Transformer

赵洲

浙江大学计算机学院

2022年10月18日

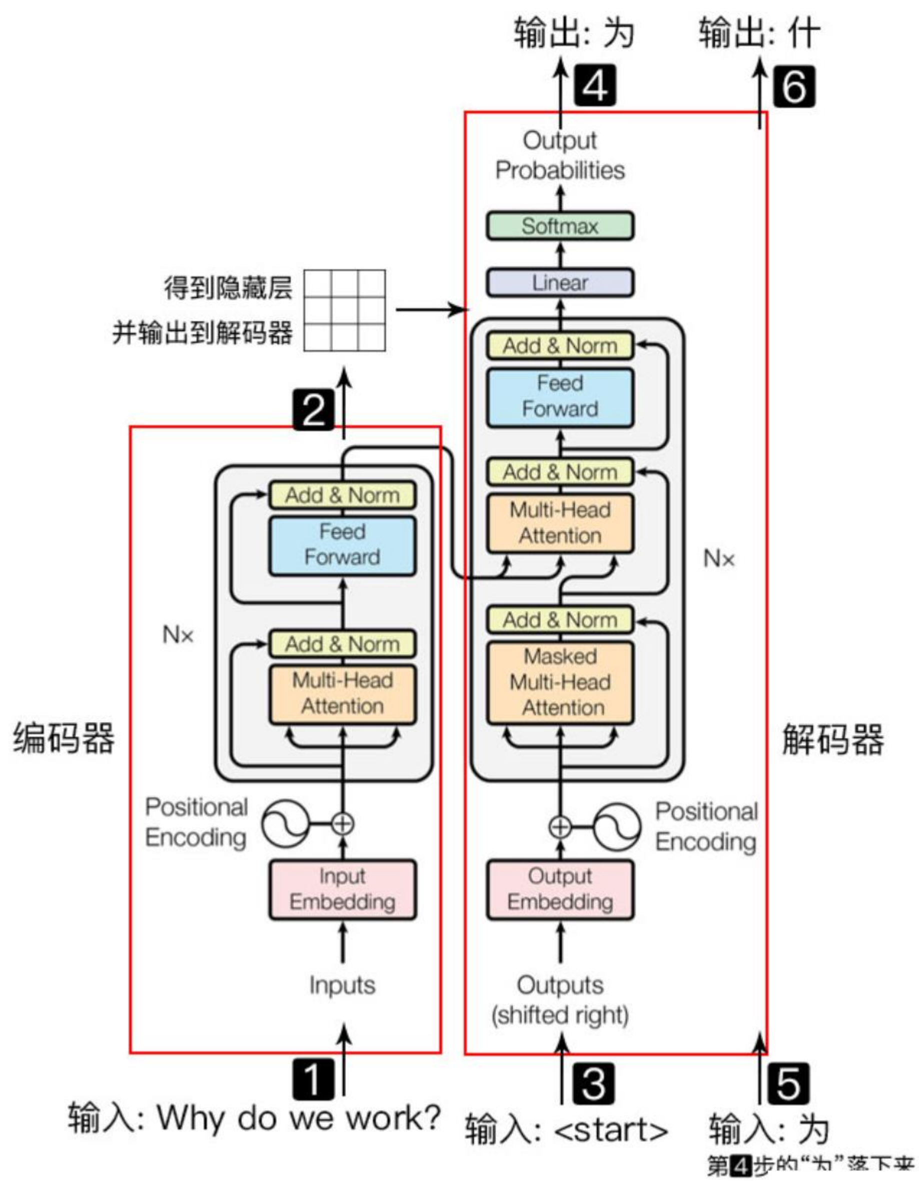
模型结构

- 目前大部分比较热门的神经序列转换模型都有Encoder-Decoder结构。Encoder将输入序列 映射到一个连续表示序列

- 对于编码得到的，Decoder每次解码生成一个符号，直到生成完整的输出序列

- 对于每一步解码，模型都是自回归的，即在生成下一个符号时将先前生成的符号作为附加输入

模型结构图



输出: 为
输出: 什
4
6
Output Probabilities
Softmax
Linear
得到隐藏层
并输出到解码器
2
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
编码器
解码器
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)
1
3
5
输入: Why do we work?
输入: <start>
输入: 为

Attention机制

- 最简单的是QKV模型，假设输入为q, Memory中以(k,v)形式储存需要的上下文。
- 如，k是question, v是answer, q是新来的question, 看看历史中memory中的哪个k更相似，根据相似k对应的v，就是当前的答案。

  - 在memory中找相似 (score function): $e_i = a(q, k_i)$
  - 归一化 (alignment function): $\alpha = softmax(e_i)$
  - 读取内容 (context vector function): $c = \Sigma_i \alpha_i v_i$

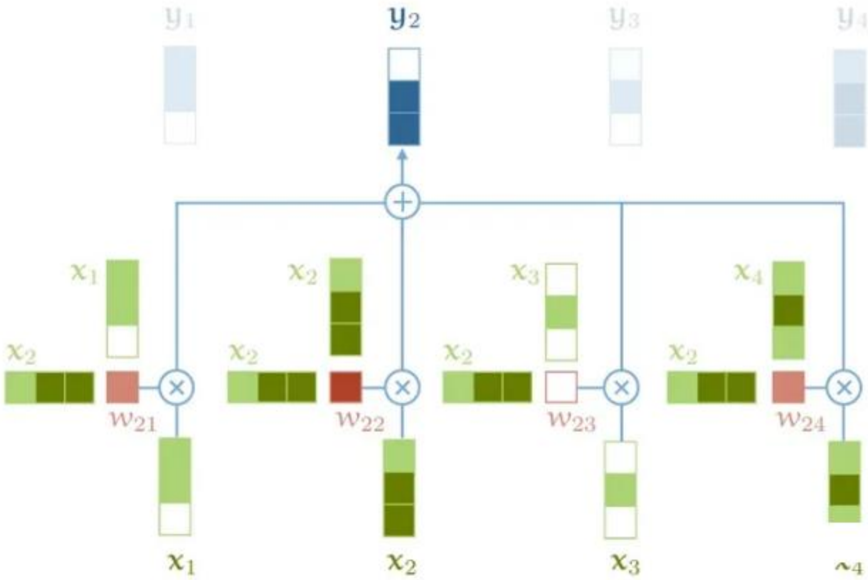
Self-Attention机制

- Self-attention本质上是一种特殊的attention。
- 权重w并不是一个需要神经网络学习的参与，来源于xi和xj的计算结果。

$$w'_{ij} = \mathbf{x}_i^{\top} \mathbf{x}_j \, .$$

- 由于点积输出的取值范围是负无穷和正无穷之间，通过softmax映射到[0,1]

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_j \exp w'_{ij}}$$



QKV聚焦机制

- Attention函数可以将Query和一组Key-Value对映射到输出，其中Query、Key、Value和输出都是向量。 输出是值的加权和，其中分配给每个Value的权重由Query与相应Key的兼容函数计算

- 称这种特殊的Attention机制为"Scaled Dot-Product Attention"。输入包含维度为的Query和Key，以及维度为的Value

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

- 首先分别计算Query与各个Key的点积，然后将每个点积除以，最后使用Softmax函数来获得Key的权重。

Self-Attention实现

```python
import torch
import torch.nn.functional as F

# assume we have some tensor x with size (b, t, k)
x = ...

raw_weights = torch.bmm(x, x.transpose(1, 2))
# - torch.bmm is a batched matrix multiplication. It
#   applies matrix multiplication over batches of
#   matrices.
```

```python
weights = F.softmax(raw_weights, dim=2)
```

```python
y = torch.bmm(weights, x)
```

Self-Attention的Trick 1

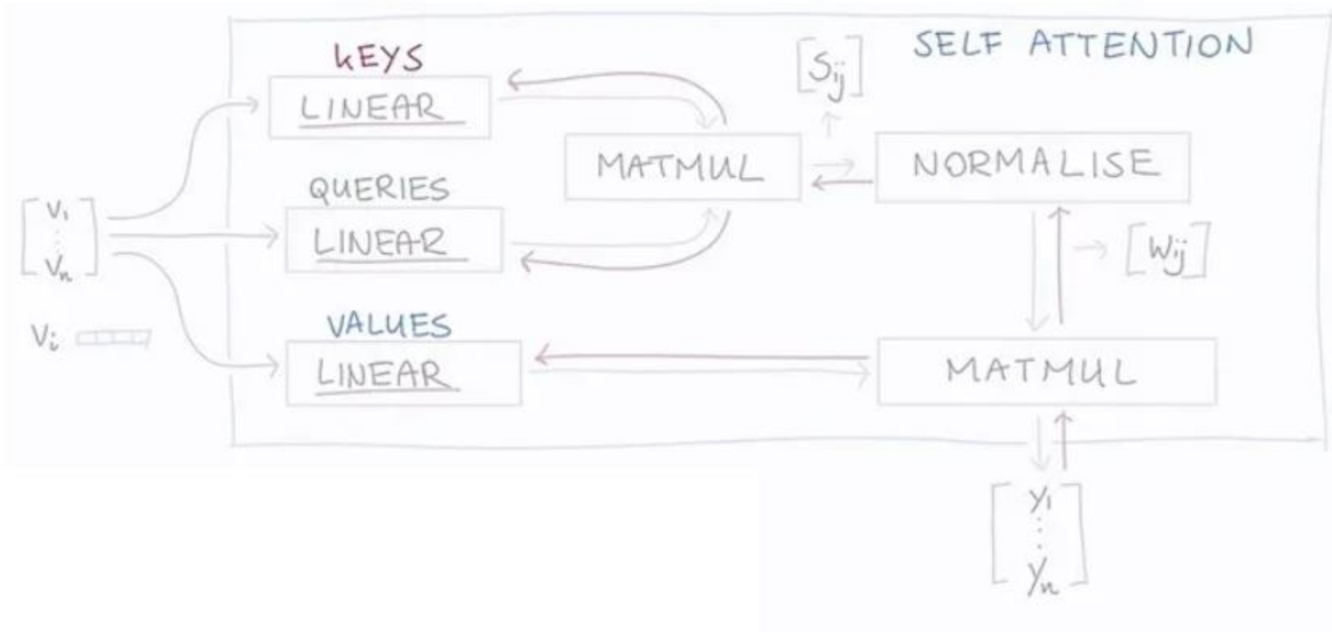
- 针对SA， 加入可学习参数

$$\mathbf{q}_i = \mathbf{W}_q \mathbf{x}_i \qquad \mathbf{k}_i = \mathbf{W}_k \mathbf{x}_i \qquad \mathbf{v}_i = \mathbf{W}_v \mathbf{x}_i$$

$$w'_{ij} = \mathbf{q}_i{}^{\mathsf{T}} \mathbf{k}_j$$

$$w_{ij} = \mathrm{softmax}(w'_{ij})$$

$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{v}_j \,.$$



Self-Attention的Trick 2

- Softmax函数对非常大的输入很敏感，会使得梯度传播出现问题(kill the gradient)，导致学习的速度下降(slow down learning), 甚至导致学习的停止。
- 用\sqrt(k)来对输入的向量做缩放，防止进入到softmax的函数增长过大

$$w'_{ij} = \frac{q_i^{T} k_j}{\sqrt{k}}$$

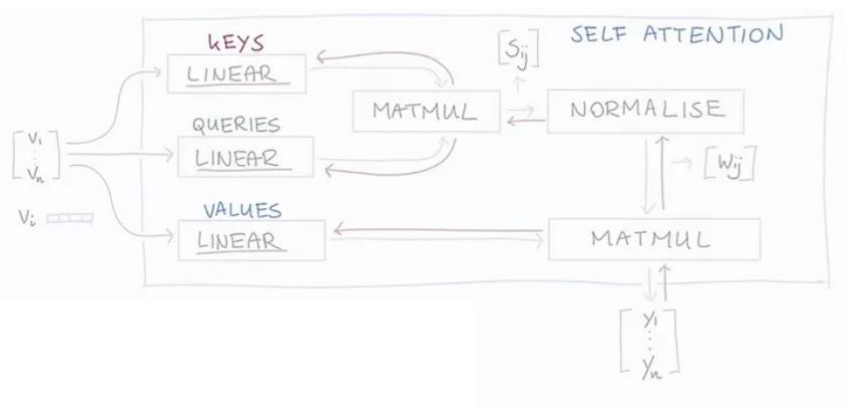
$$\frac{\partial a_j}{\partial z_i} = \frac{\partial}{\partial z_i}\left(\frac{e^{z_j}}{\sum_k e^{z_k}}\right)$$

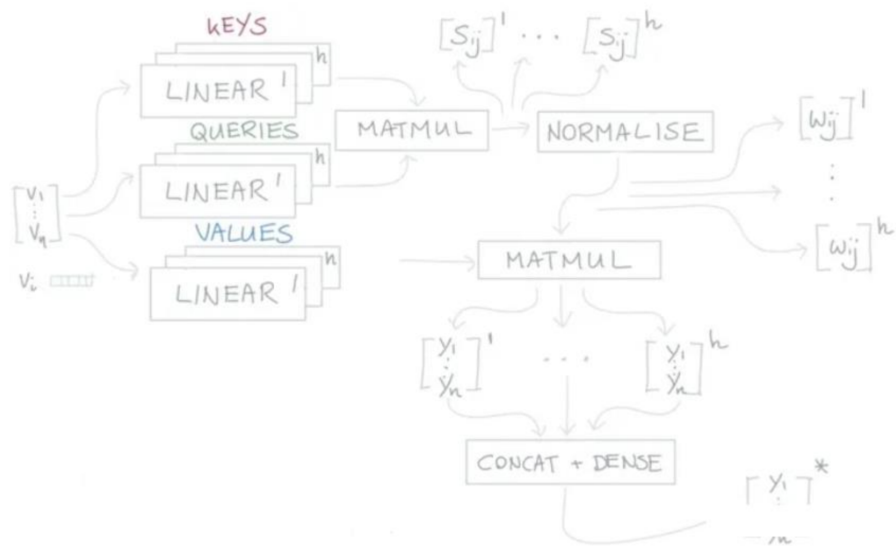
$$= \frac{(e^{z_j})' \cdot \sum_k e^{z_k} - e^{z_j} \cdot e^{z_j}}{\left(\sum_k e^{z_k}\right)^2}$$

$$= \frac{e^{z_j}}{\sum_k e^{z_k}} - \frac{e^{z_j}}{\sum_k e^{z_k}} \cdot \frac{e^{z_j}}{\sum_k e^{z_k}} = a_j(1-a_j)$$

Self-Attention的Trick 3

- 对于输入x每一个attention head都会生成一个向量yi。把这些向量进行concat操作，并且把concat的结果传递给一个全连接层，使得向量的维度重新回到k。





Self-Attention实现例子

1. 准备输入
2. 初始化参数
3. 获取key，query和value
4. 给input1计算attention score
5. 给value乘上score
6. 给value加权求和获取output1
7. 重复步骤4-6，获取output2，output3

准备输入


```
Input 1: [1, 0, 1, 0]
Input 2: [0, 2, 0, 2]
Input 3: [1, 1, 1, 1]
```

初始化参数

key的参数:

```
[[0, 0, 1],
 [1, 1, 0],
 [0, 1, 0],
 [1, 1, 0]]
```

query的参数:

```
[[1, 0, 1],
 [1, 0, 0],
 [0, 0, 1],
 [0, 1, 1]]
```

value的参数:

```
[[0, 2, 0],
 [0, 3, 0],
 [1, 0, 3],
 [1, 1, 0]]
```

获取key，query和value

对于input1的key的表示为：

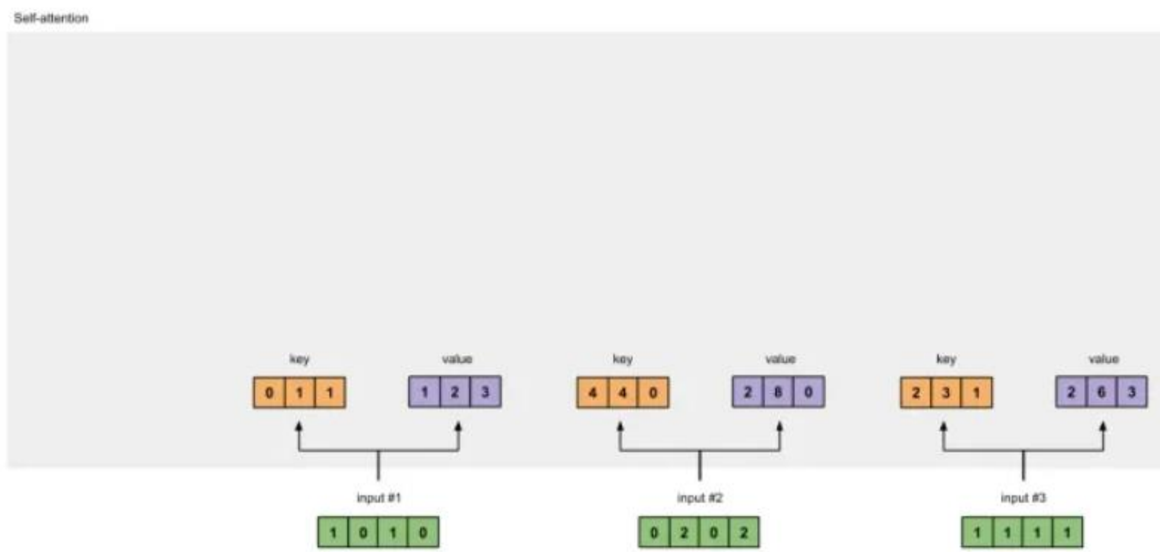
```
               [0, 0, 1]
[1, 0, 1, 0] x [1, 1, 0] = [0, 1, 1]
               [0, 1, 0]
               [1, 1, 0]
```

使用相同的参数获取input2的key的表示：

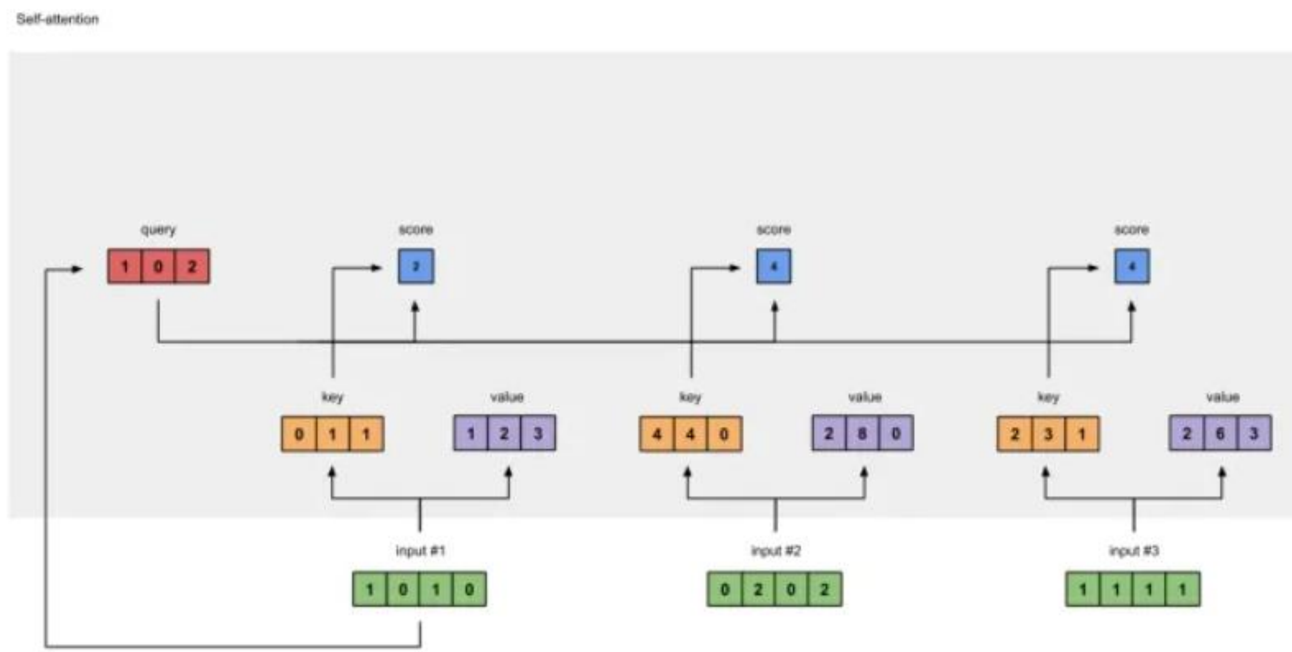
```
               [0, 0, 1]
[0, 2, 0, 2] x [1, 1, 0] = [4, 4, 0]
               [0, 1, 0]
               [1, 1, 0]
```

使用参数获取input3的key的表示：

```
               [0, 0, 1]
[1, 1, 1, 1] x [1, 1, 0] = [2, 3, 1]
               [0, 1, 0]
               [1, 1, 0]
```



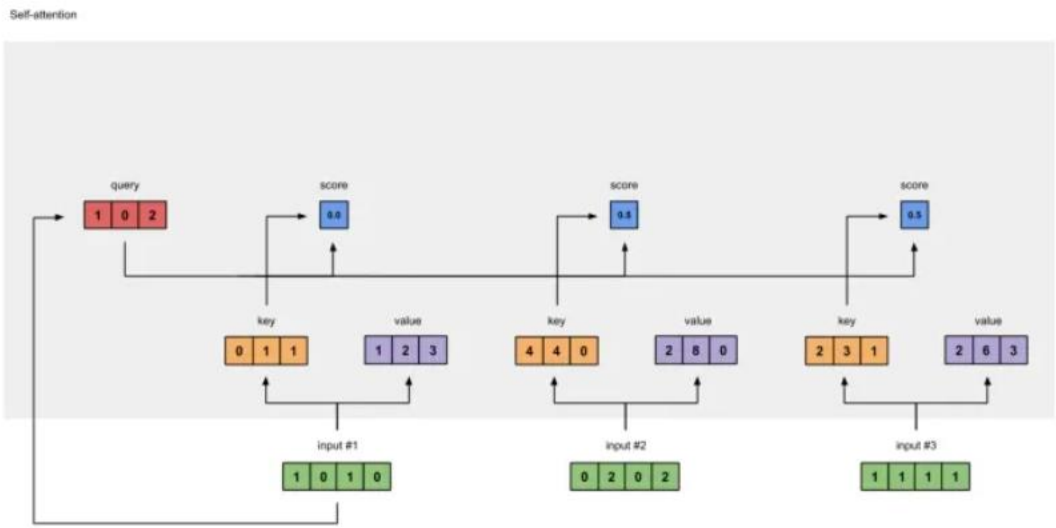
给input1计算attention score



给attention score应用softmax。

```
softmax([2, 4, 4]) = [0.0, 0.5, 0.5]
```

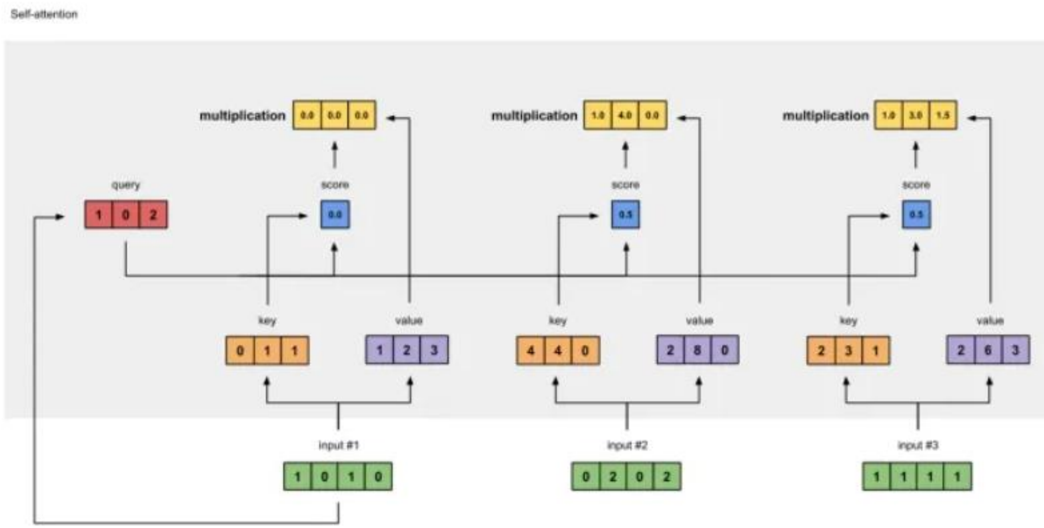
给value乘上score



使用经过softmax后的attention score乘以它对应的value值（紫色），这样我们就得到了3个weighted values（黄色）。

```
1: 0.0 * [1, 2, 3] = [0.0, 0.0, 0.0]
2: 0.5 * [2, 8, 0] = [1.0, 4.0, 0.0]
3: 0.5 * [2, 6, 3] = [1.0, 3.0, 1.5]
```

给value加权求和获取output1



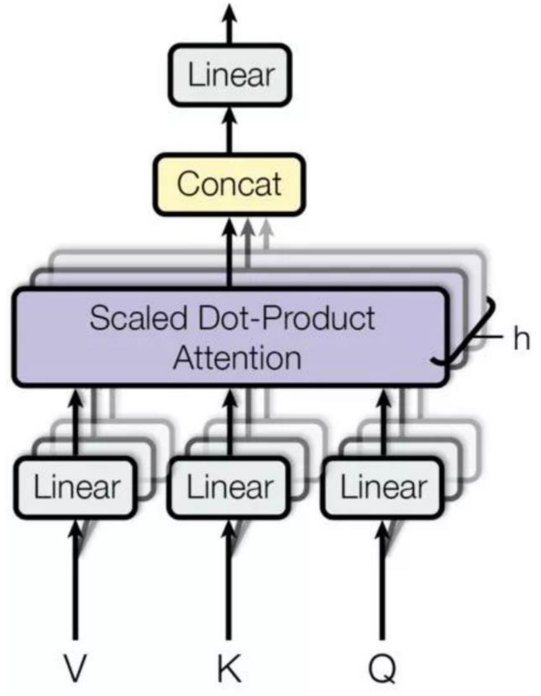
把所有的weighted values（黄色）进行element-wise的相加。

```
  [0.0, 0.0, 0.0]
+ [1.0, 4.0, 0.0]
+ [1.0, 3.0, 1.5]
-----------------
= [2.0, 7.0, 1.5]
```

多头Attention

- “多头”机制能让模型考虑到不同位置的Attention，另外“多头”Attention可以在不同的子空间表示不一样的关联关系，使用单个Head的Attention一般达不到这种效果

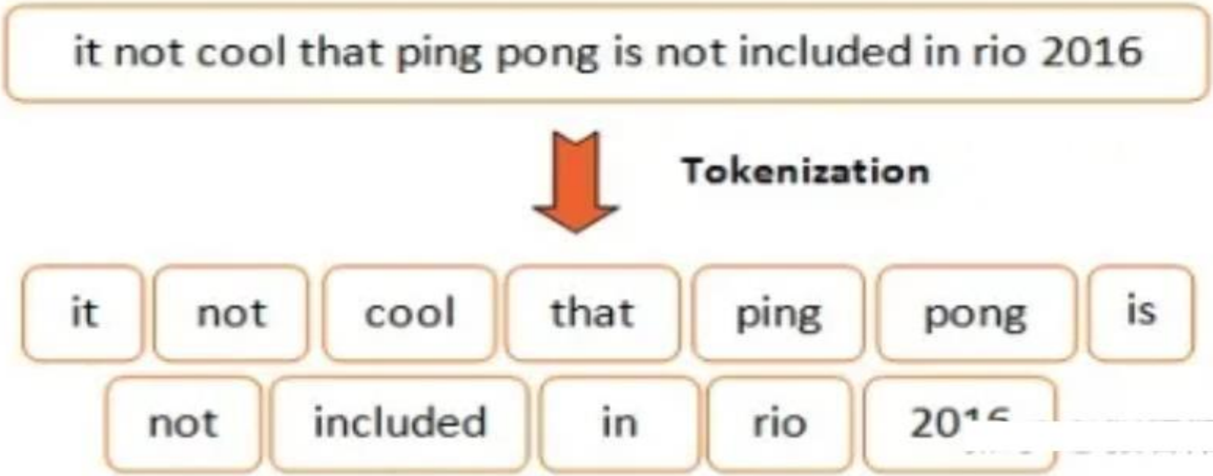
$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1},\ldots,\mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



Transformer模型by Pytorch

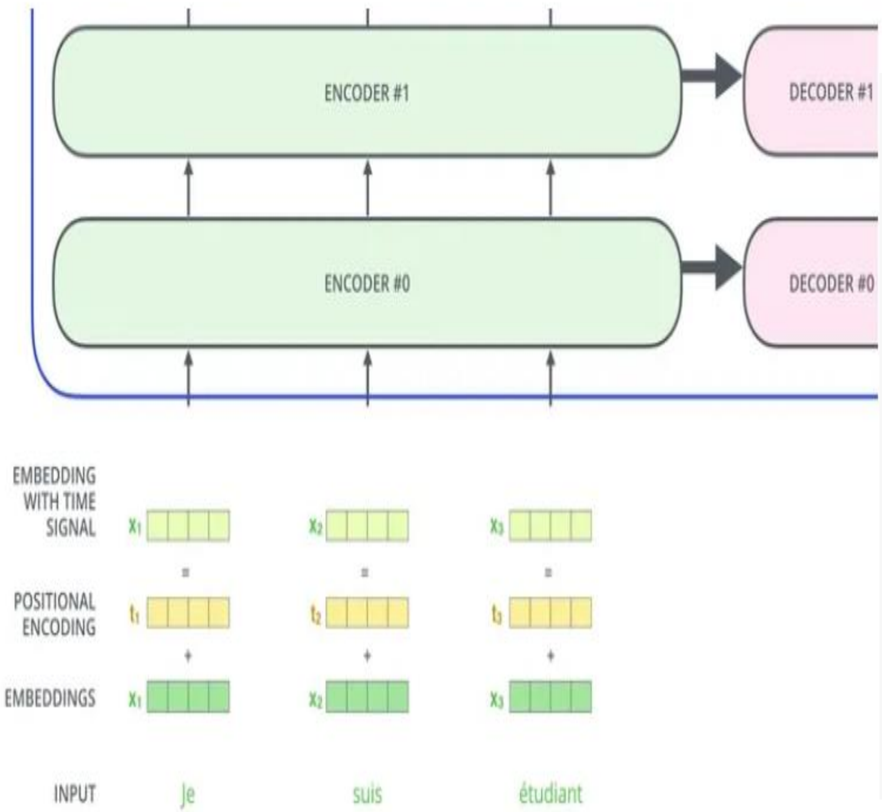
- Tokenize
- Input Embedding
- Positional Encoder
- Transformer Block
- Encoder
- Decoder
- Transformer

Tokenize




```python
class Tokenize(object):

    def __init__(self, lang):
        self.nlp = importlib.import_module(lang).load()

    def tokenizer(self, sentence):
        sentence = re.sub(
        r"[\*\"“”\n\\…\+\-\/\=\(\)‘•:\[\]\|’\!;]", " ", str(sentence))
        sentence = re.sub(r"[ ]+", " ", sentence)
        sentence = re.sub(r"\!+", "!", sentence)
        sentence = re.sub(r"\,+", ",", sentence)
        sentence = re.sub(r"\?+", "?", sentence)
        sentence = sentence.lower()
        return [tok.text for tok in self.nlp.tokenizer(sentence) if tok.text !=
```

Input Embedding

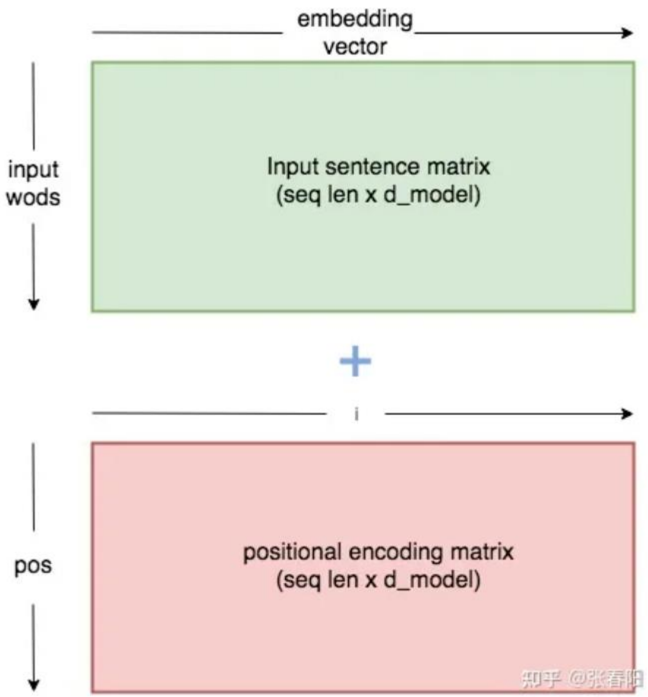



```python
class Embedding(nn.Module):

    def __init__(self, vocab_size, d_model):
        super().__init__()
        self.d_model = d_model
        self.embed = nn.Embedding(vocab_size, d_model)

    def forward(self, x):
        return self.embed(x)
```

Positional Encoder

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

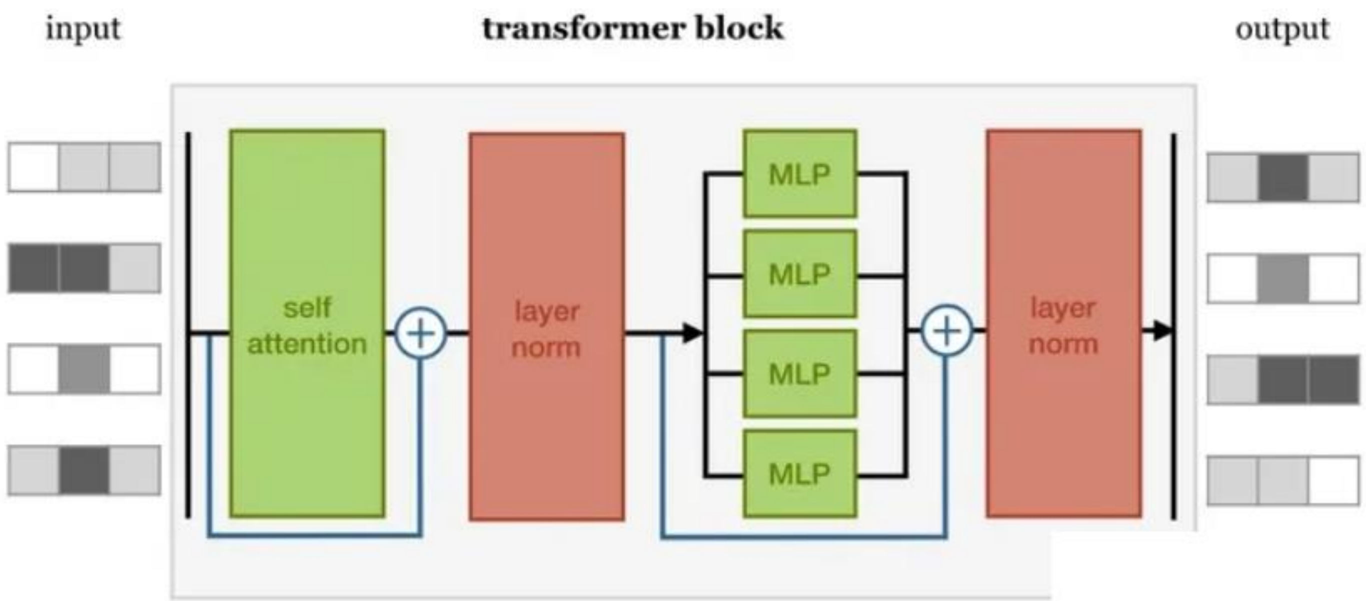
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$




```
class PositionalEncoder(nn.Module):

    def __init__(self, d_model, max_seq_len = 80):
        super().__init__()
        self.d_model = d_model

        # 根据pos和i创建一个常量pe矩阵
        pe = torch.zeros(max_seq_len, d_model)
        for pos in range(max_seq_len):
            for i in range(0, d_model, 2):
                pe[pos, i] = \
                math.sin(pos / (10000 ** ((2 * i)/d_model)))
                pe[pos, i + 1] = \
                math.cos(pos / (10000 ** ((2 * (i + 1))/d_model)))

        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # 让 embeddings vector 相对大一些
        x = x * math.sqrt(self.d_model)
        # 增加位置常量到 embedding 中
        seq_len = x.size(1)
        x = x + Variable(self.pe[:,:seq_len], \
                         requires_grad=False).cuda()
        return x
```

Transformer Block

- Transformer Block 主要是有以下4个部分构成的:
  - **self-attention layer**
  - **normalization layer**
  - **feed forward layer**
  - **another normalization layer**



self-attention layer

```python
def attention(q, k, v, d_k, mask=None, dropout=None):

    scores = torch.matmul(q, k.transpose(-2, -1)) /  math.sqrt(d_k)

    # mask掉那些为了padding长度增加的token，让其通过softmax计算后为0
    if mask is not None:
        mask = mask.unsqueeze(1)
        scores = scores.masked_fill(mask == 0, -1e9)

    scores = F.softmax(scores, dim=-1)

    if dropout is not None:
        scores = dropout(scores)

    output = torch.matmul(scores, v)
    return output
```

MultiHead Attention

- 多头的注意力机制，用来识别数据之间的不同联系

```
class MultiHeadAttention(nn.Module):

    def __init__(self, heads, d_model, dropout = 0.1):
        super().__init__()

        self.d_model = d_model
        self.d_k = d_model // heads
        self.h = heads

        self.q_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)

        self.dropout = nn.Dropout(dropout)
        self.out = nn.Linear(d_model, d_model)

    def forward(self, q, k, v, mask=None):

        bs = q.size(0)

        # perform linear operation and split into N heads
        k = self.k_linear(k).view(bs, -1, self.h, self.d_k)
        q = self.q_linear(q).view(bs, -1, self.h, self.d_k)
        v = self.v_linear(v).view(bs, -1, self.h, self.d_k)

        # transpose to get dimensions bs * N * sl * d_model
        k = k.transpose(1,2)
        q = q.transpose(1,2)
        v = v.transpose(1,2)

        # calculate attention using function we will define next
        scores = attention(q, k, v, self.d_k, mask, self.dropout)
        # concatenate heads and put through final linear layer
        concat = scores.transpose(1,2).contiguous()\
        .view(bs, -1, self.d_model)
        output = self.out(concat)

        return output
```

Layer Norm

- 使用 Layer Norm 来使得梯度更加的平稳

Algorithm 1 Batch Normalization (Every Iteration)

begin Forward Propagation:

Input: $\boldsymbol{X} \in \boldsymbol{R}^{B \times d}$

Output: $\boldsymbol{Y} \in \boldsymbol{R}^{B \times d}$

$\mu_B = \frac{1}{B}\sum_{i=1}^{B} \boldsymbol{x}_i$ `// Get mini-batch mean`

$\sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(\boldsymbol{x}_i - \mu_B)^2$ `// Get mini-batch variance`

$\breve{\boldsymbol{X}} = \frac{\boldsymbol{X} - \mu_B}{\sigma_B}$ `// Normalize`

$\boldsymbol{Y} = \gamma \odot \breve{\boldsymbol{X}} + \beta$ `// Scale and shift`

$\mu = \alpha\mu + (1-\alpha)\mu_B$ `// Update running mean`

$\sigma^2 = \alpha\sigma^2 + (1-\alpha)\sigma_B^2$ `// Update running variance`

begin Backward Propagation:

Input: $\frac{\partial \mathcal{L}}{\partial \boldsymbol{Y}} \in \boldsymbol{R}^{B \times d}$

Output: $\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}} \in \boldsymbol{R}^{B \times d}$

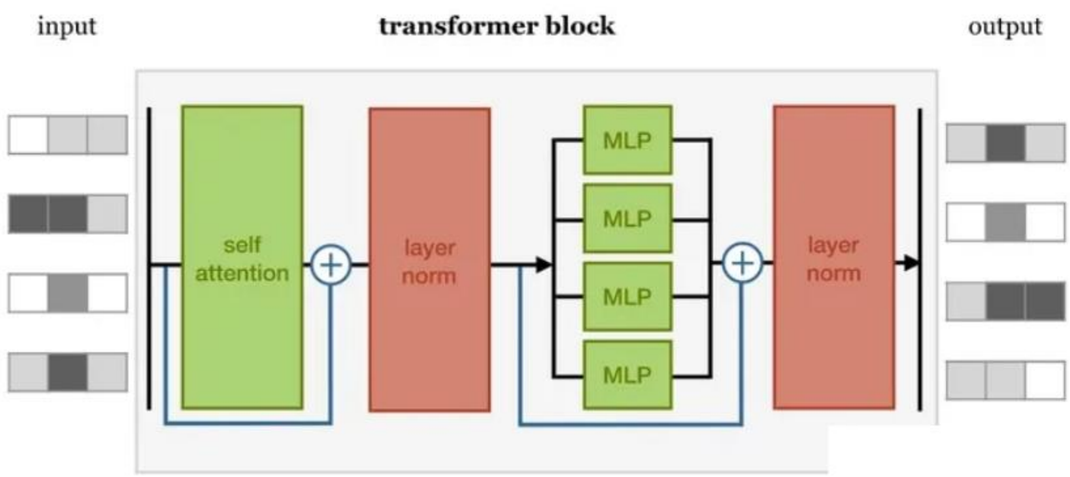
$\frac{\partial \mathcal{L}}{\partial \boldsymbol{X}}$ based on Eq. 3 `// Gradient of` $\boldsymbol{X}$

Inference: $\boldsymbol{Y} = \gamma \odot \frac{\boldsymbol{X} - \mu}{\sigma} + \beta$

```python
class NormLayer(nn.Module):

    def __init__(self, d_model, eps = 1e-6):
        super().__init__()

        self.size = d_model

        # 使用两个可以学习的参数来进行 normalisation
        self.alpha = nn.Parameter(torch.ones(self.size))
        self.bias = nn.Parameter(torch.zeros(self.size))

        self.eps = eps

    def forward(self, x):
        norm = self.alpha * (x - x.mean(dim=-1, keepdim=True)) \
        / (x.std(dim=-1, keepdim=True) + self.eps) + self.bias
        return norm
```

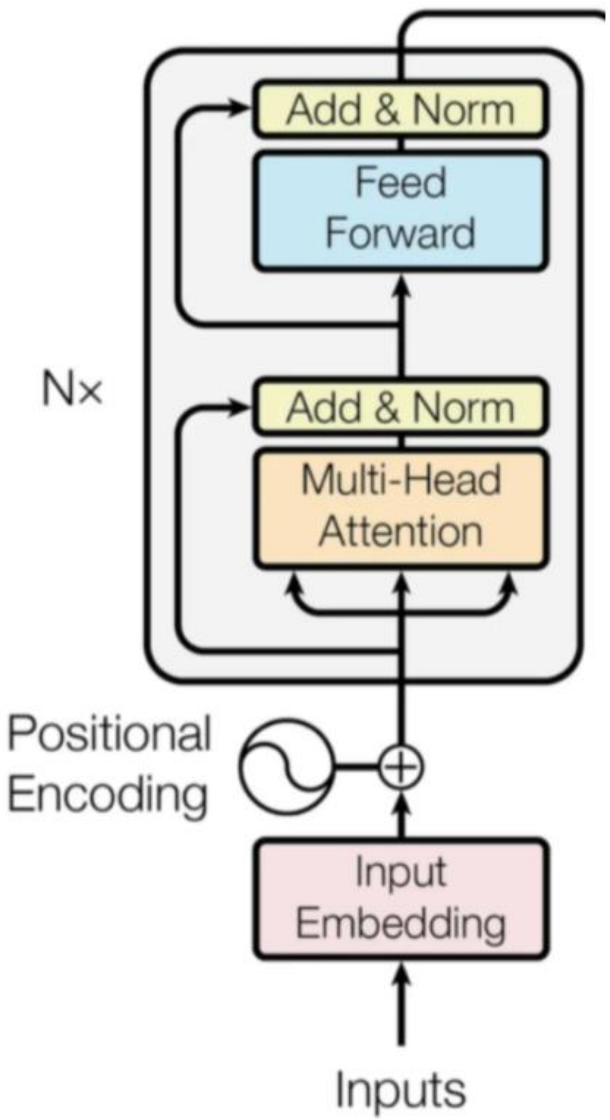
Feed Forward Layer

- 选择4倍输入大小作为我们 feedforward 层的维度，这个值使用的越小就越节省内存，但是相应的表示性也会变弱



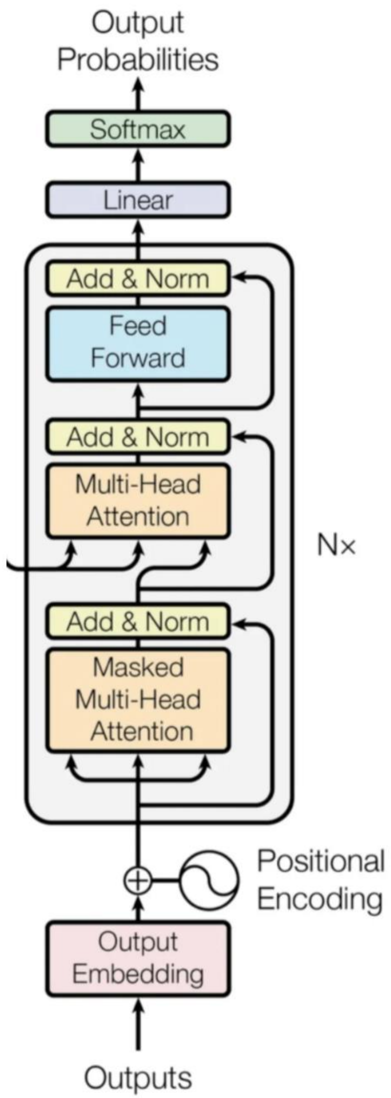

```python
class TransformerBlock(nn.Module):
  def __init__(self, k, heads):
    super().__init__()

    self.attention = SelfAttention(k, heads=heads)

    self.norm1 = nn.LayerNorm(k)
    self.norm2 = nn.LayerNorm(k)

    self.ff = nn.Sequential(
      nn.Linear(k, 4 * k),
      nn.ReLU(),
      nn.Linear(4 * k, k))

  def forward(self, x):
    attended = self.attention(x)
    x = self.norm1(attended + x)

    fedforward = self.ff(x)
    return self.norm2(fedforward + x)
```

Encoder




```python
class EncoderLayer(nn.Module):

    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.attn = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.ff = FeedForward(d_model, dropout=dropout)
        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)

    def forward(self, x, mask):
        x2 = self.norm_1(x)
        x = x + self.dropout_1(self.attn(x2,x2,x2,mask))
        x2 = self.norm_2(x)
        x = x + self.dropout_2(self.ff(x2))
        return x


class Encoder(nn.Module):

    def __init__(self, vocab_size, d_model, N, heads, dropout):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model, dropout=dropout)
        self.layers = get_clones(EncoderLayer(d_model, heads, dropout), N)
        self.norm = Norm(d_model)

    def forward(self, src, mask):
        x = self.embed(src)
        x = self.pe(x)
        for i in range(self.N):
            x = self.layers[i](x, mask)
        return self.norm(x)
```

Decoder

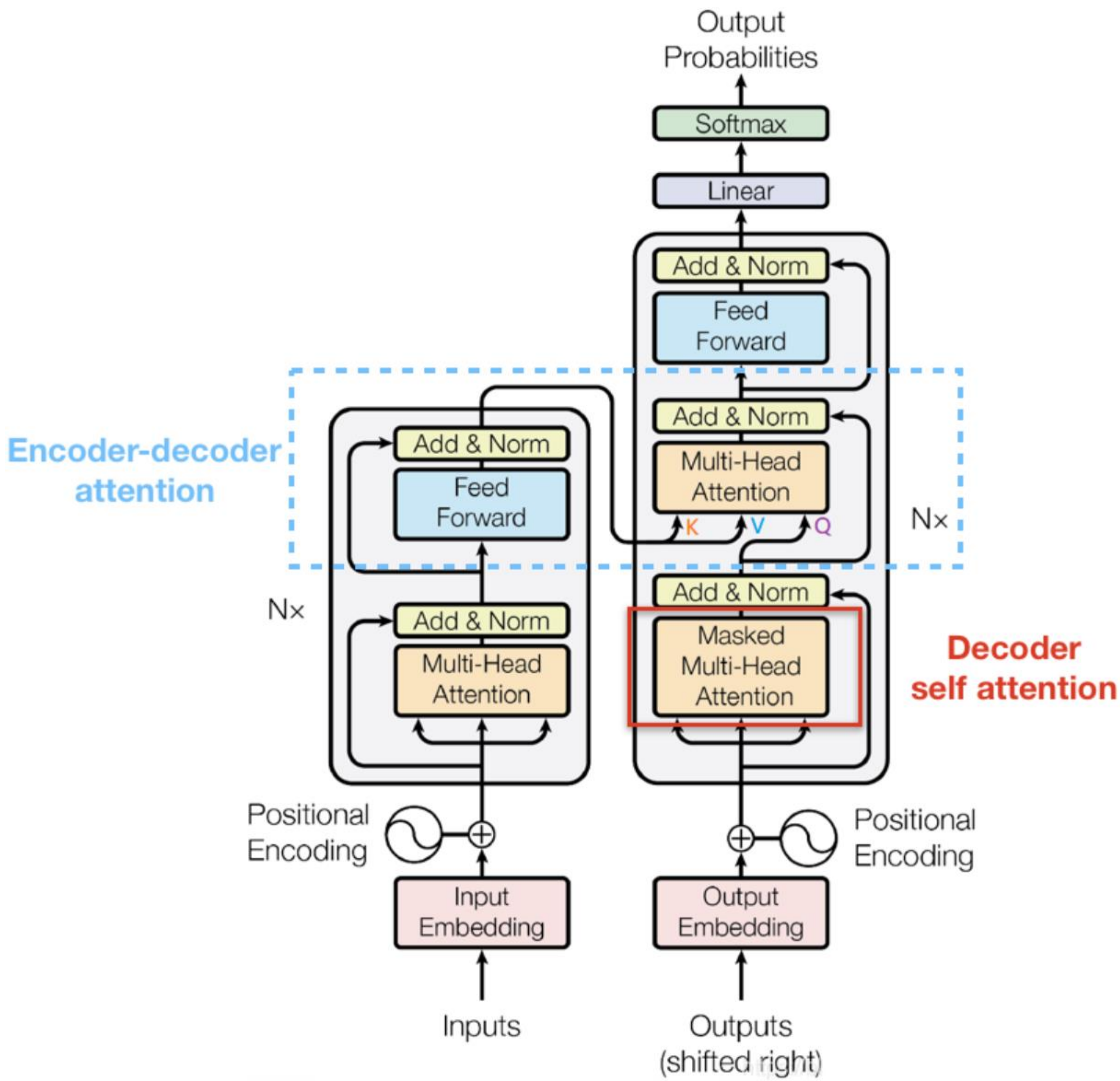



```python
class DecoderLayer(nn.Module):

    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.norm_3 = Norm(d_model)

        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)
        self.dropout_3 = nn.Dropout(dropout)

        self.attn_1 = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.attn_2 = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.ff = FeedForward(d_model, dropout=dropout)

    def forward(self, x, e_outputs, src_mask, trg_mask):
        x2 = self.norm_1(x)
        x = x + self.dropout_1(self.attn_1(x2, x2, x2, trg_mask))
        x2 = self.norm_2(x)
        x = x + self.dropout_2(self.attn_2(x2, e_outputs, e_outputs, \
        src_mask))
        x2 = self.norm_3(x)
        x = x + self.dropout_3(self.ff(x2))
        return x


class Decoder(nn.Module):

    def __init__(self, vocab_size, d_model, N, heads, dropout):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model, dropout=dropout)
        self.layers = get_clones(DecoderLayer(d_model, heads, dropout), N)
        self.norm = Norm(d_model)

    def forward(self, trg, e_outputs, src_mask, trg_mask):
        x = self.embed(trg)
        x = self.pe(x)
        for i in range(self.N):
            x = self.layers[i](x, e_outputs, src_mask, trg_mask)
        return self.norm(x)
```

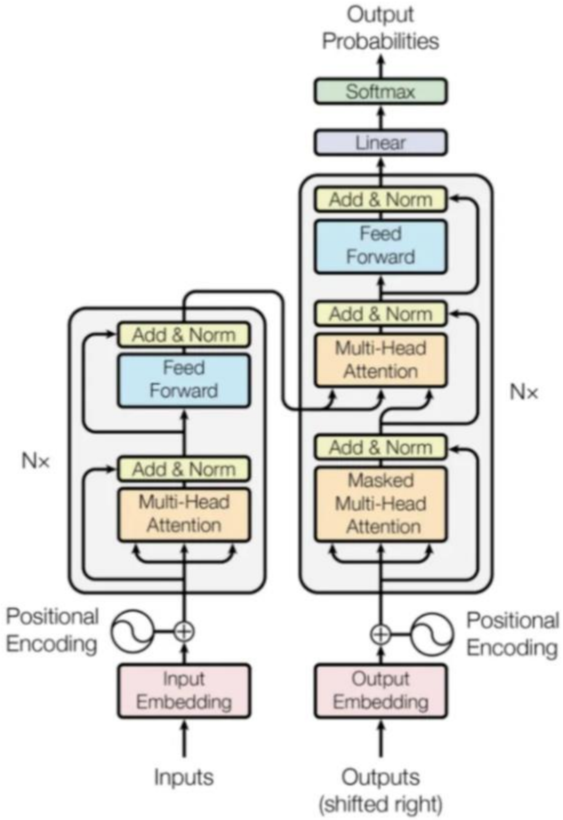
Attention在Transformer中作用

- Encoder self-attention：Encoder 阶段捕获当前 word 和其他输入词的关联
- Masked-Decoder self-attention ：Decoder 阶段捕获当前 word 与已经看到的解码词之间的关联，从矩阵上直观来看就是一个带有 mask 的三角矩阵
- Encoder-Decoder Attention：就是将 Decoder 和 Encoder 输入建立联系，和之前那些普通 Attention 一样

Attention in Decoder

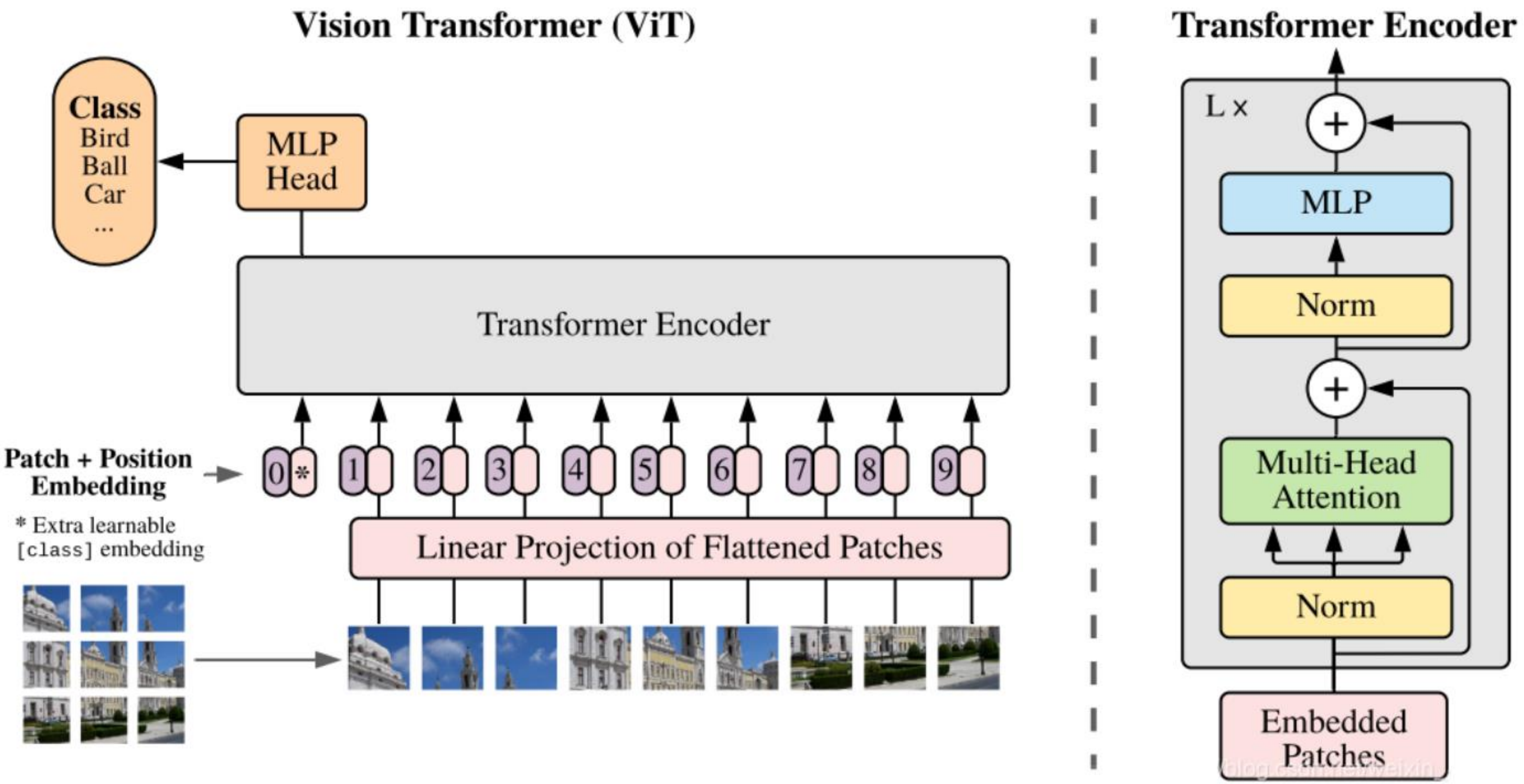


Transformer




```python
class Transformer(nn.Module):

    def __init__(self, src_vocab, trg_vocab, d_model, N, heads, dropout):
        super().__init__()
        self.encoder = Encoder(src_vocab, d_model, N, heads, dropout)
        self.decoder = Decoder(trg_vocab, d_model, N, heads, dropout)
        self.out = nn.Linear(d_model, trg_vocab)

    def forward(self, src, trg, src_mask, trg_mask):
        e_outputs = self.encoder(src, src_mask)
        d_output = self.decoder(trg, e_outputs, src_mask, trg_mask)
        output = self.out(d_output)
        return output
```

Vision Transformer 整体结构



数据处理

- 原始输入的图片数据是H*W*C, 先对图片做分块，再进行展平。假设每个块的长度为(P,P), 那么分块的数目为：

$$N = H * W / (P * P)$$

- 每个图片块展平成一维向量，每个向量大小为：

$$P * P * C$$

- 总的输入变换为：

$$N \times (P^2 \cdot C)$$

Patch Embedding

- 对每个向量都做一个线性变换（即全连接层），压缩维度为D, 称其为Patch Embedding
- 在代码里初始化一个全连接层，输出维度为dim, 然后将分块后的数据输入

```
self.patch_to_embedding = nn.Linear(patch_dim, dim)

# forward前向代码
x = rearrange(img, 'b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=p, p2=p)
x = self.patch_to_embedding(x)
```

Position Encoding

- 位置编码并没有使用传统的Transformer的cos/sin的那套编码方式

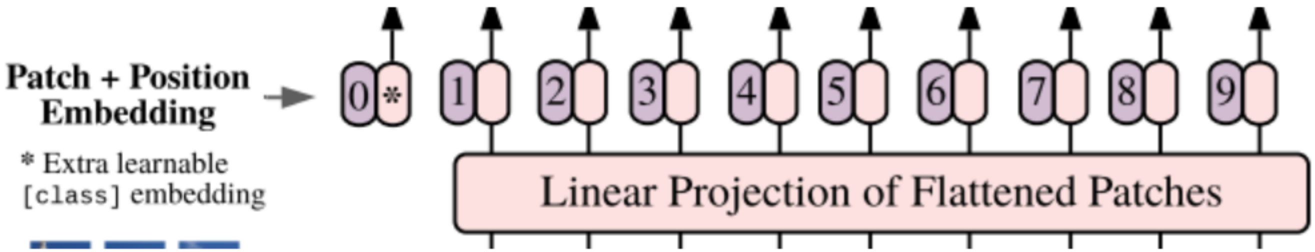
$$
\begin{cases}
PE(pos, 2i) = \sin\left(pos/10000^{2i/d_{\text{model}}}\right) \\
PE(\text{pos}, 2i+1) = \cos\left(pos/10000^{2i/d_{\text{model}}}\right)
\end{cases}
$$

- 而是采用随机初始化，之后再训练出来

```
self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
x += self.pos_embedding[:, :(n + 1)]
```

Cls Token (分类)

- ViT只用了Encoder编码器结构，缺少解码过程，因此给了额外的一个用于分类的向量，与输入进行拼接




```python
# 假设dim=128，这里shape为(1, 1, 128)
self.cls_token = nn.Parameter(torch.randn(1, 1, dim))

# forward前向代码
# 假设batchsize=10，这里shape为(10, 1, 128)
cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
# 跟前面的分块为x（10，64， 128）的进行concat
# 得到（10， 65， 128）向量
x = torch.cat((cls_tokens, x), dim=1)
```

Transformer编码器

- Transformer编码器由多个交互层的多头自注意力和MLP块组成。
- 每个块之前应用LayerNorm, 残差连接在每个块之后应用。
- MLP包含两个呈现GELU非线性层。

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\, \mathbf{x}_p^1\mathbf{E};\, \mathbf{x}_p^2\mathbf{E};\cdots;\, \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2\cdot C)\times D},\, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D}$$

$$\mathbf{z}'_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1\ldots L$$

$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \qquad \ell = 1\ldots L$$

$$\mathbf{y} = \mathrm{LN}(\mathbf{z}_L^0)$$

分类

- 分类头加入LayerNorm和两层全连接层实现，采用GELU激活函数
- 最终分类我们只取第一个，也就是用于分类的token, 输入到分类头里，得到分类结果

```
self.mlp_head = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, mlp_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(mlp_dim, num_classes)
        )
```

```
self.to_cls_token = nn.Identity()
# forward前向部分
x = self.transformer(x, mask)
x = self.to_cls_token(x[:, 0])
return self.mlp_head(x)
```

实验部分

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Table 1: Details of Vision Transformer model variants.

	Ours-JFT (ViT-H/14)	Ours-JFT (ViT-L/16)	Ours-I21K (ViT-L/16)	BiT-L (ResNet152x4)	Noisy Student (EfficientNet-L2)
ImageNet	**88.55** ± 0.04	87.76 ± 0.03	85.30 ± 0.02	87.54 ± 0.02	88.4/88.5*
ImageNet ReaL	**90.72** ± 0.05	90.54 ± 0.03	88.62 ± 0.05	90.54	90.55
CIFAR-10	**99.50** ± 0.06	99.42 ± 0.03	99.15 ± 0.03	99.37 ± 0.06	–
CIFAR-100	**94.55** ± 0.04	93.90 ± 0.05	93.25 ± 0.05	93.51 ± 0.08	–
Oxford-IIIT Pets	**97.56** ± 0.03	97.32 ± 0.11	94.67 ± 0.15	96.62 ± 0.23	–
Oxford Flowers-102	99.68 ± 0.02	**99.74** ± 0.00	99.61 ± 0.02	99.63 ± 0.03	–
VTAB (19 tasks)	**77.63** ± 0.23	76.28 ± 0.46	72.72 ± 0.21	76.29 ± 1.70	–
TPUv3-core-days	2.5k	0.68k	0.23k	9.9k	12.3k