Happy-LLM

一、NLP的基础概念

1. 文本表示的发展历程

1. Word2Vec

一种词嵌入(Word Embedding)技术，利用词在文本中的上下文信息来捕捉词之间的语义关系，进而决定词在向量空间中的分布距离。

两大架构：

1. 连续词袋模型(CBOW)：根据目标词定上下文中的词对应的词向量，计算并输出目标词的向量表示。*适用于小型数据集*
2. Skip-Gram模型：利用目标词的向量表示计算上下文中的词向量。*在大型语料中表现更好*

2. ELMo——Embeddings from Language Models

实现了一词多义、静态词向量到动态词向量的跨越式转变。

两阶段过程：

1. 利用语言模型进行 **预训练。**
2. 在做特定任务时，从预训练网络中提取对应单词的词向量作为新特征补充到下游任务中。 **特征提取**

优点：能捕捉到词汇的多义性和上下文信息，生成的词向量更加丰富、准确

二、Transformer架构

1. 注意力机制

核心变量： Query（查询值）、Key（键值）、Value（真值）

欧式距离衡量词向量的相似性：

$$v \cdot w = \sum_i v_i w_i \tag{1}$$

计算公式:

设 Query 为 "fruit",对应的词向量为 q ，Key对应的词向量矩阵为 $k = [v_{apple}, v_{banana}, v_{chair}]$ ，则利用向量点乘，可得到 Query 和每一个 Key 的相似度：

$$x = qK^T \tag{2}$$

此后利用Softmax函数将其转化为和为1的权重：称为**注意力分数**

$$softmax(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3}$$

再将得到的注意力分数和值向量做对应的乘积，即可得到**注意力机制的计算**基本公式：

$$attention(Q, K, V) = softmax(qK^T)v \tag{4}$$

如果一次性查询多个Query，将多个 Query 堆叠成一个矩阵 Q：

$$attention(Q,K,V)=softmax(QK^T)V \tag{5}$$

为了防止softmax放缩时误差较大，引入**缩放因子**d_k：

$$attention(Q,K,V)=softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{6}$$

2. 自注意力

计算本身序列中每个元素对其他元素的注意力分布。

应用在 Transformer 的 Encoder 结构中。

Q、K、V 分别是输入对参数矩阵W_q, W_k, W_v做积得到的。

作用： 可以找到一段文本中每一个 token 与其他所有 token 的相关关系大小，从而建模文本之间的依赖关系。

3. 掩码注意力

实现同时输入，并行运算

4. 多头注意力 Multi-Head Attention

对同一语料进行多次注意力计算，将多次结果拼接成作为最后的输出。

实质：将原始的输入序列进行多组的自注意力处理；然后再将每一组得到的自注意力结果拼接起来，再通过一个线性层进行处理，得到最终的输出。

$$MultiHead(Q,K,V)=Concat(head_1,\dots,head_h)W^o \tag{7}$$
$$where head_i=Attention(QW_i^Q,KW_I^K,VW_i^V)$$

三、Encoder-Decoder

1. Seq2Seq 模型

序列到序列模型：输入一个自然语言序列，输出返回一个可能不等长的自然语言序列。

一般思路：对自然语言序列进行编码再解码。

- 编码：将输入的自然语言序列通过隐藏层编码成能够表征语义的向量。
- 解码：对输入的自然语言序列编码得到的向量通过隐藏层输出，并解码成对应自然语言目标序列。

2. 前反馈网络 FNN

每一层的神经元都和上下两层的每一个神经元完全连接的网络结构。

每一个 Encoder Layer 都包含一个注意力机制和一个前反馈神经网络。

结构：

- 线性层 --> RELU 激活函数 --> 线性函数 --> Dropout （防止过拟合）

```
class MLP(nn.Module):
    '''前馈神经网络'''
    def __init__(self, dim: int, hidden_dim: int, dropout: float):
        super().__init__()
```

```
        # 定义第一层线性变换，从输入维度到隐藏维度
        self.w1 = nn.Linear(dim, hidden_dim, bias=False)
        # 定义第二层线性变换，从隐藏维度到输入维度
        self.w2 = nn.Linear(hidden_dim, dim, bias=False)
        # 定义dropout层，用于防止过拟合
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # 前向传播函数
        # 首先，输入x通过第一层线性变换和RELU激活函数
        # 最后，通过第二层线性变换和dropout层
        return self.dropout(self.w2(F.relu(self.w1(x))))
```

3. 层归一化 Layer Norm

神经网络的主流归一化：批归一化、层归一化。

归一化目的：让不同层输入的取值范围或者分布能够比较一致。

作用：将每一层的输入都归一化成标准正态分布。

批归一化是在一个 mini-batch 上进行归一化。

层归一化是在每个样本上计算其所有层的均值和方差。

公式：

计算样本的均值：

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} Z_j^i \tag{8}$$

计算样本的方差：

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(Z_j^i - \mu_j)^2 \tag{9}$$

对每个样本的值减去均值再除以标准差来将这个样本的分布转化为标准正态分布：

$$\widetilde{Z_j} = \frac{Z_j - \mu_j}{\sqrt{\sigma^2 + \epsilon}} \tag{10}$$

代码：

```
class LayerNorm(nn.Module):
    ''' Layer Norm 层'''
    def __init__(self, features, eps=1e-6):
    super().__init__()
    # 线性矩阵做映射
    self.a_2 = nn.Parameter(torch.ones(features))  # 可学习的缩放参数（gamma）
    self.b_2 = nn.Parameter(torch.zeros(features)) # 可学习的平移参数（beta）
    self.eps = eps                                 # 防止除零的小常数

    def forward(self, x):
    # 计算最后一个维度的均值和标准差
    mean = x.mean(-1, keepdim=True)   # [batch_size, seq_len, 1]
    std = x.std(-1, keepdim=True)     # [batch_size, seq_len, 1]
```

```
    # 标准化 -> 缩放 -> 平移
    return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

4. 残差链接 Residual Connection

下一层的输入 = 上一层的输出 + 上一层的输入。

公式：

$$
\begin{aligned}
x &= x + MultiHeadSelfAttention(LayerNorm(x)) \\
output &= x + FNN(LayerNorm(x))
\end{aligned} \tag{11}
$$

代码：

```
# 注意力计算
h = x + self.attention.forward(self.attention_norm(x))
# 经过前馈神经网络
out = h + self.feed_forward.forward(self.fnn_norm(h))
```

5. Encoder

Encoder Layer的实现

```
class EncoderLayer(nn.Module):
  '''Encoder层'''
    def __init__(self, args):
        super().__init__()
        # 一个 Layer 中有两个 LayerNorm，分别在 Attention 之前和 MLP 之前
        self.attention_norm = LayerNorm(args.n_embd)
        # Encoder 不需要掩码，传入 is_causal=False
        self.attention = MultiHeadAttention(args, is_causal=False)
        self.fnn_norm = LayerNorm(args.n_embd)
        self.feed_forward = MLP(args.dim, args.dim, args.dropout)

    def forward(self, x):
        # Layer Norm
        norm_x = self.attention_norm(x)
        # 自注意力
        h = x + self.attention.forward(norm_x, norm_x, norm_x)
        # 经过前馈神经网络
        out = h + self.feed_forward.forward(self.fnn_norm(h))
        return out
```

Encoder的实现

```
class Encoder(nn.Module):
    '''Encoder 块'''
    def __init__(self, args):
        super(Encoder, self).__init__()
        # 一个 Encoder 由 N 个 Encoder Layer 组成
        self.layers = nn.ModuleList([EncoderLayer(args) for _ in
range(args.n_layer)])
        self.norm = LayerNorm(args.n_embd)

    def forward(self, x):
        "分别通过 N 层 Encoder Layer"
        for layer in self.layers:
            x = layer(x)
        return self.norm(x)
```

6. Decoder

结构

- Decoder由两个注意力层和一个前馈神经网络组成。
- 其中第一个注意力层是掩码自注意力层
- 第二个注意力层是一个多头注意力层，使用第一个注意力层的输出作为 query，使用Encoder的输出作为 key 和 value，来计算注意力分数，最后再经过前反馈神经网络。

Decoder Layer的实现：

```
class DecoderLayer(nn.Module):
  '''解码层'''
    def __init__(self, args):
        super().__init__()
        # 一个 Layer 中有三个 LayerNorm，分别在 Mask Attention 之前、Self Attention
之前和 MLP 之前
        self.attention_norm_1 = LayerNorm(args.n_embd)
        # Decoder 的第一个部分是 Mask Attention，传入 is_causal=True
        self.mask_attention = MultiHeadAttention(args, is_causal=True)
        self.attention_norm_2 = LayerNorm(args.n_embd)
        # Decoder 的第二个部分是 类似于 Encoder 的 Attention，传入 is_causal=False
        self.attention = MultiHeadAttention(args, is_causal=False)
        self.ffn_norm = LayerNorm(args.n_embd)
        # 第三个部分是 MLP
        self.feed_forward = MLP(args.dim, args.dim, args.dropout)

    def forward(self, x, enc_out):
        # Layer Norm
        norm_x = self.attention_norm_1(x)
        # 掩码自注意力
        x = x + self.mask_attention.forward(norm_x, norm_x, norm_x)
        # 多头注意力
        norm_x = self.attention_norm_2(x)
        h = x + self.attention.forward(norm_x, enc_out, enc_out)
        # 经过前馈神经网络
        out = h + self.feed_forward.forward(self.ffn_norm(h))
        return out
```

Decoder的实现

```
class Decoder(nn.Module):
    '''解码器'''
    def __init__(self, args):
        super(Decoder, self).__init__()
        # 一个 Decoder 由 N 个 Decoder Layer 组成
        self.layers = nn.ModuleList([DecoderLayer(args) for _ in
range(args.n_layer)])
        self.norm = LayerNorm(args.n_embd)

    def forward(self, x, enc_out):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, enc_out)
        return self.norm(x)
```

四、搭建一个 Transformer

1. Embedding 层

作用：将自然语言的输入转化为机器可以处理的向量。

本质：一个存储固定大小的词典的嵌入向量查找表

内部实现：内部是一个可训练的权重矩阵，词表中的每一个值，都对应一行维度为 embedding_dim的向量。

输入： 一个形状为(batch_size, seq_len, 1)的矩阵

- 第一个维度是一次批处理的数量
- 第二个维度是自然语言序列的长度
- 第三个维度是token经过tokenizer（分词器）转化为的index值。

代码：

```
self.tok_embeddings = nn.Embedding(args.vocab_size, args.dim)
```

2. 位置编码

根据序列中的token的相对位置对其进行编码，再将位置编码加入词向量编码中。

目的：为了使用序列顺序信息，保留序列中的相对位置信息。

编码方式：

$$PE(pos, 2i) = \sin(pos/10000^{2i/d_{model}}) \tag{12}$$
$$PE(pos, 2i+1) = \cos(pos/10000^{2i/d_{model}})$$

其中 pos 为 token 在句子中的位置，2i 和 2i+1 则指示了token是奇数位置还是偶数位置。

代码实现：

```
import numpy as np
import matplotlib.pyplot as plt
def PositionEncoding(seq_len, d_model, n=10000):
    P = np.zeros((seq_len, d_model))
    for k in range(seq_len):
        for i in np.arange(int(d_model/2)):
            denominator = np.power(n, 2*i/d_model)
            P[k, 2*i] = np.sin(k/denominator)
            P[k, 2*i+1] = np.cos(k/denominator)
    return P

P = PositionEncoding(seq_len=4, d_model=4, n=100)
print(P)
```

位置编码层：

```
class PositionalEncoding(nn.Module):
    '''位置编码模块'''

    def __init__(self, args):
        super(PositionalEncoding, self).__init__()
        # Dropout 层
        # self.dropout = nn.Dropout(p=args.dropout)

        # block size 是序列的最大长度
        pe = torch.zeros(args.block_size, args.n_embd)
        position = torch.arange(0, args.block_size).unsqueeze(1)
        # 计算 theta
        div_term = torch.exp(
            torch.arange(0, args.n_embd, 2) * -(math.log(10000.0) / args.n_embd)
        )
        # 分别计算 sin、cos 结果
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer("pe", pe)

    def forward(self, x):
        # 将位置编码加到 Embedding 结果上
        x = x + self.pe[:, : x.size(1)].requires_grad_(False)
        return x
```