

# 通用大模型原理及训练实践

## 第2讲：预备知识

冯洋



# 本章大纲

---

## ■ 神经网络

- 前向神经网络

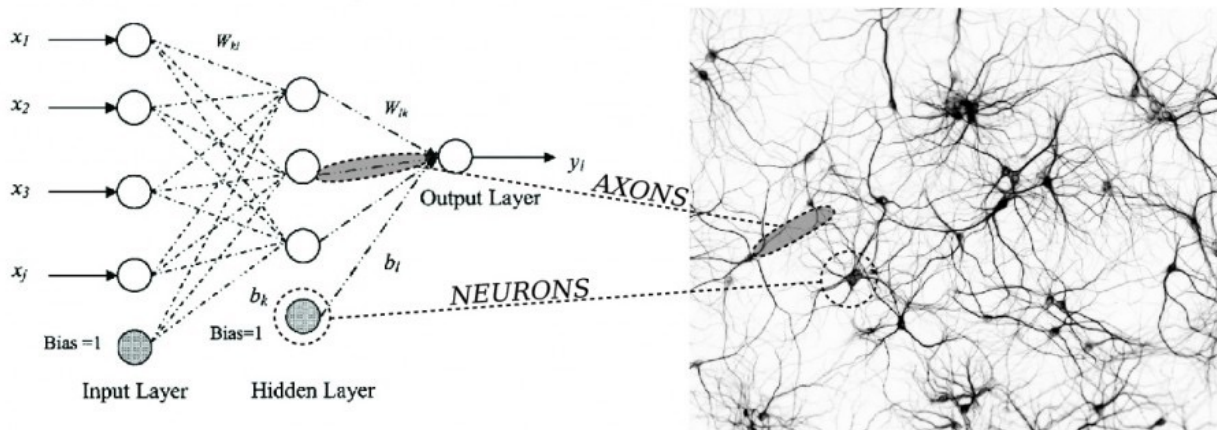
- 自注意力神经网络

## ■ Transformer

# 神经网络：人工 Vs. 生物

- 神经网络是由具有适应性的简单单元组成的广泛并行互联的网络，它的组织能够模拟生物神经系统对真实世界物体所作出的交互反应

## NEURAL NETWORK MAPPING

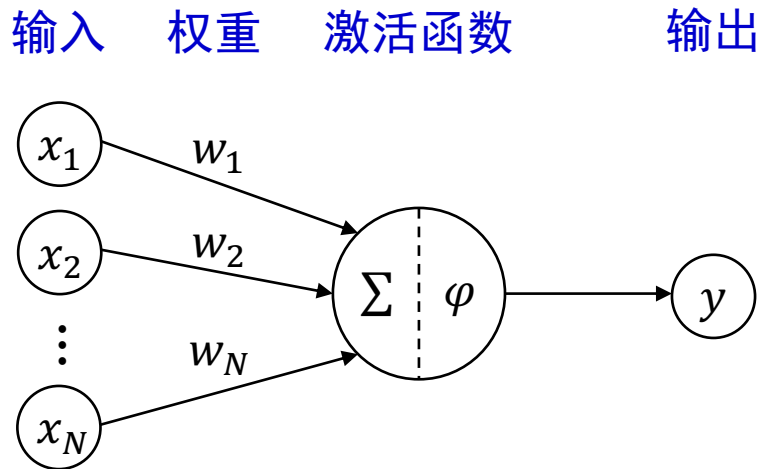


<https://rpi-cloudreassembly.transvercity.net/2012/11/04/neural-network-mapping-analysis-from-above/>

# 神经元模型

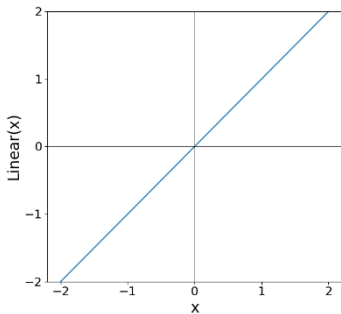
- 神经网络中最基本的成分是**神经元模型**

$$y = \varphi\left(\sum_{n=1}^N w_n \times x_n\right)$$

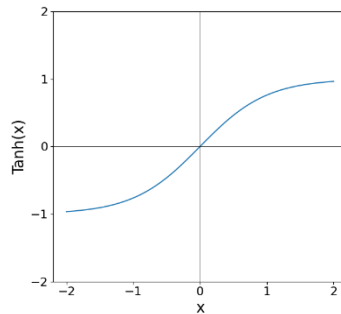


# 激活函数

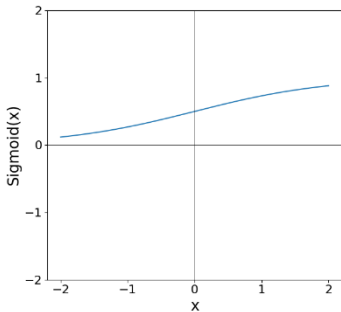
- 线性函数:  $\varphi(x) = x$



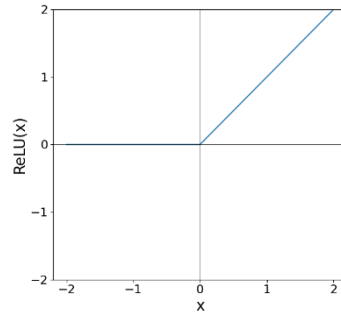
- tanh函数:  $\varphi(x) = \tanh(x)$



- Sigmoid函数:  $\varphi(x) = \frac{1}{1+\exp(-x)}$

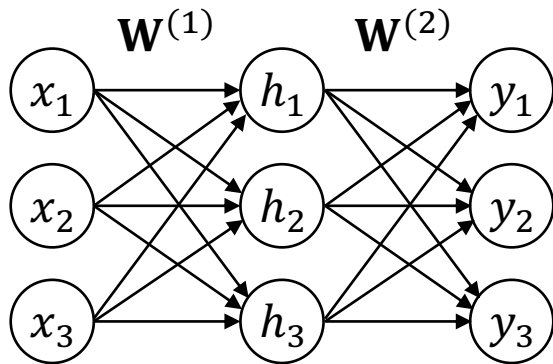


- ReLU函数:  $\varphi(x) = \max(0, x)$



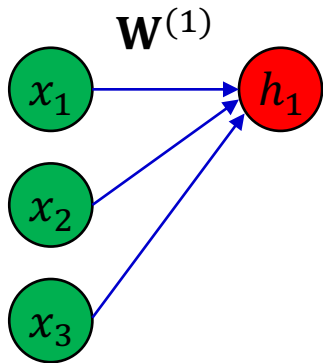
# 前向神经网络

- 前向神经网络本质上是一个多元复合函数
- 网络接受输入节点  $x_i \in \mathbb{R}$  ( $i = 1, 2, 3$ ), 经过隐藏节点  $h_j \in \mathbb{R}$  ( $j = 1, 2, 3$ ), 最后得到输出节点  $y_k \in \mathbb{R}$  ( $k = 1, 2, 3$ )
- 输入节点和隐藏节点之间的权重矩阵表示为  $\mathbf{W}^{(1)} \in \mathbb{R}^{3 \times 3}$ , 其中  $W_{ij}^{(1)}$  表示连接  $x_i$  和  $h_j$  的边的权重; 类似地, 隐藏节点和输出节点之间的权重矩阵表示为  $\mathbf{W}^{(2)} \in \mathbb{R}^{3 \times 3}$



# 前向神经网络计算过程

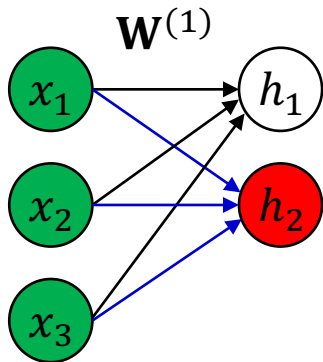
- 前向神经网络本质上是一个多元复合函数



$$h_1 = \varphi\left(\sum_{i=1}^3 W_{i,1}^{(1)} \times x_i\right)$$

# 前向神经网络计算过程

- 前向神经网络本质上是一个多元复合函数



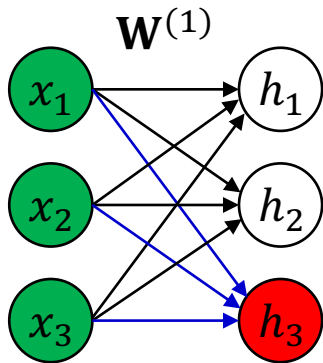
$$h_1 = \varphi\left(\sum_{i=1}^3 W_{i,1}^{(1)} \times x_i\right)$$

$$h_2 = \varphi\left(\sum_{i=1}^3 W_{i,2}^{(1)} \times x_i\right)$$



# 前向神经网络计算过程

- 前向神经网络本质上是一个多元复合函数



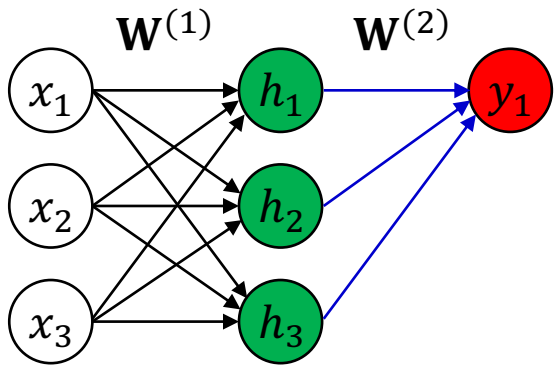
$$h_1 = \varphi\left(\sum_{i=1}^3 W_{i,1}^{(1)} \times x_i\right)$$

$$h_2 = \varphi\left(\sum_{i=1}^3 W_{i,2}^{(1)} \times x_i\right)$$

$$h_3 = \varphi\left(\sum_{i=1}^3 W_{i,3}^{(1)} \times x_i\right)$$

# 前向神经网络计算过程

- 前向神经网络本质上是一个多元复合函数



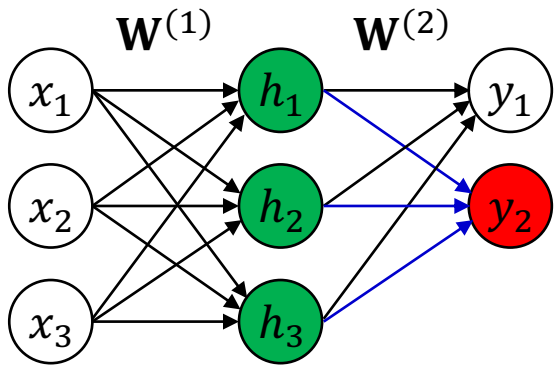
$$h_1 = \varphi\left(\sum_{i=1}^3 W_{i,1}^{(1)} \times x_i\right) \quad y_1 = \varphi\left(\sum_{j=1}^3 W_{j,1}^{(2)} \times h_j\right)$$

$$h_2 = \varphi\left(\sum_{i=1}^3 W_{i,2}^{(1)} \times x_i\right)$$

$$h_3 = \varphi\left(\sum_{i=1}^3 W_{i,3}^{(1)} \times x_i\right)$$

# 前向神经网络计算过程

- 前向神经网络本质上是一个多元复合函数



$$h_1 = \varphi\left(\sum_{i=1}^3 W_{i,1}^{(1)} \times x_i\right)$$

$$y_1 = \varphi\left(\sum_{j=1}^3 W_{j,1}^{(2)} \times h_j\right)$$

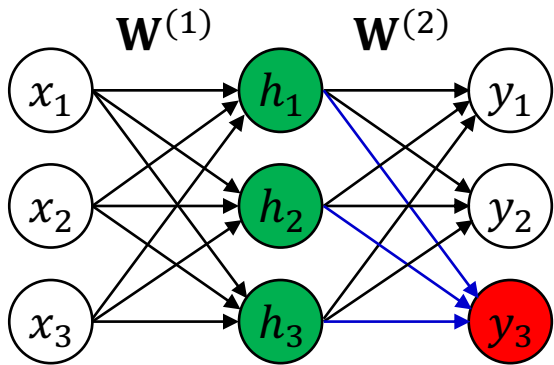
$$h_2 = \varphi\left(\sum_{i=1}^3 W_{i,2}^{(1)} \times x_i\right)$$

$$y_2 = \varphi\left(\sum_{j=1}^3 W_{j,2}^{(2)} \times h_j\right)$$

$$h_3 = \varphi\left(\sum_{i=1}^3 W_{i,3}^{(1)} \times x_i\right)$$

# 前向神经网络计算过程

- 前向神经网络本质上是一个多元复合函数



$$h_1 = \varphi\left(\sum_{i=1}^3 W_{i,1}^{(1)} \times x_i\right)$$

$$y_1 = \varphi\left(\sum_{j=1}^3 W_{j,1}^{(2)} \times h_j\right)$$

$$h_2 = \varphi\left(\sum_{i=1}^3 W_{i,2}^{(1)} \times x_i\right)$$

$$y_2 = \varphi\left(\sum_{j=1}^3 W_{j,2}^{(2)} \times h_j\right)$$

$$h_3 = \varphi\left(\sum_{i=1}^3 W_{i,3}^{(1)} \times x_i\right)$$

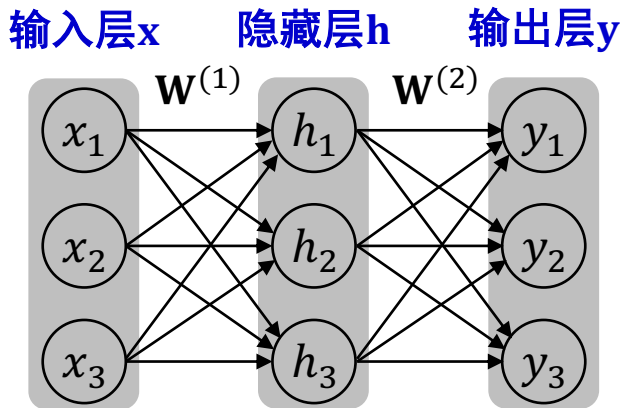
$$y_3 = \varphi\left(\sum_{j=1}^3 W_{j,3}^{(2)} \times h_j\right)$$

# 神经网络层结构与向量化表示

- 神经网络通常按照层来组织神经元。如图所示， $x_1$ 、 $x_2$ 与 $x_3$ 共同组成输入层  $\mathbf{x} \in \mathbb{R}^{3 \times 1}$ ， $h_1$ 、 $h_2$ 与 $h_3$ 共同组成隐藏层  $\mathbf{h} \in \mathbb{R}^{3 \times 1}$ ， $y_1$ 、 $y_2$ 与 $y_3$ 共同组成输出层  $\mathbf{y} \in \mathbb{R}^{3 \times 1}$ 。因此函数计算可以简化表示为：

$$\mathbf{h} = \varphi(\mathbf{W}^{(1)}\mathbf{x}), \quad \mathbf{y} = \varphi(\mathbf{W}^{(2)}\mathbf{h})$$

其中，激活函数 $\varphi(\cdot)$ 以向量作为输入时，对向量的每个元素单独计算



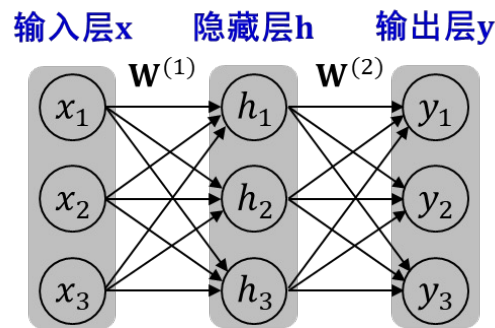
# 损失函数

- 前向神经网络通常用于有监督学习任务中，即将标注结果作为标准答案 (Ground Truth)
- 设前向神经网络为复合函数  $y = f(\mathbf{x}, \boldsymbol{\theta})$ ，其中  $\mathbf{x}$  为输入， $y$  为输出， $\boldsymbol{\theta}$  为模型参数，图示模型参数  $\boldsymbol{\theta} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}\}$
- 给定训练集  $\mathcal{D} = \{< \mathbf{x}^{(n)}, \mathbf{y}^{(n)} >\}_{n=1}^N$ ，神经网络的学习目标为：给定输入  $\mathbf{x}^{(n)}$ ，模型预测  $\mathbf{y}$  与标注结果  $\mathbf{y}^{(n)}$  尽量一致。定义损失函数如下：

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2} \sum_{n=1}^N (f(\mathbf{x}^{(n)}, \boldsymbol{\theta}) - \mathbf{y}^{(n)})^2$$

- 最优模型参数可以通过以下公式得到：

$$\hat{\boldsymbol{\theta}} = \arg \min_{\boldsymbol{\theta}} \{\mathcal{L}(\boldsymbol{\theta})\}$$



# 直接前继与后继

- 使用 $\text{heads}(u)$ 表示节点 $u$ 的**直接前继集合**， $\text{tails}(u)$ 表示**直接后继集合**，神经网络在节点 $u$ 处的**激活函数输入值**用 $a_u$ 表示，**激活函数输出值**用 $z_u$ 表示
- 将 $h$ 节点称为**头节点**， $t$ 节点称为**尾节点**， $h$ 节点是 $t$ 节点的一个直接前继， $t$ 节点是 $h$ 节点的一个直接后继，两者之间的连线上的权重用 $w_{h \rightarrow t}$ 表示， $t$ 节点的激活函数输入值和输出值计算如下：

$$a_t = \sum_{h \in \text{heads}(t)} w_{h \rightarrow t} z_h \quad z_t = \varphi(a_t)$$

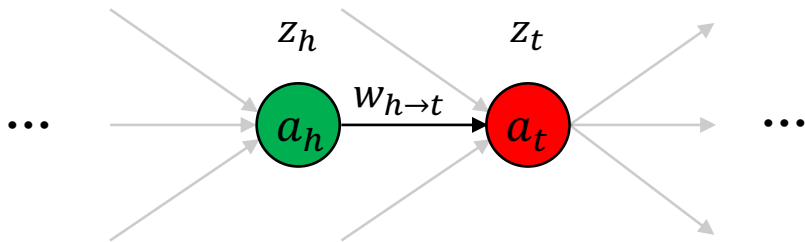
The diagram shows two nodes,  $h$  (green circle) and  $t$  (red circle). Node  $h$  has an input  $a_h$  and an output  $z_h$ . Node  $t$  has an input  $a_t$  and an output  $z_t$ . A directed arrow labeled  $w_{h \rightarrow t}$  connects  $z_h$  to  $a_t$ . There are also gray arrows representing other predecessors and successors: three arrows point to  $h$  from the left, and three arrows point away from  $t$  to the right. Ellipses (...) are used to indicate these additional connections.

# 节点错误

- 根据链式法则，损失函数对于模型参数 $w_{h \rightarrow t}$ 的偏导可以计算如下：

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial w_{h \rightarrow t}} = \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial a_t} \times \frac{\partial a_t}{\partial w_{h \rightarrow t}} = \delta_t \times z_h$$

- 将 $\delta_t$ 称为**节点错误**，即损失函数对 $t$ 节点激活函数输入值的偏导 $\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial a_t}$



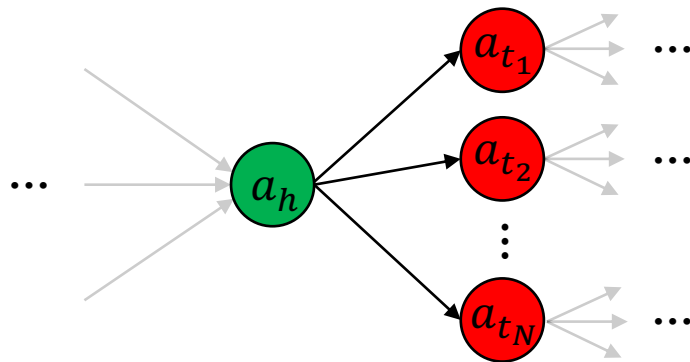


# 后向传播

- 使用链式法则进一步计算节点错误，由于某个节点的直接后继可以看成以该节点激活函数输入值为自变量的函数，因此：

$$\delta_h = \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial a_h} = \sum_{t \in \text{tails}(h)} \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial a_t} \times \frac{\partial a_t}{\partial a_h} = \varphi'(a_h) \sum_{t \in \text{tails}(h)} \delta_t \times w_{h \rightarrow t}$$

- 神经网络在计算节点错误时采用后向传播的方式

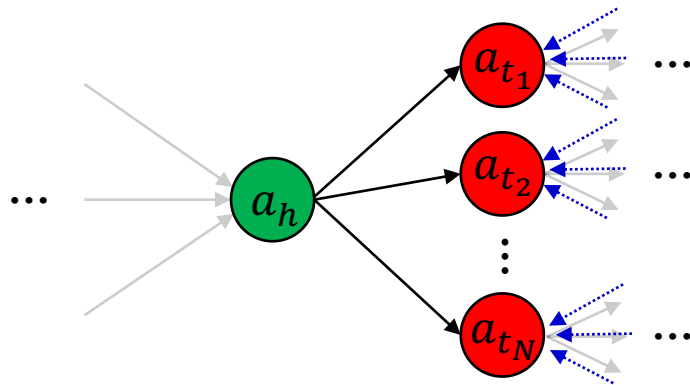


# 后向传播

- 使用链式法则进一步计算节点错误，由于某个节点的直接后继可以看成以该节点激活函数输入值为自变量的函数，因此：

$$\delta_h = \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial a_h} = \sum_{t \in \text{tails}(h)} \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial a_t} \times \frac{\partial a_t}{\partial a_h} = \varphi'(a_h) \sum_{t \in \text{tails}(h)} \delta_t \times w_{h \rightarrow t}$$

- 神经网络在计算节点错误时采用后向传播的方式

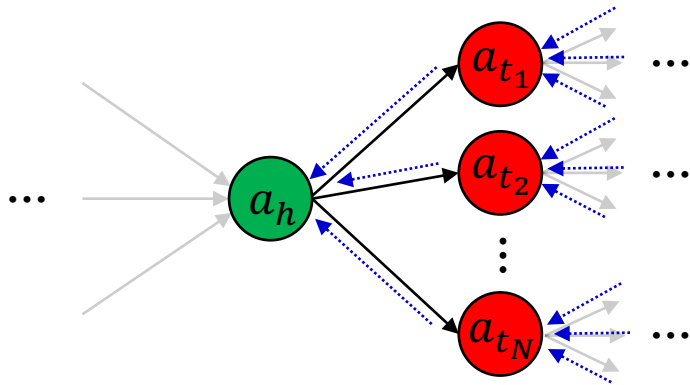


# 后向传播

- 使用链式法则进一步计算节点错误，由于某个节点的直接后继可以看成以该节点激活函数输入值为自变量的函数，因此：

$$\delta_h = \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial a_h} = \sum_{t \in \text{tails}(h)} \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial a_t} \times \frac{\partial a_t}{\partial a_h} = \varphi'(a_h) \sum_{t \in \text{tails}(h)} \delta_t \times w_{h \rightarrow t}$$

- 神经网络在计算节点错误时采用后向传播的方式

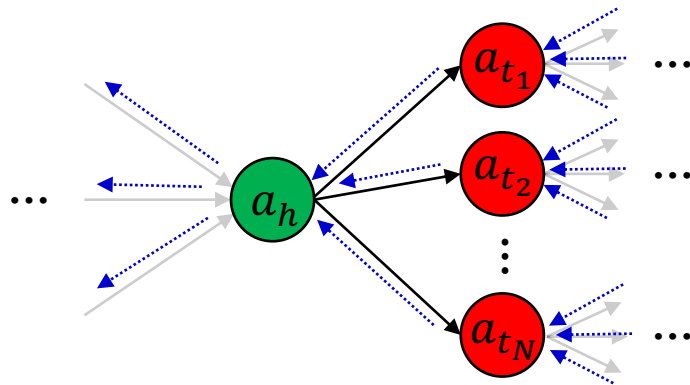


# 后向传播

- 使用链式法则进一步计算节点错误，由于某个节点的直接后继可以看成以该节点激活函数输入值为自变量的函数，因此：

$$\delta_h = \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial a_h} = \sum_{t \in \text{tails}(h)} \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial a_t} \times \frac{\partial a_t}{\partial a_h} = \varphi'(a_h) \sum_{t \in \text{tails}(h)} \delta_t \times w_{h \rightarrow t}$$

- 神经网络在计算节点错误时采用后向传播的方式

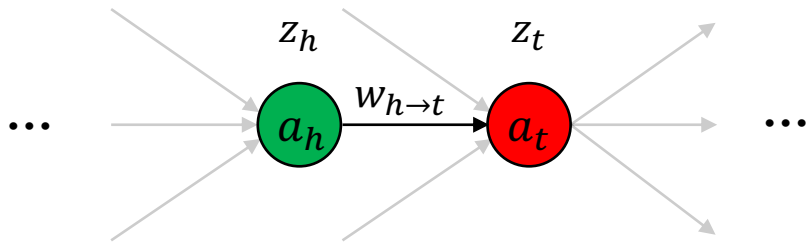


# 参数更新

- 在梯度下降策略下，参数 $w_{h \rightarrow t}$ 按照学习率 $\eta$ 沿负梯度方向进行更新：

$$w_{h \rightarrow t} \leftarrow w_{h \rightarrow t} - \eta \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial w_{h \rightarrow t}}$$

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial w_{h \rightarrow t}} = \delta_t \times z_h$$



# 后向传播算法

## ■ 后向传播算法主要包含以下步骤：

- 通过前向传播计算每个结点的激活函数输入值：

$$a_t = \sum_{h \in \text{heads}(t)} w_{h \rightarrow t} z_h \quad z_t = \varphi(a_t)$$

- 通过后向传播计算每个结点的错误：

$$\delta_h = \varphi'(a_h) \sum_{t \in \text{tails}(h)} \delta_t \times w_{h \rightarrow t}$$

- 更新参数：

$$w_{h \rightarrow t} \leftarrow w_{h \rightarrow t} - \eta \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial w_{h \rightarrow t}}$$

## ■ 后向传播算法使得自动训练大规模神经网络具备可能性

# 本章大纲

---

## ■ 神经网络

- 前馈神经网络

- 自注意力神经网络

## ■ Transformer

# 注意力

- 视觉注意力是人类大脑的天然机制：当我们看到一张图片时，往往先快速扫视，然后聚焦在关键区域，例如人脸、标题或首句内容。这样有利于高效处理信息
- 类似地，深度学习中引入“注意力机制”，使模型能够聚焦关键信息，从而提高效率与性能



A woman is throwing a frisbee in a park.



A dog is standing on a hardwood floor.

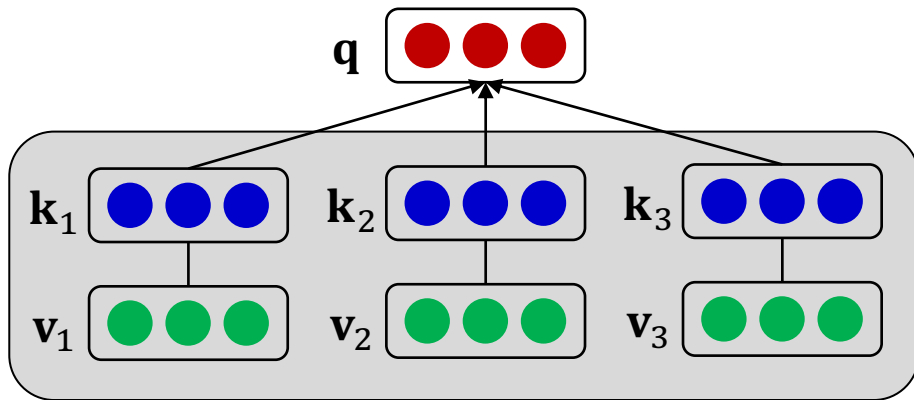


<https://zhuanlan.zhihu.com/p/379722366>



# 查询、键与值

- 注意力机制可以看成一种面向键值数据库的查询操作
- 假定一个键值数据库包含N个键 (key) :  $\mathbf{k}_1, \dots, \mathbf{k}_N$ , 其中每个键是一个实数向量  $\mathbf{k}_n \in \mathbb{R}^{D_k \times 1}$ ; 包含N个值 (value), 每个值也是一个实数向量  $\mathbf{v}_n \in \mathbb{R}^{D_v \times 1}$ , 并且与  $\mathbf{k}_n$  一一对应; 对于一个给定查询 (query)  $\mathbf{q} \in \mathbb{R}^{D_k \times 1}$ , 我们的目标是通过匹配查询和键, 将所需要的值提取出来



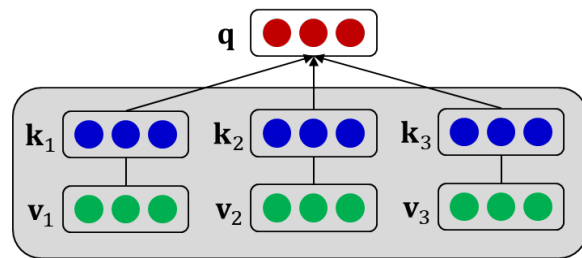
# 注意力计算

- 计算查询与键的**匹配度**是注意力机制的关键所在
- 定义函数 $\text{score}(\mathbf{q}, \mathbf{k})$ 表示查询 $\mathbf{q}$ 和键 $\mathbf{k}$ 之间的匹配度，该函数可定义为向量内积：

$$\text{score}(\mathbf{q}, \mathbf{k}) = \mathbf{q} \cdot \mathbf{k} = \sum_{d=1}^D q_d k_d$$

- 由于匹配度本身为实数，需要使用softmax函数将其转化为非负实数，以便对不同的值加权求和，因此定义查询 $\mathbf{q}$ 和键 $\mathbf{k}$ 之间的**注意力权重**如下：

$$\alpha_{\mathbf{k}_n \rightarrow \mathbf{q}} = \frac{\exp(\text{score}(\mathbf{q}, \mathbf{k}_n))}{\sum_{n'=1}^N \exp(\text{score}(\mathbf{q}, \mathbf{k}_{n'}))}$$

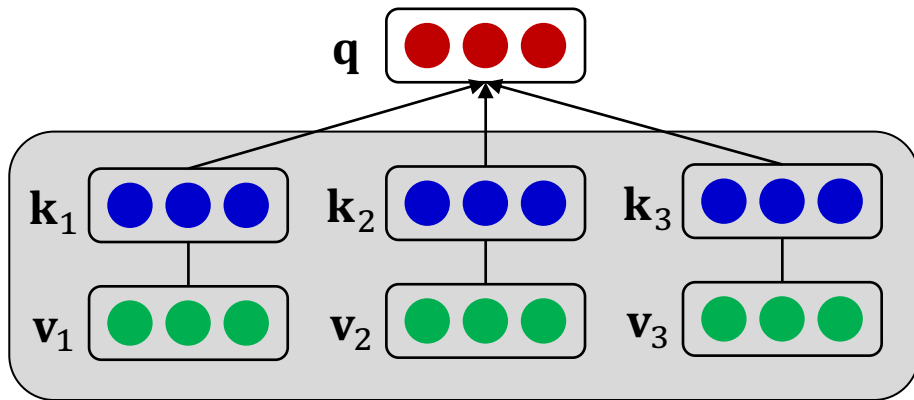


# 注意力计算

- 按照注意力权重对值 $v$ 进行加权求和得到最终查询结果：

$$\mathbf{o} = \sum_{n=1}^N \alpha_{\mathbf{k}_n \rightarrow \mathbf{q}} \mathbf{v}_n$$

其中， $\mathbf{o} \in \mathbb{R}^{D_v \times 1}$ 是一个实数向量

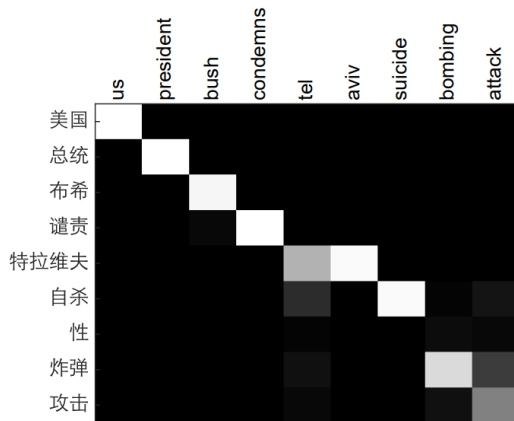


# 注意力权重矩阵

- 当存在多个查询（如 $\mathbf{q}_1, \dots, \mathbf{q}_N$ ）时，可以将第 $i$ 个查询 $\mathbf{q}_i$ 和第 $j$ 个键 $\mathbf{k}_j$ 之间的注意力权重表示为：

$$\alpha_{i,j} \equiv \alpha_{\mathbf{k}_j \rightarrow \mathbf{q}_i} = \frac{\exp(\text{score}(\mathbf{q}_i, \mathbf{k}_j))}{\sum_{n=1}^N \exp(\text{score}(\mathbf{q}_i, \mathbf{k}_n))}$$

从而获得一个二维的注意力权重矩阵 $\alpha = \{\alpha_{i,j} | i \in [1, N] \wedge j \in [1, N]\}$



[https://nlp.csai.tsinghua.edu.cn/~ly/papers/ijcai16\\_agree.pdf](https://nlp.csai.tsinghua.edu.cn/~ly/papers/ijcai16_agree.pdf)

# 矩阵形式的注意力机制

- 定义  $\mathbf{Q} \in \mathbb{R}^{N \times D_k}$  表示查询矩阵,  $\mathbf{K} \in \mathbb{R}^{N \times D_k}$  表示键矩阵,  $\mathbf{V} \in \mathbb{R}^{N \times D_v}$  表示值矩阵, 矩阵形式的注意力机制可以计算如下:

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

其中  $\frac{1}{\sqrt{D_k}}$  为**缩减因子**, 用于保持softmax输出分布相对平滑, 缓解梯度消失问题

# 自注意力机制

- 当查询矩阵 $Q$ 、键矩阵 $K$ 和值矩阵 $V$ 均由同一个输入 $X$ 通过不同线性变换得到时，我们称之为**自注意力机制**
- 自注意力机制能够有效发现输入内部元素之间的关联并获得特征表示

# 本章大纲

---

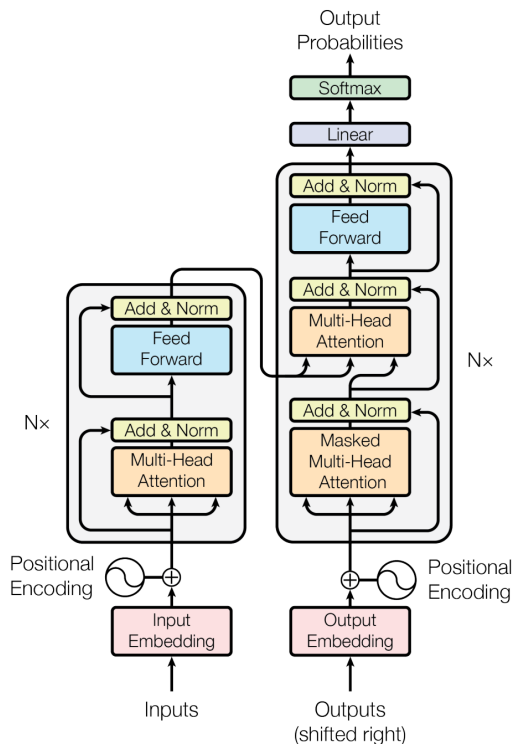
## ■ 神经网络

- 前馈神经网络（包括反向传播算法）
- 自注意力神经网络

## ■ Transformer

# Transformer

- Transformer 模型来自2017年的论文 *Attention Is All You Need*。该模型最初是为了提高机器翻译这类序列到序列 (seq2seq) 任务的效率，它的 Self-Attention 和 位置编码 (Positional Encoding) 可以替代 RNN，较容易进行并行。后来的研究发现 Self-Attention 效果优于其他结构，因此 Transformer 模型作为一种基本架构被广泛采用，如 BERT (Encoder-Only) 和 GPT (Decoder-Only) 等模型。

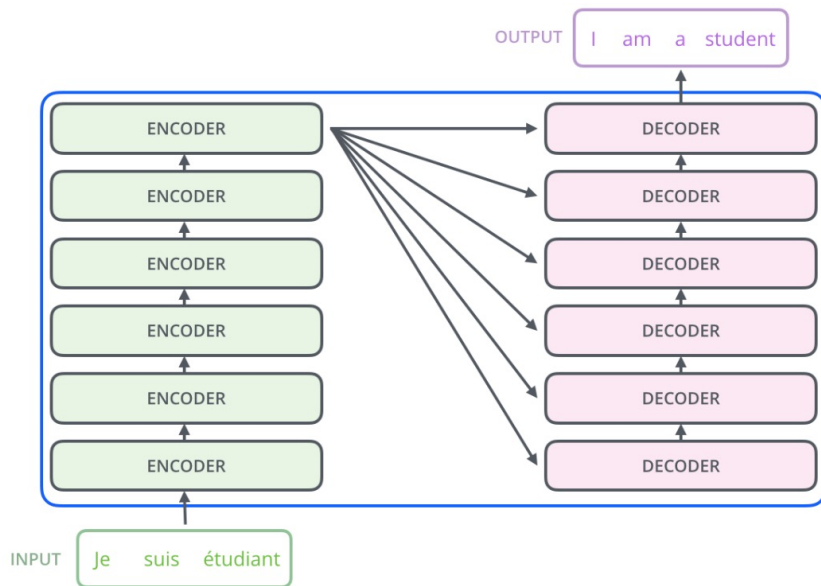


<https://arxiv.org/pdf/1706.03762>



# Transformer整体架构

- Transformer 可分为 Encoder 和 Decoder，各由多个结构相同的 Encoder / Decoder 层堆叠而成
- 词嵌入层（Embedding）：将输入的原始文本的每个 token 依次转化成向量，是一个可学习的模块
- 每一个 Encoder 层的输入是前一层 Encoder 的输出，首层 Encoder 的输入是经过词嵌入的原始输入以及位置编码之和
- Decoder 类似，但最后一层 Encoder 的输出会输入给每个 Decoder。



<https://jalamar.github.io/illustrated-transformer/>

# Transformer: Embedding & Positional Encoding

- 输入的句子是一个词 (ID) 的序列，首先通过 Embedding 层把它变成一个连续稠密的向量：



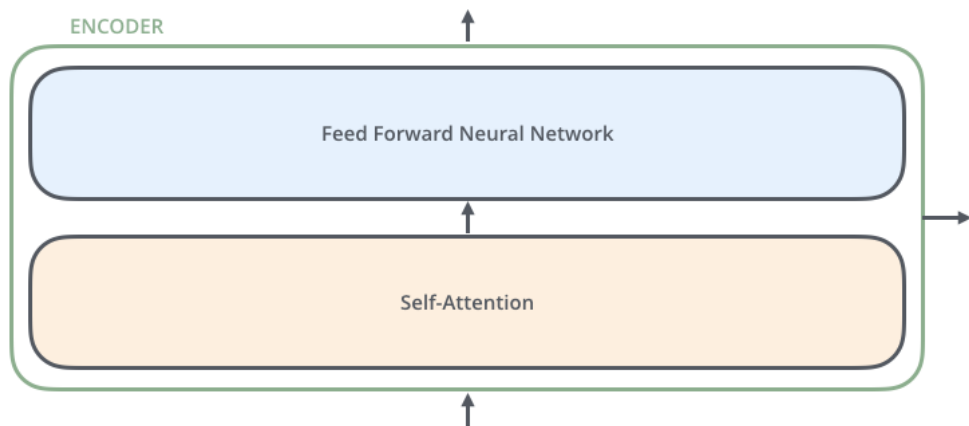
- 将经过词嵌入的序列输入 Encoder 之前，还需要一步位置编码，以编码每个词在输入的序列的位置信息（时间信息）。可以期待的是，同个词在序列中不同位置出现时将有不同的编码结果。



<https://jalammar.github.io/illustrated-transformer/>

# Transformer: Encoder

- 每一层 Encoder 都是相同的结构，它由一个 Self-Attention 层和一个 FFN 层（前馈网络，一个全连接网络）组成，大致结构如下图所示：

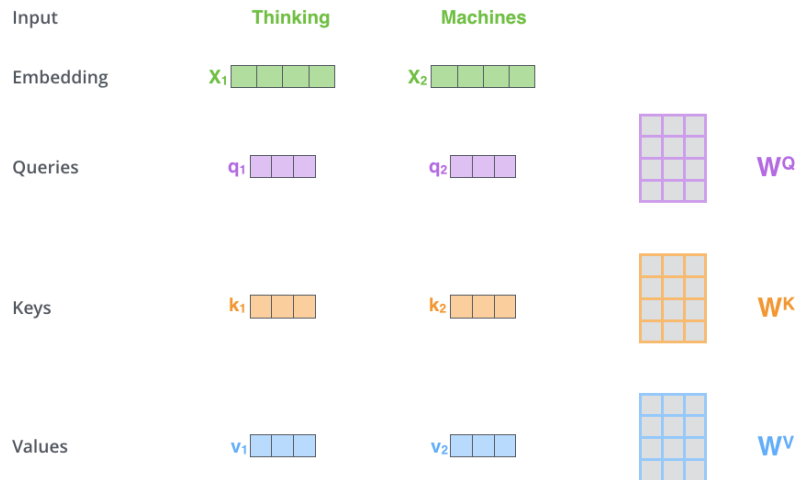


<https://jalammar.github.io/illustrated-transformer/>

# Transformer: Self-Attention

- 对输入的每一个向量  $X$ ，首先需要生成3个新的向量  $Q$ 、 $K$  和  $V$ ，分别代表查询 (query) 向量、键 (key) 向量和值 (value) 向量。 $Q$  表示为了编码当前词，要去注意其它 (包括自己) 的词，则需要一个查询向量。
- 对于输入的  $X$ ，通过三个线性变换的矩阵  $W^Q$ ， $W^K$  和  $W^V$ ，得到  $Q$ 、 $K$  和  $V$ ：

$$Q=XW^Q, \quad K=XW^K, \quad V=XW^V$$



<https://jalammar.github.io/illustrated-transformer/>

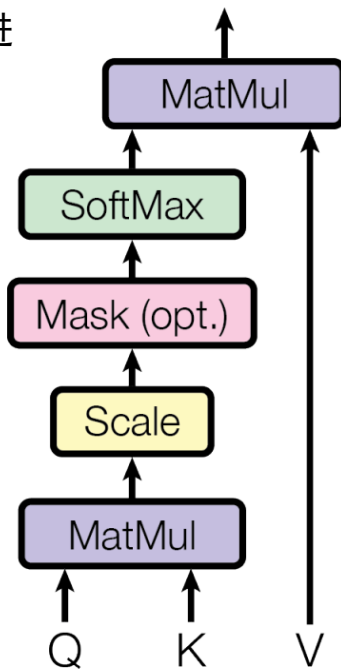
# Transformer: Self-Attention

- 对于得到的  $Q$ 、 $K$  和  $V$ ，如右图，将  $Q$  和  $K$  相乘得到内积，调整数值大小，进行 softmax 归一化，再和  $V$  相乘得到最终的 attention 值，具体公式为：

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

- 其中，这里  $d_K$  表示矩阵  $K$  的维度，除以根号  $d_K$  为了防止内积过大
- 公式括号内这个矩阵可以表示单词之间的 attention 强度
- Softmax 函数为：

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



<https://arxiv.org/pdf/1706.03762>

# Transformer: Self-Attention

## 公式:

$$\text{softmax}\left(\frac{\begin{matrix} \text{Q} \\ \begin{matrix} \square & \square & \square \\ \square & \square & \square \end{matrix} \end{matrix} \times \begin{matrix} \text{K}^T \\ \begin{matrix} \square & \square \\ \square & \square \end{matrix} \end{matrix}}{\sqrt{d_k}}\right) \begin{matrix} \text{V} \\ \begin{matrix} \square & \square \\ \square & \square \end{matrix} \end{matrix}$$

## 示例:

Input

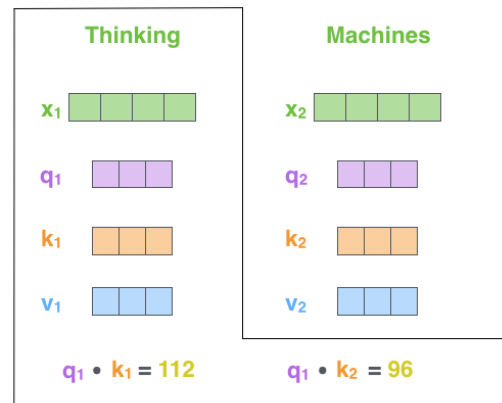
Embedding

Queries

Keys

Values

Score



<https://jalammar.github.io/illustrated-transformer/>

# Transformer: Self-Attention

## 公式：

$$\text{softmax}\left(\frac{\begin{matrix} \text{Q} \\ \begin{matrix} \square & \square & \square \\ \square & \square & \square \end{matrix} \end{matrix} \times \begin{matrix} \text{K}^T \\ \begin{matrix} \square & \square \\ \square & \square \end{matrix} \end{matrix}}{\sqrt{d_k}}\right) \begin{matrix} \text{V} \\ \begin{matrix} \square & \square \\ \square & \square \end{matrix} \end{matrix}$$

## 示例：

Input

Embedding

Queries

Keys

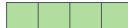
Values

Score

Divide by 8 (  $\sqrt{d_k}$  )

Softmax

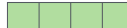
Thinking

 $x_1$   $q_1$   $k_1$   $v_1$   $q_1 \cdot k_1 = 112$ 

14

0.88

Machines

 $x_2$   $q_2$   $k_2$   $v_2$   $q_1 \cdot k_2 = 96$ 

12

0.12

# Transformer: Self-Attention

## 公式：

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

## 示例：

Input

Embedding

Queries

Keys

Values

Score

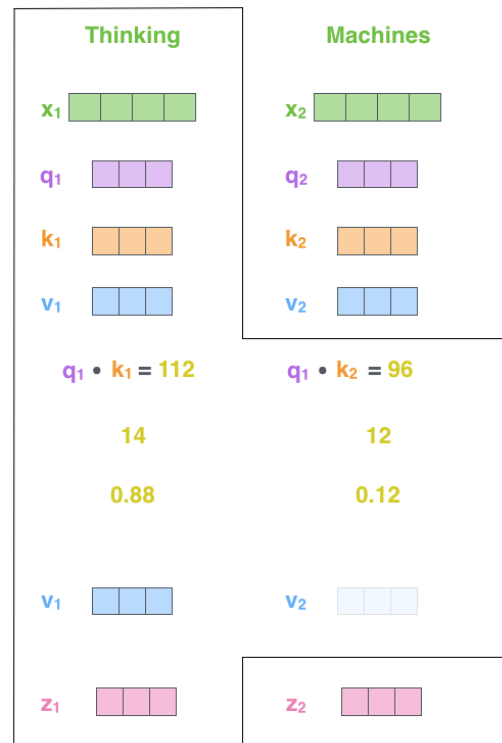
Divide by 8 ( $\sqrt{d_k}$ )

Softmax

Softmax

X  
Value

Sum



<https://jalammar.github.io/illustrated-transformer/>



# Transformer: Multi-head Attention

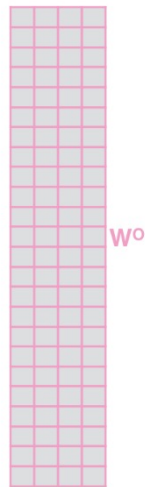
- 多头注意力：定义多组  $Q$ 、 $K$  和  $V$ ，它们分别可以关注不同的上下文，增强表示能力。
- 采用多头注意力时，每个头计算得到的结果  $Z_i$  进行拼接，

1) Concatenate all the attention heads



2) Multiply with a weight matrix  $W^O$  that was trained jointly with the model

x



3) The result would be the  $Z$  matrix that captures information from all the attention heads. We can send this forward to the FFNN

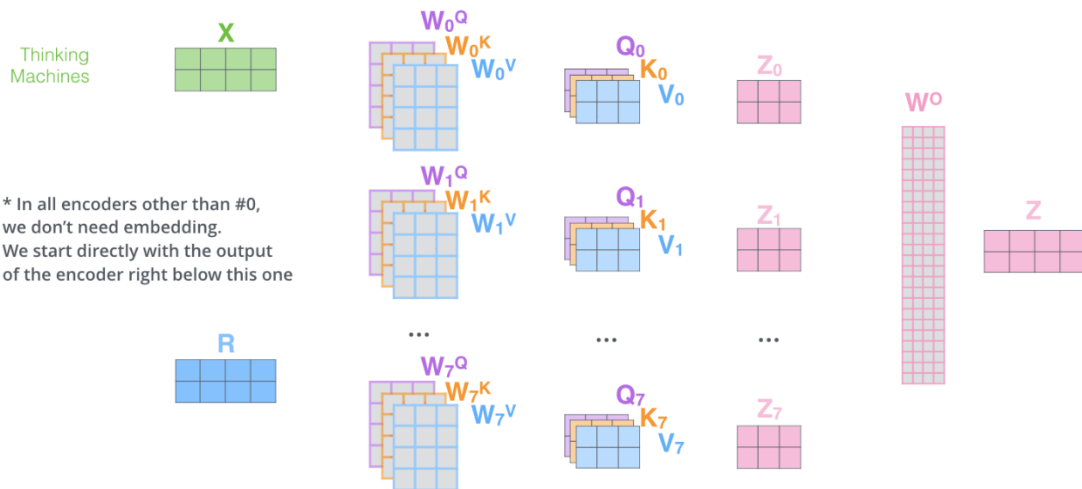


<https://jalammar.github.io/illustrated-transformer/>

# Transformer: Multi-head Attention

## ■ 以 8 个 head 为例，展示多头注意力的总体流程：

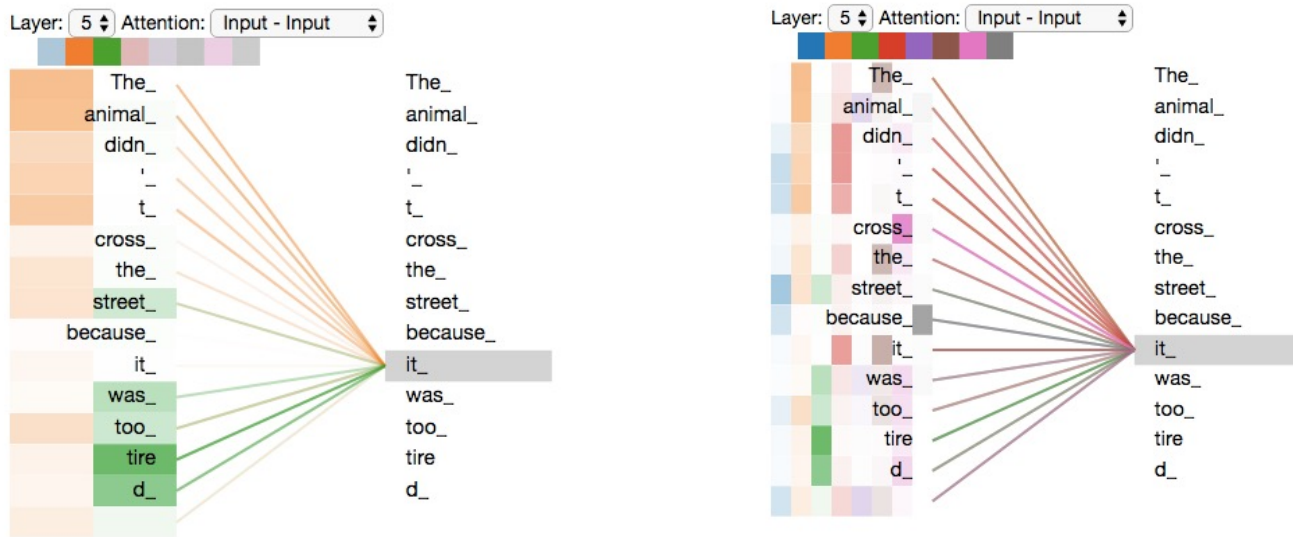
- 1) This is our input sentence\*
- 2) We embed each word\*
- 3) Split into 8 heads. We multiply  $X$  or  $R$  with weight matrices
- 4) Calculate attention using the resulting  $Q/K/V$  matrices
- 5) Concatenate the resulting  $Z$  matrices, then multiply with weight matrix  $W^O$  to produce the output of the layer



<https://jalammar.github.io/illustrated-transformer/>

# Transformer: Multi-head Attention

- 多头自注意力的两个例子，左图展示橙、绿两个 head 的注意力，橙色 head 关注对这只动物，绿色 head 关注它很累；右图展示所有 head 的注意力结果。



<https://jalammar.github.io/illustrated-transformer/>

# Transformer: Details of Encoder

- 在Encoder每个层中，除了 Self-Attention 与 FFN 外，还有 LayerNorm 与残差连接。
  - **Layer Normalization**: 对每个样本的不同特征进行 normalization，以加快收敛。对于单个输入样本  $X = [x_1, x_2, \dots, x_n]$ ,  $x \in \mathbb{R}^d$ （其中  $n$  表示每个样本的特征数量），对每个样本的所有特征计算其均值  $\mu$  和方差  $\sigma^2$ :

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

利用计算得到的均值和方差对该样本所有特征进行归一化:

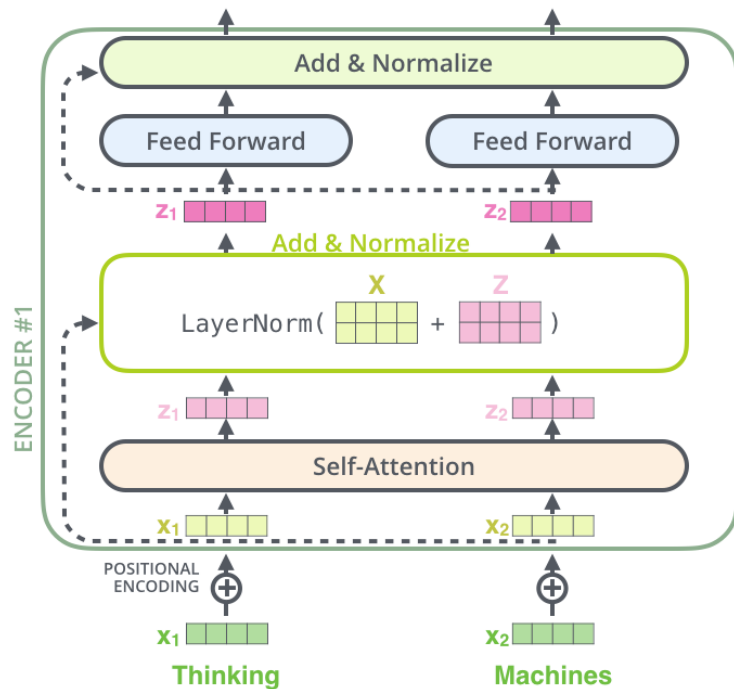
$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

其中  $\epsilon$  为很小的常数，防止分母为0。

# Transformer: Details of Encoder

- 残差连接 (residual connection): 将输入 self-attention 和 FFN 前后的向量进行相加

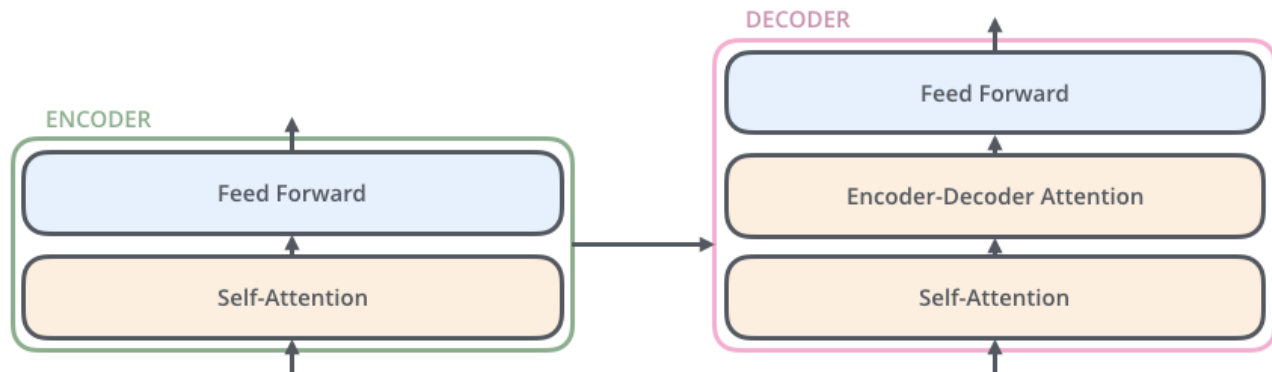
- 图中为完整的带有词嵌入 (Embedding)、位置编码、Self-Attention、FFN、LayerNorm 和残差连接的 Encoder 层示例，其中虚线表示残差连接



<https://jalammar.github.io/illustrated-transformer/>

# Transformer: Decoder

- 每一层 decoder 也是相同的结构，除了 Self-Attention 层和 FFN 层之外还多了一个 Attention 层：Encoder-Decoder Attention (Cross-Attention，交叉注意力)。这个 Attention 层使得 Decoder 在解码时会考虑最后一层 Encoder 所有时刻的输出。交叉注意力中，Q 为上一层 Decoder 的输出，K 和 V 均为最后一层 Encoder 的输出。



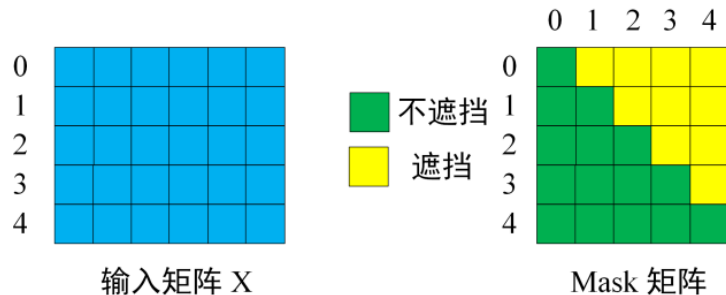
<https://jalammar.github.io/illustrated-transformer/>

# Transformer: Mask

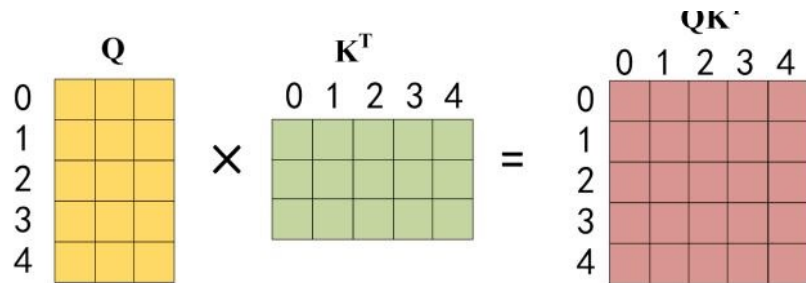
- Transformer 的训练是自回归的，在 Decoder 中，模型只能依靠 Encoder 编码的源端信息和截止到当前时间步 Decoder 已生成的内容，预测下一个 token。为了屏蔽未来信息，防止模型利用未来的信息来预测当前输出，需要在 Decoder 端添加掩码（Mask）
- 方法：在 Decoder 的 Self-Attention 层中，在计算 softmax 分数之前，应用一个掩码矩阵。
- 实现：通常用一个上三角矩阵来实现。

# Transformer: Mask Example

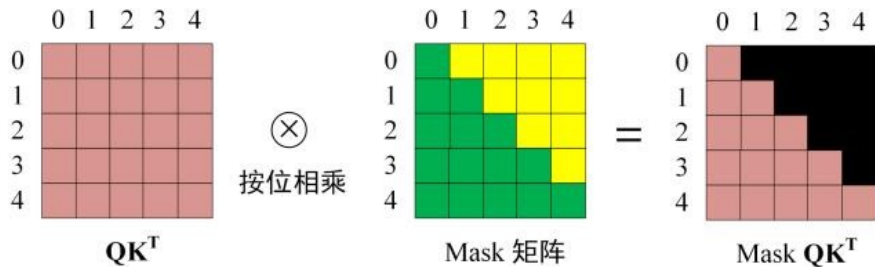
- 第1步：从输入矩阵构造 Mask 矩阵



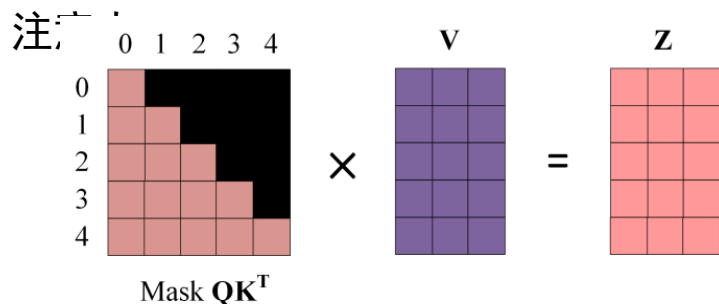
- 第2步：对 Q 矩阵和 K 矩阵求内积



- 第3步：在 softmax 之前进行 mask



- 第4步：用经掩码的  $QK^T$  与 V 相乘计算



<https://zhuanlan.zhihu.com/p/1333365028>



# Transformer: Output Layer

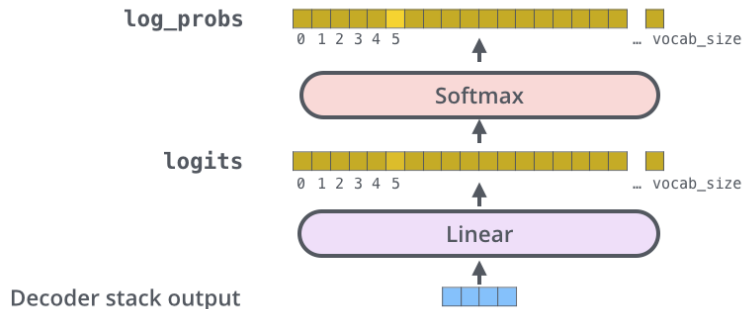
- 在经过所有 Decoder 层后，需要经由输出层输出最后的结果。输出层包含一个线性以及一个 softmax，具体步骤如下：

- Decoder 输出的向量经过词表大小的线性层，得到 logits。
- Logits 经过 softmax 操作，得到对数概率分布。
- 从概率分布中得到词表中概率最大的词。

Which word in our vocabulary is associated with this index?

am

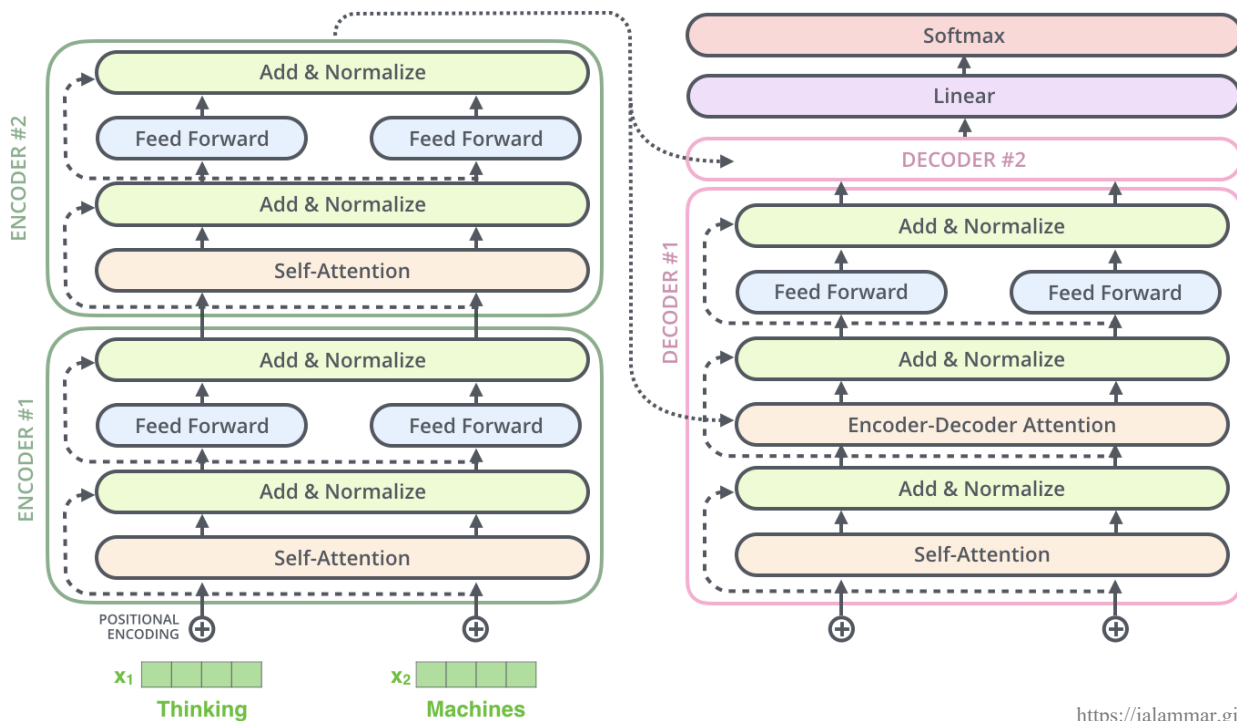
Get the index of the cell with the highest value (argmax)



<https://jalammar.github.io/illustrated-transformer/>

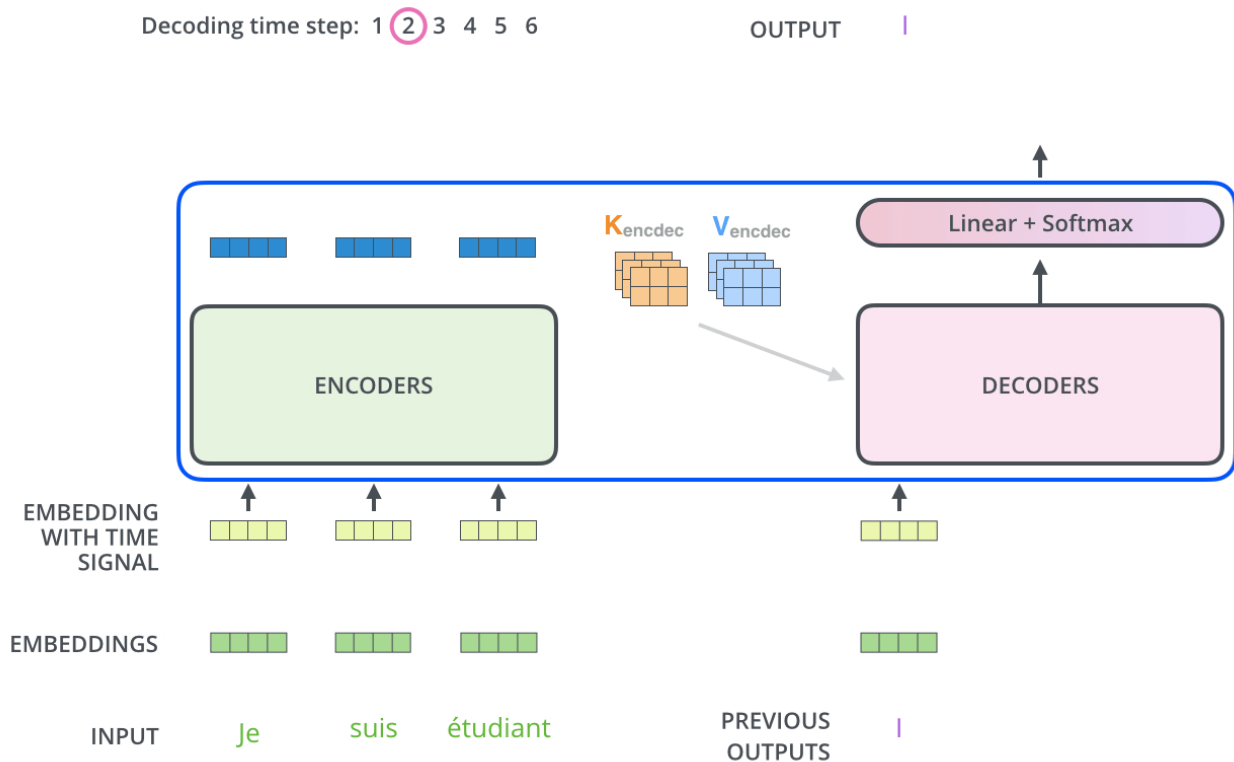
# Transformer: Architecture

- 一个2层 encoder + 2层 decoder 的 Transformer 模型的完整架构



<https://jalammr.github.io/illustrated-transformer/>

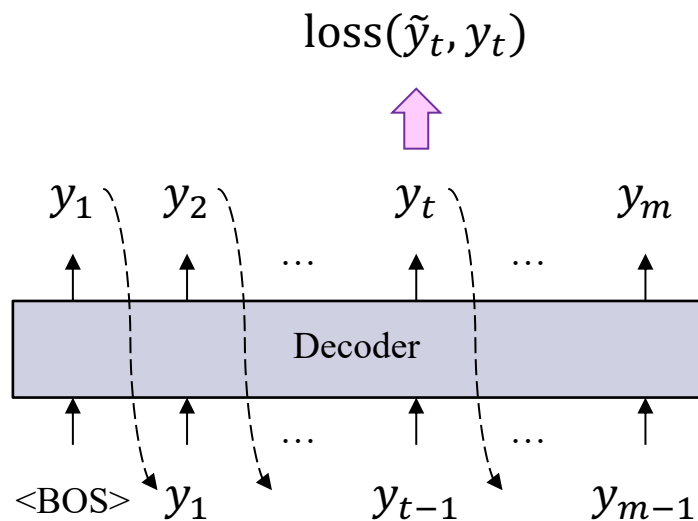
# Transformer: Architecture



<https://jalammr.github.io/illustrated-transformer/>

# Transformer: Autoregressive & Teacher-forcing

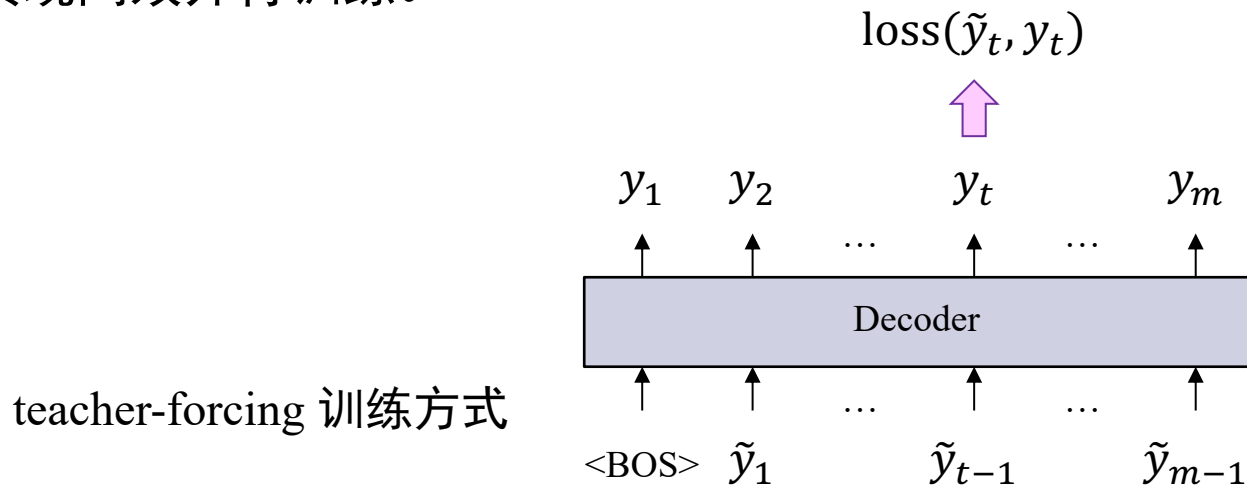
- Transformer decoder 在进行推理时采用自回归 (autoregressive) 的方式，生成序列时，每一步的预测依赖自己之前生成的输出。自回归的生成方式带来了错误累积，如果按这种方式训练，一步出错，则后面步骤将基于错误输入继续预测，错误越来越多，导致训练不稳定，模型难以收敛。且无法并行计算，训练速度慢。而 teacher-forcing 的训练方式解决了以上问题。



上图为自回归生成模式，其中  $\tilde{y}_t$  表示时刻  $t$  的标签

# Transformer: Autoregressive & Teacher-forcing

- **Teacher-forcing** 的核心思想是用“标准答案”指导训练。训练时，不再使用模型自己之前的预测值作为下一步输入，而改用**真实的目标序列值 (Ground Truth)** 作为输入。这极大提高训练稳定性和收敛速度，并能实现高效并行训练。



谢谢！