大语言模型基础

Transformer

- Transformer 结构完全通过注意力机制完成对源语言序列和目标语言序列全局依赖的建模

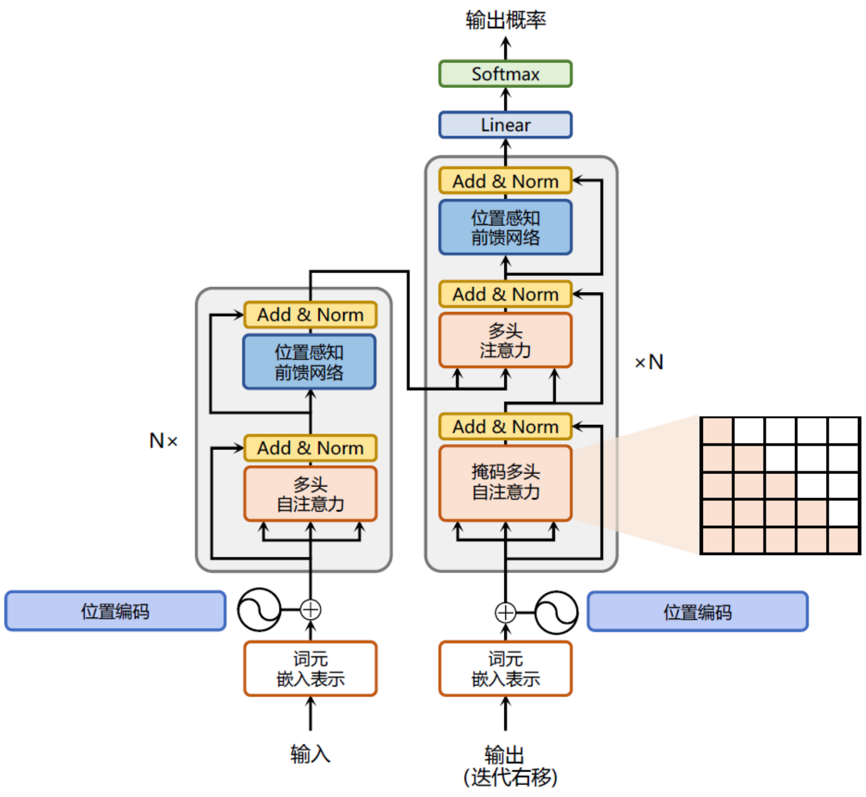

图 2.1 基于 Transformer 的编码器和解码器结构[48]

Pos Embedding好处：

-
    - 正余弦函数的范围是在[-1,+1]，导出的位置编码与原词嵌入相加不会使得结果偏离过远而破坏原有单词的语义信息
    - 依据三角函数的基本性质，可以得知第pos+k 个位置的编码是第pos 个位置的编码的线性组合，这就意味着位置编码中蕴含着单词之间的距离信息

- 位置编码代码：

```python
from kiwisolver import Variable
import torch
import torch.nn as nn
import math

class PositionEncorder(nn.Module):
    def __init__(self, d_model, max_seq_len):
        super().__init__()
        self.d_model = d_model # 模型维度
        self.max_seq_len = max_seq_len # 序列的最大长度

        pe = torch.zeros(max_seq_len, d_model) # 储存位置编码的Tensor
        for pos in range(max_seq_len): # 遍历序列每个位置
            for i in range(0, d_model, 2): # 遍历每一个维度，步长为2（两个维度一组进行正余弦计算）
                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) # 添加了一个额外的维度，使其形状从(max_seq_len, d_model)变为(1, max_seq_len, d_model)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x * math.sqrt(self.d_model) # 使得单词嵌入表示相对大一些
        seq_len = x.size(1)
        x  = x + Variable(self.pe[:,:seq_len], requires_grad=False).cuda() # 加上位置信息
```

Self Attn

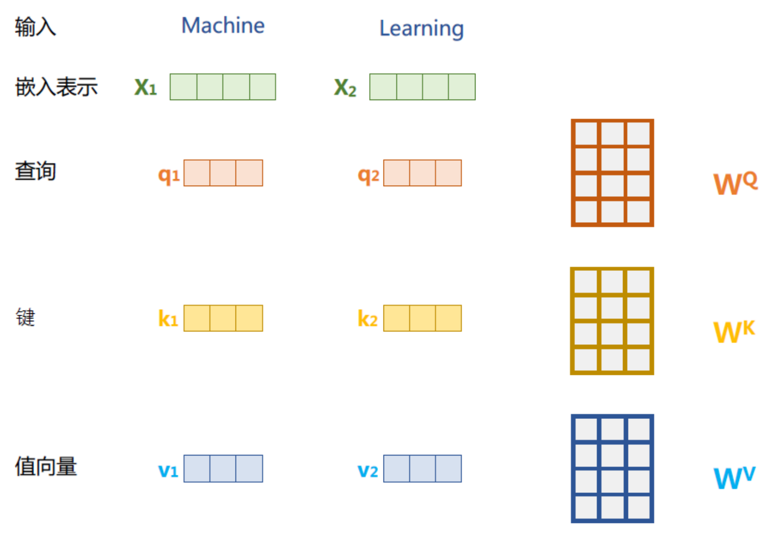

图 2.2　自注意力机制中的查询、键、值向量

- 多头注意力就是把上下文每一个单词经过多组qkv变化分别映射到不同表示空间
- MuiltiHead Atten 代码实现：

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

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()

        self.d_model = d_model
        self.d_k = d_model // heads # 确保每个头的维度加起来等于模型的总维度
        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 attention(q, k, v, d_k, mask=None, dropout=None):
        scores = torch.matmul(q, k.transpose(-2, -1)) /  math.sqrt(d_k) # q 和
k 点乘后除以根号d_k, 得到注意力分数

        # 掩盖掉那些为了填补长度增加的单元, 使其通过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

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

        bs = q.size(0)

        # 进行线性操作划分成h个头
        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)

        # 调换维度顺序使得最后一个维度为单元长度
        k = k.transpose(1,2)
        q = q.transpose(1,2)
        v = v.transpose(1,2)

        # 计算attention
        scores = attention(q, k, v, self.d_k, mask, self.dropout)

        # Concat多个头
        concat = scores.transpose(1,2).contiguous().view(bs, -1, self.d_model)
        output = self.out(concat)

        return output
```

前馈层

- 前馈层接受自注意力子层的输出作为输入，并通过一个带有Relu 激活函数的两层全连接网络对输入进行更加复杂的非线性变换
- 代码：

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

class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff=2048, dropout=0.1):
        super(FeedForward, self).__init__()

        # d_ff默认为2048
        self.linear_1 = nn.Linear(d_model, d_ff)
        self.dropout = nn.Dropout(dropout)
        self.linear_2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        x = self.dropout(F.relu(self.linear_1(x)))
        x = self.linear_2(x)
        return x
```

残差连接和归一化

- LayerNorm：用于将数据平移缩放到均值为0，方差为1 的标准分布，层归一化技术可以有效地缓解优化过程中潜在的不稳定、收敛速度慢等问题
- 代码：

```
import torch
import torch.nn as nn

class NormLayer(nn.Module):
    def __init__(self, d_model, eps=1e-6):
        super().__init__()
        self.size = d_model
        self.eps = eps

        # 两个可学习参数
        self.alpha = nn.Parameter(torch.ones(self.size))
        self.bias = nn.Parameter(torch.zeros(self.size))

    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
```

Encorder & Decoder

- 编码器：
-

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

class EncoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, dropout=0.1):
        super().__init__()

        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)

        self.attn = MultiHeadAttention(d_model, n_heads, 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 Encorder(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)
```

- 解码器:

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

class DecorderLayer(nn.Module):
    def __init__(self, d_model, heads, dropout=0.1):
        super(DecorderLayer, self).__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.attn1 = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.attn2 = 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.attn1(x2, x2, x2, trg_mask))
        x2 = self.norm_2(x)
        x = x + self.dropout_2(self.attn2(x2, e_outputs, e_outputs, src_mask))
        x2 = self.norm_3(x)
        x = x + self.dropout_3(self.ff(x2))
        return x
```

```
class Decorder(nn.Module):
    def __init__(self, vocab_size, d_model, N, heads, dropout=0.1):
        super.__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.ps = PositionalEncoder(d_model, dropout=dropout)
        self.layers = get_clones(DecorderLayer(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.ps(x)
        for i in range(self.N):
            x = self.layers[i](x, e_outputs, src_mask, trg_mask)
        return self.norm(x)
```

Transformer模块

```
import torch.nn as nn

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
```

LlaMA: 基于改进的Transformer

- 整体Trans架构跟GPT-2类似

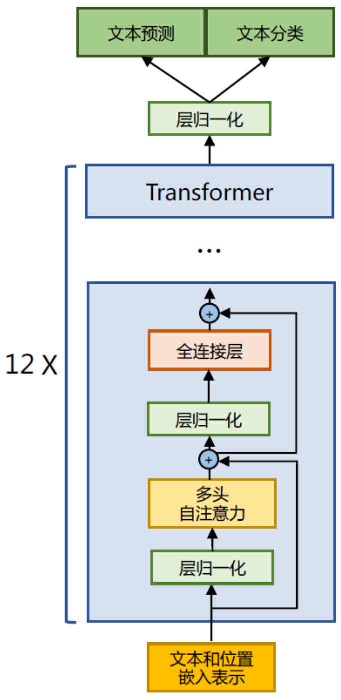

图 2.4　GPT-2 模型结构

- RMSNorm 归一化函数（前置归一化）
    - 将第一个层归一化移动到多头自注意力层之前，第二个层归一化也移动到了全连接层之前，同时残差连接的位置也调整到了多头自注意力层与全连接层之后
- SwiGLU 激活函数
- 旋转位置嵌入（RoPE）

注意力机制优化

- 稀疏注意力机制：现有的稀疏注意力机制，通常是基于以下五种基本基于位置的稀疏注意力机制的复合模式

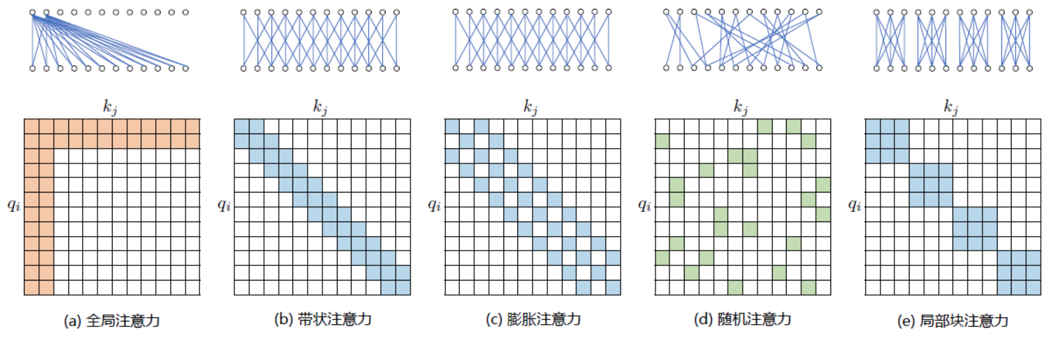

图 2.6　五种基于位置的稀疏注意力基本类型[53]

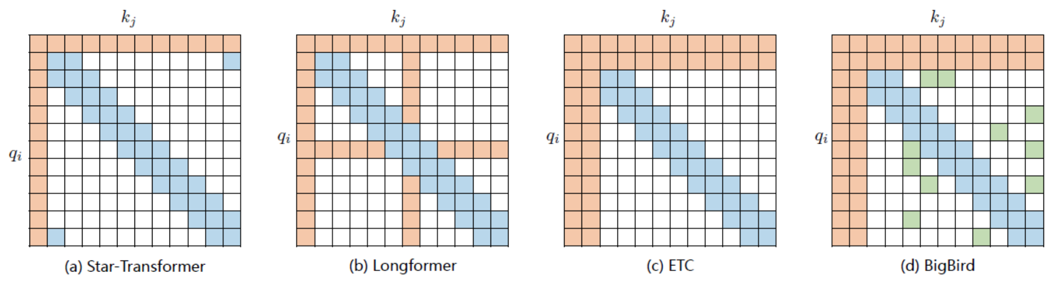

图 2.7　基于位置复合稀疏注意力类型[53]

- FlashAttention：根据Nvidia显卡的特硬件特点加速
- Muiti-Query Attention：不同的注意力头共享一个键和值的集合，减少显存占用