# Rope位置编码

在文本模型中通常需要位置编码来区分文本，如下是几个常用的位置编码。

## PE

Transform使用的位置编码，其公式如下：

：即词向量 在文本中的位置，例如【我，爱，你】中的【我】属于第0个位置

：假设我们的单词被转成了64维的特征向量input embeding，那么i的取值就是0~31

：假设我们的单词被转成了64维的特征向量input embeding，那么d的值就是64

假设 ，input embedding 为64维，那么如何计算 input embedding 0 对应的 positional embeding 呢？

首先计算：

而 positional embedding 0 等于：

## 可学习的位置编码

在bert和gpt3中使用的是可学习的位置编码，利用bert的最大长度为512，其会预先生成512个embedding向量，每个向量代表一个位置，在训练过程中通过反向传播更新位置向量信息。

## Rope编码（旋转位置编码）

Rope是llama使用的位置编码。

1. **欧拉公式**

Rope需要借助复数计算的思想，如下，欧拉公式：

假设输入向量 属于2维，其中 m 为其位置。其中我们可以使用复数表示 ，即：

1. **二维情况下的位置编码**

Rope对于 的位置编码公式如下：

总结：

1. **多维情况下的位置编码**

观察公式：

如下我们将情况扩展到6维：

即：

其中：

* d 为x的维度。

1. **rope存在的问题**
2. 短距离注意力分数由低维度分量决定，长距离注意力分数由高维度分量决定

以二维向量为例，在计算模型的注意力分数时，其公式如下（如下求解位置m和n的注意力）：

假设 ，如下是距离 从 0-19 时，注意力分数的变化：

[2.0, 1.08, -0.83, -1.98, -1.31, 0.57, 1.92, 1.51, -0.29, -1.82, -1.68, 0.01, 1.69, 1.81, 0.27, -1.52, -1.92, -0.55, 1.32, 1.98]

现在令 ，则注意力分数的变化为：

[2.0, 1.76, 1.08, 0.14, -0.83, -1.6, -1.98, -1.87, -1.31, -0.42, 0.57, 1.42, 1.92, 1.95, 1.51, 0.69, -0.29, -1.2, -1.82, -1.99]

从上面2组数据我们可以观察到：

* 距离 时其注意力分数最大
* 注意力分数呈周期性变化， 越小，其周期性越长

当 越大，其周期性越短，频率越高；当 越小，其周期性长，频率越低；

现在我们将维度扩展到多维，由于：

假设 ，对于 ， ；对于 ，.

很明显，低维度的分量（高频）计算出来的注意力对距离是敏感，当距离超过某个数值时，其注意力分数会在0附近震荡；

很明显，高维度的分量（低频）对距离不敏感（如：对于 当距离为1其注意力分数最大，当距离为 31400 时其注意力分数才达到最小）；

当距离足够大时，低维度的分量（高频）计算出来的注意力分数会在0附近震荡，如果低维度的分量的数量足够多，那么这些注意力分数加起来就约等于0，那么此时的注意力分数就完全由高维度的分量来决定。

1. **Rope支持的最大长度**

理论上我们值需要确保 ，即 就可以有效的区分他的位置信息，但实际上 m 的值会更小。

1. **线性插值**

在训练时我们一般控制文本的长度在2k之内，那么在预测时如何输入大于2k的文本呢？如果不做任何处理，直接预测就会出现 这种情况，然而在训练时 的值从来没有出现过，这就会出现预测不准确的问题。

线性插值提供了简单的扩展上下文的方法，其公式如下：

这样就可以简单的将旋转角度缩放为原来的 倍。

问：为什么一开始不将文本拼接到64k来进行训练，这样模型可以学习到64k的文本，就不用进行插值了？

答：直接使用64k进行训练占用的显存极高；

## YaRN

Yarn是基于Rope位置编码方式，其提供的插值公式如下：

* *d*：模型维度
* *n*：目标上下文长度
* *α*,*β*：可调超参数（控制高频信息保留程度, 通过实验确定，论文中默认 α=1, β=32）

除此之外，在attention的计算中引入了温度缩放：

* YaRN通过实验确定最优温度T，缓解长上下文中注意力分数过大的问题。

## 总结

建立 freqs\_cis （frequency complex numbers 复数频率向量）矩阵：

* ：文本输入的长度
* ：需要应用位置编码的数据的维度

对于输入矩阵 ，通过如下公式对其应用位置编码：

对 进行拆分：

应用位置编码：

# LoRA 微调

**原始权重**：假设模型中的某个权重矩阵是W（例如，Transformer 中的注意力权重矩阵）。

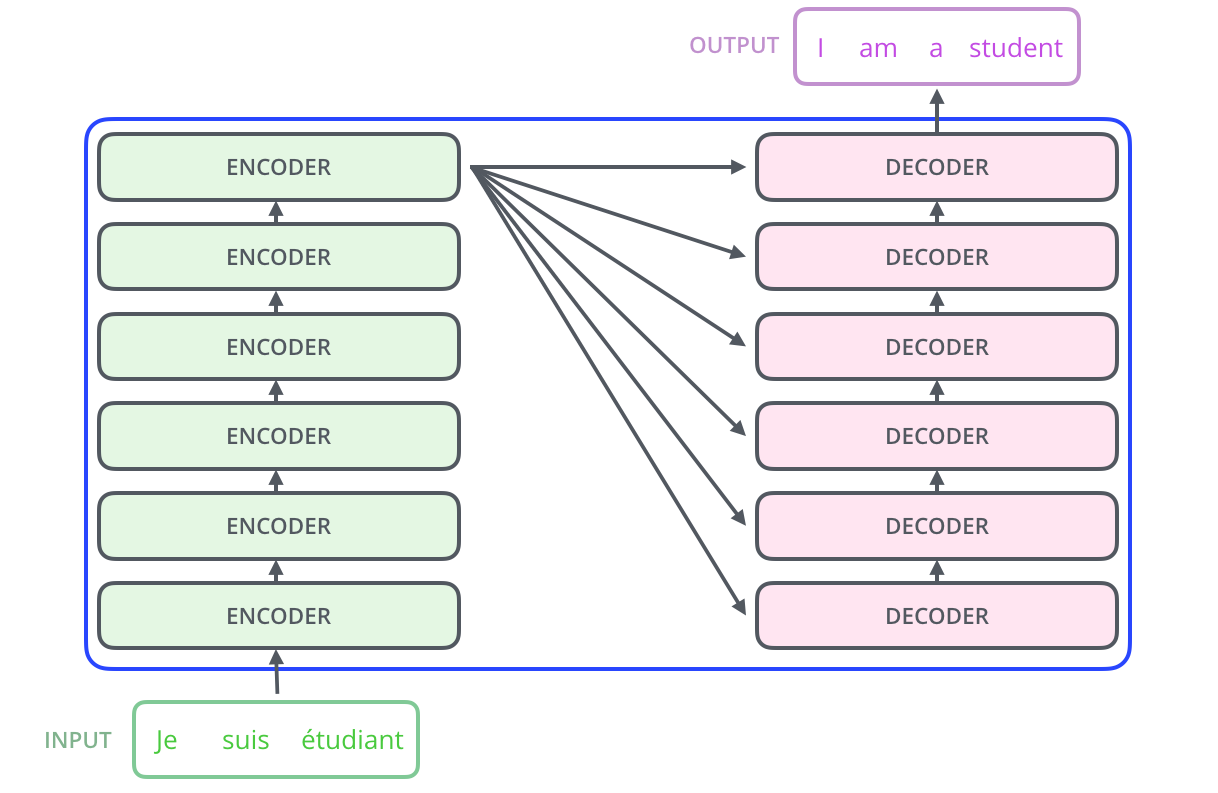
**低秩分解**：LoRA 引入两个低秩矩阵 A 和B，使得W的更新可以表示为：

**微调过程**：在微调时，只更新A和B，而不更新原始权重W。这样可以大大减少需要训练的参数数量。

低秩矩阵是指一个矩阵的秩（rank）远小于其行数或列数的矩阵，例如一个向量就是一个低秩矩阵，因为向量的秩只有1

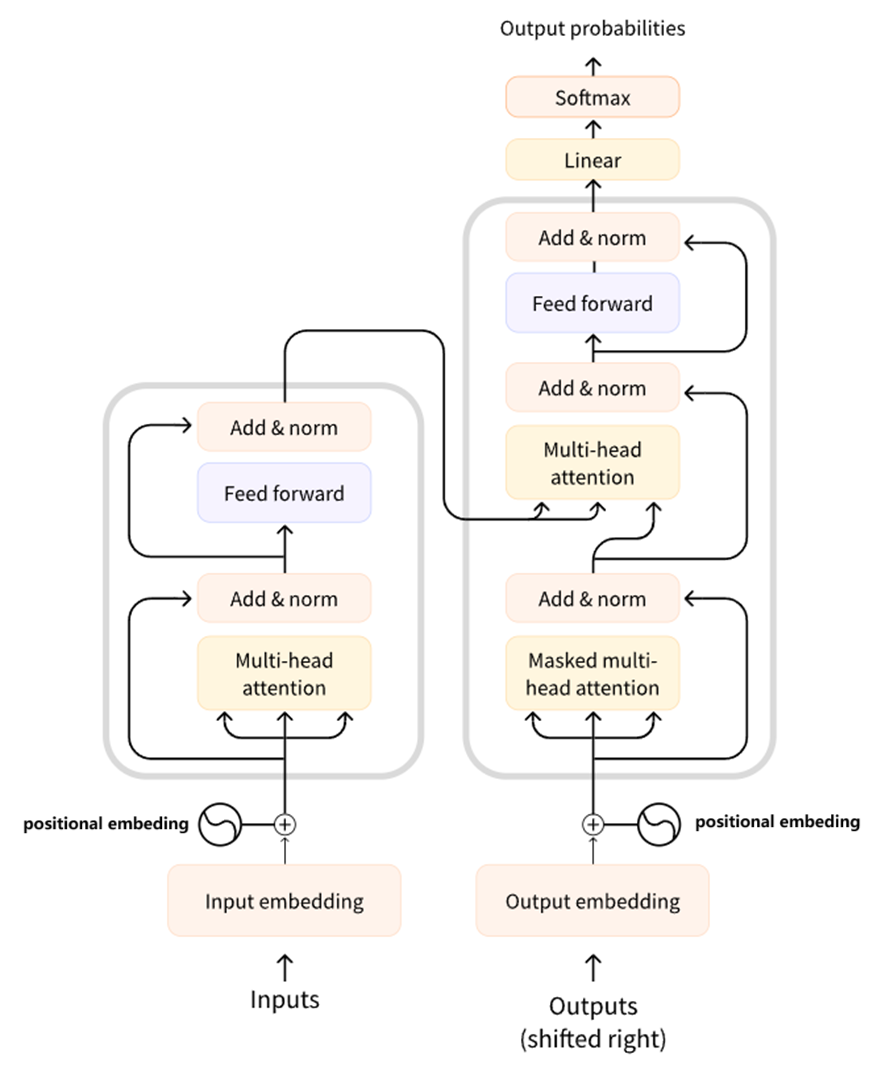
# Transformer

其基本结构如下：



其有6个encoder和6个decoder组成。

Encoder和decoder内部的处理流程如下：



下面详细讲解其过程

## Encoder

### 输入处理

首先我们的输入文本input的是由一个个单词组成的，如【我，爱，你】。

每个单词都会被转换为input embeding向量，然后再加上位置信息positional embeding，组成了我们的输入向量 。

positional embeding有具体的算法公式，如下：

：即词向量 在文本中的位置，例如【我，爱，你】中的【我】属于第0个位置

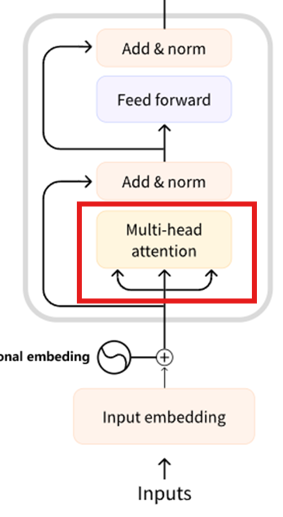
：假设我们的单词被转成了64维的特征向量input embeding，那么i的取值就是0~31

：假设我们的单词被转成了64维的特征向量input embeding，那么d的值就是64

### 多头注意力（Muti-head attention）

经过输入处理后我们得到 ，接下来我们需要使用 Muti-head attention 处理 。

Muti-head attention 是由多个自注意力模型（Self-Attention Model）组成



如下是Self-Attention Model的公式：

矩阵【 的第一列】为【第一个输入 与其他输入 的注意力分布】

图表, 箱线图

描述已自动生成

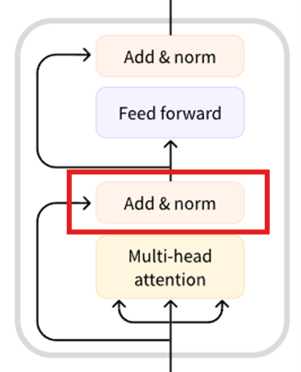
经过 Self-Attention Model 后会得到一个 矩阵数据，这里有一个特别的地方是我们需要确保 ，即

经过多个 Self-Attention Model 后会得到多个 矩阵数据

我们将输出的 纵向拼接成一个 的数据，然后再对其进行一个线性变换：

由此我们得到一个与原始输出形状相同的 矩阵

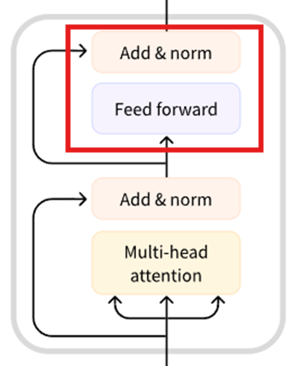
### Add & Norm



得到 矩阵后我们需要进行 Add & Norm ，其公式如下：

其实这是一个残差的形式

### Feed Forward 与 Add & Norm

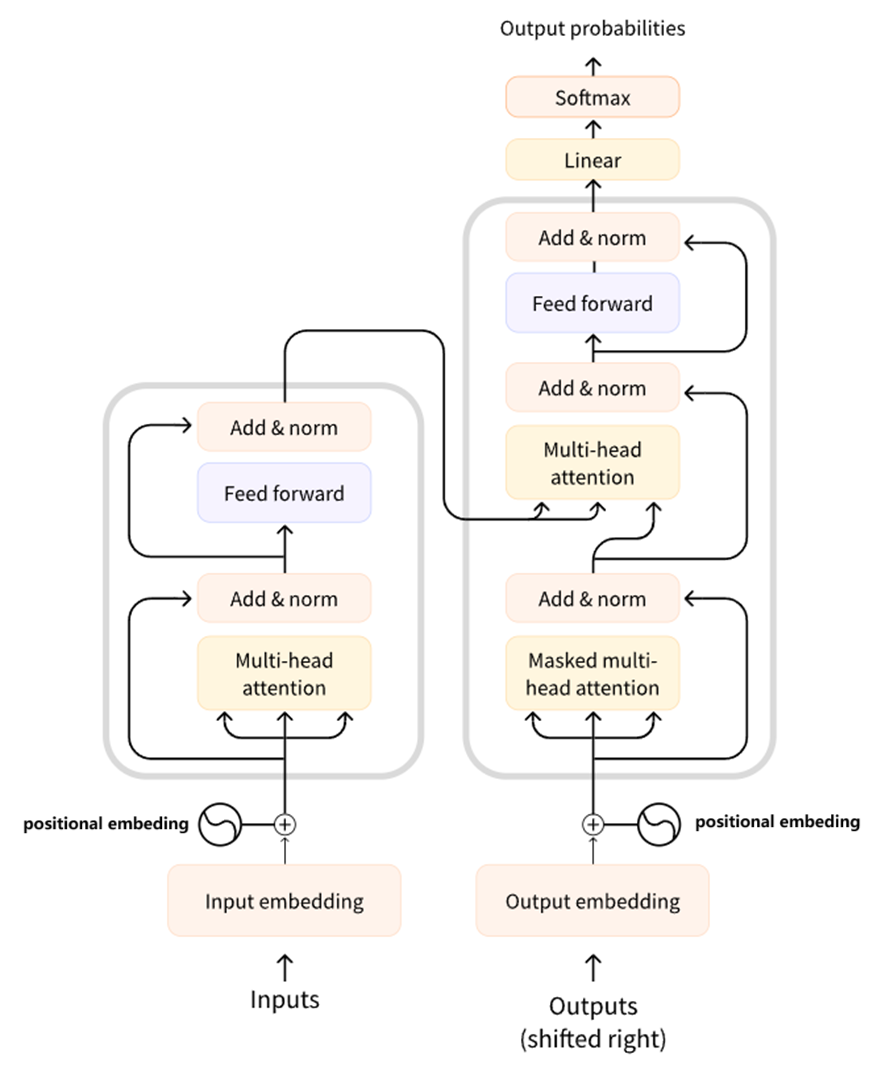


Feed Forward 是一个2层全连接网络，第一层激活函数维Relu，第二层不使用，其公式如下：

Add & Norm与上面的一样：

如此， 经过6个Encoder的处理，便可得到最终的输出 。

## Decoder



### 输入处理

其输入处理流程与Encoder一样

### Masked Muti-head attention

这是一个带掩码的多头注意力机制

其公式如下：

矩阵【 的第一列】为【第一个输入 与其他输入 的注意力分布】

该公式比 Self-Attention Model 多了一个Mask

：右上角全为1，左下角全为0的矩阵

：Hadamard积，即元素的各个对应位置进行相乘

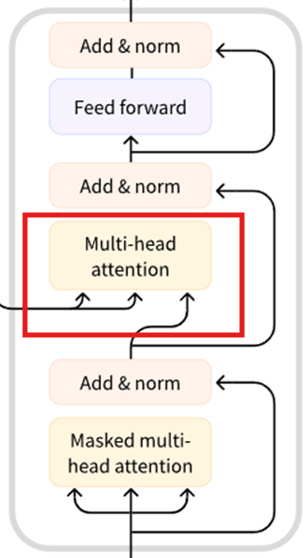
为什么要加mask？

再测试时，我们的输入 等于上一次的输出 ，不经过预测，我们是不知道， 的内容的，也就是说 只与 有关

接下来的过程与Encoder一样：

将多个 横向拼接成一个 的数据，然后再对其进行一个线性变换：

### Muti-head attention



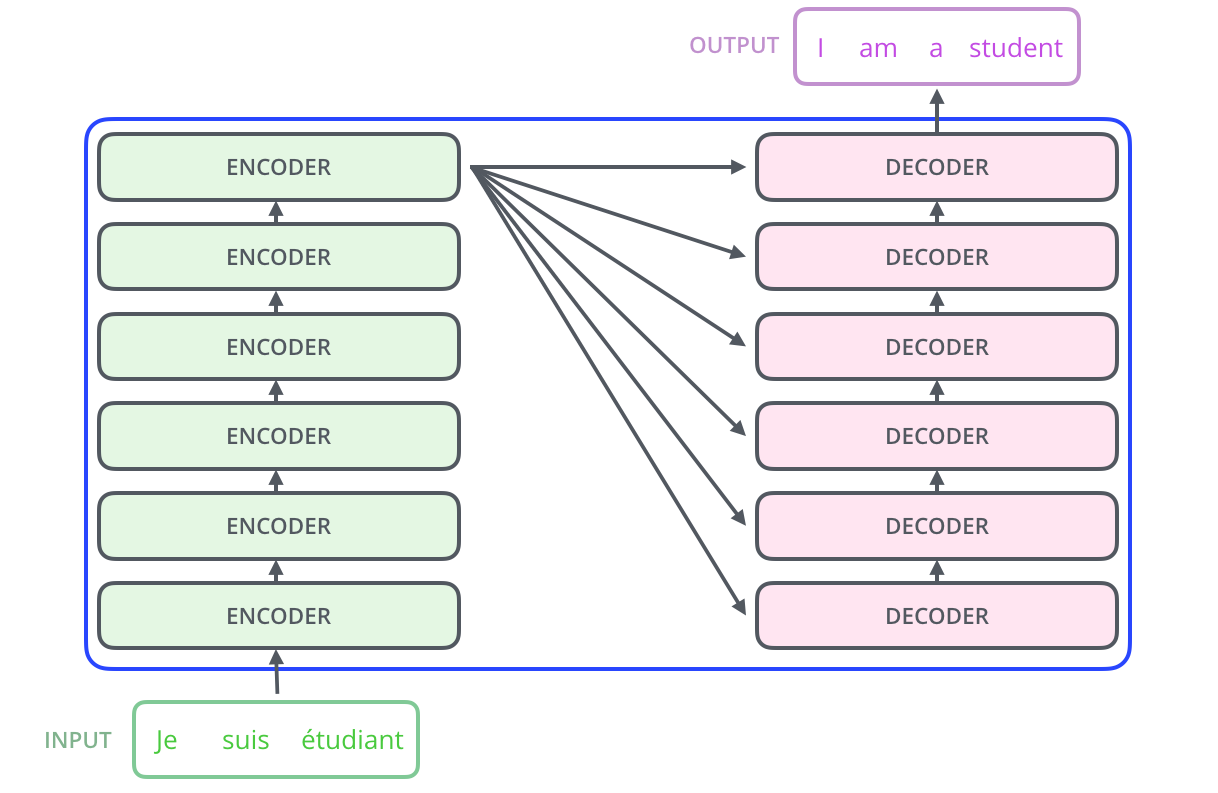
与Encoder的计算类似：

不同的是 的计算使用的是 Encoder 输出的X，而 的计算使用的是经过 Masked Muti-head attention 后的 X。

### Feed Forward 与 Add & Norm

与Encoder一样的处理流程

## 整体流程



### 训练时

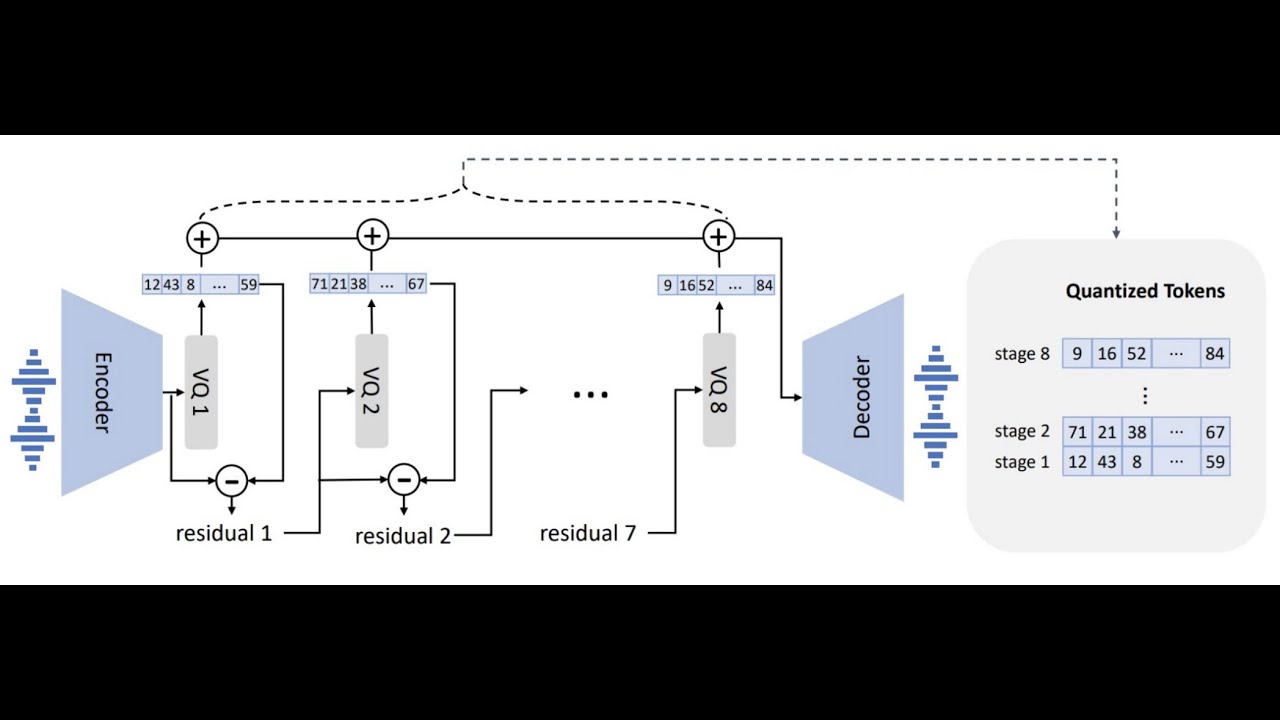
1. 我们依次输入【我，爱，你】，Encoder会输出一个X矩阵。
2. 然后我们输入【<bos>】，经过输入处理后给Decoder，Decoder会输出一个预测的字符 ；接着我们再输入 【I】，decoder预测 ；接着输入【love】，decoder预测 ；接着输入【you】，decoder预测 ；
3. 计算 【】与【I,love,you,<eos>】的损失值
4. 进行梯度下降进行训练

### 测试时

1. 我们依次输入【我，爱，你】，Encoder会输出一个X矩阵。
2. 然后我们输入【<bos>】，经过输入处理后给Decoder，Decoder会输出一个Embeding向量的预测 ；接着我们再输入 ，decoder预测 ；如此循环直到 decoder预测出【<eos>】

# Codec RVQ（Residual Vector Quantization）

RVQ是利用了残差的VQ操作



RVQ一般有8个VQ处理器 ，其伪代码如下：

*for j = 1 to 8*

**codec简要概述**

codec是一个RVQ+AE模型，其用于语音处理

1. Encoder

Encoder接受一段时序为T的音频，经过一系列卷积后输出z：

* C为通道数
* 为经过一系列卷积压缩后的时序长度
* Encoder 操作
* ：RVQ量化操作，以列向量的形式进行量化
* ：K称为AcousticToken（声学token）

有8个行向量，每一行存储特定的信息，如：第一行可能存储音素信息，最后一行存储的可能是语气信息

1. Decoder

Decoder接受K（AcousticToken），经过一系列的卷积，最终生成音频 .

* 利用 codebook 将 AcousticToken 转为
* Decoder 操作