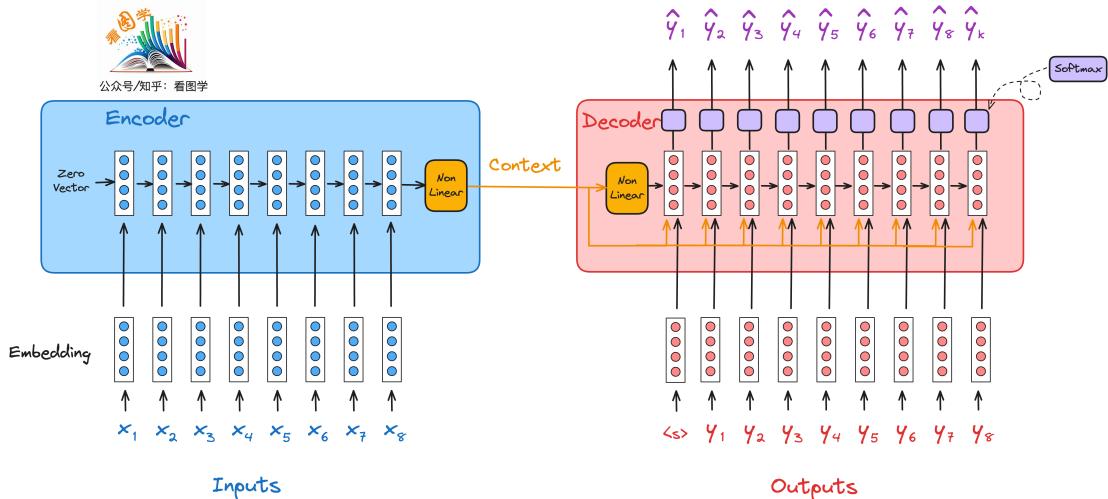
- 神经网络机器翻译的初号机。
- 实现了 Encoder-Decoder 架构。
- Encoder 提取的 Context 参与了所有目标字符串的预测。
- Decoder 用来预测下一个字符。
- 这是个 end-to-end 的神经网络模型。

RNN Encoder-Decoder (Cho et al., 2014a)

这篇论文和下面的 Bahdanau Attention 都是 Bengio 团队的成果，Bahdanau 是这篇论文的二作。

这篇论文首次明确提出了 Encoder 和 Decoder 的概念。但是整体的架构和 RCTM 基本一样，只不过是 Encoder 也变成了 RNN。可能是觉得这么做学术创新点不太够，所以本文另外还提出了和 LSTM 齐名的 GRU 架构。

RNN Encoder-Decoder 架构



从上图可以看出，和 RCTM 一样，Encoder 编码后的隐向量被用到了 Decoder 的每一个 step。
用公式来描述为：

- Encoder

$$h_0 = \mathbf{0}$$

$$h_j = \text{RNN}_{GRU}(h_{j-1}, x_j)$$

$$C = \tanh(V \cdot h_T)$$

- Decoder

$$s_0 = \tanh(V' \cdot C)$$

$$s_i = \text{RNN}_{GRU}(s_{i-1}, [y_i; C])$$

$$= \text{softmax}(\text{MLP}(s_i))$$

可惜的是这个 Encoder-Decoder 的架构并没有做成一个端到端的模型，只是把 Decoder 的输出概率作为特征喂给了统计机器翻译 (SMT) 模型。

有时候看论文确实是有这么一种感觉，总感觉离更好的答案就差一口气了，但是很可惜，这一口气就是没喘上来。这也正是科研的艰难之处，筚路褴褛。就像在山洞里挖出口，可能再挖一铲子就打通了，但是在挖下去之前，谁也不知道挖的方向对不对，但是研究经费可能不够了。要知道香农作为信息论的创始人，自己都没有找出最佳编码算法，反而是几年后被霍夫曼给想出来了。

1. 进化路径

- Encoder 采用了 RNN
- 为了解决长文本的梯度消失/爆炸，采用了 GRU 架构
- 开倒车了，从 end-to-end 变成了特征提取器。

Bahdanau Attention (Bahdanau et al., 2014)

这篇论文也是 Bengio 团队的成果，是对前一篇 RNN Encoder-Decoder 论文工作的延续。

这篇文章中，提出了 Attention 的概念。作者认为在翻译过程中，两种语言中相同含义的词或者段落应该能对齐。比如“How(多) old(大) are you(你)?”，就是一个中英文的对齐。神经网络理应学习到这个知识，但是目前好像并没有学到。所以作者就设计了一个网络来计算输入输出两种语言之间不同词之间的关联分数，也就是 Attention Score。

等一下，我设计一个全连接网络是不是也能计算输入输出的关联？和 Attention 有什么区别？

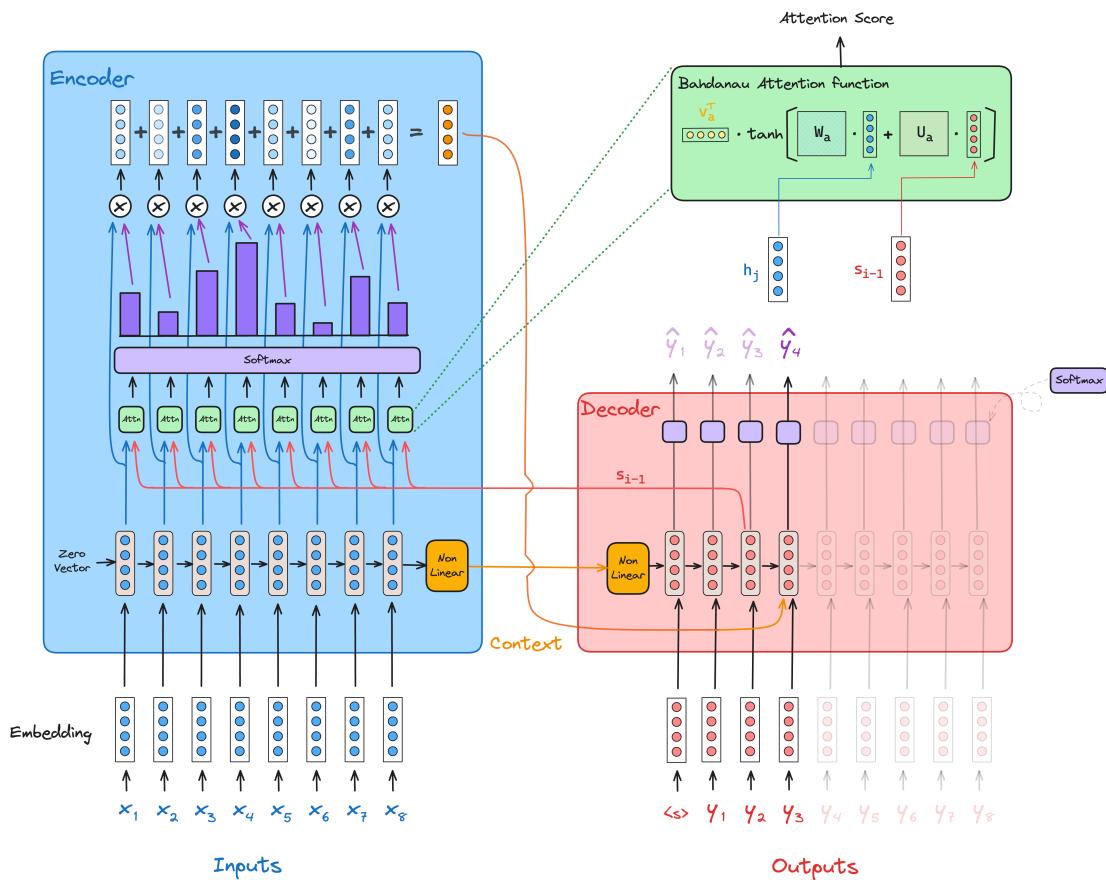
这两个其实是一静一动。全连接网络训练完成之后，权重就固定了，不能变了。但是 Attention 则可以动态的，根据输入来计算出输出应该更关注哪些输入。

Bahdanau Attention 的思路如下。

Bahdanau Attention 架构



means RNN Cell means vector



- Encoder

$$h_0 = \mathbf{0}$$

$$h_j = \begin{bmatrix} \vec{h_j} \\ \underline{h_j} \end{bmatrix}$$

$$h_j = \text{RNN}_{GRU}(h_{j-1}, x_j)$$

$$C = \tanh(V \cdot h_T)$$

- Decoder

$$s_0 = \tanh(V' \cdot \overleftarrow{h_1})$$

$$e_{ij} = v_a^\top \tanh(W_a s_{i-1} + U_a h_j)$$

$$\alpha_{ij} = \frac{e_{ij}}{\sum_{k=1}^T e_{ik}}, \alpha_{ij} \text{ is a scalar}$$

$$c_i = \sum_{j=1}^T \alpha_{ij} h_j$$

$$s_i = \text{Bi-RNN}_{GRU}(s_{i-1}, [y_i; c_i])$$

$$= \text{softmax}(\text{MLP}(s_i))$$

- 需要注意的是原论文计算词概率的时候并不是简单的 $\hat{y}_i = \text{softmax}(\text{MLP}(s_i))$, 而是在 GRU 后再加了两层网络实现了一个 maxout。这里为了方便和其他模型对比简化了公式。

1. 进化路径

- 在 Encoder-Decoder 的基础上增加了 Attention 机制。
- 终于变成了 End-to-End 模型。
- Encoder 的 context 不再作为 Decoder 的每一个 step 的输入。

Seq2Seq (Sutskever et al., 2014)

这篇是 Ilya 的论文, 当时还是很轰动的, 因为是 Google 的论文, 再一个效果确实很好。但是实际上就是将 RNN Encoder-Decoder 做成了端到端, 模型结构非常简单。然后堆叠 LSTM 的参数量取得了不错的效果。当然也有些 Trick, 比如逆序输入能够提升效果, 个人猜测可能是一句话重要的部分往往放在前面的原因。逆序后 Encoder 和 Decoder 开头的部分离的近, 信息衰减变小了。

有意思的是 seq2seq 和 Bahdanau Attention 的论文几乎是同时写的, 所以这两篇论文相互引用了。所以只将 Encoder Context 作为 Decoder 的初始化参数可能影响了 Bahdanau Attention 的结构设计, 但是也没多大影响, 因为 Bahdanau Attention 相当于 Encoder 的所有 context 都是用了。

然后 seq2seq 的效果比 Bahdanau Attention 要好, 主要原因也是参数量更大, 这么比较有点不太公平。这也符合 Ilya 的一贯思路, 大就是好, 我不弄那些乱七八糟的, 简单的结构的大量堆叠就能有很好的效果。只不过这个时候他堆叠的是 lstm, 后来等 transformers 出来后就去堆叠 transforemrs 了。

1. 进化路径

- 可能影响了 Bahdanau Attention 的结构设计。
- 对 Transformers 进化影响不大。

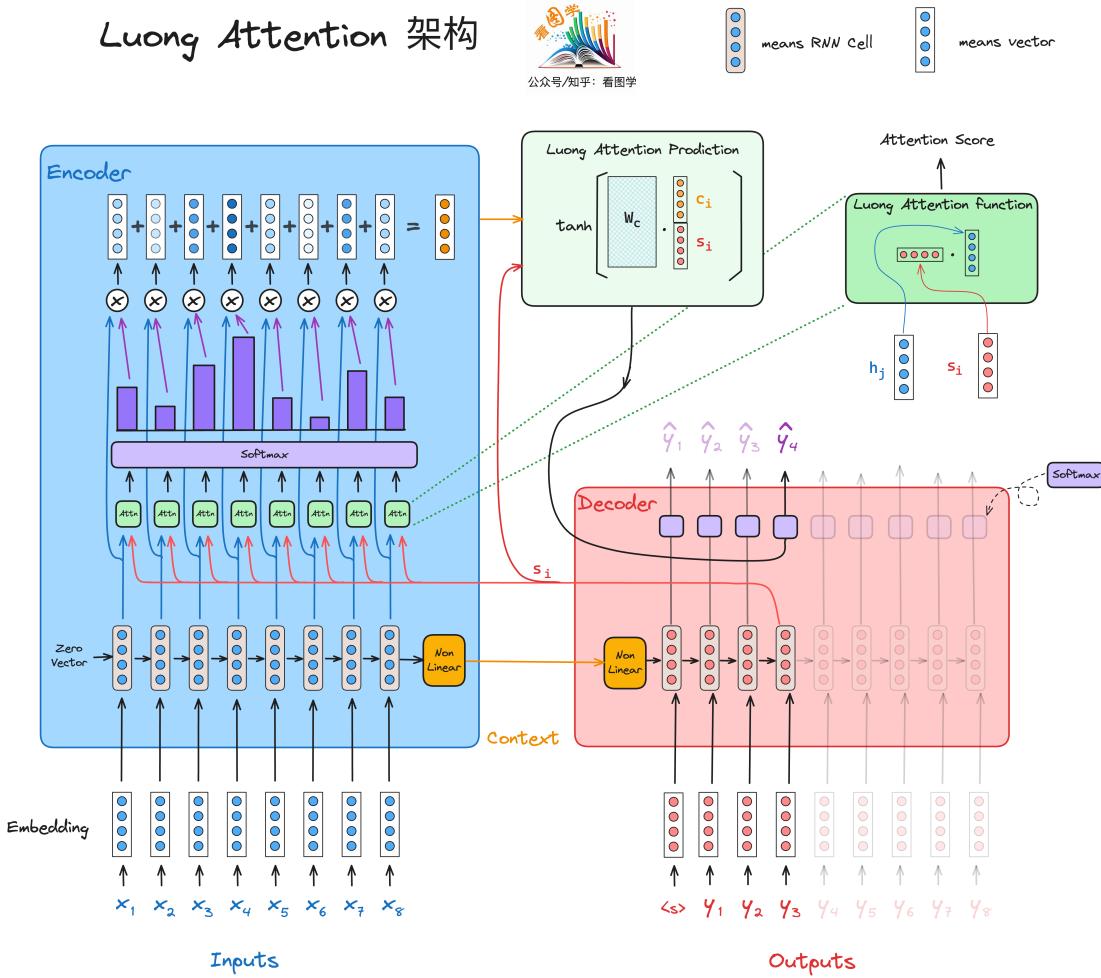
2. Seq2Seq vs. Encoder-Decoder:

- Seq2Seq but not Encoder-Decoder: RNN, HMM, GPT
- not Seq2Seq but Encoder-Decoder: VAE
- Seq2Seq and Encoder-Decoder: Transfomes

Luong Attention (Luong et al., 2015).

Bahdanau Attention 计算 Attention Score 的方法是相加。Luong Attention 则尝试了更多方法，比如点积。

结构上略有修改。



1. 进化路径

- Attention 的计算方法多样性探索。

Self Attention

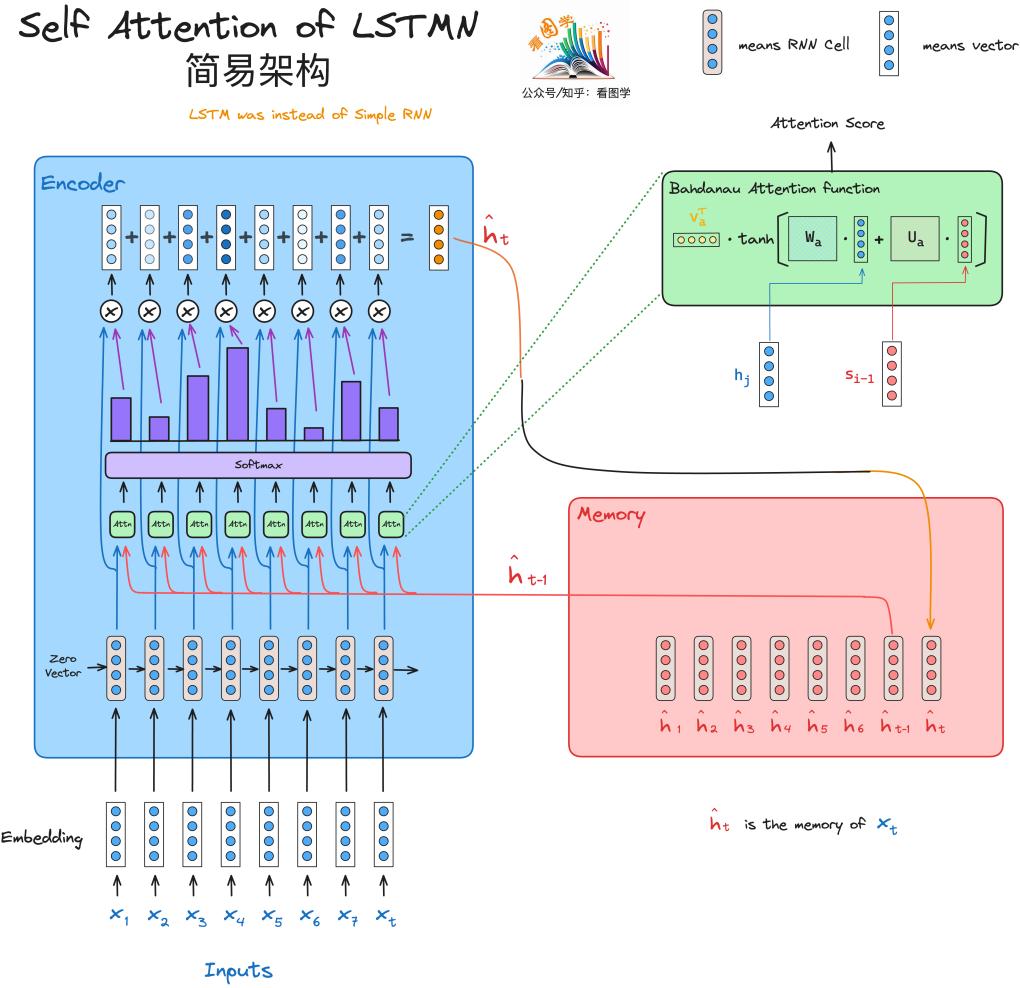
之前的 Attention 关注的都是不同序列之间的 Attention，比如机器翻译的原始文本和目标文本，文生图中的文本序列和图像序列。这种 Attention 称作 Cross Attention，Cross Attention 可以说是 Encoder-Decoder 或者 seq2seq 中使用 Attention 的标准结构。

这篇论文将 Attention 的思想借鉴到了机器阅读理解中。作者认为我们在顺序的阅读每个单词的时候，每个单词和前面单词的关联度是不一样的，如下图所示。其实之前也有很多类似的工作，比如依存分析。这个工作是否可以让 Attention 来做呢？

作者提出的框架，整体上沿用了 Bahdanau Attention 的架构，但是 Bahdanau Attention 计算分数是 Encoder 和 Decoder 之间的。这里只有 Encoder，作者就给每个输入构建了一个 Memory State。下图为了和 Attention 对比，将 Memory State 放到了 Decoder 的位置，但是要记住 Encoder 和 Memory 两个模块是同步更新的。

下图做了两点简化：

1. LSTM 简化为 Simple RNN，不然画起来有点复杂。
2. Attention 的计算少了词向量部分。论文中在计算 Attention 的时候，词向量也参与了计算。



将上图的 Simple RNN Cell 替换为 LSTM，同时计算 Attention 的时候加入词向量，就是论文的最终架构。需要注意，LSTM 中的 Carry Memory 的计算也采用了 Attention 的计算方式。

具体的计算公式为：

- Encoder

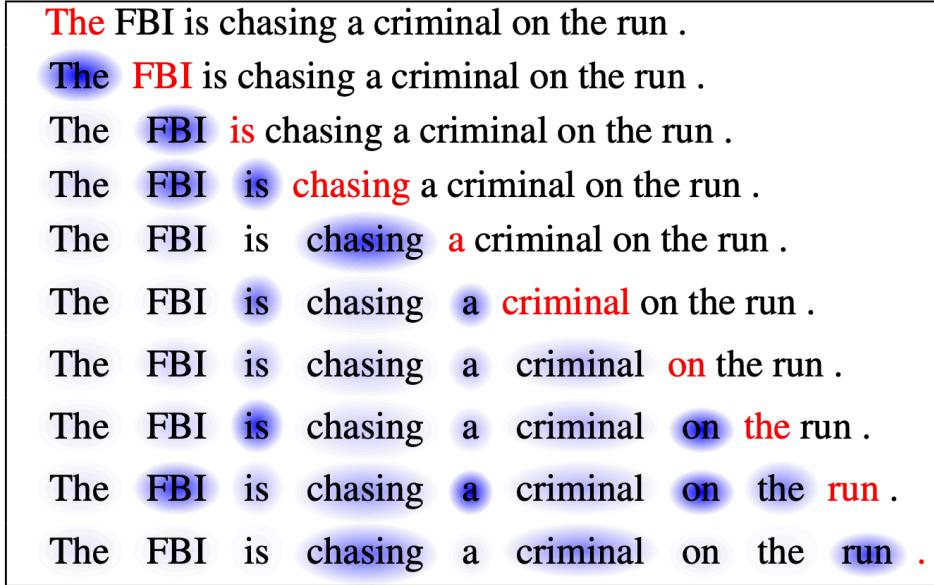
$$e_{ij} = v_a^\top \tanh(W_a \tilde{h}_{i-1} + U_a h_j + W_x x_i)$$

$$\alpha_{ij} = \frac{e_{ij}}{\sum_{k=1}^T e_{ik}}, \alpha_{ij} \text{ is a scalar}$$

$$\begin{bmatrix} \tilde{c}_i \\ \tilde{h}_i \end{bmatrix} = \sum_{j=1}^{i-1} \alpha_{ij} \begin{bmatrix} c_j \\ h_j \end{bmatrix}$$

$$\begin{bmatrix} c_i \\ h_i \end{bmatrix} = \text{LSTM}\left(\begin{bmatrix} \tilde{c}_i \\ \tilde{h}_i \end{bmatrix}, x_i\right)$$

关于 LSTM 的知识, 这里不再赘述。LSTM 的知识见:https://mp.weixin.qq.com/s/Yyf_8VlSS6VE0wKwsrkAww
 最后, 由于本论文只关注前面词的 Attention, 所以其实是 Masked Self Attention。
 最终效果的可视化结果如下:



1. 进化路径

- 在 cross attention 的基础上, 提出了 self-attention, 或者叫 intra-attention.

MultiHead Self Attention (Lin et al., 2017)

这篇论文将 Attention 的思想借鉴到了文本表示学习中。其出发点也很简单, 我们读一段文本自己有时候会划重点, 或者圈关键词, 这个工作能不能交给 Attention 呢?

作者同样参考了 Bahdanau Attention, 只不过在计算 attention score 的时候, 输入从 h_j 、 s_i 改成里双向 LSTM 的两个方向的 hidden states。

- Encoder

$$\begin{aligned}
\overrightarrow{h_t} &= R_{LSTM}(\mathbf{w}_t, \overrightarrow{h_{t-1}}) \in \mathbb{R}^u \\
\overleftarrow{h_t} &= R_{LSTM}(\mathbf{w}_t, \overleftarrow{h_{t-1}}) \in \mathbb{R}^u \\
h_t &= \begin{bmatrix} \overrightarrow{h_t} \\ \overleftarrow{h_t} \end{bmatrix} \in \mathbb{R}^{2u} \\
&\quad =
\end{aligned}$$

$$\begin{aligned}
e_{ij} &= w_{s2}^\top \tanh(W_{s1} h_j) \\
w_{s2}^\top \tanh(\begin{bmatrix} W_{s1}[:, u] & W_{s1}[u, :] \end{bmatrix} \begin{bmatrix} \overrightarrow{h_j} \\ \overleftarrow{h_j} \end{bmatrix}) \\
&= w_{s2}^\top \tanh(W_{s1}[:, u] \cdot \overleftarrow{h_j} + W_{s1}[u, :] \cdot \overrightarrow{h_j})
\end{aligned}$$

$$\begin{aligned}
\alpha_{ij} &= \frac{e_{ij}}{\sum_{k=1}^T e_{ik}}, \alpha_{ij} \text{ is a scalar} \\
m_i &= \textcolor{brown}{c_i} = \sum_{j=1}^T \alpha_{ij} \textcolor{blue}{h_j}
\end{aligned}$$

可以看出，Self Attention 计算 Score 的设计明显是参考了 Bahdanau Attention 的计算方法。

上面的计算是单个 Attention 的计算方法。然后借鉴一下卷积网络的多个 kernel 的思想，作者设计了多个 Attention，期望不同的 Attention 学习到不同的注意力，从而更好的提取特征。这种一个不行就上多个到方法在深度学习中经常使用，比如 CNN 多 kernel，MultiHead Attention，Mixture of Experts 等。

然后作者提出了 Attention 的矩阵运算形式。

$$H = (h_1, h_2, \dots, h_n) \in \mathbb{R}^{n \times 2u}$$

$$W_{s1} \in \mathbb{R}^{d \times 2u}$$

$$W_{s2} \in \mathbb{R}^{r \times d}$$

$$E = W_{s2} \tanh(W_{s1} \cdot H^\top) \in \mathbb{R}^{r \times n}$$

$$A = \text{softmax}(E) \in \mathbb{R}^{r \times n}$$

$$M = AH \in \mathbb{R}^{r \times 2u}$$

可以看出，MultiHead 是通过 r 来体现的。然后这个矩阵相乘 AH 已经和 Transformers 最终形态的 Attention $AV = \text{softmax}(\frac{QK^\top}{\sqrt{d}})V$ 有点类似了。

效果也很显著，如下图。从此之后，Self Attention 成为表示学习中的扛把子。

- I really enjoy Ashley and Ami salon she do a great job be friendly and professional I usually get my hair do when I go to MI because of the quality of the highlight and the price the price be very affordable the highlight fantastic thank Ashley i highly recommend you and ill be back
- love this place it really be my favorite restaurant in Charlotte they use charcoal for their grill and you can taste it steak with chimichurri be always perfect Fried yucca cilantro rice pork sandwich and the good tres lech I have had.The desert be all incredible if you do not like it you be a mutant if you will like diabeetus try the Inca Cola
- this place be so much fun I have never go at night because it seem a little too busy for my taste but that just prove how great this restaurant be they have amazing food and the staff definitely remember us every time we be in town I love when a waitress or waiter come over and ask if you want the cab or the Pinot even when there be a rush and the staff be run around like crazy whenever I grab someone they instantly smile acknowledge us the food be also killer I love when everyone know the special and can tell you they have try them all and what they pair well with this be a first last stop whenever we be in Charlotte and I highly recommend them
- great food and good service what else can you ask for everything that I have ever try here have be great
- first off I hardly remember waiter name because its rare you have an unforgettable experience the day I go I be celebrate my birthday and let me say I leave feel extra special our waiter be the best ever Carlos and the staff as well I be with a party of 4 and we order the potato salad shrimp cocktail lobster amongst other thing and boy be the food great the lobster be the good lobster I have ever eat if you eat a dessert I will recommend the cheese cake that be also the good I have ever have it be expensive but so worth every penny I will definitely be back there go again for the second time in a week and it be even good this place be amazing

(b) 5 star reviews

1. 进化路径

- 提出了另一种 self-attention 的方法
- 提出了 Multi Head Attention

FRUSTRATINGLY SHORT ATTENTION SPANS IN NEURAL LANGUAGE MODELING

这篇论文可能是最早提出 query, key, value attention 的论文，但是引用量并不高。

这篇论文认为现有的 Attention，隐向量至少承担了 3 个功能：

1. 计算 attention score
2. 计算 context vector
3. 作为 hidden state 来进行 RNN 的计算。

任务实在有些繁重，所以提出了使用 3 个隐向量来计算 Attention，功能如下：

1. key 用来计算 attention score.
2. value 用来和 Attention score 相乘。
3. predict 用于预测词的分布。

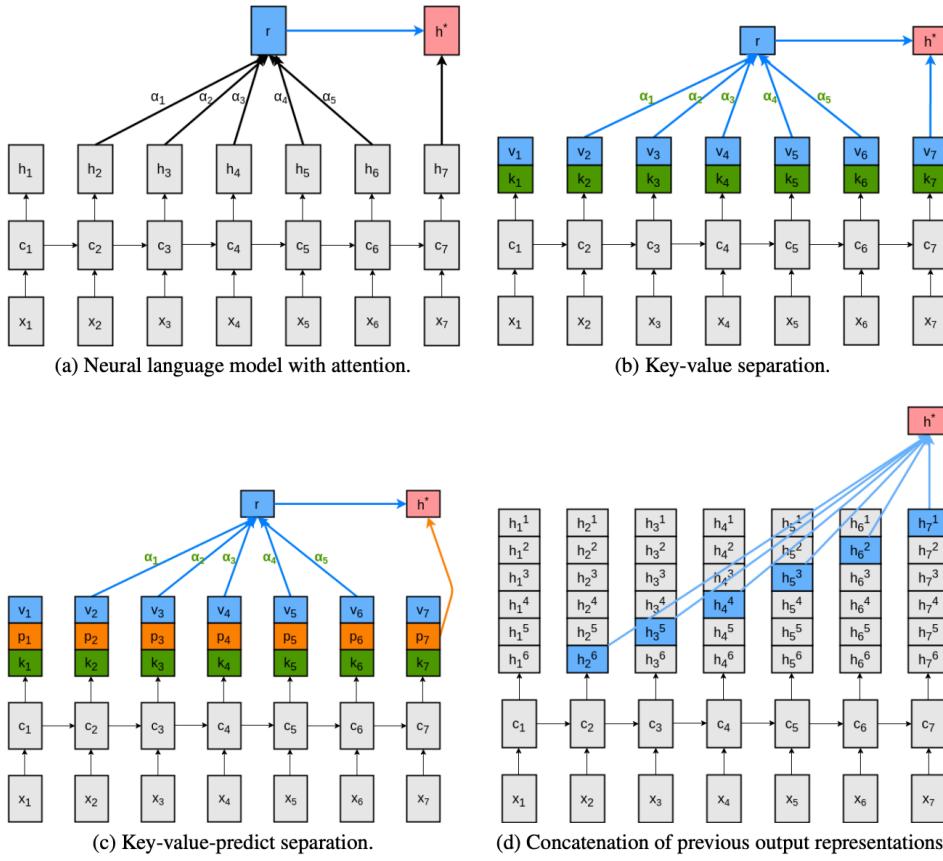


Figure 1: Memory-augmented neural language modelling architectures.

从后面 Transformers 的实现来看, 没有使用 predict, 新增了一个 query , query 和 key 用来计算 Attention Score, value 则用来鹤 Attention score 相乘。

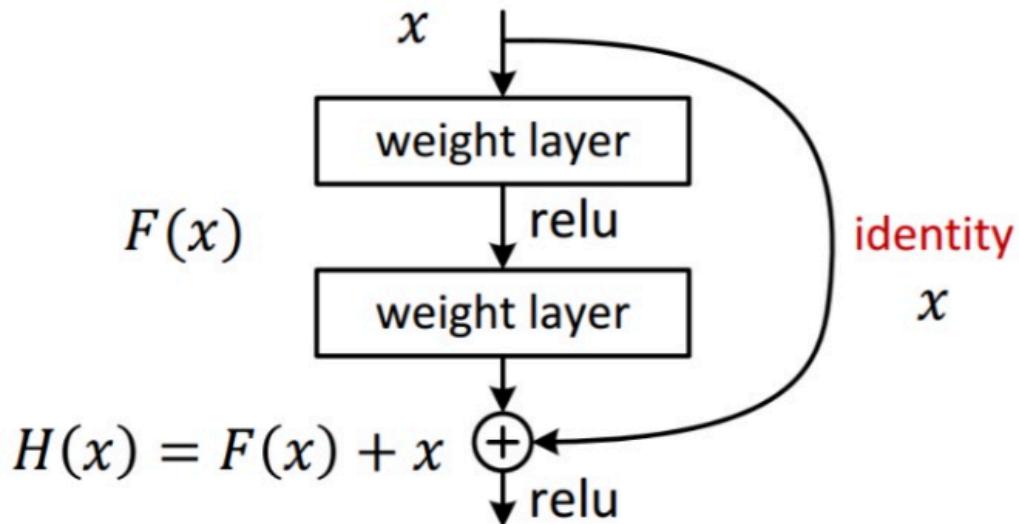
1. 进化路径

- 将隐向量投影为 3 个, 分别为 key, value, predict, 各自有不同的分工。

ResNet

随着神经网络层数增加, 梯度消失/爆炸问题越来越严重, 研究人员想了各种方法, 比如 ReLU, LSTM 等, 都取得了一定的效果, 但是基本到几十层就歇菜了。直到 ResNet 给神经网络修了一条高速公路, 神经网络层数的瓶颈才算突破。天不降生何凯明, DL 万古如长夜:)

• Residual net



关于 ResNet 的解读已经很多，何凯明自己就发了 3 篇相关的文献。分别为：

1. Deep Residual Learning for Image Recognition
2. Identity mappings in Deep Residual Networks
3. Aggregated Residual Transformation for Deep Neural Networks

这里就重点说一下个人觉得重要的几点：

1. 加入 skip connect 使得信息可以直达最后一层。
2. 再一个是在反向传播的时候， $\frac{\partial f(x)}{\partial x}$ 可能消散，但是 $\frac{\partial(f(x)+x)}{\partial x} = 1 + \frac{\partial f(x)}{\partial x}$ 保证了计算的稳定性。
1. 进化路径
 - 属于基建了。

Transformers

至此，Transformers 大部分零件已经凑齐，Let's Roll Out.

Transformers 架构

至此，Transformers 大部分零件已经凑齐，Let's Roll Out.

Transformers 也是为了机器翻译设计的，回顾一下 Transformers 之前的机器翻译模型，大多还是 RNN Encoder-Decoder 的范式，但是这样也就继承了 RNN 的所有问题。主要问题有两点：

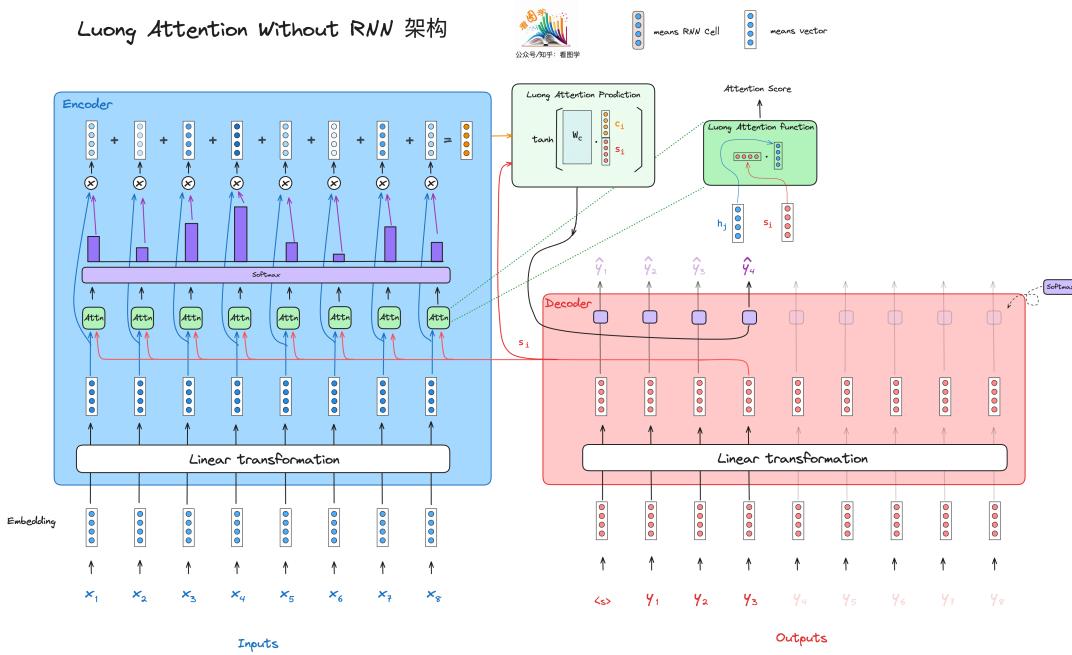
1. 长序列的梯度消失/爆炸问题。
2. 只能一步一步计算，无法并行。

通过堆叠 RNN，扩大参数量确实也取得了一定的效果，比如 Seq2Seq。但是 Bahdanau Attention 出现后，让研究人员看到了另外一种可能。

之前序列中某个节点要获取前面节点的信息，都是通过 RNN 这个中间商搬运过来的，特征提取或者信息压缩的效率么，总是有点差强人意。现在 Attention 来了，前面的节点信息可以厂家直销了，boss 直聘了，还需要 RNN 这个中间商赚差价么？

如果将 Bahdanau/Luong Attention 的 RNN 去掉，会得到了什么？我们来尝试一下。

Luong Attention without RNN



上图中，原来 RNN 的位置替换成了一个线性变换。现在的神经网络很少有将 Word Embedding 直接参与一些网络结构的计算，一般都会先做一个线性变换。这样做一方面是增加了模型可学习的参数，再一个是我们大家都希望 Word Embedding 就专注于学习词的表示。当然也有例外，比如 Word Embedding weight tying 技巧。

上图其实已经有点 Transformers 的影子了，你能找出与 Transformers 结构上的相同之处么？

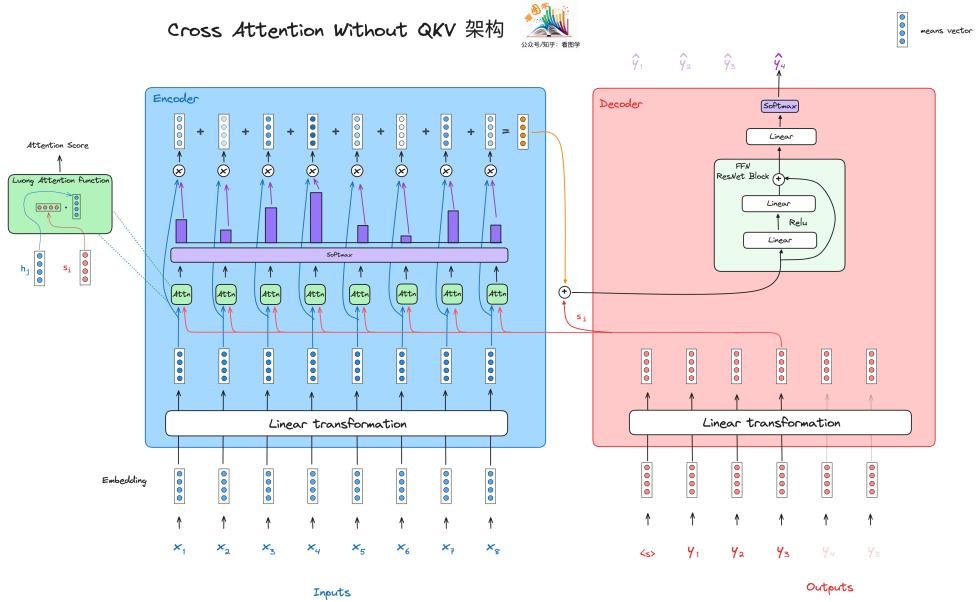
去掉 RNN 后，再也不需要一步一步的计算了，整个架构从原来的序列模型变成了一个全连接图模型，这就很方便的进行矩阵计算，从而享受并行计算或者 GPU 加速带来的运行效率的提升。关于如何实现矩阵计算，请参考后面 Attention 矩阵表示那一章。

去掉了 RNN 也有个明显的问题。那就是变成了全连接图之后，序列的位置信息丢失了。对于 Attention 来说，“八王大”、“八大王”、“王八大”还有“大王八”是一样的。这怎么行，必须让 Attention 知道位置信息，于是研究人员提出了 Position Embedding 的方法，可以认为是全连接图中每个节点带上了位置信息。设计非常巧妙，几句话说不明白，请看后面 Position Embedding 那一章。

修改 FFN 为 ResNet Block

早些年的时候 tanh, sigmoid 等激活函数还经常使用, 但是随着神经网络深度的增加, 计算量有点大。所以后来大多都是用 ReLU, GeLu 等。

Transformers 野心不小, 并不想搭几层就结束了, 所以将上图中 tanh + FFN 换成 ResNet Block 后, 上图变成了:



注意该图在 Attention 前后也加入了 skip connection。

然后论文中给这个 FNN 取名为 Position-wise feed-forward networks, 当时也没太注意这个 Position Wise。后来发现 Position Wise 其实就是常用的 [batch_size, hidden_size] 的矩阵, 在这里变成了 [batch_size, sequence_length, hidden_size] 的矩阵。上面的图就是对 sequence 中的 y_4 进行预测运算。

这里不得不提一点, 虽然论文的名字叫《Attention is All your Need》, 但是实际上, **FFN and ResNet are also your need.**

研究人员发现 FFN 和 ResNet 的 Skip Connection 无论去掉哪一个, 模型都会变得不可用。¹ 所以说 **Attention, FFN, ResNet** 可以认为是 **Transformers** 架构的三驾马车, 缺一不可。

假设只用了 Attention, 仔细看一下 Attention 的计算公式, 虽然其中有一个 softmax 的非线性运算, 但是对于 value 来说, 并没有任何的非线性变换。所以每一次 Attention 的计算相当于是对 value 代表的向量进行了加权平均, 即使在上面堆叠多个 Self Attention, 依然只是对 value 向量的加权平均而已。一层和多层没有本质的区别。这就是 FFN 必须要存在的原因, 或者说更本质的原因是因为 FFN 提供了最简单的非线性变换。

假设 $\alpha_{(i)} = \text{softmax}(q_{(i)} k_{(i)}^T)$, 表示层 i 的 attention score.

¹ Attention is Not All You Need: Pure Attention Loses Rank Doubly Exponentially with Depth

$$\begin{aligned}
v_{(1)} &= xW_{(1)} \\
x_{(1)} &= \alpha_{(1)}v_{(1)} \\
v_{(2)} &= x_{(1)}W_{(2)} \\
x_{(2)} &= \alpha_{(2)}v_{(2)} \\
&= \alpha_{(2)}x_{(1)}W_{(2)} \\
&= \alpha_{(2)}\alpha_{(1)}xW_{(1)}W_{(2)} \\
x_{(3)} &= \alpha_{(3)}v_{(3)} \\
&= \alpha_{(3)}x_{(2)}W_{(3)} \\
&= \alpha_{(3)}\alpha_{(2)}\alpha_{(1)}xW_{(1)}W_{(2)}W_{(3)} \\
&\dots \\
x_{(i)} &= \alpha_{(i)}\alpha_{(i-1)}\dots\alpha_{(1)}xW_{(1)}W_{(2)}W_{(3)}\dots W_{(i)}
\end{aligned}$$

通过上面的公式可以看出，无论堆叠多少层，都是最开始输入 x 的一个线性变换。线性变换无法处理一些非线性的特征，恰如当年马文明斯基给神经网络判的死刑，只需要加个非线性变换的激活函数就能起死回生。

Attention, FFN, ResNet 缺一不可但却各司其职，我个人的观点（并不一定准确）是，Attention 的功能是做信息的提取和聚合，Resnet 提供信息带宽，而真正学到的知识或者信息都存储在 FFN 中。在图像领域中，也有一种说法，那就是 Attention 其实是 token mixer, FNN 其实是 channel mixer. 论文中对 FNN 的解释是可以看作用 1×1 的卷积核来进行特征的升维和降维，解释的比较浅显。

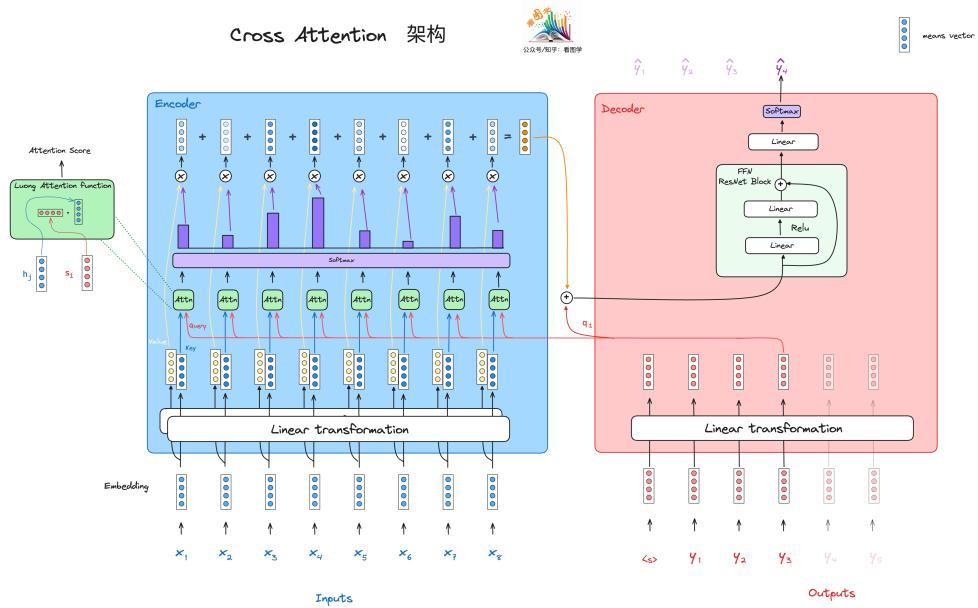
后续虽然有很多 Transformers 架构的魔改，但是真正取得不错的效果且落地的一个例子就是修改 FFN 的 MoE 架构。剩下的两辆马车，ResNet 几乎没法修改了，Attention 则在效率和效果之间寻求平衡。

关于 FFN 的作用后面专门有一章来说明，这里先稍为提一下。

Cross Attention 加上 QKV Attention

如之前论文所述，同一个 hidden state 承担了太多的职责，所以引申出了 query, key 和 value. 在 cross attention 中，decoder 的红线就是 query, key 和 value 则在 Encoder 一侧。我们给 Encoder 上增加 key 和 value。

图画到这里，应该已经可以发现，上图已经是部分 Transformers 了。对应 Transformers 架构中去除 Self Attention 的部分，如下图所示。



TODO: 上面的图中少了一个输出的线性变换。

剩余的部分就是 Encoder 和 Decoder 的 Self-Attention。

Transformers' Scaled Dot-Product Self Attention

在上面的结构中，输入的 Embedding 其实只做了一次线性变换，特征提取能力或者表示学习的能力及其有限。

前文也提到，Self Attention 在表示学习方面遥遥领先，所以 Transformers 自然也想把 Self Attention 加进去。论文从三个维度比较了当时特征提取的主流框架。这三个维度分别为：

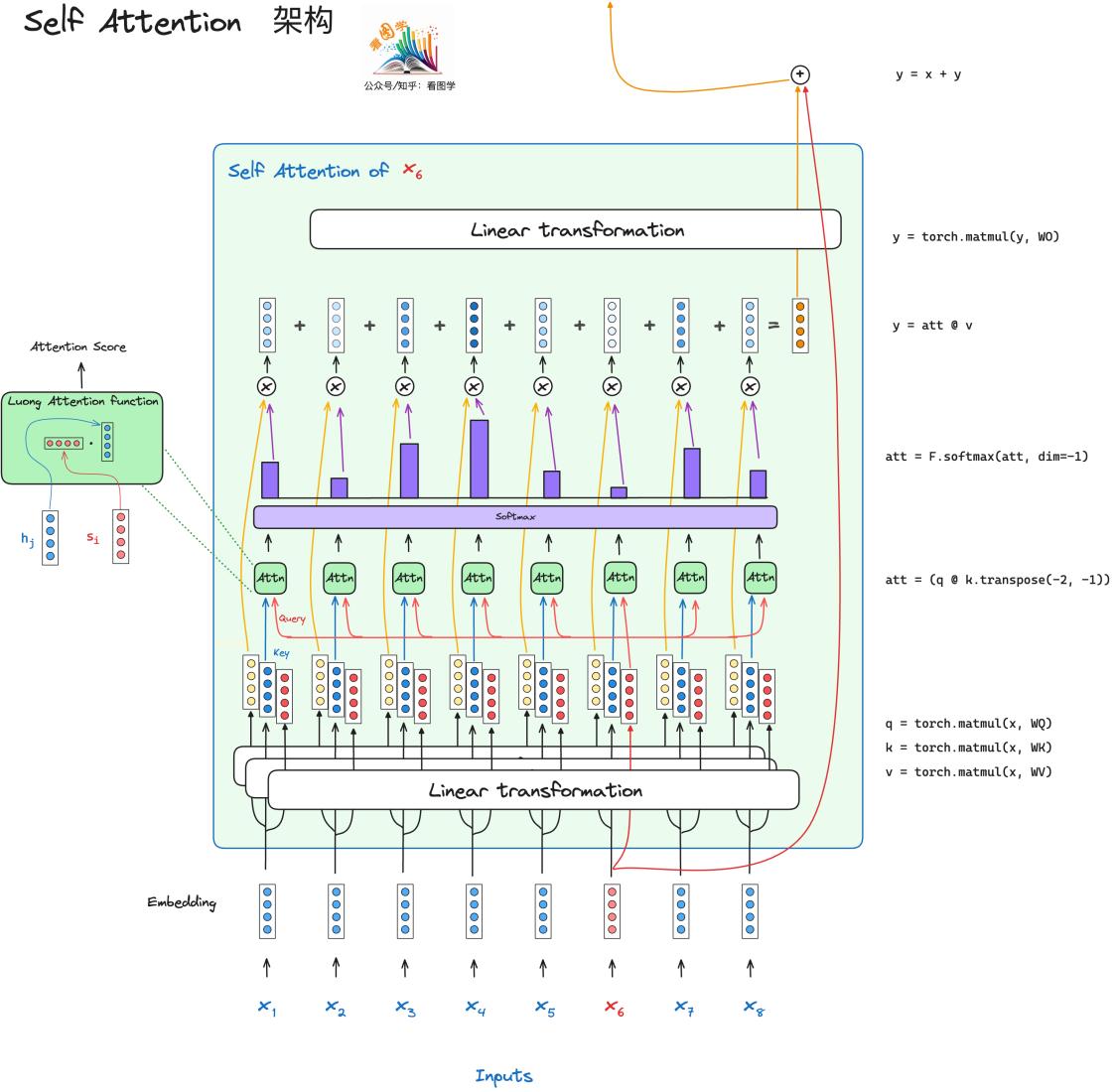
1. 计算的时间复杂度，越小越好。
2. 并行效率。这里用长度为 n 的寻猎计算完成所需要的步数来表示。如果为 1 则表示一步就可以完成，
并行度最高。RNN 则为 n，因为每一个计算都依赖前面的结果，所以需要 n 步才能完成，也就是无法
并行。
3. 序列中任意两个位置信息传递的最短路径。比如 CNN 是 $\log_k n$, self attention 因为每个位置都有链接，
值为 1，而 RNN 最坏情况下，开始位置和结束位置的距离为 n.

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

从上图上可以看出，Self Attention 的唯一弱点就是计算复杂度是 $O(n^2d)$ ，当序列长度 n 比较大的时候，时间复杂度较高。而大模型时代对长文本的诉求，使这个弱点愈发凸显。目前也有很多方法来解决这个问题，比如滑动窗口，移动平均，甚至还有门机制来降低时间复杂度，这个我们以后再说。

如何构建 Self Attention 呢？Transformers 其实将上面的 Cross Attention 中的 query 改为自身 Embedding 的线性转换即可。大概如下图所示：



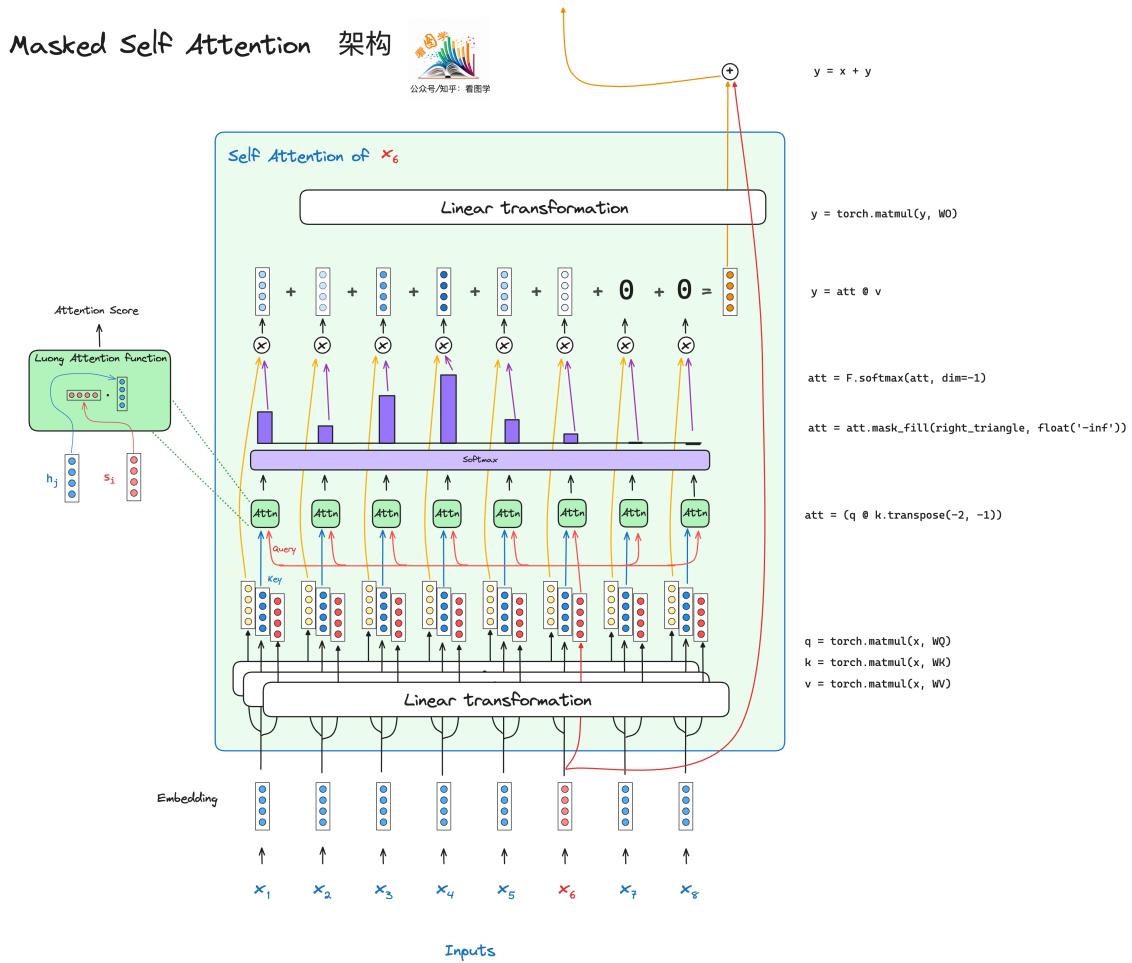
Transformers' Masked Scaled Dot-Product Self Attention

Encoder 部分既然已经加了 Self-Attention，那 Decoder 部分也必须加上。但是 Decoder 和 Encoder 有个很大的不同，那就是 Decoder 的输出，是一个一个往外蹦的。因为目前的建模就是通过前面的信息去预测下一个字。

所以最朴素的训练方法应该是一个长为 n 的预测序列，构造 n 条样本，第一个样本就根据 Encoder 预测第一个字符，最后一条样本则根据 Encoder 和前 $n - 1$ 个字符去预测第 n 个字符。但是这样做的话，训练效率未免有点太低，因为矩阵运算的加速完全用不上了。

怎么解决呢？其实也很简单，算还是照样算，但是算完之后丢弃一些数就完事了，也就是所谓的 mask

机制。在计算完所有的 Attention 后，把理论上无法看到的 Attention 直接变成 0 即可。如下图所示



Transofmers' Encoder

跟 Cross Attention 后添加 FFN 一样，Encoder 模块的 Self Attention 后面也加了 FFN 和 ResNet。所以说 Encoder 虽然只是整体模型的一部分，但是三驾马车都有，所以后来 Bert 几乎就是把 Transformers 的 Encoder 模块直接拿过来用了，在输入和输出的地方做了一点点修改。

Transformers' Decoder

Decoder 部分在之前的基础上也添加了 Self-Attention，但是这一次并没有添加 FFN。理论上增加会更好，(废话，参数变多了)，但是目前也有些研究证明，无论是 Encoder 和 Decoder 中的 FFN 模块已经有些冗余²。

当前的架构可以将 Cross Attention 和 Masked Cross Attention 是一个级联的 Attention，有点类似做了个二次索引。通过二次索引 Attend 到输入序列和输出序列的所有信息。

²One Wide Feedforward Is All You Need.

Attention 的矩阵表示

上面画的图展示的都是单个输入的计算流程。这些图也比较清晰的表明，在训练阶段，Encoder 和 Decoder 每个输入的计算流程都是一样的，所以很容易转化成矩阵的运算。

如下图所示。TODO

FFN 的作用

首先大家先猜一下整个 Transformers 中 FFN 的参数占比是多少？答案是 $2/3$ ，你猜对了么？

关于 FFN 的作用，后续研究人员做了很多实验和研究。但是直到目前也不能说就研究清楚了，因为神经网络的解释性本来就差。一个新技术的应用往往要比理论提前好久，比如 GBDT 等 ensemble 模型非常好用，但是很多年以后才用 Margin 的理论研究明白。

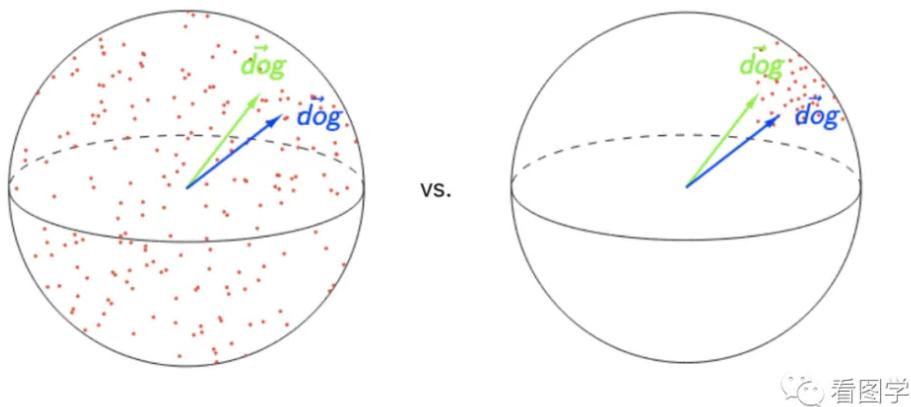
这里将一些 FFN 研究的成果做一个小汇总，只列举了目前大家都比较公认的结果。

1. FFN（还有 ResNet）是 Transformers 的必备模块，没有 FFN（或者 ResNet）的 Transformers 学不到什么东西

《Attention is Not All You Need: Pure Attention Loses Rank Doubly Exponentially with Depth》这篇论文，提出了 Transformers 架构存在 token uniformity 的归纳偏置 (inductive bias，有时候也叫归纳偏好) 问题。如果去掉 FFN 或者 Resnet，则问题更加严重。

这里解释一下这两个名词，所谓归纳偏置，可以通俗的理解为模型的“个性”，就是满足训练集合的解法有无数种，但是不同的模型架构会让模型更偏向于某些解法。比如我们常用的一些正则化方法，其实就是让模型的归纳偏置倾向于选择一些简单的解法。任何模型都有归纳偏置，尤其是碰到未见过的样本的时候，模型的归纳偏置就更容易体现出来。Transformers 的一个归纳偏执是什么呢？就是 token uniformity，有时候也叫 information diffusion，或者 anisotropic (各向异性)，也就是说训练完后的 token 会共享很多相似信息。

看下图大概就知道了，我们期望表示 token 的向量，相似的要相近，不相似的要远，而且最好是均匀的分布在整个空间中，比如下图所示。但是 Transformers 会存在各向异性的问题，也就是所有的 token 都挤到一个很窄的锥形区域了。



回到论文，论文将 FFN 和 ResNet 去掉之后做了一些消融实验，证明了 FFN 和 ResNet 是 Transformers 中的必备组件，这两个可以大大的缓解 token uniformity 或者各向异性的问题。

论文中从数学上证明了经过 Attention 变换后的输出与 rank-1 的矩阵之间的差值存在上界，但是有点复杂，我也没仔细推导过。简单一点的理解呢，就是 Attention 本质上是 value 的线性变换（虽然线性变换的权重是非线性的 softmax）。Every self-attention “layer” is a linear transformation of the previous layer (with non-linear weights)

当然并不是说 Transformers 就已经将 token uniformity 问题解决了，这个问题依然存在，所以后续又有 Bert-flow、whitening 等改进。详细可以参考：Bert 中的词向量各向异性具体什么意思啊？- 看图学的回答 - 知乎 <https://www.zhihu.com/question/460991118/answer/2353153090>

2. FFN 承担了记忆功能

这一节讲的两篇论文都非常有意思，建议大家看一看原始论文。

《Transformer Feed-Forward Layers Are Key-Value Memories》这篇文章做了很多实验和统计，得出了以下结论：

- (a) FFN 是一个 Key-Value 记忆网络，第一层线性变换是 Key Memory，第二层线性变换是 Value Memory。
- (b) FFN 学到的记忆有一定的可解释性，比如低层的 Key 记住了一些通用 pattern (比如以某某结尾)，而高层的 Key 则记住了一些语义上的 Pattern (比如句子的分类)。
- (c) Value Memory 根据 Key Memory 记住的 Pattern，来预测输出词的分布。
- (d) skip connection 将每层 FFN 的结果进行细化。

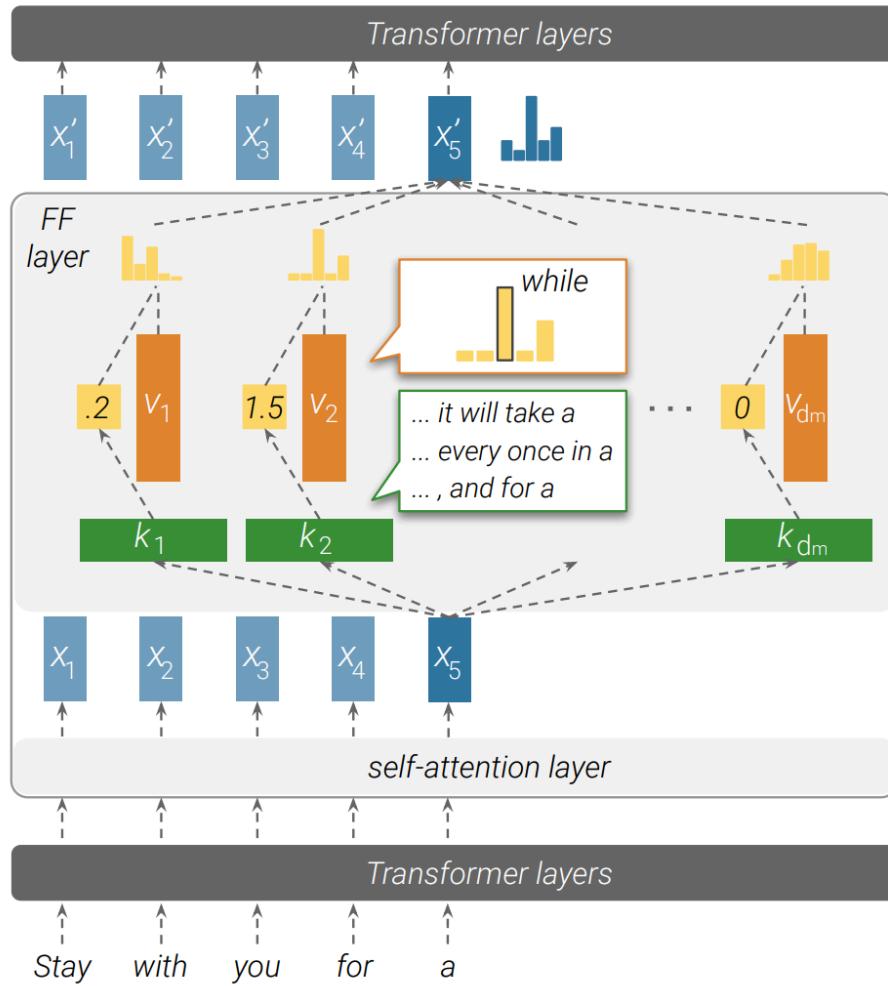
2015 年，《End-To-End Memory Networks》这篇论文提出了 Key-Value Memory 的结构，对于一个输入 x ，其网络结构为

$$\text{MemoryNet}(x) = \text{softmax}(x \cdot K^\top)V$$

FFN 的公式为，

$$\text{FFN}(x) = f(x \cdot W_1)W_2 = f(x \cdot K^\top)V$$

这里 f 是 ReLU 激活函数，可以看出两个结构的唯一区别就是一个是才用 softmax 进行归一化，另一个则采用 ReLU 进行筛选。本质上都差不多。



通过一些实验也确实证明了上面结论，也就是 FFN 确实将一些 pattern 或者知识记忆和存储起来了。这就很有意思，从这个角度来说，Attention 是对短期的信息进行提取，而 FFN 则对整个训练样本进行信息提取和记忆。这也就能解释为什么一个有限的窗口甚至对语料进行了暴力截断，模型也能记住语料库中的信息。

《Knowledge Neurons in Pretrained Transformers》这一篇就更有意思，在上一篇的基础上，对 Transformers 进行了前额叶切除手术。擎天柱瑟瑟发抖。

研究人员先是定位到对某些事实或者知识影响较大的神经元，然后神经元内的数值进行增强或者抑制，发现 Transformers 对这些事实或者知识的回答效果也会变好或者变差。如果将这个神经元删掉，也就是值全部置 0，则 Transformers 完全忘记了这些知识，更神奇的是，对于其他的知识则影响不大。

更进一步的，研究人员还对神经元的内容进行了替换操作以达到“篡改记忆”的效果。当然作者只是在 BERT 上进行了实验，随着预料和模型的增大，像定位知识的记忆也愈发的困难，但是给了人们一个可控文本生成的研究方向，未来可期。

3. FFN 是一种混合专家模型

MoEification: Transformer Feed-forward Layers are Mixtures of Experts

这是刘知远团队的论文，其实一直以来，神经网络就存在稀疏激活的现象，也就是在推理的时候，其实只有极小一部分参数参与了计算。这篇论文则通过 MoE 的思想来将 FFN 层拆分成了多个专家，并且新增了一个路由模块来确定推理的时候来挂哪个专家的门诊：）

这么做完之后，在提升推理速度的同时，效果依然能保持原来的 95% 以上。挺有价值的工作，大模型上也可以这么做一把。

下一步写什么

到目前为止，可以说刚刚把 Transformers 的架构写完。里面还有很多小细节，比如 scaled, label smoothing, 输入的一些处理等，这一些小细节准备专门写一章 Transformers 的面试题。再一个就是 Transformers 后续的发展。

* *

Position Embedding

从 Luong Attention 与 Bahdanau Attention 演变为 Transformers 的 Scaled Dot-Product Attention 后，出现了一个问题，那就是 token 的位置信息丢失了。

基于传统的 RNN 结构的 Attention 由于时刻 t 的隐层计算依赖时刻 $t - 1$ 的隐层，所以位置信息理论上是可以传递的。

Transformers 的 Attention 丢弃了 RNN 的结构之后，带来的好处就是可以并行计算了，但是信息通过时间/位置的传递的特性也就丢失了。

然而位置信息又很重要。

比如曾国藩写周报，如果写“臣屡战屡败”，结果可能是拖出去斩首。如果写“臣屡败屡战”，结果可能是忠勇无双有赏赐。

研究人员自然是想既要有要，所以就想办法从别的地方把位置信息加进去。我们来研究下 Position Embedding 的发展史。

朴素的想法

一个很直接的方法，就是直接把位置输入进去。这样做有两个缺点：

1. 泛化不好，没见过的位置模型处理不好
2. 这样模型的权重存在很大的数字，神经网络不喜欢大数字，影响训练的稳定性。

那如果是将位置做个归一化呢？也很难，因为无法找到一个归一化的标准。

绝对位置编码 (sinusoidal PE)

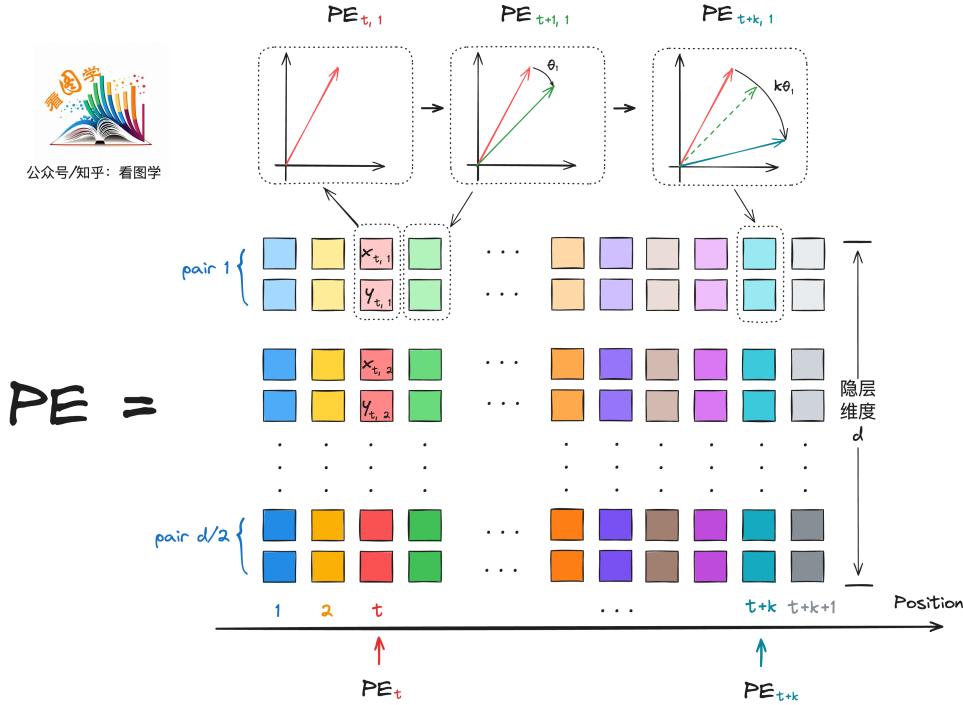
在《Attention is All your Need》的论文中，单独设置了一个绝对位置编码，叫 sinusoidal 位置编码，这个编码会和输入的词向量相加。文中的编码函数如下：

$$PE_{(t,2i)} = \sin\left(\frac{t}{10000^{2i/d}}\right)$$

$$PE_{(t,2i+1)} = \cos\left(\frac{t}{10000^{2i/d}}\right)$$

其中 t 代表了输入的位置。原文用了 pos 来表示，为了后续推导公式的方便，将 pos 统一改写为 t 。
 i 则是 Position Embedding 的向量下标，向量长度为 d , i 的取值范围是 $[0, \frac{d}{2}]$
这个函数看起来奇怪又有些恐怖，到底是什么鬼？
我们先看下结论：用图示来解释一下这个函数到底有什么特性。

Sinusoidal 绝对位置编码中，分组后，相对位置之间的线性变换等价于【顺时针旋转】



通俗一点来说，这个函数就是根据向量的下标两两配对，分成多组，每一组的二维向量根据 **位置信息** 进行 **顺时针旋转**，旋转的角度跟相对位置是线性关系。

论文中对这个函数进行了解释：“We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k , $PE_{(pos+k)}$ can be represented as a linear function of PE_{pos} .

用学术一点的说法，就是该函数是相对位置 k 的一个线性变换，也就是符合这么一个特性：

$$PE_{t+k} = f(PE_t, k)$$

其中， f 在这里是进行了顺时针旋转，旋转的角度是相对位置 k 的线性函数。

1. 下面是详细的证明：

为了方便，我们把原函数进行一些简化。原函数为：

$$PE_{(t,2i)} = \sin\left(\frac{t}{10000^{2i/d}}\right)$$

$$PE_{(t,2i+1)} = \cos\left(\frac{t}{10000^{2i/d}}\right)$$

由于函数对向量进行了两两分组，我们用 j 来表示分组 $PE_{(t,2i)}, PE_{(2i+1)}$ ，则有

$$PE(t, j) = \begin{cases} \sin(\theta_j \cdot t), & \text{if } j = 2i/2 \\ \cos(\theta_j \cdot t), & \text{if } j = (2i + 1)/2 \end{cases}$$

其中

$$\theta_j = \frac{1}{10000^{j/d}}$$

， 则 Position Embedding 则可以表示为：

$$PE_t = \begin{bmatrix} \sin(\theta_1 \cdot t) \\ \cos(\theta_1 \cdot t) \\ \sin(\theta_2 \cdot t) \\ \cos(\theta_2 \cdot t) \\ \vdots \\ \sin(\theta_{d/2} \cdot t) \\ \cos(\theta_{d/2} \cdot t) \end{bmatrix}_{d \times 1}$$

对于第 j 个 pair $PE_{t,j}$ 来说，如果 $PE_{t+k,j}$ 是 $PE_{t,j}$ 的线性变换，则存在一个矩阵 $M \in \mathbb{R}^{2 \times 2}$ 使得：

$$M \cdot PE_{t,j} = PE_{t+k,j}$$

也就是

$$\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix} \cdot \begin{bmatrix} \sin(\theta_j \cdot t) \\ \cos(\theta_j \cdot t) \end{bmatrix} = \begin{bmatrix} \sin(\theta_j \cdot (t + k)) \\ \cos(\theta_j \cdot (t + k)) \end{bmatrix}$$

根据三角函数求和的公式：

$$\sin(\alpha + \beta) = \sin \alpha \cos \beta + \cos \alpha \sin \beta$$

$$\cos(\alpha + \beta) = \cos \alpha \cos \beta - \sin \alpha \sin \beta$$

将右侧展开，左侧根据矩阵乘法展开，则有

$$\begin{aligned} \begin{bmatrix} u_1 \times \sin(\theta_j \cdot t) + v_1 \times \cos(\theta_j \cdot t) \\ u_2 \times \sin(\theta_j \cdot t) + v_2 \times \cos(\theta_j \cdot t) \end{bmatrix} &= \begin{bmatrix} \sin(\theta_j \cdot t) \cos(\theta_j \cdot k) + \cos(\theta_j \cdot t) \sin(\theta_j \cdot k) \\ \cos(\theta_j \cdot t) \cos(\theta_j \cdot k) - \sin(\theta_j \cdot t) \sin(\theta_j \cdot k) \end{bmatrix} \\ &= \begin{bmatrix} \sin(\theta_j \cdot t) \cos(\theta_j \cdot k) + \cos(\theta_j \cdot t) \sin(\theta_j \cdot k) \\ -\sin(\theta_j \cdot t) \sin(\theta_j \cdot k) + \cos(\theta_j \cdot t) \cos(\theta_j \cdot k) \end{bmatrix} \end{aligned}$$

解这个方程会得到 (根据上面公式对号入座就可以)：

$$M = \begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix} = \begin{bmatrix} \cos(\theta_j \cdot k) & \sin(\theta_j \cdot k) \\ -\sin(\theta_j \cdot k) & \cos(\theta_j \cdot k) \end{bmatrix}$$

学过线代的看着这个矩阵是不是有点眼熟？没错，这个矩阵和逆时针旋转矩阵

$$\begin{bmatrix} \cos(\theta_j.k) & -\sin(\theta_j.k) \\ \sin(\theta_j.k) & \cos(\theta_j.k) \end{bmatrix}$$

非常像。

这两个矩阵什么联系呢？其实也很简单，上面的是顺时针旋转矩阵，下面的是逆时针旋转矩阵。

比如我顺时针旋转了 θ ，就相当于逆时针旋转了 $-\theta$ ，带入逆时针旋转矩阵有：

$$R(-\theta) = \begin{bmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

所以，我们就知道了这个论文中提到的线性变换 $PE_{t+k} = f(PE_t, k)$ 就是顺时针旋转，旋转的角度为 $\theta_j.k$ ，和相对位置 k 是线性关系。

对于所有 pair 来说， PE_{t+k} 相对于 PE_t 的顺时针旋转，写成矩阵形式为：

$$\begin{bmatrix} \cos(\theta_1.k) & \sin(\theta_1.k) & 0 & 0 & \cdots & 0 & 0 & \sin(\theta_1.t) \\ -\sin(\theta_1.k) & \cos(\theta_1.k) & 0 & 0 & \cdots & 0 & 0 & \cos(\theta_1.t) \\ 0 & 0 & \cos(\theta_2.k) & \sin(\theta_2.k) & \cdots & 0 & 0 & \sin(\theta_2.t) \\ 0 & 0 & -\sin(\theta_2.k) & \cos(\theta_2.k) & \cdots & 0 & 0 & \cos(\theta_2.t) \\ 0 & 0 & 0 & 0 & \ddots & 0 & 0 & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos(\theta_{d/2}.k) & \sin(\theta_{d/2}.k) & \sin(\theta_{d/2}.t) \\ 0 & 0 & 0 & 0 & \cdots & -\sin(\theta_{d/2}.k) & \cos(\theta_{d/2}.k) & \cos(\theta_{d/2}.t) \end{bmatrix}$$

不得不说这个设计还是很精彩的，虽然还是写死的绝对位置编码，但是却巧妙的捕捉到了相对位置关系。

下面是一些疑问和思考：

- 为什么要分组？不分组然后向量整体旋转行不行？理论上也可以，但是这样做的话上面的矩阵就变成了一个稠密矩阵，影响运算速度。
- 为什么随着下标的增加，旋转的角度越来越小？这个我也没有很严谨的证明过，感觉跟时钟系统有点像吧，时针分针秒针分别表示不同粒度的时间，在钟表上顺时针旋转。这里既然也是旋转，那用不同的颗粒度应该能捕获更细粒度的位置信息。

需要注意的是，《Attention is All your Need》中，词向量和绝对位置向量采用的是相加的操作，然后再进行线性变换和 Attention 操作。具体来说就是：

$$\begin{aligned}\mathbf{q}_i &= (\mathbf{x}_i + \mathbf{p}_i) \mathbf{W}_Q \\ \mathbf{k}_j &= (\mathbf{x}_j + \mathbf{p}_j) \mathbf{W}_K \\ \mathbf{v}_j &= (\mathbf{x}_j + \mathbf{p}_j) \mathbf{W}_V \\ a_{i,j} &= \text{softmax}(\mathbf{q}_i \mathbf{k}_j^\top) \\ \mathbf{o}_i &= \sum_j a_{i,j} \mathbf{v}_j\end{aligned}$$

比如将 $\mathbf{q}_i \mathbf{k}_j^\top$ 完全展开后

$$\mathbf{q}_i \mathbf{k}_j^\top = \mathbf{x}_i \mathbf{W}_Q \mathbf{W}_K^\top \mathbf{x}_j^\top + \mathbf{x}_i \mathbf{W}_Q \mathbf{W}_K^\top \mathbf{p}_j^\top + \mathbf{p}_i \mathbf{W}_Q \mathbf{W}_K^\top \mathbf{x}_j^\top + \mathbf{p}_i \mathbf{W}_Q \mathbf{W}_K^\top \mathbf{p}_j^\top$$

上面有颜色的部分代表了位置编码，有的将其变成可训练的参数，有的甚至把中间两项都去掉了。总之，后续的 Position Embedding 很多进展都是在 x_i, p_i 还有他们之间的组合关系上做文章。

苏剑林对 sinusoidal 绝对位置编码进行了改造，提出了一种更优雅和兼具外推性的编码方式，就是旋转位置编码 (RoPE).

旋转位置编码 (RoPE)

绝对位置编码采用的是词向量与 PE 相加，然后再做线性变换的方式。公式如下：

$$\mathbf{q}_t = (\mathbf{x}_t + \mathbf{p}_t) \mathbf{W}_Q$$

而 RoPE 则更简单直接，丢弃了绝对位置编码 PE，词向量做线性变换后，按照前面说的 PE 分组方式，直接进行旋转。

这个论文已经画的十分清楚了：

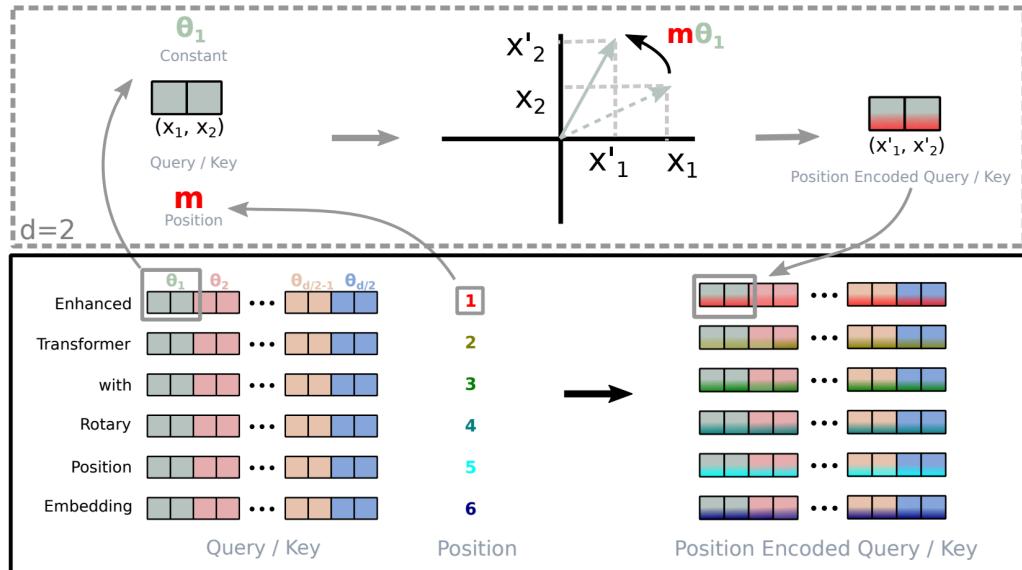


Figure 1: Implementation of Rotary Position Embedding(RoPE).

下面是详细的公式推导，去掉复数领域，只考虑旋转矩阵的一些证明。

假设 $\mathbf{q}_t = \mathbf{x}_t \mathbf{W}_Q$, 仿照绝对位置编码进行两两分组, 则

$$q_t = \begin{bmatrix} q_{t,1}^{(1)} \\ q_{t,1}^{(2)} \\ q_{t,2}^{(1)} \\ q_{t,2}^{(2)} \\ \vdots \\ q_{t,d/2}^{(1)} \\ q_{t,d/2}^{(2)} \end{bmatrix}_{d \times 1}$$

对于一个分组 $q_{t,j}$ 来说, 旋转后的 $RoPE(q_{t,j})$ 为:

$$RoPE(q_{t,j}) = R(\theta_j \cdot t)(q_{t,j}) = \begin{bmatrix} \cos(\theta_j \cdot t) & -\sin(\theta_j \cdot t) \\ \sin(\theta_j \cdot t) & \cos(\theta_j \cdot t) \end{bmatrix} \begin{bmatrix} q_{t,j}^{(1)} \\ q_{t,j}^{(2)} \end{bmatrix}$$

写成矩阵形式为:

$$RoPE(q_t) = R(\theta \cdot t)(q_t) = \begin{bmatrix} \cos(\theta_1 \cdot t) & -\sin(\theta_1 \cdot t) & 0 & 0 & \cdots & 0 & 0 \\ \sin(\theta_1 \cdot t) & \cos(\theta_1 \cdot t) & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos(\theta_2 \cdot t) & -\sin(\theta_2 \cdot t) & \cdots & 0 & 0 \\ 0 & 0 & \sin(\theta_2 \cdot t) & \cos(\theta_2 \cdot t) & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & \cos(\theta_{d/2} \cdot t) & -\sin(\theta_{d/2} \cdot t) \\ 0 & 0 & 0 & 0 & \cdots & \sin(\theta_{d/2} \cdot t) & \cos(\theta_{d/2} \cdot t) \end{bmatrix} \begin{bmatrix} q_{t,1}^{(1)} \\ q_{t,1}^{(2)} \\ q_{t,2}^{(1)} \\ q_{t,2}^{(2)} \\ \vdots \\ q_{t,d/2}^{(1)} \\ q_{t,d/2}^{(2)} \end{bmatrix}$$

RoPE 这么做的好处是, 当 q 和 k 进行 attention 操作后, 依然可以保留相对位置, 这样就很好的完成了位置编码的任务, 而且旋转位置编码就像时钟一样可以无限的旋转, 具备很好的外推性。

$$RoPE(q_{m,j}), RoPE(k_{n,j}) = RoPE(q_{m-n,j}), RoPE(k_{0,j})$$

证明如下:

假设位置 m 的 pair $q_{m,j}$ 和位置 n 的 pair $k_{n,j}$ 取点积操作计算 attention, 则有:

$$\begin{aligned}
RoPE(q_{m,j}), RoPE(k_{n,j}) &= \begin{bmatrix} \cos(\theta_j.m) & -\sin(\theta_j.m) \\ \sin(\theta_j.m) & \cos(\theta_j.m) \end{bmatrix} \begin{bmatrix} q_{m,j}^{(1)} \\ q_{m,j}^{(2)} \end{bmatrix}, \begin{bmatrix} \cos(\theta_j.n) & -\sin(\theta_j.n) \\ \sin(\theta_j.n) & \cos(\theta_j.n) \end{bmatrix} \begin{bmatrix} k_{n,j}^{(1)} \\ k_{n,j}^{(2)} \end{bmatrix} \\
&= \begin{bmatrix} q_{m,j}^{(1)} \cos(\theta_j.m) - q_{m,j}^{(2)} \sin(\theta_j.m) \\ q_{m,j}^{(1)} \sin(\theta_j.m) + q_{m,j}^{(2)} \cos(\theta_j.m) \end{bmatrix}, \begin{bmatrix} k_{n,j}^{(1)} \cos(\theta_j.n) - k_{n,j}^{(2)} \sin(\theta_j.n) \\ k_{n,j}^{(1)} \sin(\theta_j.n) + k_{n,j}^{(2)} \cos(\theta_j.n) \end{bmatrix} \\
&= (q_{m,j}^{(1)} \cos(\theta_j.m) - q_{m,j}^{(2)} \sin(\theta_j.m)) \times (k_{n,j}^{(1)} \cos(\theta_j.n) - k_{n,j}^{(2)} \sin(\theta_j.n)) \\
&\quad + (q_{m,j}^{(1)} \sin(\theta_j.m) + q_{m,j}^{(2)} \cos(\theta_j.m)) \times (k_{n,j}^{(1)} \sin(\theta_j.n) + k_{n,j}^{(2)} \cos(\theta_j.n)) \\
&= q_{m,j}^{(1)} k_{n,j}^{(1)} (\cos(\theta_j.m) \cos(\theta_j.n) + \sin(\theta_j.m) \sin(\theta_j.n)) \\
&\quad + q_{m,j}^{(1)} k_{n,j}^{(2)} (-\cos(\theta_j.m) \sin(\theta_j.n) + \sin(\theta_j.m) \cos(\theta_j.n)) \\
&\quad + q_{m,j}^{(2)} k_{n,j}^{(1)} (-\sin(\theta_j.m) \cos(\theta_j.n) + \cos(\theta_j.m) \sin(\theta_j.n)) \\
&\quad + q_{m,j}^{(2)} k_{n,j}^{(2)} (\sin(\theta_j.m) \sin(\theta_j.n) + \cos(\theta_j.m) \cos(\theta_j.n)) \\
&= q_{m,j}^{(1)} k_{n,j}^{(1)} \cos(\theta_j.(m-n)) + q_{m,j}^{(1)} k_{n,j}^{(2)} \sin(\theta_j.(m-n)) \\
&\quad + q_{m,j}^{(2)} k_{n,j}^{(1)} \sin(\theta_j.(m-n)) + q_{m,j}^{(2)} k_{n,j}^{(2)} \sin(\theta_j.(m-n)) \\
&= (q_{m,j}^{(1)} \cos(\theta_j.(m-n)) - q_{m,j}^{(2)} \sin(\theta_j.(m-n))) \times k_{n,j}^{(1)} \\
&\quad + (q_{m,j}^{(1)} \sin(\theta_j.(m-n)) + q_{m,j}^{(2)} \sin(\theta_j.(m-n))) \times k_{n,j}^{(2)} \\
&= RoPE(q_{m-n,j}), RoPE(k_{0,j})
\end{aligned}$$

其他编码方式

目前主流就是 RoPE，其他的方式后面慢慢更新。

Transformers 面试八股

为什么要除以 \sqrt{d}

1. 题目：Attention 的计算公式中 $\text{Attention}(Q, K, V) = \{\text{softmax}(\frac{QK^\top}{\sqrt{d}})\}$ 为什么要除以 \sqrt{d}

这个题目可以说是 NLP 面试中一个高频出现的问题，基本上问到 Attention 或者 Transformers 的时候都会问。

这是个好题目，我作为面试官的时候也经常问，因为很快能了解到面试同学的数学功底怎么样。

如果你是 NLP 学生或者从业者，不妨先试着回答一下。如果有更好的答案欢迎交流。

2. 最基本的答案

这个问题在《Attention is All You Need》的原始论文中是给出了一个粗略的答案的。

While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k [3]. We suspect that for large values of d_k , the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

作者说，当 d_k 的值变大的时候，softmax 函数会造成梯度消失问题，所以设置了一个 softmax 的 temperature 来缓解这个问题。这里 temperature 被设置为了 $\sqrt{d_k}$ ，也就是乘上 $\frac{1}{\sqrt{d_k}}$ 。

这个回答当然没什么问题，但是接下来就会再问两个问题：

- (a) 为什么会导致梯度消失?
- (b) 为什么是 $\sqrt{d_k}$, 有更好的值么?

下面来回答一下这两个衍生的问题。

3. d_k 变大为什么会导致梯度消失?

先说结论:

- (a) 如果 d_k 变大, $q \cdot k^\top$ 方差会变大。
- (b) 方差变大会导致向量之间元素的差值变大。
- (c) 元素的差值变大会导致 softmax 退化为 argmax, 也就是最大值 softmax 后的值为 1, 其他值则为 0。
- (d) softmax 只有一个值为 1 的元素, 其他都为 0 的话, 反向传播的梯度会变为 0, 也就是所谓的梯度消失。

下面分别证明这 4 点。

- (a) d_k 变大, $q \cdot k^\top$ 方差会变大。假设 q 和 k 为长度为 d_k , 均值为 0, 方差为 1 的向量。则 q 和 k 的点积的方差为:

$$\begin{aligned} \text{var}[q \cdot k^\top] &= \text{var}\left[\sum_{i=1}^{d_k} q_i \times k_i\right] \\ &= \sum_{i=1}^{d_k} \text{var}[q_i \times k_i] \\ &= \sum_{i=1}^{d_k} \text{var}[q_i] \times \text{var}[k_i] \\ &= \sum_{i=1}^{d_k} 1 \\ &= d_k \end{aligned}$$

所以, 当 d_k 变大时, 方差变大。证毕。

- (b) 方差变大会导致向量之间元素的差值变大。这似乎是一个显而易见的结论, 因为方差变大就是代表了数据之间的差异性变大。如果非要给出证明呢, 可以将这个问题换一个问题来侧面回答这个问题。新的问题为: 假设向量是通过独立同分布的数据采样出来的 d_k 个数据, 那么这 d_k 个数的最大值的期望是多少? 因为分布很多, 这里只给出最常用的正态分布的证明, 详细证明见: http://www.gautamkamath.com/writings/gaussian_max.pdf 这里只给出结论如下: 【image】从期望的下界, 可以看出, 方差越大, 最大值的期望越大。同时还有个结论就是 d_k 越大, 最大值的期望也越大。由于正太分布是对称的, 最小值就是最大值取负号。
- 所以方差变大, 数据分布的最大最小值的差值变大了, 也就从侧面证明了向量元素之间的差值变大了。

- (c) softmax 退化为 argmax 对于 softmax 函数中的每个分量 $\$ \text{softmax}(x_i) \$$, 我们可以写成:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

当 x_k 是最大的元素时, e^{x_k} 会显著大于其他 e^{x_i} (其中 $i \neq k$), 尤其是当这些 x_i 和 x_k 之间的差距变得非常大时。为了更清楚地看出这一点, 我们将 x_i 的每个元素表示成最大元素 x_k 减去一个差值 Δ_i , 即 $x_i = x_k - \Delta_i$, 其中 $\Delta_k = 0$ 。

因此, softmax 函数可以重写为:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} = \frac{e^{x_k - \Delta_i}}{\sum_{j=1}^n e^{x_k - \Delta_j}}$$

由于 e^{x_k} 是公因子, 可以提出:

$$\text{softmax}(x_i) = \frac{e^{x_k} e^{-\Delta_i}}{e^{x_k} \sum_{j=1}^n e^{-\Delta_j}} = \frac{e^{-\Delta_i}}{\sum_{j=1}^n e^{-\Delta_j}}$$

当 Δ_i 非常大 (即 x_i 远小于 x_k) 时, $e^{-\Delta_i}$ 会接近于 0。因此, 除了 $\Delta_k = 0$ 以外的所有项, 其他项 $e^{-\Delta_j}$ 都会非常小, 可以忽略不计。于是, 对于 $i = k$:

$$\text{softmax}(x_k) \approx \frac{1}{1} = 1$$

而对于 $i \neq k$:

$$\text{softmax}(x_i) \approx 0$$

所以说当输入向量 \mathbf{x} 的方差变得非常大时, softmax 函数将会趋近于将最大的元素赋值为 1, 而其他元素赋值为 0, 也就是 argmax 函数。用公式表示的话:

$$\lim_{\text{var}(\mathbf{x}) \rightarrow \infty} \text{softmax}(\mathbf{x}) = \text{argmax}(\mathbf{x})$$

所以方差变大时, softmax 函数会退化为 argmax 函数。

这里我们可以做个实验看一下:

```
import numpy as np

n = 10

x1 = np.random.normal(loc=0, scale=1, size=n)
x2 = np.random.normal(loc=0, scale=np.sqrt(512), size=n)
print('x1最大值和最小值的差值:', max(x1) - min(x1))
print('x2最大值和最小值的差值:', max(x2) - min(x2))

def softmax(x):
    return np.exp(x) / np.sum(np.exp(x), keepdims=True)

def softmax_grad(y):
```

```

    return np.diag(y) - np.outer(y, y)

ex1 = softmax(x1)
ex2 = softmax(x2)
print('softmax(x1) =', ex1)

print('softmax(x2) =', ex2)

```

其结果为：

```

x1最大值和最小值的差值: 1.8973472870218264
x1最大值和最小值的差值: 66.62254341144866
softmax(x1) = [0.16704083 0.21684976 0.0579299 0.05408421 0.16109133 0.14433417
0.03252007 0.05499126 0.04213939 0.06901908]
softmax(x2) = [4.51671361e-19 2.88815837e-21 9.99999972e-01 3.02351231e-17
3.73439970e-25 8.18066523e-13 2.78385563e-08 1.16465424e-29
7.25661271e-20 3.21813750e-21]

```

可以看出，在方差为 $\sqrt{512}$ 的时候，softmax 只有第三个元素接近 1，其他都几乎为 0.

(d) softmax 什么情况下会梯度消失

我们来对 softmax 函数进行求导。定义 softmax 为

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

假设我们有一个向量 $\mathbf{z} = [z_1, z_2, \dots, z_n]$, softmax 函数的输出是一个向量 $\mathbf{y} = [y_1, y_2, \dots, y_n]$, 其中：

$$y_i = \text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^n \exp(z_j)}$$

我们需要计算 softmax 函数的导数，即 $\frac{\partial y_i}{\partial z_k}$ ，分为两种情况：

- i. 当 $i = k$
- ii. 当 $i \neq k$

首先，计算 y_i 对 z_k 的导数：

- i. 1. 当 $i = k$ 时

$$\frac{\partial y_i}{\partial z_i} = \frac{\partial}{\partial z_i} \left(\frac{\exp(z_i)}{\sum_j \exp(z_j)} \right)$$

使用商的导数法则，我们得到：

$$\frac{\partial y_i}{\partial z_i} = \frac{\exp(z_i) \sum_j \exp(z_j) - \exp(z_i) \exp(z_i)}{\left(\sum_j \exp(z_j) \right)^2}$$

化简得到：

$$\frac{\partial y_i}{\partial z_i} = \frac{\exp(z_i) (\sum_j \exp(z_j) - \exp(z_i))}{(\sum_j \exp(z_j))^2} = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \left(1 - \frac{\exp(z_i)}{\sum_j \exp(z_j)} \right)$$

即：

$$\frac{\partial y_i}{\partial z_i} = y_i(1 - y_i)$$

ii. 2. 当 $i \neq k$ 时

$$\frac{\partial y_i}{\partial z_k} = \frac{\partial}{\partial z_k} \left(\frac{\exp(z_i)}{\sum_j \exp(z_j)} \right)$$

同样使用商的导数法则，我们得到：

$$\frac{\partial y_i}{\partial z_k} = \frac{0 \cdot \sum_j \exp(z_j) - \exp(z_i) \exp(z_k)}{(\sum_j \exp(z_j))^2} = -\frac{\exp(z_i) \exp(z_k)}{(\sum_j \exp(z_j))^2}$$

即：

$$\frac{\partial y_i}{\partial z_k} = -y_i y_k$$

iii. 两种情况合并一下

将两种情况合并，softmax 的导数可以表示为：

$$\frac{\partial y_i}{\partial z_k} = y_i(\delta_{ik} - y_k)$$

其中， δ_{ik} 是 Kronecker delta 函数，定义为：

$$\delta_{ik} = \begin{cases} 1, & \text{if } i = k \\ 0, & \text{if } i \neq k \end{cases}$$

最终可以用 Jacobian 矩阵表示，Jacobians 矩阵的第 i 行和第 k 列元素是 $\frac{\partial y_i}{\partial z_k}$

$$\mathbf{J} = \begin{bmatrix} y_1(1 - y_1) & -y_1 y_2 & \cdots & -y_1 y_n \\ -y_2 y_1 & y_2(1 - y_2) & \cdots & -y_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ -y_n y_1 & -y_n y_2 & \cdots & y_n(1 - y_n) \end{bmatrix}$$

iv. 然后与第三步的世界线交汇，好玩的来了

在第三步中我们证明了当方差变大的时候，softmax 退化成了 argmax，也就是变成一个只有一个 1 其他全为 0 的向量。

这个向量带入到上面的雅可比矩阵会发生什么？我们发现对于任意的 $y_k = 1, y_{j \neq k} = 0$ 的向量来说，雅可比矩阵变成了一个全 0 矩阵。

也就是说梯度全为 0 了。到这里才算是证明了为什么 $q \cdot k^\top$ 的方差不能太大，太大了就梯度消失。

v. 梯度实验

这里我们可以做个实验看一下：

```

import numpy as np

n = 10

x1 = np.random.normal(loc=0, scale=1, size=n)
x2 = np.random.normal(loc=0, scale=np.sqrt(512), size=n)
print('x1最大值和最小值的差值:', max(x1) - min(x1))
print('x1最大值和最小值的差值:', max(x2) - min(x2))

def softmax(x):
    return np.exp(x) / np.sum(np.exp(x), keepdims=True)

def softmax_grad(y):
    return np.diag(y) - np.outer(y, y)

ex1 = softmax(x1)
ex2 = softmax(x2)
print('softmax(x1) =', ex1)
print('max of gradiant of softmax(x1) =', np.max(softmax_grad(ex1)))
print('softmax(x2) =', ex2)
print('max gradiant of softmax(x2) =', np.max(softmax_grad(ex2)))

```

其结果为：

```

x1最大值和最小值的差值: 1.8973472870218264
x1最大值和最小值的差值: 66.62254341144866
softmax(x1) = [0.16704083 0.21684976 0.0579299  0.05408421 0.16109133 0.14433417
 0.03252007 0.05499126 0.04213939 0.06901908]
max of gradiant of softmax(x1) = 0.1698259433168865
softmax(x2) = [4.51671361e-19 2.88815837e-21 9.99999972e-01 3.02351231e-17
 3.73439970e-25 8.18066523e-13 2.78385563e-08 1.16465424e-29
 7.25661271e-20 3.21813750e-21]
max gradiant of softmax(x2) = 2.7839373695215386e-08

```

可以看出，在方差为 $\sqrt{512}$ 的时候，长度仅仅为 10 的向量 x_2 ，其梯度就已经快没有了，最大值为 $2.78e-8$ 。

而如果将方差控制在 1，则最大的梯度为 0.1698

4. scale 的值为什么是 $\sqrt{d_k}$, 有更好的值么?

从上一节的第一步的证明, 可以发现, scale 的值为 $\sqrt{d_k}$ 其实是把 $q \cdot k^\top$ 归一化成了一个均值为 0, 方差为 1 的向量。

至于是不是最好呢? 不好说, 因为参数的分布我们不太清楚。苏神曾经试图求解了一些常用分布的最佳 scale 值, 感兴趣的可以看下: <https://spaces.ac.cn/archives/9812>

Transformers 为什么使用 Layer Norm

1. 题目:

Transformers 中为什么使用 Layer Norm 而不是其他的? 比如 Batch Norm?

为什么使用还是比较容易回答, 为啥不用其他的则不是那么容易回答。

2. 为什么使用 Layer Norm

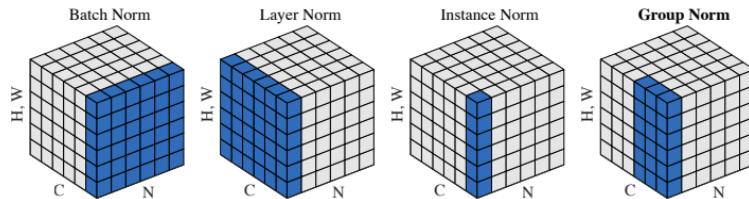
不仅仅是使用 Layer Norm, 各种 Normalize 的操作, 首先是保证训练的稳定性。因为当神经网络很深的时候, 反向传播的参数计算往往都是指数级的变化, 太大或者太小的数值送入激活函数后就容易造成梯度消失或者梯度爆炸, 训练就挂了。

其次是加速模型的收敛。这个也比较容易理解, 模型的每一层都在拟合一个数据分布, 而如果不进行 normalize, 那么每次的输入分布可能随时都在变化, 这样学习起来就很困难。在 normalize 之后, 绝大部分数值都集中在了一个可接受的范围内, 每一层的参数就安心拟合这个分布就好了, 这个范围正好又是激活函数的“舒适区”, 所以模型收敛速度会更快。

还有一个额外的好处就是让模型的训练不再那么依赖权重的初始化, 早期的时候初始化对模型结果的影响还是蛮大的, 也是个很火热的研究方向。就包括现在也有很多研究, 比如 `torch.manual_seed(3407) is all you need`

3. 为什么不用 Batch Norm

其实在 Transformers 论文刚出的那个时间点, 就是两种 Normalize 比较流行, 早一点提出的是 Batch Norm, 后来是 Layer Norm, 至于再后来的 Instance Norm 和 Group Norm 等都可以认为是这两种基础上的扩展。



当时 Batch Norm 在 CV 领域比较流行, 而 NLP 则使用 Layer Norm 比较多。但也并不是一定要按照任务这么划分。

当时为什么 CV 都使用 Batch Norm 呢? 我个人觉得是因为站在 CNN 卷积核的计算方式上看, 在 Batch 上进行 Normalize 是比较契合的, 因为这样每个卷积核在计算的时候数据的 Normalize 的方式是一样的。

但是 Batch Norm 也有些问题, 下面会简单说几点:

- (a) Batch 需要大，小的 Batch 训练不稳定。因为 Batch Norm 需要跨样本的 Normalize，所以采样要足够大才能捕捉到样本的分布。
- (b) 加大 Batch 有一些副作用。一个最明显的问题，那就是现在模型越来越大，如果想实现多机多卡的 GPU 并行，那 Batch Norm 需要额外的通信，因为一个 Batch 很可能分布在不同的机器上，而 Normalize 又需要计算整个样本的数据分布才行。现在大概有两种解法，一种是类似 mini batch，放弃跨机器通信；一种是 Pytorch 实现的 SyncBatchNorm，在前面的基础上尽量减少通信的数据。但是不管怎么样，额外的通信开销在模型足够大的时候也是个问题。
- (c) 训练和预测的不一致性。训练的时候有大批的数据可以组成 Batch，但是预测的时候，我如果只想预测一个样本，那 Batch Norm 就废了。所以在预测的时候实际上是采用了训练时候的数据分布来进行 Normalize 的。这样就必须要保证训练和预测的分布必须一致，泛化能力没那么强。
- (d) 并不适合当时的 NLP 主流框架比如 RNN。
- (e) 并不太适合长度不固定的 NLP 序列。因为每个 Batch 最后总有些 pad，这些 pad 会干扰 Batch Norm 的数据分布。如果把 pad 都不参与计算，那就相当于 batch 越来越小，根据第一条也不太好。
- (f) 还有就是现在多模态的输入，也不太适合 Batch Norm?? 存疑

关于数据分布还有 Batch 的讨论，可以看下论文 Facebook 的《Rethinking “Batch”in BatchNorm》，其中还提到了 Batch 内的信息泄漏等问题，感兴趣的可以阅读一下原始的论文。

所以说 Layer Norm 改成了按特征来 Normalize，可以很好的处理小 Batch 和变长输入的问题，也没有额外的通信，可以说基本解决了上面的问题。所以来 Layer Norm 越来越流行，Batch Norm 反而不太用了，尤其是大模型时代一直在追求更高的训练效率，Batch Norm 的额外通信就有些不合时宜了。当然现在大模型也有其他的 Normalize 方法，比如 RMSNorm、DeepNorm 等。

4. 实验论证 1：Transformers 使用 Batch Norm 效果不太好

上面说了一些 Batch Norm 和 Layer Norm 的对比，有人可能会说你这都是理论上的，有证据么？还真有，2020 年的一篇论文专门测试了把 Transformers 中的 Layer Norm 变成了 Batch Norm，打了个擂台。论文的名字是《PowerNorm: Rethinking Batch Normalization in Transformers》

这篇论文中，作者发现 Transformers 中的 LayerNorm 换成 Batch Norm 后，在分类和机器翻译的任务上性能下降明显。如下图

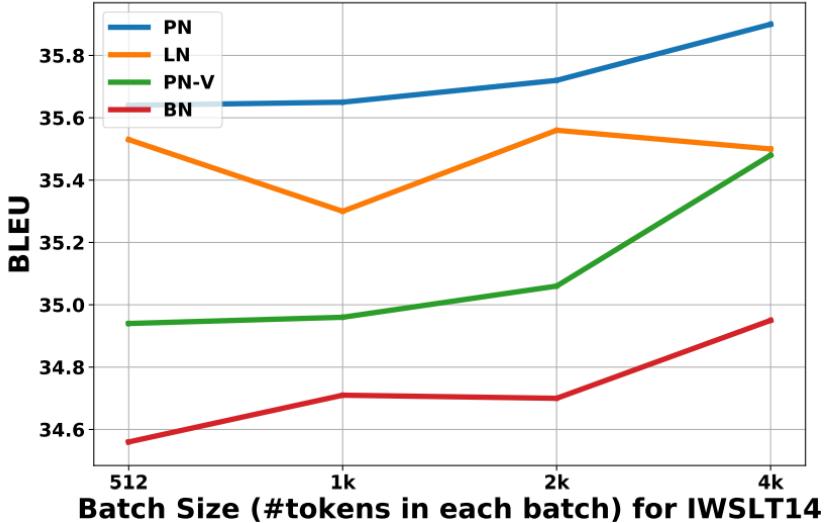


Figure 5. Ablation study of the performance of PN, PN-V, LN and BN on IWSLT14 trained using different batch sizes. Note that the performance of PN consistently outperforms LN. In the meanwhile, PN-V can only match the result of LN when mini-batch gets to 4K. Among all the settings, BN behaves poorly and abnormally across different mini-batches.

可以看出，Batch Norm 效果不太好。分析其原因呢，作者指出采用 Batch Norm 的 Transformers，其 Batch Norm 的均值和方差震荡明显，并不稳定，所以收敛的就很慢。

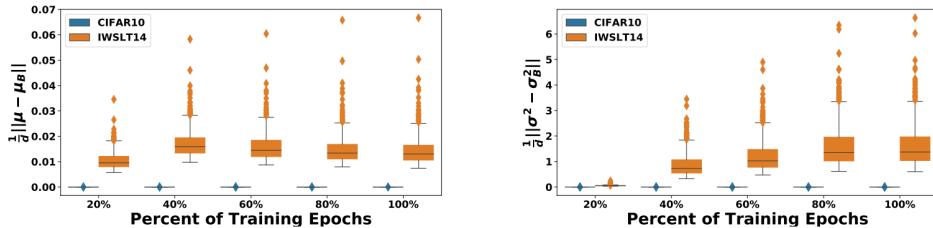


Figure 2. The average Euclidean distance between the batch statistics (μ_B, σ_B^2) and the running statistics (μ, σ^2) stored in first BN during forward pass for ResNet20 on Cifar-10 and Transformer on IWLST14. We can clearly see that the ResNet20 statistics have orders of magnitude smaller variation than the running statistics throughout training. However, the corresponding statistics in Transformer_{BN} exhibit very high variance with extreme outliers. This is true both for the mean (shown in the left) as well as variance (shown in right). This is one of the contributing factors to the low performance of BN in transformers.

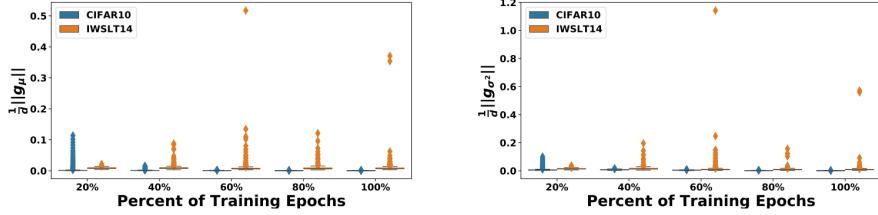


Figure 3. The average gradient norm of the input of the first BN layer contributed by μ_B and σ_B for ResNet20 on Cifar10 and Transformer_{BN} on IWSLT14 during the BP (note that $d = 16$ for Cifar-10 and $d = 512$ for IWSLT experiment). It can be clearly seen that the norm of g_μ and g_{σ^2} for ResNet20 has orders of magnitude smaller variation throughout training, as compared to that for Transformer_{BN}. Also, the outliers for ResNet20 vanish at the end of training, which is in contrast to Transformer_{BN}, for which the outliers persist. This is true both for g_μ (shown in left) as well as g_{σ^2} (shown in right).

当然作者后来对 BN 进行了改进，提出了 PowerNorm，感兴趣的可以看看。

5. 实验论证 2：Layer Norm 会改善 Transformers 的注意力 来自 2023 年的论文：«On the Expressivity Role of LayerNorm in Transformers' Attention»

这篇文章在较小的 Transformers 模型上做了实验，发现 Layer Norm 为 Attention 提供了两个功能

- (a) Projection：会将输入投影到 query 和 key 正交的超平面，这样方便所有的 key 可以同等访问。
- (b) Scaling：每一个 Key 都能被选中，都有机会获得最高分。

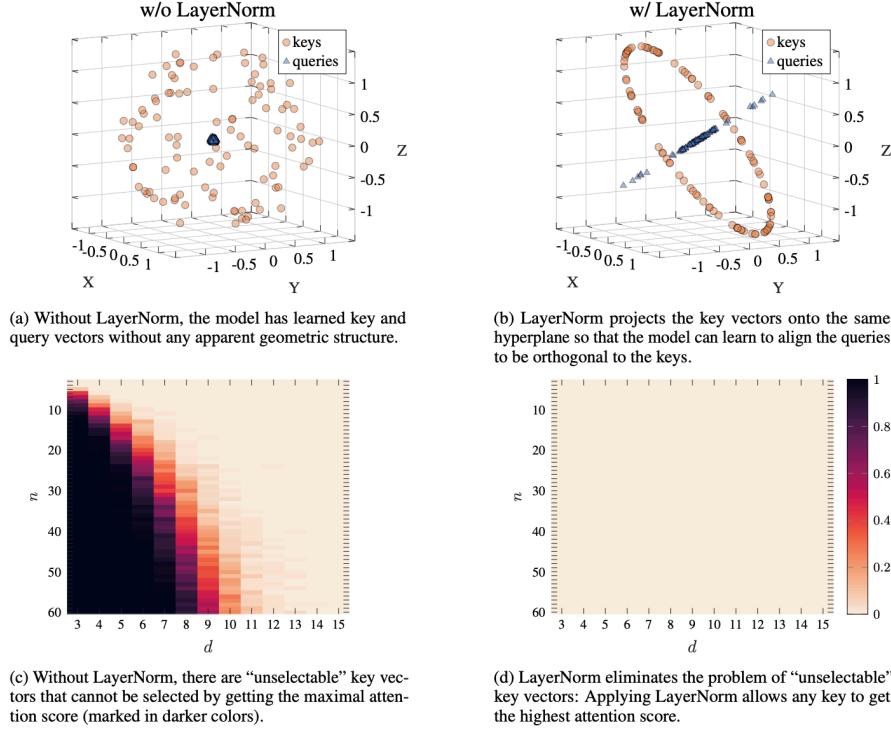


Figure 1: Figures 1a and 1b show the effect of *projection* in LayerNorm, which makes all keys lie on the hyperplane that is orthogonal to the $\vec{1}$ vector. Figures 1c and 1d show the effect of *scaling*, where n is the number of vectors, d is the dimension, and the color represents the average fraction of “unselectable” key vectors.

Layer Norm 的存在可以让 Attention 不用自己去学习这两点。但是随着模型规模的增大，这个辅助作用是被削弱的，也就是模型能自己学会这两点。但是仍旧从一个比较有意思的角度阐述了 Layer Norm 的作用。

6. 实验论证 3: Layer Norm 在图像中表现怎么样?

2018 年，何凯明提出的《Group Normalization》对 ResNet 中使用不同的 Normalize 方式进行了对比，可以看出 Layer Norm 是不如 Batch Norm 的

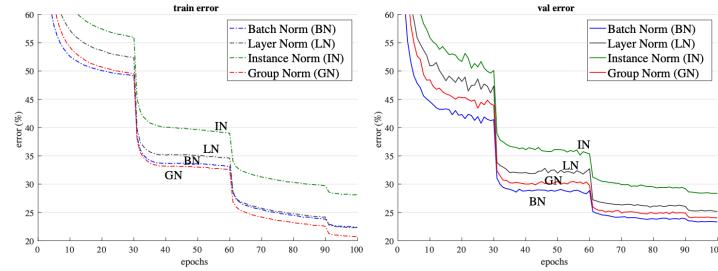


Figure 4. Comparison of error curves with a batch size of 32 images/GPU. We show the ImageNet training error (left) and validation error (right) vs. numbers of training epochs. The model is ResNet-50.

在 Transformers 大火后，CV 领域也开始使用 Transformers，Vit 中使用的就是 Layer Norm.

2022 年的一篇文章，也是何凯明之前所在的 FAIR 小组，发表了论文《A ConvNet for the 2020s》，论文里表示，在借鉴了一些 Vision Transformers 的思想对 ConvNet 修改后，使用 LayerNorm 效果比 Batch Norm 还要好一点。

7. TODO :noexport:

- <https://kyleluther.github.io/2020/02/18/batchnorm-exploding-gradients.html>
- How Does Batch Normalization Help Optimization? <https://arxiv.org/abs/1805.11604>

Transformers 常用的 Normalization 都有什么

1. 题目：

大语言模型常用的 Normalization 都有什么？

2. 答案

目前流行的就两种：LayerNorm 和 RMSNorm。早期的 GLM 系列曾经用过 DeepNorm，后来我印象在 ChatGLM2 的时候就改成 LayerNorm 或者 RMSNorm 了。

如果硬要再细分的话，可以根据 Norm 的位置分为 Pre-LayerNorm, Post-LayerNorm, Pre-RMSNorm, Post-RMSNorm.

至于 Batch Norm 为什么不流行了，可以看上一篇：【TODO】

下面重点看一下 LayerNorm 和 RMSNorm。大模型的 Normalization 的演进过程也挺有意思，像及了毕加索的《公牛》绘画过程。

【image】

研究人员从 LayerNorm 开始，为了提升训练效率也是拼了，一步一步简化成了 RMSNorm 的模样。

3. LayerNorm

其实目前主流的 Normalization 都有个通用的公式

$$\text{Norm}(\mathbf{x}) = \frac{\mathbf{x} - \mu}{\sigma} \cdot \gamma + \beta$$

其中, μ 为均值, σ 为归一化的分母, 比如对 LayerNorm 来说他是标准差, 对 WeightNorm 来说是 L2 范数。 γ 和 β 为可学习的参数, 可以让模型根据分布 scaling 和 shifting。有的文献把 γ 叫做 gain, 把 β 叫做 bias。

对于 Layer Norm 来说, 可以表示为:

$$\text{LayerNorm}(\mathbf{x}) = \frac{\mathbf{x} - \mu}{\sigma} \cdot \gamma + \beta$$

上面公式中, μ 为均值, σ 为标准差。

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

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

Batch Norm 也可以表示成上面的样子, 只不过的计算的维度不同。

Layer Norm 出来以后, 大家发现 Layer Norm 效果很好, 同时也想方设法去改进。在论文《Understanding and Improving Layer Normalization》中, 作者仔细研究了 Layer Norm 公式的各个部分。公式虽然简单, 就看我们能不能发现可研究的点。

最终, 作者做了一些实验, 证明了两点:

- (a) γ 和 β 是数据无关的参数, 如果训练和测试的分布不太一样, 那就会导致 overfitting。所以作者尝试去掉 γ 和 β , 然后发现效果并没有变差, 反而有的会更好。

Models	Machine Translation			Language Modeling		Classification			Parsing
	En-De(+)	De-En(+)	En-Vi(+)	Enwiki8(-)	RT(+)	SST5(+)	MNIST(+)	PTB(+)	
Model Layers	12	12	12	12	4	4	3	3	
w/o Norm	Diverge	34.0	28.4	1.04	76.85	38.55	99.14	88.31	
LayerNorm	28.3	35.5	31.2	1.07	77.21	39.23	99.13	89.12	
LayerNorm-simple	28.4	35.5	31.6	1.07	76.66	40.54	99.09	89.19	

- (b) μ 和 σ 又有什么用呢? 作者通过理论和实验证明了通过减去 μ , 让梯度回到 0 附近。而除以 σ 则让梯度的方差变小。所以 Layer Norm 提升了训练的稳定性。

这篇论文真的是很棒, 建议大家读一下原文。然后作者还提出了把 γ 和 β 变成了可对输入微分的函数, 然后把这个方法叫 AdaNorm。效果也很好, 不过这么做让计算变得复杂了。

4. RMSNorm

上面的 AdaNorm 尝试做了减法, 但是又做了加法, RMSNorm 则在做减法上一条路走到黑。

RMSNorm 的作者认为, 让数据和梯度变成 0 是否是必要的? 作者把 LayerNorm 中的 μ 和 β 去掉(也可以认为是这两个值变为 0), 就得到了 RMSNorm.

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\text{rms}(\mathbf{x})} \cdot \gamma$$

where

$$\text{rms}(\mathbf{x}) = \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2 + \epsilon}$$

然后通过实验证明了这样做效果不仅没有下降，计算效率还提升了 25%

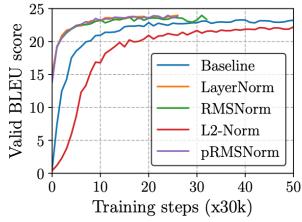


Figure 2: SacreBLEU score on newstest2013 for the RNNSearch. Models are implemented according to *Nematus* [25] in Tensorflow.

Model	Test14	Test17	Time
Baseline	21.7	23.4	$399 \pm 3.40\text{s}$
LayerNorm	22.6	23.6	$665 \pm 32.5\text{s}$
L2-Norm	20.7	22.0	$482 \pm 19.7\text{s}$
RMSNorm	22.4	23.7	$501 \pm 11.8\text{s}$ (24.7%)
pRMSNorm	22.6	23.1	$493 \pm 10.7\text{s}$ (25.9%)

Table 2: SacreBLEU score on newstest2014 (Test14) and newstest2017 (Test17) for RNNSearch using Tensorflow-version Nematus. “Time”: the time in second per 1k training steps. We set p to 6.25%. We highlight the best results in bold, and show the speedup of RMSNorm against LayerNorm in bracket.

5. 还能在优化么？

看上面的公式，似乎能优化的也不多，要么把 γ 去掉，要么再修改 σ 的计算公式，变得更简单一点。

Weight Norm 和 RMSNorm 的工作其实非常像，而且更简单，可以看下公式。

$$\text{WeightNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\|\mathbf{x}\|_2} \cdot \gamma$$

where

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^N x_i^2 + \epsilon}$$

对比一下可以发现，Weight Norm 和 RMSNorm 就差了一个 \sqrt{N} ，但是从上图的效果对比来看，也就是 L2-Norm 那根线，效果和 RMSNorm 还是差不少的。这大概率是因为变长序列带来的影响，Weight Norm 并没有考虑长度的问题。所以去掉 \sqrt{N} 这条路已经走不通了。

那改变其他的归一化方式呢？比如计算更简单的 L1 norm？并不是那么好，因为 L1 norm 在 0 点不可导，虽然可以通过一些 trick 避免，但是总归是不太优雅。如果你有想法，可以评论区留言，说不定下一个标配的 Normalization 方法就是你的。

6. 大模型使用 Normalization 汇总

Models	Norm
T5 (11B)	Pre-RMS
GPT3 (175B)	Layer
mT5 (13B)	Pre-RMS
PanGu- α (200B)	Layer
CPM-2 (198B)	Pre-RMS
Codex (12B)	Pre-Layer
ERNIE 3.0 (10B)	Post-Layer
Jurassic-1 (178B)	Pre-Layer
HyperCLOVA (82B)	Pre-Layer
Yuan 1.0 (245B)	-
Gopher (280B)	Pre-RMS
ERNIE 3.0 Titan (260B)	Post-Layer
GPT-NeoX-20B	Layer
OPT (175B)	-
BLOOM (176B)	Layer
Galactica (120B)	Layer
GLaM (1.2T)	Layer
LaMDA (137B)	Layer
MT-NLG (530B)	Pre-Layer
AlphaCode (41B)	-
Chinchilla (70B)	Pre-RMS
PaLM (540B)	Layer
AlexaTM (20B)	Pre-Layer
Sparrow (70B)	Pre-RMS
U-PaLM (540B)	Layer
UL2 (20B)	-
GLM (130B)	Deep

Transformers 的 Layer Norm 可以并行么?

1. 题目：

Transformers 中的 Layer Norm 可以并行加速么?

这个问题我之前觉得可以加速，而且给出了一个简单的实现方案。后来看 Transformers 的一些 GPU 训练的代码后，才发现我真是 too young too simple, sometimes even naive。

2. 我认为的并行方案

layernorm 的计算，重点就是计算均值和方差。

(a) 求均值

- i. 把集合中的元素分成 k 组，假设每组有 n_k 个元素，每个 thread 分别计算每组的和 s_k
- ii. 把结果聚合再求全局均值。 $\mu = \frac{1}{N} \sum_i^k s_k$

(b) 求方差

- i. 类似的方法，每组求 $(x_i - \mu)^2$ 的和 $m_k = \sum_{i \in \text{thread}_k} (x_i - \mu)^2$
- ii. 然后聚合后再求方差 $\mu = \frac{1}{N} \sum_m^k s_k$

3. 实际上的并行方案

上面的方案当然没什么问题，但是并不是最优的。

上面的算法需要遍历 2 次数据，一次计算均值，一次计算方差。能不能只遍历数据一次就能并行的把均值和方差算出来呢？

相信你会立马想到这个公式：

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

并行的时候，一边算平方和，一边算全部的和。最后平方和与均值都可以算出来，然后按公式一减就出来了，看上去很完美。

但是这个公式只是理论上很完美，受限于计算机计算精度的问题，这个公式当两个平方项都很大的时候，精度会失真，导致算出来的方差很不稳定，甚至有可能是负数。后面会有代码演示数值稳定性的问题。

那能像上一节的算法那样，分别计算均值和方差最后聚合么？似乎有些反直觉，不需要知道全局的均值就可以计算方差。但是我们要相信数学家的折腾能力，搞出了无数匪夷所思的东西。就连加百列号角 (Gabriel's Horn) 这种鬼玩意都能搞出来，数学有无限的可能。(注：加百列号角 Gabriel's Horn 的体积是有限的，但是表面积是无限的。)

Transfromers 无论是在 pytorch，还是在 apex，还是在其他一些加速框架比如 oneflow 中，都采用了 Welford online Algorithm。这个算法是 Welford 在 1962 年发表的《Note on a Method for Calculating Corrected Sums of Squares and Products》中提出。他给出的算法，可以在一个集合新增一个元素的时候，均值和方差的不需要把所有的数都遍历一遍，而是根据之前集合的均值和方差就可以直接计算出来。

而在 1972 年, Chan 发表了《Updating Formulae and a Pairwise Algorithm for Computing Sample Variances》, 可以认为是 Welford Algorithm 的一个升级版本, 可以根据两个集合的均值和方差直接计算出整体的均值和方差。当然如果两个集合中, 某一个集合只有一个元素, 算法就退化成 Welford Algorithm 了。这个算法为大规模并行计算均值和方差提供了理论基础。

由于 Welford's Algorithm 是 Chan's Algorithm 的一个特例, 所以下面简单说一下 Chan's Algorithm 是怎么一回事。

这里首先给出一个定义, 定义与均值差的平方和为

$$M2 = N\sigma^2 = \sum_{x_i \in S} (x_i - \mu)^2$$

假设两个集合 S_1 和 S_2

- S_1 的个数为 n_1 , 均值为 μ_1 , 方差为 σ_1^2 , 与均值差的平方和 $M2_1$
- S_2 的个数为 n_2 , 均值为 μ_2 , 方差为 σ_2^2 , 与均值差的平方和 $M2_2$

均值还是比较好算

$$\mu_{\text{total}} = \frac{n_1\mu_1 + n_2\mu_2}{n_1 + n_2}$$

我们只要知道如何计算 $M2$, 就等于知道如何计算方差, 因为再除以一个元素个数就行了。而 $M2$ 正好有这么一个性质:

$$M2_{\text{total}} = M2_1 + M2_2 + \frac{n_1 n_2 (\mu_1 - \mu_2)^2}{n_1 + n_2}$$

也就是说我们只需要两个集合各自的均值和 $M2$ 我们就可以计算出方差。

上面这个式子怎么来的呢? 我们来证明一下:

$$\begin{aligned}
M2_{\text{total}} &= \sum_{i=1}^{n_1} (x_i - \mu_{\text{total}})^2 + \sum_{i=1}^{n_2} (x_i - \mu_{\text{total}})^2 \\
&= \sum_{i=1}^{n_1} (x_i - \mu_1 + \mu_1 - \mu_{\text{total}})^2 + \sum_{i=1}^{n_2} (x_i - \mu_2 + \mu_2 - \mu_{\text{total}})^2 \\
&= \sum_{i=1}^{n_1} ((x_i - \mu_1)^2 + 2(x_i - \mu_1)(\mu_1 - \mu_{\text{total}}) + (\mu_1 - \mu_{\text{total}})^2) \\
&\quad + \sum_{i=1}^{n_2} ((x_i - \mu_2)^2 + 2(x_i - \mu_2)(\mu_2 - \mu_{\text{total}}) + (\mu_2 - \mu_{\text{total}})^2) \\
&= M2_1 + 0 + n_1(\mu_1 - \mu_{\text{total}})^2 + M2_2 + 0 + n_2(\mu_2 - \mu_{\text{total}})^2 \\
&= M2_1 + M2_2 + n_1(\mu_1 - \mu_{\text{total}})^2 + n_2(\mu_2 - \mu_{\text{total}})^2 \\
&= M2_1 + M2_2 + n_1\left(\mu_1 - \frac{n_1\mu_1 + n_2\mu_2}{n_1 + n_2}\right)^2 + n_2\left(\mu_2 - \frac{n_1\mu_1 + n_2\mu_2}{n_1 + n_2}\right)^2 \\
&= M2_1 + M2_2 + n_1\left(\frac{n_2(\mu_1 - \mu_2)}{n_1 + n_2}\right)^2 + n_2\left(\frac{n_1(\mu_2 - \mu_1)}{n_1 + n_2}\right)^2 \\
&= M2_1 + M2_2 + \frac{(n_1n_2^2 + n_1^2n_2)(\mu_1 - \mu_2)^2}{(n_1 + n_2)^2} \\
&= M2_1 + M2_2 + \frac{n_1n_2(n_1 + n_2)(\mu_1 - \mu_2)^2}{(n_1 + n_2)^2} \\
&= M2_1 + M2_2 + \frac{n_1n_2(\mu_1 - \mu_2)^2}{n_1 + n_2}
\end{aligned}$$

方差 $\sigma^2 = \frac{M2_{\text{total}}}{n_1 + n_2}$ ，再除以一个 $n_1 + n_2$ 即可。

证明完了好像也没那么神奇，陷入了人生三大错觉之一：我上我也行，只恨自己生的太晚。

4. 代码学习

由于 Chan 和 Welford 算法的并行体质，Nvidia 的 Apex 库率先实现了这个方法，叫做 Fused Layer Norm。为啥叫 Fused？因为把所有的计算都融合 (fuse) 到一个核函数里了，不需要与 CPU 来回通信。可以重点看代码开头的 cuWelfordOnlineSum 和 cuChanOnlineSum 两个函数。对应 python 代码入口为 apex.normalization.fused_layer_norm.FusedLayerNorm。代码见：https://github.com/NVIDIA/apex/blob/c3fad1ad120b23055f6630da0b029c8b626db78f/csrc/layer_norm_cuda_kernel.cu#L670
pytorch 后来也实现实现了，可以看 cuWelfordOnlineSum 和 cuWelfordCombine 两个函数，代码见：https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/cuda/layer_norm_kernel.cu

Oneflow 后来又进一步根据输入的大小优化了 Fused Layer Norm 的性能，代码见：https://github.com/Oneflow-Inc/oneflow/blob/master/oneflow/core/cuda/layer_norm.cuh

所以现在我们使用 pytorch 和其他加速库的 layernorm 函数底层已经实现了并行。我们这些调包侠在用 python 写代码的时候，要记住，哪有什么岁月静好，都是 C++ 和 Cuda 大佬们在负重前行。

下面我用 python 模拟了一下 c++ cuda 的实现，同时测试了一下在数字比较大的时候的数值稳定性问题。可以发现，用平方和减去均值平方的方法，方差就算错了，成为了负值。

测试普通数字 ...

全局均值：-0.10384651739409387

```
Welford 并行全局均值: -0.10384651739409384
串行全局方差: 0.8165221946938586
平方差串行全局方差: 0.816522194693858
Welford 并行全局方差: 0.8165221946938584
```

```
测试大数...
全局均值: 999999999.8961536
Welford 全局均值: 999999999.8961536
串行全局方差: 0.8165221933047772
平方差串行全局方差: -512.0
Welford 并行全局方差: 0.8165221874239014
```

核心代码如下，全部的代码实在是有些又臭又长，就放在开篇提到的电子书里了。

```
# 部分代码来自: https://nbviewer.org/github/changyaochen/changyaochen.github.io/blob/master/assets/
import numpy as np
# set random seed
np.random.seed(42)

def native_mean(x):
    N = len(x)
    sum_ = sum(x)

    return sum_ / N

def native_var(x):
    N = len(x)
    mu = native_mean(x)
    sum_ = sum([(e - mu)**2 for e in x])

    return sum_ / N

def semi_native_var(x):
    N = len(x)
    mu = native_mean(x)
    var_ = sum([e **2 for e in x]) / N - mu**2

    return var_

def online_mean(old_mean, new_x, N):
    new_mean = old_mean + (new_x - old_mean) / (N + 1)
```

```

    return new_mean

def welford(old_var, old_mean, new_x, new_mean, N):
    new_var = old_var + ((new_x - old_mean) * (new_x - new_mean) - old_var) / (N + 1)

    return new_var

def welford_combine(val, mean, m2, count):
    """新增一个数"""
    count += 1
    delta1 = val - mean
    mean += delta1 / count
    delta2 = val - mean
    m2 += delta1 * delta2
    return mean, m2, count

def welford_combine_two(b_mean, b_m2, b_count, mean, m2, count):
    """合并两个集合"""
    if b_count == 0:
        return mean, m2, count
    new_count = count + b_count
    nb_over_n = b_count / new_count
    delta = b_mean - mean
    mean += delta * nb_over_n
    m2 += b_m2 + delta * delta * count * nb_over_n
    count = new_count
    return mean, m2, count

# 生成示例数据

def parallel_var(data, chunk_size):

    # 每25个元素计算一次均值和方差
    # chunk_size = 25
    chunk_means = []
    chunk_variances = []
    chunk_counts = []

    for i in range(0, len(data), chunk_size):
        chunk = data[i:i + chunk_size]
        mean = 0.0

```

```

m2 = 0.0
count = 0
for val in chunk:
    mean, m2, count = welford_combine(val, mean, m2, count)
    variance = m2 / count
    chunk_means.append(mean)
    chunk_variances.append(variance)
    chunk_counts.append(count)

# 初始化全局均值、方差和计数
global_mean = 0.0
global_m2 = 0.0
global_count = 0
# 将每组数据合并到全局均值和方差中
for i in range(len(chunk_means)):
    b_mean = chunk_means[i]
    b_m2 = chunk_variances[i] * chunk_counts[i]
    b_count = chunk_counts[i]
    global_mean, global_m2, global_count = welford_combine_two(b_mean, b_m2, b_count, global_mean,
                                                               global_m2, global_count)

# 计算全局方差
global_variance = global_m2 / global_count

return global_mean, global_variance

```

```

N = 100
x = np.random.normal(size=N)
print("测试普通数字...")
global_mean, global_variance = parallel_var(x, 24)
print(f"全局均值: {native_mean(x)}")
print(f"Welford 并行全局均值: {global_mean}")
print(f"串行全局方差: {native_var(x)}")
print(f"平方差串行全局方差: {semi_native_var(x)}")
print(f"Welford 并行全局方差: {global_variance}")
print('-' * 50)

# 大数测试

```

```
x += 1e9
```

```
global_mean, global_variance = parallel_var(x, 25)
print("测试大数...")
print(f"全局均值: {native_mean(x)}")
print(f"Welford 全局均值: {global_mean}")
print(f"串行全局方差: {native_var(x)}")
print(f"平方差串行全局方差: {semi_native_var(x)}")
print(f"Welford 并行全局方差: {global_variance}")
print('-' * 50)
```

Chapter 3: 大语言模型 pipeline

Stage 1: Pretrain

Continue Pretrain

Stage 2: SFT

- SFT 是否必须存在

Stage 3: Alginment

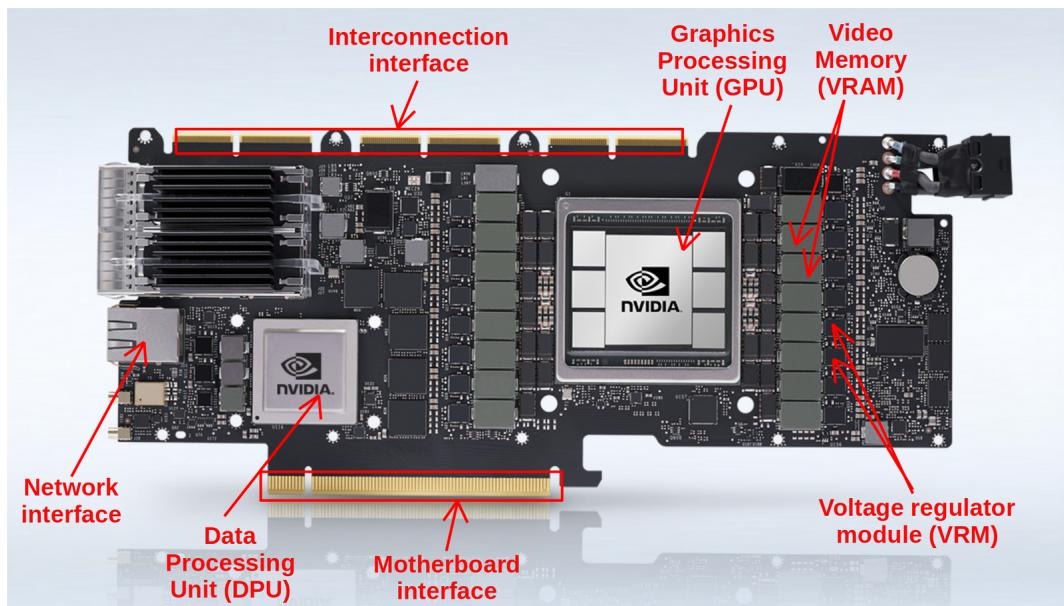
Stage 0: 数据处理

Chapter 4: 大语言模型训练

显卡和模型训练基础知识

显卡的架构说明

1. 一块显卡的基本构成 现在在你面前的是一块 A100 显卡。目前仍然是 AI 领域的香饽饽，性能如此强劲以至于老美对我们禁售了。大模型火之前我印象售价也就七八万的样子，结果现在一张 A100 估计得十几万，而且你还买不到。



如上图所示，一块显卡大概有如下部分组成：

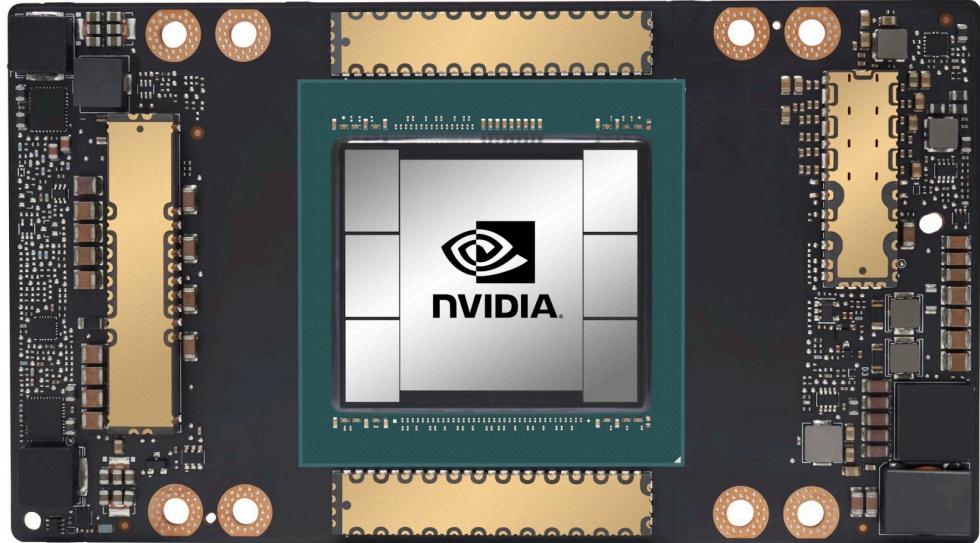
- Graphics Processing Unit (GPU) 图形处理单元
- Video Memory (VRAM) 视频内存
- Motherboard interface 主板接口
- Interconnection interface 互连接口
- Network interface 网络接口
- Data Processing Unit (DPU) 数据处理单元
- Video BIOS (VBIOS) 视频 BIOS
- Voltage regulator module (VRM) 稳压模块
- Output Interfaces 输出接口
- Cooling system 冷却系统

对于我们使用者来说，主要关心前四部分即可，其他部分暂时不需要了解。唯一可能需要讨论的就是男朋友天天吵着要给安装水冷系统，确实挺好，但是先不用管。

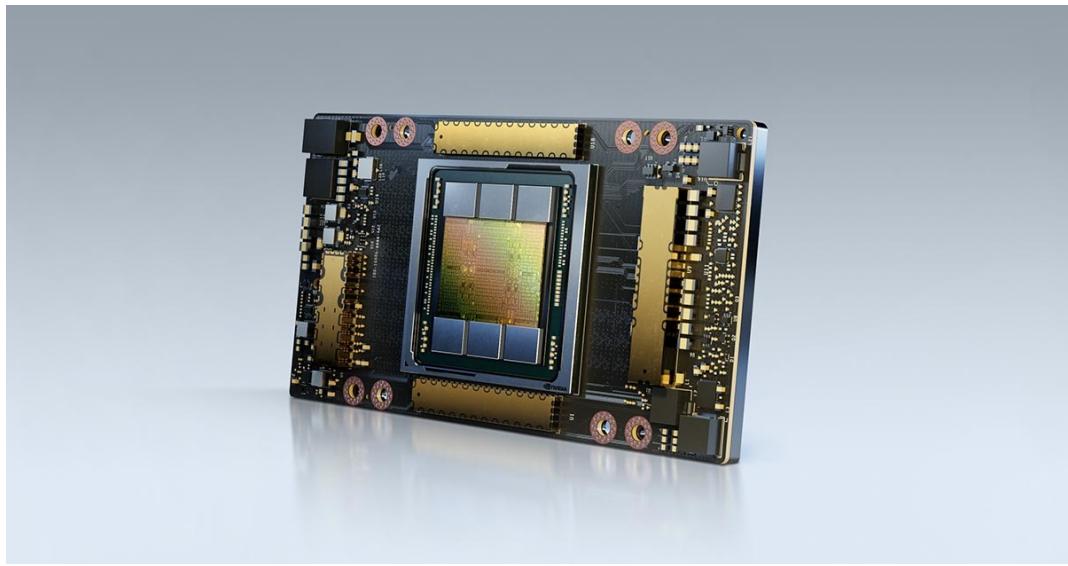
下面分别讲述一下前 4 部分。

2. Graphics Processing Unit (GPU) 图形处理单元

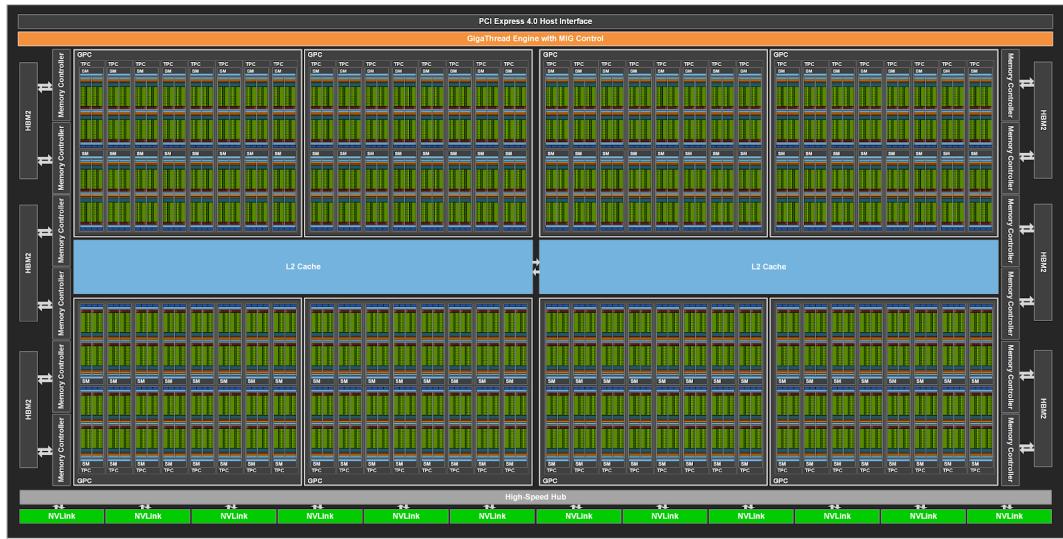
最核心的部件，就是中间的 GPU 模块。如果扣下来，大概长这样：



反过来的样子：



可以看到和 CPU 有点像了。中间金闪闪的就是 GPU Core。如果有能力拆开的话，可以看到下面的架构。当然这个是 Nvidia 绘画的



仔细看的话，可以发现 A100 一共有 8 个 GPC (Graphics Processing Cluster)，每个 GPC 包含 8 个 TPC (Texture Processing Cluster)，而每个 TPC 又包含 2 个 SM (Streaming Multiprocessor)。

所以 A100 一共有 128 个 SM。至于 GPC 和 TPC 暂时可以先不管，会有一些图像处理的能力。

注意，这只是设计的时候有 128 个 SM，但是实际上在芯片这个东西是有良品率的，有时候光刻的时候刻坏了 1 个 SM，那难道把整个板子扔了？所以最终卖向市场的时候统一锁 20 个 SM，这样的话即使 SM 刻坏了几个，也能通过短路的方法保证 108 个 SM 可用。所以市场上 A100 都是 108 个 SM。

如果拿着放大镜，再看看 SM 的结构，是下面这个样子：



可以看出一个 SM 除了上面的 L1 Instruction Cache (一级指令缓存) 和下面的 L1 Data Cache/Share Memory (一级数据缓存)、Tex (纹理缓存, Texture cache)，有 4 个相同的部分。

每一个部分由如下几部分组成：

- Scheduler 调度 (Warp Scheduler 和 Dispatch Unit)
- 寄存器 (Register File)
- 数据加载、存储队列 (LD/ST)

- 指令缓存 (L0 Instruction Cache)
- 特殊函数单元 SFU (Special Function Unit)
- Cuda Core (INT32,FP32,FP64,TENSOR CORE)

(a) CUDA Core

我们经常说 GPU 有多少个 core，其实就是这上面绿色的小方块，这些就是计算的基本单元。由于衡量 GPU 的性能通常用的是浮点数计算，所以一般说的 Cuda Core，指的就是有多少个 FP32 计算单元。数一数就知道，一个 SM 有 64 个 FP32 计算单元，A100 有 108 个 SM，所以一般就说 A100 有 $108 * 64$ 个 Cuda Core.

早期的 GPU 只有 INT32 和 FP32，也就是 32 位的整数和浮点数运算。打游戏足够了，最多再加上 SFU 来计算一些 sin/cos/log/exp 等运算。比如你打 CS，视角旋转了一下，实际上是整个场景要做一个旋转矩阵的运算。

对于一些高精度的计算比如 64 浮点数可以用算法拆分成 FP32 的多步计算，但是后来有一段时间 Nvidia 猛攻高精度计算，所以直接加上了 FP64 计算单元。再往后神经网络来了，天天矩阵运算算不过来，所以 Volta 架构加上了 Tensor Core。

Tensor Core 就是专门对矩阵运算做了很多优化，计算速度激增。看下面的图就知道了

[./images/Turing-Tensor-Core_30fps_FINAL_736x414.gif](#)

Turing 架构的 Tensor Core 新增了 FP32, FP16, INT8, INT4 类型，到了 A100 的 Ampere 架构，增加了 TF32 和 FP64，还有稀疏矩阵运算。可以说 Nvidia 能在 AI 领域横行霸道，Tensor Core 功不可没。

(b) Warp Scheduler

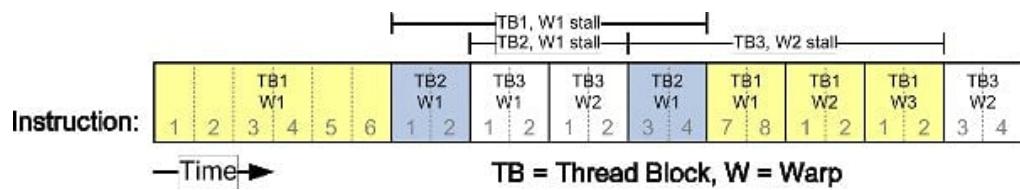
这个负责 Cuda Core 的调度。

在 GPU 里，所有的 Threads 都会被分割成 32 个一组的 Warp，可以看到这里有 4 个 Warp Scheduler，然后会根据每个 Warp 执行的指令和状态分配到对应的 Cuda Core 上。

比如需要 LD/ST，或者需要进入 SFU 进行计算，或者送到 Tensor Core 矩阵运算，或者就是单纯的 INT32 或者 FP32 计算。

计算的时候也是分时计算的，比如只有 16 个 Cuda Core 可用，那一个 Warp 的指令需要 2 个周期来完成。如果只有 2 个 Cuda Core 可用（比如 FP64），那就需要 16 个周期来完成这个 Warp 的指令。

下面是一个 Warp 轮转的例子。



感谢 CUDA 工具帮我们隐藏了这么多的细节，不然程序写起来那可费了老劲了。但是真正想要发挥 GPU 的最大性能，还得知道工作的原理，比如已经成为标配的 Flash Attention，就是很好的优化了一些默认 GPU 计算的瓶颈，从而达到了好几倍的计算速度提升。

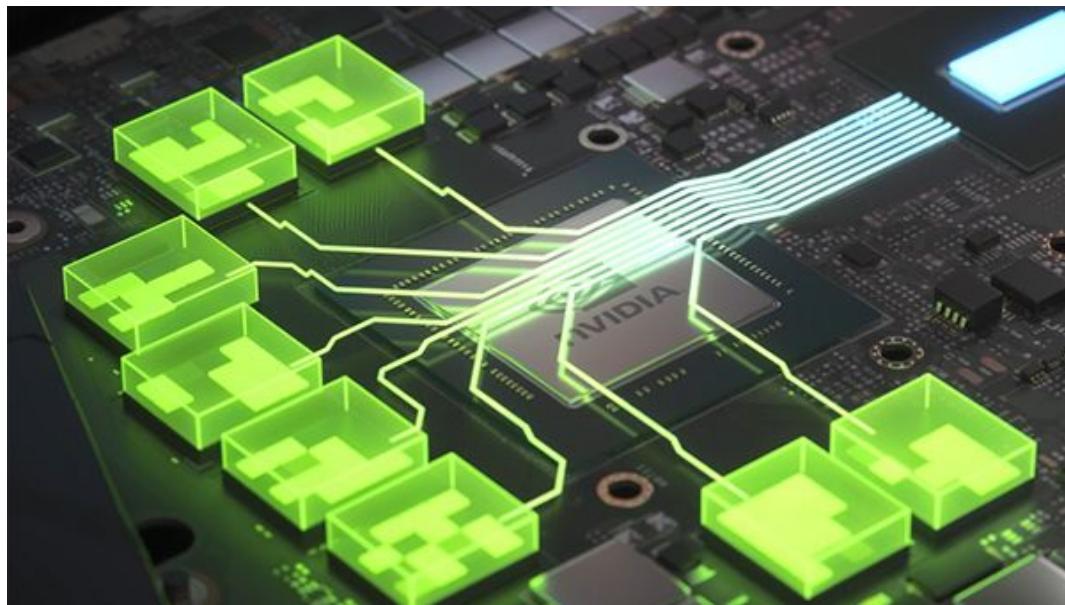
(c) 全局的调度和通信

这么多的 SM 会同意受全局的调度器 (GigaThread Engine) 的指挥，同时还有 L2 Cache 进行通信。

这基本上就是 GPU 的大致结构。

(d) 光线追踪 (RT Core) 如果一款 GPU 是面向游戏市场的话，那么会牺牲一些计算单元，新加一个 RT Core。

3. Video Memory (VRAM) 视频内存 也就是大家经常说的显存。比如 A100 有 80G 和 40G 两款。



我们日常电脑用的内存都是 DDR 的标准，显卡的显存大部分都是 GDDR 或者 HBM。总体来说，HBM 的标准更快，但是更贵。

有多快呢？可以看下 A100 的参数：

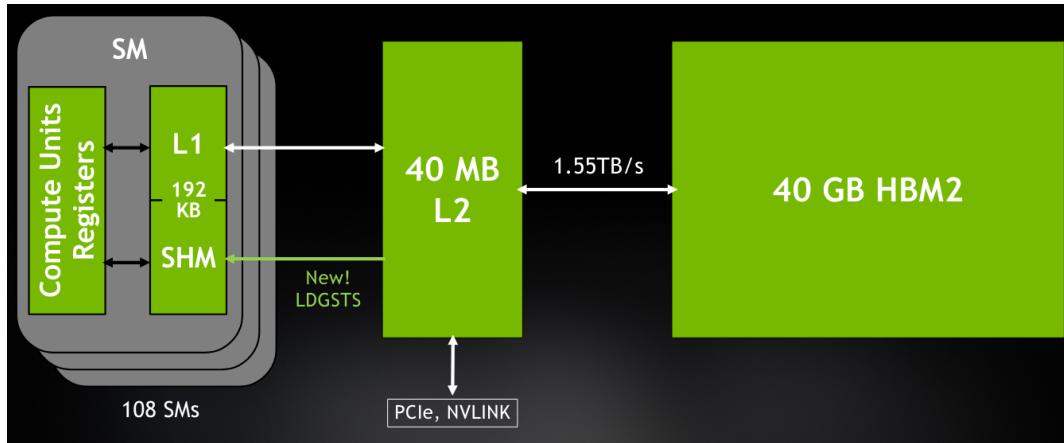
NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64		9.7 TFLOPS		
FP64 Tensor Core		19.5 TFLOPS		
FP32		19.5 TFLOPS		
Tensor Float 32 (TF32)		156 TFLOPS 312 TFLOPS*		
BFLOAT16 Tensor Core		312 TFLOPS 624 TFLOPS*		
FP16 Tensor Core		312 TFLOPS 624 TFLOPS*		
INT8 Tensor Core		624 TOPS 1248 TOPS*		
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs		NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs	

* With sparsity

** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

A100 80G 的 GPU 和显存的通信带宽可以达到 2T/s, 40G 的版本可以达到 1.5T。而 3090 的显存带宽为 936GB/s。下图是一个显存 40G A100 的通信示意图：

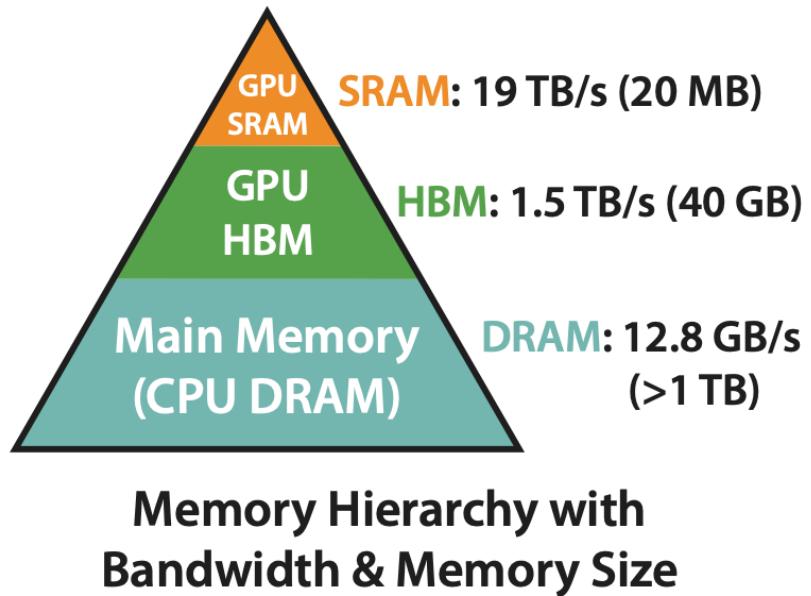


4. Motherboard interface 主板接口

GPU 最终还是要受到 CPU 的指挥，也要从 CPU 那里获取数据，所以必须要与 CPU 通信。现在的通信标准基本上就是 PCIe 接口。其每一代的通信速度如下

	Bandwidth	Gigatransfer	Frequency
PCIe 1.0	8 GB/s	2.5 GT/s	2.5 GHz
PCIe 2.0	16 GB/s	5 GT/s	5 GHz
PCIe 3.0	32 GB/s	8 GT/s	8 GHz
PCIe 4.0	64 GB/s	16 GT/s	16 GHz

可以看出这个速度和前面与显卡的通信速度简直不能比，所以这里面就可以做很多优化，比如 Flash Attention 给的示意图

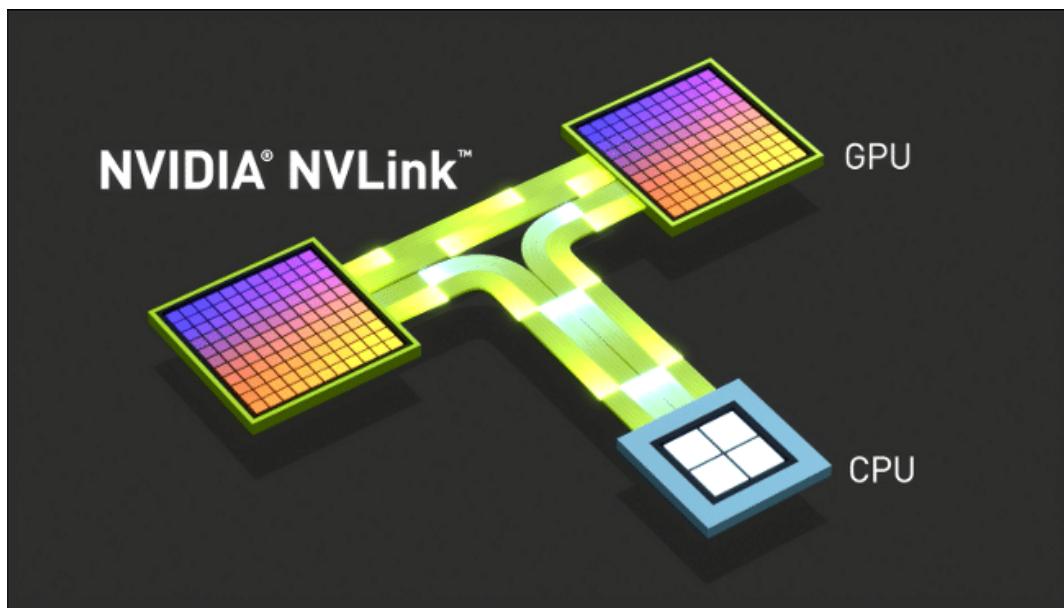


看来他们没钱使用最新的 PCIe 4.0 [捂脸]

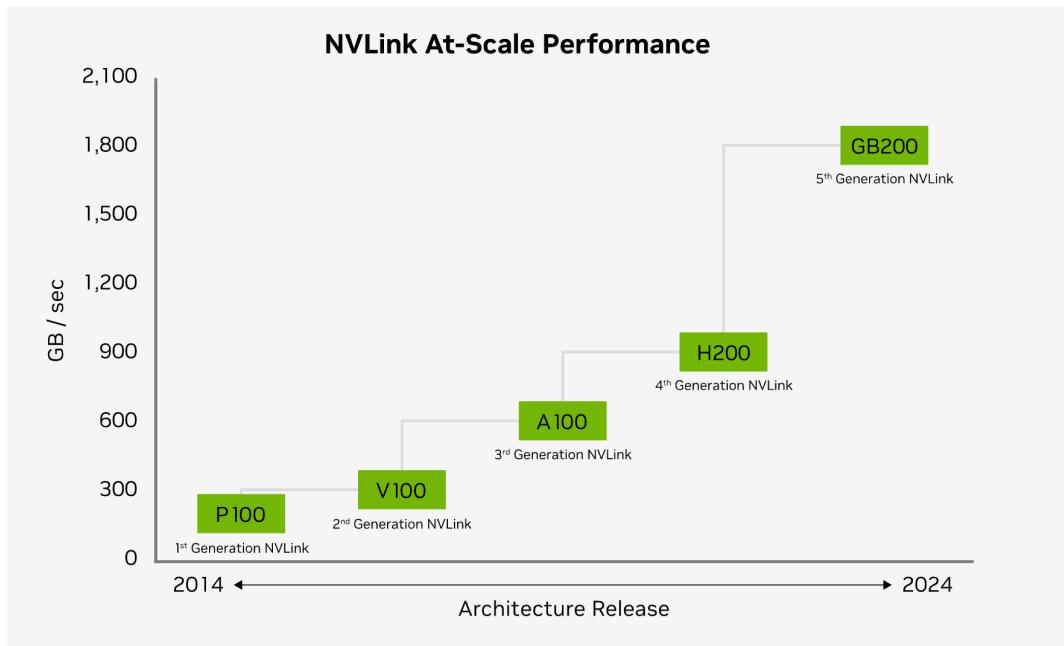
5. Interconnection interface 互连接口

现在大模型训练基本都是需要多机多卡，比如一个模型的参数可能拆分到多个 GPU 上，各种梯度和中间的结果进行 reduce 和 gather 就需要通信，如果都借助 CPU 的 PCI 进行通信，那就太慢了，最高也就是 PCIe Gen4: 64GB/s。

所以 Nvidia 开发了 NVLink 和 NVSwitch，两个 A100 GPU 之间可以达到 600G/s. 而 NVSwitch 则可以连接 8 张卡让他们之间的通信速度达到 600G/s。



而 H100 则更快，达到了 900G/s，下面是 NVLink 的发展图



由于 A100 在华禁售，所以 Nvidia 推出了 A100 的阉割版 A800，和 A100 的唯一区别就是这里的通信带宽被砍到了 400G/s。这就像是百度网盘和迅雷一样恶心，数据已经算完了，明明可以以极快的速度传输，但是你就是要等着。

6. GPU 集群通信

现在上万张卡怎么通信呢？那就只能借助高性能网络通信了。比如 llama2 和 3 里面提到他们有两个集群。分别为：

- InfiniBand: 用于高性能计算集群的高速网络通信协议，提供低延迟和高带宽的通信，适合跨节点的大规模 GPU 集群。常见的 InfiniBand 版本包括 HDR (200 Gbps) 和 NDR (400 Gbps)。
- RoCE (RDMA over Converged Ethernet): 另一种高性能网络技术，允许在以太网上实现远程直接内存访问 (RDMA)，提供低延迟、高带宽的通信。

然后报告里还提到，RoCE 我们还用的起，InfiniBand 实在是有点扛不住。

7. 总结

本文算是一篇 GPU 结构的简单介绍，这些信息对学习大模型还是很有用的。觉得有帮助可以关注一波，点个赞什么的。

看完了之后你就知道大概怎么选购 GPU 了，到底要看哪些参数。

- 业务中整形运算比较多，就选 INT32 core 多的
- 如果是浮点数计算多，就选 FP32 core 多的
- 如果是科学计算，就选 FP64 多的
- 如果是矩阵运算，就选 Tensor Core 多且功能多的
- 如果打游戏就选择有 RT Core 的
- 如果需要搭建集群，还要关注每个环节的通信吞吐量。

CUDA 编程入门

我上小学的时候，学校在偏远山区。学校也没钱请保洁，所以每次开学第一件事情，就是去操场上捡垃圾。

校长就类似 CPU，把“捡垃圾”的函数 (kernel) 发送给教务处。教务处就类似 GPU，教务处把 10 个班级 (Blocks) 叫过来，打包成一个 Grid，取名为“拾酷跑团”。班级 (Blocks) 里的每个学生 (Thread) 都被叫到操场上站成一排，能把操场的一个边占满。

当校长喊：“目标是操场另一边，开始！”后，每个学生 (Thread) 一边往前走，一边捡起操场上的垃圾。等每个学生 (Thread) 都走到操场对面时，整个操场的每一寸土地都被我们小学生征服过，整个校园焕然一新。我们校长简直就是使用 GPU 的高手！

捡垃圾的过程中，班级 (Block) 内会有一些协作，比如某个同学手里实在是拿不过来了，让班里其他同学帮忙拿一拿，你让其他班级 (Blocks) 帮你拿，人家才不理你（除非你长的好看）。

上面这个故事看懂了，就已经知道 GPU 的工作原理了。如果想再详细的了解一下，可以再往下看。

聊到 GPU 这个话题，不由得佩服长者的智慧。在 2008 年 10 月，江总书记在《新时期我国信息技术产业的发展》上发表的文章里，里面有提到要大力发展 GPU 和高性能计算。

4.2.2 高效能计算 计算技术是信息技术产业的核心。发展高效能计算是做强我国信息技术产业的重大战略选择,可考虑以高性能CPU、GPU、高效能超级计算机、网络计算技术为重点,实现能力计算和容量计算同步发展,大幅提高综合信息处理效能,力争在高效能计算领域达到国际先进水平。

(1) 高性能CPU和GPU。CPU直接决定计算机系统性能。为满足我国计算机产业自主发展需要,应当集中力量攻克高性能CPU芯片。适应多核化趋势,开发高性能多核(multi-core)和众核(many-core)CPU,发展高性能嵌入式CPU,使自主的高性能CPU芯片在国际上占有一席之地。GPU主要是用于处理图像的专用芯片,最近它强大的浮点运算性能也受到广泛重视,应当加大GPU的研究开发力度。

(2) 高效能超级计算机。超级计算机是大型复杂计算的必要技术手段,在国民经济和尖端科学领域具有重要作用。需要自主研发综合效能更高的超级计算机,掌握高效能计算、海量存储和低功耗设计等技术,同时应开发配套的大型应用系统。根据国民经济和国防建设对高效能超高速计算的需求,适时

② 硅基新材料主要有绝缘体硅(SOI, Silicon On Insulator)、应力硅、锗硅BiCMOS等;非硅基新材料主要有砷化镓、磷化铟、氮化镓等

要知道这个时间点 Nvidia 等公司还是专注于图像处理, AlexNet 还要等 4 年才能发表。

1. Nvidia 自己的解释

关于 GPU 的优点，Nvidia 在 2009 年的时候，曾经做了一次非常形象的演示。可以看下面视频。

【Video】

当然上面的演示里面 CPU 显得有点弱智了，实际上 CPU 是比 GPU 更“聪明”的。

GPU 之前的市场主要是游戏市场，但是深度学习和比特币出来以后，大家发现 GPU 真的是太适合做这两件事了，所以 Nvidia 一飞冲天。搞 GPU 的公司有很多，为啥就 Nvidia 搞起来了呢？我个人觉得有三点：

- (a) Nvidia 的飞轮跟 Intel 有点像，目前运转良好。
- (b) Nvidia 为了卖显卡，其实会跟很多厂商有深度的合作，甚至会根据厂商的需求定制和赠送一些解决方案。这样 Nvidia 对市场的需求是很了解的。
- (c) 打造了 Cuda 的生态，降低开发者的使用门槛。这点真的是太重要了。

下面会简单讲一下 GPU 的原理和 Cuda，这对我们了解大模型如何训练和推理至关重要。但是本文也不是一个手把手入门教程，更多的是提供一些原理性的解释，先从宏观的角度了解 GPU 是怎么运行的。

2. GPU 的简易工作原理

CPU 实际上是串行的执行任务，就像上面视频演示的那样，只不过操作系统帮我们隐藏了这一点。

老早的时候，CPU 都是单核的，但是给人的感觉是在并行的处理很多任务，比如我们可以一边写代码一边听歌，电脑自己还在收邮件，接收群里发来的消息。但实际上这些事情都被切成一个个的小任务，CPU 根据优先级，这个任务做一下，那个任务做一下。做的足够快以至于我们没有感觉到。就像电影，只要 24 帧以上肉眼就没有卡顿感了。这个技术就是约翰·麦卡锡（John McCarthy）在 1955 年提出的分时系统。插句题外话，一年后，约翰·麦卡锡组织了达特茅斯会议，提出了 AI 的概念。

后来 CPU 又出现了双核，16 核等，并行度有提升，但是并不是 CPU 的主要卖点（超算除外）。

CPU 的设计是全能型的，非常擅长处理复杂的任务，所以设计上有非常复杂的逻辑控制，昂贵的缓存，还有分支预测、乱序执行等高级天赋，还要负责整个系统的通信。

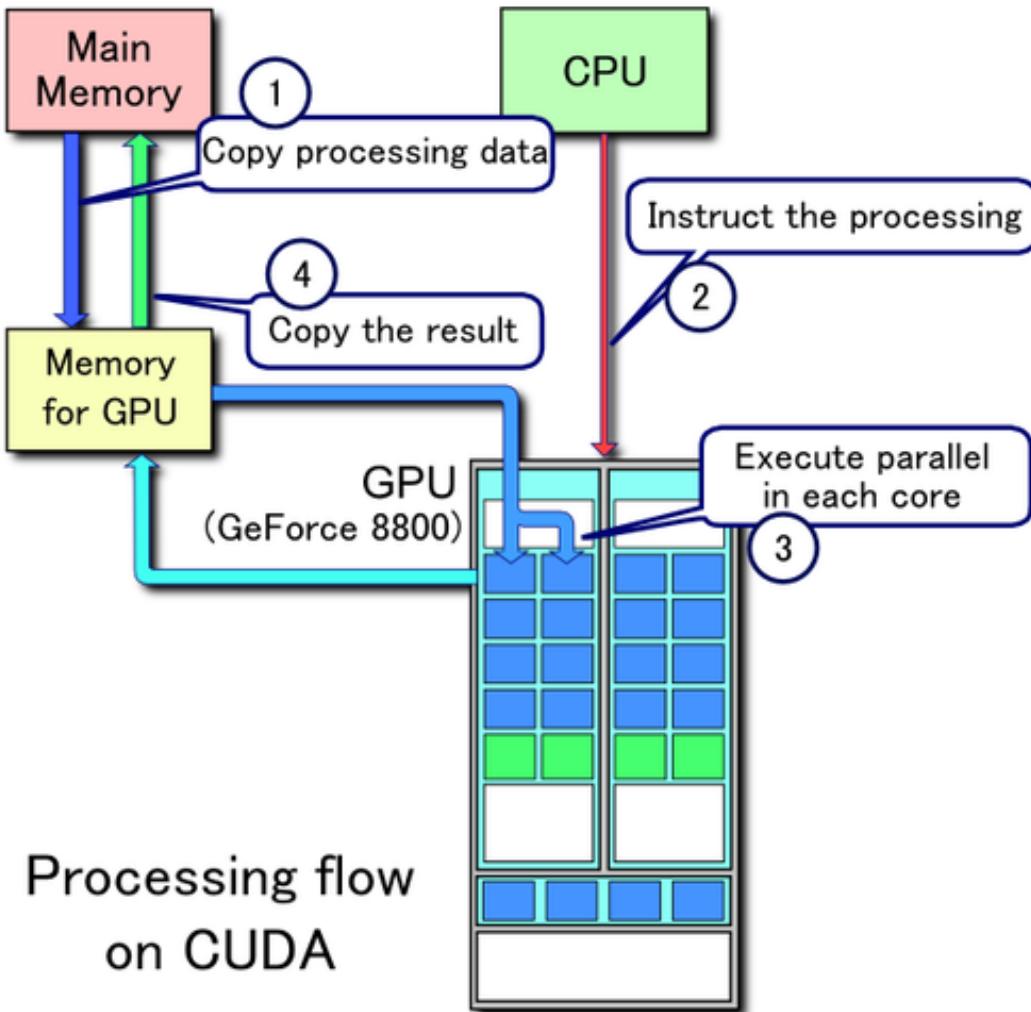
GPU 则主打一个偏科，没有那么多高级天赋，设计极其简单，高级天赋一个不点，就把计算和通信这两个天赋点加到了极致。比如现在比较新 i9 CPU，售价 5000 左右，一共 24 个核。而 3090 显卡售价 1 万多，一共有 10496 个核。

如果只是计算任务的话，CPU 说：“我要打 10 个”，GPU 说：“不好意思，我有一万多个，你还打不打？”

但是 GPU 头脑过于简单，只能执行计算这种简单任务，所以 GPU 天天被 CPU PUA。

GPU 运行大概的流程是这样的：

- (a) CPU 把数据扔给 GPU。这里会有一次内存搬运的工作，将数据从主机 (host) 内存拷贝到显卡 (device) 的显存里。
- (b) CPU 把运行的指令扔给 GPU。
- (c) GPU 根据指令对数据进行处理。
- (d) CPU 把 GPU 运行的结果搬回主机 (host)。这是第二次内存搬运。



这只是最简单的一个流程，现实应用可能要复杂的多，比如多机多卡的时候还有 GPU 之间的通信。GPU 内部还有很多计算的细节，这个我们后面再说。

CPU 和 GPU 的通信，还有 GPU 之间的通信，很容易成为整个系统的瓶颈，所以 GPU 的吞吐量也是衡量 GPU 很重要的指标。

3. CUDA 简介

CUDA 全称为 Compute Unified Device Architecture，它为开发人员提供了 GPU 的开发环境，隐藏了很多硬件实现和驱动的细节。

举一个简单的例子，两个向量 a 和 b 相加。假设向量的长度为 8192。

CPU 能算么？当然也能算，但是 CPU 的一个核的话那就要顺序执行 8192 次加法。

```
for(int i = 0; i < 8192; ++i)
    c[i] = a[i] + b[i];
```

GPU 怎么计算呢？因为有 1 万多个核，我第一个核执行 $c[0] = a[0] + b[0]$ ，第 k 个核执行 $c[k] = a[k] + b[k]$ 。由于是并行执行的，假设并行度是 8192，那么 GPU 可以在一次加法的时间把所有计算完成。

上面是一个理想的情况，在 CUDA 的框架上怎么完成这个任务呢？

我们要写一个在 GPU 上运行的函数，这个函数叫做 Kernel。

然后 CUDA 提供了三个层次结构来并行的运行这些函数 (Kernels)，分别是 Threads、Blocks 和 Grid。

- Threads

逻辑上 GPU 运行的最小单元，Kernels 会并行的在 Threads 中运行。比如向量相加的任务中，每个 Threads 只负责一个下标的运算。

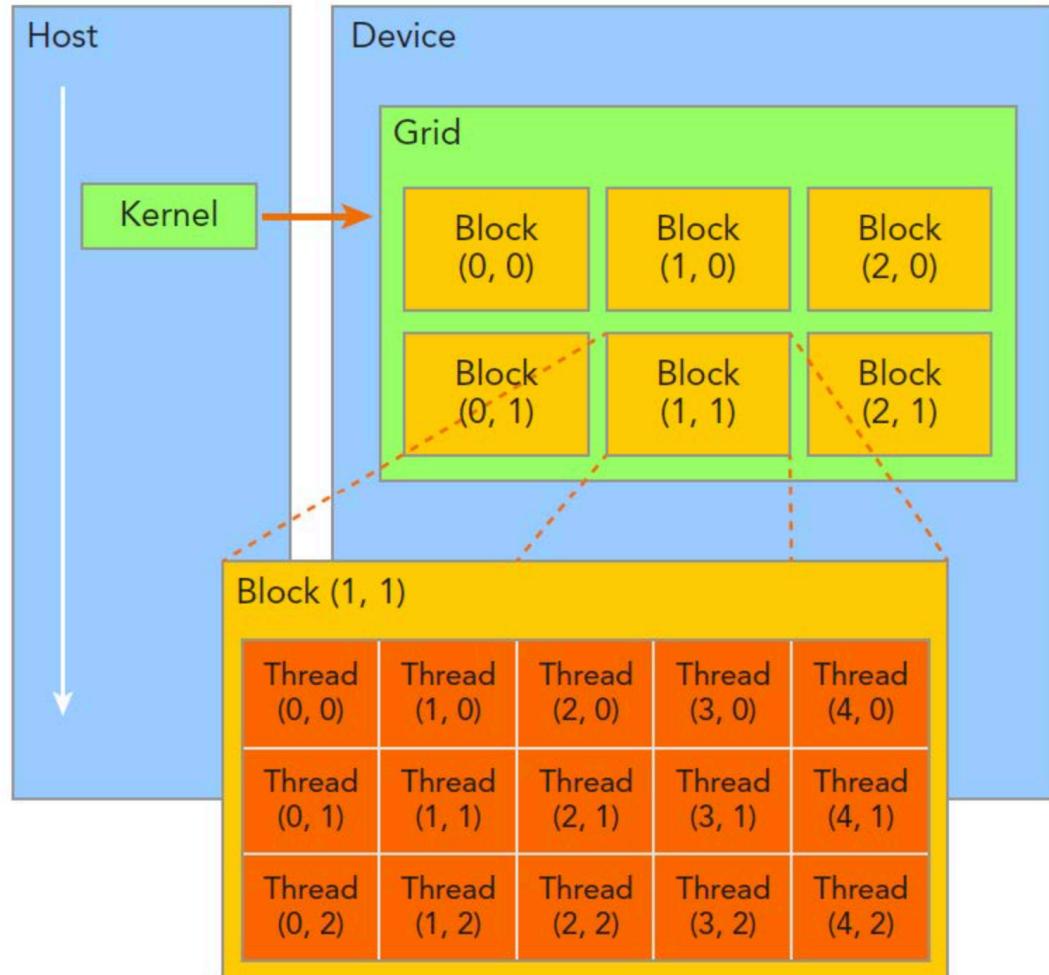
- Blocks

由一组 Threads，Blocks 内的 Threads 可以通过共享内存进行通信，这样可以更加灵活的处理一些 Threads 协作的情况。Blocks 内的共享内存访问速度要比访问全局内存快的多。

- Grid

多个 Blocks 组成一个 Grid。一个 Kernel 只有 1 个 Grid。

其结构如下：



我们举个例子来类比一下 CUDA 的层次结构。

我上小学的时候，学校在偏远山区。学校也没钱请保洁，所以每次开学第一件事情，就是去操场上捡垃圾。

校长就类似 CPU，把“捡垃圾”的函数 (kernel) 发送给教务处，教务处就类似 GPU。教务处把 10 个班级 (Blocks) 叫过来，打包成一个 Grid，取名为“拾酷跑团”。班级 (Blocks) 里的每个学生 (Thread) 都被叫到操场上站成一排。

当校长喊：“目标是操场另一边，开始！”后，每个学生 (Thread) 一边往前走，一边捡起操场上的垃圾。等每个学生 (Thread) 都走到操场对面时，整个操场的每一寸土地都被我们小学生征服过，整个校园焕然一新。

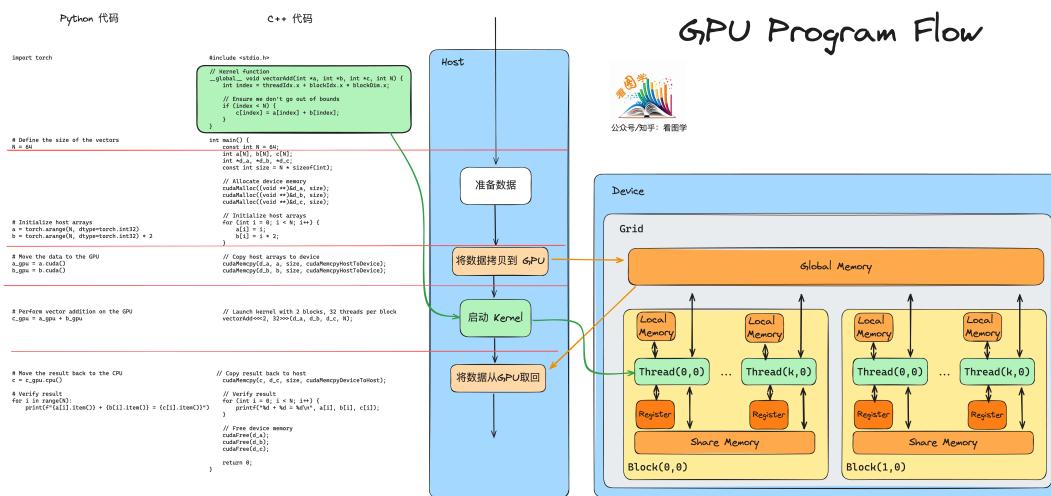
我们校长简直就是使用 GPU 的高手！

当然捡垃圾的过程中，班级 (Block) 内会有一些协作，比如手里实在是拿不过来了，让班里其他同学帮忙拿一拿，你让其他班级 (Blocks) 帮你拿，人家才不理你，除非你长的好看。但是在 GPU 的世界里，大家长的都一样，所以 Block 之间一般是没有通信的。

4. CUDA 的执行流程

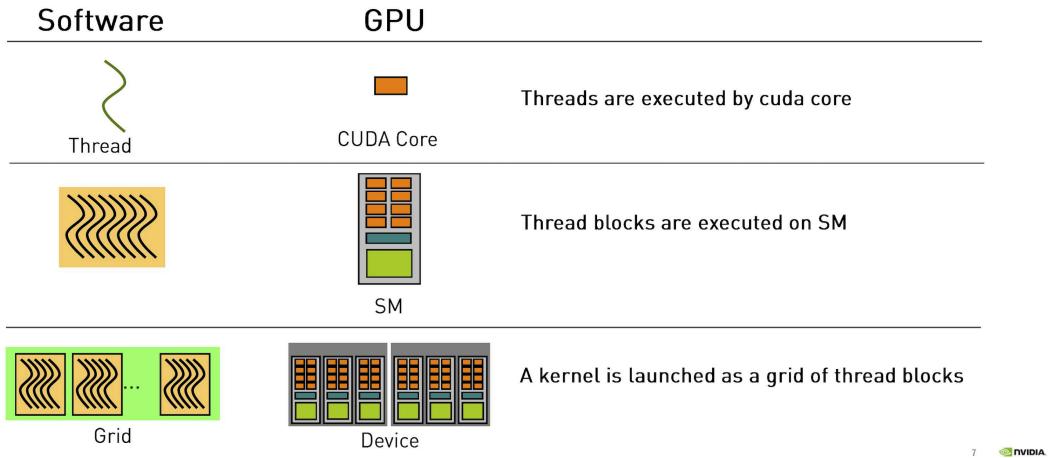
下面举一个简单的向量相加的例子。大多数算法的同学可能都是通过 pytorch 来实现的，然而 pytorch 底层则最终会以 C/C++ 来编写。

所以下图中同时给出了 pytorch 和 Cuda 的代码。但是这两段代码只是功能性一样，实际 pytorch 的实现要比着复杂的多。



5. 硬件层面的视角

虽然 CUDA 给开发者提供了很多便利，但是仍然有大量的 GPU 实现细节被隐藏。CUDA 和 GPU 硬件在层次上的关系大致如下：



从 GPU 的结构来看，也可以大致分成三层结构，依次为 Device / SM / Cuda Core.

这里的 SM 指的是 streaming multiprocessor，它包含一些 CPU 类似的功能，比如有 warp scheduler, register, shared memory 等。

比如一张 3090 (Device) 有 82 个 SM，每个 SM 管理 128 个 Cuda Core。所以一共有 10496 个 core。在开发者眼里，可能觉得 GPU 运行的最小单元是 threads，我开 1 个 thread 就只跑一个 thread。但是 SM 在设计的时候，把 32 个 Threads 放到一起执行，称作 warp。32 这个数字是 Nvidia 这么设计的，人家设计成这样我们也只能这样用。

所以对于 GPU 来说，warp 才是运行和调度的基本单元，也就是说一个 warp(32 个 Threads) 每次都要一起运行。铁索连环了。

为什么要有 warp scheduler 呢？是因为有时候 kernel 执行了一半暂时不计算了，比如在加载数据，这个时候就把计算资源腾出来给别的 warp 用。所以宏观上开发人员可以按照 Grid/Blocks/Threads 这么编写程序，但是微观上，SM 也是以分时系统在运行的。

warp 的这种设计，有时候就会有问题。比如向量相加的时候，向量有 65 个。这 65 个可以分成 3 个 warp，最后一个只处理 1 个元素，剩余 31 个 threads 就浪费了。所以为什么模型设计的 size 通常都是 32 的倍数。

然后还有因为 kernel 的分支导致的 Warp divergence 问题等，这些都是在 GPU 编程时要考虑的问题。

关于硬件的细节实在是有点多，只能先写个大概，等周三再更新一篇文章粗略写一下 Nvidia 显卡的架构设计。

多卡并行训练

当前流行训练框架

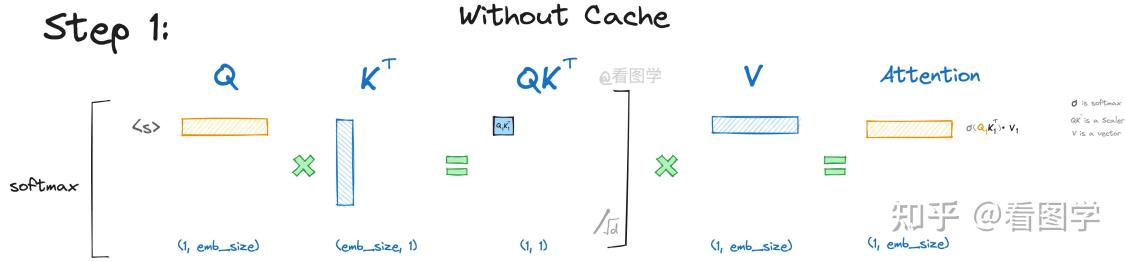
Chapter 5: 大语言模型推理

KV Cache

KV Cache 是 Transformer 标配的推理加速功能，transformer 官方 use_cache 这个参数默认是 True，但是它只能用于 Decoder 架构的模型，这是因为 Decoder 有 Causal Mask，在推理的时候前面已经生成的字符不需要与后面的字符产生 attention，从而使得前面已经计算的 K 和 V 可以缓存起来。

我们先看一下不使用 KV Cache 的推理过程。假设模型最终生成了“遥遥领先”4 个字。

当模型生成第一个“遥”字时，input=“<s>”，“<s>”是起始字符。Attention 的计算如下：

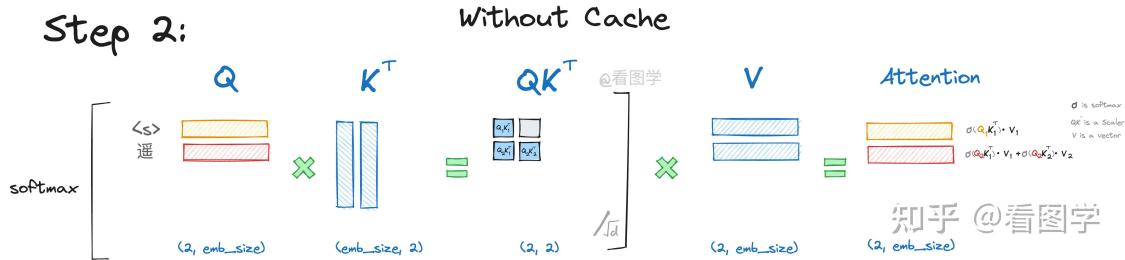


为了看上去方便，我们暂时忽略 scale 项 \sqrt{d} ，但是要注意这个 scale 面试时经常考。

如上图所示，最终 Attention 的计算公式如下，(softmaxed 表示已经按行进行了 softmax)：

$$Att_1(Q, K, V) = \text{softmax}(Q_1 K_1^T) \vec{V}_1 = \text{softmaxed}(Q_1 K_1^T) \vec{V}_1$$

当模型生成第二个“遥”字时，input=“<s> 遥”Attention 的计算如下：



当 QK^T 变为矩阵时，softmax 会针对 行进行计算。写详细一点如下，softmaxed 表示已经按行进行了 softmax。

$$\begin{aligned} Att_{step2}(Q, K, V) &= \text{softmax}\left(\begin{bmatrix} Q_1 K_1^T & -\infty \\ Q_2 K_1^T & Q_2 K_2^T \end{bmatrix}\right) \begin{bmatrix} \vec{V}_1 \\ \vec{V}_2 \end{bmatrix} \\ &= \left(\begin{bmatrix} \text{softmax}(Q_1 K_1^T) & \text{softmax}(-\infty) \\ \text{softmax}(Q_2 K_1^T) & \text{softmax}(Q_2 K_2^T) \end{bmatrix}\right) \begin{bmatrix} \vec{V}_1 \\ \vec{V}_2 \end{bmatrix} \\ &= \left(\begin{bmatrix} \text{softmax}(Q_1 K_1^T) & 0 \\ \text{softmax}(Q_2 K_1^T) & \text{softmax}(Q_2 K_2^T) \end{bmatrix}\right) \begin{bmatrix} \vec{V}_1 \\ \vec{V}_2 \end{bmatrix} \quad \$\$ \\ &= \left(\begin{bmatrix} \text{softmax}(Q_1 K_1^T) \times \vec{V}_1 + 0 \times \vec{V}_2 \\ \text{softmax}(Q_2 K_1^T) \times \vec{V}_1 + \text{softmax}(Q_2 K_2^T) \times \vec{V}_2 \end{bmatrix}\right) \\ &= \left(\begin{bmatrix} \text{softmax}(Q_1 K_1^T) \times \vec{V}_1 \\ \text{softmax}(Q_2 K_1^T) \times \vec{V}_1 + \text{softmax}(Q_2 K_2^T) \times \vec{V}_2 \end{bmatrix}\right) \end{aligned}$$

假设 $\text{Att}_1(Q, K, V)$ 表示 Attention 的第一行， $\text{Att}_2(Q, K, V)$ 表示 Attention 的第二行，则根据上面推导，

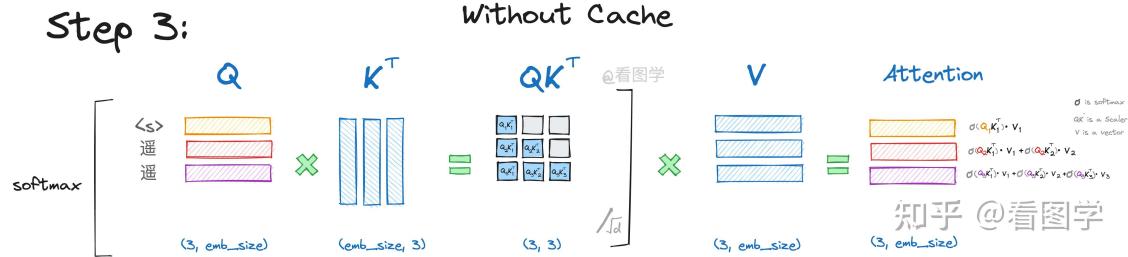
其计算公式为：

$$\begin{aligned} \text{Att}_1(Q, K, V) &= \text{softmax}(Q_1 K_1^T) \vec{V}_1 \\ \text{Att}_2(Q, K, V) &= \text{softmax}(Q_2 K_1^T) \vec{V}_1 + \text{softmax}(Q_2 K_2^T) \vec{V}_2 \end{aligned}$$

你会发现，由于 $Q_1 K_2^T$ 这个值会 mask 掉

- Q_1 在第二步参与的计算与第一步是一样的，而且第二步生成的 V_1 也仅仅依赖于 Q_1 ，与 Q_2 毫无关系。
- V_2 的计算也仅仅依赖于 Q_2 ，与 Q_1 毫无关系。

当模型生成第三个“领”字时，input="<s> 遥遥" Attention 的计算如下：

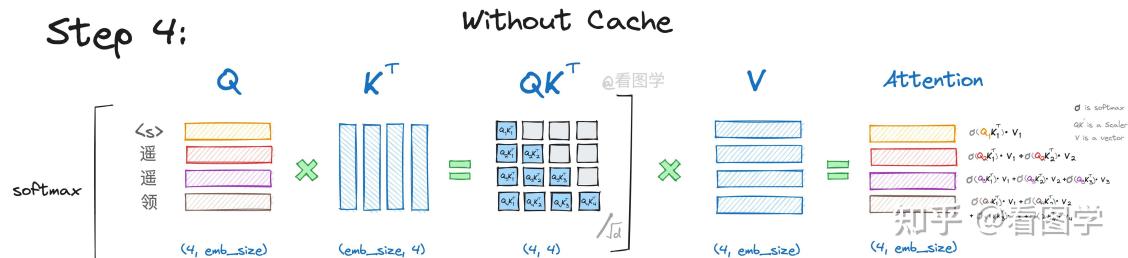


详细的推导参考第二步，其计算公式为：

$$\begin{aligned} \text{Att}_1(Q, K, V) &= \text{softmax}(Q_1 K_1^T) \vec{V}_1 \\ \text{Att}_2(Q, K, V) &= \text{softmax}(Q_2 K_1^T) \vec{V}_1 + \text{softmax}(Q_2 K_2^T) \vec{V}_2 \\ \text{Att}_3(Q, K, V) &= \text{softmax}(Q_3 K_1^T) \vec{V}_1 + \text{softmax}(Q_3 K_2^T) \vec{V}_2 + \text{softmax}(Q_3 K_3^T) \vec{V}_3 \end{aligned}$$

同样的， Att_k 只与 Q_k 有关。

当模型生成第四个“先”字时，input="<s> 遥 遥 领" Attention 的计算如下：



$$\begin{aligned} \text{Att}_1(Q, K, V) &= \text{softmax}(Q_1 K_1^T) \vec{V}_1 \\ \text{Att}_2(Q, K, V) &= \text{softmax}(Q_2 K_1^T) \vec{V}_1 + \text{softmax}(Q_2 K_2^T) \vec{V}_2 \\ \text{Att}_3(Q, K, V) &= \text{softmax}(Q_3 K_1^T) \vec{V}_1 + \text{softmax}(Q_3 K_2^T) \vec{V}_2 + \text{softmax}(Q_3 K_3^T) \vec{V}_3 \\ \text{Att}_4(Q, K, V) &= \text{softmax}(Q_4 K_1^T) \vec{V}_1 + \text{softmax}(Q_4 K_2^T) \vec{V}_2 + \text{softmax}(Q_4 K_3^T) \vec{V}_3 + \text{softmax}(Q_4 K_4^T) \vec{V}_4 \end{aligned}$$

和之前类似，不再赘述。

看上面图和公式，我们可以得出结论：

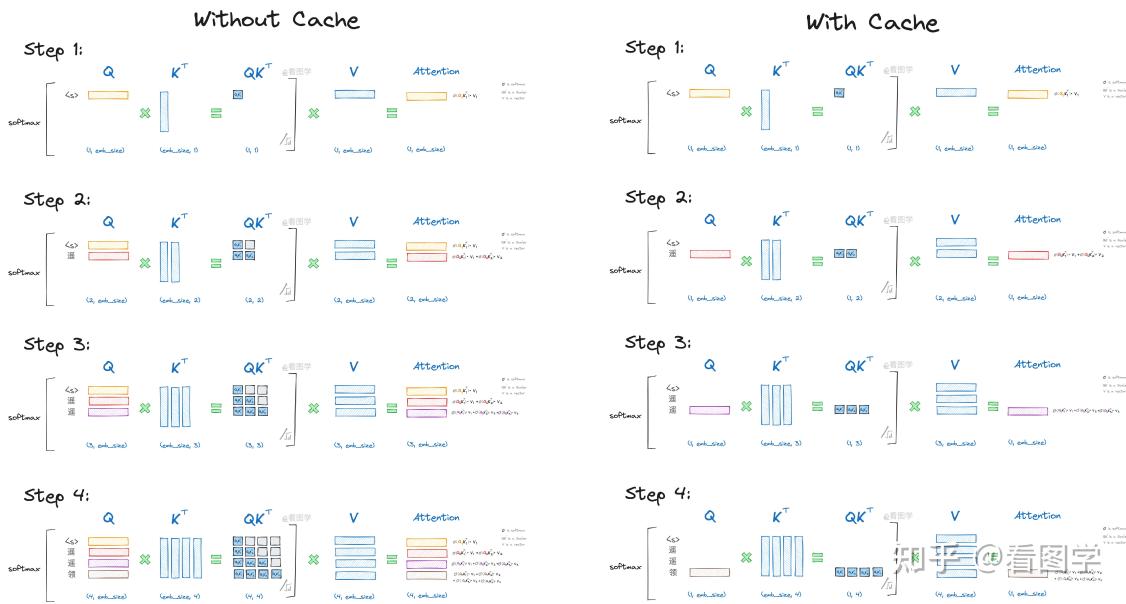
1. 当前计算方式存在大量冗余计算。
2. Att_k 只与 Q_k 有关。
3. 推理第 x_k 个字符的时候只需要输入字符 x_{k-1} 即可。

我们每一步其实之需要根据 Q_k 计算 Att_k 就可以，之前已经计算的 Attention 完全不需要重新计算。但是 K 和 V 是全程参与计算的，所以这里我们需要把每一步的 K, V 缓存起来。所以说叫 KV Cache 好像有点不太对，因为 KV 本来就需要全程计算，可能叫增量 KV 计算会更好理解。

下面 4 张图展示了使用 KV Cache 和不使用的对比。

推理加速KV Cache 示意图

看图学



下面是 gpt 里面 KV Cache 的实现。其实明白了原理后代码实现简单的不得了，就是 concat 操作而已。

https://github.com/huggingface/transformers/blob/main/src/transformers/models/gpt2/modeling_gpt2.py#L318C1-L331C97

```

if layer_past is not None:
    past_key, past_value = layer_past
    key = torch.cat((past_key, key), dim=-2)
    value = torch.cat((past_value, value), dim=-2)

if use_cache is True:
    present = (key, value)
else:
    present = None

if self.reorder_and_upcast_attn:
    attn_output, attn_weights = self._upcast_and_reordered_attn(

```

```
    query, key, value, attention_mask, head_mask)
else:
    attn_output, attn_weights = self._attn(query, key, value,
        attention_mask, head_mask)
```

最后需要注意当 sequence 特别长的时候，KV Cache 其实还是个 Memory 刺客。

比如 batch_size=32, head=32, layer=32, dim_size=4096, seq_length=2048, float32 类型，则需要占用的显存为 $2 * 32 * 4096 * 2048 * 32 * 4 / 1024/1024/1024 = 64G$ 。

针对内存的问题，学者们研发出了 Page Attention 之类的方法来优化。

Chapter 6: MOE, 多模态等

Chapter 7: 大语言模型评估

Chapter 8: 大语言模型应用

Prompt Engineering

Agent

Chapter 1: 大模型发展史

AI 的起源

翻一翻历史其实是蛮有意思的一件事情，我们可以看到很多现在与历史上一些相似的瞬间，仿佛 Yesterday Once More。

AI 的发展到今天差不多也有 70 多年的历史了，让我们来回顾一下这段跌宕起伏的历史。

机器能不能思考？

1900 年的科幻小说《绿野仙踪》里，有个铁皮人，这可能是早期人们对人工智能机器人的一种幻想。



50 年后，图灵认为人类通过信息和推理来解决问题和做出决策，那机器能不能做同样的事情？1950 年，图灵发表了论文《计算机械和智能》，在里面讨论了如何构建智能机器以及如何测试机器的智能，也就是大家都熟知的“图灵实验”，开始把幻想映射到现实。

但是图灵提出设想后并没有做出一台智能机器。原因有很多，但是主要的原因可能有两个。

一个是图灵在研究人工智能的时候，冯诺依曼正在研究计算机体系结构，所以那个时候计算机还没有使用冯诺依曼结构，自然也就缺乏一个智能体的关键结构：记忆。那个时候的计算机只存储数据，并不存储命

令。也就是说你可以告诉计算机要干啥，但是它并没有记住自己干了什么。是不是感觉跟现在的大模型有点像？

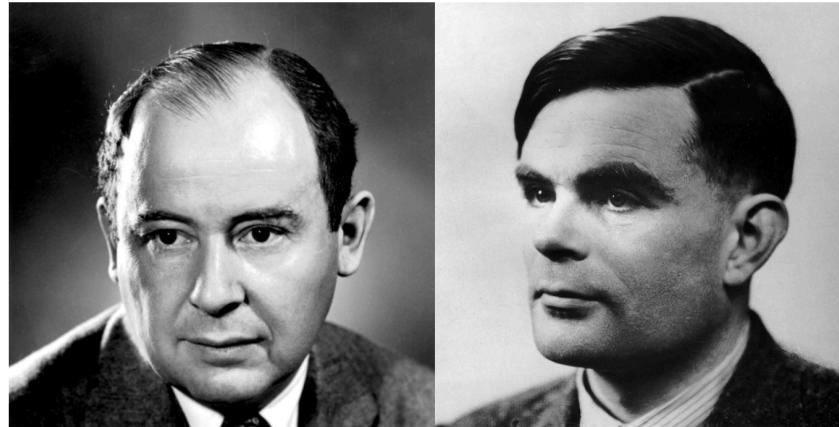
再一个，那个时候计算机是真的贵，当时租用一台计算机要花 20 万美元，只有高科技公司和大学才有机会使用。现在大模型训练成本也高的离谱。

相似瞬间

历史	现在	启发
当时的计算机只存储了数据，没有存储指令，没有记忆	大模型也可以认为是存储了数据，目前也没有记忆	Agent目前是外挂记忆，后续大模型会不会内置一个指令存储单元
成本高，计算机租金每月20万美元	大模型训练成本高，GPT3 最开始训练一次几千万美元	成本在当时是瓶颈，摩尔定律会解决这个瓶颈

这段历史中出现了两个大佬。

冯诺依曼被人们称作“计算机之父”，“博弈论之父”。图灵被人们称作“计算机科学之父”，“人工智能之父”。



冯诺依曼
计算机之父
博弈论之父

艾伦图灵
“计算机科学之父”
“人工智能之父”

冯诺依曼比图灵大 10 岁，这两个都是天才级别的人物。

达特矛斯会议：AI 一大

目前大家普遍认为是 AI 的起源是 1956 年的达特矛斯会议。因为这次会议上，人工智能研究的先行者聚集在了一起，搞了个为期 2 个月的头脑风暴。第一次提出了“人工智能”这个词，虽然后来考证可能是麦卡锡把维纳的 Cybernetics 换了个名字。

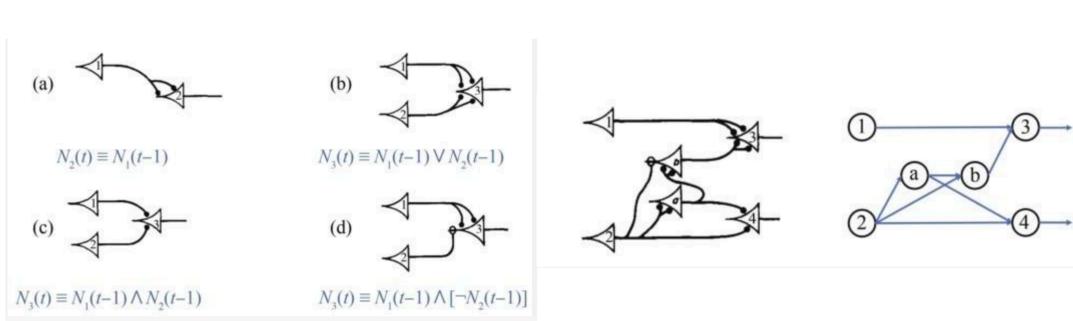
当时参加会议的人，大都是某学科的开山鼻祖，混的差一点的也都是几年后的图灵奖获得者。

这帮人之间也有千丝万缕的关系，这也告诉我们一个道理，你要想成为大佬，先要接近大佬。

我们按时间线来讲一下参加会议的人是怎么凑到一起的。

1936 年，图灵在论文《论可计算数及其在判定问题上的应用》中，提出了图灵机理论。

1943 年，神经生理学家沃伦·麦卡洛克 (Warren McCulloch) 和数学家沃尔特·皮茨 (Walter Pitts) 参考了图灵机里面的一些思想，开发了神经元基本模型，将神经网络的概念引入计算机领域，被认为人工智能第一项工作。³

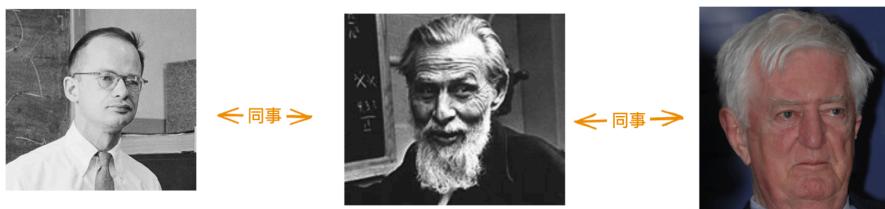


上图为论文中的神经元结构，右侧的图甚至有点像后来的 GRU 和 LSTM。

皮茨是个天才，12 岁时，3 天读完了罗素的《数学原理》，还给罗素写了封信，指出了其中的几处错误。这个罗素就是提出“罗素悖论”的那个，看到信后邀请他去剑桥读他的研究生，根本没想到是个 12 岁的小屁孩，结果这个小屁孩没钱，十动然拒，罗素痛失天才弟子。皮茨由于都是自学成才，高中都没毕业，没学位也让他四处碰壁。但是他实在过于优秀，很多大佬给他站台。曾跟随芝加哥大学的逻辑学家卡尔纳普 (Rudolf Carnap) 学习，卡尔纳普无论是在学习还是生活都给了皮茨很大的帮助。后来皮茨被沃伦·麦卡洛克 (Warren McCulloch) 推荐给控制论之父诺伯特·维纳 (Norbert Wiener)，维纳发现了皮茨的潜力，直接收了他作为博士生，要注意皮茨连高中和本科学位都没有啊。沃伦·麦卡洛克 (Warren McCulloch) 本来在芝加哥大学好好的当教授，可能是比较崇拜维纳，辞去教授去 MIT 当了个研究员，就是为了和维纳一起工作。

1945 年以后，奥利弗·塞尔弗里奇 (Oliver Selfridge) 在 MIT 读研究生，他的导师就是上面提到的诺伯特·维纳 (Norbert Wiener)。这个时候他与沃尔特·皮茨 (Walter Pitts) 和沃伦·麦卡洛克 (Warren McCulloch) 在一起工作，研究人类智能。⁴

q



沃尔特·皮茨
(Walter Pitts)
美国逻辑学家和计算神经科学家

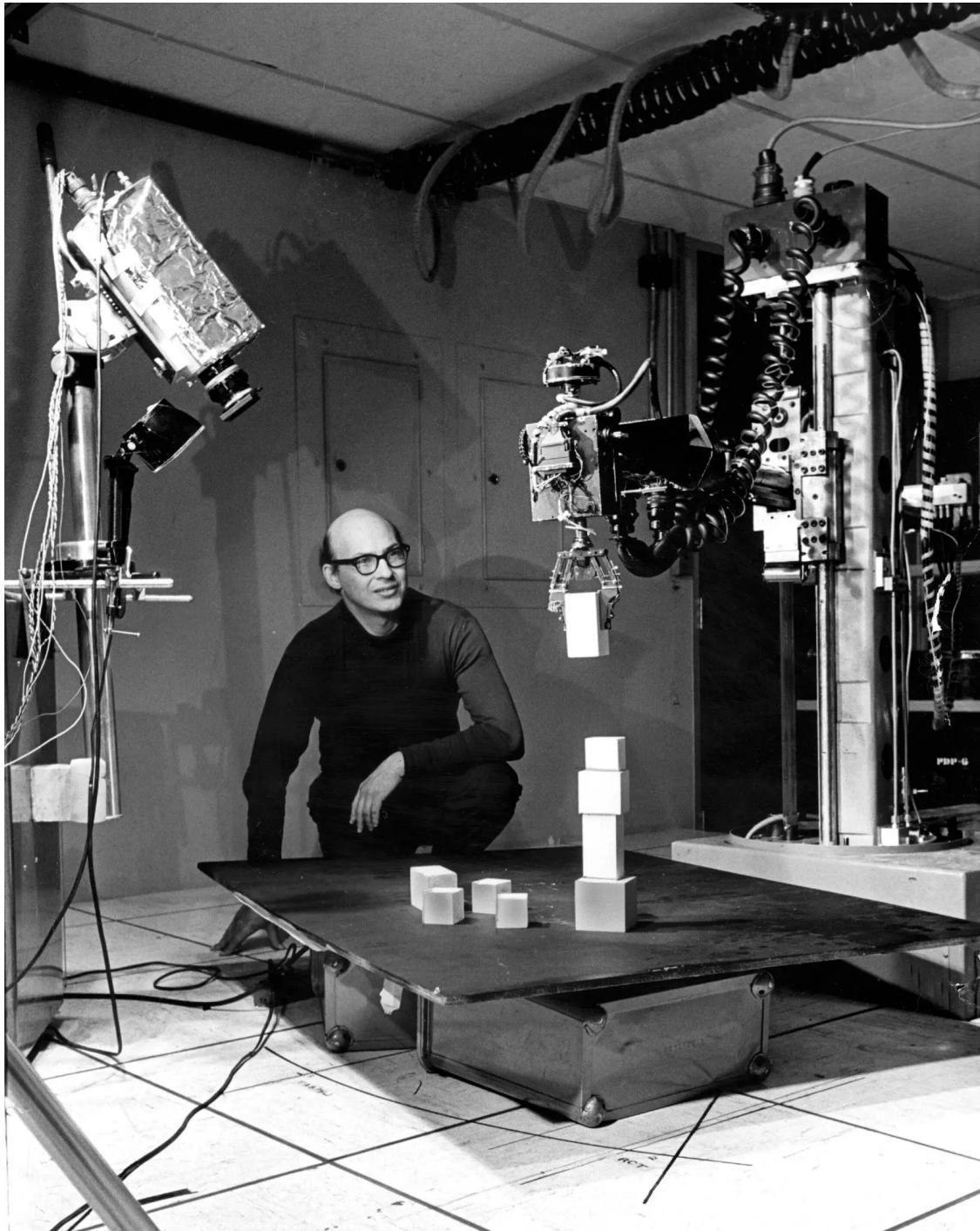
沃伦·麦卡洛克
(Warren McCulloch)
美国神经科学家和控制论学者

奥利弗·塞尔弗里奇
(Oliver Selfridge)
模式识别的奠基人，被称为“机器知觉之父”

³<https://www.cs.cmu.edu/~epxing/Class/10715/reading/McCulloch.and.Pitts.pdf>

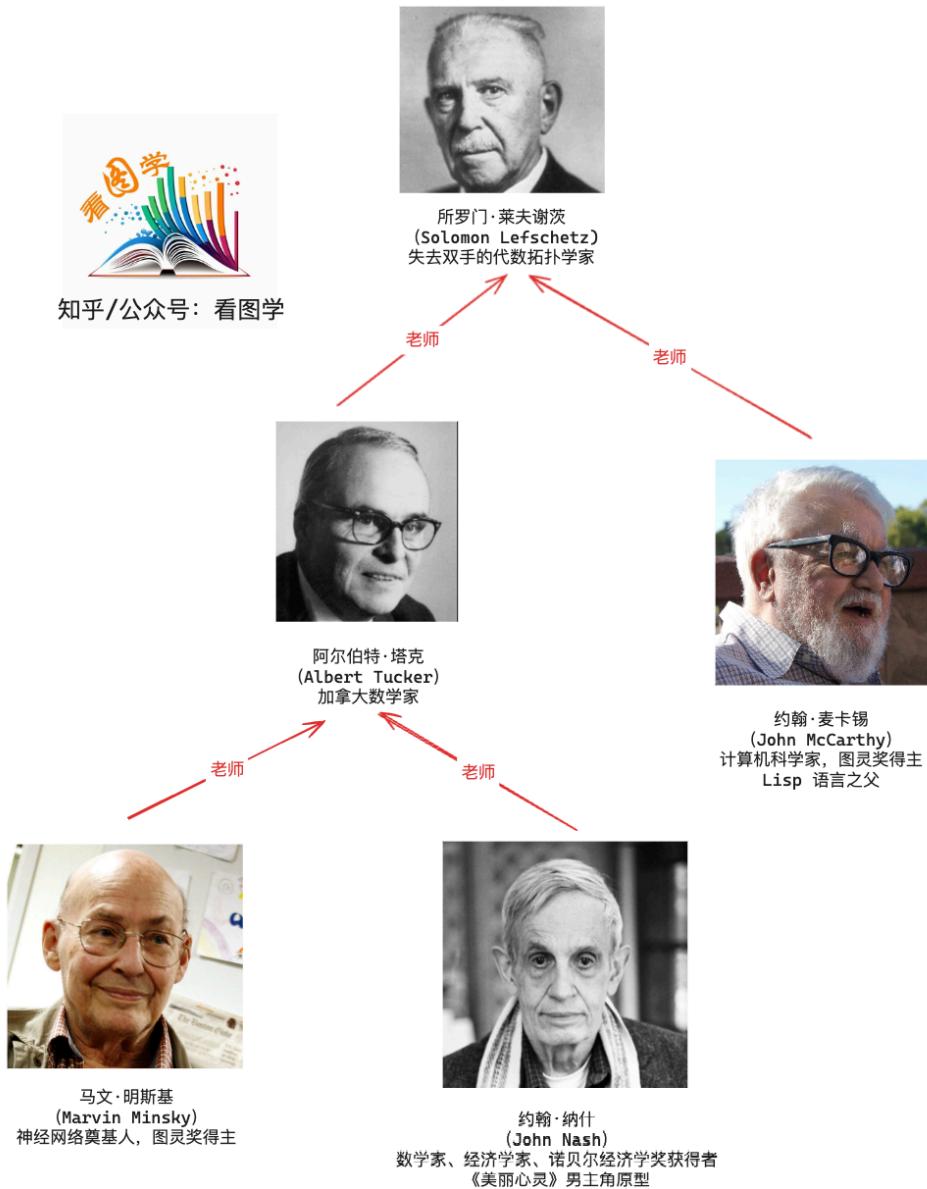
⁴<https://vineyardgazette.com/obituaries/2009/01/29/oliver-selfridge-computer-pioneer-loved-chappy>

1951 年，塞尔弗里奇加入了麻省理工学院的林肯实验室，很快成为一个研究通信技术、模式识别、指挥和控制以及交互式计算的团队的组长。折腾一些机器识别摩尔斯电码的早期工作。塞尔弗里奇可以认为模式识别的奠基人，他写了第一个可工作的 AI 程序，后来被人们称为“机器知觉之父”。那一年的夏天，他招了伙计，这个人叫马文·明斯基（Marvin Minsky）。



马文·明斯基（Marvin Minsky）呢，在这一年搞出了第一部能自我学习的人工神经网络机器，SNARC。他奠定了人工神经网络的研究基础，然而造化弄人，在不久的将来他却亲自给人工神经网络判了死刑，导致很长一段时间神经网络变成了冷板凳。

马文·明斯基的导师是塔克 (Tucker)，塔克还有个学生就是电影《美丽心灵》的主角约翰纳什 (John Nash)。所以纳什是明斯基的师兄。塔克的导师是失去双手的代数拓扑学家所罗门·莱夫谢茨 (Lefschetz)。莱夫谢茨还有个学生，叫约翰·麦卡锡 (John McCarthy)。从师承上讲约翰·麦卡锡是马文·明斯基和纳什的师叔，算是老相识。



马文·明斯基的博士论文是神经网络，但是他当时是数学系的学生，博士委员会看了就有点迷糊，神经网络算是数学么？当时冯诺依曼也是博士委员会的，说虽然现在可能不算，但是不久之后就是数学了。所以马文·明斯基的博士答辩通过，冯诺依曼是大恩人。

约翰·麦卡锡是个共产主义战士。他是达特茅斯会议的发起人。他是怎么到达特茅斯学院的呢？当时达特茅斯数学系的系主任是克门尼 (Kemeny)。克门尼是图灵的师弟，都师从普林斯顿逻辑学家丘奇 (Church)，

他战时和物理学家费曼一起工作，还一度当过爱因斯坦的数学助理，就问牛不牛逼。当时达特茅斯数学系一下子退休了 4 个教授，克门尼压力山大，人都走光了，工作还怎么展开。就去普林斯顿找了 4 个博士，其中一个就是麦卡锡。克门尼和麦卡锡合作还是很愉快的，一起琢磨出了计算机的分时系统，克门尼创造了 Basic 编程语言，而麦卡锡则创造了 Lisp。Lisp 现在可能很多人没听说，但是 Lisp 的设计理念可以说是影响了后续所有的编程语言，我现在因为使用 Emacs 还偶尔用一下。

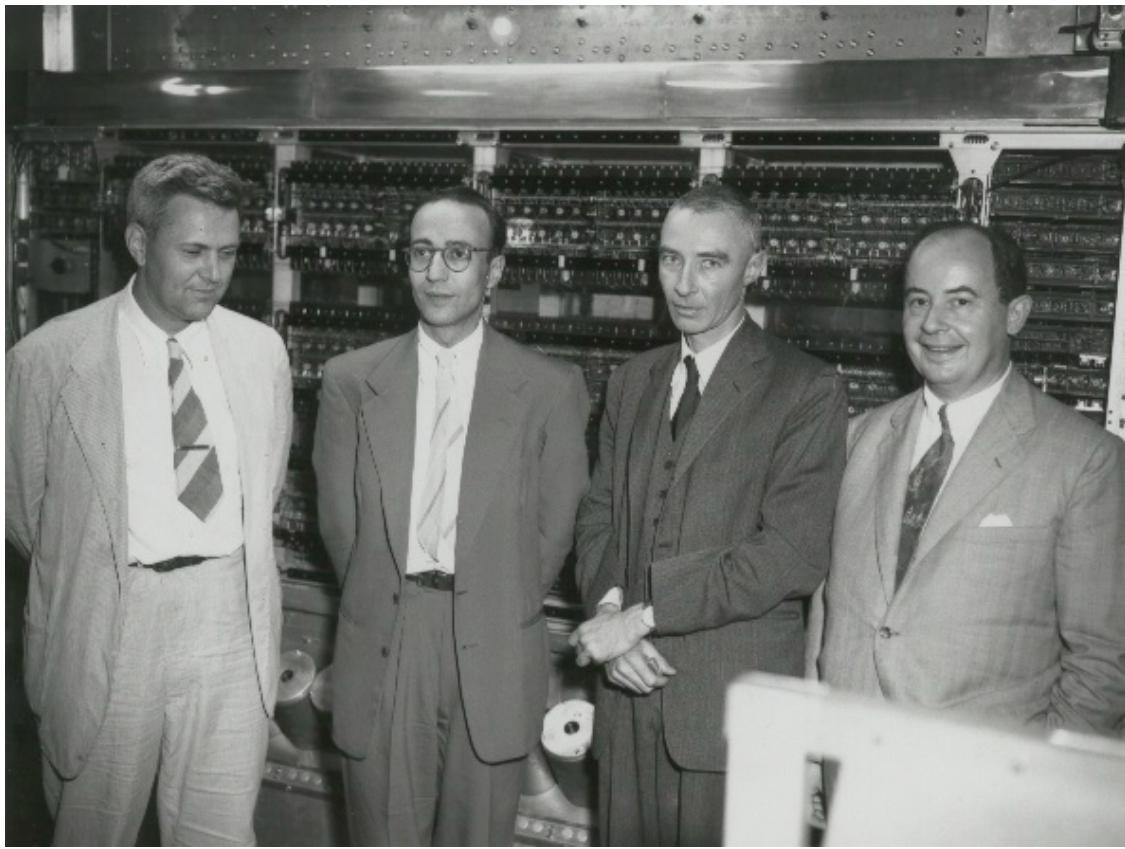
1953 年夏天，麦卡锡和明斯基都在贝尔实验室为克劳德·香农（Claude Shannon）打工。香农是信息论的创始人，和图灵，冯诺依曼是同一个级别的，就不多介绍了。这里有个段子就是香农作为学生的时候，天天问诺伯特·维纳（Norbert Wiener）问题，借鉴他的思想。导致后来维纳拒绝跟香农见面，说：“香农就是来挖我脑浆子的”。后来被认为是创立了信息论的香农的硕士论文《通信的数学原理》中，香农自己都说：“Credit should also be given to Professor N. Wiener”，也就是荣誉也当属于维纳教授。⁵

Acknowledgments

The writer is indebted to his colleagues at the Laboratories, particularly to Dr. H. W. Bode, Dr. J. R. Pierce, Dr. B. McMillan, and Dr. B. M. Oliver for many helpful suggestions and criticisms during the course of this work. Credit should also be given to Professor N. Wiener, whose elegant solution of the problems of filtering and prediction of stationary ensembles has considerably influenced the writer's thinking in this field.

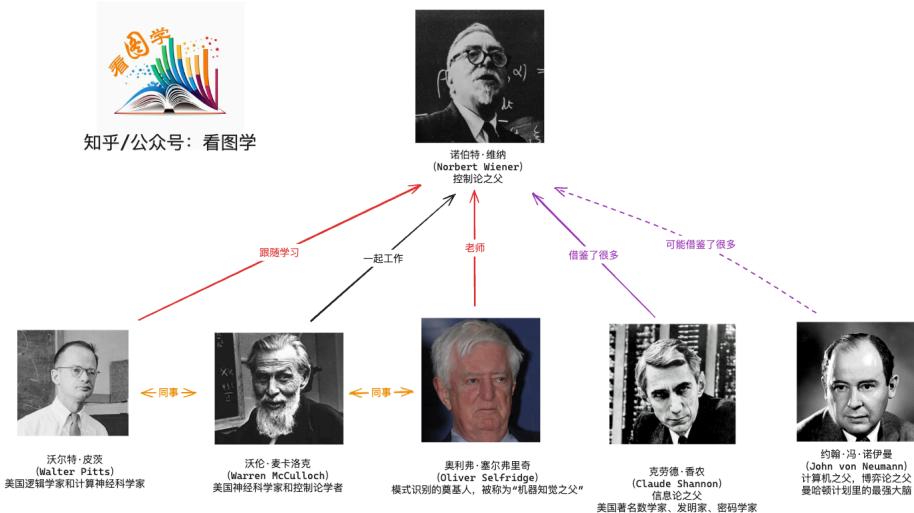
在一本书《维纳传：信息时代的隐秘英雄》曾经提到，冯诺依曼也经常参加维纳的控制论相关的会议，而且是“毫无保留地或者几乎毫无保留地把自己对计算机和自动机器的想法告诉了冯·诺依曼”，而且还派了朱利安·毕格罗（Julian Bigelow）给冯诺依曼当助手。所以有人认为冯诺依曼架构应该叫做维纳-冯诺依曼架构。维纳对后世的贡献很有可能被低估了。

⁵https://pure.mpg.de/rest/items/item_2383164/component/file_2383163/content



从左到右依次是朱利安·毕格罗、赫尔曼·戈德斯汀、罗伯特·奥本海默(原子弹之父)和约翰·冯·诺依曼

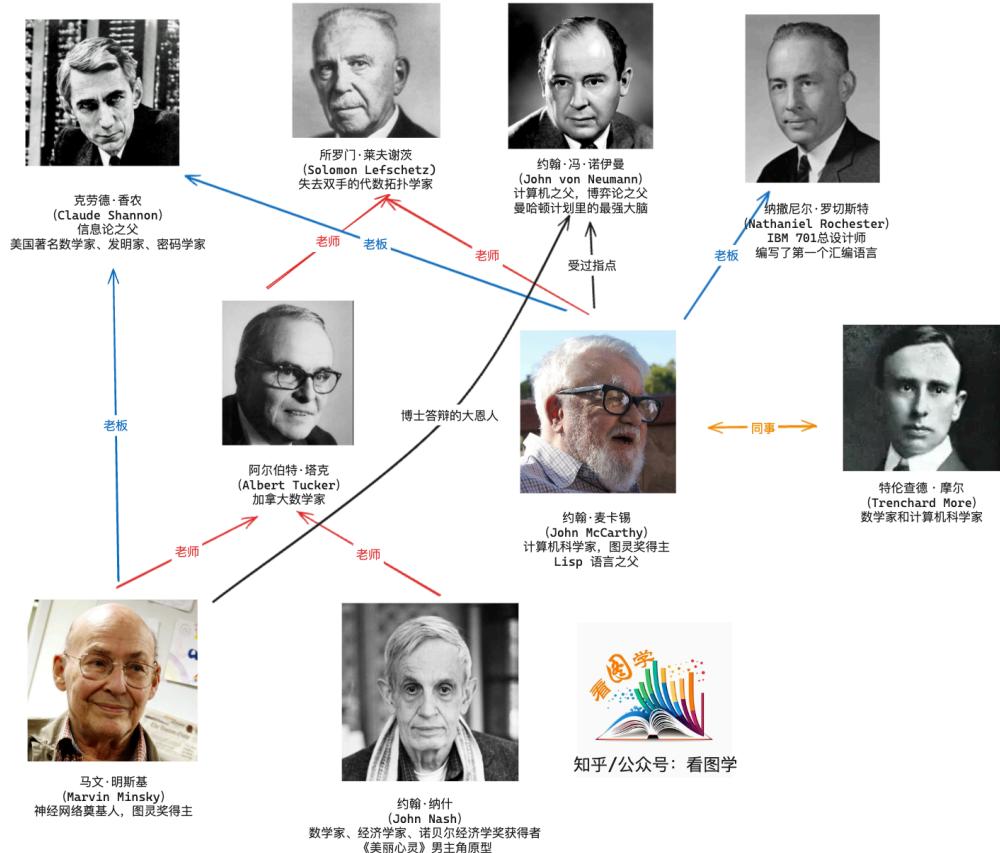
维纳的影响力在当时是巨大的。



1954 年，一个叫艾伦·纽厄尔 (Allen Newell) 的年轻人在一次研讨会上听到奥利弗·塞尔弗里奇 (Oliver Selfridge) 描述“运行一个计算机程序，学会了识别字母和其他模式”，也开始做人工智能相关的工作，但是他与他的博导司马贺 (Herbert Simon) 后来开辟了一条全新的道路。

1955 年夏天，麦卡锡又跑去 IBM 打工了。为啥麦卡锡老在夏天跑出去打工？因为那个时候美国的教授只发 9 个月工资，你要是没有科研经费，就得趁着暑假出去打工，年轻教授真是不容易。打工的老板是纳撒尼尔·罗切斯特 (Nathaniel Rochester)，这个老板对神经网络很感兴趣。麦卡锡一看，机会来了，有现老板和前老板站台，再加上前同事明斯基，4 个人决定搞一次人工智能的会议。

麦卡锡之所以拉上另外 3 个人，是因为要申请会议经费，另外三个名气当时都很大。很明显，麦卡锡是达特茅斯会议的 KOL。



1955 年还发生了一件事，那就是在美国西部计算机联合大会上，前面提到的奥利弗·塞尔弗里奇 (Oliver Selfridge) 发表了一篇模式识别的文章，他之前都跟神经网络的大牛们混在一起，所以自然是研究神经网络。但是当时研究智能还有另外一派，那就是模拟人的心智。艾伦·纽厄尔 (Allen Newell) 作为这一方向的研究者在会上探讨了计算机下棋。

讨论会的主持人是前面讲对最早提出人工神经元对沃尔特·皮茨 (Walter Pitts)，他最后总结时说：“（一派人）企图模拟神经系统，而纽厄尔则企图模拟心智 (mind) ……但殊途同归。”

但是从后面的 AI 的发展来看，这两派丝毫没有同归的意思，反而想要同归于尽。

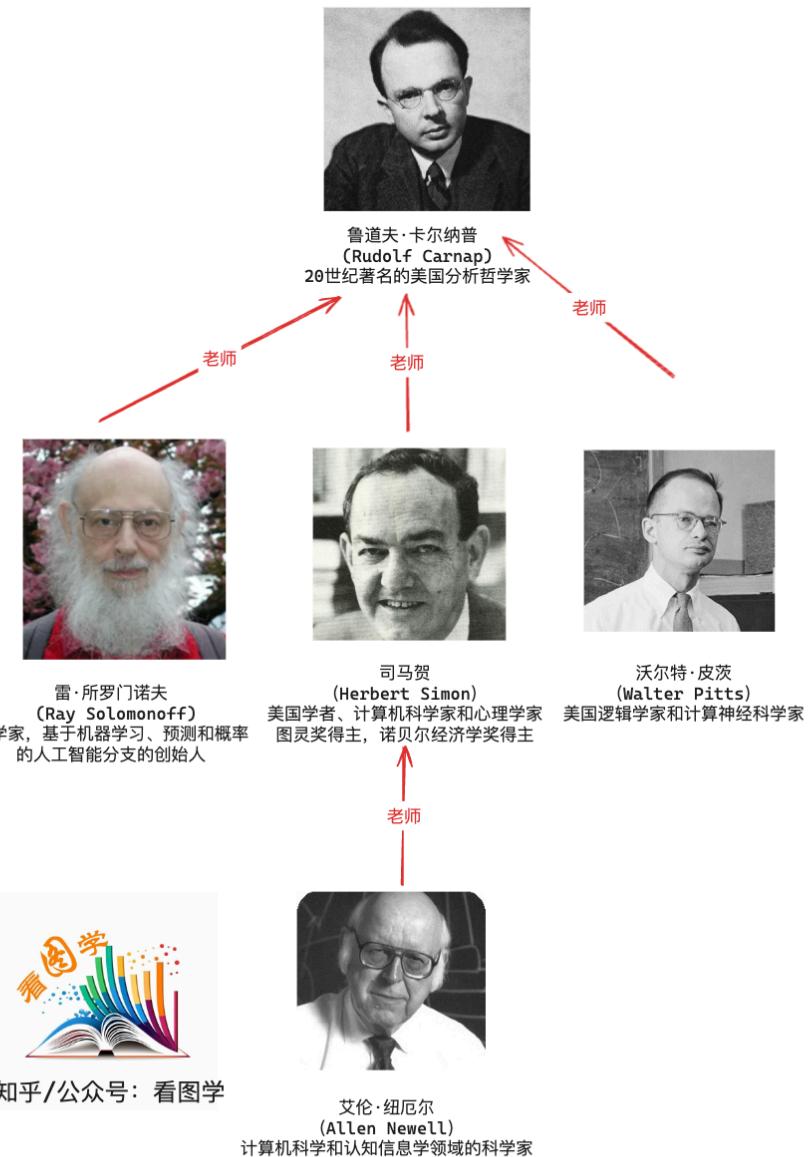
艾伦·纽厄尔 (Allen Newell) 的博导是司马贺 (Herbert Simon)，这两个人是后来人工智能符号派的代

表。这两个人后来和第一届图灵奖获得者阿兰·珀里思（Alan Perlis）一起创办了卡内基梅隆大学的计算机系，硬是把一个三本提升到了世界一流学校。



司马贺（Herbert Simon）对中国大陆学术界有较深影响。“乒乓外交”打破了中美坚冰后的 1972 年 7 月，赫伯特·西蒙作为美国计算机科学代表团成员首次访问中国，后多次访华交流讲学及合作研究。其中文名字司马贺，即是他 1980 年作为美国心理学代表团成员第二次访华时所起，其本人 70 多岁的年龄开始学习汉语。1994 年当选为中国科学院外籍院士。

司马贺在芝加哥大学求学时，也曾经跟皮茨的老师卡尔纳普（Rudolf Carnap）学习，受他的影响开启了研究人工智能之路。所以虽然后面的人工智能分成了连接派和符号派，但是他们的启蒙老师都是卡尔纳普（Rudolf Carnap）。



终于，在 1956 年，麦卡锡筹备的会议开始了，这次会议名就叫：达特茅斯夏季人工智能研究计划（Dartmouth Summer Research Project on Artificial Intelligence）。

下面这张照片，展示了本次参会的主要 7 个人。



Nathaniel Rochester Marvin L. Minsky John McCarthy
Oliver G. Selfridge Ray Solomonoff Trenchard More Claude E. Shannon

August 1956

当时还有很多人也参加了。比如同样受到卡尔纳普 (Rudolf Carnap) 影响的所罗门诺夫 (Solomonoff)，还有两个是来自 IBM 的撒缪尔 (Arthur Samuel)，达特茅斯的教授摩尔 (Trenchard More)。

所以后来有些文献说主要有 10 个人参加。这 10 个人有时候被称为 AI 之父。

a medal". All of them are heroes. But did this workshop leave out some

1956 Dartmouth Conference: The Founding Fathers of AI



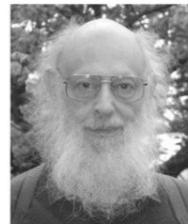
John McCarthy



Marvin Minsky



Claude Shannon



Ray Solomonoff



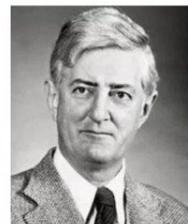
Alan Newell



Herbert Simon



Arthur Samuel



Oliver Selfridge



Nathaniel Rochester



Trenchard More

after him?

麦卡锡后来把参会名单弄丢了，好在所罗门诺夫（Solomonoff）全程参与了两个月的会议，他有个好习惯是记笔记。他的笔记中，记录了 20 个人参加了这次会议，比如纳什。

T.M.

from 196

People at Summer research project.

Solomonoff

Maurice Wilsky MIT Lincoln

John McCarthy IBM, Dartmouth

Claude Shannon MIT, Bell

French More IBM, MIT

Mat Rochester IBM Poughkeepsie

Oliver Selfridge MIT Lincoln

Julian Bigelow IAS

W. Ross Ashby Barnwood house (?)

W.S. McCulloch, MIT, RLE

Abraham Robinson Montreal logic

Tom Etter

John Nash MIT

David Sayre IBM New York

Samuels (IBM) non checkers

Shoulders MIT RLE or Lincoln components man

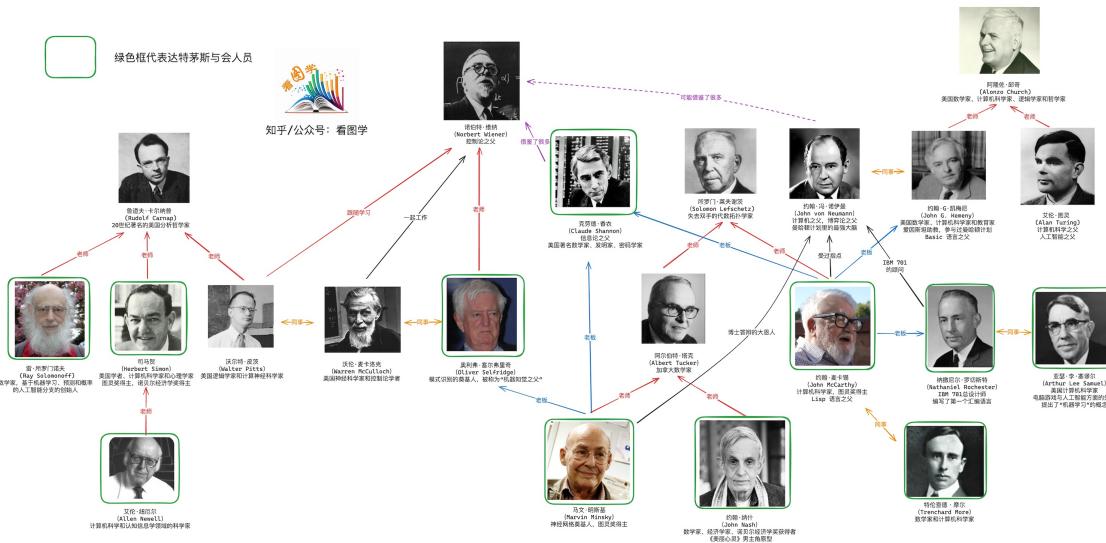
... (with Shoulders)

Alex Bernstein IBM (New York) chess

Herbert Simon: U of Pa (?)

Allen Newell: Rand

上面提到的人建立个关系图谱，大概是下面这个样子。



AI 一大都研究了什么？

会议的主要议题如下：

1. 自动计算机

如果一台机器可以做一项工作，则一台可编程自动计算机能用来模拟这台机器。现有计算机的速度和内存容量可能不足以模拟人脑的许多高级功能，但主要障碍不是缺乏机器容量，而是我们无法编写充分利用我们所拥有优势的程序。

作者注：这个目标似乎已经实现，目前图像识别，语音识别等都已经做的不错了。

2. 如何使用语言对计算机进行编程

可以推测，人类思想的很大一部分包括根据推理规则和猜想规则来操控单词。从这个角度来看，形成包括承认一个新词和一些规则的概括，其中含有它的句子暗示并被其他人暗示。这个想法从未如此精确地制定，也没有制定出实例。

作者注：站在现在的角度看，这就是 NLP 发展的一个基本思想，如何表示一个单词？大佬们 70 年前就已经尝试从 NLP 这个领域来突破 AI 了。

3. 神经网络

如何安排一组（假设的）神经元以形成概念。乌特利·拉什夫斯基 (Uttley, Rashevsky) 和他的团队，法利 (Farley) 和克拉克 (Clark)，皮茨 (Pitts) 和麦卡洛克 (McCulloch)，明斯基 (Minsky)，罗切斯特 (Rochester) 和霍兰德 (Holland) 等人在这个问题上做了大量的理论和实验工作。已经获得了部分结果，但问题是需要更多的理论工作。

作者注：最最早期的神经网络的探索。

4. 计算大小的理论

如果给出一个定义明确的问题（可以用机械方式测试提出的答案是否是有效答案），解决问题的方法是按顺序尝试所有可能的答案。这种方法效率低，要排除它，必须有一些计算效率的标准。一些考虑将表明，为了测量计算的效率，有必要手头有一种测量计算装置复杂性的方法，如果有一个具有功能复杂性的理论，则可以这样做。香农 (Shannon) 和麦卡锡 (McCarthy) 也获得了关于这个问题的部分结果。

作者注：这个属于计算机科学中的算法理论的范畴了。

5. 自我改进

可能真正智能的机器将开展可以最好地描述为自我改进的活动。已经提出了一些这样做的方案，值得进一步研究。这个问题似乎也可以抽象地进行研究。

作者注：已经是机器学习的基本套路了。

6. 抽象

许多类型的“抽象”可以明确定义，而其他几个则不那么明显。直接尝试对这些进行分类并描述从感官数据和其他数据形成抽象的机器方法似乎是值得的。

作者注：如何数据中归纳出知识，也可以归到机器学习。

7. 随机性和创造力

一个相当有吸引力但又不完全不完整的猜想是，创造性思维和缺乏想象力的能力思维之间的区别在于注入一些随机性。随机性必须由直觉引导才能有效。换句话说，受过教育的猜测或预感包括在其他有序思维中的受控随机性。

作者注：这个就有点 AGI 的味道了。当时是通过加入随机性来提高创造性，现在可以通过 Temperature 来调整。那个时候的大牛们是真的没想到后来 GPT “涌现”出来的智能。

AI 一大的影响

从上面可以看出，大佬们在会上还是讨论了很多东西，而且很多东西站在现在来看也并没有过时。这次会议上关于 AI 的研究方向已经隐隐划分为两派：

1. 符号派。他们开创性的认为：知识可以由一组规则来表示，而计算机程序可以使用逻辑来操控这些知识。这一派参会的主要代表人物是：艾伦·纽厄尔 (Allen Newell)、司马贺 (Herbert Simon)，约翰·麦卡锡 (John McCarthy)，马文·明斯基 (Marvin Minsky)。明斯基早期应该是连接派的，因为他的博士论文就是研究神经网络，后来给神经网络死刑，然后叛变成符号派了。在这个会上，符号派的代表纽厄尔和司马贺公布了一款程序“逻辑理论家”(Logic Theorist)，这个程序可以证明怀特海和罗素《数学原理》中命题逻辑部分的一个很大子集。是整个会议最令人印象深刻的。此时此刻符号派稳压连接派一头。
2. 连接派。连接派就是通过一种刻画人类大脑中神经元之间相互连接的机制，来模拟人类行为。当时在会上只是讨论，当时还没有特别亮眼的表现，提到了皮茨 (Pitts) 和麦卡洛克 (McCulloch)。但是会后没多久，神经网络也有了突破。

有意思的是，鲁道夫·卡尔纳普 (Rudolf Carnap) 的两个学生，司马贺与皮兹，分别成了符号派和连接派最早期的代表，还有个学生雷·所罗门诺夫 (Ray Solomonoff) 则在机器学习领域有开创性的贡献，从这个角度来看，AI 的起源，鲁道夫·卡尔纳普是有巨大贡献的。

还有一个流派也隐藏在上面的图中，那就是起源于维纳的控制论，也就是后来的行为主义。采用“感知-动作”来研究，通过环境反馈和智能行为之间的因果去探寻智能，后来就发展除了强化学习和现在正火的 Agent 。

维纳对 AI 的贡献甚至比想象中的还要大，以至于 Michael Jordan (2018) 曾讽刺的说到：“维纳提出的方法却披着麦卡锡发明的术语的外衣”。维纳提出的 Cybernetics，被翻译成了控制论，但实际上“机械大脑论”可能更符合这个词的本意。只不过当时维纳突然与自己的学生决裂，导致年轻学生们都想“离维纳的控制论越远越好”。麦卡锡当时就是这么想的，他曾在“控制论”和“自动机”之间徘徊不定，最后选择了“人工智能”。而且当时计算机的三大“之父”，除了图灵以外，香农、冯诺依曼都深深的受到维纳的影响。图灵是真的天才，自己独立搞了一套。所以维纳可能是活成了 AI 的里子。

AI 的故事到现在才刚刚开始，后续几年，符号派和连接派的内战就要开始了，我们下次再讲。

符号派、链接派，相爱又相杀

从 one-hot 到 ChatGPT