AI 大模型开发工程师之大模型核心之算法

讲师：李希沅

从代码层面理解Transformer

模型	结构	位置编码	激活函数	layer norm方法
原生 Transformer	Encoder-Decoder	Sinusoidal编码	ReLU	Post layer norm
BERT	Encoder	绝对位置编码	GeLU	Post layer norm
LLaMA	Casual decoder	RoPE	SwiGLU	Pre RMS Norm
ChatGLM-6B	Prefix decoder	RoPE	GeGLU	Post Deep Norm
Bloom	Casual decoder	ALiBi	GeLU	Pre Layer Norm

Pytyon

Pytorch

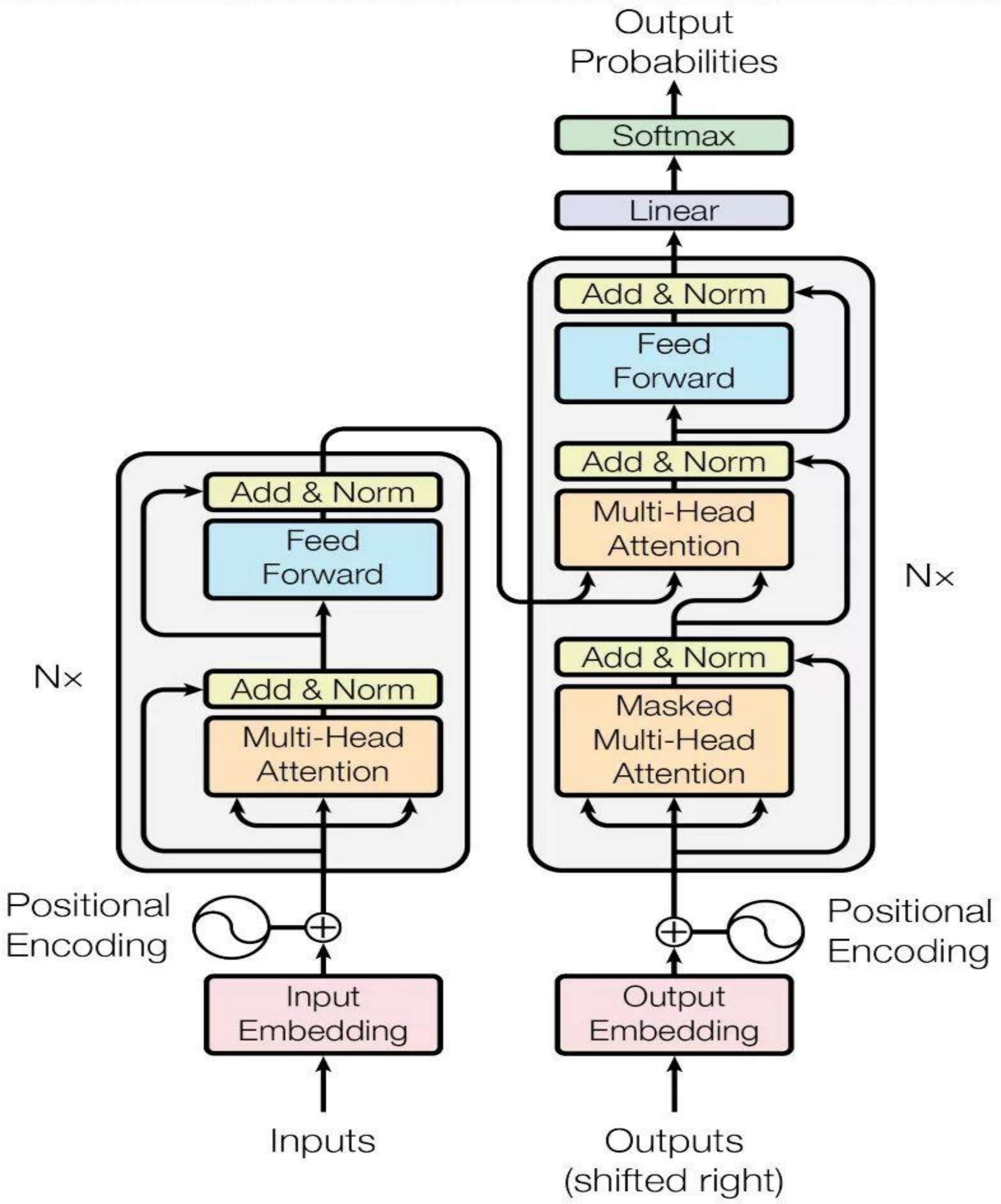


Figure 1: The Transformer - model architecture.

1、准备数据

```python
x_data = torch.tensor([[1.0],[2.0],[3.0]])
y_data = torch.tensor([[2.0],[4.0],[6.0]])
```

2、设计模型

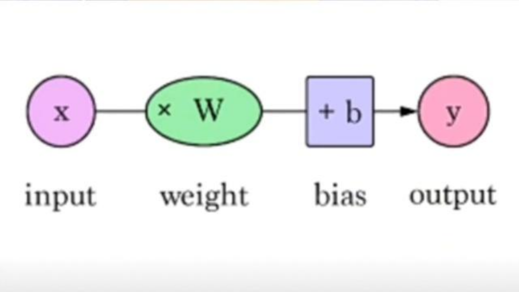
```python
class LinearModel(torch.nn.Module):
    def __init__(self):
        super(LinearModel,self).__init__()
        self.linear = torch.nn.Linear(1,1)

    def forward(self, x):
        y_pred = self.linear(x)
        return y_pred

model = LinearModel()
```

3、构造损失函数和优化器

```python
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(),lr=0.01)
```

4、训练过程

```python
epoch_list = []
loss_list = []
w_list = []
b_list = []
for epoch in range(1000):
    y_pred = model(x_data)                    # 计算预测值
    loss = criterion(y_pred, y_data)    # 计算损失
    print(epoch,loss)

    epoch_list.append(epoch)
    loss_list.append(loss.data.item())
    w_list.append(model.linear.weight.item())
    b_list.append(model.linear.bias.item())

    optimizer.zero_grad()   # 梯度归零
    loss.backward()         # 反向传播
    optimizer.step()        # 更新
```



5、结果展示

展示最终的权重和偏置：

```python
# 输出权重和偏置
print('w = ',model.linear.weight.item())
print('b = ',model.linear.bias.item())
```

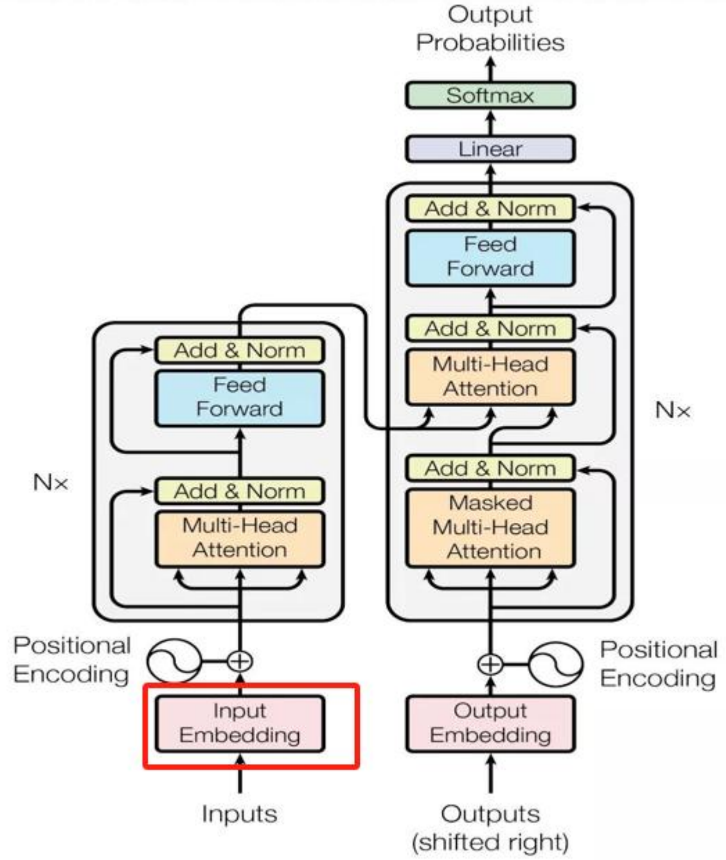
结果为：

```
w =  1.9998501539230347
b =  0.0003405189490877092
```

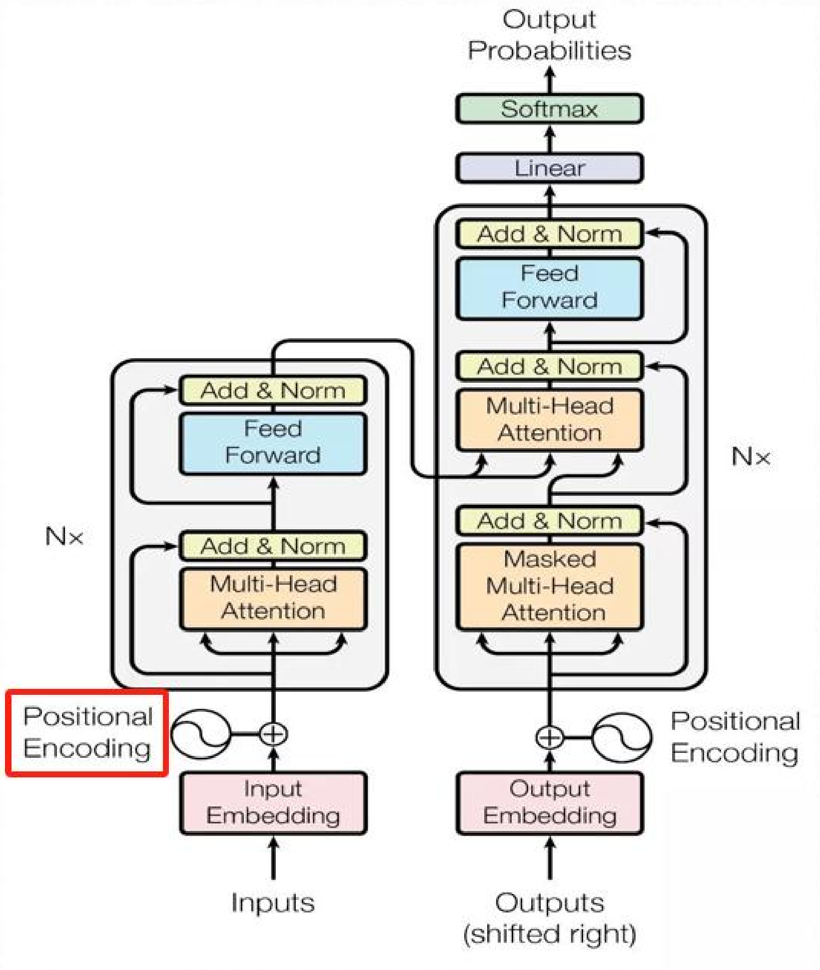
模型测试：

```python
# 测试模型
x_test = torch.tensor([[4.0]])
y_test = model(x_test)
print('y_pred = ',y_test.data)
```

```
y_pred =  tensor([[7.9997]])
```

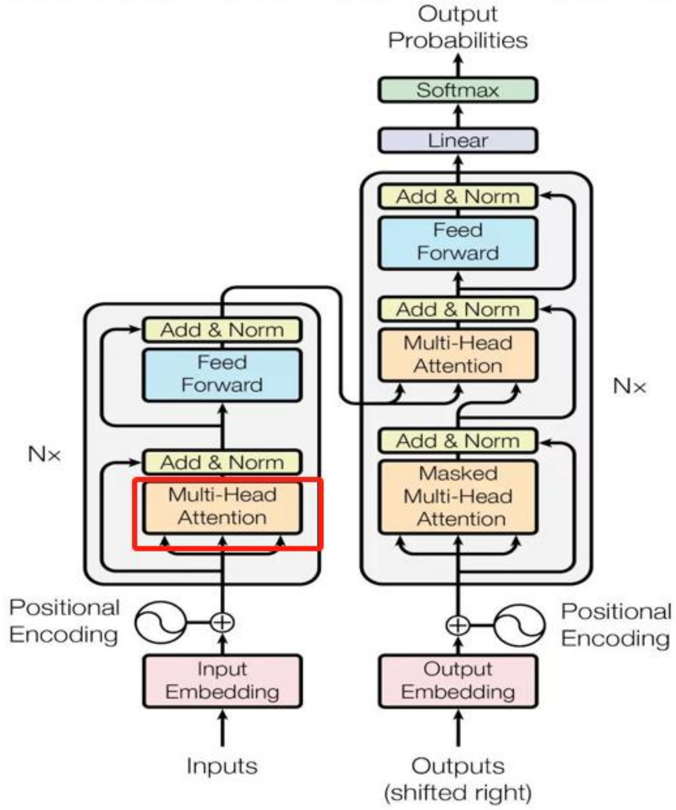


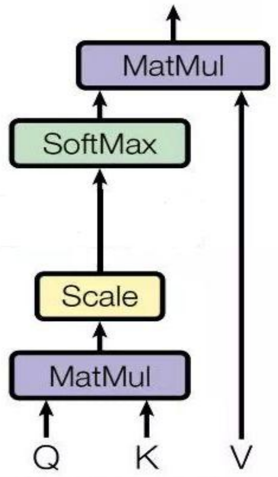

```python
class WordEmbedding(nn.Module):
    """
    把向量构造成d_model维度的词向量，以便后续送入编码器
    """
    def __init__(self, vocab_size, d_model):
        """
        :param vocab_size: 字典长度
        :param d_model: 词向量维度
        """
        super(WordEmbedding, self).__init__()
        self.d_model = d_model
        # 字典中有vocab_size个词，词向量维度是d_model，每个词将会被映射成d_model维度的向量
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.embed = self.embedding

    def forward(self, x):
        return self.embed(x) * math.sqrt(self.d_model)
```



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

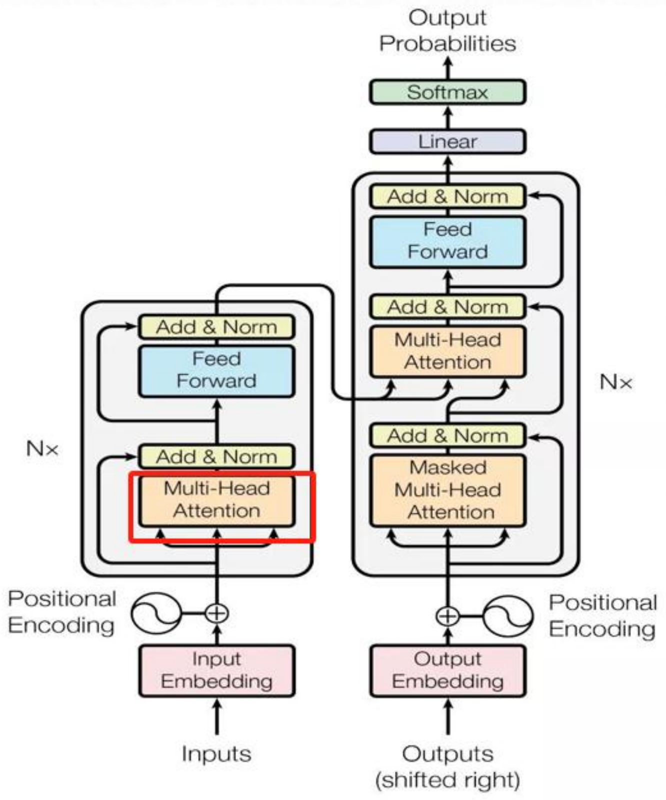
```python
class PositionalEncoding(nn.Module):
    def __init__(self, dim: int, dropout: float, max_len=5000):
        # 判断能够构建位置向量
        if dim % 2 != 0:
            raise ValueError(f"不能使用 sin/cos 位置编码，应该使用偶数维度")
        """
        pe公式为:
        PE(pos,2i/2i+1) = sin/cos(pos/10000^{2i/d_{model}})
        """
        pe = torch.zeros(max_len, dim)  # 初始化pe
        position = torch.arange(0, max_len).unsqueeze(1)  # 构建pos，为句子的长度，相当于pos
        div_term = torch.exp((torch.arange(0, dim, 2, dtype=torch.float) * torch.tensor(
            -(math.log(10000.0) / dim))))  # 复现位置编码sin/cos中的公式
        pe[:, 0::2] = torch.sin(position.float() * div_term)  # 偶数使用sin函数
        pe[:, 1::2] = torch.cos(position.float() * div_term)  # 奇数使用cos函数
        pe = pe.unsqueeze(1)  # 扁平化成一维向量
        super(PositionalEncoding, self).__init__()
        self.register_buffer('pe', pe)  # pe不是模型的一个参数，通过register_buffer把pe写入内存缓冲区，当做一个内存中的常量
        self.drop_out = nn.Dropout(p=dropout)
        self.dim = dim

    def forward(self, emb, step=None):
        """
        词向量和位置编码拼接并输出
        :return: 词向量和位置编码的拼接
        """
        emb = emb * math.sqrt(self.dim)
        if step is None:
            emb = emb + self.pe[:emb.size(0)]  # 拼接词向量和位置编码
        else:
            emb = emb + self.pe[step]
        emb = self.drop_out(emb)
        return emb
```



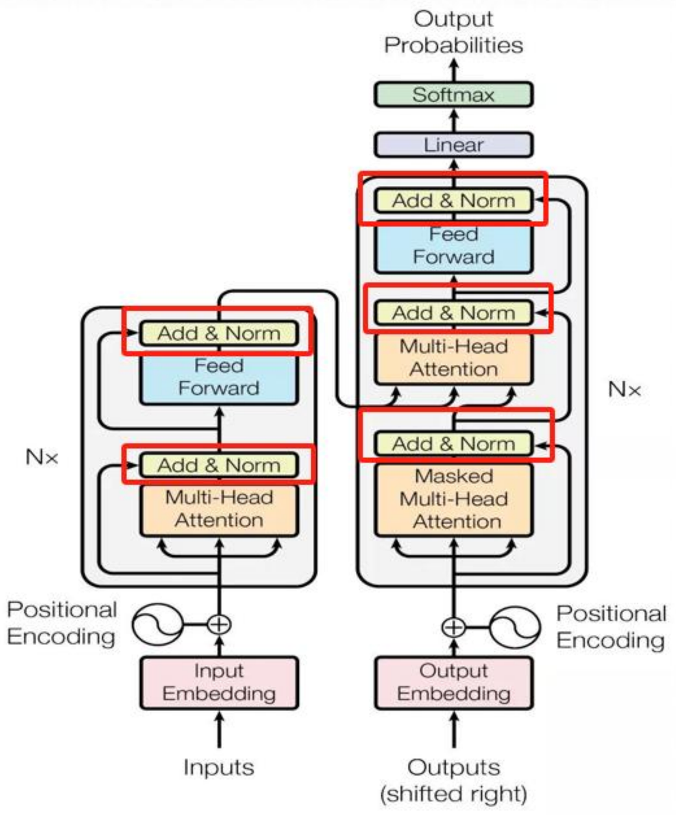


$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

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

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

    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
```



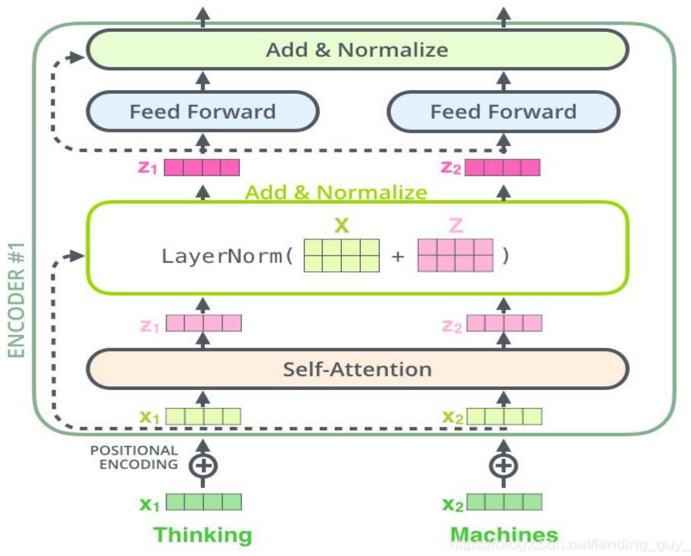

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, heads, d_model, dropout = 0.1):
        super().__init__()
        self.d_model = d_model
        self.d_k = d_model // heads
        self.h = heads

        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 = self_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
```

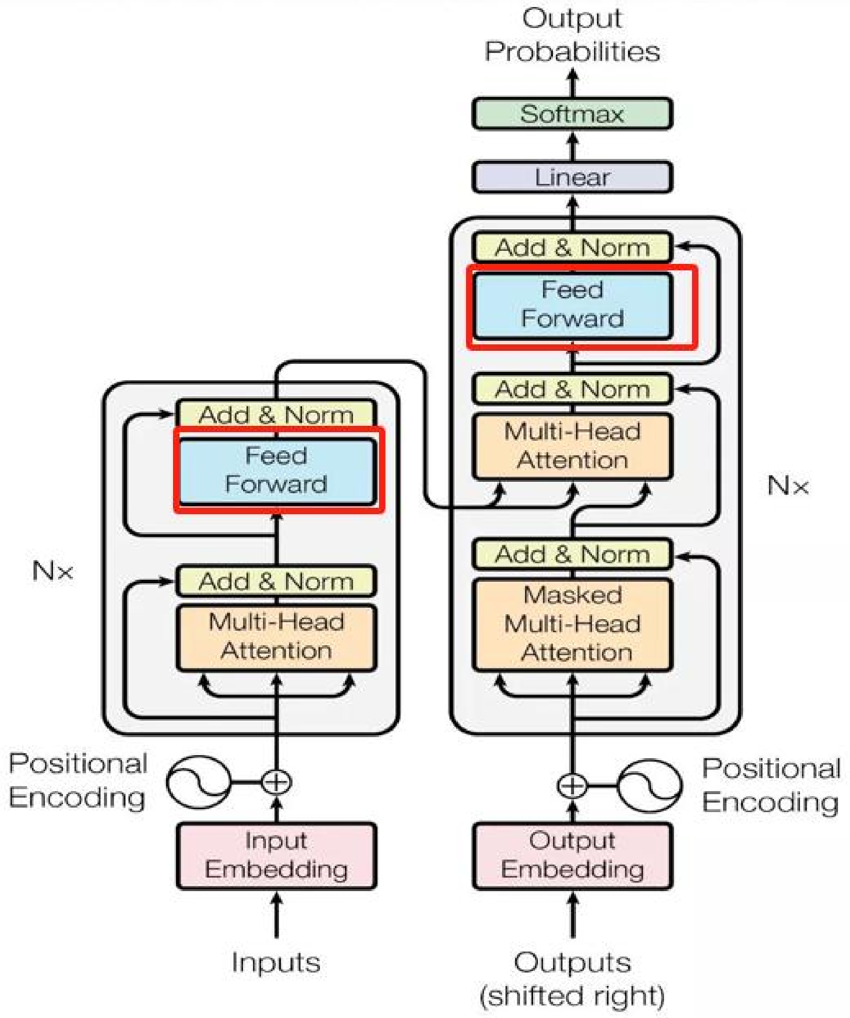



```python
class LayerNorm(nn.Module):
    """
    构建一个LayerNorm Module
    LayerNorm的作用：对x归一化，使x的均值为0，方差为1
    LayerNorm计算公式：x-mean(x)/\sqrt{var(x)+\epsilon} = x-mean(x)/std(x)+\epsilon
    """

    def __init__(self, x_size, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.ones_tensor = nn.Parameter(torch.ones(x_size))
        self.zeros_tensor = nn.Parameter(torch.zeros(x_size))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)  # 求标准差
        return self.ones_tensor * (x - mean) / (std + self.eps) + self.zeros_tensor  # LayerNorm的计算公式
```

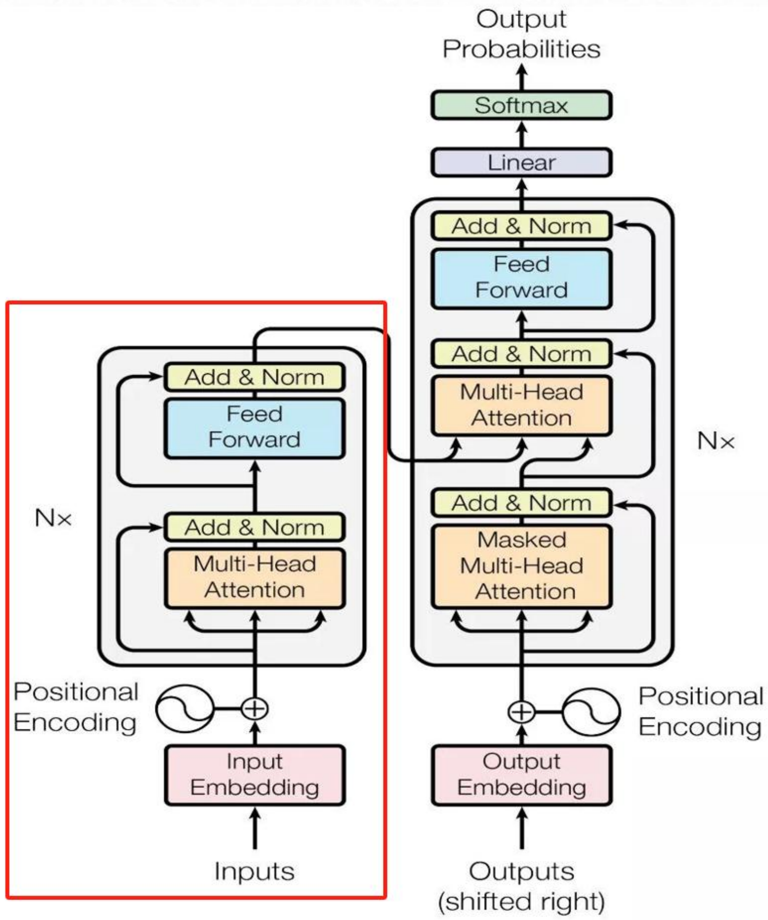
$$\tilde{Z}_j = \gamma_j \cdot \frac{Z_j - \mu_j}{\sqrt{\sigma^2 + \epsilon}} + \beta_j$$

```python
class SublayerConnection(nn.Module):
    """
    子层的连接: layer_norm(x + sublayer(x))
    上述可以理解为一个残差网络加上一个LayerNorm归一化
    """

    def __init__(self, size, dropout=0.1):
        """
        :param size: d_model
        :param dropout: drop比率
        """
        super(SublayerConnection, self).__init__()
        self.layer_norm = LayerNorm(size)
        # TODO: 在SublayerConnection中LayerNorm可以换成nn.BatchNorm2d
        # self.layer_norm = nn.BatchNorm2d()
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, x, sublayer):
        return self.dropout(self.layer_norm(x + sublayer(x)))
```





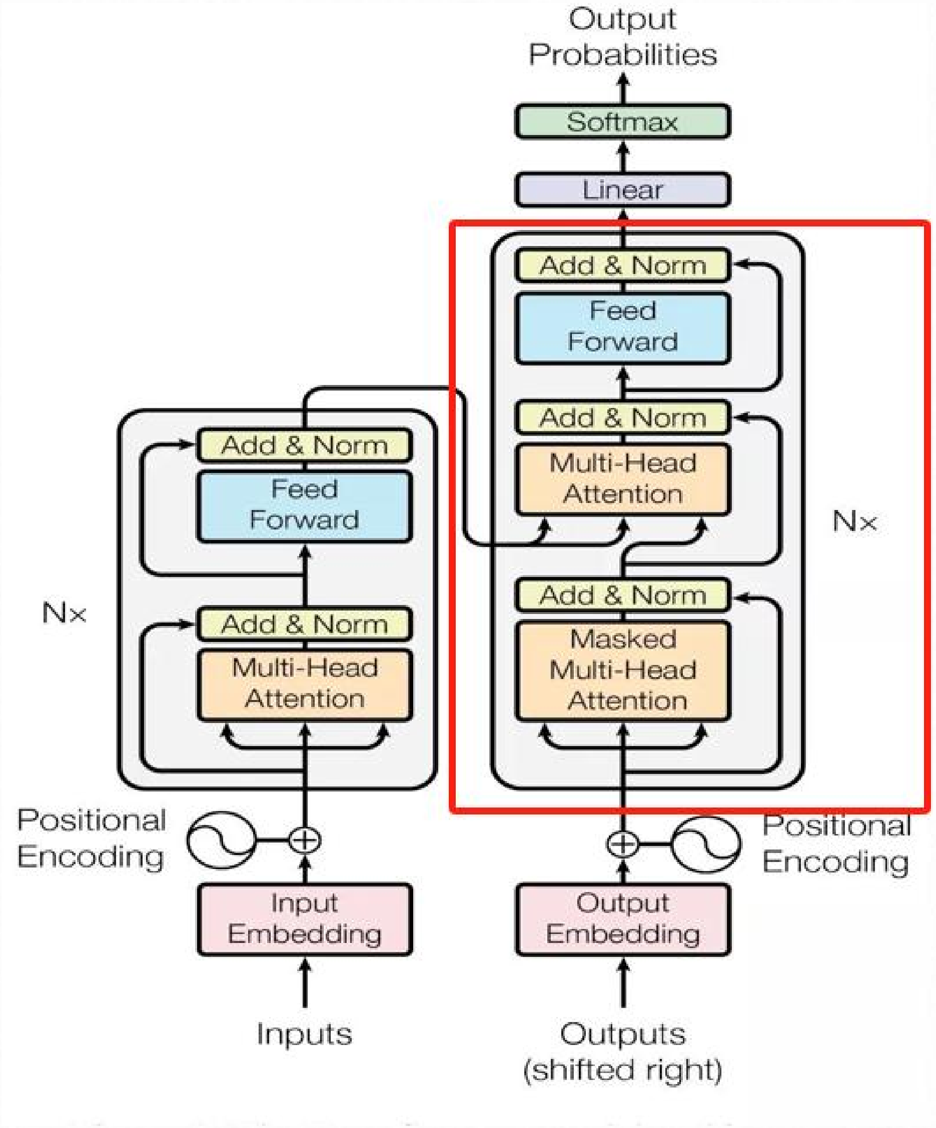

```python
class FeedForward(nn.Module):
    """
    两层具有残差网络的前馈神经网络，FNN网络
    """

    def __init__(self, d_model: int, d_ff: int, dropout=0.1):
        """
        :param d_model: FFN第一层输入的维度
        :param d_ff: FNN第二层隐藏层输入的维度
        :param dropout: drop比率
        """
        super(FeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.layer_norm = nn.LayerNorm(d_model, eps=1e-6)
        self.dropout_1 = nn.Dropout(dropout)
        self.relu = nn.ReLU()
        self.dropout_2 = nn.Dropout(dropout)

    def forward(self, x):
        """
        :param x: 输入数据，形状为(batch_size, input_len, model_dim)
        :return: 输出数据（FloatTensor），形状为(batch_size, input_len, model_dim)
        """
        inter = self.dropout_1(self.relu(self.w_1(self.layer_norm(x))))
        output = self.dropout_2(self.w_2(inter))
        # return output + x，即为残差网络
        return output  # + x
```




```python
def clone_module_to_modulelist(module, module_num):
    """
    :param module: 被克隆的module
    :param module_num: 被克隆的module数
    :return: 装有module_num个相同module的ModuleList
    """
    return nn.ModuleList([deepcopy(module) for _ in range(module_num)])
```

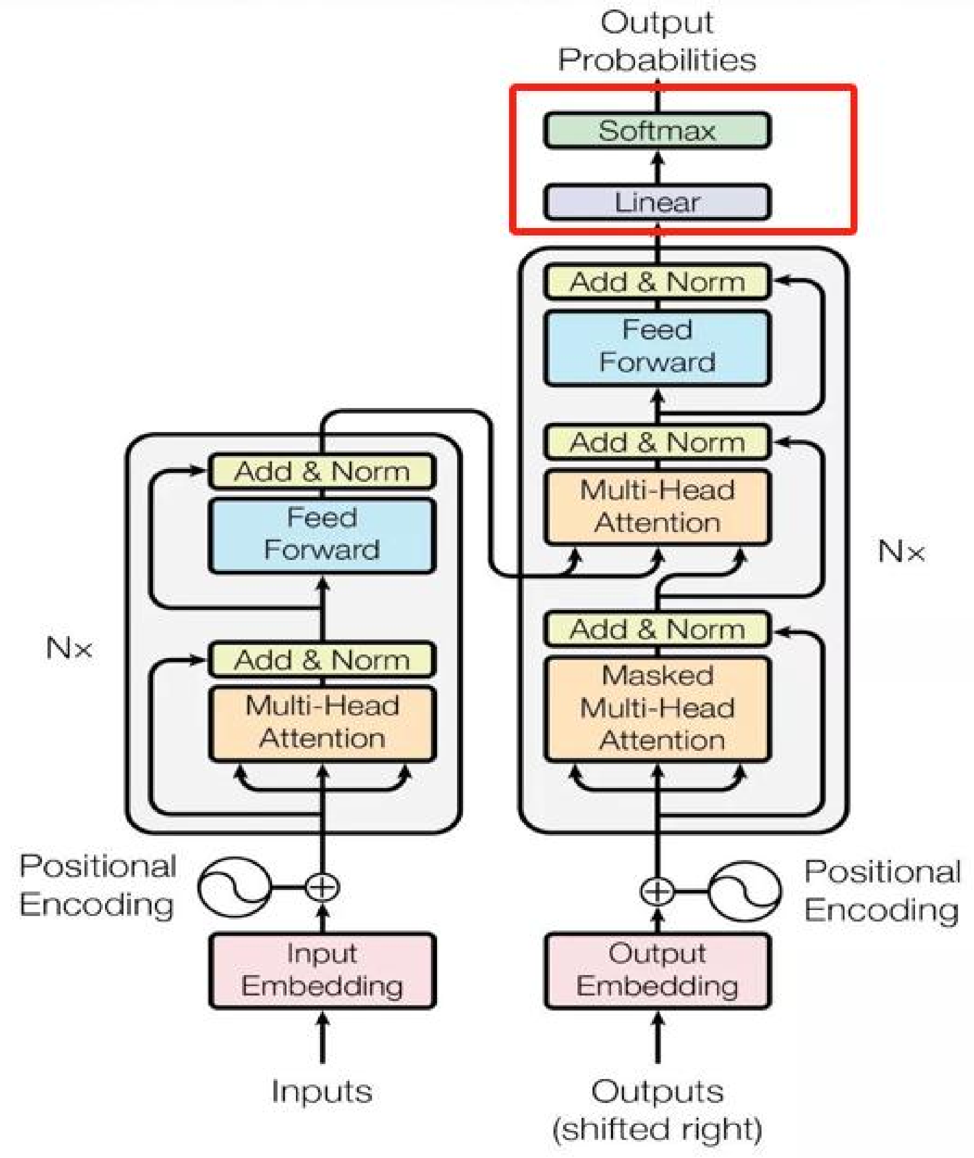
```python
class EncoderLayer(nn.Module):
    """
    一层编码Encoder层
    """
    def __init__(self, size, attn, feed_forward, dropout=0.1):
        """
        :param size: d_model
        :param attn: 已经初始化的Multi-Head Attention层
        :param feed_forward: 已经初始化的Feed Forward层
        :param dropout: drop比率
        """
        super(EncoderLayer, self).__init__()
        self.attn = attn
        self.feed_forward = feed_forward
        self.sublayer_connection_list = clone_module_to_modulelist(SublayerConnection(size, dropout), 2)

    def forward(self, x, mask):
        """
        编码层第一层子层
        self.attn 应该是一个已经初始化的Multi-Head Attention层
        把Encoder的输入数据x和经过一个Multi-Head Attention处理后的x_attn送入第一个残差网络进行处理得到first_x
        """
        first_x = self.sublayer_connection_list[0](x, lambda x_attn: self.attn(x, x, x, mask))

        """
        编码层第二层子层
        把经过第一层子层处理后的数据first_x与前馈神经网络送入第二个残差网络进行处理得到Encoder层的输出
        """
        return self.sublayer_connection_list[1](first_x, self.feed_forward)
```

```python
class Encoder(nn.Module):
    def __init__(self, n, encoder_layer):
        """
        :param n: Encoder层的层数
        :param encoder_layer: 初始化的Encoder层
        """
        super(Encoder, self).__init__()
        self.encoder_layer_list = clone_module_to_modulelist(encoder_layer, n)
    def forward(self, x, src_mask):
        """
        :param x: 输入数据
        :param src_mask: mask标志
        :return: 经过n层Encoder处理后的数据
        """
        for encoder_layer in self.encoder_layer_list:
            x = encoder_layer(x, src_mask)
        return x
```

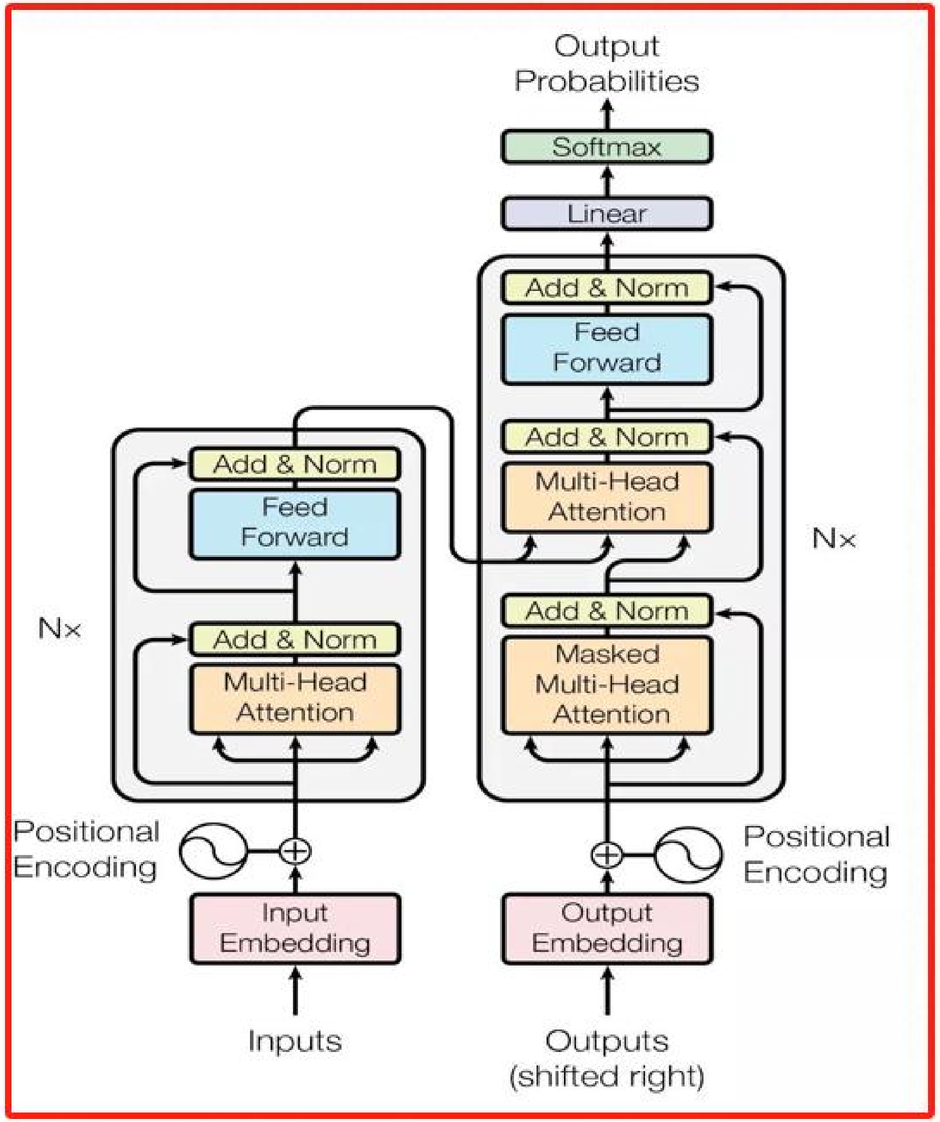



```python
class DecoderLayer(nn.Module):
    """
    一层解码Decoder层
    """
    def __init__(self, d_model, attn, feed_forward, sublayer_num, dropout=0.1):
        super(DecoderLayer, self).__init__()
        self.attn = attn
        self.feed_forward = feed_forward
        self.sublayer_connection_list = clone_module_to_modulelist(SublayerConnection(d_model, dropout), sublayer_num)

    def forward(self, x, l2r_memory, src_mask, trg_mask, r2l_memory=None, r2l_trg_mask=None):
        """
        解码器第一层子层
        把Decoder的输入数据x和经过一个Masked Multi-Head Attention处理后的first_x_attn送入第一个残差网络进行处理得到first_x
        """
        first_x = self.sublayer_connection_list[0](x, lambda first_x_attn: self.attn(x, x, x, trg_mask))

        """
        解码器第二层子层
        把第一层子层得到的first_x和
        经过一个Multi-Head Attention处理后的second_x_attn（由first_x和Encoder的输出进行自注意力计算）
        送入第二个残差网络进行处理
        """
        second_x = self.sublayer_connection_list[1](first_x,
                                                    lambda second_x_attn: self.attn(first_x, l2r_memory, l2r_memory,
                                                                                    src_mask))

        return self.sublayer_connection_list[-1](second_x, self.feed_forward)
```

```python
class R2LDecoder(nn.Module):
    """
    n个含有R2L自注意计算的解码层，该解码层只有3个残差网络
    """

    def __init__(self, n_layers, decoder_layer):
        """
        :param n_layers: Decoder层的层数
        :param decoder_layer: 初始化的Decoder层
        """
        super(R2LDecoder, self).__init__()
        self.decoder_layer_list = clone_module_to_modulelist(decoder_layer, n_layers)

    def forward(self, x, memory, src_mask, trg_mask):
        for decoder_layer in self.decoder_layer_list:
            # 没有传入r2l_memory和r2l_trg_mask，默认值为None，即该Decoder只有3个残差网络
            x = decoder_layer(x, memory, src_mask, trg_mask)
        return x
```




```python
class WordProbGenerator(nn.Module):
    """
    文本生成器，即把Decoder层的输出通过最后一层softmax层变化为词概率
    """

    def __init__(self, d_model, vocab_size):
        """
        :param d_model: 词向量维度
        :param vocab_size: 词典大小
        """
        super(WordProbGenerator, self).__init__()
        # 通过线性层的映射，映射成词典大小的维度
        self.linear = nn.Linear(d_model, vocab_size)

    def forward(self, x):
        # 通过softmax函数对词概率做出估计
        return F.log_softmax(self.linear(x), dim=-1)
```




```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)
        #print("DECODER")
        d_output = self.decoder(trg, e_outputs, src_mask, trg_mask)
        output = self.out(d_output)
        return output
```

谢谢观看