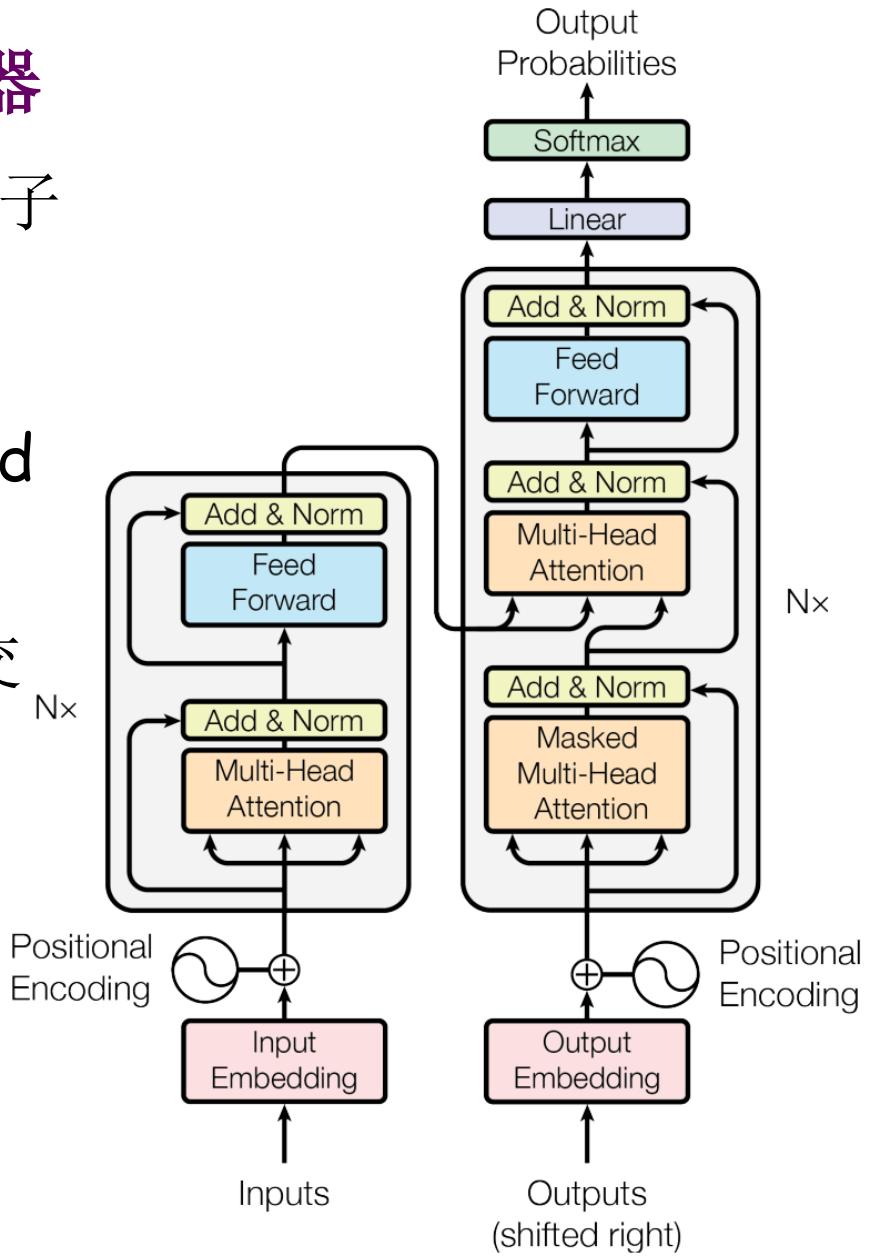
# 解码器

- 一层解码器内部包含3个主要子层，即masked multi-head attention (掩码多头注意力)，multi-head attention， Feed Forward (前馈网络)
- 第二个注意力与编码器进行交互，即不是self-attention
- add & norm操作



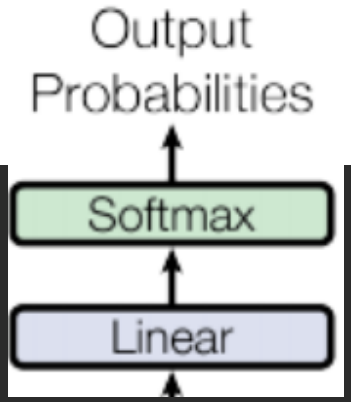
# 解码器实现：堆叠N个block

```
class Decoder(nn.Module):
    "Generic N layer decoder with masking."
    def __init__(self, layer, N):
        # 初始化函数的参数有两个，第一个就是解码器层layer，第二个是解码器层的个数N
        super(Decoder, self).__init__()
        # 首先使用clones方法克隆了N个layer，然后实例化一个规范化层
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, memory, src_mask, tgt_mask):
        # forward函数中的参数有4个，x代表目标数据的嵌入表示，memory是编码器层的输出，
        # source_mask，target_mask代表源数据和目标数据的掩码张量，
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

# 解码器block实现

```python
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size  # size代表词嵌入的维度大小，同时也代表解码器的尺寸，
        self.self_attn = self_attn  # self_attn，多头自注意力对象，也就是说这个注意力机制需要Q=K=V，
        self.src_attn = src_attn  # src_attn,多头注意力对象，这里Q!=K=V
        self.feed_forward = feed_forward
        # 按照结构图使用clones函数克隆三个子层连接对象
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        # 参数：来自上一层的输入x，来自编码器层的语义存储变量memory，以及源数据掩码张量和目标数据掩码张量，
        m = memory  # 将memory表示成m之后方便使用。
        # 将x传入第一个子层结构，因为是自注意力机制，所以Q,K,V都是x. mask
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        # 常规注意力机制，q是输入x; k,v是编码层输出memory，source_mask遮蔽掉对结果没有意义的padding。
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        # 最后一个子层就是前馈全连接子层，经过它的处理后就可以返回结果，这就是我们的解码器结构
        return self.sublayer[2](x, self.feed_forward)
```

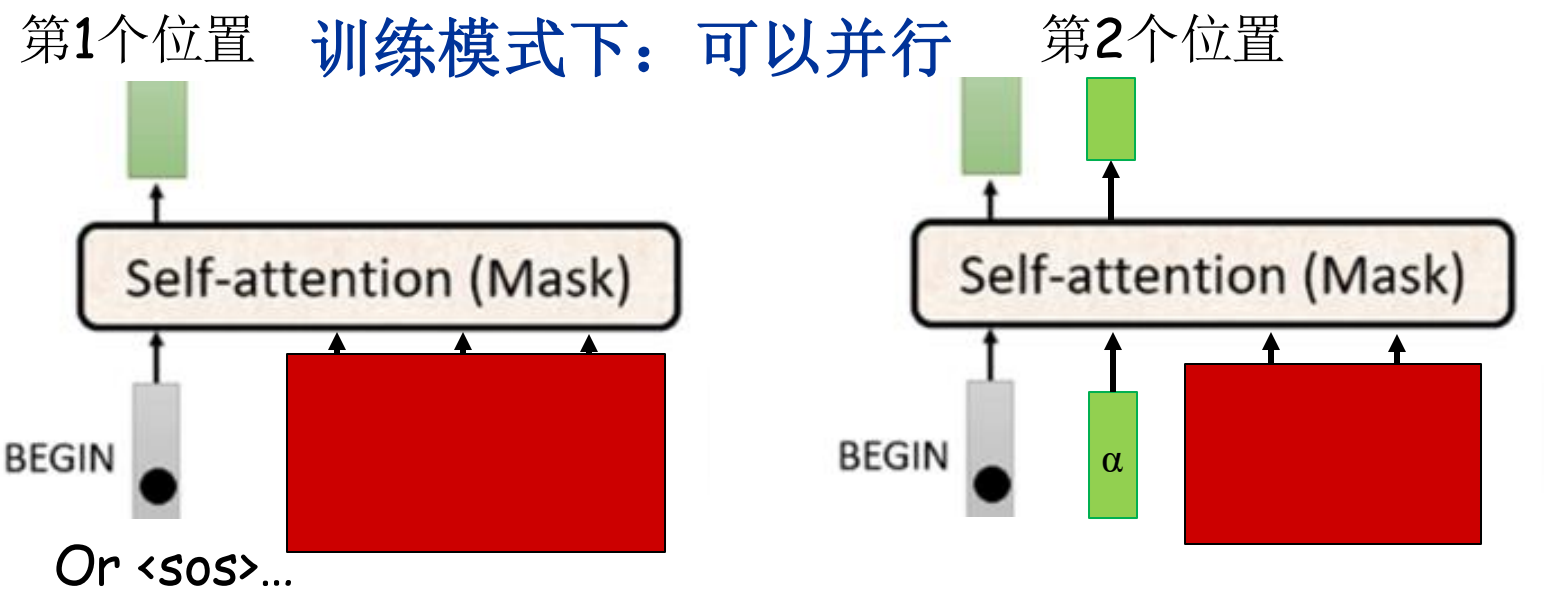
# 解码器预测输出




```python
class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        # d_model代表词嵌入维度，vocab.size代表词表大小
        super(Generator, self).__init__()
        # 首先就是使用nn中的预定义线性层进行实例化，得到一个对象self.proj等待使用
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        # 前向逻辑函数中输入是上一层的输出张量x,在函数中,
        # 首先使用上一步得到的self.proj对x进行线性变化,然后使用F中已经实现的log_softmax进行softmax处理。
        return F.log_softmax(self.proj(x), dim=-1)
```

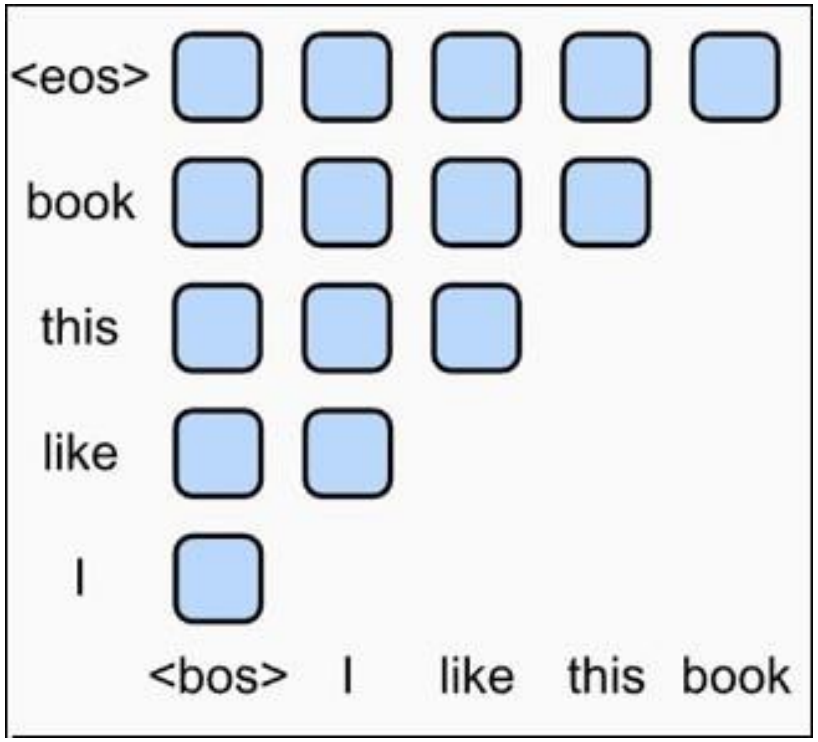
# 掩码注意力

- 推理时，没有完整句子，只有已经生成的部分序列。
- 训练时，一次性输入完整序列，但为了和推理过程保持范围上的一致，在每个位置主动mask。 self-attention只能计算和当前位置、前面位置的相关性，相当于未来要产生的位置被掩藏 (Masked attention)。



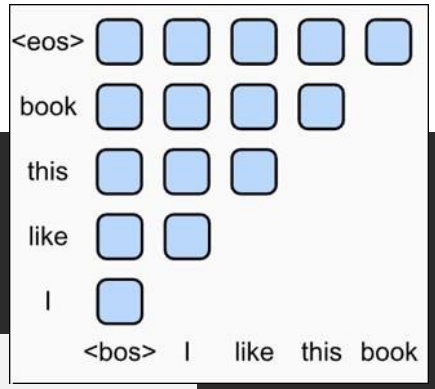
# 掩码注意力

- 掩码后，每个位置能看到的区域形成了一个上三角。




解码时每个时间步的输入token

# 掩码注意力代码实现




```python
def subsequent_mask(size):
    # 生成向后的掩码张量，参数size是掩码张量最后两个维度的大小，它最后两维形成一个方阵

    "Mask out subsequent positions."
    attn_shape = (1, size, size)

    # 然后使用np.ones方法向这个形状中添加1元素，形成上三角阵
    subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')

    # 将numpy转化为tensor，内部做一个1- 操作。其实是做了一个三角阵的反转，subsequent_mask中的每个元素都会被1减。
    # 如果是0，subsequent_mask中的该位置由0变成1
    # 如果是1，subsequent_mask中的该位置由1变成0
    return torch.from_numpy(subsequent_mask) == 0
```

upper triangle
k=1表示对角线的位置上移1个对角线

这样attention函数才知道此处要修改为-1e9

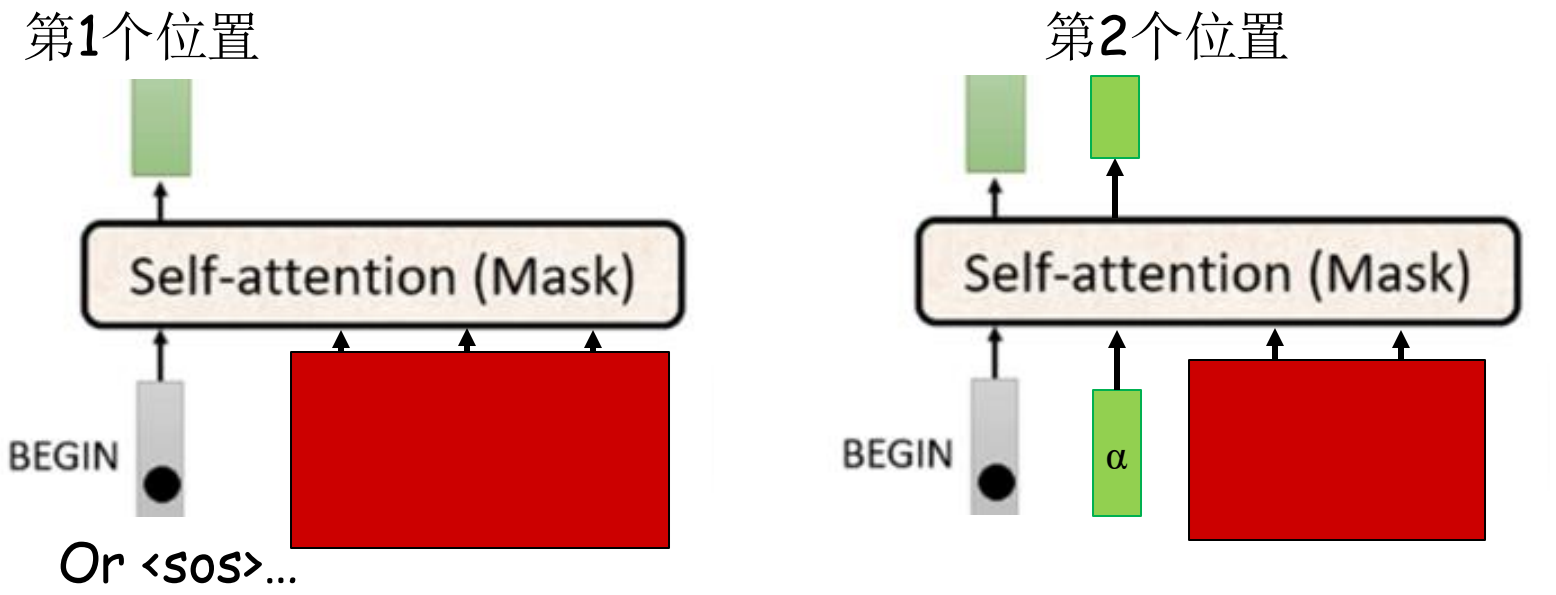
# 掩码注意力

- 思考：和RNN解码相比，掩码自注意力有什么优缺点？

**优点**：训练模型下可并行化

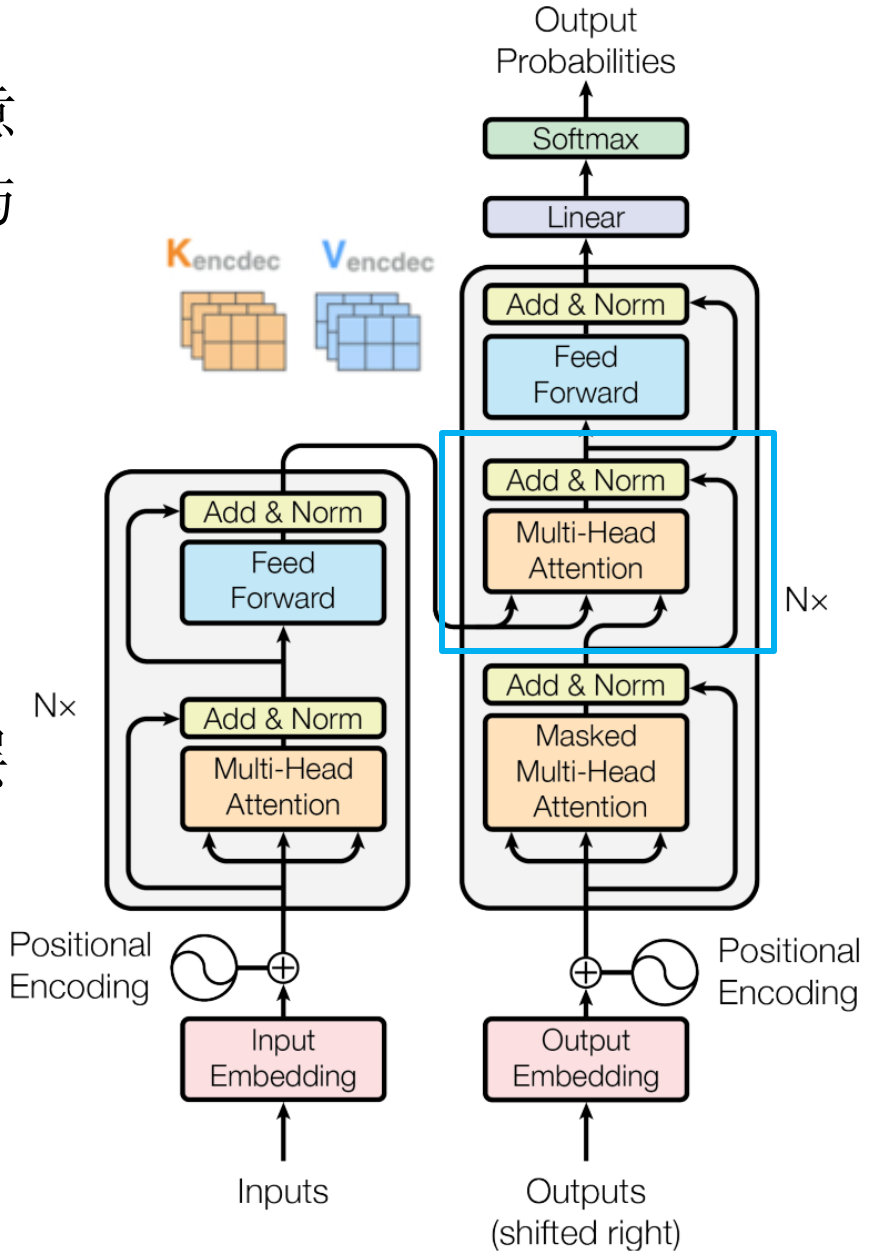
**缺点**：由于teacher forcing，训练模式下每个位置的输出无法结合上一步的预测，即存在暴露偏差（Exposure Bias）

掩码自注意力示意：



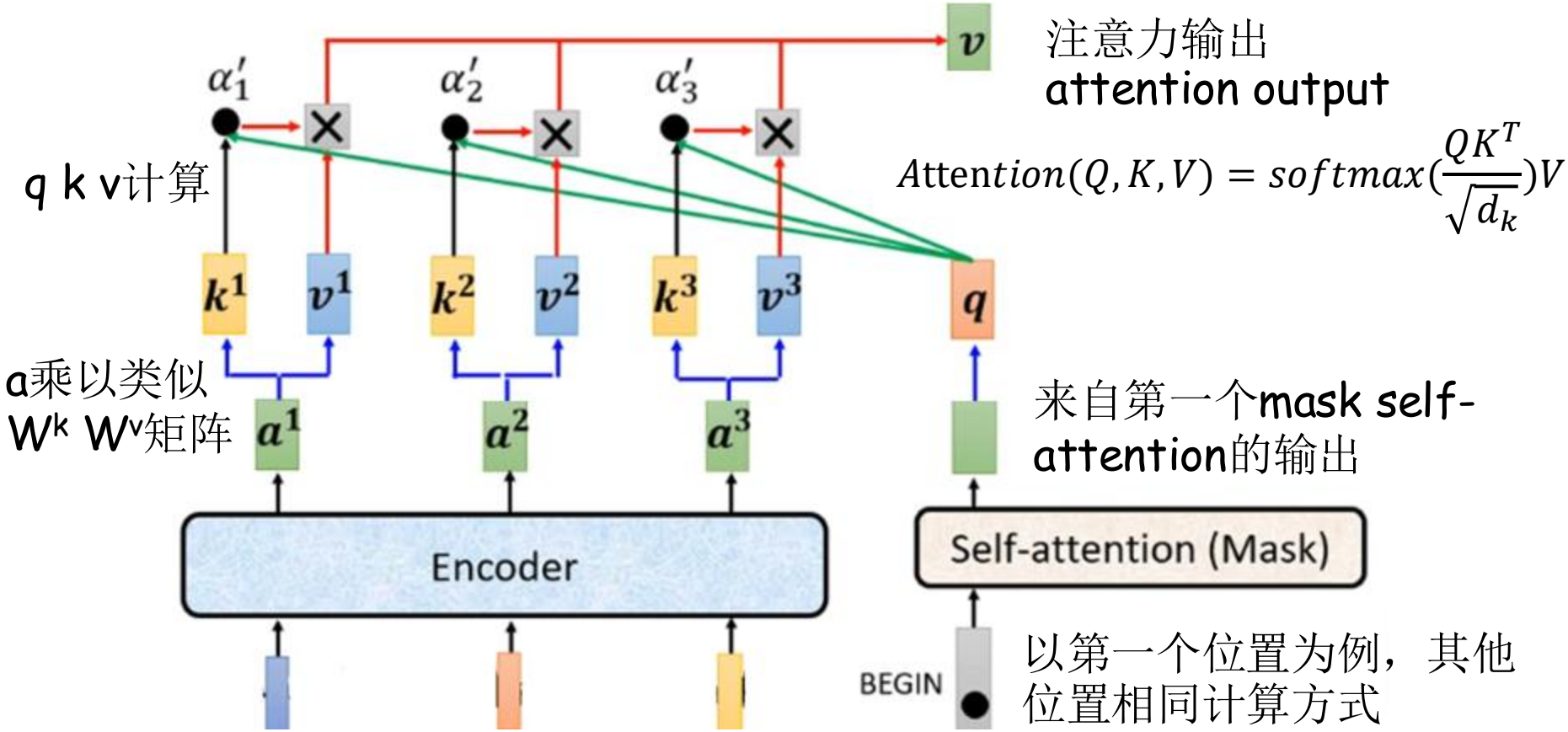
# 交叉注意力

- 每一层解码器中的第二个注意力将第一个注意力的输出，与编码器结果进行交互，又叫 Encoder-decoder attention 或 Cross Attention，即交叉注意力
- K 和 V来自编码器
- Q来自上一个解码注意力子层
- Attention(K, V, Q)计算方式相同



# 交叉注意力

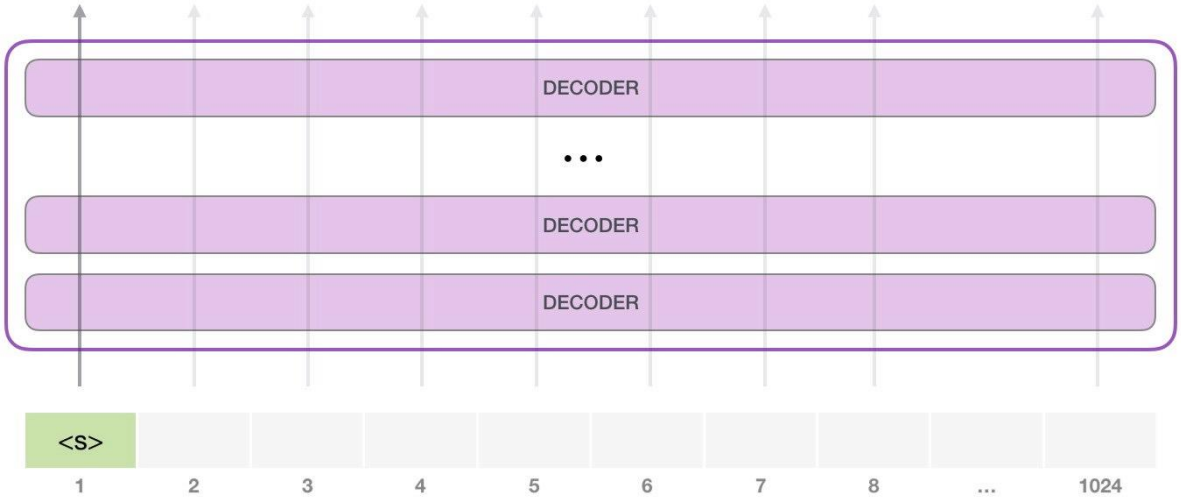
- 在交叉注意力attention(Q, K, V)计算中，Q, K来自编码器输出，V来自第一个mask self-attention子层的输出，因此属于编解码器交互的交叉注意力。



# 解码总过程

训练模式下一步完成；推理模式下分步骤每次预测一个词

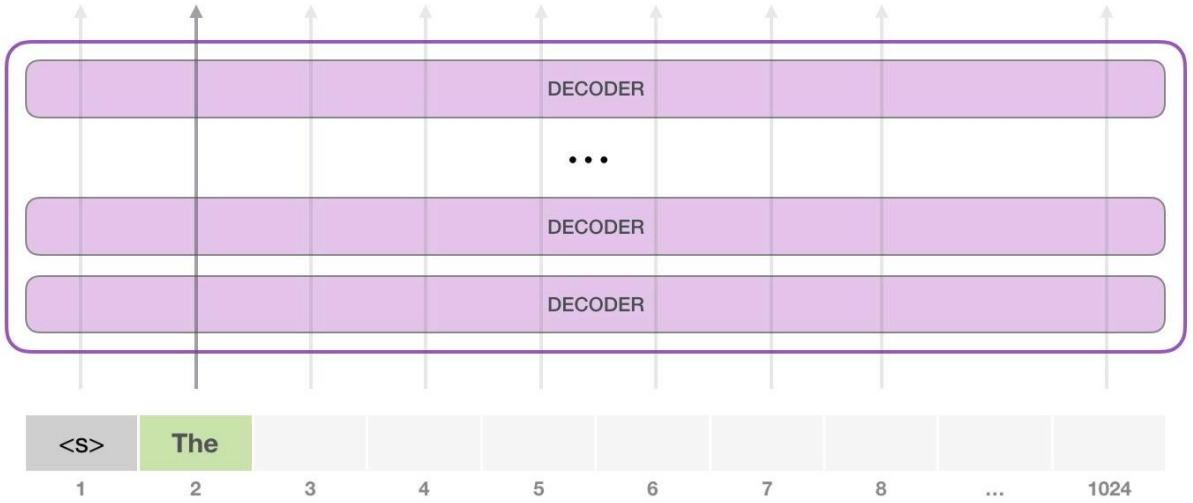
第1步




N层

第1步走完decoder所有模块，产生一个输出词，再接着第2步走完decoder所有模块

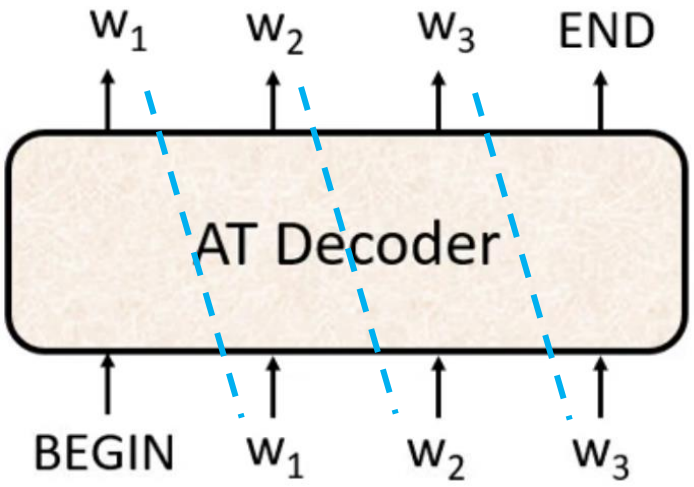
第2步



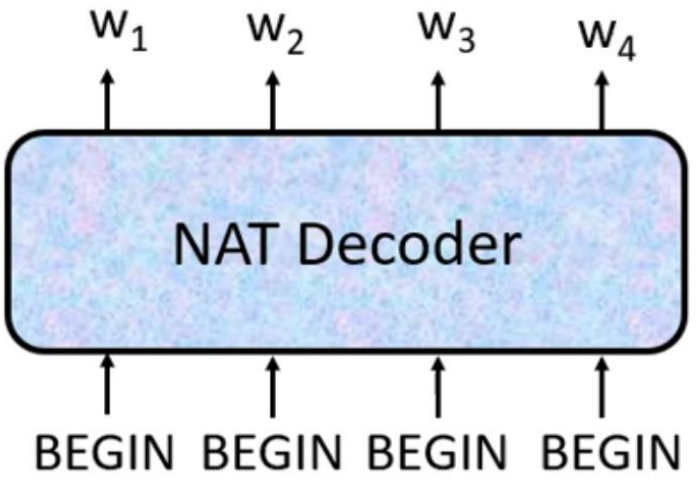

N层

# 解码方式

- 逐词(字)产生输出：自回归 (auto-regressive) (已学方式)
- 同步生成每一个词：非自回归 (non auto-regressive)




Auto-regressive自回归



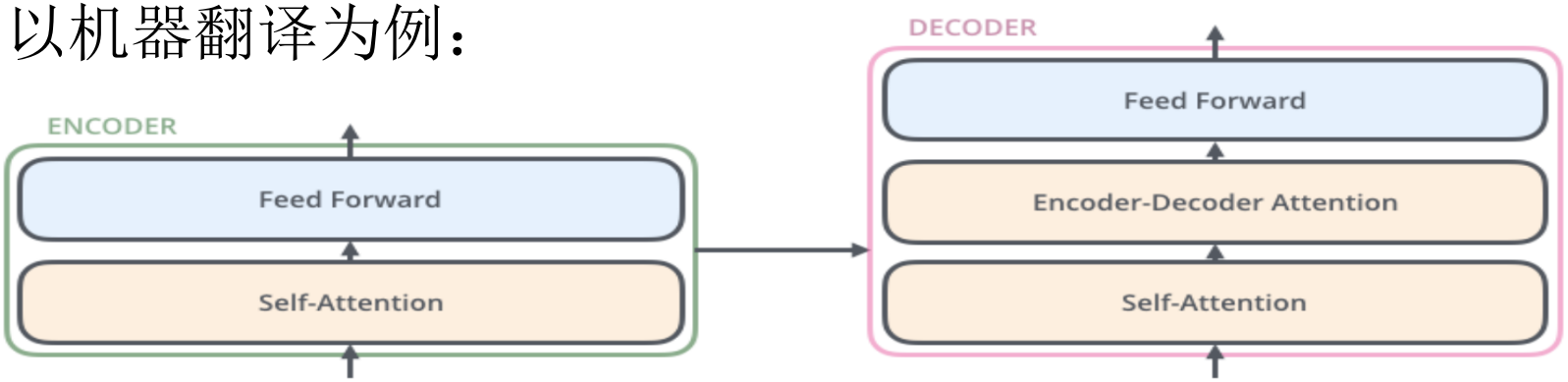

Non Auto-regressive 非自回归

非自回归的解码停止机制：

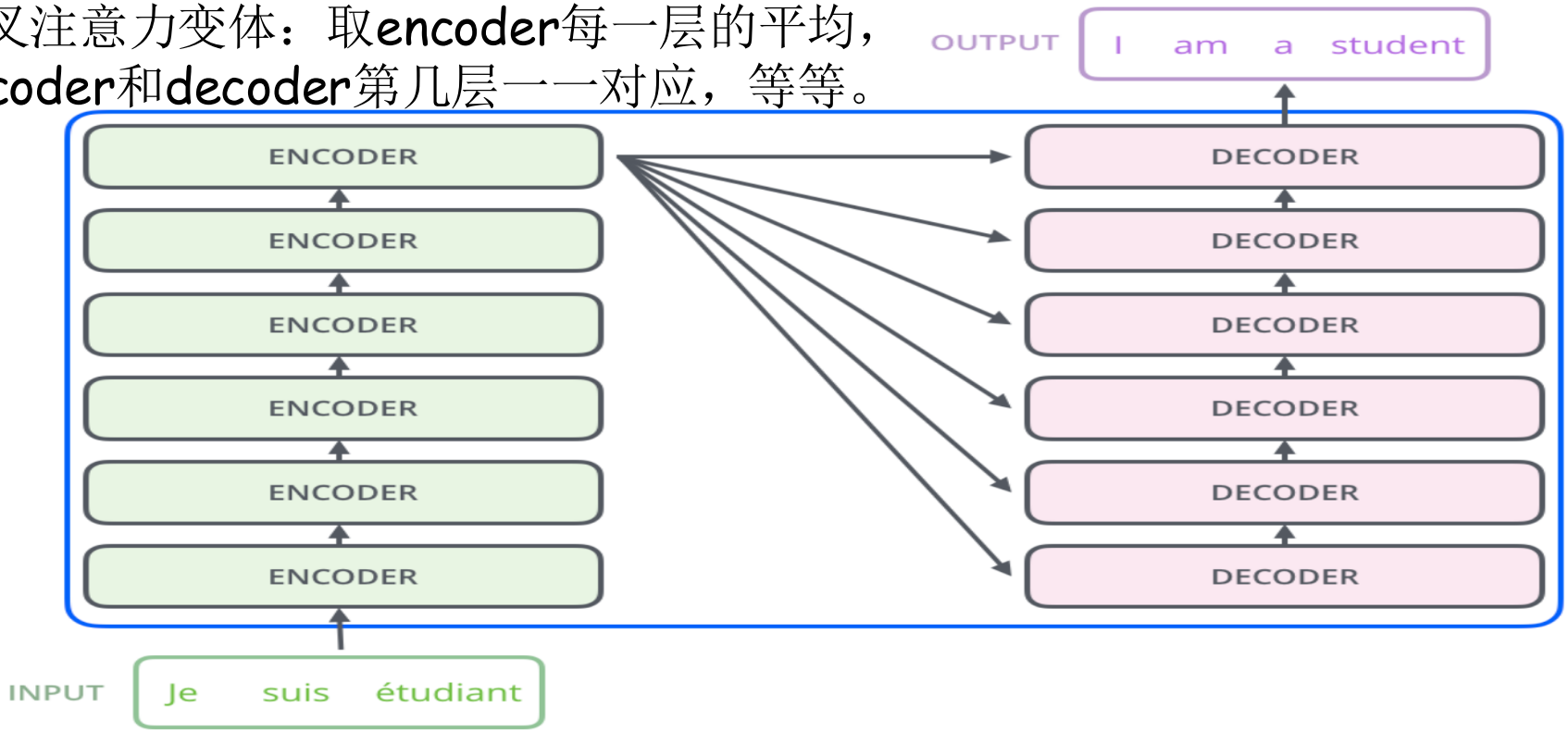
- 输出长度作为一个预测值
- 在第一个EOS位置处截断

# 整体结构

以机器翻译为例：




交叉注意力变体：取encoder每一层的平均，encoder和decoder第几层一一对应，等等。



# Transformer总体结构

```python
class EncoderDecoder(nn.Module):
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed  # input embedding module(token emb + position embed)
        self.tgt_embed = tgt_embed  # output embedding module
        self.generator = generator  # output generation module

    def encode(self, src, src_mask):  # 编码函数
        src_embedds = self.src_embed(src)
        return self.encoder(src_embedds, src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):  # 解码函数
        target_embedds = self.tgt_embed(tgt)
        return self.decoder(target_embedds, memory, src_mask, tgt_mask)

    def forward(self, src, tgt, src_mask, tgt_mask):
        memory = self.encode(src, src_mask)
        res = self.decode(memory, src_mask, tgt, tgt_mask)
        return res
```

# Transformer总体结构

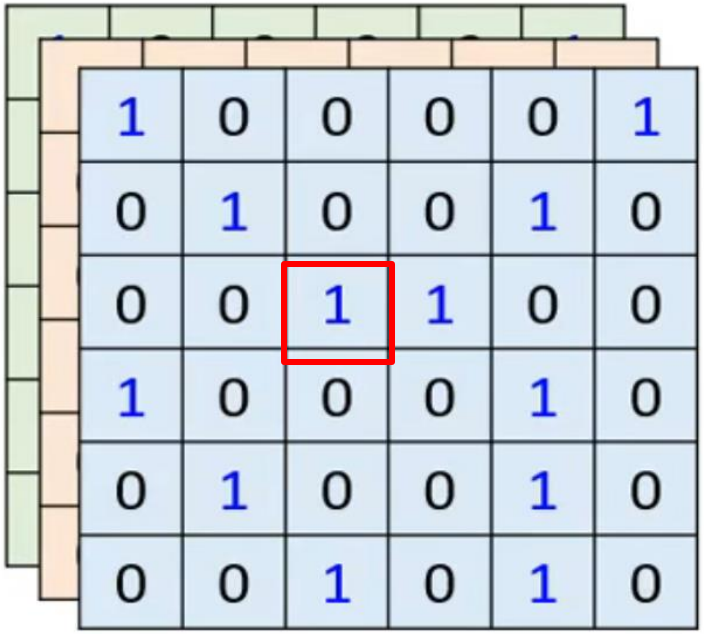
```python
# Full Model
def make_model(src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, dropout=0.1):
    """..."""
    c = copy.deepcopy
    attn = MultiHeadedAttention(h, d_model)
    ff = FeedForward(d_model, d_ff, dropout)
    position = PositionalEncoding(d_model, dropout)
    model = EncoderDecoder(
        Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),  # 对应编码器
        Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff), dropout), N),  # 解码器
        nn.Sequential(Embeddings(d_model, src_vocab), c(position)),  # 源数据嵌入函数
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),  # 目标数据嵌入函数
        Generator(d_model, tgt_vocab))  # 输出部分的类别生成器

    # Initialize parameters with Glorot / fan_avg.
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
    return model
```

# 推荐阅读

- The Annotated Transformer (harvard.edu)
- huggingface/transformers
- Pytorch的Transformer案例
- Tianyang Lin, Yuxin Wang, Xiangyang Liu, Xipeng Qiu (邱锡鹏). *A Survey of Transformers*. 2021. PDF

# 自注意力 vs CNN

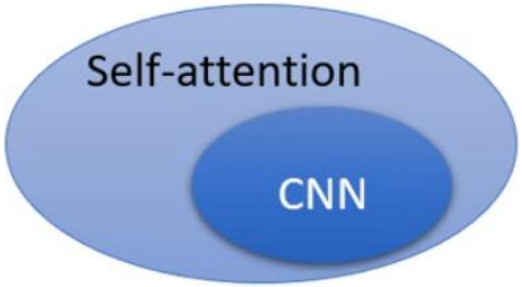
以二维数据X为例



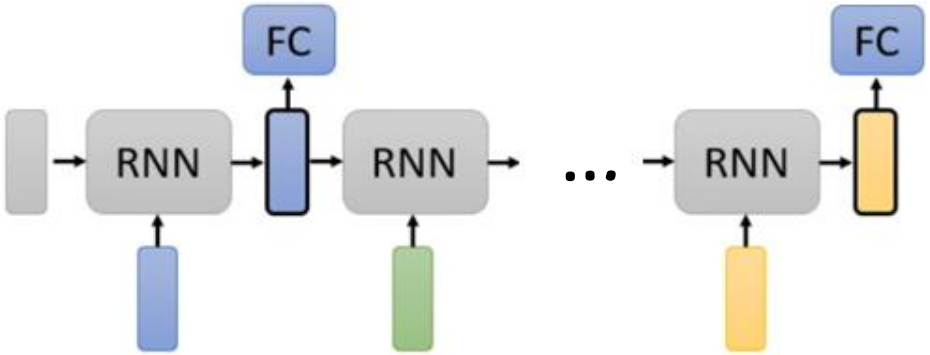

- 做的计算都类似加权和
- 使用卷积：滤波器在X上滑动，一个patch内的数据参与计算
- 使用自注意力：红圈内的数据是query，X内其他数据是values

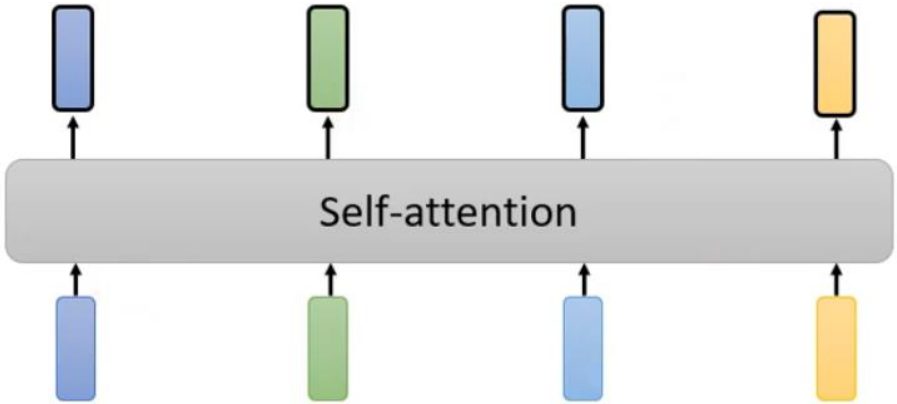
# 自注意力 vs CNN

- CNN ≈ 简化版的**局部**自注意力，滤波器大小是局部范围，在X所有位置使用同一组权重。
- 自注意力 ≈ 复杂版卷积。卷积感受野通过滤波器大小和网络层数计算得出，相当于人为给定，不同位置共享滤波器。自注意力自动学习**动态权重**，权重大即在感受野范围内，相当于**自动计算感受野**。
- 自注意力只要给定特定的参数，可以实现和CNN一样的功能。
- 自注意力是一种更加灵活的CNN，但更加灵活的模型一般需要更多的训练数据进行拟合。



# 自注意力 vs RNN




两者都可以考虑所有输入位置的信息

**RNN:**

- 难以考虑较远位置的信息，不同位置的信息传输不是直接的
- 不能并行化

**自注意力:**

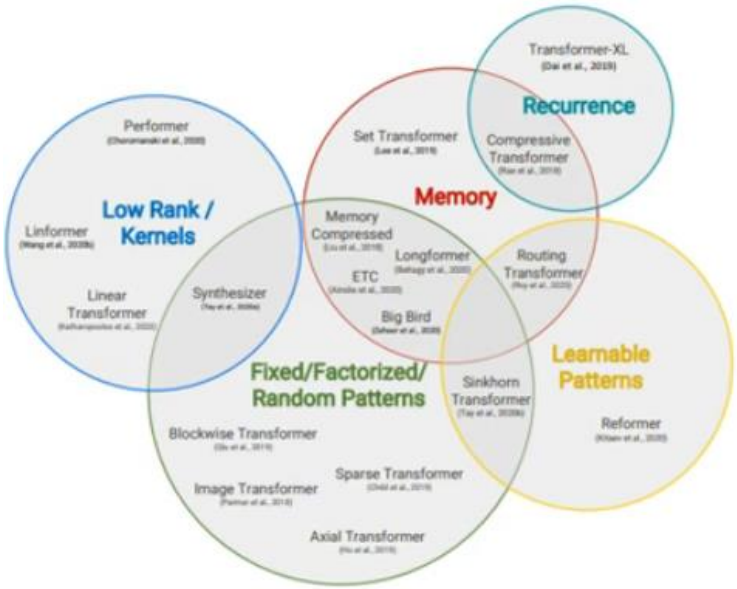
- 两两位置直接进行交互
- 并行化

Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention, 2020, ICML

# Transformer讨论

- Transformer模型做大的方式一般是增加宽度，而不是深度，例如基于Transformer解码器的GPT-3有96层，远少于深层CNN。
- “宽而浅”的模型所需的算力不会比“窄而深”的模型少多少，所以算力并非主要限制。
- 本质原因在于Transformer固有的训练困难。深模型的训练困难源于梯度消失或者梯度爆炸。近年工作指出，深模型训练的根本困难在于“增量爆炸”，在层数变多时，参数的微小变化就会导致损失函数的大变化。

# Transformer变体


## XXformer

- ✓ 基于递归连接的改进: Transformer-XL
- ✓ 基于稀疏注意力的改进: Longformer, Sparse Transformer, Big Bird, Reformer
- ✓ 基于低秩分解的改进: Linformer
- ✓ 基于线性注意力的改进: Linear Transformer, Flash

LONG RANGE ARENA: A BENCHMARK FOR EFFICIENT TRANSFORMERS, 2021, ICLR

Efficient Transformers: A Survey, 2020

A Survey of Transformers, 2021