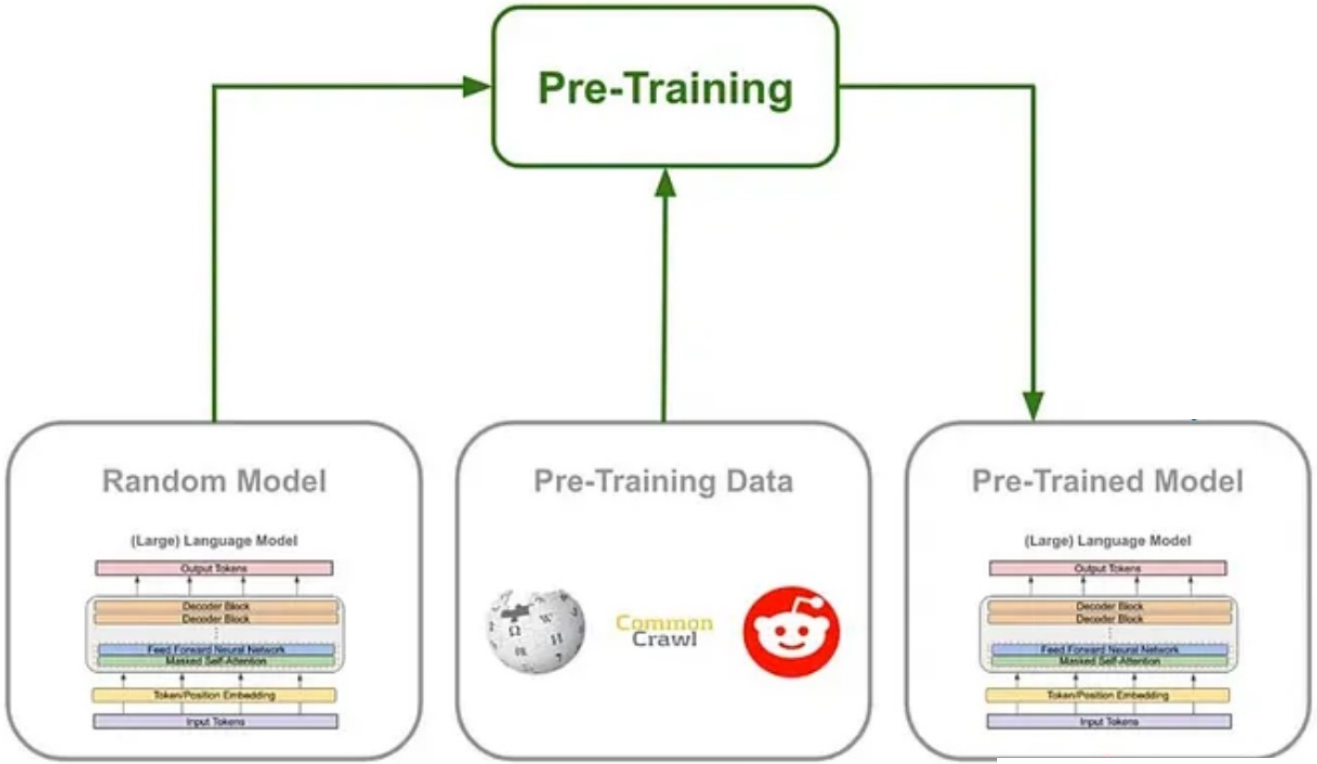
第五章 预训练语言模型

目录

5.1 概述
5.2 Seq2Seq模型
5.3 注意力机制
5.4 Transformer 模型
5.5 预训练语言模型
5.6 语言模型使用范式

5.1 概述



预训练语言模型（英文： Pre-trained Language Models ）通过在大规模未标记文本数据上进行自监督学习，来预先训练通用的语言表示。它不再局限于简单的生成任务，而是通过各种自监督任务（如**掩码语言建模**和**下一句预测**）学习到更丰富和普适的语言理解能力。

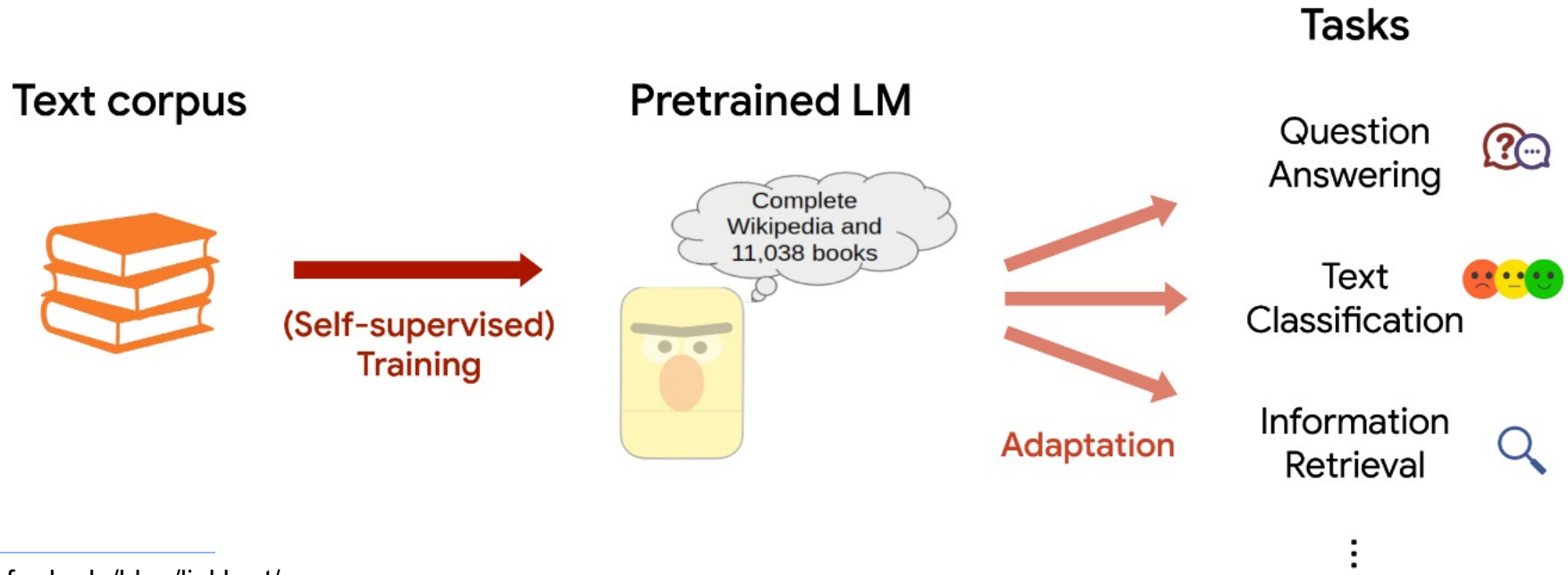
图片来源：https://medium.com/@bijit211987/the-evolution-of-language-models-pre-training-fine-tuning-and-in-context-learning-b63d4c161e49

5.1 概述

数据稀缺问题：在现实世界中，收集并标注大量数据既耗时又昂贵，特别是在某些专业领域，标记数据的获取更为困难。

预训练思想

将大量的无标签文本数据输入到神经网络中进行训练，让模型学习到语言的规律和语义信息，生成一个具有一定语言理解能力的模型。



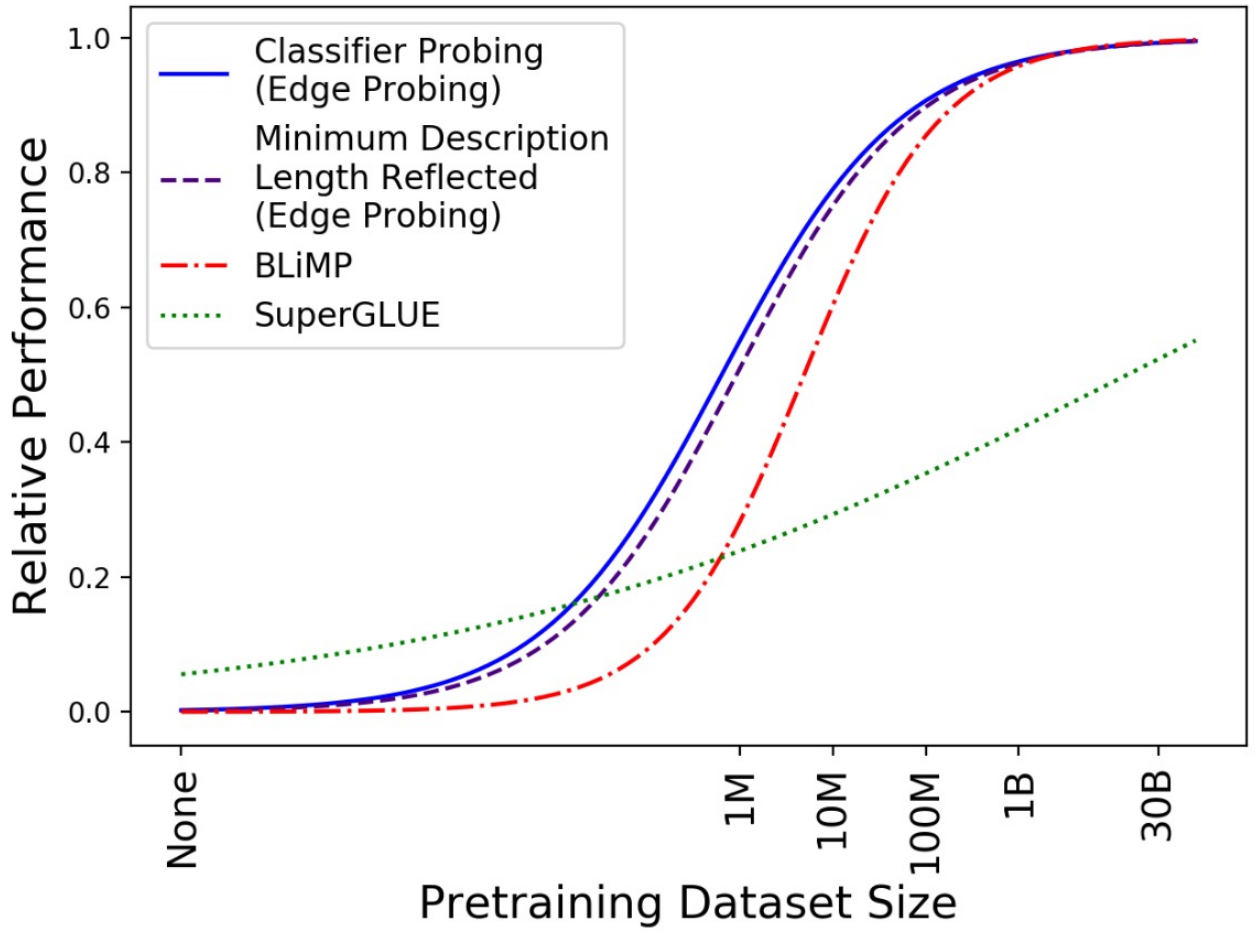
图片来源：http://ai.stanford.edu/blog/linkbert/

需要多少语料库才能学会”文字接龙”？

1. 语言知识

“**我马上就**”后面可以接“**写**”等动词，而不能接“**猫**”等名词

When Do You Need Billions of Words of Pretraining Data?



图片来源：https://arxiv.org/pdf/2011.04946

需要多少语料库才能学会”文字接龙”？

2. 世界知识

“**中国的首都是**”
后面接“**北京**”，
而不能接“**上海**”

模型	年份	数据规模
GPT-1	2018	7000本书
GPT-2	2019	40GB文本
GPT-3	2020	从45TB数据清洗得到570GB文本
Llama 3	2024	15TB文本
Qwen 2.5	2024	大概60TB数据

In terms of Qwen2.5, the language models, all models are pretrained on our latest large-scale dataset, encompassing up to **18 trillion tokens**.

来源：https://qwenlm.github.io/blog/qwen2.5/

目录

5.2 Seq2Seq模型

5.2.1 模型结构

5.2.2 模型训练与使用技巧

5.3 注意力机制

5.3.1 定义与原理

5.3.2 引入注意力机制的编码器-解码器模型

5.3.3 查询、键和值

Sequence to Sequence Learning with Neural Networks

Ilya Sutskever
Google
ilyasu@google.com

Oriol Vinyals
Google
vinyals@google.com

Quoc V. Le
Google
qvl@google.com

Abstract

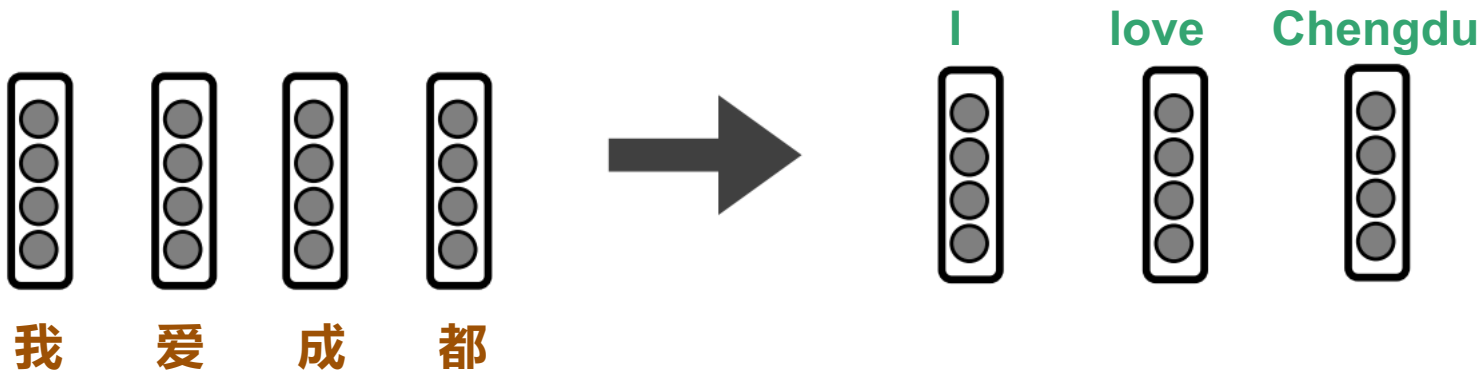
Deep Neural Networks (DNNs) are powerful models that have achieved excellent performance on difficult learning tasks. Although DNNs work well whenever large labeled training sets are available, they cannot be used to map sequences to sequences. In this paper, we present a general end-to-end approach to sequence learning that makes minimal assumptions on the sequence structure. Our method uses a multilayered Long Short-Term Memory (LSTM) to map the input sequence to a vector of a fixed dimensionality, and then another deep LSTM to decode the target sequence from the vector. Our main result is that on an English to French translation task from the WMT'14 dataset, the translations produced by the LSTM

来源：https://arxiv.org/pdf/1409.3215

5.2 Seq2Seq模型

RNN模型：解决序列建模问题[1]。

存在的限制：受限于模型结构，输入和输出的长度固定，不适用处理输入和输出长度不固定的任务。



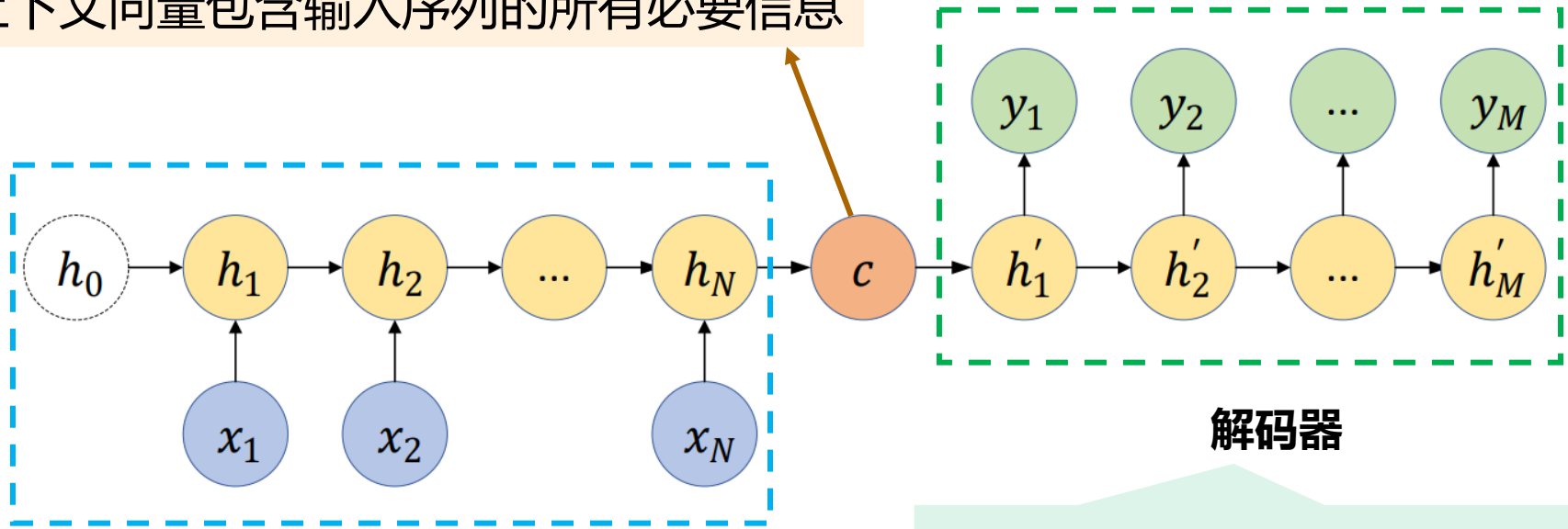
序列到序列模型（Seq2Seq）是一种常用于序列数据处理的深度学习模型，特别适合处理输入和输出长度不固定的任务。虽然它基于RNN构建，但能够灵活地处理长度为 N 的输入序列与长度为 M 的输出序列，因此可以视为N对M循环神经网络。

[1]序列建模问题是指输入和/或输出是数据序列（单词、字母等）的问题。

5.2.1 模型结构

Seq2Seq 模型由**编码器**和**解码器**两个主要部分组成。

上下文向量包含输入序列的所有必要信息



编码器

读取输入序列（比如一个句子），并将其转换为**一个固定长度的上下文向量**

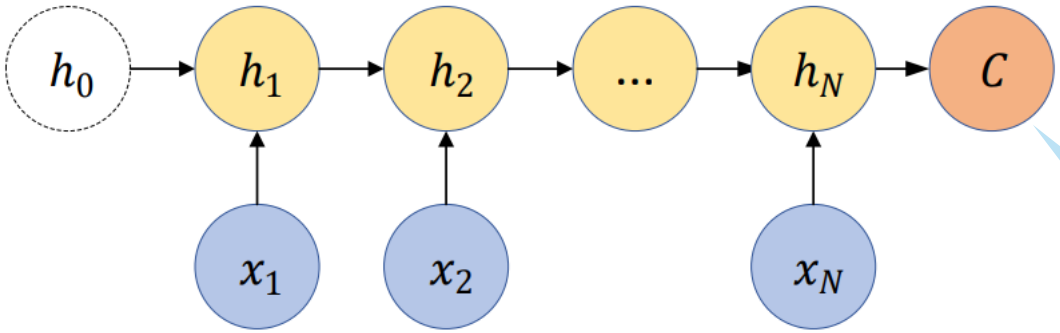
解码器

接收上下文向量，并利用它来生成输出序列

5.2.1 模型结构

1. 编码器计算

$$h_t = g(\boldsymbol{U}h_{\{t-1\}} + \boldsymbol{W}x_t)$$



上下文向量的选择：

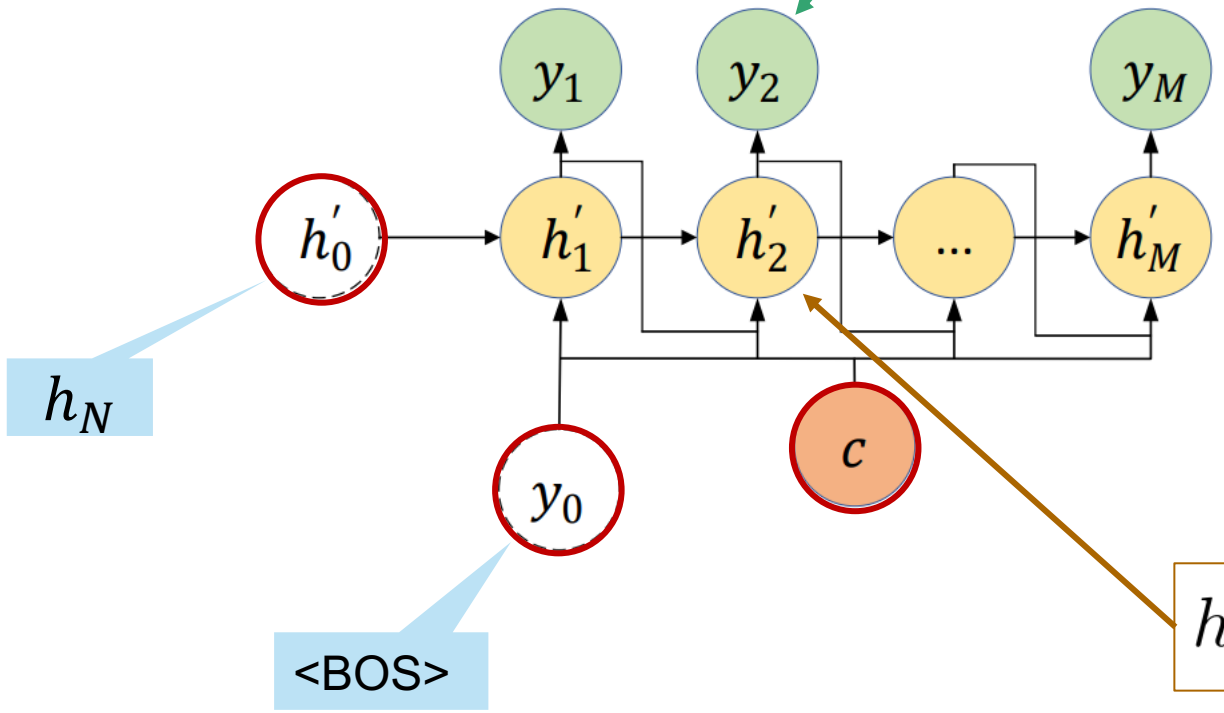
1. $c = h_N$
2. $c = q(h_N)$
3. $c = q(h_1, h_2, \ldots, h_N)$

1. 接收长度为 N 的 输入序列 $x_1, x_2, \ldots, x_N$。
2. 通过 RNN 生成一系列隐藏状态 $h_1, h_2, \ldots, h_N$ 。
3. 最后一个隐藏状态 h_N 被视为上下文向量 c。

5.2.1 模型结构

2. 解码器计算

$$y_t = g(h'_t)$$

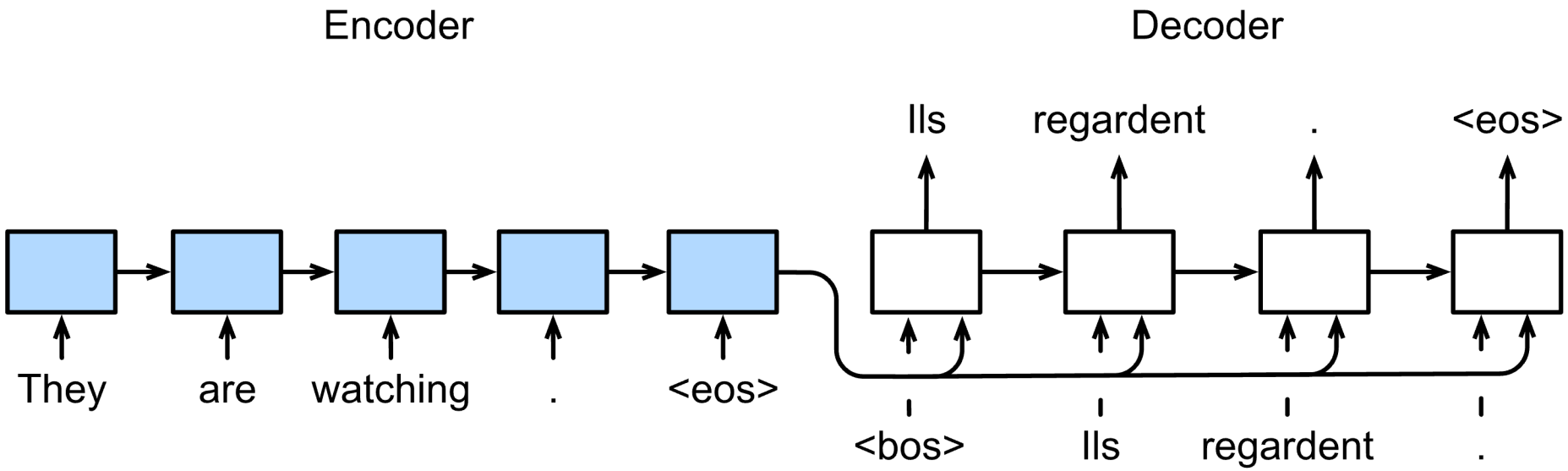


$$h'_t = f(c, h'_{t-1}, y_{t-1})$$

① 上下文向量
② 上一个神经元的隐藏状态
③ 上一个神经元的输出

5.2.1 模型结构

举例



图片来源：https://d2l.ai/chapter_recurrent-modern/seq2seq.html

5.2.2 模型训练与使用技巧

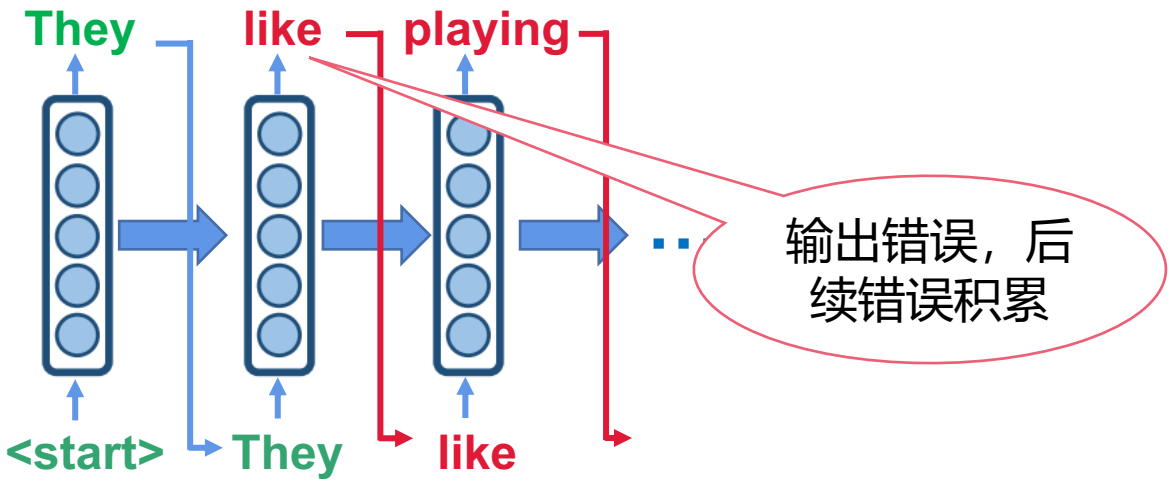
Free-running

Free-running是RNN等模型的常见训练方式，将上一个时间步的输出作为下一个时间步的输入。该训练方式理论上允许模型学习到序列内部的依赖关系。

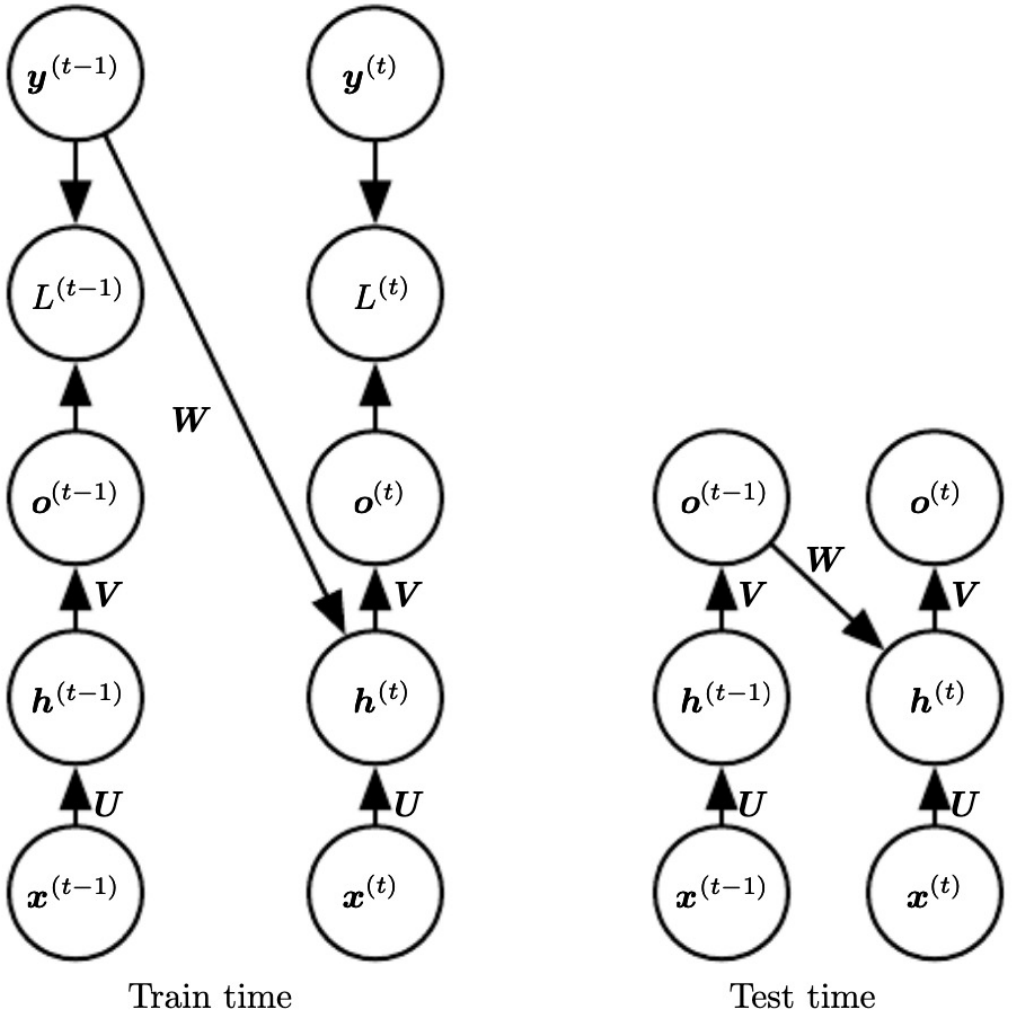
“错误累积”问题

在训练初期，模型由于参数尚未充分优化， 可能会生成错误的输出。这些错误的输出一旦作为后续时间步的输入，就可能引发“错误累积”问题，导致模型难以正确学习序列的长距离依赖关系。

考虑真实输出序列：They moved to Chengdu last month



5.2.2 模型训练与使用技巧



Teacher Forcing

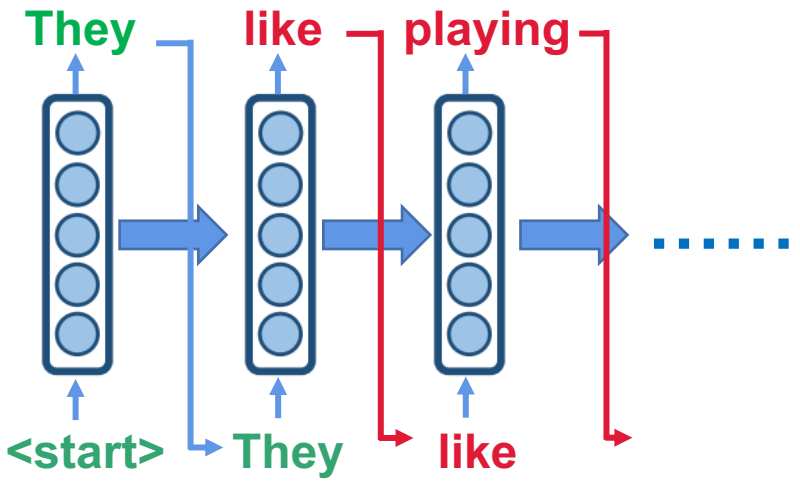
Teacher forcing is a procedure …, in which during training the model receives the ground truth output $y^{(t)}$ as input at time $t + 1$.

图片来源：Deep Learning, Ian Goodfellow, Yoshua Bengio and Aaron Courville. 2016

5.2.2 模型训练与使用技巧

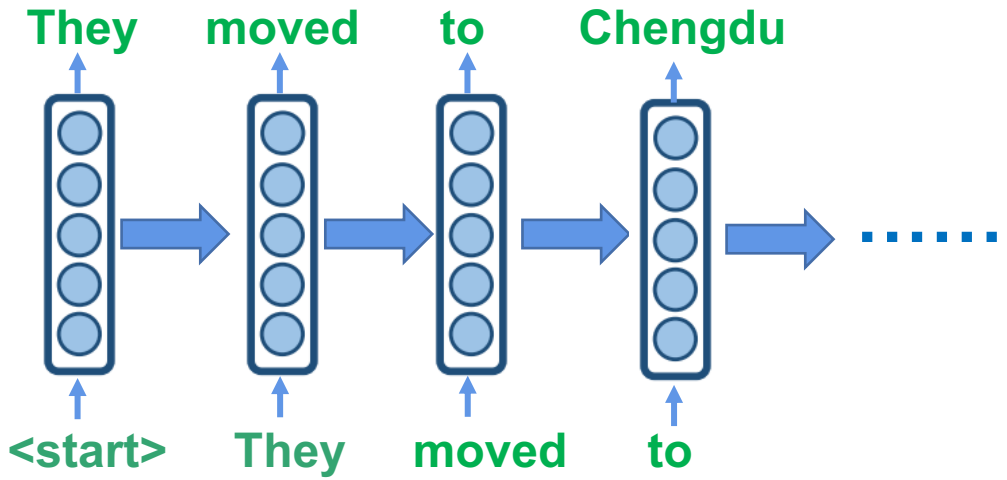
Teacher Forcing思想

在训练过程中，模型的每一步输入使用真实的目标序列（ground truth），而不是模型自己生成的输出。



神经元使用上一个时间步的输出作为当前神经元的输入

Free-running

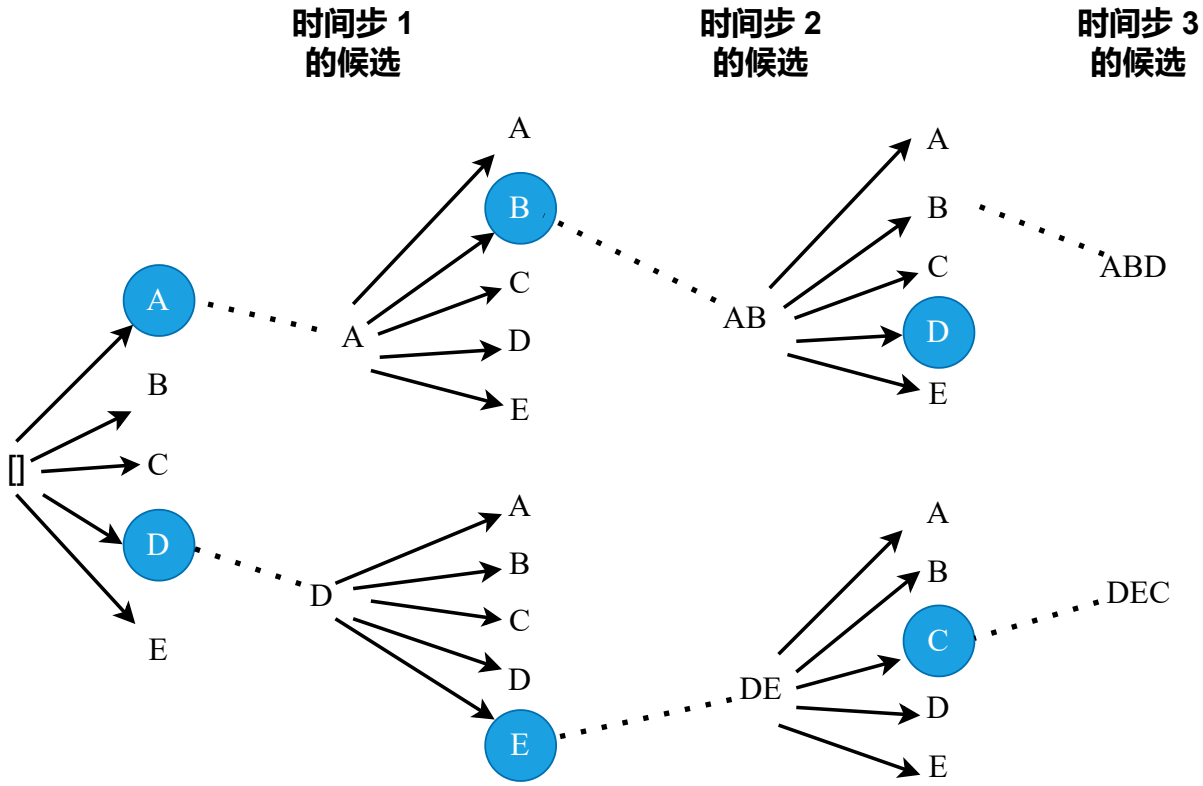


神经元直接使用正确输出作为当前神经元的输入

Teacher Forcing

5.2.2 模型训练与使用技巧

推理阶段无法使用Teacher Forcing，如何避免错误传递?



束搜索（Beam Search）

在每个时间步骤保留最有希望的多个候选项（而非仅保留一个最优候选项），并在后续步骤中继续扩展这些候选项。通过设置一个参数束宽度（Beam Width），束搜索可以控制搜索的宽度，即在每个时间步骤选择保留的最有希望的候选项数量。

5.2.2 模型训练与使用技巧

输入序列：I love Chengdu
输出序列：我爱成都

穷举搜索

穷举所有可能的输出序列，遍历所有可能的解空间来寻找最优解。

输出为3个时间步长，每个步长3种选择，共计27（3的3次方） 种排列组合。

我我我　我我爱　我我成都　我爱我　我爱爱　.....

从所有的排列组合中找到输出条件概率最大的序列。

5.2.2 模型训练与使用技巧

输入序列：I love Chengdu
输出序列：我爱成都

$$y_t = \text{argmax}_{y \in \mathcal{Y}} P(y|y_{t-1}, y_{t-2}, \cdots, y_1, c)$$

贪心搜索

在时间步 t 从词表中选择选择概率最高的单词作为输出，然后将其作为输入继续生成下一个单词，直到生成完整的序列。

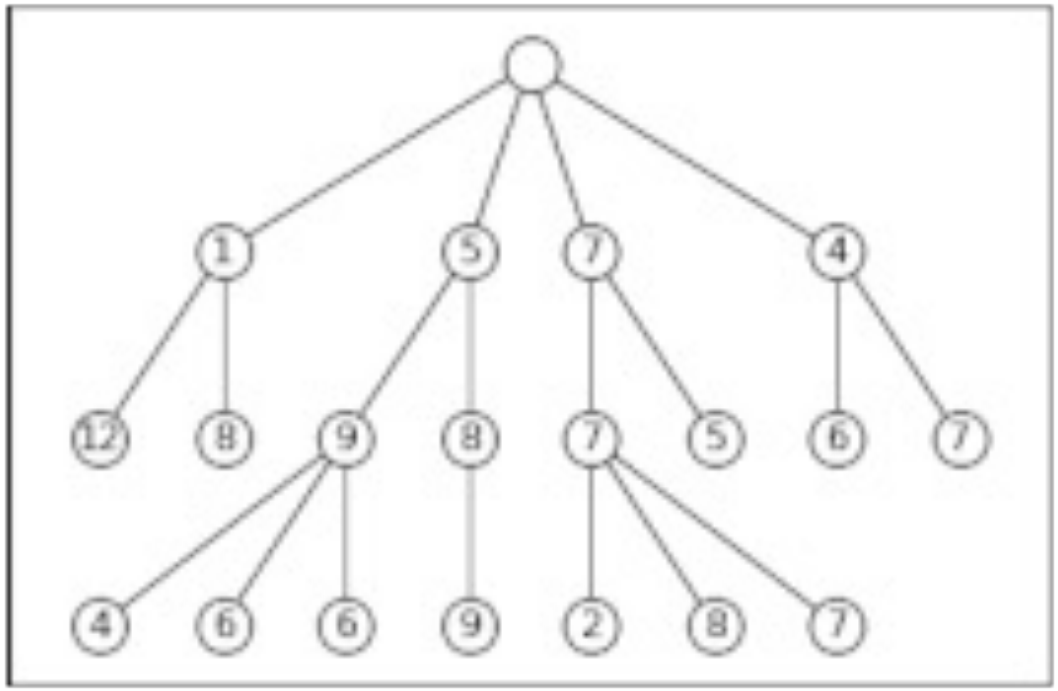
时间步	1	2	3
我	0.6	0.1	0.3
爱	0.3	0.8	0.1
成都	0.1	0.1	0.6

时间复杂度：$O(|\boldsymbol{y}|T)$

5.2.2 模型训练与使用技巧

束搜索

在时间步 1，选择当前条件概率最大的k（束宽）个词，当做候选输出序列的第一个词。之后的每个时间步长，基于上个步长的输出序列，从中挑出具有最高条件概率的k个候选输出序列。



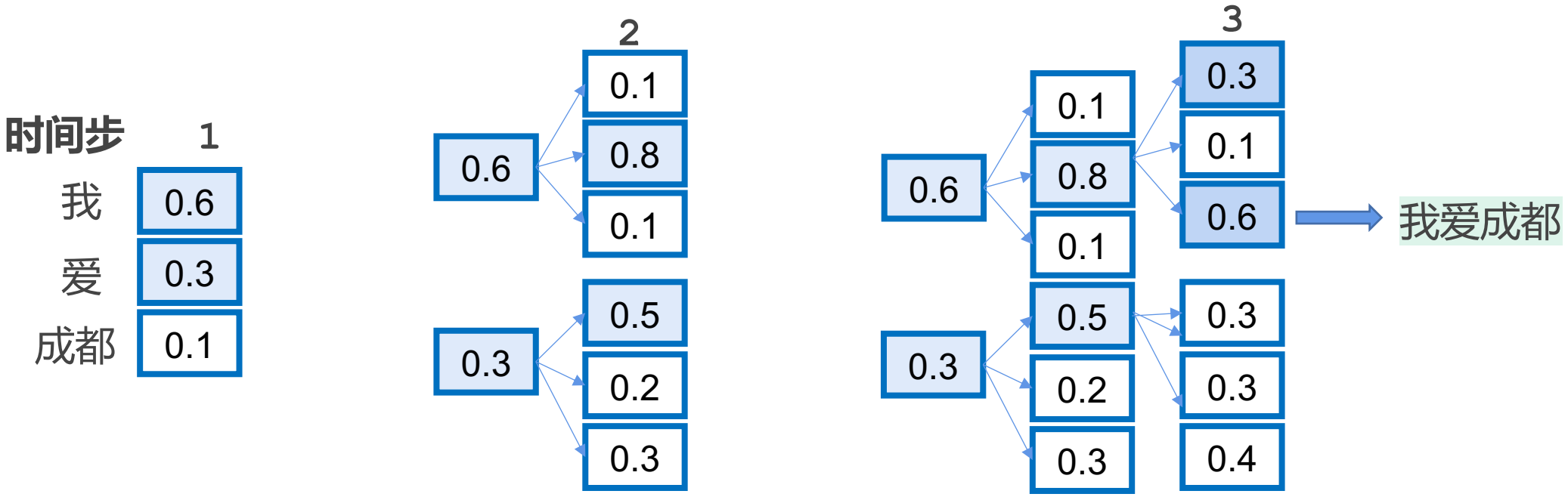
时间复杂度：$O(|k||\boldsymbol{y}|T)$

图片来源：https://en.wikipedia.org/wiki/Beam_search

5.2.2 模型训练与使用技巧

举例

输入序列：I love Chengdu
输出序列：我爱成都



目录

5.2 Seq2Seq模型

5.2.1 模型结构

5.2.2 模型训练与使用技巧

5.3 注意力机制

5.3.1 定义与原理

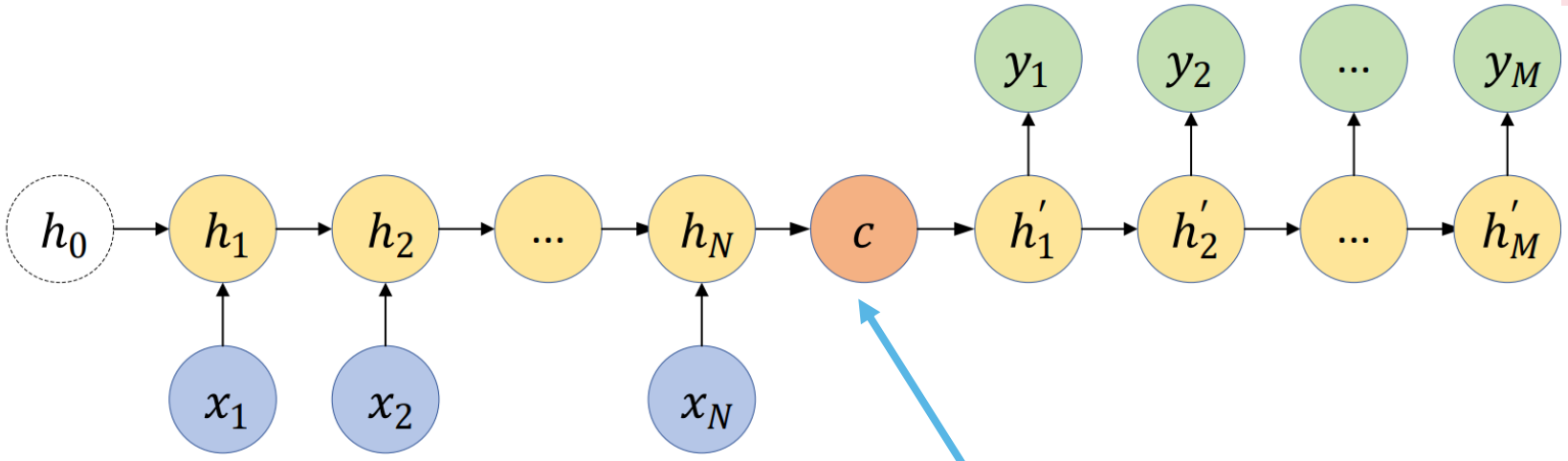
5.3.2 引入注意力机制的编码器-解码器模型

5.3.3 查询、键和值

5.3 注意力机制

信息丢失问题

编码器需要将整个输入序列编码成一个固定长度的向量，但是当输入信息太长时，固定长度的向量不能满足信息存储的要求，从而导致信息丢失。



1. 整个序列的“意义”都被打包到这个上下文中，然后编码器不再与解码器交互。

2. 序列中每个单词的贡献一样。

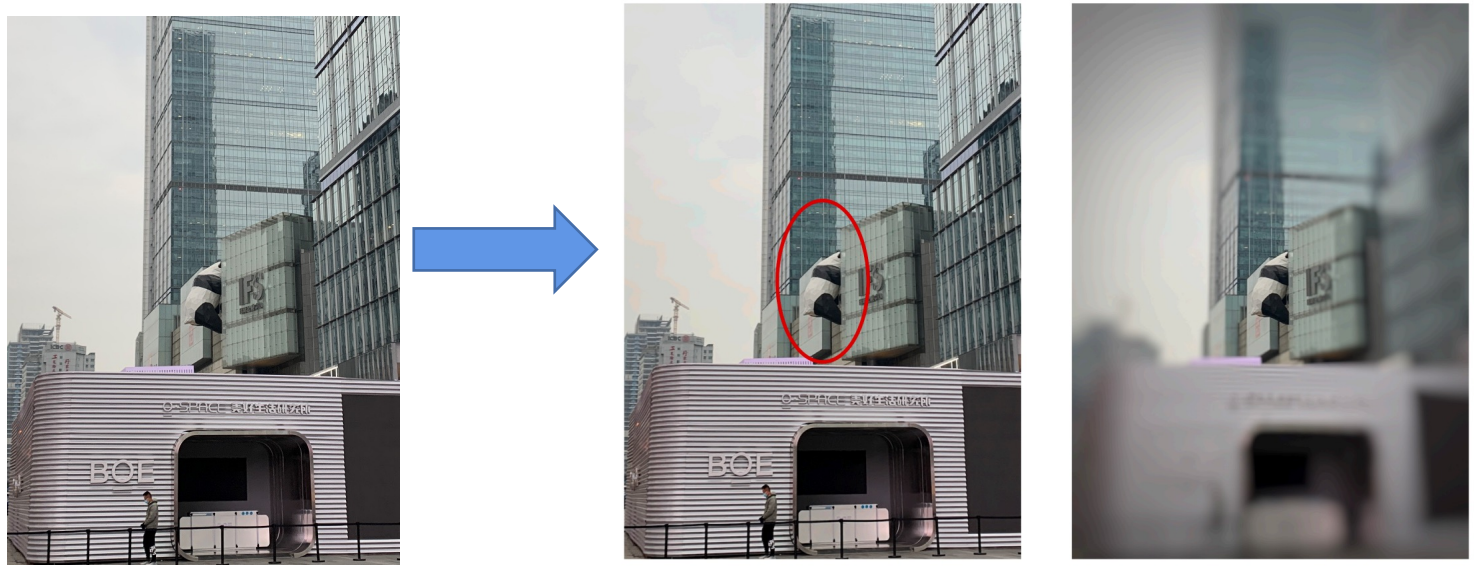
是否可以只关注输入序列的最重要信息?

5.3.1 定义与原理

- **视觉中的注意力**

人眼通常并非全面审视图片的所有细节，而是将注意力集中在显著或重要的部分上。

从关注全部到关注重点



"The true art of memory is the art of attention."

Samuel Johnson, 1759

5.3.1 定义与原理

Published as a conference paper at ICLR 2015

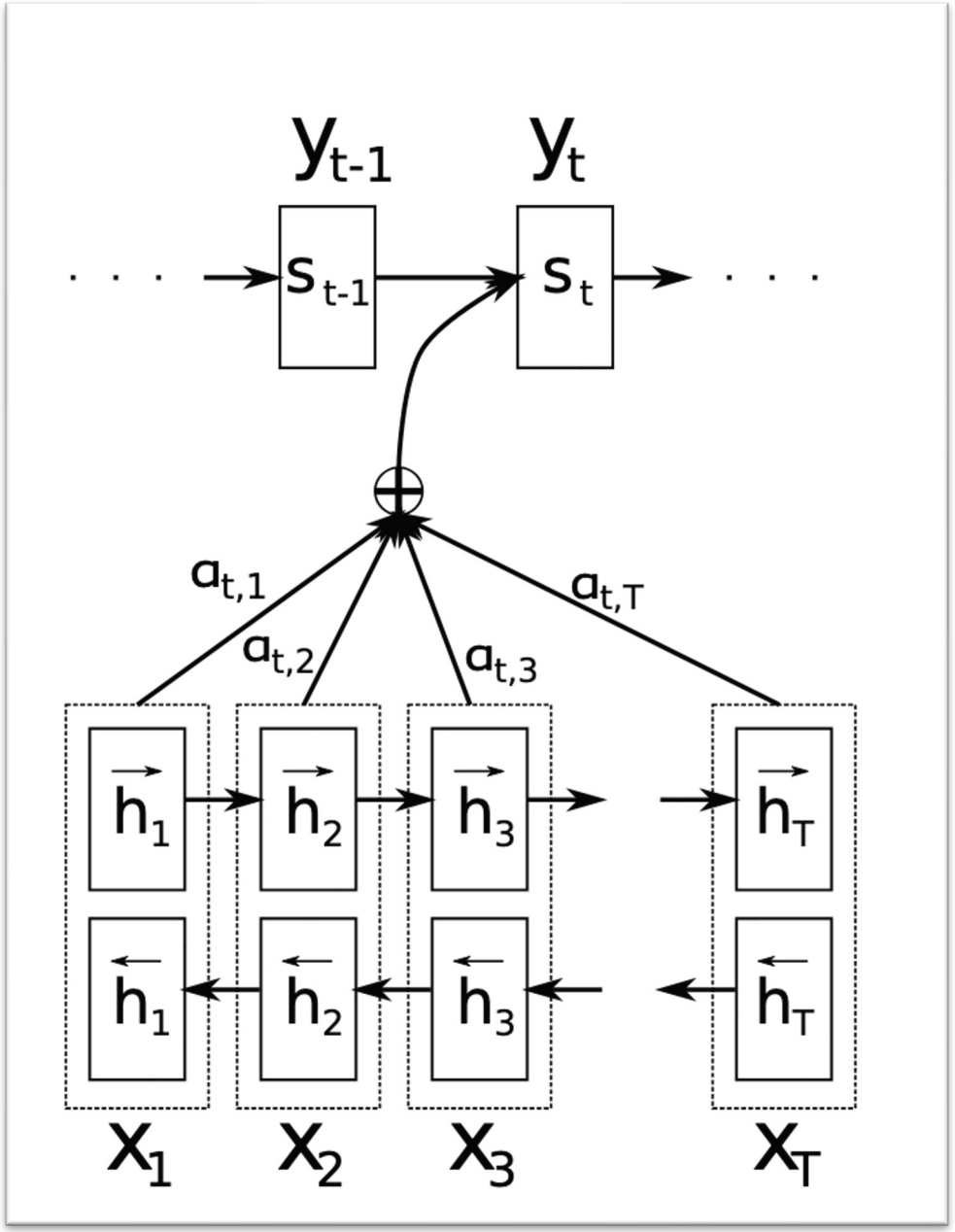
NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

ABSTRACT

Neural machine translation is a recently proposed approach to machine translation. Unlike the traditional statistical machine translation, the neural machine

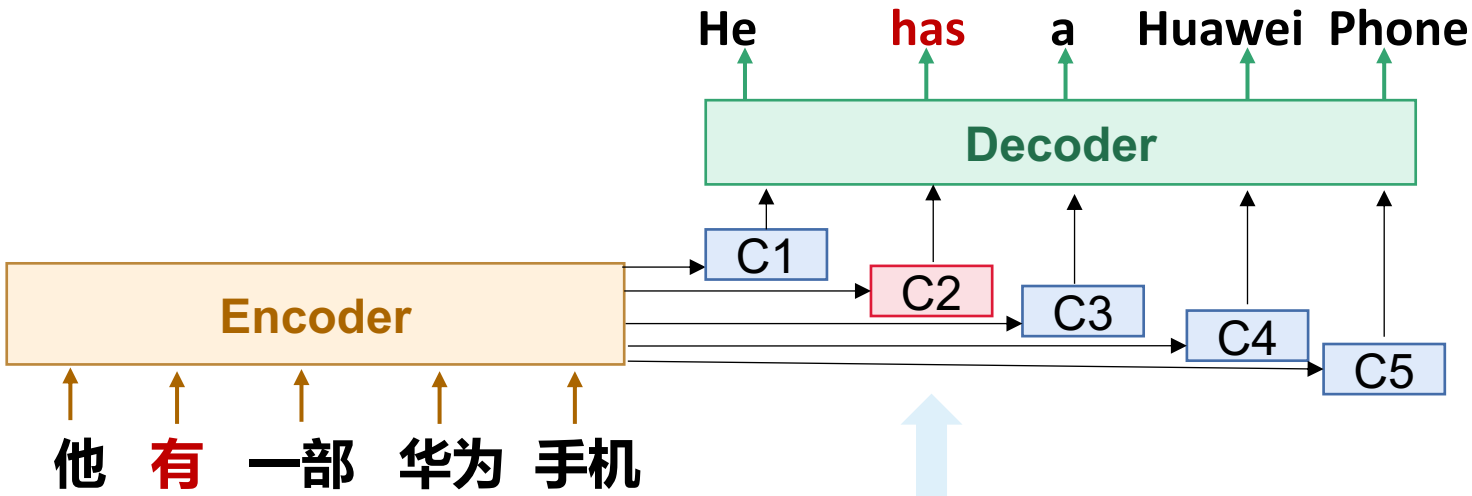


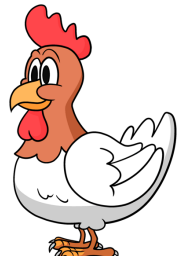
来源：https://arxiv.org/pdf/1409.0473

5.3.1 定义与原理

注意力机制（Attention Mechanism）

核心思想是从全局信息中筛选出局部且关键的信息进行更高效的处理。

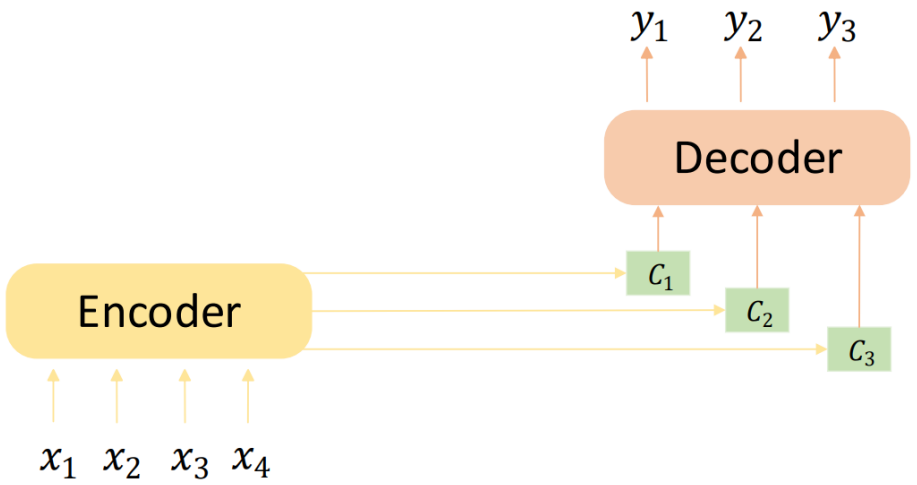




- **The chicken** didn't cross the road because **it** was too tired.
- **The chicken** didn't cross the road because **it** was too wide.

输入不再被压缩成单一的上下文向量，而是在输出的每个时间步，根据不同“注意力”重新计算关于输入序列上下文向量。

5.3.2 引入注意力机制的编码器-解码器模型



- 假设输入序列的长度是N，输出序列的长度是M，编码器生成一个动态长度的向量序列$(c_1, c_2, \cdots, c_M)$。

$$c_t = \sum_{i=1}^{N} \alpha_{ti} h_i$$

$$\alpha_{ti} = \text{Softmax}(e_{ti}) = \text{Softmax}(\mathcal{A}(h'_{t-1}, h_i))$$

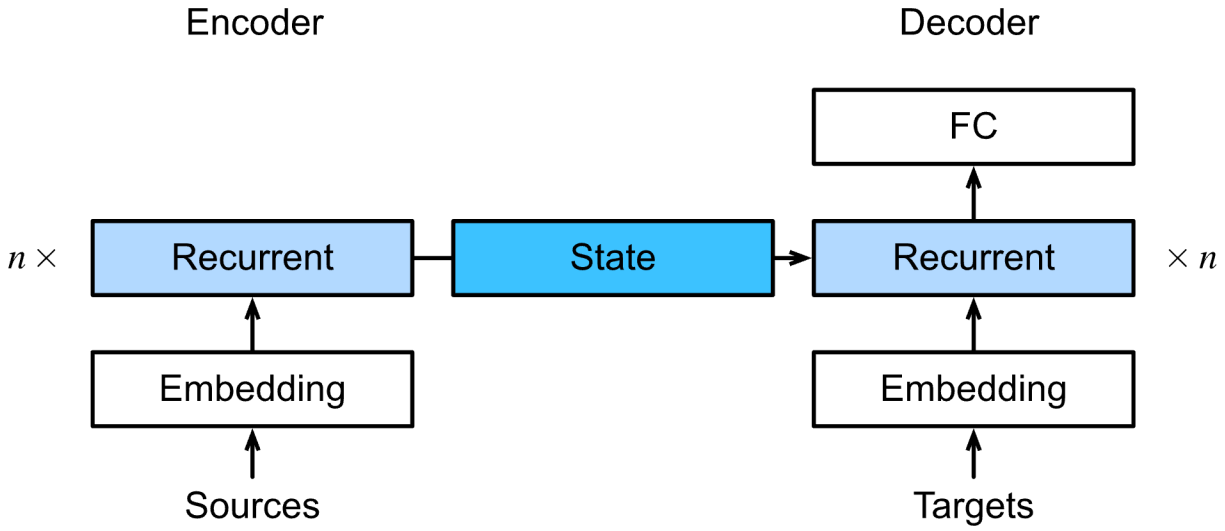
表示输入序列位置 i 和输出时间步 t 的匹配程度

- 在时间步 t，解码器的隐状态 h'_t和输出 y_t 分别可以表示为：

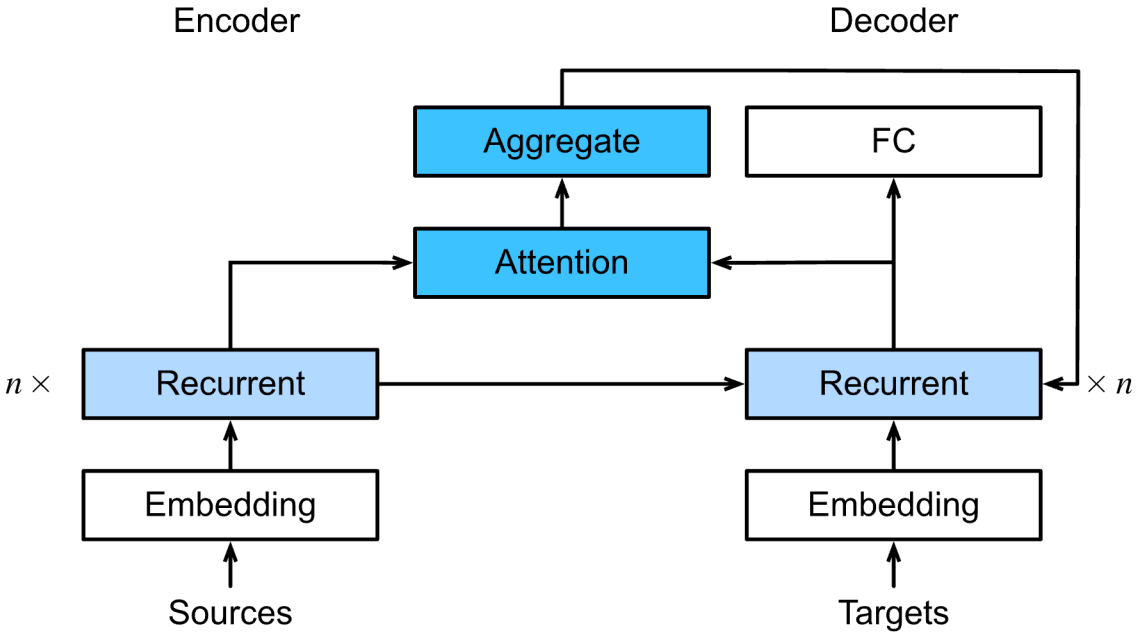
$$h'_t = f(c_t, h'_{t-1}, y_{t-1})$$

$$y_t = g(c_t, y_{t-1}, h'_t)$$

5.3.2 引入注意力机制的编码器-解码器模型



Seq-2-Seq



The Bahdanau Attention Mechanism

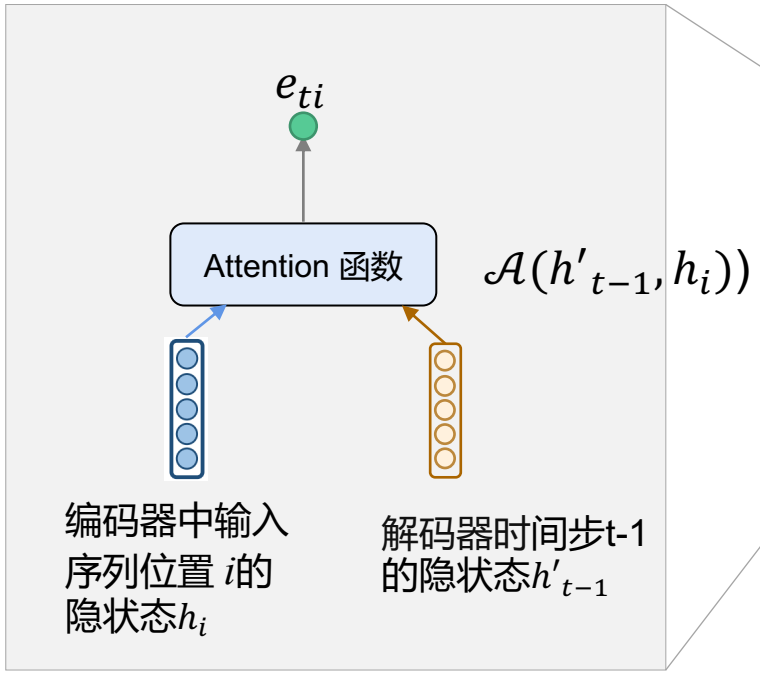
图片来源：https://d2l.ai/chapter_attention-mechanisms-and-transformers/bahdanau-attention.html

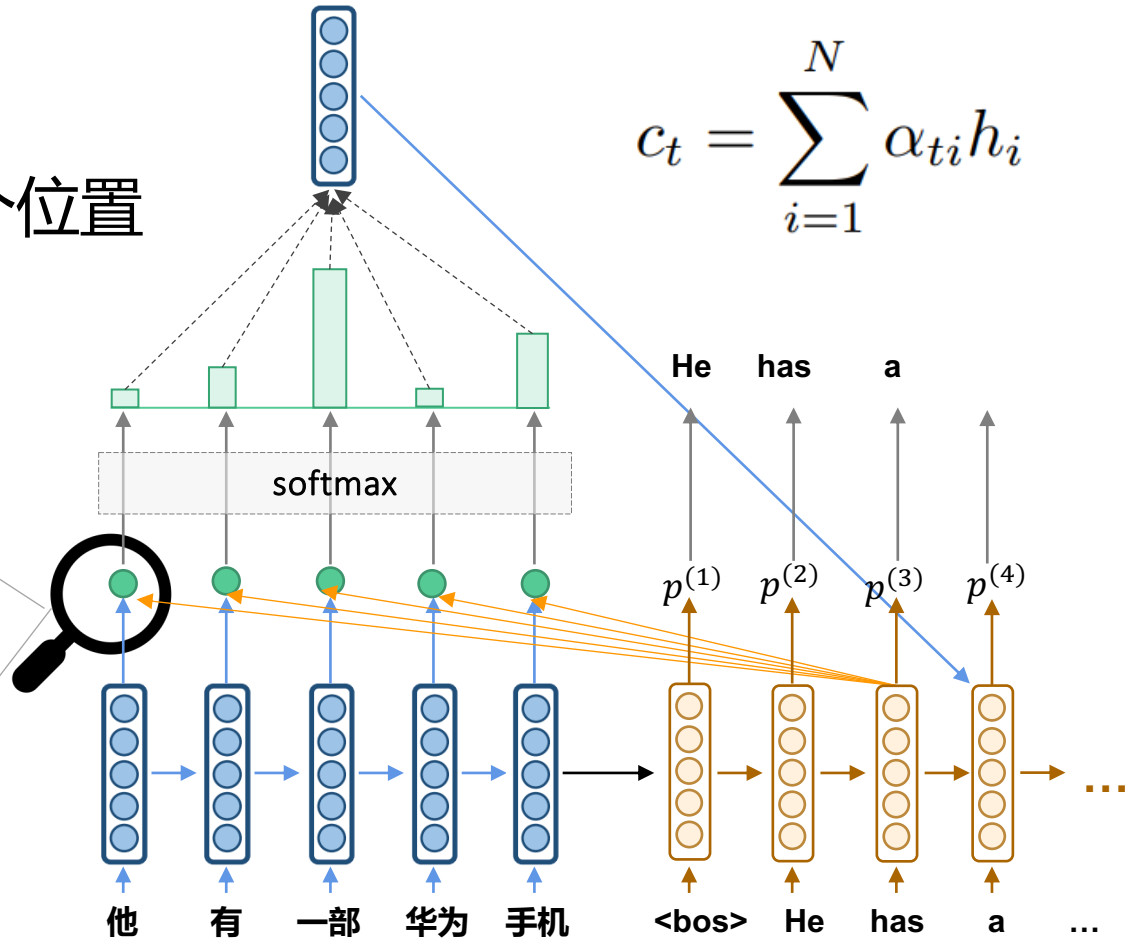
5.3.2 引入注意力机制的编码器-解码器模型

注意力输出：注意力权重的编码器状态的加权和

$$c_t = \sum_{i=1}^{N} \alpha_{ti} h_i$$

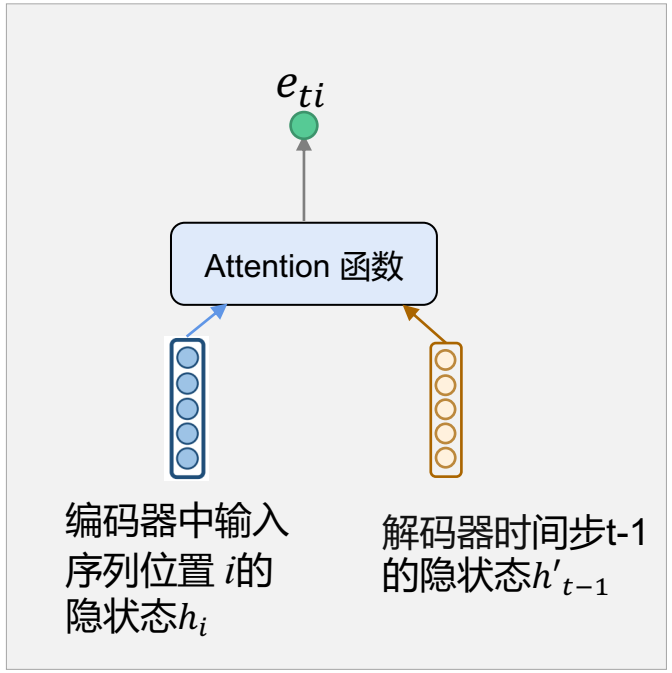
注意力权重：输入序列每个位置重要程度分布





5.3.2 引入注意力机制的编码器-解码器模型

注意力计算



注意力输出

(加权和)

注意力权重

(softmax)

注意力函数

输入

$$c_t = \sum_{i=1}^{N} \alpha_{ti} h_i$$

解码器时间步 t 的上下文向量

$$\alpha_{ti} = \text{Softmax}(e_{ti}) \; i = 1, \ldots, N$$

时间步t的输入位置 i 的注意力权重

$$e_{ti} = \mathcal{A}(h'_{t-1}, h_i) \; i = 1, \ldots, N$$

位置i与时间步 t-1 的相关性

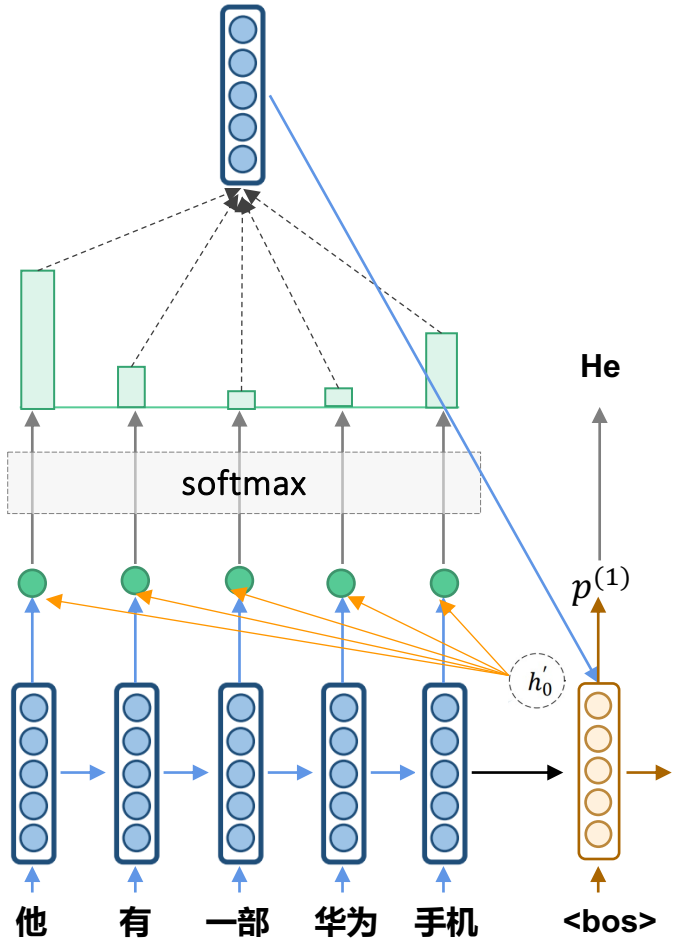
编码器所有位置的隐状态h_1, h_2, …, h_N

解码器时间步t-1的隐状态h'_{t-1}

5.3.2 引入注意力机制的编码器-解码器模型

输入序列:他 有 一部 华为 手机

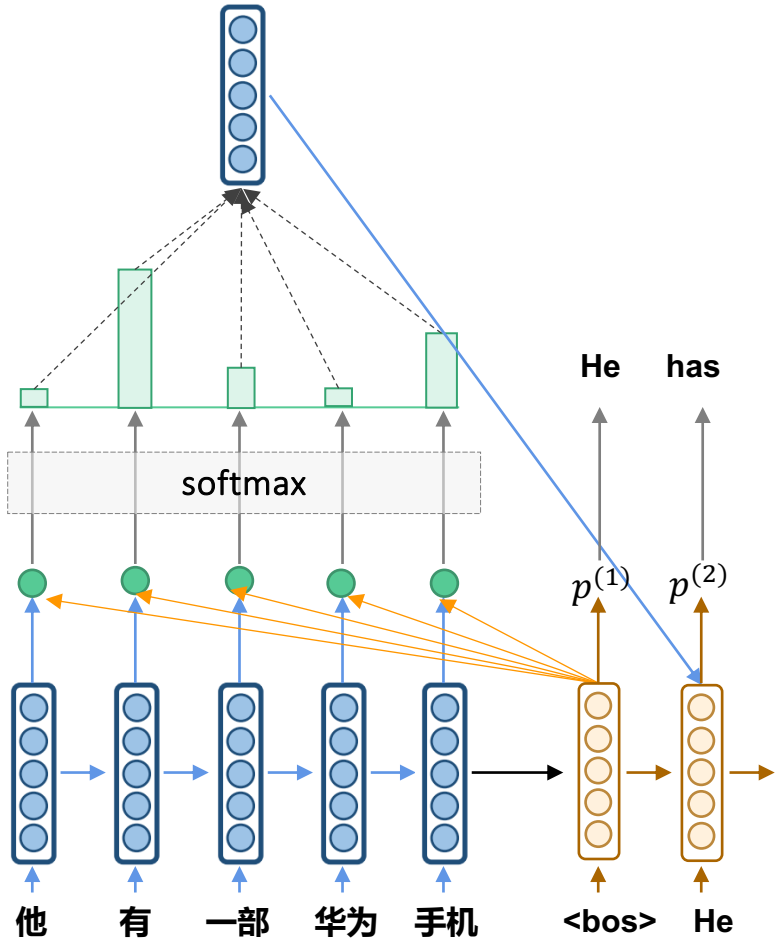
输出序列: He



5.3.2 引入注意力机制的编码器-解码器模型

输入序列:他 有 一部 华为 手机

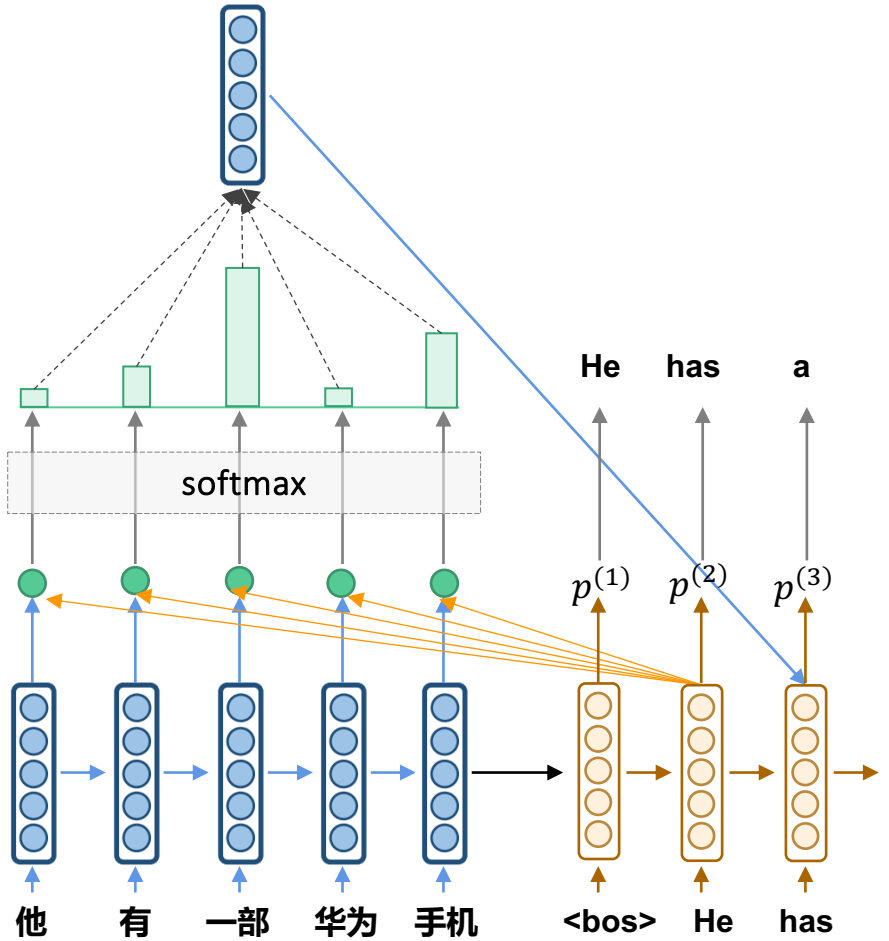
输出序列: He has



5.3.2 引入注意力机制的编码器-解码器模型

输入序列:他 有 一部 华为 手机

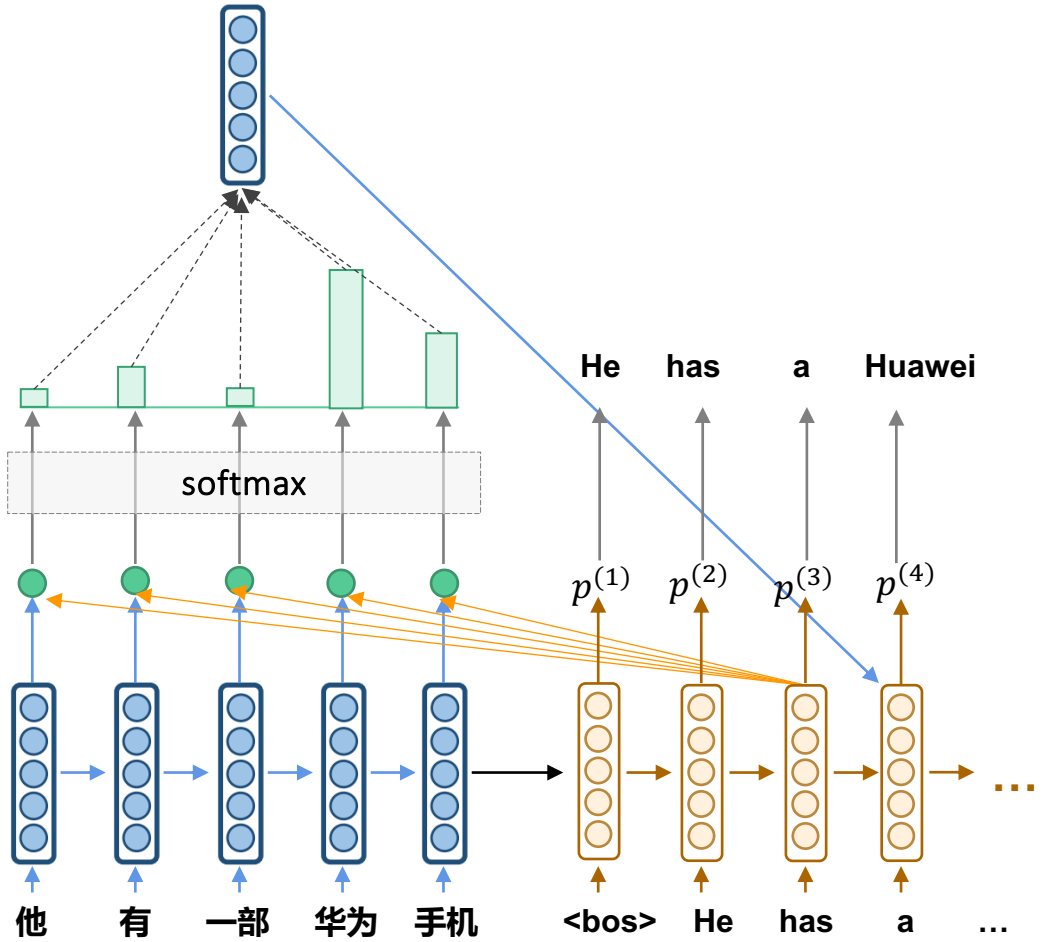
输出序列: He has a



5.3.2 引入注意力机制的编码器-解码器模型

输入序列：他 有 一部 华为 手机

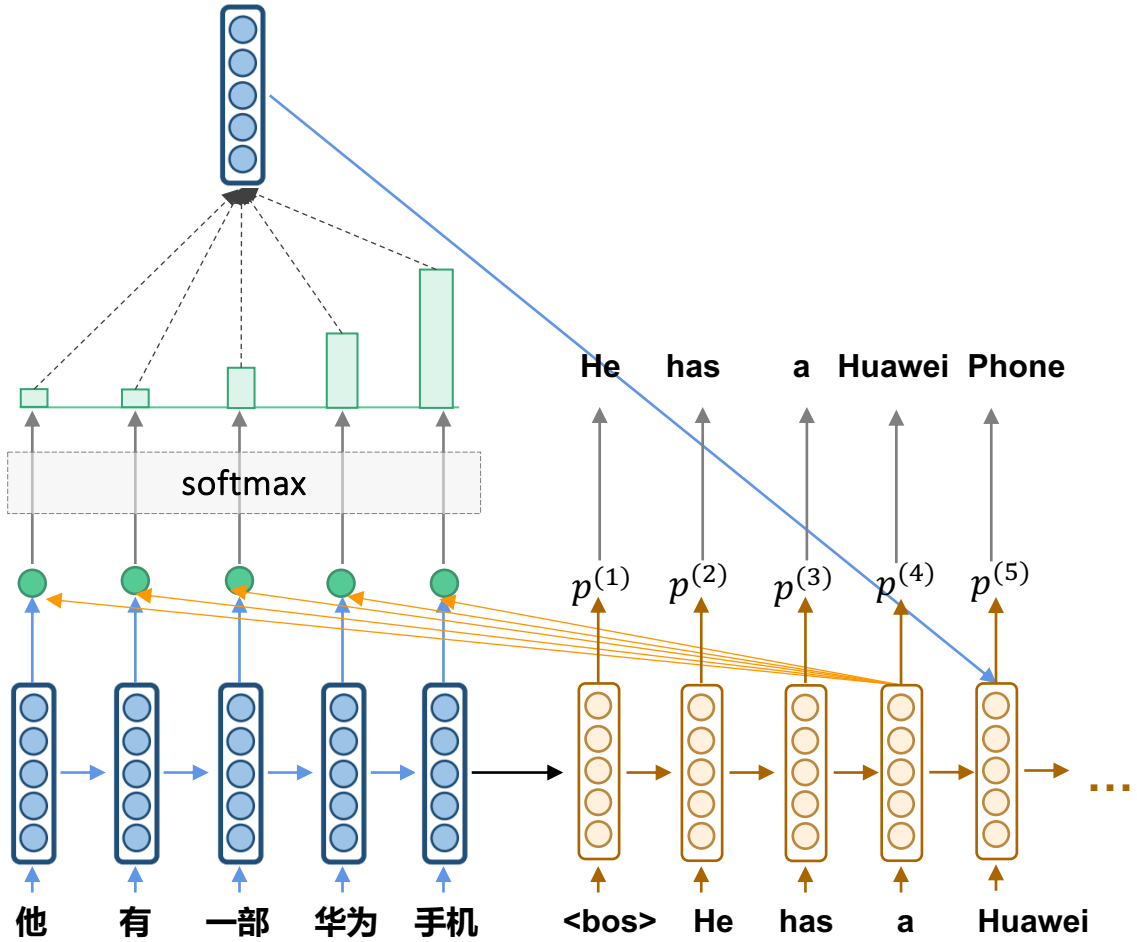
输出序列：He has a Huawei



5.3.2 引入注意力机制的编码器-解码器模型

输入序列：他 有 一部 华为 手机

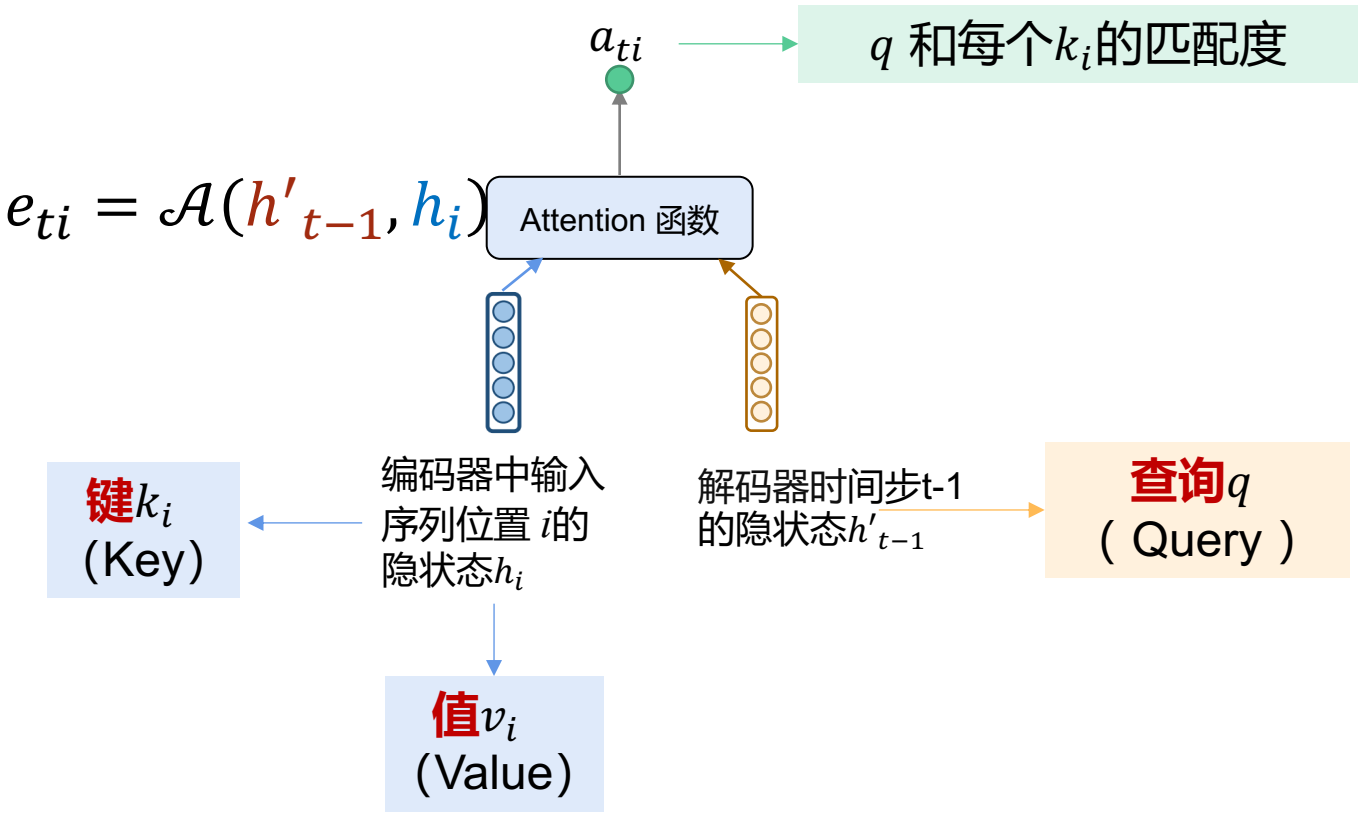
输出序列：He has a Huawei Phone

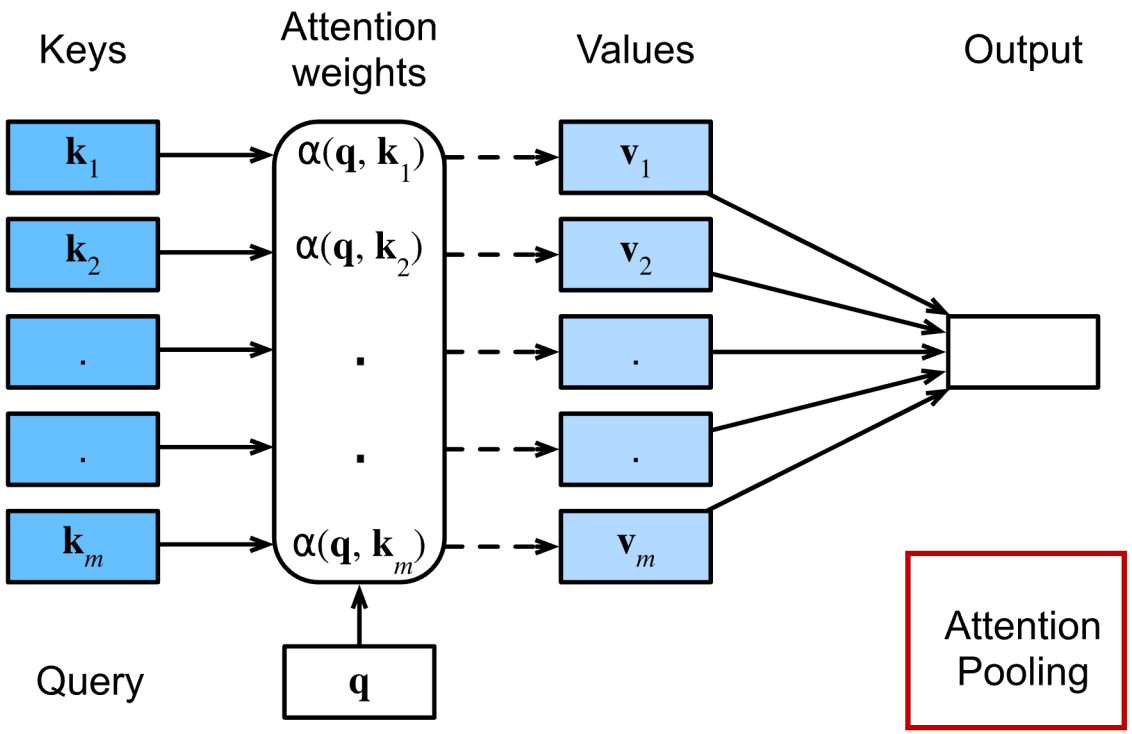


5.3.3 查询、键和值（Q、K和V）

数据库的键值对 $\mathcal{D} = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$

D 的注意力可以表示为 $\sum_{i=1}^{n} \alpha(q, k_i) v_i$





图片来源：https://d2l.ai/chapter_attention-mechanisms-and-transformers/queries-keys-values.html

5.3.3 查询、键和值（Q、K和V）

$$\sum_{i=1}^{n} \alpha(q, k_i) v_i$$

$$\alpha(\mathbf{q}, \mathbf{k}) = \exp\left(-\frac{1}{2}\|\mathbf{q} - \mathbf{k}\|^2\right)$$

$$\alpha(\mathbf{q}, \mathbf{k}) = 1 \text{ if } \|\mathbf{q} - \mathbf{k}\| \leq 1$$

$$\alpha(\mathbf{q}, \mathbf{k}) = \max(0, 1 - \|\mathbf{q} - \mathbf{k}\|)$$

$$a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i / \sqrt{d}.$$

缩放点积（Scaled Dot-Product）

这里忽略了Softmax操作，仅关注“相关”计算

目录

5.4 Transformer 模型

5.4.1 模型整体结构

5.4.2 模型细节

5.5 预训练语言模型

5.5.1 BERT 模型

5.5.2 GPT-1 模型

5.6 语言模型使用范式

5.6.1 预训练-传统微调范式

5.6.2 大模型-提示工程范式

Transformer

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

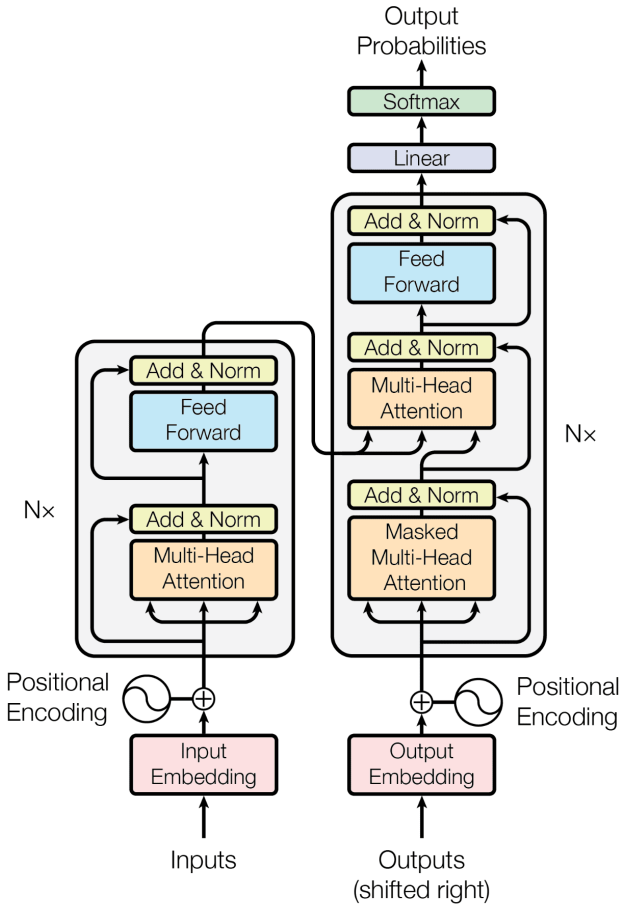
Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

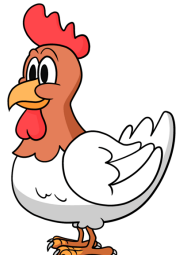


图片来源：https://arxiv.org/pdf/1706.03762

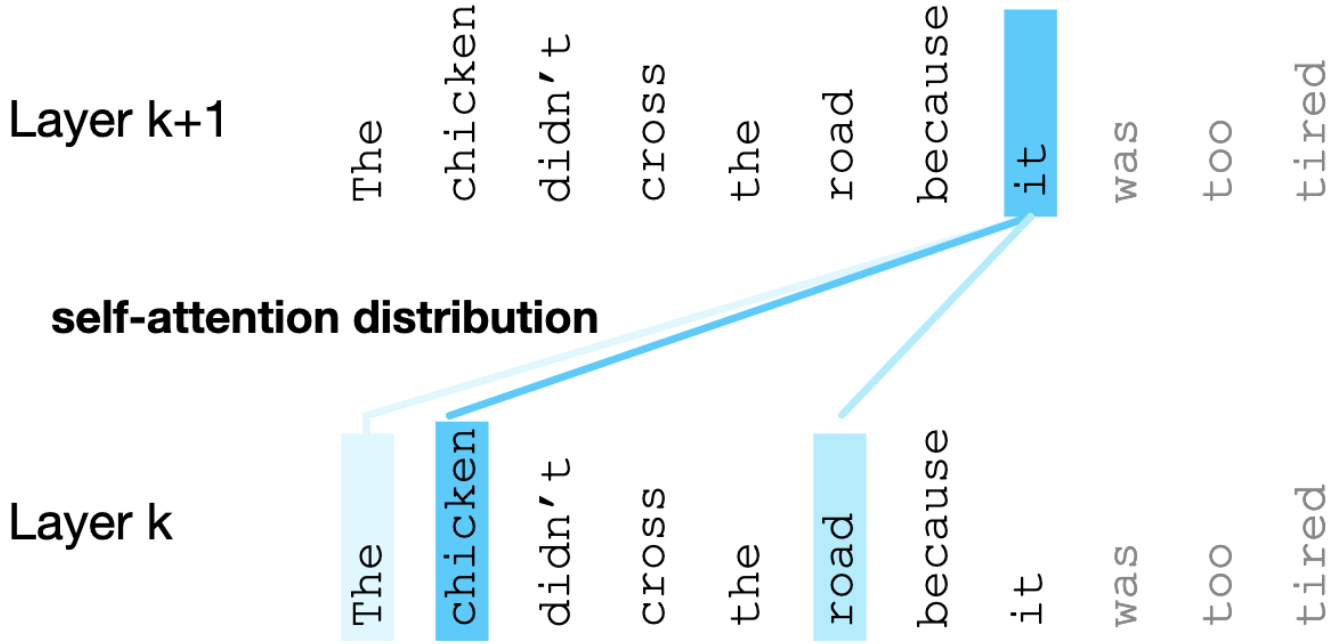
5.4.0 自注意力机制

给定输入序列$x_1, \ldots, x_n$（$x_i \in \mathbb{R}^d$），它的**自注意力（Self-Attention）**输出是相同长度的序列$a_1, \ldots a_n$，其中

$$a_i = f\left(x_i, (x_1, x_1), \ldots, (x_n, x_n)\right) \in \mathbb{R}^d$$



- **The chicken** didn't cross the road because **it** was too tired.
- **The chicken** didn't cross the road because **it** was too wide.



图片来源：https://web.stanford.edu/~jurafsky/slp3/9.pdf

5.4.0 自注意力机制

1. 自注意力机制的Q、K和V

Transformer通过引入三个权重矩阵，将x_i表示成Q、K和V：

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{Q}};\quad \mathbf{k}_j = \mathbf{x}_j\mathbf{W}^{\mathbf{K}};\quad \mathbf{v}_j = \mathbf{x}_j\mathbf{W}^{\mathbf{V}}$$

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

$W^Q \in \mathbb{R}^{d\times d_k}, W^K \in \mathbb{R}^{d\times d_k}, W^V \in \mathbb{R}^{d\times d_v}$.

在原始论文中，$d = 512$, $d_k = d_v = 64$

5.4.0 自注意力机制

2. 多头（Multi-Head）

每个单个自注意力可以被认为是关注输入数据的某个方面，通过叠加实现“多头”（h 个并行执行）可以扩展模型的能力。

$$\mathbf{q}_i^c = \mathbf{x}_i \mathbf{W}^{\mathbf{Qc}};\quad \mathbf{k}_j^c = \mathbf{x}_j \mathbf{W}^{\mathbf{Kc}};\quad \mathbf{v}_j^c = \mathbf{x}_j \mathbf{W}^{\mathbf{Vc}};$$

$$\text{score}^c(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i^c \cdot \mathbf{k}_j^c}{\sqrt{d_k}}$$

$$\alpha_{ij}^c = \text{softmax}(\text{score}^c(\mathbf{x}_i, \mathbf{x}_j))$$

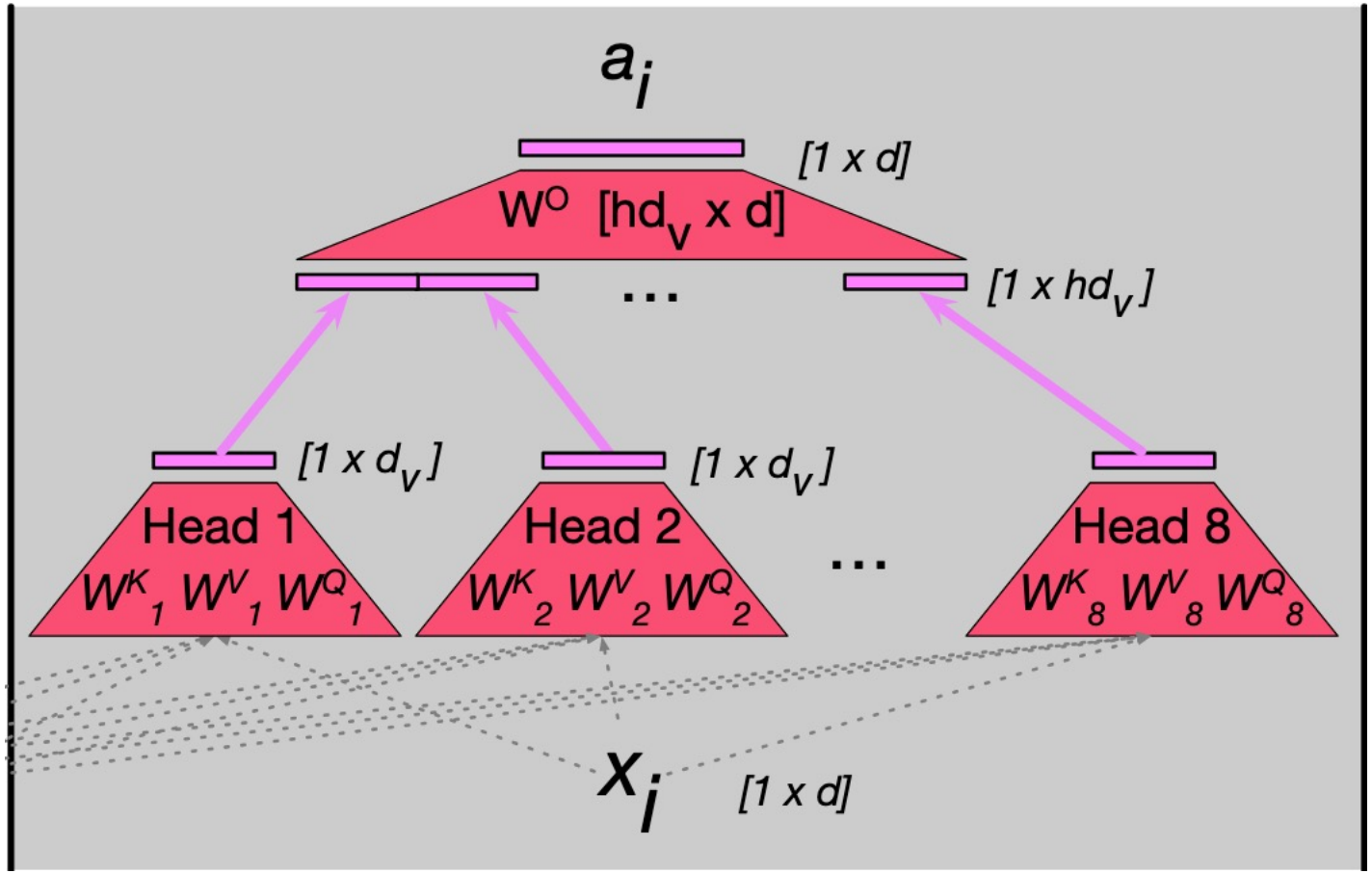
$$\mathbf{head}_i^c = \sum_{j \le i} \alpha_{ij}^c \mathbf{v}_j^c$$

$W^O \in \mathbb{R}^{hd_v \times d}$

$$\mathbf{a}_i = (\mathbf{head}^1 \oplus \mathbf{head}^2 \ldots \oplus \mathbf{head}^h)\mathbf{W}^O$$

5.4.0 自注意力机制

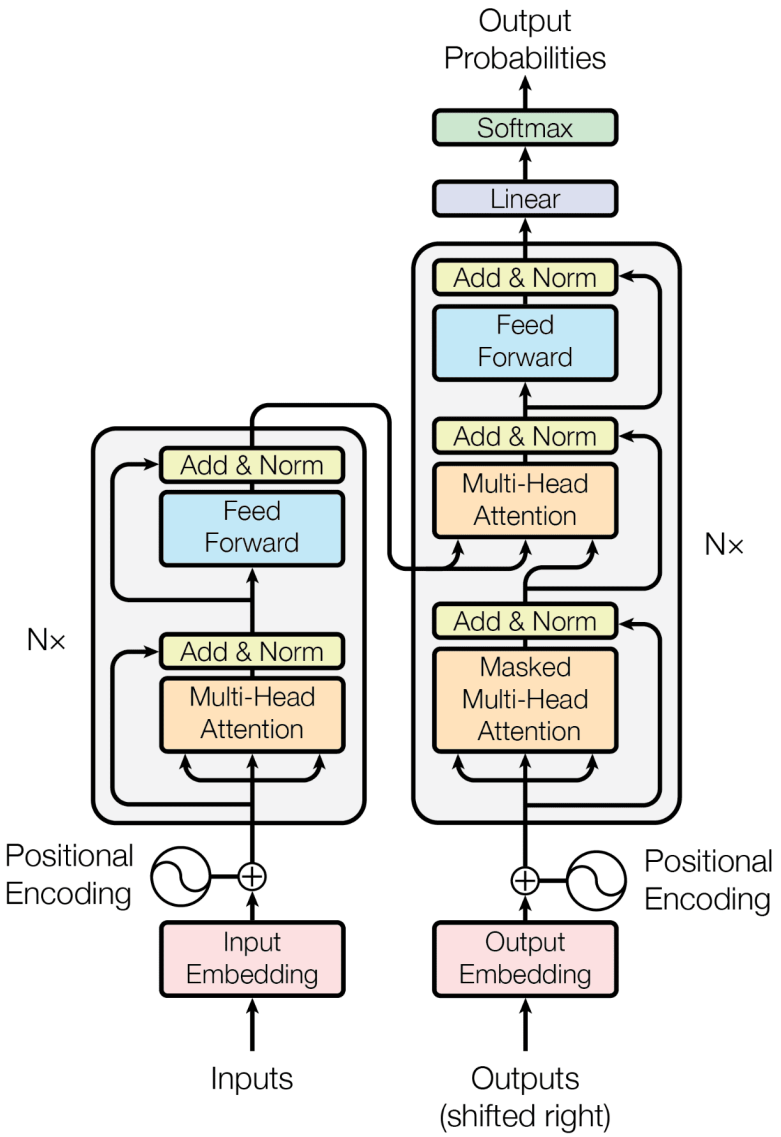
2. 多头（Multi-Head）

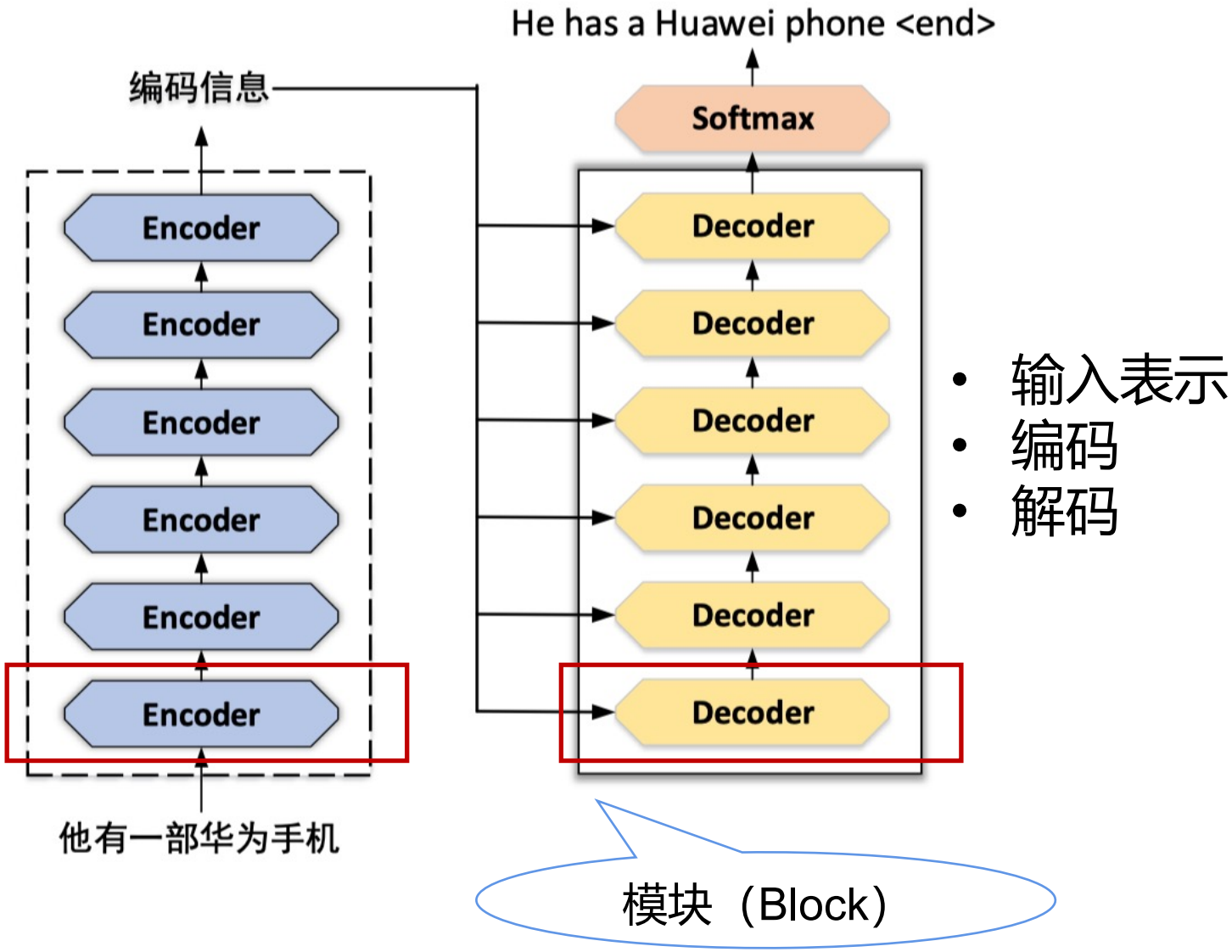


图片来源：https://web.stanford.edu/~jurafsky/slp3/9.pdf

5.4.1 模型的整体结构

可视化：https://bbycroft.net/llm





- 输入表示
- 编码
- 解码

5.4.1 模型的整体结构

1. 输入表示

输入的文本经过分词算法得到长度为 N **词元**（Token）序列，再进一步得到向量序列$\mathbb{R}^{N\times d}$。

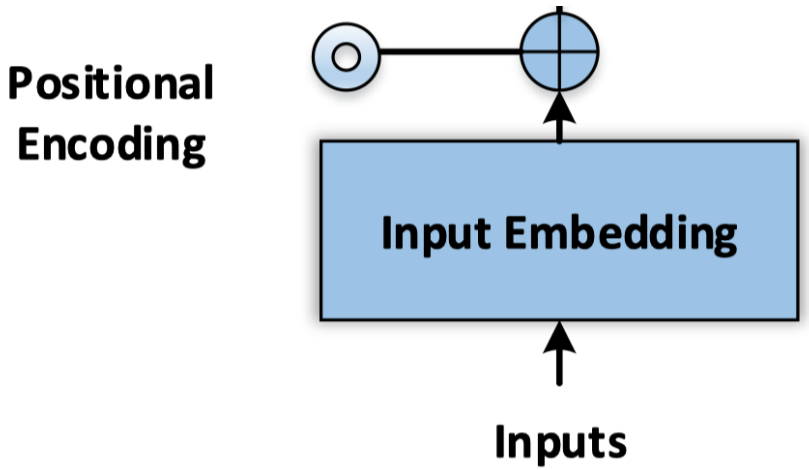
https://platform.openai.com/tokenizer

5.4.1 模型的整体结构

1. 输入表示

输入的文本经过分词算法得到长度为 n 的 **词元**（Token）序列，再进一步得到向量序列 $\mathrm{X} \in \mathbb{R}^{n \times d}$ 。每个向量由两部分组成：词元嵌入 + **位置嵌入**（两者之和）。

位置嵌入用于编码词元在序列中的绝对/相对位置，从而赋予其顺序感知的能力。



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

词元在序列中的位置

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

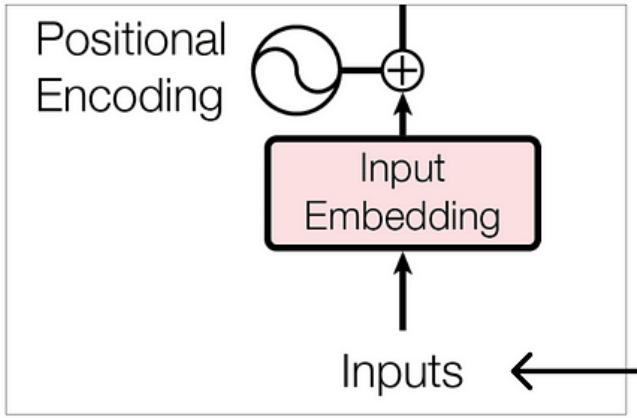
奇数维度位置

5.4.1 模型的整体结构

例子：原始输入转换词元编号

1	2	3	4	5	6	7	8	9	10	11	12
I	drink	things	Know	When	won't	play	out	true	storm	brings	game

13	14	15	16	17	18	19	20	21	22	23
the	win	of	enemy	you	wait	thrones	and	or	die	He

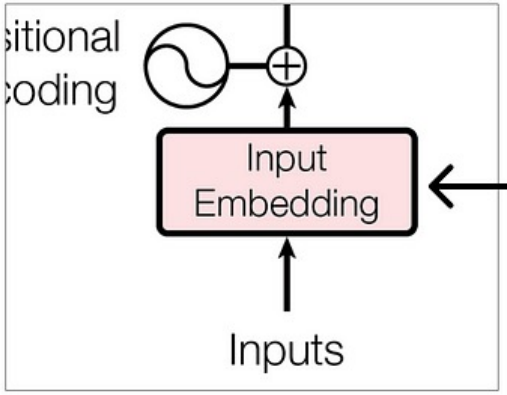


When	you	play	game	of	thrones
5	17	7	12	15	19

图片及例子来源：https://levelup.gitconnected.com/understanding-transformers-from-start-to-end-a-step-by-step-math-example-16d4e64e6eb1

5.4.1 模型的整体结构

例子：计算词元嵌入



When	you	play	game	of	thrones
5	17	7	12	15	19

e1	e2	e3	e4	e5	e6
0.79	0.38	0.01	0.12	0.88	0.6
0.6	0.12	0.51	0.6	0.41	0.33
0.96	0.06	0.27	0.65	0.79	0.75
0.64	0.79	0.31	0.22	0.62	0.48
0.97	0.9	0.56	0.07	0.5	0.94
0.2	0.74	0.59	0.37	0.7	0.21

为了方便演示，这里假设词元向量维度是6。注意：理论部分的矩阵是 $\mathbb{R}^{n\times d}$，*这里例子是* $\mathbb{R}^{d\times n}$

5.4.1 模型的整体结构

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

例子：计算位置嵌入（词元when）

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

d=6
pos=0

位置pos	e1	i	2i 或 2i + 1	公式	位置编码
0	0.79	0	0	$\sin(\frac{0}{10000^{0/6}})$	0
0	0.6	0	1	$\cos(\frac{0}{10000^{0/6}})$	1
0	0.96	1	2	$\sin(\frac{0}{10000^{2/6}})$	0
0	0.64	1	3	$\cos(\frac{0}{10000^{2/6}})$	1
0	0.97	2	4	$sin(\frac{0}{10000^{4/6}})$	0
0	0.2	2	5	$\cos(\frac{0}{10000^{4/6}})$	1

5.4.1 模型的整体结构

http://nlp.seas.harvard.edu/annotated-transformer/

代码

```
class PositionalEncoding(nn.Module):
    "Implement the PE function."

    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(
            torch.arange(0, d_model, 2) * -(math.log(10000.0) / d_model)
        )
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer("pe", pe)

    def forward(self, x):
        x = x + self.pe[:, : x.size(1)].requires_grad_(False)
        return self.dropout(x)
```

5.4.1 模型的整体结构

例子：词元嵌入+位置嵌入

When	you	play	game	of	thrones
5	17	7	12	15	19

Position Embedding Matrix

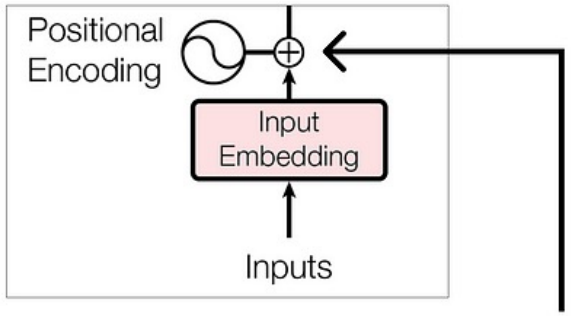
p1	p2	p3	p4	p5	p6
0	0.8415	0.9093	0.1411	-0.7568	-0.9589
1	0.0464	0.9957	0.1388	0.1846	0.9732
0	0.0022	0.0043	0.0065	0.0086	0.0108
1	0.0001	1	0.0003	0.0004	1
0	0	0	0	0	0
1	0	1	0	0	1

+

Word Embedding Matrix

e1	e2	e3	e4	e5	e6
0.79	0.38	0.01	0.12	0.88	0.6
0.6	0.12	0.51	0.6	0.41	0.33
0.96	0.06	0.27	0.65	0.79	0.75
0.64	0.79	0.31	0.22	0.62	0.48
0.97	0.9	0.56	0.07	0.5	0.94
0.2	0.74	0.59	0.37	0.7	0.21

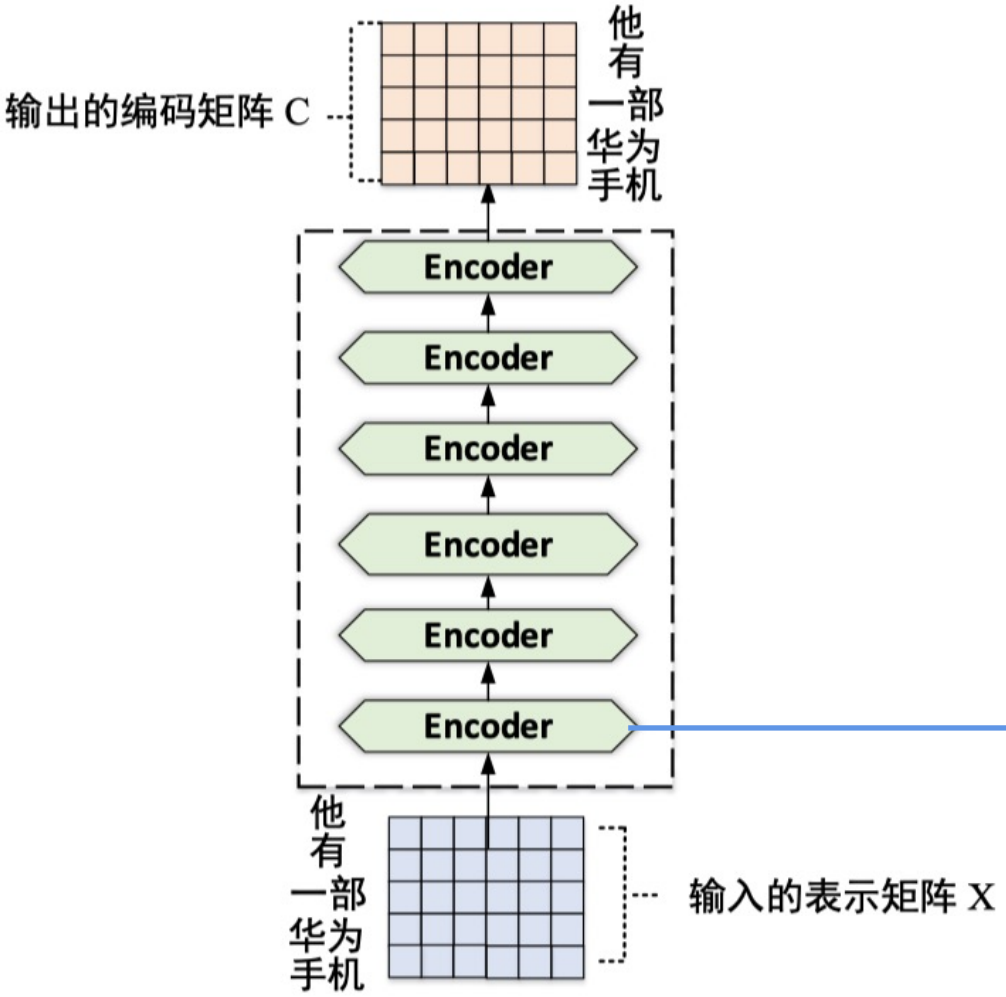
=

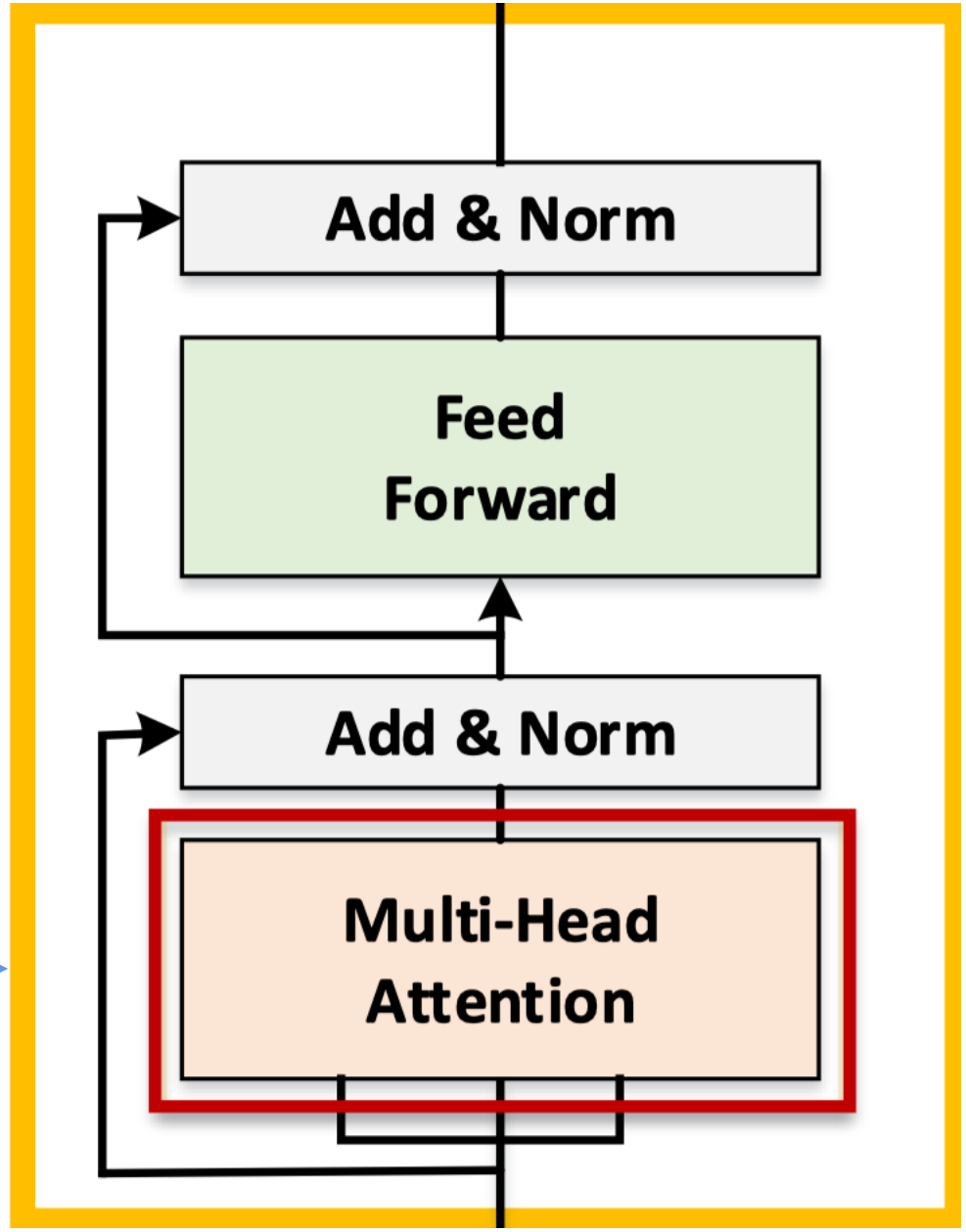


ep1	ep2	ep3	ep4	ep5	ep6
0.79	1.22	0.92	0.26	0.12	-0.36
1.6	0.17	1.51	0.74	0.59	1.3
0.96	0.06	0.27	0.66	0.8	0.76
1.64	0.79	1.31	0.22	0.62	1.48
0.97	0.9	0.56	0.07	0.5	0.94
1.2	0.74	1.59	0.37	0.7	1.21

5.4.1 模型的整体结构

2. 编码





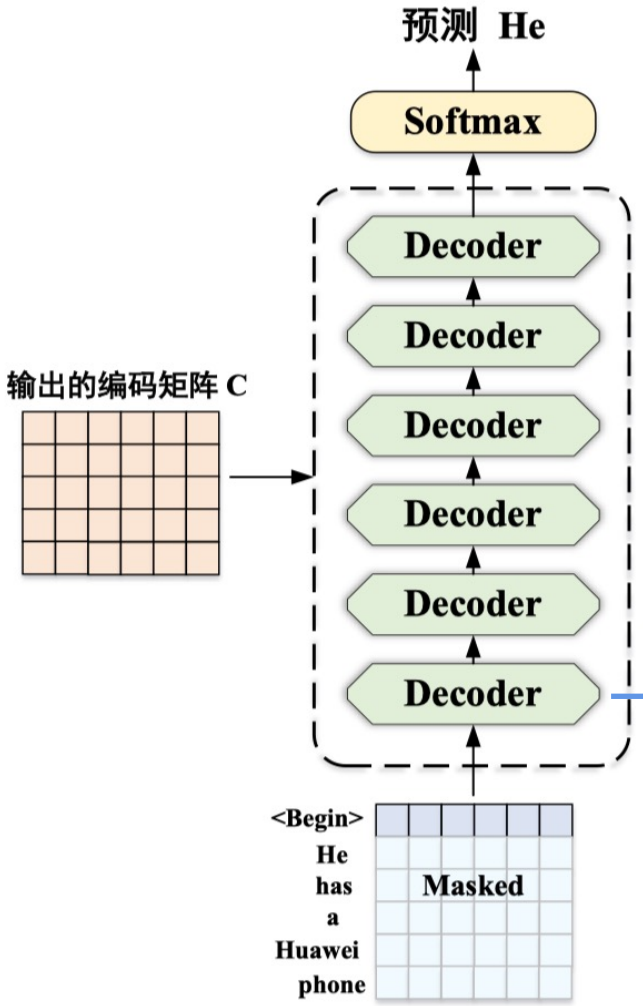
5.4.1 模型的整体结构

2. 编码

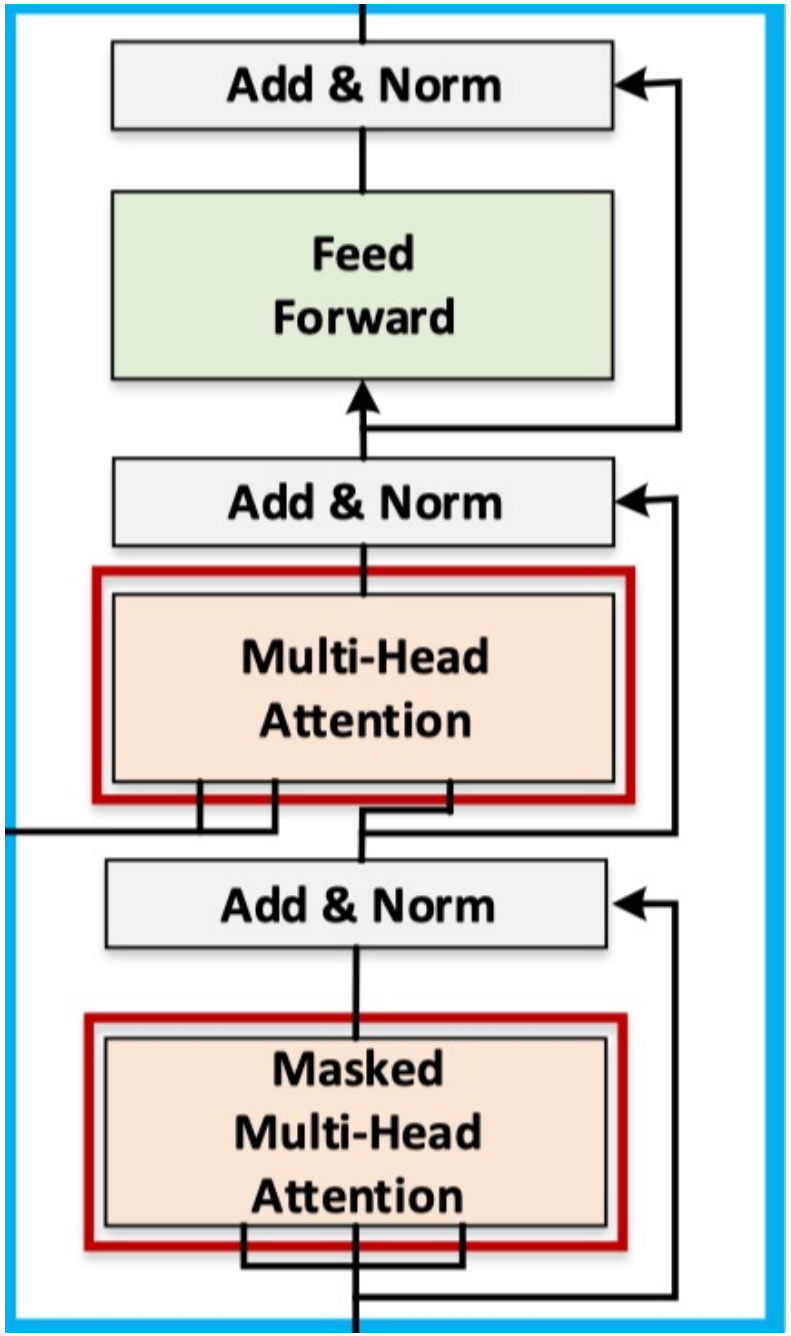
```
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"

    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

5.4.1 模型的整体结构

3. 解码



为了要防止模型“看到”未来的信息，需要使用**掩码**机制来遮盖掉当前位置之后的所有信息。



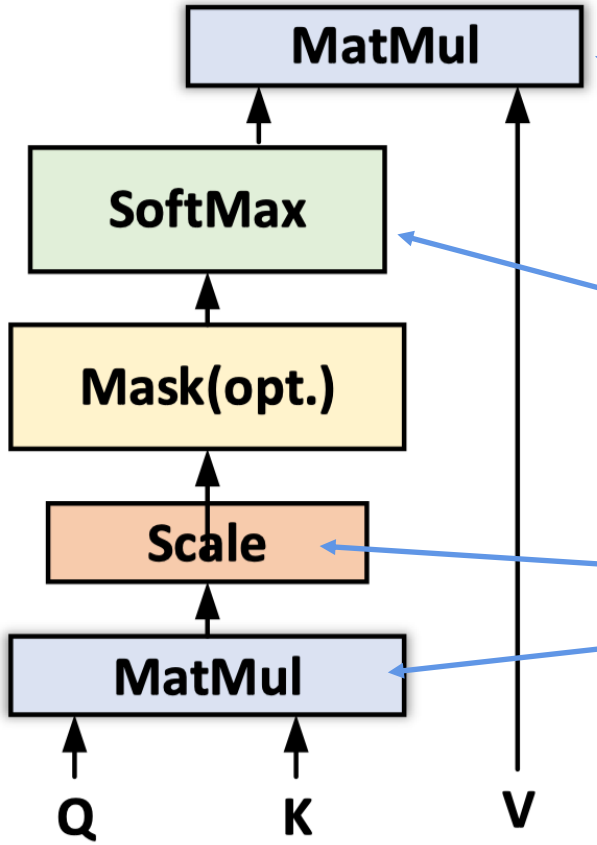
5.4.2 模型细节

1. 点积缩放注意力（单个头）

$$Q = X \cdot W_Q$$

$$K = X \cdot W_K$$

$$V = X \cdot W_V$$



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

编码过程不需要掩码（Mask）

5.4.2 模型细节

例子：Q，K和V计算示意图

$$Q = X \cdot W_Q$$

$$K = X \cdot W_K$$

$$V = X \cdot W_V$$

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6

X

Linear weights for query

0.52	0.45	0.91	0.69
0.05	0.85	0.37	0.83
0.49	0.1	0.56	0.61
0.71	0.64	0.4	0.14
0.76	0.27	0.92	0.67
0.85	0.56	0.57	0.07

6 x 4

5.4.2 模型细节

例子：Q，K和V计算示意图

$$Q = X \cdot W_Q$$

$$K = X \cdot W_K$$

$$V = X \cdot W_V$$

Query

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

key

3.71	4.04	4.15	3.41
2.18	2.51	1.64	1.93
3.28	3.11	3.65	3.01
1.07	1.13	1.64	1.35
1.49	1.97	2.14	1.81
2.51	3.04	3.45	2.28

6 x 4

value

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

5.4.2 模型细节

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

例子：点积注意力

$Q \in \mathbb{R}^{n \times d_k}$

$K \in \mathbb{R}^{n \times d_k}$

Query

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

X

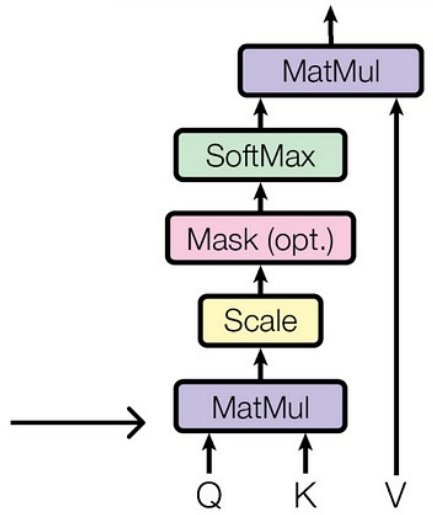
Transpose (Key)

3.71	2.18	3.28	1.07	1.49	2.51
4.04	2.51	3.11	1.13	1.97	3.04
4.15	1.64	3.65	1.64	2.14	3.45
3.41	1.93	3.01	1.35	1.81	2.28

4 x 6

=

58.341	31.2882	49.7306	19.7538	28.1886	43.1644
34.5402	18.2058	29.6169	11.7761	16.6133	25.6698
50.8796	27.3994	43.2409	17.0909	24.5349	37.695
18.1304	9.728	15.4544	6.2134	8.851	13.3894
26.3707	14.0937	22.5509	8.9445	12.6967	19.4858
41.8941	22.668	35.6369	14.004	20.103	30.9265



5.4.2 模型细节

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

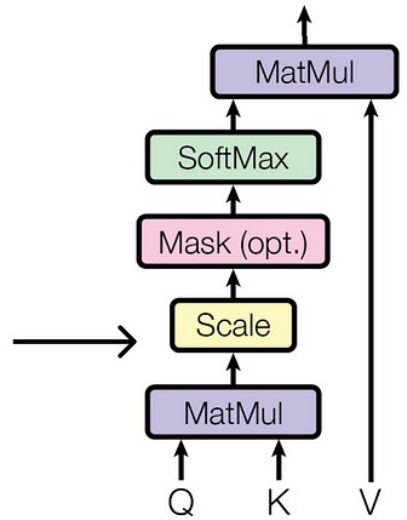
例子：缩放

58.341	31.2882	49.7306	19.7538	28.1886	43.1644
34.5402	18.2058	29.6169	11.7761	16.6133	25.6698
50.8796	27.3994	43.2409	17.0909	24.5349	37.695
18.1304	9.728	15.4544	6.2134	8.851	13.3894
26.3707	14.0937	22.5509	8.9445	12.6967	19.4858
41.8941	22.668	35.6369	14.004	20.103	30.9265

$\sqrt{d_k}$ where d (dimension) is 6

=

23.81721	12.77409	20.30219	8.062904	11.50852	17.62
14.1009	7.434201	12.09231	4.809165	6.781004	10.47973
20.77167	11.186	17.65266	6.976963	10.01433	15.39096
7.401542	3.972256	6.307436	2.535222	3.612997	5.466445
10.76551	5.752218	9.205999	3.64974	5.184753	7.956759
17.10152	9.254989	14.54997	5.715476	8.205791	12.62712



5.4.2 模型细节

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

例子：Softmax计算（对每一行）

$QK^T \in \mathbb{R}^{n\times n}$

23.82	12.77	20.3	8.06	11.51	17.62
14.1	7.43	12.09	4.81	6.78	10.48
20.77	11.19	17.65	6.98	10.01	15.39
7.4	3.97	6.31	2.54	3.61	5.47
10.77	5.75	9.21	3.65	5.18	7.96
17.1	9.25	14.55	5.72	8.21	12.63

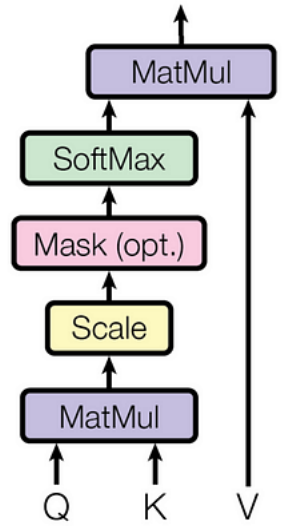
SoftMax

$$s\left(x_i\right)=\frac{e^{x_i}}{\sum_{j=1}^{n}e^{x_j}}$$

$$\text{softmax}(23.82)=\frac{e^{23.82}}{e^{23.82}+e^{12.77}+e^{20.3}+e^{8.06}+e^{11.51}+e^{17.62}}$$

=

0.9693	0	0.0287	0	0	0.002
0.86	0.0011	0.1152	0.0001	0.0006	0.023
0.9534	0.0001	0.0421	0	0	0.0044
0.6476	0.021	0.2177	0.005	0.0146	0.094
0.7803	0.0052	0.164	0.0006	0.0029	0.047
0.9174	0.0004	0.0716	0	0.0001	0.0105



5.4.2 模型细节

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

例子：加权求和

$QK^TV \in \mathbb{R}^{n\times d_v}$

$\text{softmax}\left(\frac{Q.K^T}{\sqrt{d_k}}\right)$

0.9693	0	0.0287	0	0	0.002
0.86	0.0011	0.1152	0.0001	0.0006	0.023
0.9534	0.0001	0.0421	0	0	0.0044
0.6476	0.021	0.2177	0.005	0.0146	0.094
0.7803	0.0052	0.164	0.0006	0.0029	0.047
0.9174	0.0004	0.0716	0	0.0001	0.0105

6 x 6

X

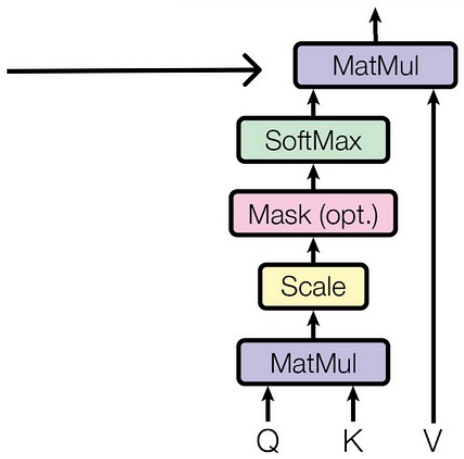
Value

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

=

3.864257	3.79246	4.060367	3.39751
3.801295	3.75252	3.977937	3.30861
3.855542	3.787426	4.04909	3.385086
3.622841	3.584936	3.750419	3.081834
3.745786	3.706744	3.904894	3.233519
3.835366	3.77523	4.022837	3.356435



5.4.2 模型细节

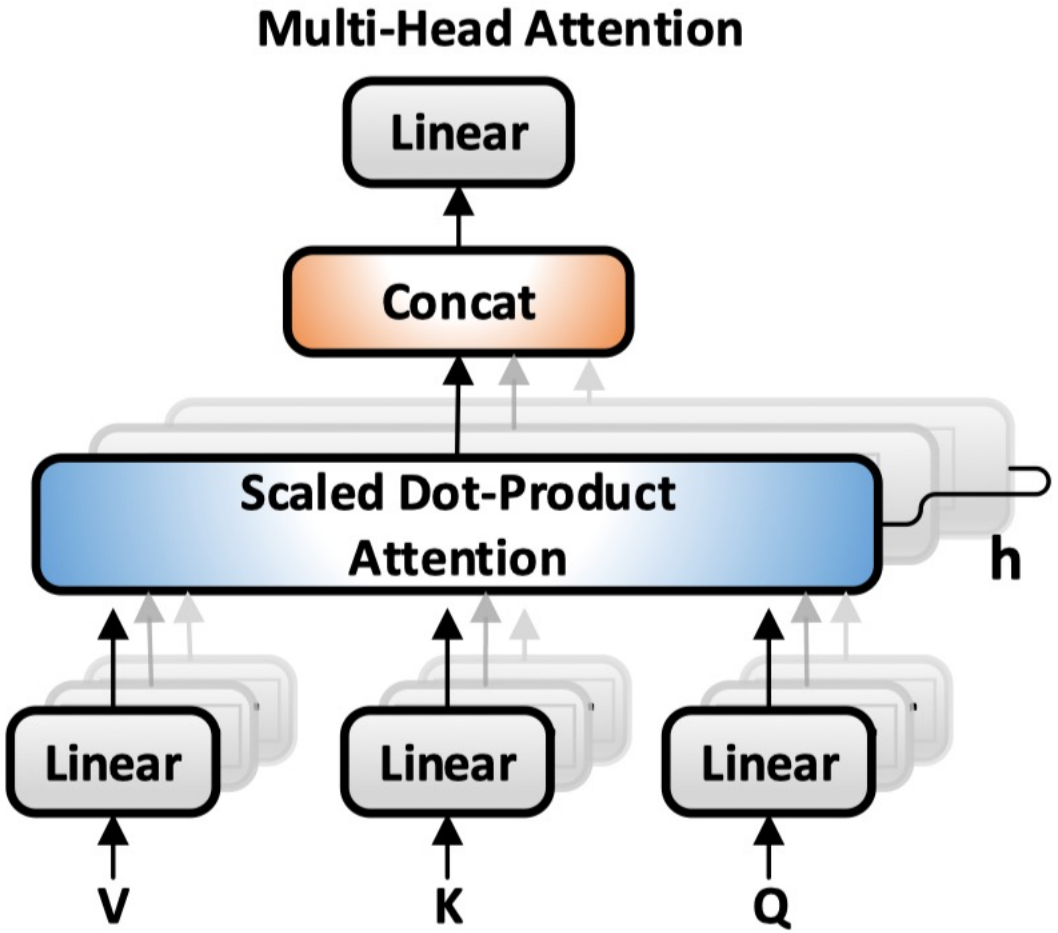
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

代码实现

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

5.4.2 模型细节

2. 多头注意力



有 h 个头，每个头 i 的参数矩阵：

$$W_Q^i \in \mathbb{R}^{d \times d_k}$$
$$W_K^i \in \mathbb{R}^{d \times d_k}$$
$$W_V^i \in \mathbb{R}^{d \times d_v}$$

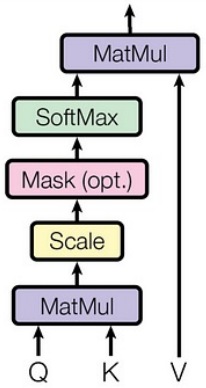
最后一层的权重矩阵$W^O \in \mathbb{R}^{hd_v \times d}$，
*多头的输出*是 $\mathbb{R}^{n \times d}$

5.4.2 模型细节

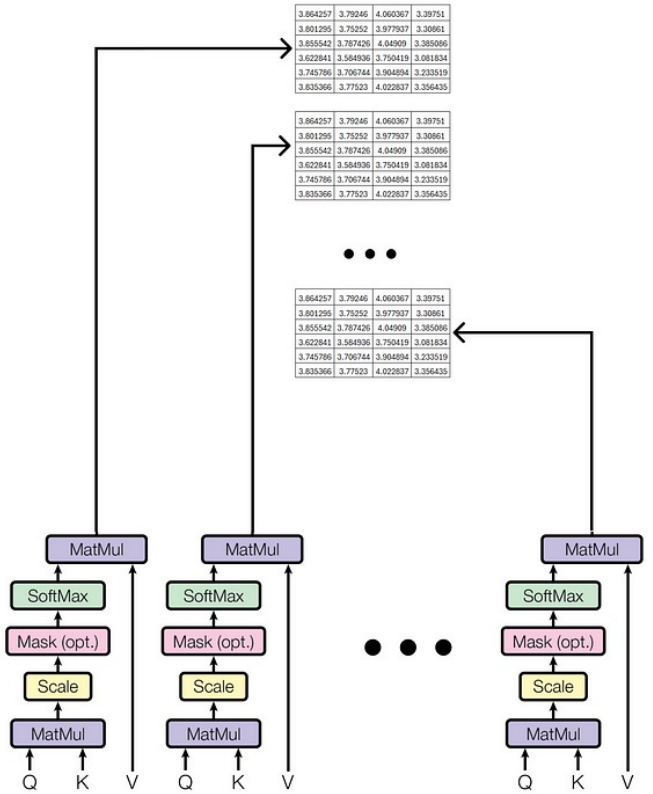
例子：多头注意力

Single Head Attention
Our Case

3.864257	3.79246	4.060367	3.39751
3.801295	3.75252	3.977937	3.30861
3.855542	3.787426	4.04909	3.385086
3.622841	3.584936	3.750419	3.081834
3.745786	3.706744	3.904894	3.233519
3.835366	3.77523	4.022837	3.356435



Multi Head Attention (N Heads)
Real world Case Concatenation



5.4.2 模型细节

例子：多头注意力

例子假设只有一个注意力头

$$QK^TV \in \mathbb{R}^{n\times d_v}$$

最后一层的权重矩阵 $W^O \in \mathbb{R}^{hd_v\times d}$，

多头的输出是 $\mathbb{R}^{n\times d}$

$\text{softmax}\left(\frac{Q.K^T}{\sqrt{d_k}}\right)$ X Value

3.86	3.79	4.06	3.4
3.8	3.75	3.98	3.31
3.86	3.79	4.05	3.39
3.62	3.58	3.75	3.08
3.75	3.71	3.9	3.23
3.84	3.78	4.02	3.36

6 x 4

X

Linear weights
columns length must be
(embedding+positional) matrix columns length

0.8	0.34	0.45	0.54	0.07	0.53
0.85	0.74	0.78	0.5	0.75	0.55
0.53	0.81	0.55	0.59	0.49	0.14
0.7	0.6	0.12	0.42	0.29	0.87

4 x 6

=

10.84	9.45	7.33	7.8	6.09	7.66
10.65	9.28	7.22	7.67	5.99	7.51
10.83	9.43	7.33	7.79	6.08	7.65
10.08	8.77	6.85	7.25	5.67	7.09
10.48	9.12	7.11	7.54	5.89	7.38
10.77	9.38	7.29	7.75	6.05	7.6

5.4.2 模型细节

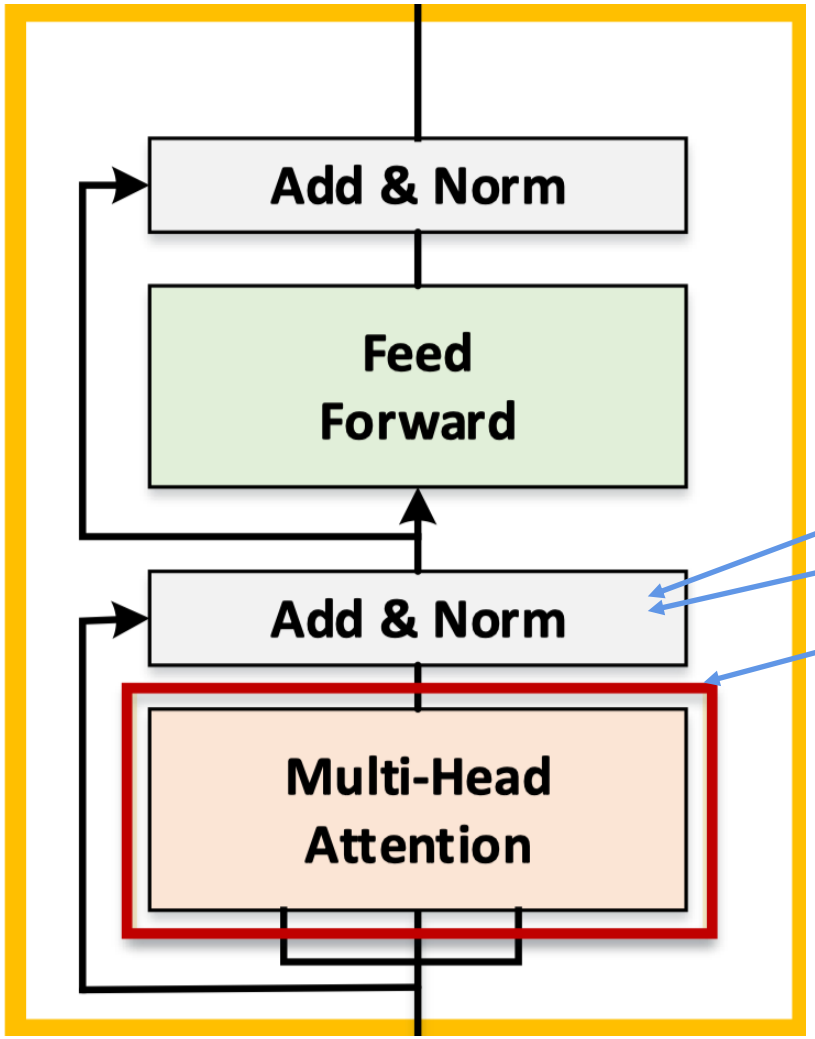
思考

这里的96x128如何理解?

*The smallest GPT-3 model (125M) has 12 attention layers, each with **12x64 dimension heads**. The largest GPT-3 model (175B) uses 96 attention layers, each with **96x128 dimension heads**.*

5.4.2 模型细节

3. 相加 & 归一化



$$\mathrm{LayerNorm}\,(X + \mathrm{MultiHeadAttention}(X))$$

相加部分是一种残差连接

$$\mathrm{F}(X) + X$$

层归一化是一种正则化技术，X_i被映射成

$$(X_i - \mu)/\sigma$$

5.4.2 模型细节

例子：相加

$$\text{LayerNorm}\left(X + \text{MultiHeadAttention}(X)\right)$$

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6

+

Output of Multi Head attention

10.84	9.45	7.33	7.8	6.09	7.66
10.65	9.28	7.22	7.67	5.99	7.51
10.83	9.43	7.33	7.79	6.08	7.65
10.08	8.77	6.85	7.25	5.67	7.09
10.48	9.12	7.11	7.54	5.89	7.38
10.77	9.38	7.29	7.75	6.05	7.6

6 x 6

=

11.63	11.05	8.29	9.44	7.06	8.86
11.87	9.45	7.28	8.46	6.89	8.25
11.75	10.94	7.6	9.1	6.64	9.24
10.34	9.51	7.51	7.47	5.74	7.46
10.6	9.71	7.91	8.16	6.39	8.08
10.41	10.68	8.05	9.23	6.99	8.81

5.4.2 模型细节

例子：层归一化

$$\text{LayerNorm}\left(X + \text{MultiHeadAttention}(X)\right)$$

$$mean = \frac{\sum_{i=1}^{N} X_i}{N} \qquad standard\ dev. = \sqrt{\frac{\sum_{i=1}^{N}(X_i - \mu)^2}{N}}$$

Row Wise Implementation

						Mean	Standard Deviation
11.63	11.05	8.29	9.44	7.06	8.86	9.26	1.57
11.87	9.45	7.28	8.46	6.89	8.25	8.56	1.64
11.75	10.94	7.6	9.1	6.64	9.24	9.04	1.76
10.34	9.51	7.51	7.47	5.74	7.46	7.86	1.51
10.6	9.71	7.91	8.16	6.39	8.08	8.37	1.35
10.41	10.68	8.05	9.23	6.99	8.81	8.93	1.28

5.4.2 模型细节

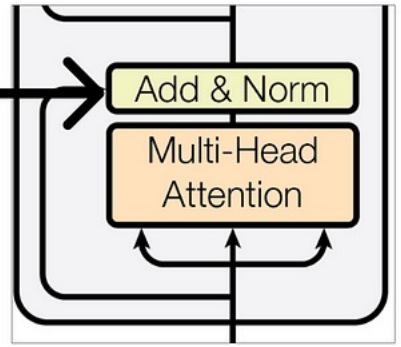
例子：层归一化

$$\text{LayerNorm}\left(X + \text{MultiHeadAttention}(X)\right)$$

11.63	11.05	8.29	9.44	7.06	8.86
11.87	9.45	7.28	8.46	6.89	8.25
11.75	10.94	7.6	9.1	6.64	9.24
10.34	9.51	7.51	7.47	5.74	7.46
10.6	9.71	7.91	8.16	6.39	8.08
10.41	10.68	8.05	9.23	6.99	8.81

Mean	Std
9.26	1.57
8.56	1.64
9.04	1.76
7.86	1.51
8.37	1.35
8.93	1.28

$$\frac{\text{value} - \text{mean}}{\text{std} + \text{error}} = \frac{11.63 - 9.26}{1.57 + 0.0001}$$



=

1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09

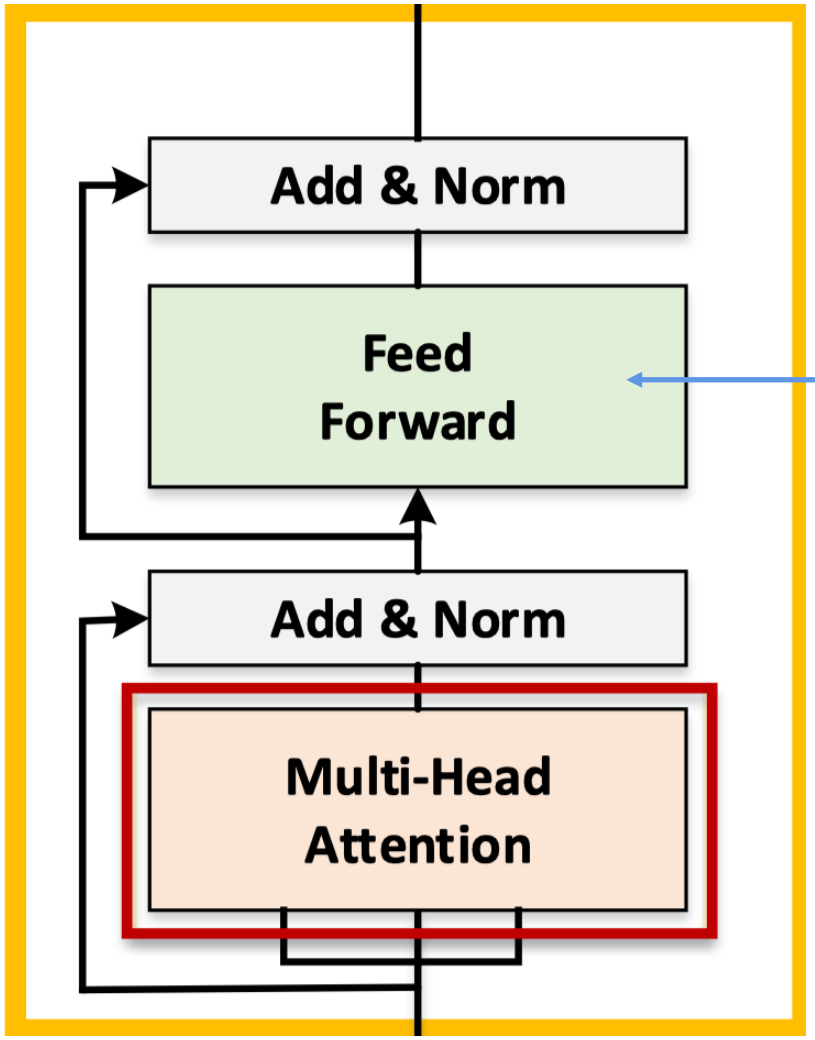
5.4.2 模型细节

代码

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."

    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

5.4.2 模型细节

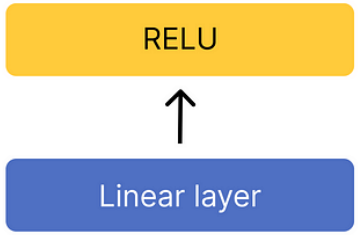
4. 前馈神经网络



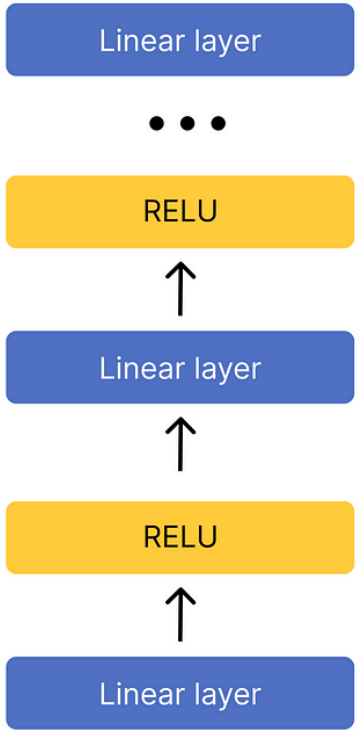
$$\mathrm{ReLU}(x) = \max(0, x)$$

$$Linear\ Layer = X \cdot W + b$$

our case (one linear layer)



Real world case (multiple layers)



5.4.2 模型细节

例子：前馈神经网络

Matrix afrer add and norm step

1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09

6 x 6

X

W

0.5	0.05	0.97	0.22	0.56	0.02
0.17	0.52	0.63	0.48	0.06	0.6
0.53	0.87	0.47	0.1	0.31	0.79
0.83	0.58	0.38	0.09	0.64	0.25
0.81	0.85	0.74	0.35	0.31	0.53
0.25	0.31	0.22	0.77	0.57	0.85

6 x 6

$X \cdot W$ ·

0.49	1.07	0.84	0.14	0.22	0.7
0.24	1.26	1.11	0.12	0.46	0.97
0.53	1.18	-0.82	0.39	0.33	0.59
0.53	0.97	0.98	0.15	0.16	0.52
0.56	1.11	-0.87	0.11	0.2	0.64
0.62	1.02	0.61	0.26	0.14	0.52

6 x 6

+

Bias

b1	b2	b3	b4	b5	b6
0.42	0.18	0.25	0.42	0.35	0.45

=

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	-0.57	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	-0.62	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

6 x 6

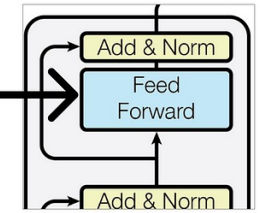
$$\mathrm{ReLU}(x) = \max(0, x)$$

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	-0.57	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	-0.62	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

6 x 6

$\max(0, 0.91)$

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	0	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	0	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97



5.4.2 模型细节

例子：再次相加+归一化

Matrix from Feed Forward Network

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	0	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	0	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

+

Matrix from Previous Add and Norm Step

1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09

=

2.42	2.39	0.47	0.67	-0.83	0.9
2.68	1.98	0.58	0.48	-0.21	1.23
2.49	2.44	-0.82	0.84	-0.68	1.15
2.59	2.24	1	0.31	-0.89	0.71
2.63	2.28	-0.34	0.37	-0.92	0.88
2.2	2.57	0.17	0.91	-1.03	0.88

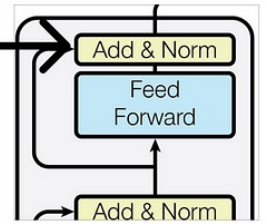
→

Mean	Std
1.0033	1.103534
1.1233	1.214349
0.9033	1.301837
0.9933	1.289055
0.8167	1.306016
0.95	1.320773

=

1.28	1.26	-0.48	-0.3	-1.66	-0.09
1.28	0.71	-0.45	-0.53	-1.1	0.09
1.22	1.18	-1.32	-0.05	-1.22	0.19
1.24	0.97	0.01	-0.53	-1.46	-0.22
1.39	1.12	-0.89	-0.34	-1.33	0.05
0.95	1.23	-0.59	-0.03	-1.5	-0.05

→



5.4.2 模型细节

示例代码

```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """

    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

5.4.2 模型细节

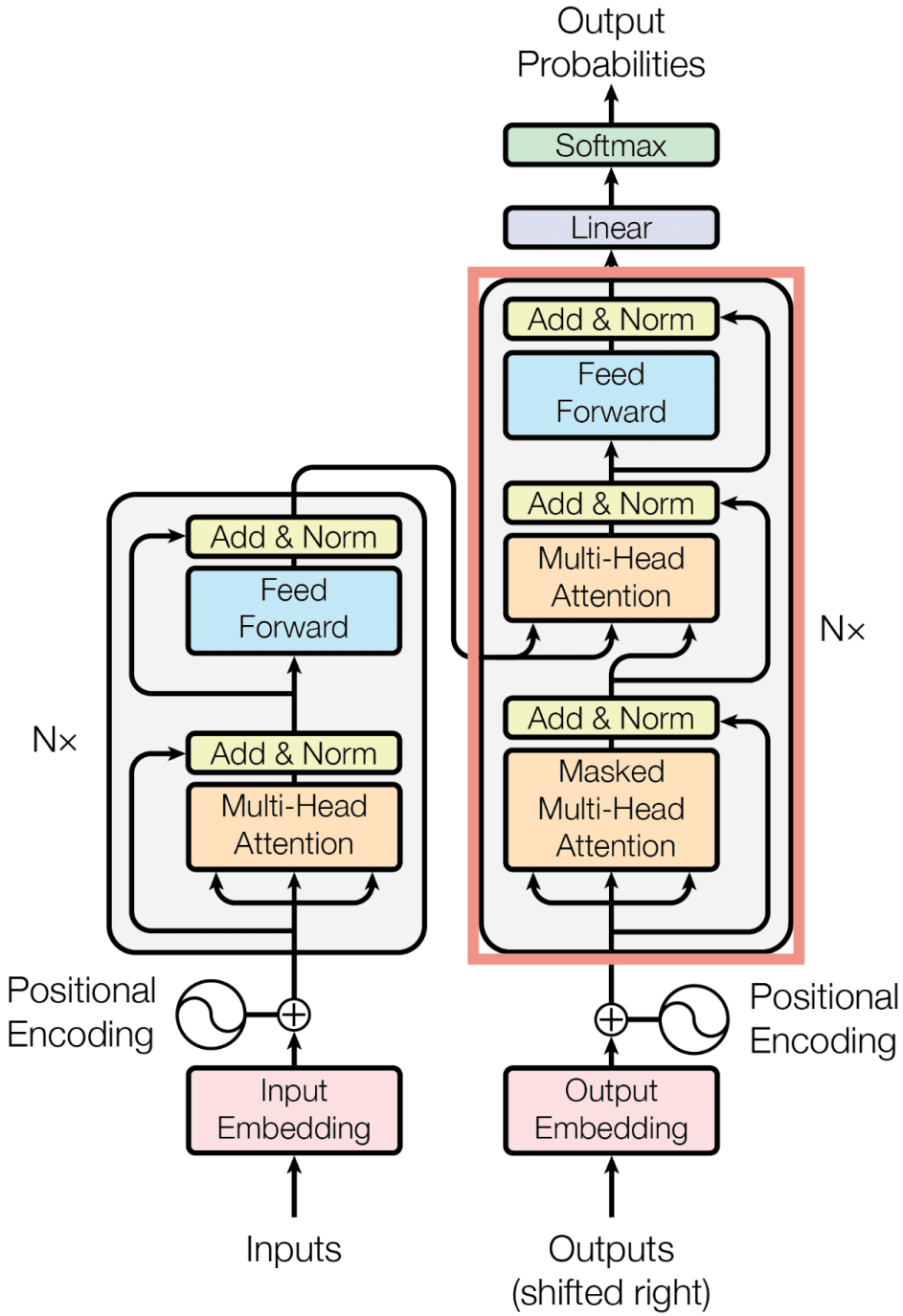
示例代码

```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"

    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

5.4.2 模型细节

5. 解码

编码器模块的输出（最后一次 add&norm）是$\mathbb{R}^{n\times d}$，被用于解码器模块中第二个多头注意力的的 K 和 V；而此时的 Q 来自解码器模块中第一个（掩码）多头注意力的输出。



5.4.2 模型细节

5. 解码：掩码多头注意力机制

而对应生成任务，它应该采用**因果自注意力**（**Causal Self-Attention**），即只能关注当前位置及其之前的词元，这样保证在训练时不能“看到”未来的信息。

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{Q}};\quad \mathbf{k}_j = \mathbf{x}_j \mathbf{W}^{\mathbf{K}};\quad \mathbf{v}_j = \mathbf{x}_j \mathbf{W}^{\mathbf{V}}$$

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \quad \forall j \le i$$

$$\mathbf{a}_i = \sum_{j \le i} \alpha_{ij} \mathbf{v}_j$$

5.4.2 模型细节

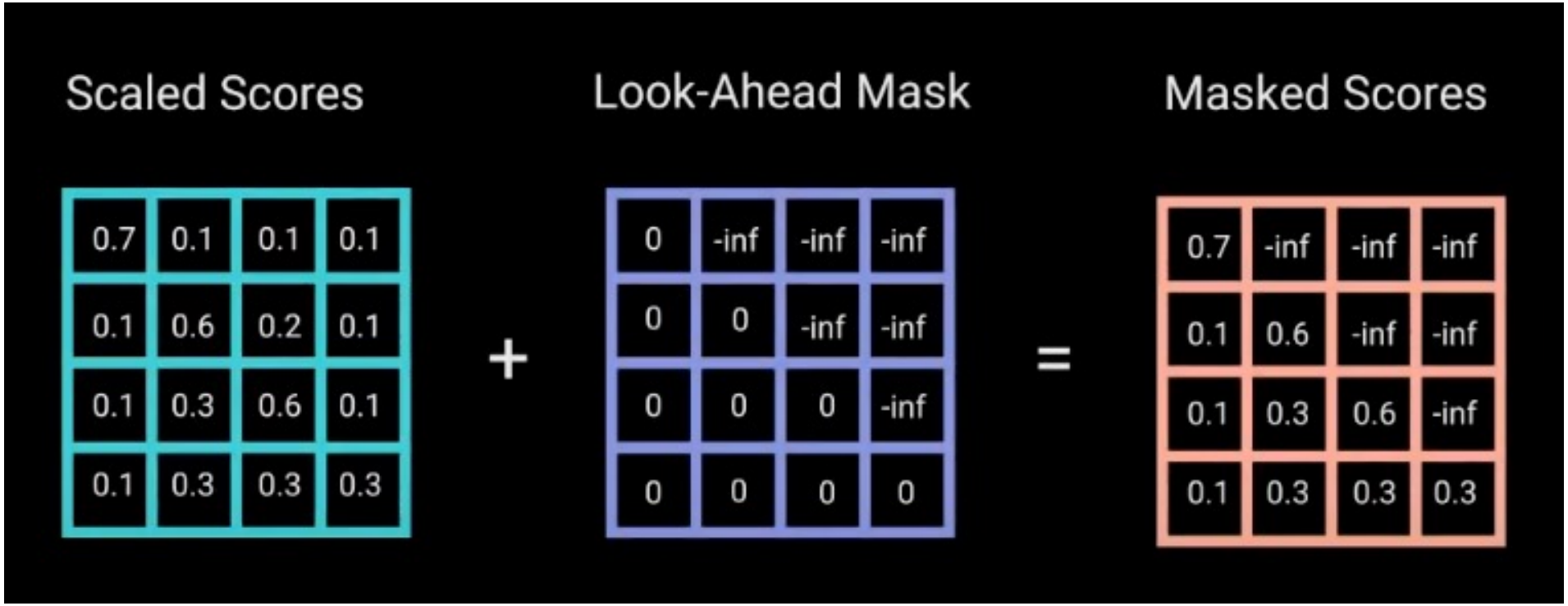
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

5. 解码：掩码多头注意力机制

在Transformer中，因果注意力机制通过**掩码**实现，在 Softmax 之前进行。

$QK^T \in \mathbb{R}^{n \times n}$

上三角是负无穷大



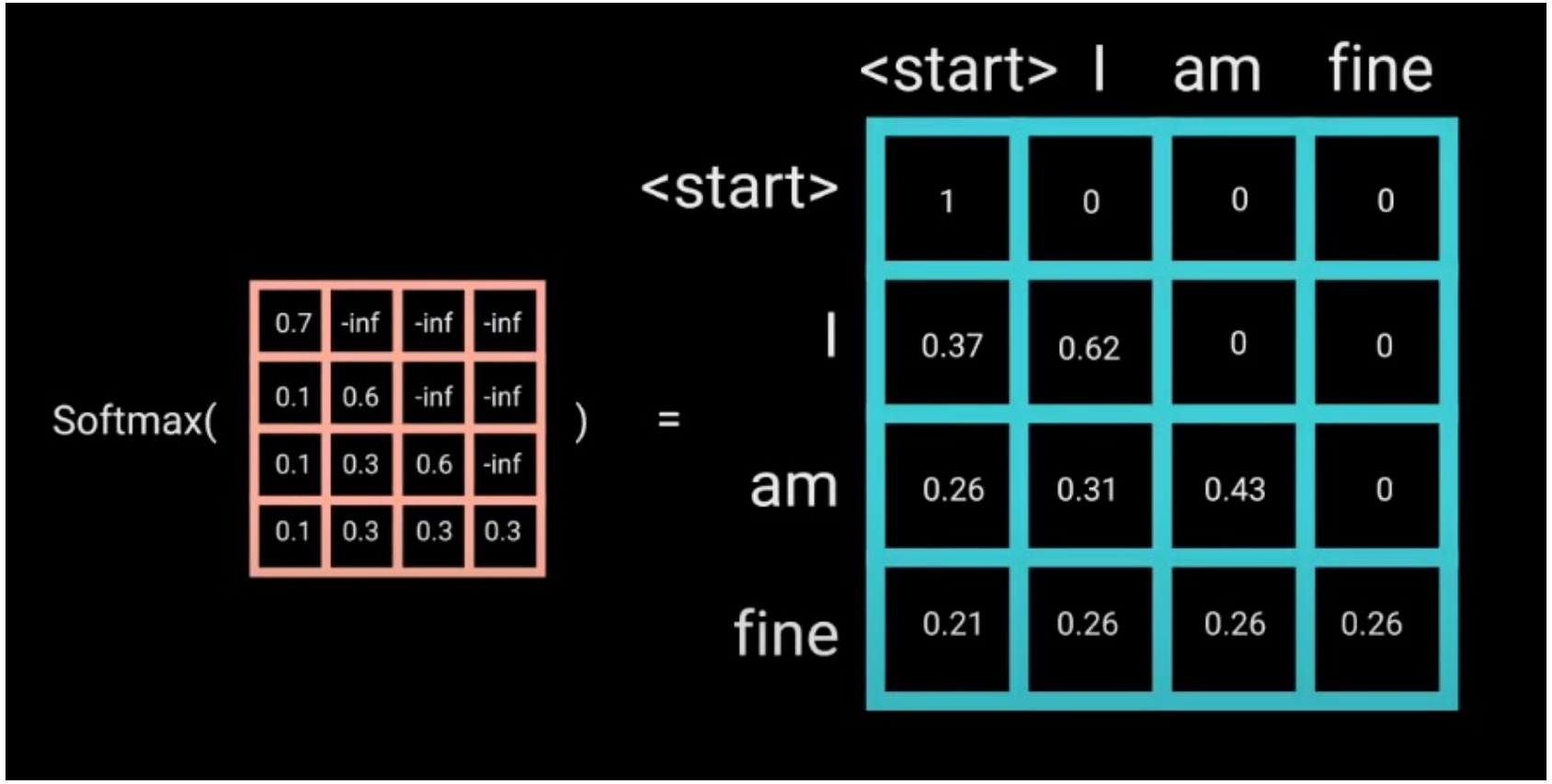
图片来源：https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0

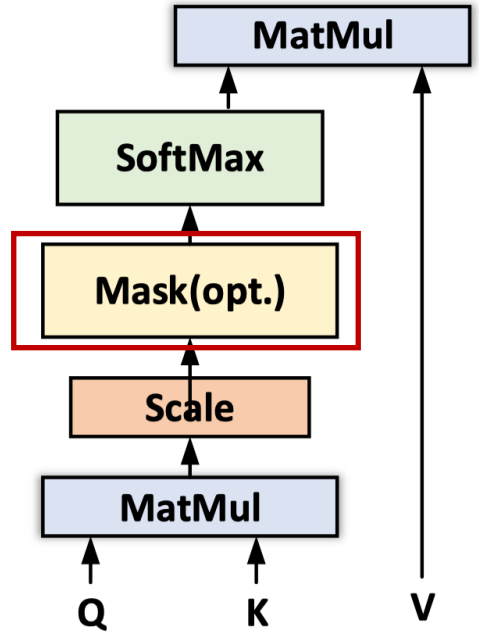
5.4.2 模型细节

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

5. 解码：掩码多头注意力机制

在Transformer中，因果注意力机制通过**掩码**实现，在 Softmax 之前进行。





图片来源：https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0

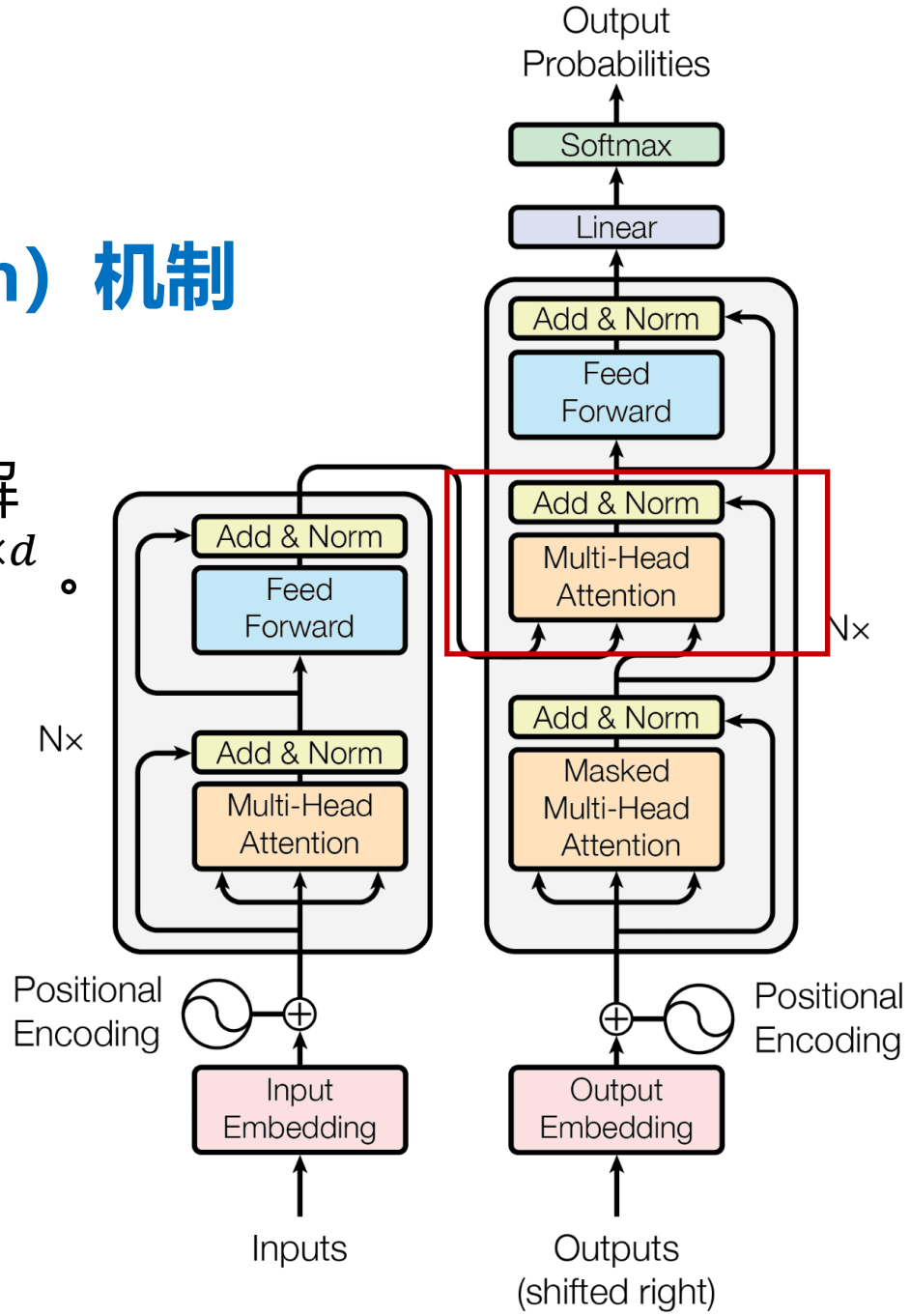
5.4.2 模型细节

5. 解码：交叉注意力（Cross-Attention）机制

编码器模块的输出（最后一次 add&norm）是 $\mathbb{R}^{n\times d}$。如果训练时，目标序列长度是 m，那么解码器模块的第一个多头注意力机制的输出是 $\mathbb{R}^{m\times d}$。

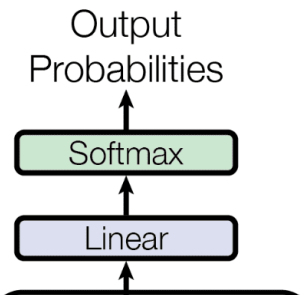
$$\text{Attention}(Q,K,V)=\text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

最终的输出还是$\mathbb{R}^{m\times d}$。

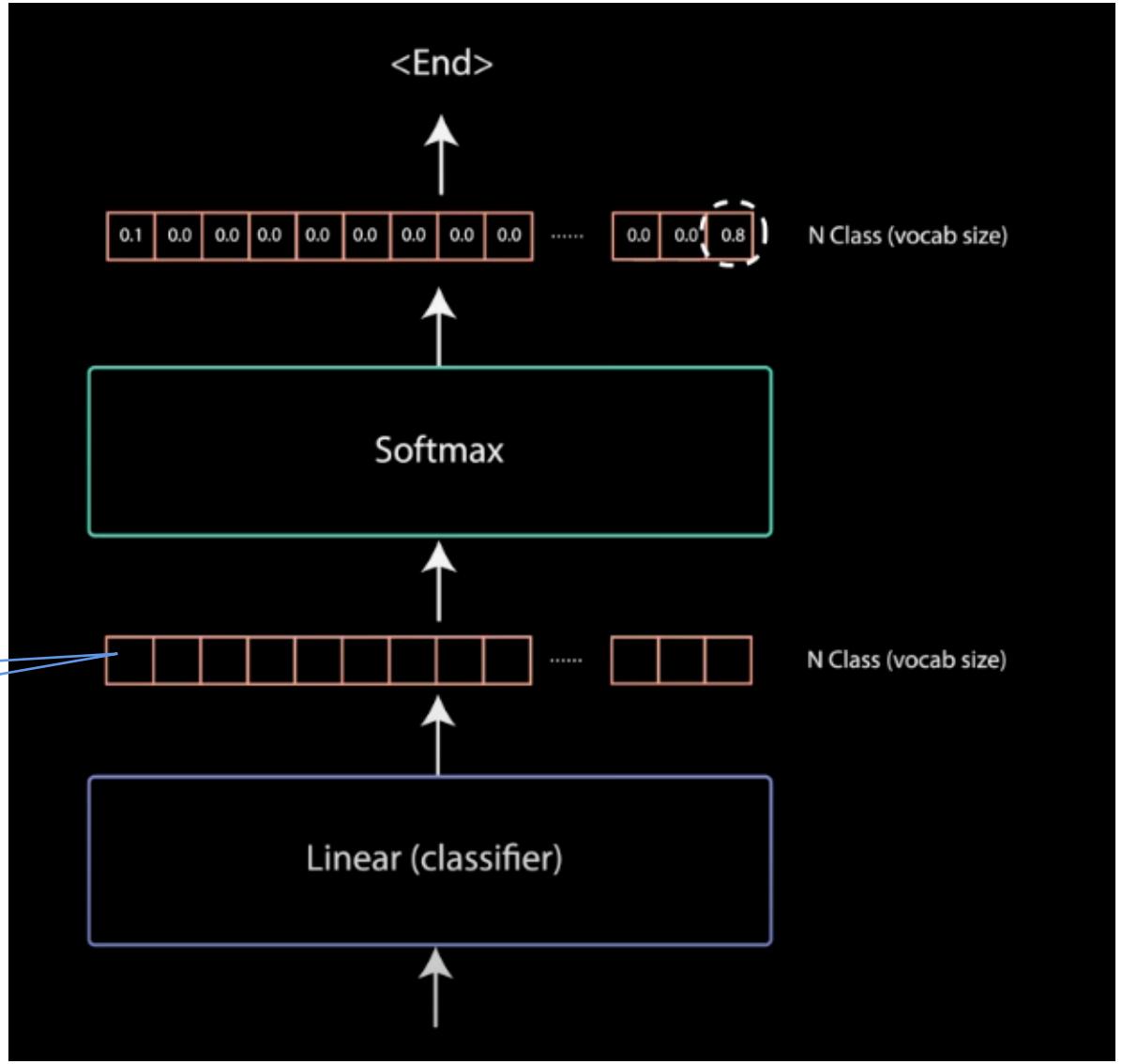


5.4.2 模型细节

5. 解码：预测



被称为**logits**（或得分）



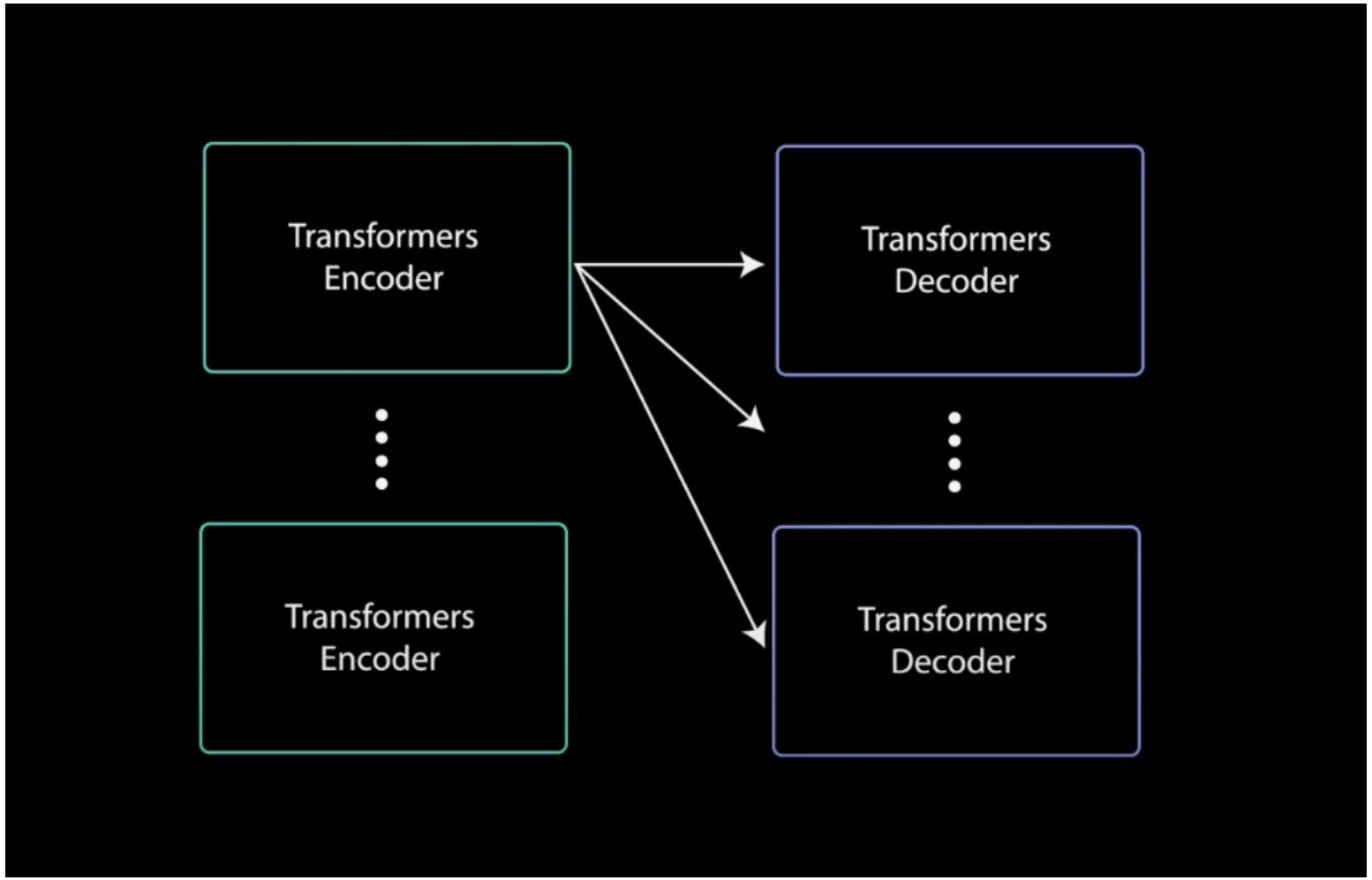
思考

解码器在训练和推理时区别?

图片来源：https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0

5.4.2 模型细节

6. 回顾整体结构



GPT-3堆叠了96层Decoder

图片来源：https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0

5.4.2 模型细节

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

7. 计算复杂度与并行

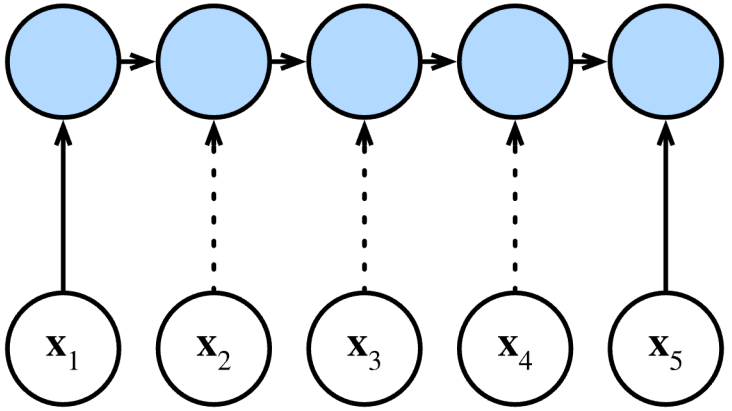
考虑输入和输出序列长度都是 *n*，维度是 *d*。

对于RNN结构，其隐藏层权重是$\mathbb{R}^{d\times d}$，*n* 个时间步的总复杂度是$O(nd^2)$；无法并行!

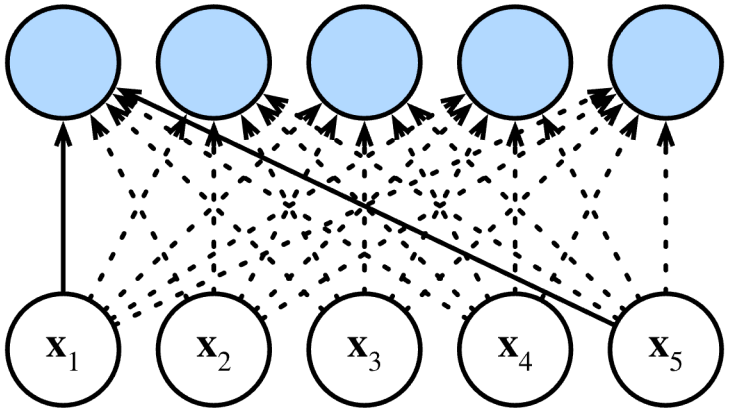
对于自注意力结构，Q、K和V都是$\mathbb{R}^{n\times d}$，总复杂度是$O(n^2d)$；能够并行!

不适合长序列

RNN



Self-attention



目录

5.4 Transformer 模型

5.4.1 模型整体结构

5.4.2 模型细节

5.5 预训练语言模型

5.5.1 BERT 模型

5.5.2 GPT-1 模型

5.6 语言模型使用范式

5.6.1 预训练-传统微调范式

5.6.2 大模型-提示工程范式

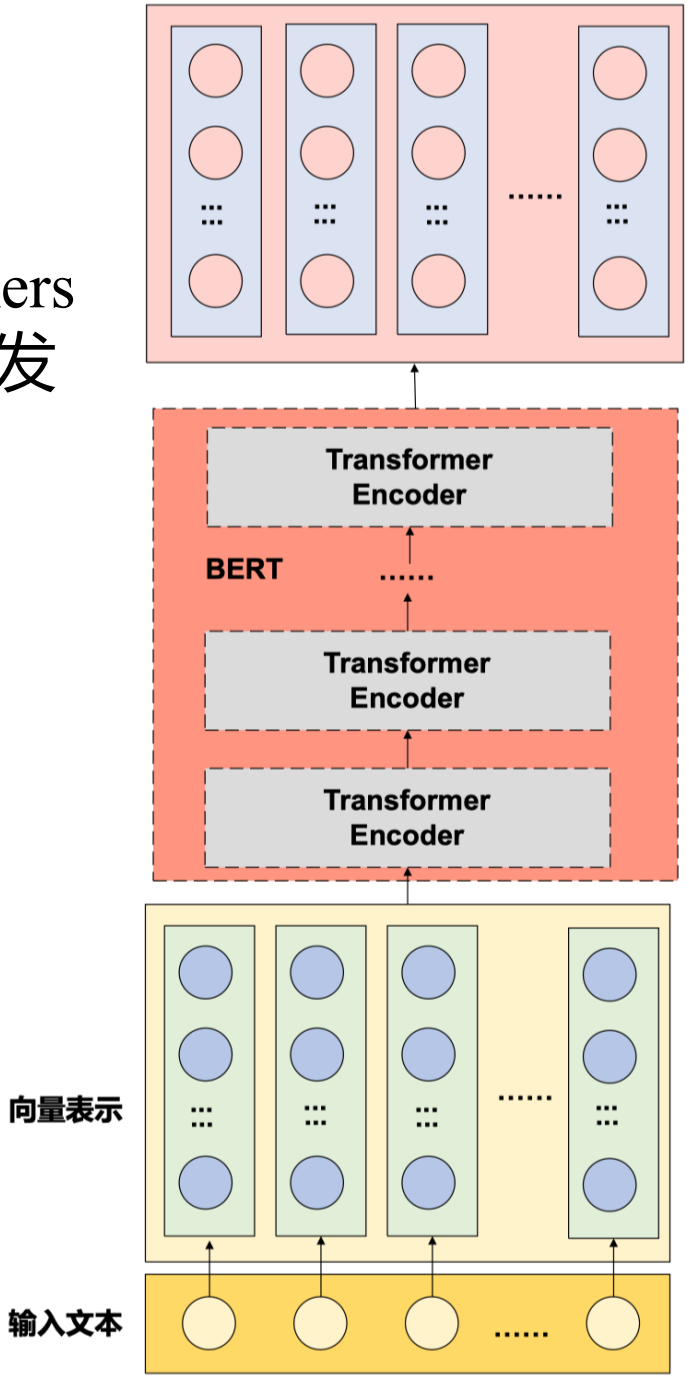
5.5.1 BERT模型

BERT（Bidirectional Encoder Representations from Transformers）由 Google 在 2018 年提出并基于 Transformer 架构进行开发的预训练语言模型。

理解BERT的两个关键词：

- 何为**Encoder only**?
- 何为**Bidirectional**?

Base 版本包含 12 层 Transformer 编码器，而 Large 版本包含 24 层 Transformer 编码器，其参数总数分别为 110M 和 340M。



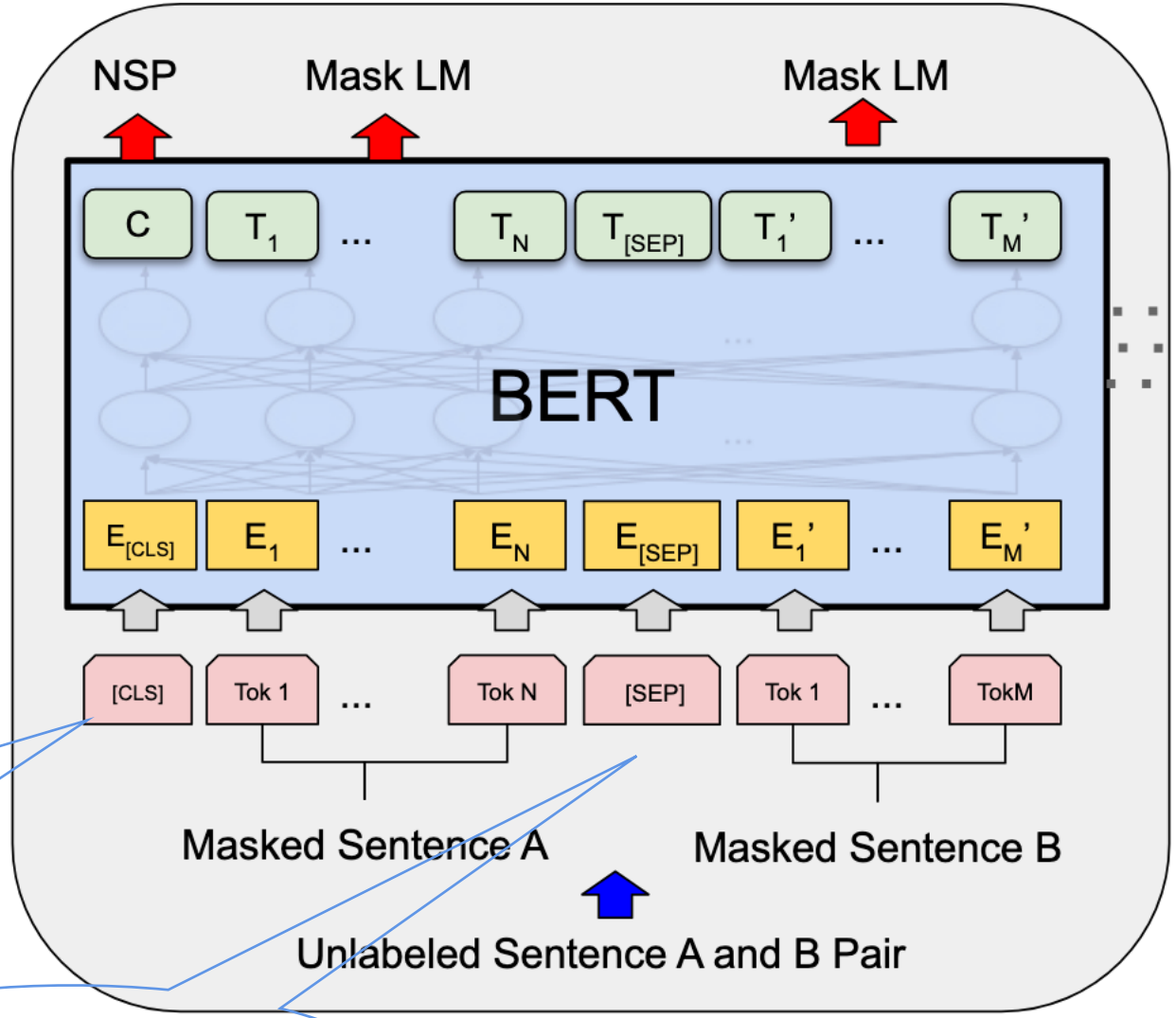
谷歌发布的AI模型名称叫“Bert”，这个名字来源于美国著名儿童节目《芝麻街》（Sesame Street）中的角色。 剧中和 Bert 住在一起的好友就叫 Ernie。由此，2019年百度推出自己的AI模型，取名为“ Ernie”

5.5.1 BERT模型

1. 预训练

BERT设计了两个预训练任务：

- 掩码语言模型（Marked Language Model，**MLM**）任务
- 下一句预测（Next Sentence Predication，**NSP**）任务



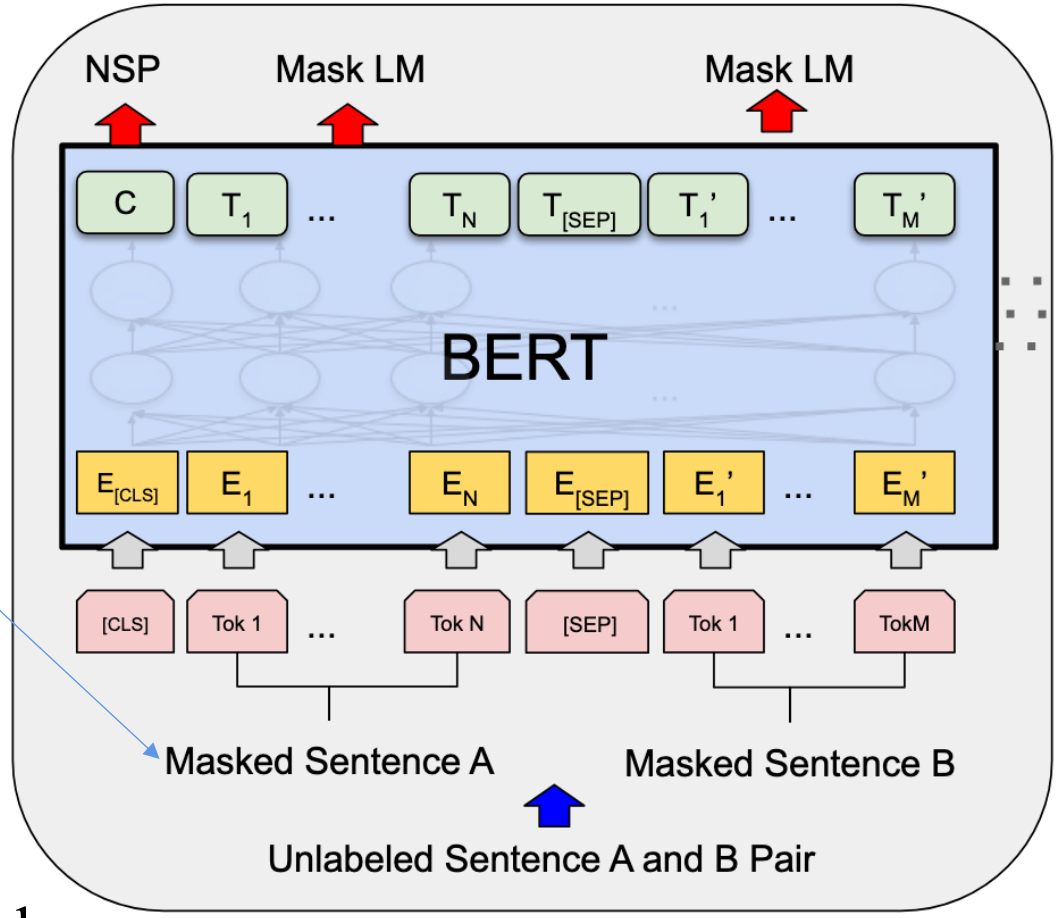
5.5.1 BERT模型

1. 预训练：MLM

输入的词元中有 k% （原文中15%）被遮挡，然后预测被遮住是什么。

- 使用特殊的[MASK]代替
- 使用随机词元代替

举头望[ ]月 低[ ]思故乡



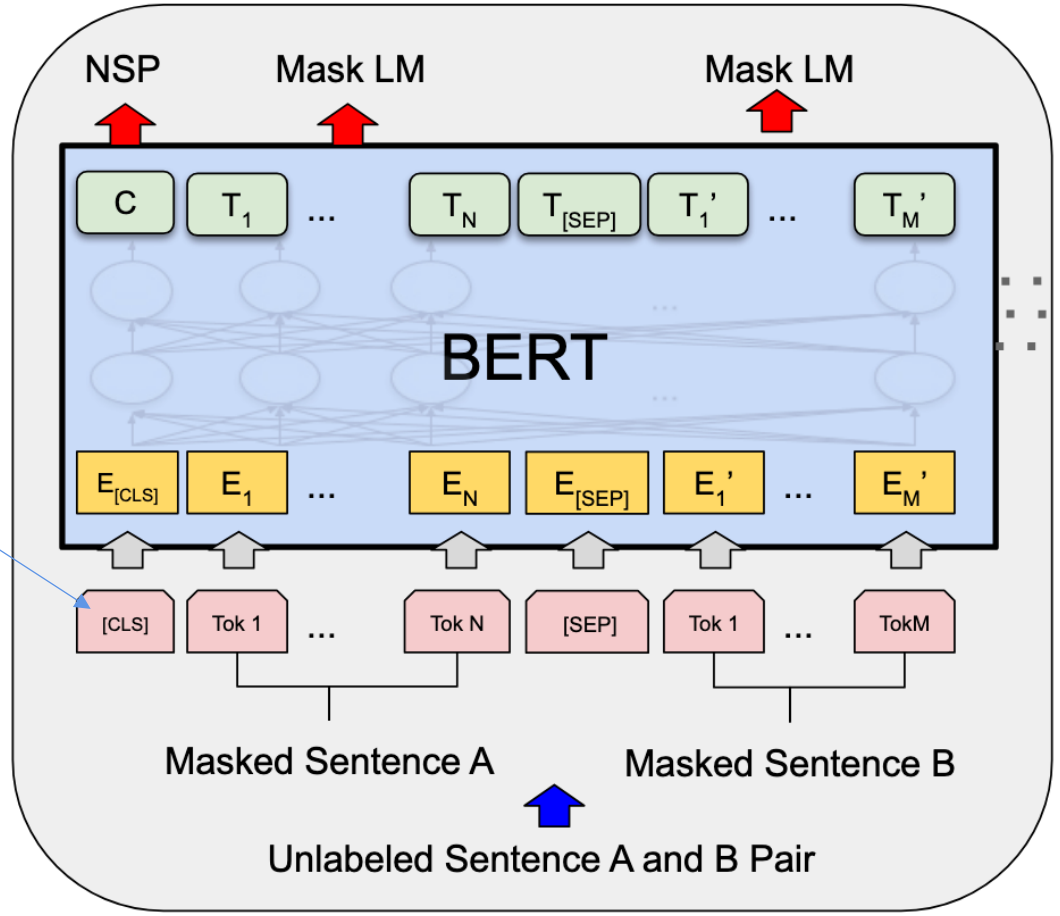
这是无监督学习，但是从参数优化角度来讲，被Mask的就是训练的标签。

5.5.1 BERT模型

1. 预训练：NSP

预测一个句子是否会紧挨着另一个出现。
在许多下游任务中，例如问答和自然语言推理，两个句子之间的关联至关重要。

训练数据的生成方式是从语料库中随机抽取的连续两句话，其中 50% 保留抽取的两句话，它们符合 **IsNext** 关系，另外 50% 的第二句话是随机从预料中提取的，它们的关系是 **NotNext** 的。

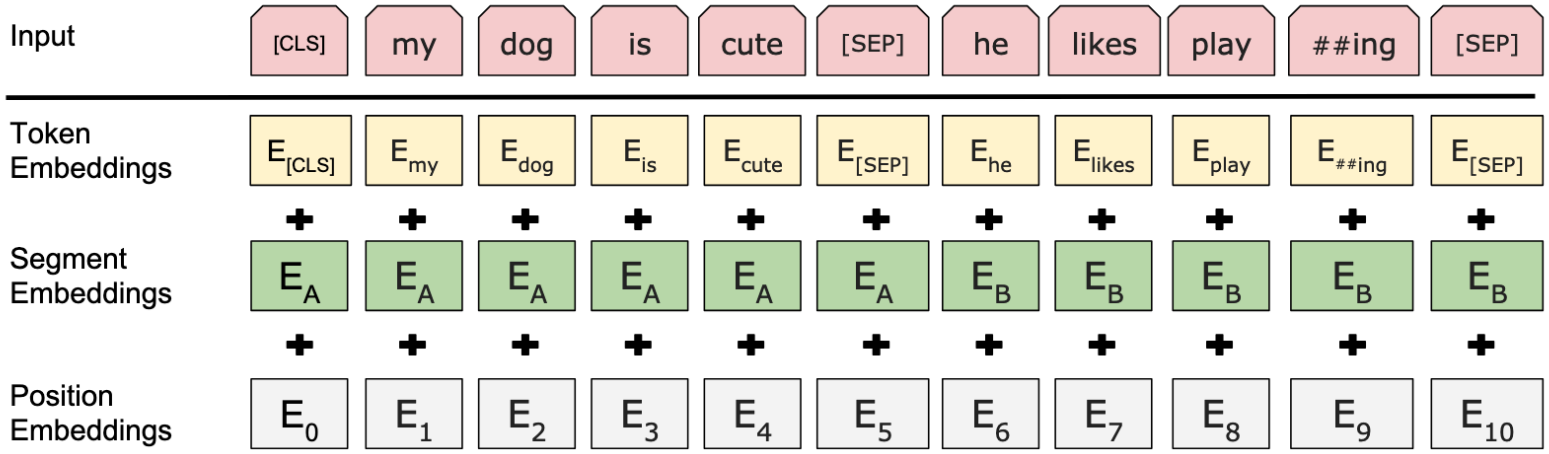


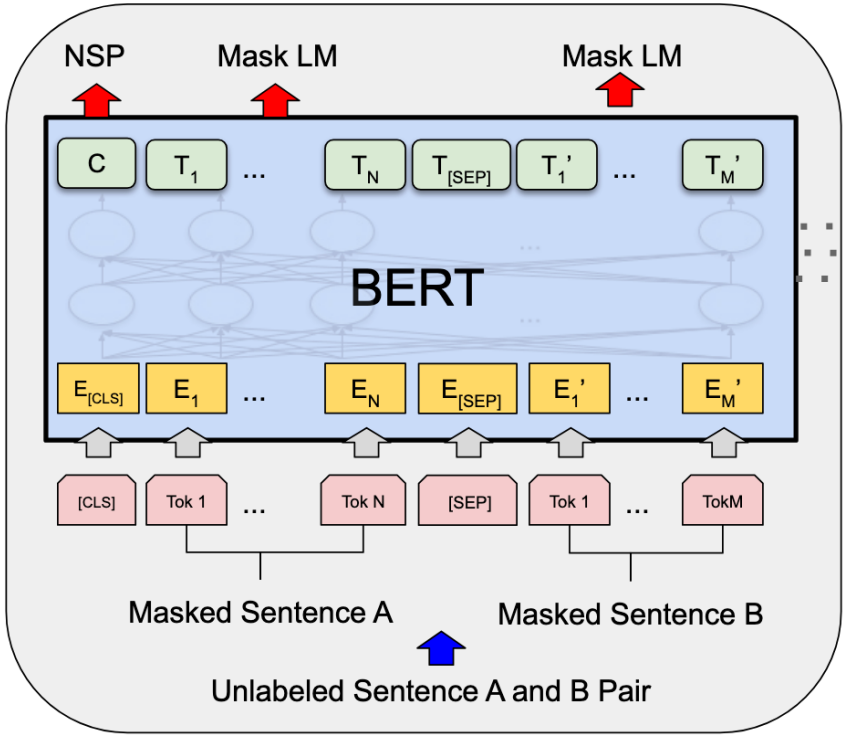
5.5.1 BERT模型

1. 预训练：NSP

思考

在输入时如何区分两个句子?

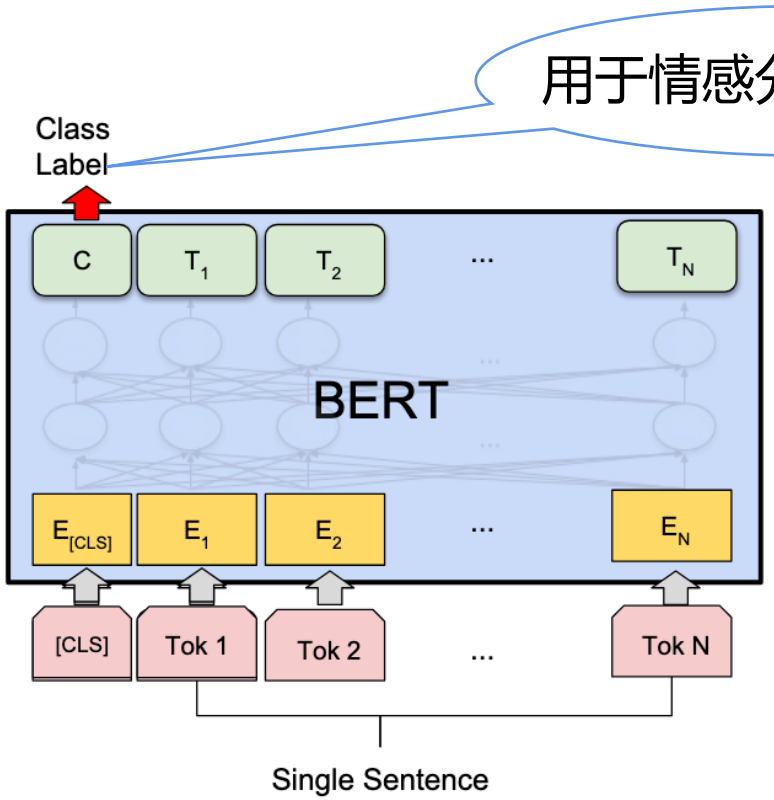




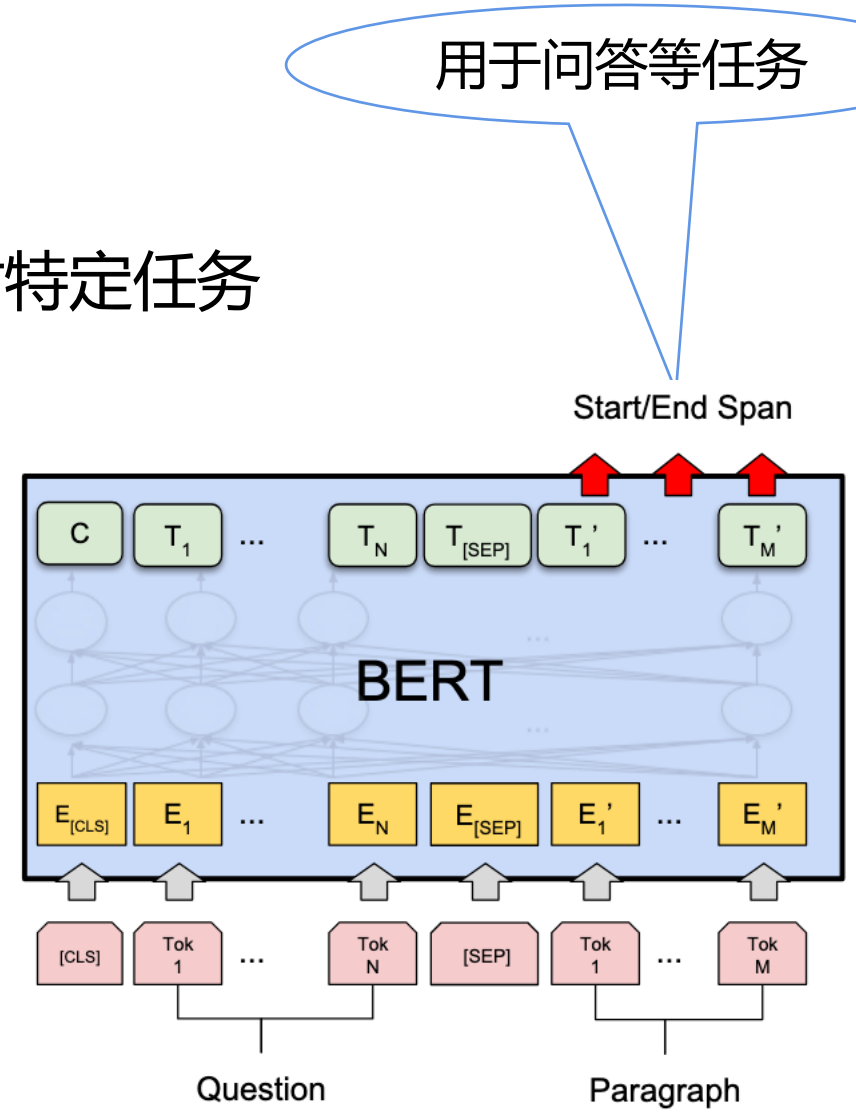
5.5.1 BERT模型

2. 微调

微调训练阶段是在预训练的 BERT 模型基础上，针对特定任务进行的训练。这一阶段使用具有标签的数据。



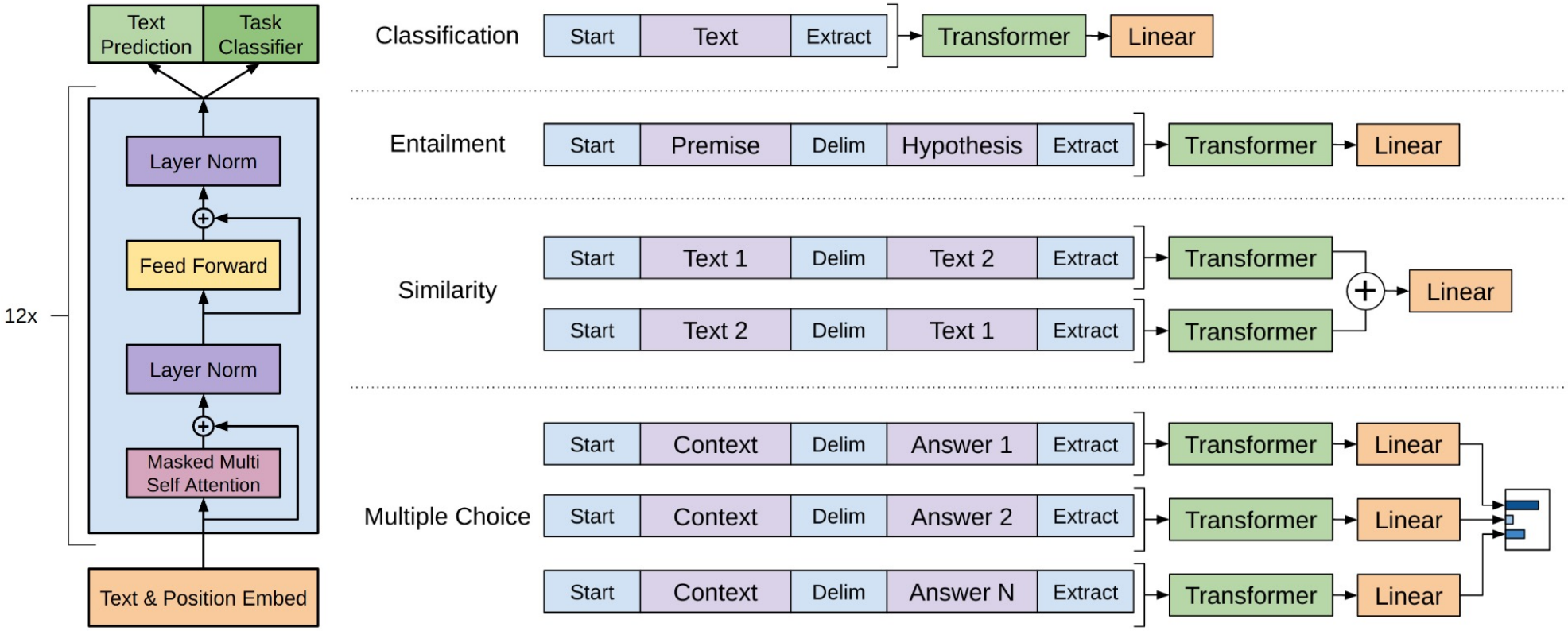
对于不同任务，需要在模型顶部添加一个输出层，这被称为 **language model head**。



5.5.2 GPT模型

1. Generative Pre-Training

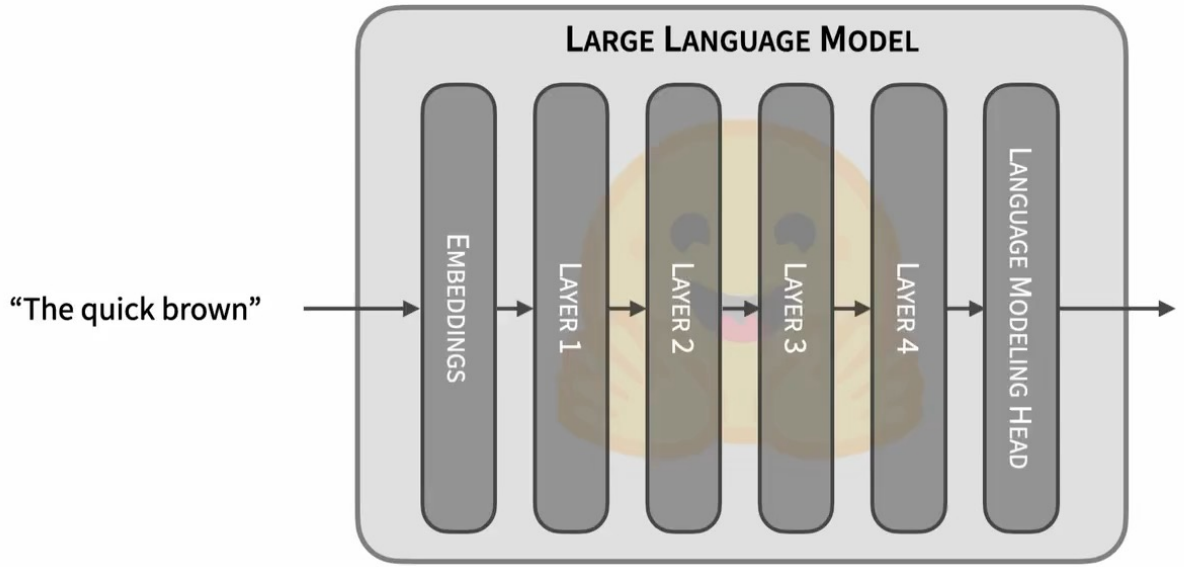
GPT-1 模型由 OpenAI 在2018年研发并首次提出，基于 12 层的 Transformer 解码器架构构建而成（**decoder-only**）。



5.5.2 GPT模型

2. GPT：自回归

自回归（Auto-regressive）是NLP的一种训练方法，其核心思想是基于已有序列来预测下一个。



动画来源：https://huggingface.co/docs/transformers/en/llm_tutorial

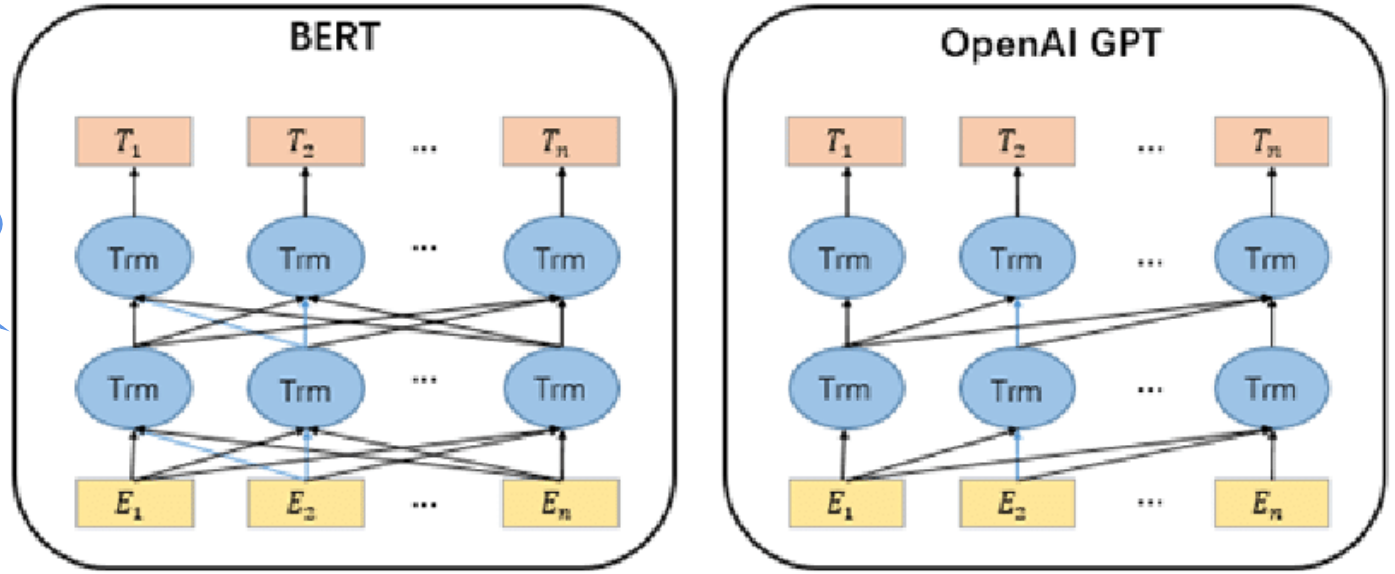
5.5.2 GPT模型

3. GPT vs BERT

GPT的出现比BERT稍早。

- GPT是单向语境（从左到右）；而BERT考虑了上下文的信息。
- GPT的主要任务是给定上下文预测下一个词，，即**因果语言模型**（Causal Language Modeling）；而BERT的掩码语言模型类似填空。

双向



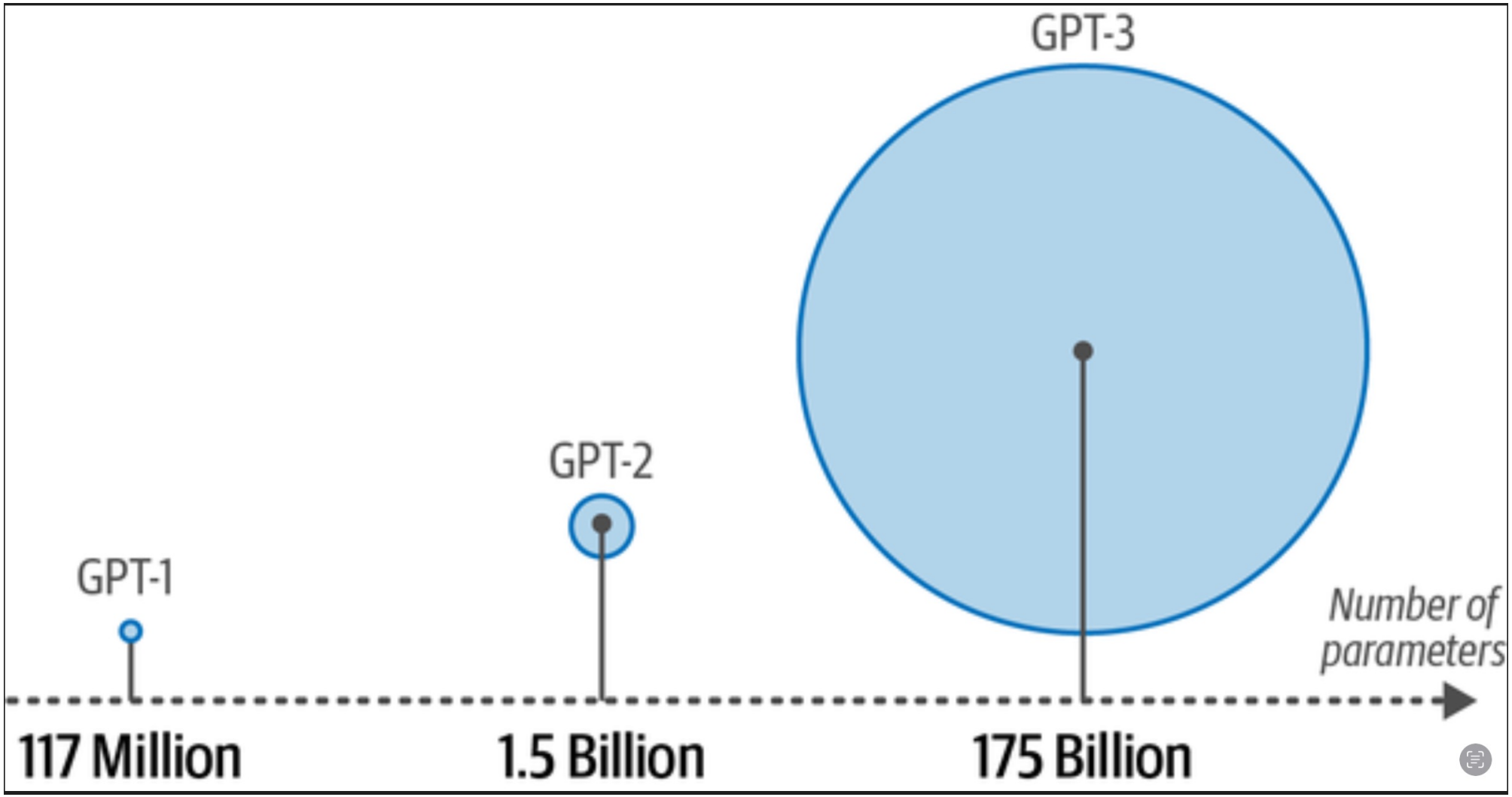
单向

5.5.2 GPT模型

3. GPT vs BERT

BERT	GPT
双向（关注上文和下文）	单向（仅关注上文）
仅编码器	仅解码器
掩码语言模型（自编码）	因果语言模型（自回归）
面向理解任务（如文本分类、命名实体识别）	面向生成任务（如文本生成）

5.5.2 GPT模型


GPT-3
GPT-2
GPT-1
Number of parameters
117 Million
1.5 Billion
175 Billion

目录

5.4 Transformer 模型

5.4.1 模型整体结构

5.4.2 模型细节

5.5 预训练语言模型

5.5.1 BERT 模型

5.5.2 GPT-1 模型

5.6 语言模型使用范式

5.6.1 预训练-传统微调范式

5.6.2 大模型-提示工程范式

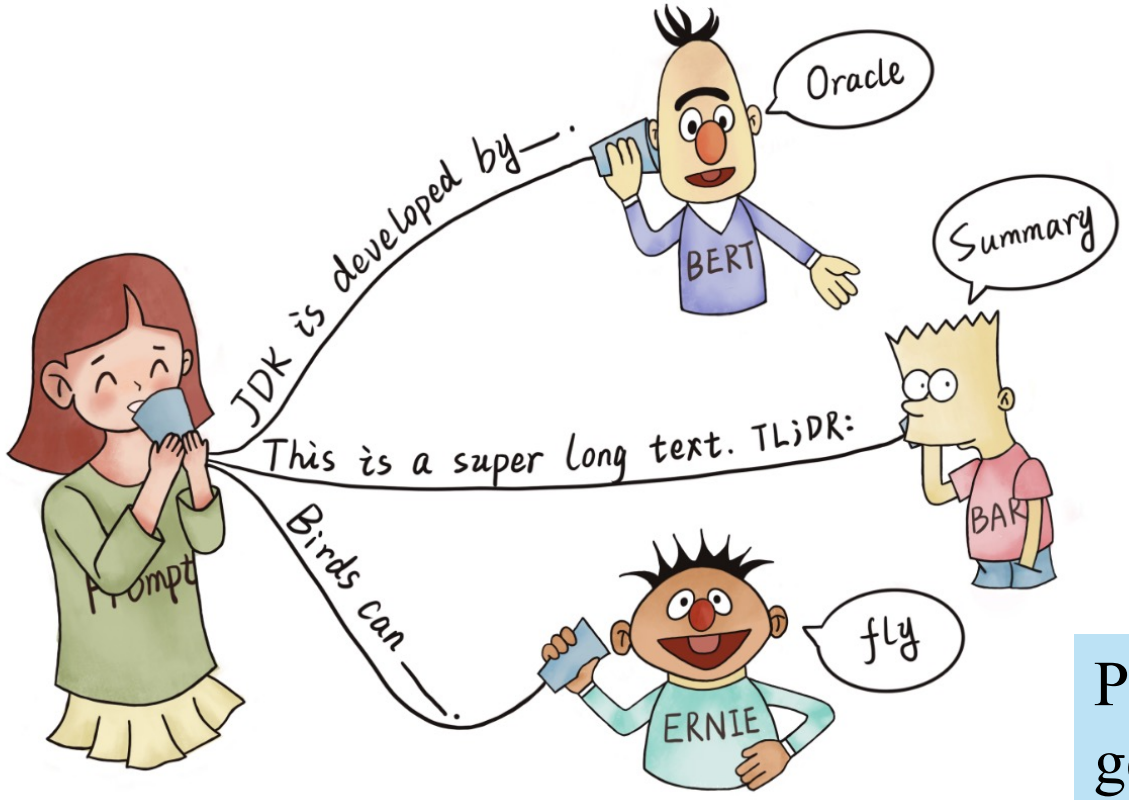
5.6.1 预训练-传统微调范式

（b）基础模型的“预训练-微调”范式

在预训练-微调的范式中，可以使用预训练好的模型（如 BERT 或 GPT1），并在特定任务上进行微调，以适应任务的特殊需求。这种方法即使在有限的标注数据的情况下，也能够从大规模无监督数据的预训练中获益。

图片来源：https://www.cs.tsinghua.edu.cn/info/1088/5295.htm

5.6.2 大模型-提示工程范式



Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing

这个范式的优势在于，大型预训练模型已经学到了丰富的语言表示，因此 具备强大的生成能力，能够适应多种任务。通过调整输入**提示**，可以引导模型执行各种任务，而不需要为每个任务单独训练一个模型。

Prompt engineering is the process of iterating a generative AI prompt to improve its accuracy and effectiveness.

图片来源：https://arxiv.org/pdf/2107.13586

5.7讨论

- **讨论5.1：**

  请探讨位置编码在Transformer 模型中的作用与必要性，如何通过位置信息增强模型的序列处理能力？。

- **讨论5.2：**

  请比较 Transformer 模型中的多头自注意力机制与多头编码器-解码器注意力机制的应用和效果，并解释它们如何分别影响信息处理和任务性能。