LLM Notes

Y R

Updated on 2025-10-08

1 一些专业名词简单定义

中文分词	Chinese Word Segmentation, CWS	处理中文文本时，由于词与词之间灭有明显分隔（空格），所以无法直接通过空格来确定词的边界。其目的是将连续的中文文本切分成有意义的词汇序列。
子词切分	Subword Segmentation	特别适合处理词汇稀疏的问题，即当遇到罕见词或者未见过的新词时，能够通过已知的子词单位来理解或生成这些词汇。在处理拼写复杂，合成词多的语言（德语）或预训练语言模型（BERT，GPT）中尤为重要。常用的方法有Byte Pair Encoding (BPE), WordPiece, Unigram, SentencePiece。
词性标注	Part of speech Tagging, POS Tagging	为文本中的每一个单词分配一个词性标签，如名词动词形容词。 POS tagging对理解句子结构，进行句法分析，语义角色标注等高级NLP任务至关重要。计算机可以更好地理解文本的含义，进行信息提取，情感飞，机器翻译。。通常依赖于机器学习模型，如隐马尔科夫模型(Hidden Markov Model HMM), 条件随机场（COnditional Random Field CRF),或RNN，LSTM等。通过学习大量的标注数据来预测新句子中每个单词的词性。
文本分类	Text Classfication	将给定的文本自动分配到一个或多个预定义的类别中。应用包括但不限于情感分析，垃圾邮件检测，新闻分类，主题识别等。文本分类的关键在于理解文本的含义和上下文，并基于此将文本映射到特定的类别。文本分类的关键在于选择合适的特征表示和分类算法，以及拥有高质量的训练数据。

实体识别 (又名，命名实体识别)	Named Entity Recognition, NER	自动识别文本中具有特定意义的实体，并将它们分类为预定的类别，如人名，地点，组织，日期，时间等。实体识别任务对于信息提取，知识图谱构建，问答系统，内容推荐等应用很重要，它能够帮助系统理解文本中的关键元素及其属性。
关系抽取	Relation Extraction	它的目标是从文本识别实体之间的语义关系。这些关系可以是因果关系，拥有关系，亲属关系，地理关系等。对理解文本内容，构建知识图谱，提升机器理解语言的能力具有重要意义。
文本摘要	Text Summarization	目的是生成一段剪辑真确的摘要，来概括原文的主要内容。根据生成的方式不同，文本摘要可以分为两大类：抽取式摘要(Extractive Summarization)和生成式摘要(Abstractive Summarization)。
	抽取式摘要	通过直接从原文中选取关键句子或短语来组成摘要。优点：信息玩去哪来自原文，因此准确性较高。缺：由于仅仅是原文中句子的拼接，有时候可能不够流畅。
	生成式摘要	不仅需要涉及选择文本片段，还需要对这些片段进行重新组织和改写，并生成新的内容。它更具有挑战性，因为需要理解文本的深层含义，并能够以新的方式表达相同的信息。生成式摘要通常需要更复杂的模型，如注意力机制的Seq2Seq.
机器翻译	Machine Translation （MT）	
自动问答	Automatic Question Answering, QA	可大致分为三类：检索式回答（Retrieval-based QA），通过搜索引擎等方式从大量文本中检索答案；知识库问答（knowledge-based QA），通过结构化的知识库来回答问题；社区问答（Community QA），依赖于用户生成的问答数据，如问答社区，论坛等。

表 1: LLM 任务类别

2 文本表示

- 词向量： Vector Space Model (VSM).

  向量空间模型通过将文本（包括单词，句子，段落或整个文档）转换为高维空间中的向量来实现文本的数学化表示。

  在这个模型中，每个维度代表一个特征项（例如字，词，词语或短语），而向量中每个元素值代表该特征项在文本中的权重。这种权重通过特定的计算公式（如词频TF，逆文档频率TF-IDF）来确定，反映了特征项在文本中的重要程度。

VSM应用及其广泛，包括但不限于文本相似度计算，文本分类，信息检索等。它将复杂的文本数据转换为易于计算和分析的数学形式。

VSM也存在很多问题。其中最主要的是数据稀疏性和维数灾难，因为特征数量庞大导致向量维度极高，同时多数元素值为零。此外，由于模型基于特征项之间的独立性假设，忽略了文本中的结构信息（如词序和上下文信息），限制了模型了表现力。特征项的选择和权重计算方法的不足也是VSM需要解决的问题。

为了解决这些问题，对VSM的研究主要集中在两方面： 1.改进特征表示方法，如借助图方法，主题方法等关键词抽取。 2. 改进和优化特征项权重的计算方法，可以在现有的方法基础上进行融合计算或提出新的计算方法。

VSM例子：

“雍和宫的荷花很美”

词汇表大小：16384. 句子包含词汇：[“雍和宫”，“的”，“荷花”，“很”，“美”] 共5个词。

vector = [0, 0, ... 1, 0, ...1,0, ... 1, 0, ...1, 0, ..., 1, 0, ...]

16384维的vector中有5个位置为1，其余为0 （16384-5 = 16379）

实际有效维度： 5

稀疏率： 16379 / 16384 = 99.97%

- N-gram

  N-gram 是一种基于统计的语言模型，应用于语音识别，手写识别，拼写纠错，机器翻译和搜索引擎等。

  N-gram的核心思想基于马尔科夫假设，即一个词的出现概率仅依赖于它前面的N-1个词。这里的N代表连续出现单词的数量，可以是任意正整数。

  当 N=1时称为 unigram，仅考虑单个词的概率。

  当 N=2时称为bigram，考虑前一个词来估计当前词的概率。

  当 N=3时称为 trigram，考虑前两个词来估计第三个词的概率，以此类推N-gram。

  N-gram通过条件概率链式规则来估计整个句子的概率。具体而言，对于给定的句子，模型会计算每个N-gram出现的条件概率，并将这些概率相乘得到整个句子的概率。

  优点：实现简单，容易理解。缺点：当N较大时，会出现数据稀疏性的问题。模型的参数空间急剧增大，相同的N-gram序列出现的概率变得非常低，导致模型无法有效学习，泛化能力下降。此外，N-gram忽略了词之间的范围依赖关系，无法捕捉到句子中的复杂结构和语义信息。

  N-gram例子：

  句子："The quick brown fox", 做trigram模型。

  需要计算 $P(\text{"}brown\text{"}|\text{"}The\text{"},\text{"}quick\text{"})$, $P(\text{"}fox\text{"}|\text{"}quick\text{"},\text{"}brown\text{"})$ 的概率，并相乘。

- Word2vec

是一种基于神经网络NNLM的语言模型，通过学习词与词之间的上下文关系来生成词的密集向量表示。核心思想是利用词在文本中的上下文信息来捕捉词之间的语义关系，从而使得语义相似或相关的词在向量空间中距离较近。

主要两种架构：

连续词袋模型 CBOW （Continuous Bad of Words). 根据目标词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。

Skip-Gram：利用目标词的向量表示计算上下文中的词向量。适用大型语料。

相比较传统的高维稀疏表示（like one-hot encoding), Word2vec生成低维密集向量（通常几百维），有助于减少计算复杂度和存储需求。

Word2vec能捕捉到词与词之间的语义关系，比如“国王”和“王后”在向量空间中的位置比较接近。但由于CBOW/Skip-gram是基于局部上下文的，无法捕捉到长距离的依赖关系，缺乏整体的词与词之间的关系，因此在一些复杂的语义任务上表现不佳。

- ELMo Embeddings from Language Models

  该模型实现了一词多义，静态词向量到动态词向量的跨越式转变。

  首先在大型语料库上训练语言模型，得到词向量模型，然后在特定任务上对模型进行微调，得到更适合该任务的词向量。 ELMo首次将预训练思想引入到词向量的生成中，适用双向LSTM结构，能够捕捉到词汇的上下文信息，生成更加丰富和准确的词向量表示。

  第一阶段是利用语言模型进行预训练。第二阶段是在做特定任务时，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重。基于RNN的LSTM模型训练时间长，特征提取是ELMo模型优化和提升的关键。

  ELMo的主要问题是模型复杂度高，训练时间长，计算资源消耗大。

3 Attention Mechanism 注意力机制

注意力机制三个核心变量: Query(查询值), Key（键值）, Value（真值）

$$attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{1}$$

Q is the Query matrix, K is the Key matrix, V is the Value matrix, and d_k is the dimension of the key vectors. Usually Q and K are of the same dimension d_k

注意力机制的本质是对两段序列的元素依次进行相似度的计算，寻找出一个序列的每个元素对另一个序列的每个元素的相关度，然后基于相关度进行加权，即分配注意力。而这两段序列即是我们计算过程中QKV的来源。

利用pytorch来实现attention机制的代码如下：

```
def attention(query, key, value, drouput=None):
```

```
d_k = query.size(-1)
scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)

p_attn = scores.softmax(dim=-1)
if dropout is not None:
    p_attn = dropout(p_attn)
return torch.matmul(p_attn, value), p_attn
```

但是在实际应用中，我们往往只需要计算Query和Key之间的注意力结果，很少存在额外的真值Value。在经典注意力机制中，Q往往来自于一个序列，K与V来自另一个序列，都通过参数矩阵计算得到，从而可以你和这两个序列之间的关系。例如在Transformer的Decoder结构中，Q来自于Decoder的输入，K与V来自于Encoder的输出，从而拟合了编码信息与历史信息之间的关系，实现了Decoder对Encoder输出的关注。

但在Transformer的Encoder结构中，使用的是自注意力机制（self-attention)。计算本身序列中的每个元素对其他元素的注意力分布，即在计算过程中，QKV都由同一个输入通过不同的参数矩阵计算得到。在Encoder中QKV分别是输入对参数矩阵W_q, W_k, W_v 做积得到，从而拟合输入语句中每一个token对其他的所有token的关系。

通过自注意力机制，我们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。在代码中的实现，self-attention是通过QKV的输入传入同一个参数实现:

$attention(X, X, X)$

4 Mask Self-Attention 掩码自注意力机制

是指使用注意力掩码的自注意力机制。掩码是一种用于控制注意力分布的技术，通常用于处理序列数据中的某些特殊情况，如填充（padding）或未来信息的屏蔽。掩码的作用是屏蔽一些特定位置的token，模型在学习的过程中会忽略被屏蔽的token。

使用注意力掩码的核心动机是让模型只能使用历史信息进行预测而不能看到未来的信息。使用注意力机制的Transformer模型也是通过类似于n-gram的语言模型任务来学习的，也就是对一个文本序列，不断根据之间的token来预测下一个token，直到将整个文本序列补全。

eg：待学习的文本序列是 [BOS] I Like you [EOS], 那么模型会按如下顺序进行预测和学习：

- step 1: 输入【BOS】, 输出 I
- step 2: 输入【BOS, I】, 输出 Like
- step 3: 输入【BOS, I, Like】, 输出 you
- step 4: 输入【BOS, I, Like, you】, 输出 [EOS]

理论上，只要学习的语料足够多，通过上述的过程，模型可以学会任意一种文本序列的建模方式，也就是可以对任意的文本进行补全。

但是需要注意的是，上述的过程是一个串行的过程，也就是，需要先完成step1，才能进行step2, 以此类推。而Transformer模型的优势在于其并行计算的能力，也就是可以同时对一个文本序列的所有token进行计算，而不是像RNN那样一个个token顺序计算。因此，为了让Transformer模型既能并行计算，又能保证每个token只能看到历史信息，不能看到未来的信息，就需要使用掩码技术。

例如我们待学习的文本仍旧是 [BOS] I Like you [EOS]，我们使用的注意力掩码是[MASK]，那么模型的输入为：

- <BOS> [MASK] [MASK] [MASK] [MASK]
- <BOS> I [MASK] [MASK] [MASK]
- <BOS> I Like [MASK] [MASK]
- <BOS> I Like you [MASK]
- <BOS> I Like you [/EOS]

在每一行输入中，模型仍然只能看到前面的token，预测下一个token。但是上述输入不再是串行的过程，而是可以并行计算的过程。模型只需要每一个样本根据未被遮蔽的token来预测下一个token即可。

观察上述的掩码，它可以看作是一个与文本序列等长的上三角矩阵。我们可以通过创建一个和输入等长的上三角矩阵来实现掩码的功能。也就是说，当输入维度为 $(batch_size, seq_len, d_model)$时，我们创建的掩码矩阵维度为 $(1, seq\_len, seq\_len)$，上三角矩阵中的1表示该位置的token可以被看到，0表示该位置的token被遮蔽。

在具体实现中，通过以下代码生成MASk矩阵：

```
# create a 1*seq_len*seq_len matrix use full function
mask = torch.full((1, args.max_seq_len, args.max_seq_len), float('-inf'))
# use triu function to make it upper triangular matrix
mask = torch.triu(mask, diagonal=1)
```

生成的MASK是一个上三角矩阵,上半部分值为-inf，下半部分值为0.0。在注意力计算时，将计算得到的注意力分数和这个掩码做和，再进行softmax计算。

```
scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)
scores = scores + mask[:, :seq_len, :seq_len]  # mask的维度是1*seq_len*seq_len
scores = F.softmax(scores.float(), dim=-1).type_as(xq)  # (batch_size, seq_len, seq_len)
if mask is not None:
    scores = scores + mask
attn = F.softmax(scores, dim=-1)
output = torch.matmul(attn, V)
```

求和之后上三角区域（被遮蔽的token对应位置）注意力分数结果都是-inf，下三角区保持不变。经过softmax处理后， -inf会变成0，从而忽略了上三角区域计算的注意力分数，从而实现注意力遮蔽。

5 多头注意力:Multi-Head Attention, MHA

注意力机制可以实现并行化与长期依赖关系拟合，但一次注意力计算只能拟合一种相关关系。单一的注意力机制很难全面拟合语句序列里相关关系，因此使用了多头注意力机制(Multi-Head Attention, MHA)。

MHA,即同时对一个语料进行多次注意力计算，每次注意力计算都能拟合不同的关系，将最后多次的结果拼接起来作为最后的输出，即可更全面深入地拟合语言信息。

用公式可以表示为：

$$MultiHead(Q,K,V) = Concat(head_1, head_2, ..., head_h)W^O \tag{2}$$

where $head_i = attention(QW_i^Q, KW_i^K, VW_i^V)$.

代码实现不复杂，即n个头就有n组QKV参数矩阵，分别计算n次注意力，然后拼接起来。

```
import torch.nn as nn
import torch

'''多头自注意力计算模块'''
class MultiHeadAttention(nn.Module):

    def __init__(self, args: ModelArgs, is_causal=False):
        # 构造函数
        # args: 配置对象
        super().__init__()
        # 隐藏层维度必须是头数的整数倍，因为后面我们会将输入拆成头数个矩阵
        assert args.dim % args.n_heads == 0
        # 每个头的维度，等于模型维度除以头的总数。
        self.head_dim = args.dim // args.n_heads
        self.n_heads = args.n_heads

        # Wq, Wk, Wv 参数矩阵，每个参数矩阵为 n_embd x dim
        # 这里通过三个组合矩阵来代替了n个参数矩阵的组合，其逻辑在于矩阵内积再拼接其实等同于拼接矩阵再内积，
        # 不理解的读者可以自行模拟一下，每一个线性层其实相当于n个参数矩阵的拼接
        self.wq = nn.Linear(args.n_embd, self.n_heads * self.head_dim, bias=False)
```

```
        self.wk = nn.Linear(args.n_embd, self.n_heads * self.head_dim, bias=False)
        self.wv = nn.Linear(args.n_embd, self.n_heads * self.head_dim, bias=False)
        # 输出权重矩阵，维度为 dim x dim（head_dim = dim / n_heads）
        self.wo = nn.Linear(self.n_heads * self.head_dim, args.dim, bias=False)
        # 注意力的 dropout
        self.attn_dropout = nn.Dropout(args.dropout)
        # 残差连接的 dropout
        self.resid_dropout = nn.Dropout(args.dropout)
        self.is_causal = is_causal

        # 创建一个上三角矩阵，用于遮蔽未来信息
        # 注意，因为是多头注意力，Mask 矩阵比之前我们定义的多一个维度
        if is_causal:
            mask = torch.full((1, 1, args.max_seq_len, args.max_seq_len), float("-inf"))
            mask = torch.triu(mask, diagonal=1)
            # 注册为模型的缓冲区
            self.register_buffer("mask", mask)

    def forward(self, q: torch.Tensor, k: torch.Tensor, v: torch.Tensor):

        # 获取批次大小和序列长度，[batch_size, seq_len, dim]
        bsz, seqlen, _ = q.shape

        # 计算查询（Q）、键（K）、值（V）,输入通过参数矩阵层，维度为 (B, T, n_embed) x (n_embed, dim) ->
        xq, xk, xv = self.wq(q), self.wk(k), self.wv(v)

        # 将 Q、K、V 拆分成多头，维度为 (B, T, n_head, dim // n_head)，然后交换维度，变
成 (B, n_head, T, dim // n_head)
        # 因为在注意力计算中我们是取了后两个维度参与计算
        # 为什么要先按B*T*n_head*C//n_head展开再互换1、2维度而不是直接按注意力输入展开，
是因为view的展开方式是直接把输入全部排开，
        # 然后按要求构造，可以发现只有上述操作能够实现我们将每个头对应部分取出来的目标
        xq = xq.view(bsz, seqlen, self.n_heads, self.head_dim)
        xk = xk.view(bsz, seqlen, self.n_heads, self.head_dim)
        xv = xv.view(bsz, seqlen, self.n_heads, self.head_dim)
        xq = xq.transpose(1, 2)
        xk = xk.transpose(1, 2)
        xv = xv.transpose(1, 2)
```

```
        # 注意力计算
        # 计算 QK^T / sqrt(d_k)，维度为 (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T)
        scores = torch.matmul(xq, xk.transpose(2, 3)) / math.sqrt(self.head_dim)
        # 掩码自注意力必须有注意力掩码
        if self.is_causal:
            assert hasattr(self, 'mask')
            # 这里截取到序列长度，因为有些序列可能比 max_seq_len 短
            scores = scores + self.mask[:, :, :seqlen, :seqlen]
        # 计算 softmax，维度为 (B, nh, T, T)
        scores = F.softmax(scores.float(), dim=-1).type_as(xq)
        # 做 Dropout
        scores = self.attn_dropout(scores)
        # V * Score，维度为(B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
        output = torch.matmul(scores, xv)

        # 恢复时间维度并合并头。
        # 将多头的结果拼接起来，先交换维度为 (B, T, n_head, dim // n_head)，再拼接成 (B, T, n_head *
        # contiguous 函数用于重新开辟一块新内存存储，因为Pytorch设置先transpose再view会
报错，
        # 因为view直接基于底层存储得到，然而transpose并不会改变底层存储，因此需要额外存
储
        output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)

        # 最终投影回残差流。
        output = self.wo(output)
        output = self.resid_dropout(output)
        return output
```

6 Encoder-Decoder

Encoder -Decoder是注意力机制的两个核心组件。 BERT只使用Encoder,GPT只使用Decoder。

6.1 Seq2Seq

Seq2Seq是只模型输入的是一个自然语言序列 $input = (x_1, x_2, x_3, ...x_n)$, 输出是一个可能不等长的自然语言序列$output = (y_1, y_2, y_3, ...y_n)$.

几乎所有的NLP任务都可以转化为Seq2Seq任务，例如机器翻译，文本摘要，文本生成，问答系统等。

对于Seq2Seq的任务，一般的思路是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或矩阵，可以简单理解为更复杂的词向量表示。解码：对输入的自然语言序列编码得到的向量或矩阵通过隐藏层输出，再解码成对应的自然语言目标序列。

Transformer由Encoder和Decoder组成。每一个Encoder/Decoder又由6个Decoder/Encoder Layer 组成。输入的源序列会进入Encoder进行编码，到Encoder Layer的最顶层再将编码结果输出给Decoder Layer的每一层，通过Decoder进行解码后得到输出目标序列。

6.1.1 Feed Forward Neural Network, FNN

每一层的神经元都和上下两层的每一个神经元完全链接的网络结构。每一个Encoder Layer都包含一个注意力机制和一个前馈神经网络。代码实现可参考如下：

```
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))))
```

Transformer的前馈神经网络是由两个线性层中间加一个RELU组成，以及FNN还加入了一个droupout防止过拟合。

6.1.2 Layer Normalization, LN. 层归一化

归一化的核心，是为了让不同层的输入的取值范围或分布能够比较一致。由于DNN每一层的输入都是上一层的输出，因此多层传递下，对于网络中较高的层，之前的所有神经层的参数变化会导致其输入

的分布发生较大改变。即，随着神经网络参数的更新，各层的输出分布是不同的，且差异会随着网络深度的增大而增大。但是需要预测的条件分布始终是相同的，从而造成了预测的误差。

批归一化（Batch Norm）是指在一个mini-batch上进行归一化，相当于对一个Batch样本拆分出来一部分，首先计算样本的均值：

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^i \tag{3}$$

x_j^i 表示样本i在第j个维度上的值，m表示一个Batch的样本数量。

再计算样本的方差：

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^i - \mu_j)^2 \tag{4}$$

最后对每个样本的值减去均值在除以标准差来将这个mini-batch的样本分布转化成正态分布：

$$\hat{x}_j^i = \frac{x_j^i - \mu_j}{\sqrt{\sigma^2 + \epsilon}} \tag{5}$$

此处加上一个很小的数ϵ是为了防止分母为0.

但是Batch norm有一些缺陷：

- 当显存有限，mini-batch较小时，BatchNorm取得样本均值和方差不能反映全局的统计分布信息，从而导致结果变差。
- 对于在时间维度展开的RNN，不同的句子的同一分布大概率不同，所以BatchNorm的归一化用处不大。
- 在训练时，BatchNorm需要保存每个step的统计信息（均值和方差）。在测试时由于变长句子的特性，测试机可能出现比训练集更长的句子。所以对于后面位置的step，没有训练的统计特征使用。
- 应用BatchNorm每个step都需要保存和计算batch统计量增加了计算和内存开销。

因此层归一化（Layer Norm）被提出。

相比较BatchNorm在每一层统计所有样本的均值和方差，LayerNorm在每个样本上计算其所有层的均值和方差，从而使每个样本的分布达到稳定。其方式是一样的，只是统计的维度不同。

```
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))
```

```
self.b_2 = nn.Parameter(torch.zeros(features))
self.eps = eps

   def forward(self, x):
# 在统计每个样本所有维度的值，求均值和方差
mean = x.mean(-1, keepdim=True) # mean: [bsz, max_len, 1]
std = x.std(-1, keepdim=True) # std: [bsz, max_len, 1]
   # 注意这里也在最后一个维度发生了广播
return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```