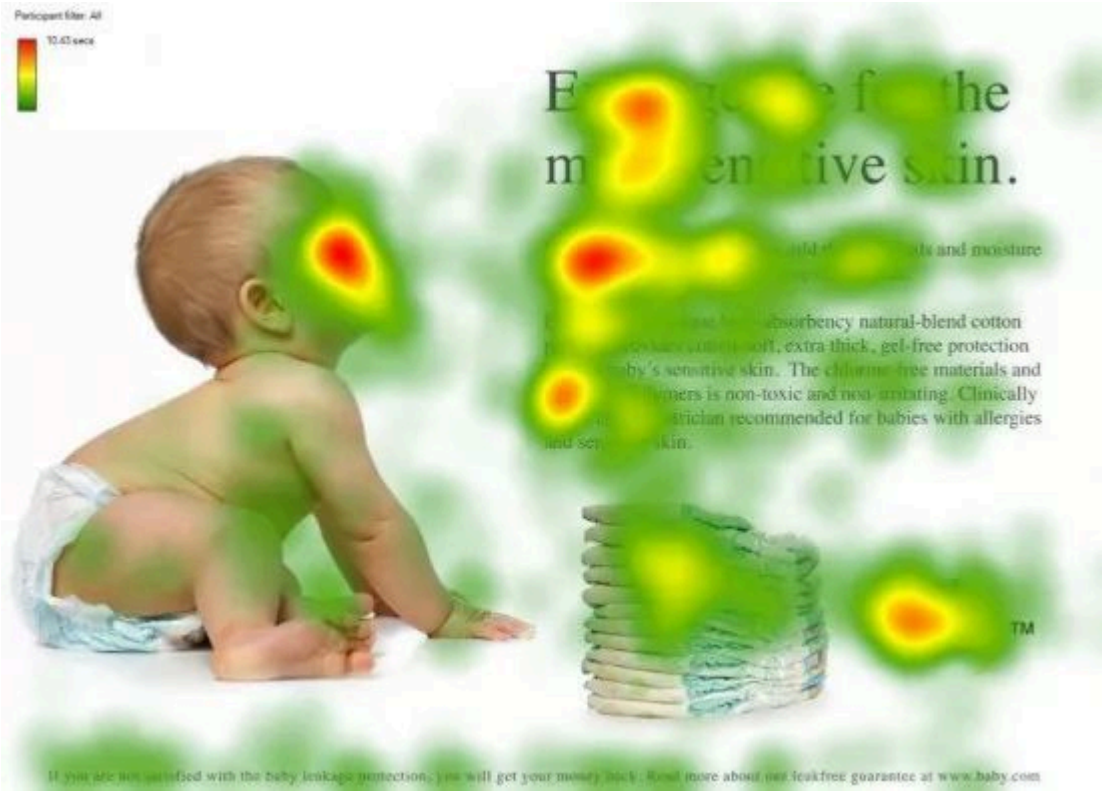
注意力机制（Attention Mechanism）允许模型在处理输入时，能够动态地关注输入序列中的不同位置，从而提高模型的性能和灵活性。Attention 机制的核心思想是让模型在处理每个输入时，能够关注到不同位置的重要信息，从而使得模型在处理长序列时更加高效

一. 注意力机制的语义

注意力机制的基本思想是，在处理序列数据（如文本、语音或时间序列）时，模型不必对整个序列进行平均处理，而是可以根据当前任务的需求，对序列中的某些部分给予更多的关注。这种机制可以增强模型的能力，使其能够更好地捕捉序列中的长距离依赖关系。



上图形象化展示了人类在看到一副图像时是如何高效分配有限的注意力资源的，**其中红色区域表明视觉系统更关注的目标**，很明显对于上图所示的场景，人们会把注意力更多投入到人的脸部，文本的标题以及文章首句等位置。

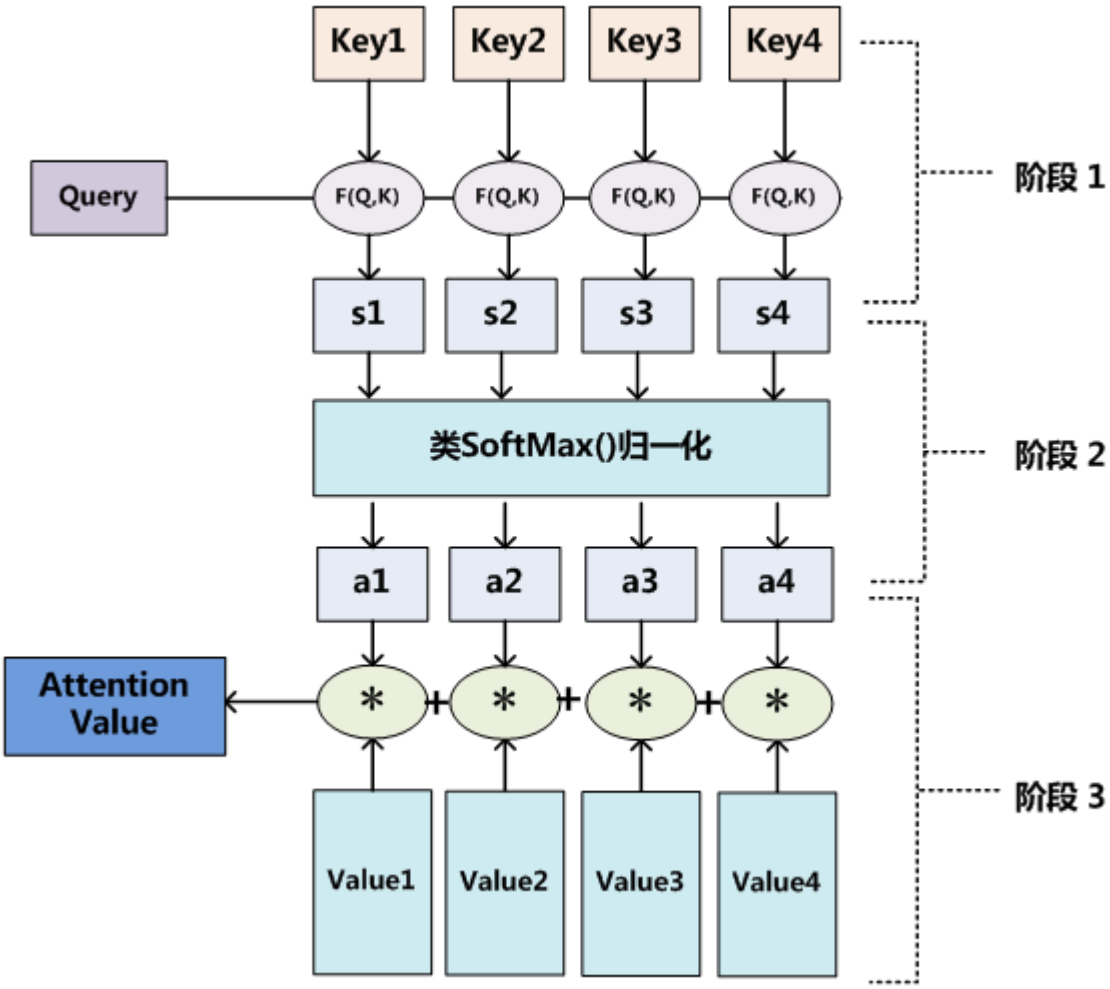
深度学习中的注意力机制从本质上讲和人类的选择性视觉注意力机制类似，**核心目标也是从众多信息中选择出对当前任务目标更关键的信息。**

注意力模型从大量信息 Values 中筛选出少量重要信息，这些重要信息一定是相对于另外一个信息 Query 而言是重要的，例如对于上面那张婴儿图，Query 就是观察者。也就是说，我们要搭建一个注意力模型，我们必须得要有一个 Query 和一个 Values，然后通过 Query 这个信息从 Values 中筛选出重要信息。

通过 Query 这个信息从 Values 中筛选出重要信息，简单点说，**就是计算 Query 和 Values 中每个信息的相关程度。**

其中对Q、K、V的简单描述如下：

- **查询向量（Query, Q）**：表示当前要处理或聚焦的输入信息。
- **键向量（Key, K）**：表示所有潜在匹配的信息，通常与查询向量进行比较以计算注意力得分。
- **值向量（Value, V）**：表示与键向量对应的实际信息，最终的注意力输出是这些值向量的加权和。



如上图，Attention 通常可以进行如下描述，表示为将 Query(Q) 和 key-value pairs（**把 Values 拆分成了键值对的形式**） 映射到输出上，其中 query、每个 key、每个 value 都是向量，输出是 V 中所有 values 的加权，其中权重是由 Query 和每个 key 计算出来的，计算方法分为三步：

1. 第一步：计算比较 Q 和 K 的相似度，用 f 来表示：$f(Q,K_i), i=1,2,\cdots,m$，一般第一步计算方法包括四种
    1. 点乘（**Transformer 使用**）：$f(Q,Ki)=Q^TK_i$
    2. 权重：$f(Q,K_i)=Q^TWK_i$

3. 拼接权重：$f(Q,K_i)=W[Q^T;K_i]$
4. 感知器：$f(Q,K_i)=V^Ttanh(WQ+UK_i)$

2. 第二步：将得到的相似度进行 softmax 操作，进行归一化：$\alpha_i=softmax(\frac{f(Q,K_i)}{\sqrt{d_k}})$
   - 这里简单讲解除以 $\sqrt{d_k}$ 的作用：假设 Q, K里的元素的均值为0，方差为 1，那么 $A^T=Q^TK$中元素的均值为 0，方差为 d。当 d 变得很大时， A 中的元素的方差也会变得很大，如果 A 中的元素方差很大(分布的方差大，分布集中在绝对值大的区域)，**在数量级较大时， softmax 将几乎全部的概率分布都分配给了最大值对应的标签**，由于某一维度的数量级较大，进而会导致 softmax 未来求梯度时会消失。总结一下就是 softmax(A) 的分布会和d有关。因此 A 中每一个元素乘上$\frac{1}{\sqrt{d_k}}$后，方差又变为 1，并且 A 的数量级也将会变小。
3. 第三步：针对计算出来的权重 α_i，对 V 中的所有 values 进行加权求和计算，得到 Attention 向量：$Attention=\sum_{i=1}^m\alpha_iV_i$

二. 注意力机制的计算过程

在Transformer架构中，注意力机制是其核心组件之一，主要用于捕获输入序列中不同位置之间的相关性。以下是几种主要的注意力机制：

1. Scaled Dot-Product Attention（缩放点积注意力）：

这是最常用的注意力机制。输入是查询向量（Query）、键向量（Key）和值向量（Value）。注意力权重通过计算查询与键的点积并进行缩放（除以键向量维度的平方根），然后通过Softmax函数归一化得到。最后，将这些权重应用于对应的值向量。

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

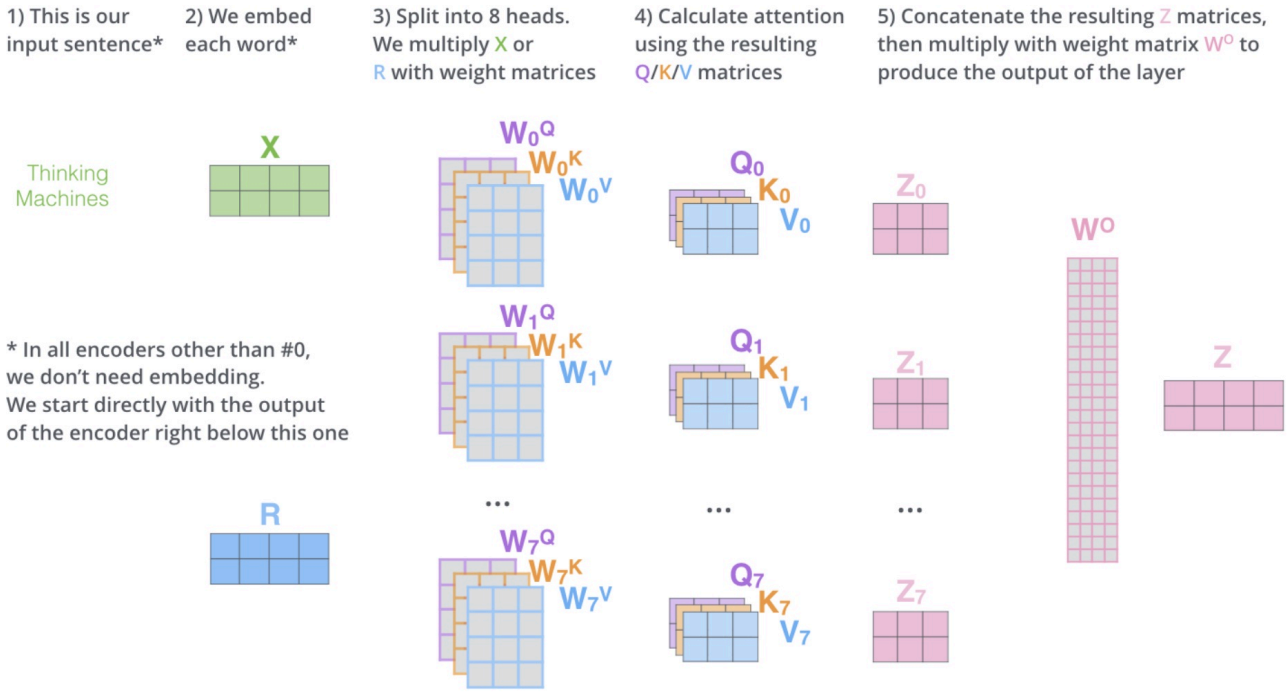
2. Multi-Head Attention（多头注意力）：

多头注意力机制是将输入分成多个头（head），对每个头分别进行缩放点积注意力计算，然后将所有头的输出拼接起来并通过一个线性变换。这样可以使模型在多个子空间上独立地关注不同部分的信息，增强模型的表达能力。

$$\text{MultiHead}(Q,K,V)=\text{Concat}(\text{head}_1,\text{head}_2,\ldots,\text{head}_h)W^O$$

$$\text{head}_i=\text{Attention}(QW_i^Q,KW_i^K,VW_i^V)$$

可以通过下图直观的看到multi_head attention的整个流程：（图中Z_i相当于公式中$head_i$）



3. Self-Attention（自注意力）：

自注意力是在Transformer中非常重要的一种注意力机制，尤其是在编码器和解码器内部。它是指查询、键和值向量都来自同一个输入序列，用来让每个位置的向量与其他位置的向量进行关联，从而捕获序列内部的依赖关系。也可以看做是对同一个序列X，通过**Scaled Dot-Product Attention**计算注意力分数。即：

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

- X是输入序列的表示。
- W_Q, W_K, W_V 是可训练的参数矩阵。

4. Cross-Attention（交叉注意力）：

交叉注意力常用于Transformer的解码器部分。在这种机制下，查询向量来自解码器的输入，而键和值向量来自编码器的输出。这样解码器可以在生成每个新词时参考编码器编码的信息。
交叉注意力的基本步骤与自注意力相似，但有一些关键的区别：

1. **输入**：
   - 源序列：通常表示为键（Key, K）和值（Value, V），由编码器序列提供，例如在机器翻译任务中，这可能是输入句子的编码表示。
   - 目标序列：通常表示为查询（Query, Q），为解码器序列，例如在机器翻译任务中，这可能是已生成的部分译文的编码表示。
2. **计算注意力权重**：

- 交叉注意力机制通过计算目标序列的查询Q和源序列的键K之间的点积来获得注意力权重。这些权重表示目标序列中每个位置与源序列中每个位置之间的相关性。
- 再使用$\text{Attention}(Q, K, V)$计算公式计算

3. **应用注意力权重**：
   - 使用计算得到的注意力权重，对源序列中的值进行加权求和，从而得到目标序列中每个位置的上下文表示。

5. Causal Attention（因果注意力）：

因果注意力（Causal Attention），有时也称为“遮掩注意力”（Masked Attention）或“自回归注意力”（Autoregressive Attention），是一种特殊的注意力机制，主要用于生成式模型中，以确保预测当前序列中的某一位置时，只能利用该位置之前的信息。这种机制在语言模型、文本生成、时间序列预测等任务中尤为重要。

因果注意力的主要特点:

1. **时间顺序约束**：在因果注意力机制中，模型在生成序列的第t个位置的输出时，只能利用从第1到第t个位置的信息，不能访问第$t+1$及其后的位置。这确保了生成过程的一致性和时间顺序约束。
2. **遮掩矩阵**：通过引入一个遮掩矩阵，明确规定了哪些位置的信息是可见的，哪些是不可见的，这通常实现为一个上三角矩阵。

因果注意力计算过程:
给定输入序列 $\mathbf{X} = [x_1, x_2, \ldots, x_n]$，再计算$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$

然后计算未归一化的注意力权重：

$$\mathbf{S} = \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}$$

使用mask matrix：

$$\mathbf{S}' = \mathbf{S} + \mathbf{M}$$

其中：$\mathbf{M}_{ij} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$
然后，使用Softmax函数来计算归一化的注意力权重：$\mathbf{W} = \text{softmax}(\mathbf{S}')$
最后，计算注意力输出：

$$\text{Attention}(Q, K, V) = WV$$

因为S经过mask后，右上部分（不包括对角线）已经全部变为$-\infty$，即得到的矩阵为

$$\begin{bmatrix} a_{11} & -\infty & \cdots & -\infty \\ a_{21} & a_{22} & \cdots & -\infty \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

再对此矩阵应用Softmax函数，得到的权重矩阵右上三角部分都为0。

6. Flash Attention

Flash Attention是一种高效的注意力机制实现方法，通过优化计算步骤和内存访问模式来显著提升自注意力机制（self-attention）的计算效率。自注意力机制是Transformer架构的核心组件，其计算复杂度通常为$O(N^2)$，其中N是序列的长度。这种二次方复杂度导致在处理长序列时计算资源和时间的消耗非常高。

Flash Attention的基本思想

1. 分块计算（Chunking）： 将输入序列划分为较小的块，并在这些块上分别执行注意力计算。这种方法减少了每次计算的内存需求，因为计算是在较小的数据块上进行的。此外，分块计算还利用了现代硬件（如GPU）的并行计算能力。
2. 内存访问优化： 通过优化内存访问模式，Flash Attention减少了内存带宽瓶颈。在传统的自注意力实现中，大量的内存读写操作会导致性能瓶颈。Flash Attention通过更高效的内存布局和预取技术，显著减少了这种开销。
3. 增量计算（Incremental Computation）：在分块计算的基础上，Flash Attention使用增量计算技术来避免重复计算。例如，在计算分块注意力时，可以利用之前块的计算结果来加速后续块的计算。

Flash Attention的具体实现

1. 首先定义$m(X) = max[x_1, x_2, \cdots, x_n]$，$f(X) = [e^{x_1 - m(X)}, \cdots, e^{x_n - m(X)}]$，$g(X) = \sum_i e^{x_i - m(X)}$，我们将X进行分块，得到$X = [X^1, X^2]$
2. 可以得到$softmax(X^1) = \frac{f(X^1)}{g(X^1)}$，至此$softmax(X^1)$并不能算是子向量$X^1$的最终结果，因为在$softmax(X^1)$中，指数项减去的值为$m(X^1)$，而不是整个向量的最大值$m(X)$，所以$softmax(X^1)$的计算结果是“局部的”
3. 保存一些额外的变量，在处理完X^2后更新X^1的softmax值，对X^1的计算完成后，保存$m(X^1)$和$g(X^1)$，相比于保存子向量X^1，保存两个标量的空间会小的多，其次再保存两个全局标量：m_{max}和g_{all}
    1. m_{max}表示当前最大值，因为暂时只处理了X^1，所以暂时$m_{max} = m(X^1)$
    2. g_{all}表示全局EXP求和项，因为暂时只处理了X^1，所以暂时$g_{all} = g(X^1)$
4. 接着采用类似的方法处理X^2，可得$softmax(X^2)$，同理，至此$softmax(X^2)$也并不能算是子向量X^2的最终结果，再进行如下操作

    1. 定义$m_{max}^{new} = max([m_{max}, m(X^2)])$，$g_{all}^{new} = e^{m_{max} - m_{max}^{new}} g_{all} + e^{m(X^2) - m_{max}^{new}} g(X^2)$
5. 可以得到更新后

$$softmax^{new}(X^1) = \frac{softmax(X^1) \cdot g_{all} \cdot e^{m_{max} - m_{max}^{new}}}{g_{all}^{new}}$$

$$softmax^{new}(X^2) = \frac{softmax(X^2) \cdot g(X^2) \cdot e^{m(X^2) - m_{max}^{new}}}{g_{all}^{new}}$$

6. 最后即可得到最终最整个向量X的softmax结果为

$$softmax(X) = \text{Concat}(softmax^{new}(X^1), softmax^{new}(X^2))$$

7. 可以看到，我们更新的过程只需要保存几个标量，不需要用到子向量进行更新，是一个增量计算过程
    1. 我们首先计算一个分块的局部softmax值，然后存储起来
    2. 当处理完下一个分块时，可以根据此时的新的全局最大值和全局EXP求和项来更新旧的softmax值，接着再处理下一个分块，然后再更新
    3. 当处理完所有分块后，此时的所有分块的softmax值都是“全局的”
8. 公式总结
    1. 对于向量$X^1, X^2, \cdots, X^n \in R^d$，有拼接向量$X = [X^1, X^2, \cdots, X^n] \in R^{nd}$
    2. $m(X) = max(m(X^1), \cdots, m(X^n))$
    3. $f(X) = [e^{m(X^1) - m(X)} f(X^1), \cdots, e^{m(X^n) - m(X)} f(X^n)]$
    4. $g(X) = g([X^1, X^2, \cdots, X^n]) = e^{m(X^1) - m(X)} g(X^1) + \cdots + e^{m(X^n) - m(X)} g(X^n)$
    5. $softmax(X) = \frac{f(X)}{g(X)}$

7. Paged Attention

PagedAttention，这是一种新颖的注意力算法，它将在操作系统的虚拟内存中分页的经典思想引入到 LLM 服务中。

Paged Attention通过以下几种方式优化标准自注意力机制：

1. **分页机制**：将输入序列划分为多个“页面”，每个页面包含固定数量的元素。页面之间的注意力计算是稀疏的，仅在需要时进行计算。
2. **块稀疏矩阵**：在每个页面内部和页面之间使用稀疏矩阵来表示注意力权重，从而减少不必要的计算。
3. **局部注意力机制**：主要关注局部页面内的注意力计算，减少全局注意力计算的频率。

数学表述
假设输入序列 $X \in R^{n \times d}$ 被划分为多个页面，每个页面大小为 P。给定查询 (Query)、键 (Key) 和值 (Value) 矩阵：

1. **计算查询、键和值矩阵**：

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

其中 $W_Q, W_K, W_V \in \mathbb{R}^{d \times d_k}$ 是可学习的权重矩阵

2. **分页划分**：将 Q、K、V 矩阵划分为多个页面，每个页面包含 P 个元素：

$$Q = [Q_1, Q_2, \ldots, Q_m]^T$$

其中 $Q_i \in R^{P \times d_k}$，$m = \frac{n}{P}$是页面的数量。

3. **局部注意力（页面内注意力计算）**：

$$A_{ii} = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)$$

其中 $A_{ii} \in R^{P \times P}$

4. **跨页面注意力（页面间注意力计算）**：
对于相邻页面 i 和 j（假设 i 和 j 相邻），计算跨页面的注意力权重矩阵：
$A_{ij} = \text{softmax}\left(\frac{Q_i K_j^T}{\sqrt{d_k}}\right)$
其中 $A_{ij} \in R^{P \times P}$。通常可以忽略远距离页面的注意力计算，以减少计算量

5. **注意力输出**：
汇总所有页面内部和页面间的注意力输出：

$$O_i = \sum_j A_{ij} V_j$$

其中$O_i \in R^{P \times d_k}$是页面 i 的输出。

8. Efficient Attention

Efficient Attention 是一类优化自注意力机制（Self-Attention Mechanism）的方法，旨在降低计算复杂度和内存使用，特别是在处理长序列时。不同于标准的自注意力机制，这些方法通过各种技巧和算法来减少计算量，使得处理长序列成为可能。

Efficient Attention的主要方法

1. **Linformer**
2. **Performer**
3. **Longformer**
4. **Reformer**

以下是每种方法的详细介绍

1. Linformer
Linformer 通过将注意力矩阵的投影降维，降低了计算复杂度。

核心思想：
使用两个线性投影矩阵$E \in R^{n \times k}$和$F \in R^{n \times k}$ 对键和值进行降维，其中$k \ll n$。 **公式**：

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

通过这样的方法，注意力矩阵的计算复杂度从$O(n^2)$降低到 $O(nk)$。

2. **Performer**
Performer 使用了核函数近似（Kernel Approximation）来近似标准注意力矩阵，从而降低计算复杂度。 **核心思想**：
利用核函数 $\phi(\cdot)$ 将点积注意力转化为线性复杂度的形式。 **公式**：

$$\text{Attention}(Q, K, V) \approx D^{-1}\left(\phi(Q)(\phi(K)^T V)\right)$$

其中：

$$D = \text{diag}\left(\phi(Q)(\phi(K)^T \mathbf{1})\right)$$

核函数常用高斯核或其他正交核函数。

3. **Longformer**
Longformer 通过局部窗口注意力和全局稀疏注意力来降低计算复杂度。 **核心思想**：
为每个位置定义一个固定大小的局部窗口，只在窗口内部计算注意力；对于少数全局关注的位置，进行全序列的稀疏注意力计算。 **公式**：

$$\text{Attention}(Q, K, V) = \text{LocalAttention}(Q, K, V) + \text{GlobalAttention}(Q, K, V)$$

4. **Reformer**
Reformer 通过使用局部敏感哈希（LSH, Locality Sensitive Hashing）来近似最近邻点，从而降低注意力计算复杂度。 **核心思想**：
使用 LSH 将相似的键（Key）聚集在一起，从而减少计算量。 **公式**:

$$\text{Attention}(Q, K, V) = \text{LSH-Attention}(Q, K, V)$$

9. Context Attention

Context Attention是用于增强Transformer模型中注意力机制的技术，它利用上下文信息来改进注意力权重的计算。这种方法有助于模型在处理长序列或复杂依赖关系时，更加有效地捕捉关键信息。Context Attention可以通过多种方式实现，包括引入全局上下文、层次上下文、动态上下文等

背景
Transformer模型中的标准自注意力机制在计算注意力权重时，仅依赖于查询（Query）和键（Key）之间的相似度，忽略了更广泛的上下文信息。这可能导致在处理长序列或复杂依赖关系

时，模型难以捕捉到重要的全局信息。Context Attention在计算注意力权重时，加入了对上下文信息的考虑，以提高模型的性能

Context Attention的主要方法

1. 全局上下文注意力（Global Context Attention）
   全局上下文注意力通过引入全局信息（如句子的整体表示或段落的总结）来增强注意力机制。全局信息通常通过池化操作或其他聚合操作获得 **核心思想**：利用全局上下文信息 C 来调整查询和键的相似度。 **公式**：

   $$A = \text{softmax}\left(\frac{QK^T + Q(CK^T)}{\sqrt{d_k}}\right)$$

   其中 C 是通过聚合操作（如平均池化）获得的全局上下文表示
2. 层次上下文注意力（Hierarchical Context Attention）
   层次上下文注意力通过分层次聚合信息，利用不同层次的上下文信息来改进注意力机制。常见的层次包括词级、句级和段落级。 **核心思想**：引入不同层次的上下文信息，并将其结合到注意力权重的计算中。 **公式**：

   $$A = \text{softmax}\left(\frac{QK^T + Q(C_{\text{word}}K^T) + Q(C_{\text{sentence}}K^T) + Q(C_{\text{paragraph}}K^T)}{\sqrt{d_k}}\right)$$

   其中 C_{word}、C_{sentence} 和 $C_{\text{paragraph}}$ 分别表示词级、句级和段落级的上下文信息
3. 动态上下文注意力（Dynamic Context Attention）
   动态上下文注意力通过动态调整上下文信息，以适应不同输入的需求。上下文信息可以根据当前查询的位置、内容等动态生成。 **核心思想**：根据当前查询动态生成上下文信息，并将其结合到注意力权重的计算中。 **公式**：

   $$C = f(Q, K, V)$$

   $$A = \text{softmax}\left(\frac{QK^T + Q(CK^T)}{\sqrt{d_k}}\right)$$

   其中 $f(\cdot)$ 是一个函数，用于动态生成上下文信息

10. Multi-Query Attention

Multi-Query Attention 是一种改进的注意力机制，旨在提高Transformer模型在多任务学习和多模态数据处理中的效率和性能。在标准的自注意力机制中，每个头（head）都有独立的查询（Query）、键（Key）和值（Value）矩阵。而在Multi-Query Attention中，查询矩阵是独立的，但键和值矩阵在所有头之间共享，这可以显著减少计算和内存需求，从而提高模型的效率。

背景
Multi-Query Attention通过共享键和值矩阵来减少计算和内存开销。具体来说，每个注意力头都有

独立的查询矩阵，但共享相同的键和值矩阵。这种方法保持了多头注意力的优点，同时显著降低了计算复杂度

核心思想
Multi-Query Attention通过共享键和值矩阵来减少计算和内存开销。具体来说，每个注意力头都有独立的查询矩阵，但共享相同的键和值矩阵。这种方法保持了多头注意力的优点，同时显著降低了计算复杂度

Multi-Query Attention的计算过程
假设输入序列 $X \in R^{n \times d}$，其中 n 是序列长度，d 是嵌入维度。对于标准的多头自注意力机制，每个头 h 都有独立的查询、键和值矩阵：

$$Q_h = XW_Q^h, \quad K_h = XW_K^h, \quad V_h = XW_V^h$$

在Multi-Query Attention中，查询矩阵是独立的，但键和值矩阵是共享的：

$$Q_h = XW_Q^h, \quad K = XW_K, \quad V = XW_V$$

对于每个注意力头 h，计算注意力权重和输出如下：
1. 计算注意力权重

$$A_h = \text{softmax}\left(\frac{Q_h K^T}{\sqrt{d_k}}\right)$$

2. 计算注意力输出

$$O_h = A_h V$$

3. 合并所有注意力头的输出

$$O = \text{Concat}(O_1, O_2, \ldots, O_H)W_O$$

其中 W_O 是线性投影矩阵，H 是注意力头的数量

优点：
1. **降低计算复杂度**：通过共享键和值矩阵，减少了计算量和内存需求。
2. **保持多头注意力的优点**：仍然可以捕捉到多种不同的注意力模式。
3. **提高效率**：特别是在处理大规模数据或需要高效计算的任务中，显著提高了模型的效率。

11. Additive Attention

Additive Attention，也称为Bahdanau Attention，是一种用于神经机器翻译（Neural Machine Translation, NMT）模型中的注意力机制，由Dzmitry Bahdanau等人在2014年提出。与Multiplicative Attention（点积注意力）不同，Additive Attention使用一个前馈神经网络来计算查询和键之间的相似度，从而生成注意力权重。

核心思想

Additive Attention通过引入可学习的权重矩阵，把查询和键投影到一个共同的表示空间，然后通过一个可学习的向量和非线性激活函数来计算相似度。这种方法在拟合能力上更加强大，特别适用于处理复杂的序列间依赖关系。

Additive Attention计算过程

1. **计算相似度分数**：对于每个查询 Q_i 和键 K_j，通过前馈神经网络计算其相似度分数 e_{ij}：

$$e_{ij} = v^T \tanh(W_Q Q_i + W_K K_j + b)$$

2. **计算注意力权重**：使用softmax函数将相似度分数归一化，得到注意力权重 α_{ij}：

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}$$

3. **计算加权求和值**：对值向量 V_j 进行加权求和，得到注意力输出：

$$\text{Attention}(Q, K, V) = \sum_j \alpha_{ij} V_j$$

总结

注意力机制通过对输入序列进行加权求和，使模型能够动态地关注序列中的不同部分，从而更好地捕捉长距离依赖关系。其核心计算过程包括计算注意力得分、缩放、应用掩码、计算注意力权重和加权求和。多头注意力机制通过并行计算多个注意力头，增强了模型的表现力和泛化能力。