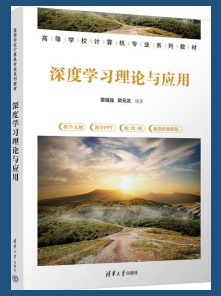
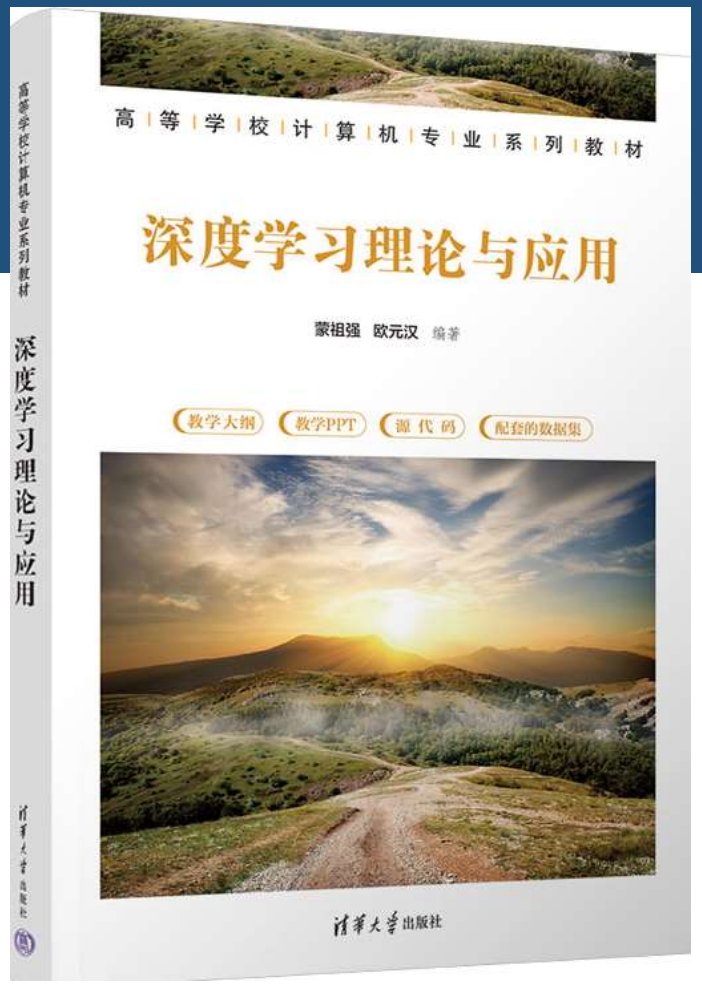


深度学习理论与应用

Deep Learning Theory and Applications

蒙祖强，欧元汉 编著



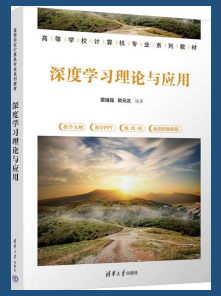
教　材

全国各大书店网店均有销售

- **教学大纲**：提供面向教育工程认证的教学大纲
- **教学PPT**：提供课堂教学用的PPT课件
- **源代码**：提供教材涉及的全部源代码
- **数据集**：提供教材示例、案例用到的全部数据集

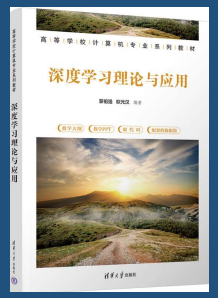
获取教学资源：
http://www.tup.tsinghua.edu.cn/booksCenter/book_09988101.html

教材： 蒙祖强，欧元汉 编著. 深度学习理论与应用. 北京: 清华大学出版社，2023年7月. (书号: 978-7-302-63508-6)



第 8 章 基于预训练模型的自然语言处理

蒙祖强，欧元汉 编著. 深度学习理论与应用. 北京: 清华大学出版社，2023年7月.



本章内容

contents

8.1 Seq2Seq结构与注意力机制

8.1.1 Seq2Seq 结构

Seq2Seq 结构定义：Seq2Seq结构是自然语言处理领域中一个著名的网络结构，一般翻译为“序列到序列结构”。一个 Seq2Seq 结构通常由 Encoder和 Decoder构成。其中，Encoder 用于接收由若干个元素构成的序列作为输入，并对输入序列进行编码和特征提取，形成特征向量；Decoder 则对形成的特征向量进行解码，形成由若干个元素构成的输出序列。

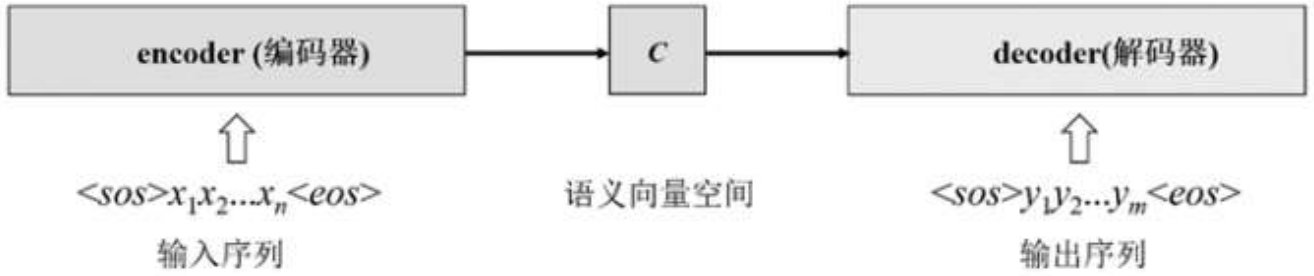


图 8-1　Seq2Seq 结构示意图

8.1 Seq2Seq结构与注意力机制

8.1.1 Seq2Seq 结构

输入和输出：输入序列和输出序列分别为<sos>$x_1x_2...x_n$<eos>和<sos>$y_1y_2...y_m$<eos>，其中<sos>和<eos>分别表示序列的起始符号和终止符号，一般 n 和 m 不相等，输入序列 $x_1x_2...x_n$ 和输出序列 $y_1y_2...y_m$ 中的元素更不会一一对应。

编码器和解码器：在 Seq2Seq 结构中，编码器 Encoder 负责提取输入序列 $x_1x_2...x_n$的语义特征，产生语义向量 C，解码器 Decoder 则负责将 C 转化为另一种序列 $y_1y_2...y_m$——输出序列。这种结构的应用领域非常广泛，如文本翻译（英文翻译为中文或中文翻译为英文等）、图像描述（图像翻译为文本）、文章摘要（长文本翻译为短文本）、语音翻译（语音到文本）等。

8.1 Seq2Seq结构与注意力机制

8.1.1 Seq2Seq 结构

【例 8.1】创建一个基于 Seq2Seq 结构的网络模型，用于实现英文到中文的翻译。

- **数据集：**本例使用的训练语料来自 WMT18 网站（https://www.statmt.org/wmt18/translation-task.html）从中下载英文中文语料集，经过预处理后保在./data/translate 目录下的 en_zh_data.txt 文件中，一共有 10 万条英文-中文句子对，英文和中文句子以“--->” 隔开。

8.1 Seq2Seq结构与注意力机制

8.1.1 Seq2Seq 结构

（1）**编码器定义：**使用了 GRU 作为循环神经网络来对输入序列进行处理。GRU (Gated Recurrent Unit) 是 LSTM 的改进版，具有结构简单、效率高等优点，受到越来越多人的青睐

```
class Encoder(nn.Module):
    #en_vocab_num 表示英文单词数，hidden_size 表示词向量长度
    def __init__(self, en_vocab_num, hidden_size):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(en_vocab_num, hidden_size) #定义嵌入层
        #使用 GRU 作为循环神经网络，它是 LSTM 的变体
        self.gru = nn.GRU(hidden_size, hidden_size)
    def forward(self, x, h):
        x = self.embedding(x)
        x = x.reshape(1, 1, -1) #改变形状，以符合 GRU 的输入格式
        o, h = self.gru(x, h)
        return o, h #返回新的输出和隐层向量
```

8.1 Seq2Seq结构与注意力机制

8.1.1 Seq2Seq 结构

GRU 对 LSTM 的改进:

- 将输入门、遗忘门、输出门变为更新门和重置门，简化了结构;
- 将长期状态 c 和短时状态 h 合并为一个状态 h。GRU 只有两个返回结果，而 LSTM 有三个返回结果。

GRU 和 LSTM 返回格式的区别:

```
o, h = self.gru(x, h)
o, (h, c) = self.lstm(x, h)
```

GRU 的使用：GRU 没有返回长期状态向量 c，相当于只有返回 o 和 h，它们相当于 LSTM的第一和第二个输出，即 o 表示各个计算单元的输出的堆叠结果，h 表示最后一个计算单元的输出。h 就是图 8-1 中的语义向量 C；当然也可以根据需要，利用各个计算单元的输出的平均值作为语义向量 C。但 GRU 和 LSTM 的参数调用设置相似。下面两个语句是等同的:

```
self.gru = nn.GRU(hidden_size, hidden_size)
self.gru = nn.GRU(input_size = hidden_size, hidden_size = hidden_size,
num_layers=1, bidirectional=False, batch_first=False)
```

8.1 Seq2Seq结构与注意力机制

8.1.1 Seq2Seq 结构

（2）**解码器定义：**接收传入解码器的张量是 x 和 h，其中 x 表示当前输入解码器的词的编码，h 则表示编码器产生的语义向量，是对应英文句子的特征向量。解码器的任务就是产生紧跟输入 x 后面的词，然后不断循环这个过程，直到遇到结束符为止，这样就可以产生中文句子。

```
class Decoder(nn.Module):
    #hidden_size 表示词向量的长度，en_vocab_num 表示中文词的数量
    def __init__(self, hidden_size, zh_vocab_num):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(zh_vocab_num,
                                      hidden_size) #定义嵌入层
        #使用 GRU 作为循环神经网络，它是 LSTM 的变体
        self.gru = nn.GRU(hidden_size, hidden_size)
        #全连接网络，用于预测输出为各个词的概率
        self.fc = nn.Linear(hidden_size, zh_vocab_num)
    def forward(self, x, h):
        x = self.embedding(x)
        x = F.relu(x)
        x = x.reshape(1, 1, -1)
        o, h = self.gru(x, h)
        o = o.squeeze(0)
        o = self.fc(o)
        return o, h
```

注：完整代码见教材219页

8.1 Seq2Seq结构与注意力机制

8.1.1 Seq2Seq 结构

在补充完其他代码（如读数据、预处理、测试代码等）后，训练编码器 Encoder 和解码器 Decoder，然后就可以用它们来翻译给定的一个英文句子。例如，给出如下的英文句子：

> But before this new order appears, the world may be faced with spreading disorder if not outright chaos.

在笔者计算机上，利用训练好的编码器 Encoder 和解码器 Decoder，得到如下的中文句子：

> 但是 在 这 一新 秩序 的 出现 之前 世界 可能 会 面临 更 广泛 的 混沌 如果 不 是 彻底 的 混乱 的话

8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

深度学习中的注意力机制:

- 一是一组权值参数的学习,
- 二是基于权值参数的加权求和

例子：假设要研究 n 个元素（$x_1, x_2, ..., x_n$）的“合成”问题。因为每个元素对合成结果有不同程度的影响，可以为每个元素 x_j 分别设置一个权重参数α_j，这些参数便构成了相应的权重向量$(\alpha_1, \alpha_2, ..., \alpha_n)$，这些权重参是可学习的。通过训练，使得关键元素的权重参数比较大，而非关键元素的权重参数比较小（甚至为 0）。这样，通过加权求和：$\alpha_1x_1 + \alpha_2x_2 + ... + \alpha_nx_n$，便得到它们有效的合成结果 y。

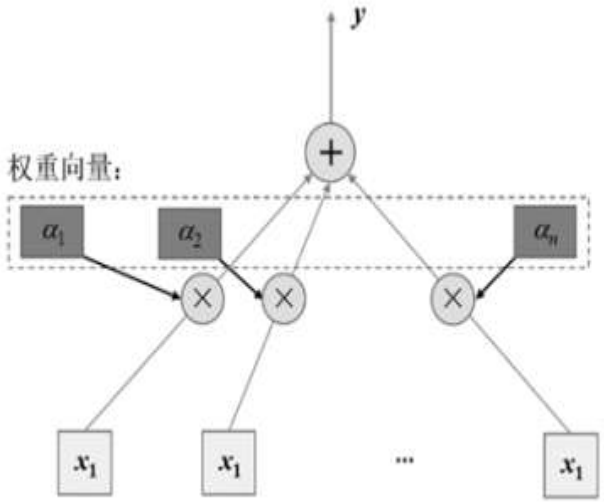


图 8-2 注意力机制中加权求和示意图

8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

注意力机制的表示：在深度学习中，为计算α_j，输入序列中的每个元素 x_j 通常被视为由键（Key）和值（Value）两部分组成，分别表示为 K_j 和 V_j。其中，键 K_j 可以定义为 x_j 的特征向量，如词嵌入向量等；值 V_j 可定义为 x_j 在进入计算单元处理后产生的输出等。在训练过程中，对于目标序列中的元素 y_t一般视为查询（Query），用 Q_t表示。首先计算 Q_t与各 K_j的相似度，记为 $s(Q_t, K_j)$，其中 $j = 1,2,\ldots,n$。

8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

相似度 $s(Q_t, K_j)$的计算方法：

$$s(Q_t, K_j) = \begin{cases} V \cdot \tanh(W \cdot [Q_t; K_j]) & (1)\ \text{多层感知机} \\ Q_t^T \cdot \mathrm{W} \cdot K_j & (2)\ \text{双线性} \\ Q_t^T \cdot K_j & (3)\ \text{点积} \\ \frac{Q_t^T \cdot K_j}{\sqrt{d}} & (4)\ \text{缩放点积} \end{cases}$$

其中，Q_t 表示在时间步t 向解码器输入的查询（元素），“[]”表示张量拼接，“T”表示矩阵转置，d 为向量的长度；“·”在第一和第二条公式中表示矩阵相乘，在第三和第四条中表示向量之间的点积，第四条公式中除以$\sqrt{d}$的目的是为了避免因为向量长度过大而导致点积结果过大。注意，公式中 V 和 W 均为待学习的权重参数矩阵，它们实际上就是相应的全连接网络层参数。缩放点积就是著名的 Transformer 框架的注意力计算机制。

8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

在计算 s(Qt, Kj)之后，再对其进行 softmax 归一化（即概率归一化），得到权重向量。
归一化计算:

$$\alpha_{tj} = \frac{\exp(s(Q_t, K_j))}{\sum_{k=1}^{n} \exp(s(Q_t, K_k))}, t = 0,1,\ldots,m\text{-}1, j = 1,2,\ldots,n$$

其中，m 为输出序列的长度，n 为输入序列的长度。计算结果α_{tj}是一实数值，(α_{t1}, α_{t2}, …, α_{tn})为一个权重向量。接着，利用 softmax 归一化的权重向量(α_{t1}, α_{t2}, …, α_{tn})来计算查询 Qt 关于输入序列 S 的注意力向量。
注意力向量计算:

$$Att(Q_t, S) = \alpha_{t1} \cdot V_1 + \alpha_{t2} \cdot V_2 + \ldots + \alpha_{tn} \cdot V_n$$

其中，S 为输入序列 x_1, x_2,…, x_n。

8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

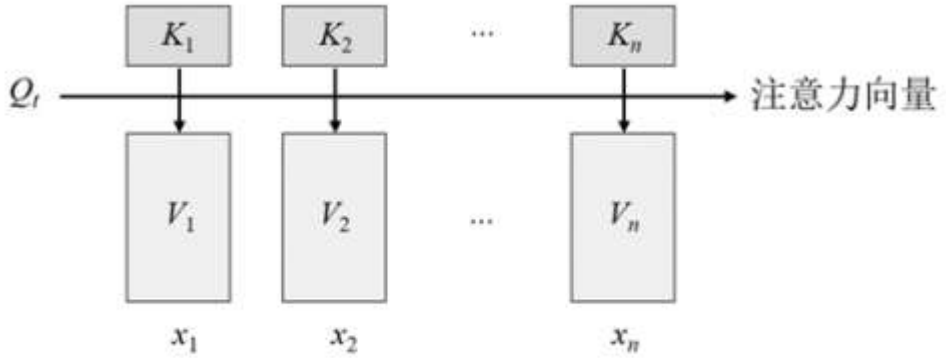


图 8-3 注意力向量的计算过程示意图

图8-3中，每个 x_i 可以视为一个由序键值对<K_i, V_i>构成，其中 K_i 可以表示输入 x_i 的特 征向量，V_i 表示在处理 x_i后产生的输出，这主要是在编码过程中完成。实际上，在很多情况下，K_i 和 V_i二者是相等的。在解码过程中，在时间步 t 的输入 y_t 相当于一个查询 Q_t。

8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

利用注意力向量 Att(Q_t, S)，可进一步构造循环神经网络的隐层输入，例如跟时间步 t 输入的查询编码进行拼接，然后再做线性变换等，表示如下：

$$\boldsymbol{h}_{t+1} = f(\boldsymbol{h}_t, Att(Q_t, S))$$

最后，送入循环神经网络，产生下一个元素的概率分布：

$$p(Q_t \mid Q_1, Q_2, \ldots, Q_{t-1}, S) = rnn(Q_{t-1}, \boldsymbol{h}_{t-1})$$

进而，利用 $p(Q_t \mid Q_1, Q_2, \ldots, Q_{t-1}, S)$计算损失函数值，不断优化网络参数。在测试阶段，将 Q_i 替换为 y_i，得到下列表达式：其中，$y_1, y_2, \ldots, y_{t-1}$ 表示已经生成的由 t-1 个元素构成的序列，$rnn(y_{t-1}, h_{t-1})$表示利用 y_{t-1}和 h_{t-1} 来生成第 t 个元素的概率分布 $p(y_t \mid y_1, y_2, \ldots, y_{t-1}, S)$，以此确定第 t 个元素。

$$p(y_t \mid y_1, y_2, \ldots, y_{t-1}, S) = rnn(y_{t-1}, \boldsymbol{h}_{t-1})$$

8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

按照一般 Seq2Seq 结构，对给定的一个源句子，网络模型先对源句子进行编码，形成语义向量，然后再对该语义向量进行解码，形成目标句子。假设给定的英文源句子是“I went to the game yesterday”，并假设该句子被编码器编码为一个有固定长度的语义向量 C，即 C 被假定包含了这个句子的全部语义信息。然后，利用 C 来生成中文目标句子“我昨天去参加比赛”中的每一个词，即每一个词的生成都利用同一个 C。这个翻译过程是没有运用注意力机制的，可表示如图 8-4 所示：

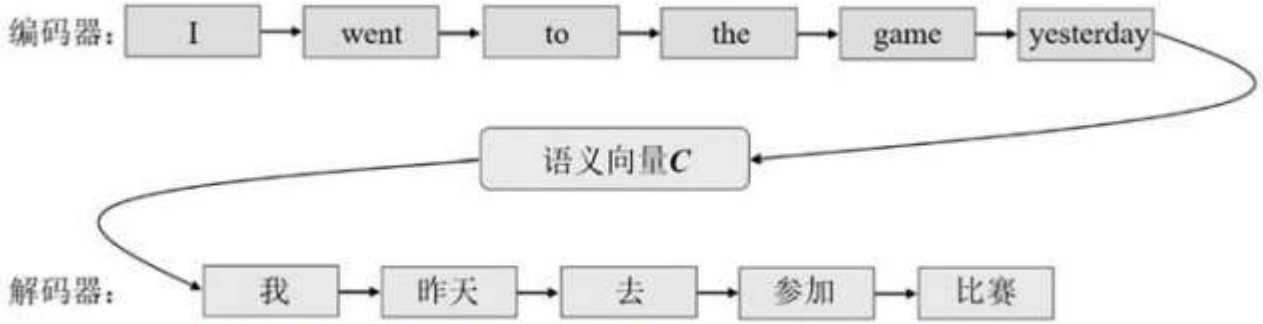


图 8-4　编码器-解码器工作示意图（无注意力机制）

8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

Seq2Seq 结构的缺陷（无注意力机制）：

（1）根据神经网络的特点，源句子中越在前面的单词，对形成 C 的影响就越小；如果源句子过长，那么源句子前面的单词的语义信息几乎就没有被编码到 C 当中，因而后面解码器就很难翻译出前面的单词的语义。

（2）当解码器生成中文目标句子中的每一个词时，都利用同一个 C；这意味着，不管生成哪个词，都按照同一信息量使用源句子中的每个单词，这显然不合理。例如，在生成“去”的时候，显然“went”和“to” 的作用是最大的，但这种模型未能考虑到这个情况。

（3）不管多长的句子，都将其包含的信息压缩到 C 当中，这使得这个 C 可能难以装载过多的信息，从而解码器难以翻译源句子的含义。这属于信息过载问题。

8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

Seq2Seq 结构（含注意力机制）：图 8-5 展示了在时间步 t=3 时生成“去”的情况。其中，假设学习到的参数向量为[0.0, 0.7, 0.3, 0.0, 0.0, 0.0]，由此生成语义向量 C3，进而生成“去”。这意味着在生成“去”时，主要使用了单词“went”，其次是“to”，而其他单词几乎可以忽略；生成不同的词，所使用的语义向量是不一样的，因而涉及的单词也不同，而且也只有少数的几个单词。这样，不管源句有多长，对生成结果的影响都不大。从这些分析可以看出，注意力机制可以较好解决前面提出的三个问题。

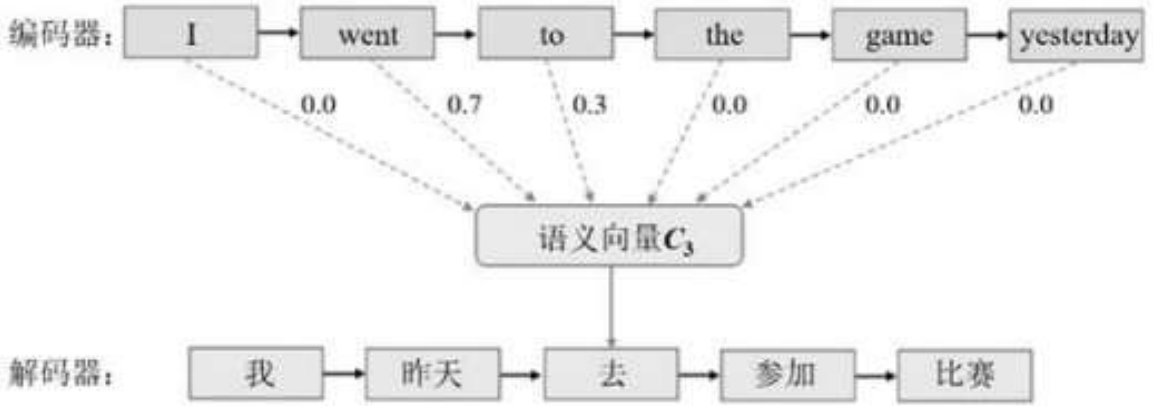


图 8-5 编码器-解码器工作示意图（含注意力机制）

8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

【例 8.2】改写例 8.1 中的 Seq2Seq 结构模型，将注意力机制运用于该翻译任务。

```
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(AttnDecoderRNN, self).__init__()
        self.embedding = nn.Embedding(output_size, hidden_size) #定义嵌入层
        #第 1 次拼接用到的全连接层
        self.attention = nn.Linear(hidden_size * 2, MAX_LENGTH)
        #第 2 次拼接用到的全连接层
        self.attention_com = nn.Linear(hidden_size * 2, hidden_size)
        self.dropout = nn.Dropout(0.1)
        self.gru = nn.GRU(hidden_size, hidden_size) #运用 GRU 作为循环神经网络
        #再做线性变换，以调整输出尺寸跟目标词汇数一样
        self.out = nn.Linear(hidden_size, output_size)
```

注：完整代码见教材224页

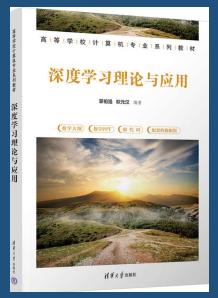
8.1 Seq2Seq结构与注意力机制

8.1.2 注意力机制

修改实例 Decoder 的创建代码，下列三处代码需要修改：

```
#Decoder 的创建代码：
decoder = AttnDecoderRNN(hidden_size,zh_lang.word_num).to(device)
#在训练过程调用 Decoder 的代码：
de_output, de_hidden = decoder(de_input, de_hidden, en_outputs) #添加 en_outputs
#在测试过程调用 Decoder 的代码：
de_output, de_hidden = decoder(de_x, de_hidden, en_outputs) #添加 en_outputs
```

运行修改后的程序，从结果可以看出：在同等条件损失函数值降低得更快。也可以将保存于张量 attention_weights 中的权重向量输出，进而利用 plt.matshow 函数来绘制注意力机制作用的效果图，以此查看注意力机制作用的效果。从例 8.1 和例 8.2 可以看出，传统的基于 Seq2Seq 结构的网络模型虽然可以处理自然语言，但效率比较低。其主要原因在于，这些网络模型在处理序列数据时，大多只能以串行方式进行，这对大规模海量数据而言是致命的。



本章内容

contents

8.2 Transformer及其在NLP中的应用

8.2.1 Transformer 中的注意力机制

Transformer 定义：Transformer 是谷歌公司于 2017 年提出来的一种只依靠注意力机制来处理序列数据的框架。Transformer 是完全依赖于注意力机制的一种序列数据处理框架，它没有包含任何 CNN和 RNN 的成分。其核心是使用了自注意力机制和多头注意力机制。

自注意力（Self-attention）是把同一个输入 x 通过线性变换分别映射为三个向量：

$$\boldsymbol{Q} = \boldsymbol{W}_q \cdot \boldsymbol{x}$$
$$\boldsymbol{K} = \boldsymbol{W}_k \cdot \boldsymbol{x}$$
$$\boldsymbol{V} = \boldsymbol{W}_v \cdot \boldsymbol{x}$$

然后再按照一般的注意力机制来计算 Q、K 和 V 的注意力向量：

$$Att(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Softmax}(\frac{\boldsymbol{Q} \cdot \boldsymbol{K}^T}{\sqrt{d_k}}) \cdot \boldsymbol{V}$$

8.2 Transformer及其在NLP中的应用

8.2.1 Transformer 中的注意力机制

多头注意力机制的概念：自注意力可以自主学习同一个序列内部的关系，因而可以获得更多深层次的信息。如果将上述映射得到的向量 Q、K 和 V 都分别进一步均等分割为 h 个部分：Q_i、K_i 和V_i，$i = 1, 2, \ldots, h$，然后对每一组 Q_i、K_i 和 V_i 运用上述相同的方法计算注意力向量 $Att(Q_i, K_i, V_i)$——这就形成了 h 头注意力（多头注意力），最后将结果拼接起来。已有研究表明，多头注意力机制不只可以自主学习内部关系，而且可以捕获更多维度的信息，效果优于单头注意力机制。图 8-6 是多头注意力机制示意图。

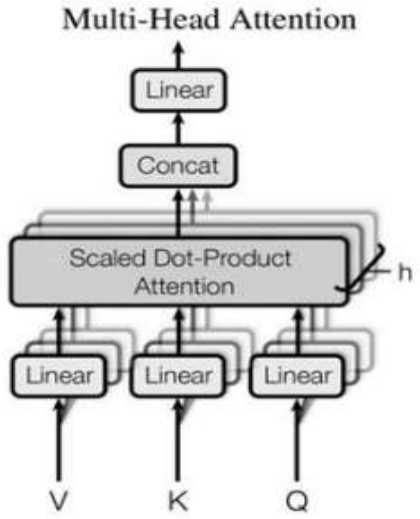


图 8-6　多头注意力机制示意图

8.2 Transformer及其在NLP中的应用

8.2.2 Transformer 的结构

Transformer 也是基于 Seq2Seq 结构，也由编码器和解码器两部分组成。Transformer编码器和解码器的结构图如图 8-7 所示（该图来自论文 Attention is all you need）。左边和右边的大方框分别是编码器和解码器的基本单元。编码器单元包括多头注意力模块、前馈网络模块和两个归一化模块；解码器单元主要比编码器单元多了一个多头注意力模块，其他部分基本相同。

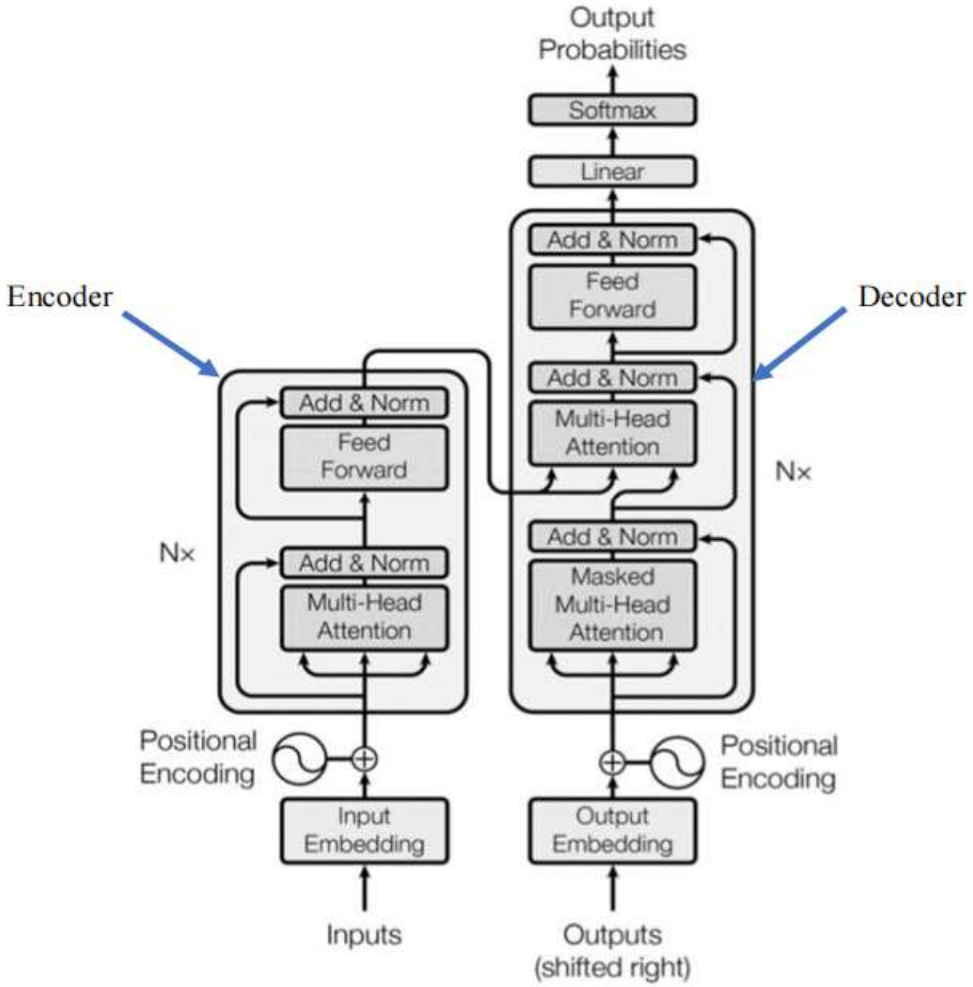


图 8-7 Transformer 编码器和解码器的结构图

8.2 Transformer及其在NLP中的应用

8.2.2 Transformer 的结构

整个编码器和解码器则分别由 N 个结构相同的编码器单元和解码器单元组成（默认 N=6），编码器输入 en_inputs 在依次经过 6 个编码器处理单元后，将结果发给 6个解码器单元；类似地，解码器输入 de_inputs 在依次经过 6 个解码器单元处理后，形成解码器输出de_outputs。

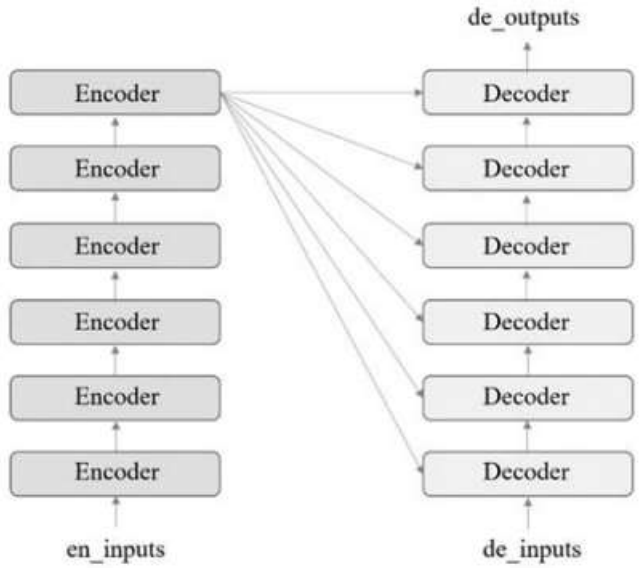


图 8-8 Transformer 中的编码器和解码器及其关系

8.2 Transformer及其在NLP中的应用

8.2.3 Transformer 的位置编码与嵌入

位置编码与嵌入（Positional Encoding）：与基于 RNN 网络的 Seq2Seq 模型不同Transformer 可以并行处理序列中的元素，因而其效率比较高，这得益于位置编码的引入。序列的串行处理方式实际上蕴含了各个元素的相对位置。当采用并行处理方式时，这种相对位置的自然蕴含就消失了，无法捕捉到序列顺序信息。但Transformer 引入的位置编码方法可为该问题提供了完美的解决方案。这里所提的位置是指一个元素在序列中的位置，一般用该元素的索引（下标值）来表示。例如，假设有一个由列表表示的序列：[2,6,10,12]，则其中的元素 2、6、10、12 的索引分别为 0、1、2、3，因此它们的位置分别为 0、1、2、3。可见，元素的位置跟元素的内容没有关系，只由元素的顺序确定。

8.2 Transformer及其在NLP中的应用

8.2.3 Transformer 的位置编码与嵌入

位置编码是使用一个函数将给定的位置（整数）映射为一个 d 维向量。这里用 PE 表示该函数，则 PE 的解析式如下：

$$PE(p,i)=\begin{cases}\sin\left(\dfrac{p}{10000^{\frac{i}{d}}}\right), & \text{如果}i\text{为偶数}\\ \cos\left(\dfrac{p}{10000^{\frac{i-1}{d}}}\right), & \text{如果}i\text{为奇数}\end{cases}$$

其中，$0\le p<P$ 表示序列中元素的位置索引，P 表示序列的长度，即位置的总数（一般为句子的最大长度），$0\le i<d$ 表示编码向量中分量的位置索引，d 为设置的位置向量的长度，一般与词向量的长度相等。通常，有两种方式为输入序列提供编码：一种是直接编码方式，另一种是嵌入方式。

8.2 Transformer及其在NLP中的应用

8.2.3 Transformer 的位置编码与嵌入

直接编码：这种方式的基本思路是，对于给定的输入 x（由词的索引构成的张量），利用其形状（句子的长度和句子的数量）和编码函数 PE(p, i)，即时生成 x 的编码（若干位置向量构成的张量）并返回。为实现这一思路，定义一个类及其实例来实现，相应代码如下：

```
class PosEncoding(nn.Module):
    def __init__(self, d_model, max_len):
        super(PosEncoding, self).__init__()
        self.max_len = max_len
        self.d_model = d_model #d_model 需为偶数
    def forward(self, x): #x 为由词的索引表示的张量
        p = torch.arange(0, self.max_len).float().unsqueeze(1) #产生所有的索引（位置）
        #以下根据函数 PE(p, i)，构建长度为 max_len 的序列的编码，
        #即形成 max_len 个位置向量，向量的长度为 d_model
        p_2i = torch.arange(0, self.d_model, 2)
        p_2i = 1./np.power(10000.0, (p_2i.float() / self.d_model))
        pos_code = torch.zeros(self.max_len, self.d_model)
        pos_code[:, 0::2] = torch.sin(p * p_2i)
        pos_code[:, 1::2] = torch.cos(p * p_2i)
```

注：完整代码见教材228页

8.2 Transformer及其在NLP中的应用

8.2.3 Transformer 的位置编码与嵌入

嵌入： （1）先利用函数 PE(p, i)生成所有位置的编码（位置向量），然后定义一个嵌入层并以此编码作为初始数据，同时冻结该层参数（位置参数不应该被更新）

```
#预先生成所有的位置向量，用于初始化嵌入层
def PosEncoding_for_Embedding(d_model, max_len):
    p = torch.arange(0, max_len).float().unsqueeze(1)
    p_2i = torch.arange(0, d_model, 2)
    p_2i = 1./np.power(10000.0, (p_2i.float() / d_model))
    pos_code = torch.zeros(max_len, d_model)
    pos_code[:, 0::2] = torch.sin(p * p_2i)
    pos_code[:, 1::2] = torch.cos(p * p_2i)
    pos_code = pos_code.to(device) #pos_code 的形状为(max_len, d_model)
    return pos_code
#定义嵌入层，用上述函数生成初始数据并冻结参数
self.pos_embedding = nn.Embedding.from_pretrained(
    PosEncoding_for_Embedding(d_model, MAX_LENGTH), freeze=True)
```

8.2 Transformer及其在NLP中的应用

8.2.3 Transformer 的位置编码与嵌入

（2）为利用嵌入层中的位置向量，对于由词索引构成的输入 x，需要改变其内容，将位置索引“改写”到 x 中，并返回改写结果，这样才能通过嵌入层获得相应的位置向量。

```
def pos_code(x):
    one_sen_poses = [pos for pos in range(x.size(1))]
    all_sen_poses = torch.LongTensor(one_sen_poses).unsqueeze(0).to(device)
    all_sen_poses = all_sen_poses.repeat(x.size(0),1)
    return all_sen_poses
```

不管是用直接编码方式还是用嵌入方式，对给定的输入 x，都会得到相应的位置向量。把该位置向量和词嵌入层输出的向量相加，所得结果即可以作为 x 的嵌入向量表示，进而可以送入编码器或解码器作进一步处理。由于每个 x 的词嵌入向量都加上了相应的位置向量，因此即便词嵌入向量相同，但它们所处的位置不同，因而位置向量不同，从而最终的向量表示也不会相同。这样就解决了自注意力机制无法对序列进行建模的问题，也使得并行处理同一个序列中的元素成为可能。

8.2 Transformer及其在NLP中的应用

8.2.4 Transformer 的使用方法

Transformer 是一种处理序列数据的框架，可以按照这个框架去构建 Transformer 程序。但 Transformer 框架比较复杂，从零开始构建是非常繁琐的。可以利用 torch.nn 模型库中封装好的 nn.Transformer()类来开发 Transformer 程序，这样会事半功倍。nn.Transformer()主要实现了 Transformer 框架的编码和解码功能。它有两个输入和一个输出，分别是编码器的输入、解码器的输入和解码器的输出，

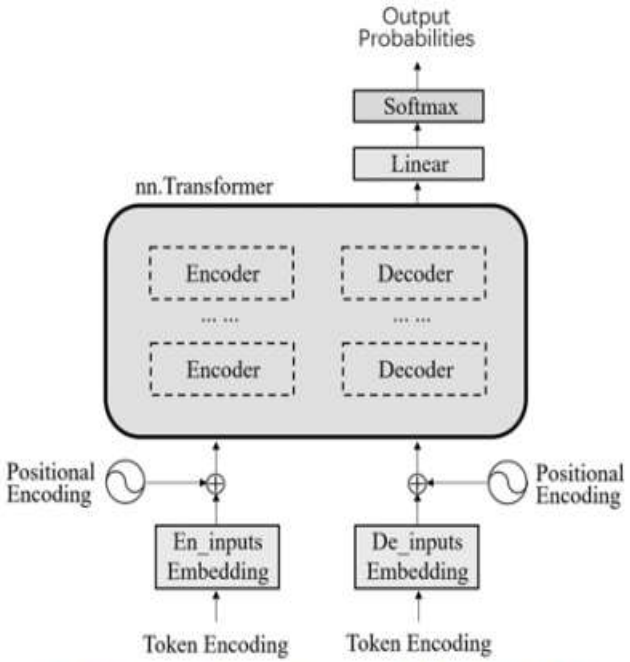


图 8-9 nn.Transformer()的功能示意图

8.2 Transformer及其在NLP中的应用

8.2.5 Transformer 应用案例

Transformer 框架可以用于解决许多 NLP 任务，下面给出一个用 Transformer 实现英中文翻译的例子。

【例 8.3】构建一个基于 Transformer 框架的神经网络模型，用于实现例 8.2 的功能，即完成英文到中文的翻译。

（1）**加载数据**：首先从文件 en_zh_data.txt 中读取数据，以英文-中文句子对的形式保存到列表 pairs 当中：

```
fg = open(path + '\\' + "en_zh_data.txt", encoding='utf-8')
lines = list(fg)
fg.close()
pairs = []
for line in lines:
    line = line.replace('\n', '')
    pair = line.split('--->') #英中文句子以字符串'--->'隔开
    if len(pair) != 2:
        continue
    en_sen = pair[0] #英文句子
    zh_sen = pair[1] #中文句子
```

注：完整代码见教材233页

8.2.5 Transformer 应用案例

（2）**定义字典类**：定义词表类，以用于索引编码。相关代码如下：

```
class Word_Dict:
    def __init__(self, name):
        self.name = name
        self.word2index = {"<PAD>": 0, "<UNK>": 1, "<SOS>": 2, "<EOS>": 3}
        self.index2word = {0: "<PAD>", 1: "<UNK>", 2: "<SOS>", 3: "<EOS>"}
    def addOneSentence(self, sentence):
        if self.name == 'eng':
            for word in sentence.split(' '): #英文用 split(' ')分词
                self.addOneWord(word)
        elif self.name == 'chi': #中文用 jieba 分词
            split_chi = [char for char in jieba.cut(sentence) if char != ' ']
            for word in split_chi:
                self.addOneWord(word)
```

注：完整代码见教材233页

8.2 Transformer及其在NLP中的应用

8.2.5 Transformer 应用案例

（3）**构建英文和中文词汇的词表实例**：分别形成英文词表和中文词表，为每个词汇确定一个唯一的索引。相关函数代码如下：

```
def getData(pairs):
    temp = []
    for pair in pairs:
        split_eng = pair[0].split(' ') #切分英文单词
        split_chi = [word for word in jieba.cut(pair[1]) if word != ' '] #中文分词
        if len(split_eng) < MAX_LENGTH and len(split_chi) < MAX_LENGTH:
            temp.append(pair) #保留长度小于 MAX_LENGTH 的句子对
    pairs = temp
    eng_lang = Word_Dict('eng') #初始化英文词表
    chi_lang = Word_Dict('chi') #初始化中文词表
    for pair in pairs: #对每个句子对构造词表
        eng_lang.addOneSentence(pair[0]) #建立英文单词索引词表
        chi_lang.addOneSentence(pair[1]) #建立中文词索引词表
    return eng_lang, chi_lang, pairs #返回构造好的英文词表和中文词表以及句子对
```

8.2 Transformer及其在NLP中的应用

8.2.5 Transformer 应用案例

（4）**编码**：利用英文词表和中文词表，对各个句子进行索引编码，并对结果等长化和张量化。

```
def sentence2tensor(lang, sentence,flag):
    indexes = []
    if flag=='encoder_in': #编码器的输入（英文句子）
        words = [word for word in sentence.split(' ') if word.strip() != ''] #分词
        words = words[0:MAX_LENGTH]
        words = words + ['<PAD>']*(MAX_LENGTH-len(words)) #等长化
        indexes = [lang.word2index.get(word, 1) for word in words]
    #1 为'<UNK>'的索引号
    elif flag == 'decoder_in': #解码器的输入（中文句子）
        words = [word for word in jieba.cut(sentence) if word.strip() != ''] #分词
        words = ['<SOS>'] + words
        words = words[0:MAX_LENGTH]
        words = words + ['<PAD>'] * (MAX_LENGTH - len(words)) #等长化
        indexes = [lang.word2index.get(word, 1) for word in words]
```

注：完整代码见教材234页

8.2 Transformer及其在NLP中的应用

8.2.5 Transformer 应用案例

（5）**定义数据集类**：调用 sentence2tensor()函数来完成各个句子的索引编码，并张量化。定义代码如下：

```
class MyDataSet(Dataset):
    def __getitem__(self, idx):
        pair = self.pairs[idx]
        en_sentence = pair[0] #英文句子
        zh_sentence = pair[1] #中文句子
        #传入英文词表和英文句子，返回英文句子的张量（由单词的索引构成）
        en_input = sentence2tensor(eng_lang, en_sentence, flag='encoder_in')
        #传入中文词表和中文句子，返回中文句子的张量（由词的索引构成）
        de_input = sentence2tensor(chi_lang, zh_sentence, flag='decoder_in')
        #传入中文词表和中文句子，返回中文句子的张量，是解码器期望的输出
        #（相当于标记）
        de_output = sentence2tensor(chi_lang, zh_sentence, flag='decoder_out')
        return en_input, de_input, de_output
```

注：完整代码见教材235页

8.2 Transformer及其在NLP中的应用

8.2.5 Transformer 应用案例

（6）**定义实现翻译任务的类**：其主要功能是对送入 Transformer 前的索引张量进行预处理，如位置编码、词嵌入表示等，形成可以输入 Transformer 进行处理的张量；调用Transformer对输入的张量进行处理；对输出 Transformer 的张量进行线性变换，以符合生成目标词汇的要求。

8.2 Transformer及其在NLP中的应用

8.2.5 Transformer 应用案例

```
class MyTransformer(nn.Module):
    def __init__(self, d_model, nhead, layer_num, dim_ff, src_vocab_size, tgt_vocab_size):
        super(MyTransformer, self).__init__()
        #利用调用 nn.Transformer()来实例化类的对象，构建 Transformer 模型
        self.transformer = nn.Transformer(d_model=d_model, nhead=nhead,
        num_encoder_layers=layer_num,
        num_decoder_layers=layer_num,
        dim_feedforward=dim_ff)
        #定义面向英文单词的嵌入层
        self.src_embedding = nn.Embedding(src_vocab_size, d_model)
        #定义面向中文词的嵌入层
        self.tgt_embedding = nn.Embedding(tgt_vocab_size, d_model)
        self.pos_encoding = PosEncoding(d_model, max_len=MAX_LENGTH)
        #在本例中源句子和目标句子的最大长度设置为一样长，
        #故可共享编码函数 PosEncoding_for_Embedding（编码的嵌入方式）
        #self.pos_embedding = nn.Embedding.from_pretrained(\
        #PosEncoding_for_Embedding(d_model, MAX_LENGTH), freeze=True)
        self.fc = nn.Linear(d_model, tgt_vocab_size, bias=False)
```

注：完整代码见教材235页

8.2.5 Transformer 应用案例

（7）设置训练主要参数：

```
d_model = 256 #嵌入向量的长度
nhead = 4 #多头注意力的头数
layer_num = 2 #编码器和解码器的层数
dim_ff = 512 #FeedForward 的维度，隐含层神经元个数
src_vocab_size = len(eng_lang.word2index) #英文单词数量
tgt_vocab_size = len(chi_lang.word2index) #中文词的数量
transformer_model = MyTransformer(d_model=d_model, nhead=nhead,
                                   layer_num=layer_num, dim_ff=dim_ff,
src_vocab_size=src_vocab_size,
                                   tgt_vocab_size=tgt_vocab_size).to(device)
```

注：完整代码见教材236页

8.2 Transformer及其在NLP中的应用

8.2.5 Transformer 应用案例

（8）测试模型：

```
while True:
    #解码器输入最开始为标志位"<SOS>"，然后逐个拼接新生成的词，直到遇到
    #结束标识符"<EOS>"，最后得到的 de_input 即为翻译的句子（的编码）
    de_input = torch.cat([de_input.detach(), torch.tensor([[next_index]]).to(device)],
-1)
    de_pre_y = transformer_model(en_input, de_input) #调用 Transformer 实例
    prob = de_pre_y.max(dim=-1, keepdim=False)[1]
    next_index = prob.data[-1]
    if next_index == chi_lang.word2index["<EOS>"]:
        break
word_indexes = de_input.squeeze().cpu()
out_words = [chi_lang.index2word[index.item()] for index in word_indexes]
out_sentence = ' '.join(out_words[1:])
print('翻译得到的句子：', out_sentence)
```

注：完整代码见教材237页

8.2 Transformer及其在NLP中的应用

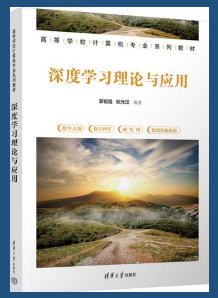
8.2.5 Transformer 应用案例

执行上述代码构成的.py 程序，在笔者计算机上输出下列句子：

```
翻译得到的句子： 然而 作为 地域 战略 学家 无论是 从 政治 意义 还是 从 经济
意义 上 让 我 自然 想到 的 年份 是 1989 年
```

这也说明，该程序基本上能够翻译简单的英文句子，但翻译水平有待进一步提高——需要更多的数据来训练，同时需要不断完善相关参数。

我们注意到，同样的数据量，Transformer 程序会比循环神经网络程序快得多。这是由 于 Transformer 采用了位置编码，使得待处理数据可以并行地送入 Transformer 进行计算。但是，Transformer 不宜处理过长的序列（如长度超过 50 的序列），否则其训练时间会急剧增 加。也就是说，在处理长文本（如长度超过 200）时，LSTM 等传统循环神经网络比 Transformer 会表现更好的效果。



本章内容

contents

8.3 BERT 及其在 NLP 中的应用

8.3.1 关于 BERT

BERT的概念：BERT（Bidirectional Encoder Representations from Transformer）是谷歌公司于2018 年 10 月发布的一种语言表示模型。BERT 是一种面向 NLP 任务的大型预训练模型，有上亿 个参数。它是利用维基百科和书籍语料组成的大规模语料进行训练而得到。BERT 的出现是 自然语言处理领域中的一个里程碑事件，预示自然语言处理进入一个新的时代。

BERT 的结构：BERT 完全是在 Transformer 框架的基础上构建的，它包含了两层双向 Transformer 模型， 多头注意力机制仍然是其核心部件。但BERT只利用了Transformer的左边部分——编码器， 主要用于学习序列的特征，但没有解码器，因而不利于文本生成。

8.3 BERT 及其在 NLP 中的应用

8.3.1 关于 BERT

BERT 的输入可以是一个句子也可以是一个句子对，且都不需要对句子进行标记，它使用 WordPiece 对句子进行切分。WordPiece 跟一般的单词切分和中文分词不一样。对于英文句子，WordPiece 会把单词本身和时态表示拆分开来。例如，work、worked 和 working 这三个单词分别被拆分成 work、work 和 ##ed、work 和##ing。这种拆分方法不但可以减少词表的大小，而且还可以提升单词的区分度。对于中文句子，WordPiece 则按字对句子进行切分，实际上是将一个句子拆分成为一系列的字和标点符号。由于 BERT 主要是用英文语料训练的，因此 BERT 对中文的处理没有对英文处理有那么好的效果。

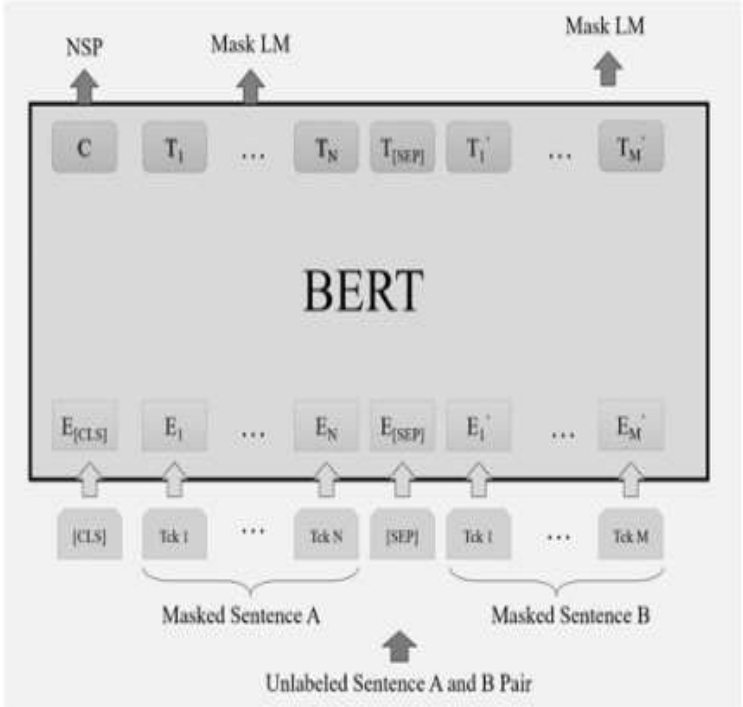


图 8-11 BERT 的结构

8.3 BERT 及其在 NLP 中的应用

8.3.1 关于 BERT

BERT 预训练任务：

- 掩码语言模型（Masked Language Model, MLM）；
- 下一句子预测（Next Sentence Prediction, NSP）。

为使用预训练模型 BERT 及后面介绍的 GPT，需要先安装 pytorch_transformers：

```
pip install pytorch_transformers==1.0
```

8.3 BERT 及其在 NLP 中的应用

8.3.1 关于 BERT

1. MLM 模型

该模型将输入文本中的部分单词进行掩码（mask），即以[mask]替换被掩码的单词，同时用[CLS]表示句子的开始，用[SEP]隔开不同的句子或标志句子的结束，（训练时一次输入一个句子对，即两个句子）。例如，如果输入下列句子对“我是中国人。我爱我的祖国！”，则表示成下列输入格式：

texts = ['[CLS]我是中国人。[SEP]我爱我的[MASK]国！[SEP]’]

其中，“祖”被掩码了。

BERT 可以接收一个句子或一个句子对。输入的句子必须以'[CLS]'开始，如果有两个句子则'[SEP]'隔开。然后，对输入的文本进行切分（token 化）、索引编码。

8.3 BERT 及其在 NLP 中的应用

8.3.1 关于 BERT

对 texts 进行token 化：

```
from pytorch_transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained('bert-base-chinese’)   #加载词表
tokenized_texts = [tokenizer.tokenize(word) for word in texts]     #token 化
```

结果得到下列内容（即 tokenized_texts 的内容）：
[['[CLS]', '我', '是', '中', '国', '人', '。', '[SEP]', '我', '爱', '我', '的', '[MASK]', '国', '！', '[SEP]']]
这个结果是按字来切分的，这跟分词不一样。进一步对其进行索引编码，可调用下列代码来实现：

```
input_ids = [tokenizer.convert_tokens_to_ids(token) for token in tokenized_texts]
input_ids = torch.LongTensor(input_ids)
```

8.3 BERT 及其在 NLP 中的应用

8.3.1 关于 BERT

结果，input_ids 的内容为：

tensor([[101, 2770, 3222, 705, 1745, 783, 512, 102, 2770, 4264, 103, 4639, 4863, 1745, 8014, 102]])

从 token 化到索引编码，整个过程都是调用 BertTokenizer 模块来完成的。当然，为了构造一个数据批量（batch），还需要对 input_ids 进行填充，实现等长化。此后input_ids 就可以输入 BERT 进行训练了。MLM 模型正是通过对某些词进行掩码，然后训练模型，使之可以根据上下文来预测被掩码的词。这样，只要输入无标记的文本，我们就可以得到能够预测部分残缺词的模型，实现无监督学习。这就是 MLM 模型的任务。

8.3 BERT 及其在 NLP 中的应用

8.3.1 关于 BERT

2. NSP 模型

构造样本集：BERT 还有一个功能就是对给定一个句子，预测下一个句子。为此，需要给 BERT 输入一系列的句子对，以此来训练模型。从语料库中选择相邻的两个句子 A 和 B，然后由 A 和 B 组成一个训练样本。例如，'[CLS]我是中国人。[SEP]我爱我的[MASK]国！[SEP]'就是一个训练样本。

当 A 和 B 的顺序跟语料库中的原始顺序一样时，相应的样本称为正样本，否则称为负样本。一般通过随机调整 A 和 B 的顺序来构造正样本和负样本，并维持正负样本各占大约50%，以保持类平衡。

NSP 模型利用这样的样本集来训练，使得该模型可以预测两个句子的顺序是否正确。

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

加载预训练模型 BERT:

```
from pytorch_transformers import BertTokenizer, BertModel
#处理中文文本
model = BertModel.from_pretrained('bert-base-chinese', cache_dir="./Bert_model")
#处理英文文本
#model = BertModel.from_pretrained('bert-base-uncased', cache_dir="./Bert_model")
```

在 Transformer 模型库中，预训练模型一般有 3 类文件，分别是模型参数文件、词表文件和配置文件。模型参数文件是以二进制格式保存，而词表文件和配置文件是文本文件。下载时，应确保要下载这三类文件。

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

BERT 模型的 forward()含有多个参数，其中对常用的参数说明如下：

- **input_ids**：接收由 token 在词表中的索引构成的张量。该选项为必选项目，形状为(batch_size, seq_length)，其中 batch_size 和 seq_length 分别表示当前批量的大小和序列的长度。
- **attention_mask**：用于标识 input_ids 中哪些元素是填充值（填充值没有实际意义，不参与注意力计算）。当 attention_mask 中某一位置的值为 0，则表示 input_ids 中对应位置的元素为填充值，为 1 表示为非填充值。该选项为可选项，其形状为(batch_size, seq_length)；如果缺省，则表示 input_ids 中的元素均为非填充值。
- **token_type_ids**：用于标识当前 token 属于哪一句子，0 表示属于第一个句子，1 表示属于第二个句子。该选项为可选项，其形状为(batch_size, seq_length)。当输入序列中包含两个句子时，需要使用该选项。如果该选项缺省，则表示每个序列中只包含一个句子。

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

BERT 的输出：forward()函数的返回值即为 BERT 模型的返回值。默认情况下，该函数返回值是一个元组（tuple），其中包含两个张量。假设用 outputs 表示 forward()的返回值，则 outputs 为一个元组，其中第一个元素 outputs[0]和第二个元素 outputs[1]都是张量，它们形状分别为(batch_size, seq_length, hidden_size)和(batch_size, seq_length)，而 hidden_size 固定等于 768。

- outputs[0]是 BERT 最后一层输出的隐层状态，该状态向量多用于进一步微调。
- outputs[1]是 BERT 最后一层输出的第一个 token(classification token)的隐层状态（对应于特殊标记符号[CLS]的输出向量），通常以该状态向量作为句子的特征，送入全连接网络进行分类。

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

输入 BERT 的是索引编码后形成的张量。BERT 可以接收一个句子的输入，或者接收两个句子的输入。下面通过一个例子来说明如何构造输入张量，即在调用 BERT 模型之前如何进行数据预处理。假设欲向 BERT 中输入如下两条语句：

```
我是中国人。我爱我的祖国！
他努力！他学习英语。
```

数据预处理的过程如下：

（1）添加特殊符号。在句子中添加特殊标记符号[CLS]、[SEP]，其中[CLS]表示文本的起始符号，[SEP]表示句子的结束符号。添加结果如下：

```
[CLS]我是中国人。我爱我的祖国！[SEP]
[CLS]他努力！他学习英语。[SEP]
```

在上述输入中，将“我是中国人。我爱我的祖国！”被视为一个句子，“他努力！他学习英语。”也被视为一个句子。

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

如果将“我是中国人。我爱我的祖国！”视为两个句子，分别是句子“我是中国人。”和句子“我爱我的祖国！”，则相应输入应该表示为：

```
[CLS]我是中国人。[SEP]我爱我的祖国！[SEP]
```

类似地，如果“他努力！他学习英语。”也被视为两个句子：“他努力！”和“他学习英语。”，则相应输入应该表示为：

```
[CLS]他努力！[SEP]他学习英语。[SEP]
```

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

（2）填充至固定长度。在使用 BERT 时，需要确定句子的固定长度（句子中 token 的个数）。这里设置固定长度为 20，不够 20 的，就在句子后面添加特殊符号[PAD]，它表示句子的填充符号，其索引值为 0。填充结果如下：

```
[CLS]我是中国人。[SEP]我爱我的祖国！[SEP][PAD][PAD][PAD][PAD]
[CLS]他努力！[SEP]他学习英语。[SEP][PAD][PAD][PAD][PAD][PAD][PAD][PAD]
```

填充后，上述每个句子的长度都为 20。注意，[CLS]、[SEP]和[PAD]都只算一个 token。

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

（3）token 化。将填充完的两个句子保存在变量 texts 中，然后对其中的每个句子进行 token 化，即切分为一系列的字：

```
from pytorch_transformers import BertTokenizer, BertModel
texts = ['[CLS]我是中国人。[SEP]我爱我的祖国！[SEP][PAD][PAD][PAD][PAD]',
'[CLS]他努力！[SEP]他学习英语。[SEP][PAD][PAD][PAD][PAD][PAD][PAD]' +'[PAD]']
tokenizer = BertTokenizer.from_pretrained('bert-base-chinese')
tokenized_texts = [tokenizer.tokenize(word) for word in texts] #token 化
```

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

这时，tokenized_texts 的内容如下：

```
[['[CLS]', '我', '是', '中', '国', '人', '。', '[SEP]', '我', '爱', '我', '的', '祖', '国', '！', '[SEP]', '[PAD]',
'[PAD]', '[PAD]', '[PAD]'], ['[CLS]', '他', '努', '力', '！', '[SEP]', '他', '学', '习', '英', '语', '。', '[SEP]',
'[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]']]
```

（4）索引编码。进一步对 tokenized_texts 的内容进行索引编码并张量化：

```
input_ids = [tokenizer.convert_tokens_to_ids(token) for token in tokenized_texts]#索引编码
input_ids = torch.LongTensor(input_ids) #张量化
```

这时，张量 input_ids 的内容如下：

```
tensor([[101, 2770, 3222, 705, 1745, 783, 512, 102, 2770, 4264, 2770, 4639, 4863, 1745,
8014, 102, 0, 0, 0, 0], [101, 801, 1223, 1214, 8014, 102, 801, 2111, 740, 5740, 6428, 512, 102, 0,
0, 0, 0, 0, 0, 0]])
```

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

上述代码编写的思路很清晰，但略显啰嗦，可用下面更为简洁的代码实现上述编码功能：

```
input_ids2 = [tokenizer.encode(text) for text in texts]
input_ids2 = torch.LongTensor(input_ids2)
#texts2 = tokenizer.decode(input_ids2[0].tolist()) #解码获得原文本
```

其中，input_ids2 和 input_ids 的内容是完全一样的。input_ids 中这些整数就是各个 token 的索引，它们对 BertTokenizer 而言是固定的。同时可以看到，三个特殊标记符号[CLS], [SEP]和[PAD]的索引分别为 101, 102 和 0。

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

（5）构造注意力掩码矩阵。矩阵中，0 表示对应的位置为填充符号，不需要参与注意力计算：

```
attention_mask = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0],
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]]
attention_mask = torch.tensor(attention_mask)
```

其中，input_ids2 和 input_ids 的内容是完全一样的。input_ids 中这些整数就是各个 token 的索引，它们对 BertTokenizer 而言是固定的。同时可以看到，三个特殊标记符号[CLS], [SEP]和[PAD]的索引分别为 101, 102 和 0。

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

（6）构造句子掩码矩阵。矩阵中，0 表示第一个句子中的 token，1 表示第二个句子中的 token：

```
token_type_ids = [[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
token_type_ids = torch.tensor(token_type_ids)
```

8.3 BERT 及其在 NLP 中的应用

8.3.2 BERT 的使用方法

（7）加载 BERT 模型并将预处理后的句子送入 BERT 模型：

```
from pytorch_transformers import BertModel
model = BertModel.from_pretrained('bert-base-chinese', cache_dir="./Bert_model")
#调用 BERT 模型对输入的两个句子对进行处理
outputs = model(input_ids=input_ids,
                attention_mask=attention_mask,
                token_type_ids=token_type_ids)
```

BERT 模型返回的 outputs 是一个元组（tuple），包含两个元素，其中第一个元素 outputs[0] 的形状为(2, 20, 768)，第二个元素 outputs[1] 的形状为(2, 768)，其中 2 为批量大小，20 为句子的固定长度，768 为表示句子中每个 token（此处为字）的向量的长度。前者一般用于进一步微调，后者一般用作输入句子（对）的特征向量。

8.3 BERT 及其在 NLP 中的应用

8.3.3 基于 BERT 的文本分类

【例 8.4】 利用预训练模型 BERT 实现一个文本分类的网络程序。

数据集：从清华大学自然语言处理实验室网站（http://thuctc.thunlp.org/）。网站上下载THUCNews.zip文件，解压后从中随机取 18 万条新闻标题构成文本文件 train.txt、1 万条构成文件 test.txt，分别用做训练集和测试集。这些数据一共分为 10 个类别，分别是金融、房地产、股票、教育、科学、社会、政治、体育、游戏和娱乐等 10 个新 闻类别，其类别索引分别是 0, 1, …, 9。在文件 train.txt 和文件 test.txt 中，每条样本占一行，一行中前面是文本内容，后面是类别索引，中间用制表键（\t）隔开。数据集的存储结构如图 8-12 所示

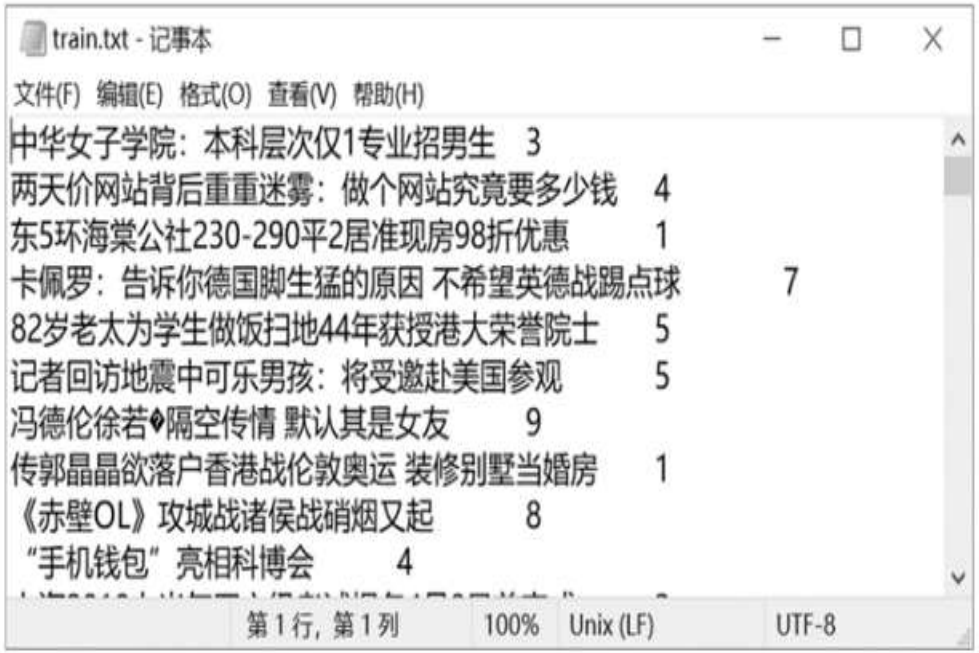


图 8-12 数据集的保存格式

8.3 BERT 及其在 NLP 中的应用

8.3.3 基于 BERT 的文本分类

按如下步骤来编写该神经网络程序：

（1）**定义函数 getTexts_Labels(fn)**：从指定 txt 文件中读取数据，将文本和类别索引分开，以字符串的形式保存文本，保存在列表 texts 中；以整数的形式保存类别索引，保存在列表 labels 中。texts 和 labels 中的元素一一对应。

```
def getTexts_Labels():
    with open(fn, 'r', encoding='utf-8') as f:
        lines = list(f)
    texts,labels = [],[]
    for line in lines:
        line = line.strip().replace('\n','')
        line = line.split('\t')
        if len(line)!=2:
            continue
        text,label = line[0],line[1]
        texts.append(text)
        labels.append(int(label))
    return texts, labels
```

8.3 BERT 及其在 NLP 中的应用

8.3.3 基于 BERT 的文本分类

（2）**定义函数 equal_len_coding()**：从保存文本的列表中读取文本，然后按字对其进行切分（token 化），再对每个 token 进行索引编码，最后进行等长化和张量化。

```
def equal_len_coding(texts):
    train_tokenized_text = [tokenizer.tokenize(sentence) for sentence in texts]        #token 化
    #按索引对每个 token 进行编码
    input_ids = [tokenizer.convert_tokens_to_ids(char) for char in   train_tokenized_text]
    #input_ids 中列表的长度可能不一样，下面代码通过截取和填充，将每个列表
    #设置为固定长度 MAX_LEN
    for i in range(len(input_ids)):
        tmp = input_ids[i]
        input_ids[i] = tmp[:MAX_LEN] #截取
        #填充 0，使之达到固定长度 MAX_LEN，其中 0 是填充符号<PAD>的索引
        input_ids[i].extend([0] * (MAX_LEN - len(input_ids[i])))
    input_ids = torch.LongTensor(input_ids) #张量化
    return input_ids
```

8.3 BERT 及其在 NLP 中的应用

8.3.3 基于 BERT 的文本分类

（3）**数据处理**：利用上面两个函数从磁盘文件中读取数据，然后利用 WordPiece 工具切分文本，接着进行索引编码、等长化和张量化，最后打包数据。

```
MAX_LEN = 50 #设置句子的最大长度
batch_size = 32
tokenizer = BertTokenizer.from_pretrained('bert-base-chinese') #加载WordPiece工具
path = r'./data/THUCNews'
name = r'train.txt'
fn = path + '\\' + name
train_texts, train_labels = getTexts_Labels(fn) #读取训练集
name = r'test.txt'
fn = path + '\\' + name
test_texts, test_labels = getTexts_Labels(fn) #读取测试集
train_input_ids = equal_len_coding(train_texts) #切分文本，索引编码，等长化和张量化
train_labels = torch.LongTensor(train_labels)
test_input_ids = equal_len_coding(test_texts)
test_labels = torch.LongTensor(test_labels)
#分别对训练集和测试集进行打包
train_set = TensorDataset(train_input_ids,train_labels)
train_loader = DataLoader(dataset=train_set,batch_size=batch_size,shuffle=True)
test_set = TensorDataset(test_input_ids,test_labels)
test_loader = DataLoader(dataset=test_set,batch_size=batch_size,shuffle=True)
```

8.3 BERT 及其在 NLP 中的应用

8.3.3 基于 BERT 的文本分类

（4）**定义文本处理类**：实现文本的特征学习和文本的分类功能，其中加载了用于处理中文文本的 BERT 模型。

```
class Bert_Model(nn.Module):
    def __init__(self):
    super(Bert_Model, self).__init__()
    #加载预训练模型
    self.model = BertModel.from_pretrained('bert-base-chinese’,
                  cache_dir="./Bert_model").to(device)
    self.dropout = nn.Dropout(0.1)
    self.fc = nn.Linear(768, 10) #有 10 个类
    def forward(self, x, attention_mask=None):
        outputs = self.model(input_ids=x,
                  attention_mask=attention_mask,
                  token_type_ids=None #只有一句话，故不需设置该参数
                  )
        o = outputs[1] #取池化后的结果，形状为(batch_size, 768)
        o = self.dropout(o)
        o = self.fc(o) #分类
        return o
```

8.3 BERT 及其在 NLP 中的应用

8.3.3 基于 BERT 的文本分类

（5）编写训练代码并进行训练：

```
bert_Model = Bert_Model().to(device)
optimizer = optim.Adam(bert_Model.parameters(), lr=1e-5)
for ep in range(5):
    for i, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        #构造注意力掩码矩阵
        mask = data.data.eq(0) #0 为[PAD]的索引
        mask = mask.logical_not().byte() #转化为 0,1 矩阵
        output = bert_Model(data,mask) #调用 BERT 模型
        loss = nn.CrossEntropyLoss()(output, target) #计算损失函数值
        if i%10==0:
            print(ep+1,i,len(train_loader.dataset),loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

8.3 BERT 及其在 NLP 中的应用

8.3.3 基于 BERT 的文本分类

（6）利用测试集对训练好的模型 bert_Model 进行测试，代码如下：

```
bert_Model.eval()
correct = 0
for i, (data, target) in enumerate(test_loader):
    data,target = data.to(device),target.long().to(device)
    mask = data.data.eq(0)
    mask = mask.logical_not().byte()
    output = bert_Model(data,mask) #调用训练好的模型 bert_Model
    pred = torch.argmax(output, 1)
    correct += (pred == target).sum().item() #统计正确预测的样本数
print('正确分类的样本数：{}，样本总数：{}，准确率：{:.2f}%'.format(correct,
len(test_loader.dataset), 100. * correct / len(test_loader.dataset)))
```

8.3 BERT 及其在 NLP 中的应用

8.3.3 基于 BERT 的文本分类

在笔者计算机上，预测输出的结果如下：

```
… …
5 5590 180000 0.09294451028108597
5 5600 180000 0.10058289766311646
5 5610 180000 0.08927912265062332
5 5620 180000 0.03275177627801895
正确分类的样本数：8938，样本总数：10000，准确率：89.38%
```

结果显示，在 10 个类别的中文数据集上，BERT 仍然获得比较高的准确率，而且编写的代码量比较少。这得益于预训练模型 BERT 强大的上下文表示能力。

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

【例 8.5】 利用预训练模型 BERT 实现一个阅读理解的网络程序。

单篇章抽取式阅读理解的任务可描述为：对于一个给定的问题 query 和一个篇章 passage，机器需要根据篇章内容，给出该问题的答案 answer。因此，在训练阶段，数据集中的样本具有三元组<query, passage, answer>的结构。例如，下面是一个样本的例子：

> **问题 query**：乾隆通宝一枚多少钱？
>
> **篇章 passage**：您好，根据七七八八收藏上乾隆通宝的价位，目前大概价位在几十到上千之间，这个和您商品品相和您的商品类别有很大的关系建议您可进入七七八八收藏上进行查看，同时您亦可注册商店自行销售，不仅可出售钱币还可以出售其他老旧物品，如书籍，旧海报，旧报纸，各种老旧物品，您可进入查看，谢谢！
>
> **答案 answer**：几十到上千。

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

在开发该程序时，有几个关键技术问题需要注意：

（1）在用 BERT 开发阅读理解程序时，需要解决的一个关键问题是答案 answer 起止位置索引在编码前后的映射问题。具体地，在将 query 和 passage 输入 BERT 之前，需要对它们进行索引编码。但是在进行索引编码后，答案 answer 的起止位置索引可能发生变化。例如，对于下列样本：

```
query = '谁去参加比赛？'
passage = '2022 年，张三去参加比赛。'  #注：'2022'和'年'之间有一个空格
answer = '张三'
```

answer 在 passage 中起止位置索引分别是 7 和 11，即(start_ind, end_ind) = (7, 11)。但是，在调用 BERT 的分词器 WordPiece 对 passage 和 answer 进行分词后，分别得到下面的结果：

```
['2022', '年', '，', '张', '三', '去', '参', '加', '比', '赛', '。']
['张', '三']
```

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

获取索引映射关系：在上述结果中，answer 被切分为'张'和'三'，它们在['2022', '年', '，', '张', '三', '去', '参', '加', '比', '赛', '。']中的位置索引分别为 3 和 4。也就是说，(start_ind, end_ind)由分词前的(7, 11)变为分词后的(3, 4)。造成这个变化的原因主要是分词器 WordPiece 在分词时会去掉空格以及把一些子串（如'2022'等）当作一个整体来处理。然而，BERT 实际使用的是分词后的位置索引，因此我们需要给出分词前后之间字符位置的索引映射关系，以便把语料中给出的位置索引转化为编码后的位置索引，进而用于训练。这可以利用 BertTokenizerFast 来实现。该模块可以通过参数 offset_mapping 返回处理后每个token 是对应于分词前 passage 中的哪些字符。

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

例子：利用下面代码可以获得 passage 的索引编码以及编码前后位置索引之间的关系信息

```
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained('bert-base-chinese')
tokenizing_result = tokenizer.encode_plus(passage,
                    return_offsets_mapping=True,    #需要返回
                    add_special_tokens=False)       #不添加 CLS,SEP 等特殊符号
paragraph_ids = tokenizing_result['input_ids']
token_span = tokenizing_result['offset_mapping']
```

执行上述代码时，返回结果 tokenizing_result 是一个字典，其中tokenizing_result['input_ids']保存的是 passage 中各词的索引编码（略过了分词这个步骤），tokenizing_result['offset_mapping']保存的是编码前后位置索引之间的关系信息，其中 tokenizing_result['input_ids'] 和 tokenizing_result['offset_mapping']的长度是一样的，它们的元素是一一对应的。

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

tokenizing_result['input_ids']和 tokenizing_result['offset_mapping']的内容分别如下：

```
[10550, 2399, 8024, 2476, 676, 1343, 1346, 1217, 3683, 6612, 511]
[(0, 4), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10), (10, 11), (11, 12), (12, 13), (13, 14), (14, 15)]
```

在这个结果中，10550 是'2022'的索引编码、2399 是'年'的索引编码、6612 是'赛'的索引编码，等。同时，(0, 4)表示：编码 10550 是对应分词前 passage 中位置索引为 0 至 4-1=3 之间的字符，即对应'2022'；(5, 6)表示：编码 2399 是对应分词前 passage 中位置索引为 5 至 6-1=5之间的字符，即对应'年'，其他情况亦可类推。

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

显然，利用 tokenizing_result['offset_mapping']提供的信息，我们可以构造一个函数，使其可以计算 passage 中任一位置索引在编码后的位置索引值，代码如下：

```
def getIndInEncoding(pos, token_span, passage):
    #先构建数组 CharInd_TokenInd，其长度跟 passage 的长度一样，用于存放 passage
    #中每一字符对应编码后的位置索引
    CharInd_TokenInd = [[] for _ in range(len(passage) + 1)]
    CharInd_TokenInd[len(passage)] = [len(passage)] #
    #通过倒序遍历 token_span 来获得数组 CharInd_TokenInd 中的元素值
    for token_ind, char_sp in enumerate(token_span):
        for text_ind in range(char_sp[0], char_sp[1]):
            CharInd_TokenInd[text_ind] += [token_ind]
    for k in range(len(CharInd_TokenInd) - 2, -1, -1): #填补空格
        if CharInd_TokenInd[k] == []:
            CharInd_TokenInd[k] = CharInd_TokenInd[k + 1]
    return CharInd_TokenInd[pos]
```

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

然后执行下列代码：

```
for pos in range(len(passage)):
    ind = getIndInEncoding(pos, token_span, passage)[0]
    print(pos,'--->',ind)
```

产生如下结果：

```
0 ---> 0
1 ---> 0
2 ---> 0
3 ---> 0
4 ---> 1
5 ---> 1
6 ---> 2
7 ---> 3
8 ---> 4
9 ---> 5
10 ---> 6
11 ---> 7
12 ---> 8
13 ---> 9
14 ---> 10
```

该结果给出了 passage 中每一个字符的位置索引（0 至 14）的映射结果。'2022'中的字符在passage中的位置索引为0至3，在编码后它们都变为0（因为'2022'编码为10550，而该编码的位置索引为 0），4 和 5 都变为 1（'2022'和'年'之间有一个空格，该空格和'年'的位置索引分别为 4 和 5，而'年'被编码为 2399，该编码的索引为 1）。

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

（2）模型需要预测答案在编码后序列中的起始位置索引和终止位置索引，而预测每一个索引都是一个多分类问题，因而是一个双任务的多分类预测问题。图 8-13 也展示了该双任务的基本示意图。

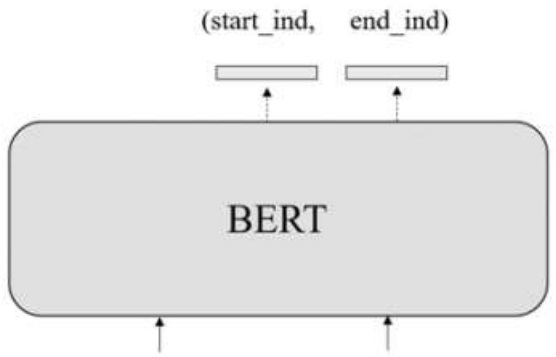


图 8-13 抽取式阅读理解的任务示意图

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

定义类 **BertForReading()**来实现双预测任务：

```
class BertForReading(nn.Module):
    def __init__(self):
        super(BertForReading, self).__init__()
        #加载预训练模型
        self.model = BertModel.from_pretrained('bert-base-chinese',
        cache_dir="./Bert_model").to(device) #加载模型
        self.qa_outputs = nn.Linear(768, 2) #有两个预测任务
    def forward(self, b_input_ids, b_attention_mask, b_token_type_ids):
        #各输入的形状都是 torch.Size([8, 512])
        outputs = self.model(input_ids=b_input_ids,
                             attention_mask=b_attention_mask,
                             token_type_ids=b_token_type_ids
                             )
        sequence_output = outputs[0]
        logits = self.qa_outputs(sequence_output) #torch.Size([8, 512, 2])
        return logits
```

注：完整代码见教材250页

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

数据读取：下面函数用于从 fn 指定的 train.json 文件中读取数据：

```
def get_query_passage_answer(fn):
    with open(fn, 'r', encoding='utf-8') as reader:
        data = json.load(reader)['data']
    examples = []
    data = data[0] #len(data)=1
    for paragraph in data['paragraphs']: #data['paragraphs']是长度为 100 的 list
        paragraph_text = paragraph['context'] #篇章的内容
        qa = paragraph['qas'][0] #paragraph['qas']是长度为 1 的 list，里面有一个字典
        query = qa['question'] #问题文本
        id = qa['id']
        #问题和篇章以及三个特殊符号的总长度不超过固定长度 MAX_LEN
        if len(query+paragraph_text)+3>MAX_LEN:
            continue
        answer = qa['answers'][0]['text'] #答案文本
        answer_start = qa['answers'][0]['answer_start'] #起始位置索引
        answer_end = answer_start+len(answer) #终止位置索引
        item = (query,paragraph_text,answer_start,answer_end)
```

注：完整代码见教材251页

8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

get_query_passage_answer()函数返回由一系列(query,paragraph_text,answer_start,answer_end)四元组组成的列表，其中每一个四元组表示一个样本。上述代码中直接弃用长度超过固定长度 MAX_LEN 的样本（BERT 只允许最大长度为512）。在由函数 get_query_passage_answer()获得以四元组表示的样本后，需要对问题文本和篇章文本进行分词和索引编码，以及对答案的起始位置索引和终止位置索引进行映射。将这些操作放在数据集类 MyDataSet()来实现。

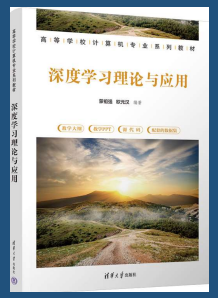
8.3 BERT 及其在 NLP 中的应用

8.3.4 基于 BERT 的阅读理解

定义数据集类：

```
class MyDataSet(Dataset):
    def __init__(self, query_passage_answer):
        super(MyDataSet, self).__init__()
        self.query_passage_answer = query_passage_answer
    def __len__(self):
        return len(self.query_passage_answer)
    def __getitem__(self, idx):
        query_passage_answer = self.query_passage_answer[idx]
        query = query_passage_answer[0] #获得问题文本
        passage = query_passage_answer[1] #获得篇章文本
        answers_start = query_passage_answer[2] #获得答案的起始位置索引
        answers_end = query_passage_answer[3] #获得答案的终止位置索引
        #answers = passage[answers_start:answers_end]
        #对篇章文本进行索引编码，同时返回编码前后位置索引之间的关系信息
        tokenizing_result = tokenizer.encode_plus(passage,
                                return_offsets_mapping=True,
                                add_special_tokens=False)
```

注：完整代码见教材252页



本章内容

contents

8.4 基于 GPT 的文本生成

8.4.1 关于 GPT

- **GPT的概念**：GPT 的全称是 Generative Pre-trained Transformer，它是 OpenAI 公司于 2018 年提出的一种生成式预训练语言模型。GPT 只采用 Transformer 框架的解码器结构，其主要功能是利用上文预测下一个单词出现的概率，是一种典型的语言模型，适合于自然语言生成类的任务 (NLG)，如摘要生成、机器翻译、诗歌创作等。而 BERT 主要适合于自然语言理解任务(NLU)，如文本分类、阅读理解、文本相似度计算等。
- **GPT的特点**：GPT 采用一种无监督学习方法，使其可以对大量无标注的文本数据进行学习，形成了规模庞大的预训练模型，并且可以为下游任务提供良好的支持。也就是说，在未知下游任务的前提下，在用足够无标注文本数据训练而得到的 GPT 也可以在具体的下游任务上有很好的表现，如阅读理解、机器翻译、知识问答和文本摘要等。即使不做微调而直接应用，在部分下游任务中它也可以获得相当不错的效果。在做微调时，GPT 一般也只是需要少量的训练数据即可达到或超过 state-of-the-art 的方法。

8.4 基于 GPT 的文本生成

8.4.2 使用 GPT2 生成英文文本——直接使用

执行上述代码构成的.py 程序，在笔者计算机上输出下列句子：

```
翻译得到的句子： 然而 作为 地域 战略 学家 无论是 从 政治 意义 还是 从 经济
意义 上 让 我 自然 想到 的 年份 是 1989 年
```

这也说明，该程序基本上能够翻译简单的英文句子，但翻译水平有待进一步提高——需要更多的数据来训练，同时需要不断完善相关参数。
同样的数据量，Transformer 程序会比循环神经网络程序快得多。这是由于 Transformer 采用了位置编码，使得待处理数据可以并行地送入 Transformer 进行计算。但是Transformer 不宜处理过长的序列（如长度超过 50 的序列），否则其训练时间会急剧增加。在处理长文本（如长度超过 200）时，LSTM 等传统循环神经网络比 Transformer 会表现更好的效果。

8.4 基于 GPT 的文本生成

8.4.2 使用 GPT2 生成英文文本——直接使用

例子：生成以"I am a student"作为开头的一段文本，可以按照下面步骤来完成：

（1）**加载预训练模型 GPT2 以及分词器**：

```
from pytorch_transformers import GPT2LMHeadModel, GPT2Tokenizer
gpt2_model = GPT2LMHeadModel.from_pretrained('gpt2')
gpt2_model.eval()
tokenizer = GPT2Tokenizer.from_pretrained('gpt2’)
```

（2）**对给定句子进行索引编码并张量化**：

```
text = "I am a student"
token_ids = tokenizer.encode(text) #索引编码
token_tensor = torch.tensor([token_ids]) #张量化，torch.Size([1, 4])
```

（3）**调用 GPT2 模型，生成下一个单词。其中，先将 token_tensor 输入 GPT2 模型**：

```
outputs = gpt2_model(token_tensor)
```

8.4 基于 GPT 的文本生成

8.4.2 使用 GPT2 生成英文文本——直接使用

形成的 outputs 是一个二元组，其中 outputs[0]的形状为(batch_size, seq_len, vocab_size) = (1, 4, 50257)。可以简单地理解为：在处理序列中每个单词时都输出一个长度为 50257 的向量。一般用最后一个单词的输出向量作为当前整个序列的特征向量，即 outputs[0][0, -1, :] 作 为当前整个序列的特征向量，其形状为(50257)。
实际上，50257 为词表的长度，因而可理解为：该向量是词表中各个单词的权重向量。于是，选择其中权重最大的单词作为当前句子的下一个单词。但这样做可能会导致模型连续输出同一个单词，无法构成有意义的句子。通常做法是，从权重排在前面的若干个（如 6 个）单词中随机选择一个单词作为下一个单词。

```
word_weight = torch.topk(outputs[0][0, -1, :], 6)[1].tolist() #选中权重最大的前 6 个单词
random.shuffle(word_weight) #随机排列单词（的索引）
next_index = word_weight[0] #选择第一个作为下一个单词（效果相当于随机选择了）
```

8.4 基于 GPT 的文本生成

8.4.2 使用 GPT2 生成英文文本——直接使用

然后，将 next_index 表示的单词“加入”到 token_tensor 当中，重复上面的操作即可生成指定长度的文本。例如，下列代码可以生成长度为 100 的英文文本：

```
for _ in range(100):
    outputs = gpt2_model(token_tensor)
    word_weight = torch.topk(outputs[0][0, -1, :], 6)[1].tolist()
    random.shuffle(word_weight)
    next_id = word_weight[0]
    token_ids = token_ids + [next_id]
    token_tensor = torch.tensor([token_ids])
generated_text = tokenizer.decode(token_tensor[0].tolist())
print('生成的文本：',generated_text)
```

8.4 基于 GPT 的文本生成

8.4.2 使用 GPT2 生成英文文本——直接使用

（4）**模块引入**：

```
import torch
from pytorch_transformers import GPT2Tokenizer
from pytorch_transformers import GPT2LMHeadModel
import random
```

最后执行由上述代码构成的.py 文件，在笔者计算机上生成下列文本：

> I am a student, and have worked with my family, but this was not what we had planned to achieve, which is what I was trying so hard for, so this would have never occurred, which is the worst of everything I know of my life, which I can say I've been through, which is not the case, and this was my first time doing that," said the mother-child duo of two at one of their most emotional rallies, which they say was attended in solidarity by their families and the

8.4 基于 GPT 的文本生成

8.4.3 使用 GPT2 生成中文文本——微调方法

【例 8.6】 利用 GPT2 构建一个中文文本生成程序。

本例利用例 8.4 中的教育类文本作为训练语料，即用下列代码从文件 train.txt 中读取类别索引为 3 的文本，在经过索引编码后保存在列表 input_ids 中。代码如下：

```
tokenizer = BertTokenizer.from_pretrained('bert-base-chinese') #利用 BERT 的分词器
path = r'./data/THUCNews'
name = r'train.txt'
fn = path + '\\' + name
train_texts, train_labels = getTexts_Labels(fn) #该函数代码见例 8.4
input_ids = []
for text,label in zip(train_texts, train_labels):
    if label != 3:
        continue
    text = text + '[SEP]'
    text_ids = tokenizer.convert_tokens_to_ids(tokenizer.tokenize(text))
    input_ids += text_ids
```

8.4 基于 GPT 的文本生成

8.4.3 使用 GPT2 生成中文文本——微调方法

结果，input_ids 为长度为 336162 的索引列表。进而用上述方法，对 input_ids 进行“等长折断”并张量化和打包。代码如下：

```
seq_len = 512 #序列的长度设置为 512（序列越长，对内存的要求越高）
#使得 input_ids 的长度为 sample_num*seq_len 并运用所有的训练文本
sample_num = len(input_ids)//seq_len
if len(input_ids)%seq_len>0:
    input_ids = input_ids[:sample_num * seq_len] + input_ids[-seq_len:]
    sample_num = sample_num + 1
else:
    input_ids = input_ids[:sample_num * seq_len]
input_ids = torch.LongTensor(input_ids)
input_ids = input_ids.reshape(-1,seq_len) #torch.Size([657, 512])
train_loader = DataLoader(input_ids,batch_size=3, shuffle=False)
print('数据集大小：',len(train_loader.dataset))
```

8.4 基于 GPT 的文本生成

8.4.3 使用 GPT2 生成中文文本——微调方法

接着，加载 GPT2 模型，采用梯度累计方法对模型进行训练。代码如下：

```
text_model = GPT2LMHeadModel.from_pretrained('gpt2').to(device)
text_model.train()
optimizer = torch.optim.Adam(text_model.parameters(), lr=1e-5)
acc_steps = 4
for ep in range(30):
    for k,b_input_ids in enumerate(train_loader):
        b_input_ids = b_input_ids.to(device)
        #输入 GPT2 模型，对其进行训练
        outputs = text_model.forward(input_ids=b_input_ids, labels=b_input_ids)
        loss, logits = outputs[:2]
        loss = loss / acc_steps
        loss.backward() #梯度累加
        if (k+1)%acc_steps ==0: #采用梯度累计方法对模型进行训练
            print(ep, loss.item())
        optimizer.step() #梯度更新
        optimizer.zero_grad() #梯度清零
torch.save(text_model,'text_model')
```

8.4 基于 GPT 的文本生成

8.4.3 使用 GPT2 生成中文文本——微调方法

梯度累计方法：主要是用于解决 GPU 显存不足的问题。在数据打包时，如果批量的大小 batch_size 设置得比较大，那么容易导致内存溢出。所以，对于长序列，batch_size 一般设置得比较小，如本例设置为 3。但是，当 batch_size 过小（尤其等于 1）时，容易造成程序收敛不稳定，甚至引起收敛震荡而难以收敛。一种解决方法就是使用梯度累加方法。该方法每计算一个批量的梯度时，不进行梯度清零，也不做参数更新，而只是做梯度累加；当累加到既定的次数以后，再做网络参数更新，并将梯度清零。假设既定的累加次数是 acc_steps，则梯度累加方法的效果几乎相当于设置批量大小为 acc_steps*batch_size 的数据打包效果，而不易于产生 GPU 显存溢出。

8.4 基于 GPT 的文本生成

8.4.3 使用 GPT2 生成中文文本——微调方法

最后，参照上一节中直接使用 GPT2 模型生成英文文本的方法，利用训练好的模型text_model 来生成中文文本。

```
with torch.no_grad():
    for _ in range(100):
        outputs = text_model(generated_tonken_tensor)
        next_token_logits = outputs[0][0, -1, :]
        next_token_logits[tokenizer.convert_tokens_to_ids('[UNK]')] = -float('Inf')
        top6 = torch.topk(next_token_logits, 6)[0] #torch.Size([6])
        top6 = top6[-1] #取最小的权值
        #将低于这 6 个权值的分量值都设置为负无穷小（-float('Inf')）
        next_token_logits[next_token_logits < top6] = -float('Inf')
        #按归一化后的权重概率选择下一个词
        next_token = torch.multinomial(torch.softmax(next_token_logits, dim=-1), num_samples=1)
        #将选中的词加入到 generated_tonken_tensor 当中，以便用于产生下一个词
        generated_tonken_tensor = torch.cat((generated_tonken_tensor, next_token.unsqueeze(0)),
dim=1)
generated_tonkens = tokenizer.convert_ids_to_tokens(generated_tonken_tensor[0].tolist())
generated_text = ''.join(generated_tonkens).replace('[SEP]','。')
```

注：完整代码见教材259页

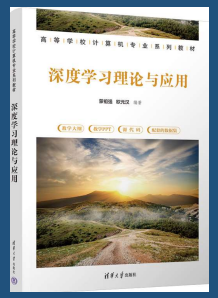
8.4 基于 GPT 的文本生成

8.4.3 使用 GPT2 生成中文文本——微调方法

执行由上述代码构成的.py 文件，在笔者计算机上产生如下的结果：

产生的中文文本：高考试成绩查询开通。2010 年北京高考录取结果查询系统开通。2010 年考研复试线公布公务员考试复习资料(二)。2011 年北京中招三本院校录计划公布。高招本科一批录取结束一本录取结果。高考状元：高分生源的五大要求。高考分

该模型似乎能够产生像样的中文文本了。但要达到实用水平，显然还需增加训练数据、优化程序代码、不断尝试调参等。



本章内容

contents

8.5 视觉 Transformer(ViT)

8.5.1 关于 ViT

ViT 的全称是 Vision Transformer，就是面向图像处理的 Transformer，它是由谷歌技术团队于 2020 年提出来的。对于尺寸为 224*224*3 的输入图像，ViT 的主要实现步骤说明如下：

- 将输入图像**划分**成 196 个 16*16*3 的**图像块**（Patch），然后将每一个图像块扁平化为向量（长度为 16*16*3=768），所有 196 个这样的向量放在一起就形成了一个输入序列 。
- **定义**一个**类别嵌入向量**（Class Embedding），其长度也为 768，该向量跟上面的向量放在一起，而且放在序列的最左边（索引值为 0），其作用相当于[CLS]向量在 BERT 中的作用，即模型收敛后该向量的值刻画了整个序列的特征，也可以简单地将[CLS]向量理解为ViT 模型的输出向量。这样，这个长度为 197 的序列便是 ViT 模型的输入序列。
- **定义** 197 个**位置嵌入向量**，跟上面的 197 个输入向量分别相加，然后送入Transformer。
- 按照输入的参数结构对 Transformer 模型进行设计，同时利用类别嵌入向量的输出作为输入，在 Transformer 模型的输出端再构建一个分类网络，进而形成一个完整的深度网络。
- 按照 Transformer 的一般训练方法对该网络进行训练，直到收敛为止。

8.5 视觉 Transformer(ViT)

8.5.2 ViT 预训练模型的使用方法

为了使用 ViT 预训练模型，通过下列步骤下载代码文件和参数文件并使用他们：

（1）**代码下载**：在网站 https://github.com/WZMIAOMIAO/deep-learning-for-image-processing/tree/master/pytorch_classification/vision_transformer 上下载代码文件。下载后，自动打包为压缩包文件 vision_transformer.zip，解压后产生 vit_model.py 等文件。

（2）**定义模型**：文件 vit_model.py 包含了 ViT 不同实现版本的代码。下面是其中的 6 种版本：

- vit_base_patch16_224()
- vit_base_patch16_224_in21k()
- vit_base_patch32_224()
- vit_base_patch32_224_in21k()
- vit_large_patch16_224()
- vit_large_patch16_224_in21k()

8.5 视觉 Transformer(ViT)

8.5.2 ViT 预训练模型的使用方法

假设欲使用函数 vit_base_patch16_224_in21k()来定义 ViT 模型，则可使用下列代码先引入该函数：

```
from vit_model import vit_base_patch16_224_in21k as create_model
```

之后就可以调用它来创建相应的 ViT 模型了：

```
model = create_model()
```

（3）**载入参数文件**：该模型中的参数是随机初始化的，因而当前并没有预测功能。为此，还需下载谷歌已经训练好的参数文件，然后装入到该模型中。在文件 vit_model.py 中，对每个函数的使用都有相应的说明，包括其参数文件的下载地址等。例如，对函数vit_base_patch16_224_in21k()给出如下的说明：

> ViT-Base model (ViT-B/16) from original paper (https://arxiv.org/abs/2010.11929).
> ImageNet-21k weights @ 224x224, source https://github.com/google-research/vision_transformer.
> weights ported from official Google JAX impl:
> https://github.com/rwightman/pytorch-image-models/releases/download/v0.1-vitjx/**jx_vit_base_patch16_224_in21k-e5005f0a.pth**

8.5 视觉 Transformer(ViT)

8.5.2 ViT 预训练模型的使用方法

相应的参数文件可下载来自链接：
https://github.com/rwightman/pytorch-image-models/releases/download/v0.1-vitjx/jx_vit_base_patch16_224_in21k-e5005f0a.pth，
下载后得到的pth文件就是函数 vit_base_patch16_224_in21k()对应的参数文件。

运用下列代码将该参数文件中的参数装入上面创建的模型 model 中：

```
weights = './weights/jx_vit_base_patch16_224_in21k-e5005f0a.pth'
weights_dict = torch.load(weights, map_location=device)
model.load_state_dict(weights_dict, strict=False)
```

8.5 视觉 Transformer(ViT)

8.5.3 基于 ViT 的图像分类

【例 8.7】 基于 ViT 预训练模型的图像分类。

- **数据集：**本例使用的图像数据集是 CIFAR-100，保存在./data/cifar100 目录下。该数据集按类别文件夹存放，即一个类别的图像文件存放在同一个子目录下。./data/cifar100一共包含 100 个子目录，每个子目录下存放 500 张图像，其中图像的尺寸为 32*32*3。也就是说，一共有 5 万张图像，分为 100 类别。
- **下载代码和参数文件：**按照上一节介绍的方法下载代码文件和参数文件，其中解压后产生的代码文件存放./vision_transformer 目录下，下载的参数文件 jx_vit_base_patch16_224_in21k-e5005f0a.pth存放在./vision_transformer/weights 目录下。

8.5 视觉 Transformer(ViT)

8.5.3 基于 ViT 的图像分类

下载好相关文件后，按下列步骤开发程序：

（1）**创建模型并加载参数**：利用函数 vit_base_patch16_224_in21k()创建 ViT 模型并装入参数：

```
model = create_model().to(device)
weights = './weights/jx_vit_base_patch16_224_in21k-e5005f0a.pth'
weights_dict = torch.load(weights, map_location=device)          #读取参数文件
model.load_state_dict(weights_dict, strict=False)                #装入参数
```

8.5 视觉 Transformer(ViT)

8.5.3 基于 ViT 的图像分类

（2）**微调模型**：通过查看函数 vit_base_patch16_224_in21k()的参数或利用 print(model)打印模型结构，我们可以发现该模型有 21843 个输出。因而，需要做微调，使其变为 100个输出。为此，定义类 ViTforCifar100，对模型 model 的输出进行微调，结果类 ViTforCifar100有 100 个输出：

```
class ViTforCifar100(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = model
        self.fc1 = nn.Linear(21843, 1024) #第 1 个全连接层
        self.fc2 = nn.Linear(1024, 100)   #第 2 个全连接层
    def forward(self,x): #torch.Size([32, 3, 224, 224])
        out = self.model(x) #torch.Size([32, 21843])
        out = torch.relu(out)
        #以下对 model 的输出做微调
        out = self.fc1(out)
        out = torch.relu(out)
        out = self.fc2(out) #输出张量的形状为#torch.Size([32, 100])
        return out
```

8.5 视觉 Transformer(ViT)

8.5.3 基于 ViT 的图像分类

（3）**读取数据并划分为训练集和测试集**：CIFAR-100 数据集的一个特点是按类别文件夹存放图像文件，即一个类别的图像文件都存放在一个目录下，而这些目录都位于./data/cifar100 目录下。这样，利用 datasets 类的ImageFolder()方法，通过指定./data/cifar100为根目录，自动读取这些图像文件并自动对图像文件进行标记（标记为0,1,…,99）。

```
dataset = datasets.ImageFolder(
    root="../data/cifar100", #指定根目录（不能设置为别的目录，否则不能正确读取数据）
    transform=transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224), #调整为 224*224*3 尺寸的图像
        transforms.ToTensor(),
        transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]),
    ]),
)
```

8.5 视觉 Transformer(ViT)

8.5.3 基于 ViT 的图像分类

读取后，图像数据都存放在对象 dataset 中。进一步，按照 7:3 将数据划分为训练集和测试集：

```
#以下代码将数据集对象划分为训练集和测试集，最后分别打包
sample_indices = list(range(len(dataset)))
random.shuffle(sample_indices) #打乱样本的索引顺序
train_len = int(0.7*len(sample_indices))   #确定训练集的长度
train_indices = sample_indices[:train_len] #训练集样本的索引
test_indices = sample_indices[train_len:]  #测试集样本的索引
#利用索引来构建数据集对象
train_set = torch.utils.data.Subset(dataset, train_indices) #Subset 类型
test_set = torch.utils.data.Subset(dataset, test_indices)
train_loader = DataLoader(dataset=train_set, batch_size=32, shuffle=True) #训练集
test_loader = DataLoader(dataset=test_set, batch_size=32, shuffle=True)   #测试集
```

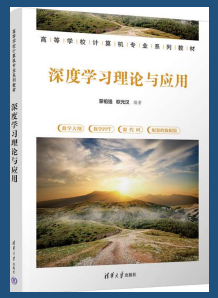
8.5 视觉 Transformer(ViT)

8.5.3 基于 ViT 的图像分类

（4）**训练模型**。先利用类 ViTforCifar100 创建模型，然后利用训练集 train_loader 对模型进行训练。

```
vit_cifar100_model = ViTforCifar100().to(device)
optimizer = optim.Adam(vit_cifar100_model.parameters())
start=time.time()
for epoch in range(20): #迭代 20 代
    for i, (imgs, labels) in enumerate(train_loader):
        imgs = imgs.to(device)
        labels = labels.to(device)
        pre_imgs = vit_cifar100_model(imgs) #torch.Size([32, 100])
        loss = nn.CrossEntropyLoss()(pre_imgs, labels) # 使用交叉熵损失函数
        print(epoch,loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
end = time.time()
torch.save(vit_cifar100_model,'vit_cifar100_model') #保存训练好的模型
print('time cost(耗时)：%0.2f 分钟'% ((end - start)/60.0))
```

（完整代码见教材P263）



本章内容

contents

8.6 ChatGPT 及其使用方法

8.6.1 关于 ChatGPT

ChatGPT：全名为 Chat Generative Pre-Trained Transformer，是由美国 OpenAI 公司开发的人工智能聊天机器人（软件）。自从 2022 年 11 月 30 日发布后，在短短两个月的时间内，ChatGPT 的月活跃用户数就高达 1 亿个之多，这使其成为史上用户数增长最快的聊天软件。相比之下，TikTok（抖音国际版）达到 1 亿用户需要 9 个月的时间，而 Instagram（照片墙）则花费了两年半的时间。

8.6 ChatGPT 及其使用方法

8.6.1 关于 ChatGPT

GPT的发展：OpenAI 因推出自然语言处理模型系列——GPT 系列而闻名于世。GPT 是一种语言模型，即用于生成语言文本。GPT 系列是从 2018 年 6 月开始推出的，相关信息如表 8.1 所示。

表 8.1 GPT 系列模型的基本信息

模型系列	发布时间	参数量	所需训练数据量
GPT-1	2018 年 6 月	1.17 亿	5GB
GPT-2	2019 年 2 月	15 亿	40G
GPT-3	2020 年 5 月	1750 亿	45TB
ChatGPT	2022 年 11 月	?	?

模型的参数量和预训练所需的数据量几乎都是呈指数增长的态势。虽然 OpenAI 公司还没有明确公开 ChatGPT 所使用的数据集（及其代码和技术），但据估计模型的参数量应不低于千亿级，训练数据量可能达到百 T 级。OpenAI 将在不久的将来继续推出 GPT-4。或许，ChatGPT 是 OpenAI 在推出 GPT-4 之前的演练，也可能用于为训练 GPT-4 而收集大量的用户对话数据。

8.6 ChatGPT 及其使用方法

8.6.1 关于 ChatGPT

ChatGPT 的特点：

- **超强的归纳能力**。ChatGPT支持连续多轮对话，它能够捕捉以前的对话内容来回答当前问题，甚至能够回答一些假设性问题，回答内容非常流畅，符合用户的意图。这体现了其非常强的上下文理解能力和对问题的归纳能力，大大增强了用户在对话互动模式下的用户体验。这也是它受到“世人敬仰”的重要原因。
- **良好的纠错能力**。ChatGPT 可以主动承认自身错误，能够接受用户指出其存在的错误，会听取用户意见并据此对答案进行优化。
- **具有一定的创造能力**。ChatGPT的创造能力主要体现在语言任务方面，比如它可以编写故事、撰写小说和诗歌等，并且它可以对其创造的作品进行不断的完善。
- **具有一定的“安全意识”**。ChatGPT 可以在考虑道德、政治、法律等因素的情况下，拒绝回答不安全的问题或生成引发安全问题的答案。例如，在试图问它如何制造抢械时，它会善意拒绝回答。

8.6 ChatGPT 及其使用方法

8.6.1 关于 ChatGPT

ChatGPT 的局限性:

- **逻辑推理能力不足**。ChatGPT 的推理能力还停留在浅层的逻辑推理，难以准确处理深层逻辑问题，尤其是对含数字的文字材料的逻辑归纳和总结往往是不准确的。另外，由于训练集等问题，ChatGPT 在 专业性领域的建模和推理以及创新创造等方面还有很大的不足。也就是说，对于非常专业的事情，ChatGPT 是难以完成的。
- **可靠性和可解释性有待提高**。在谈到 ChatGPT 缺点时，经常听用户说的比较多的话是：“正儿八经的胡说八道”，或“胡编乱造”等。一方面，这说明 ChatGPT 还会犯错误；另一方面，由于 ChatGPT 对其每一次回答未能给出有效、简便的解释，从而给用户造成极大的不信任感。因此，ChatGPT 要真正落地，其可靠性和可解释性仍然需要进一步提高。
- **知识学习的实时性尚需提升**。ChatGPT 无法在线更新知识，目前知识更新的方式还是重新训练 GPT 模型，耗费成本高，不现实。
- **稳定性需要提高**。例如，笔者也曾尝试问它：为何孙悟空和鲁智深当年都离开了贾府？我们发现，有时候它回答得非常准确、到位，但有时候还是正儿八经地胡说八道。因 此，它的回答还有很大的不确定性。

8.6 ChatGPT 及其使用方法

8.6.2 ChatGPT 的使用方法

2023 年 2 月，微软推出了新的人工智能搜索引擎 Bing（中文一般称“Bing”为“必应”）和Edge 浏览器。该搜索引擎整合了最新的 ChatGPT 技术，为用户提供基于多轮对话模式的网络搜索和内容创建服务。本小节主要介绍如何使用微软的搜索引擎 Bing，主要步骤如下：

（1）需要先安装 Edge 浏览器。为此，从微软官方网站上下载 Microsoft Edge 浏览器安装工具 MicrosoftEdgeSetup.exe

（2）打开 Edge 浏览器，在地址栏中输入网址 https://www.bing.com/new 并回车

8.6 ChatGPT 及其使用方法

8.6.2 ChatGPT 的使用方法

出现如图所示的页面要求用户先加入候补名单。为此，按提示，点击【加入候补名单】按钮，在文本框中输入 Microsoft 登录名（如果没有，则点击链接“创建一个”，然后按要求创建一个自己的 Microsoft 登录名）。

8.6 ChatGPT 及其使用方法

8.6.2 ChatGPT 的使用方法

输入 Microsoft 登录名后，接着点击【下一步】按钮，进入输入密码界面。在相应的文本框输入密码后，点击【登录】按钮，随后进入如图 所示的等待界面。

该页面表示，刚输入的 Microsoft 登录名已经进入 Bing 的候补名单，这意味着在等待 Bing 官方的审核。这个等待过程一般需要几天时间。

8.6 ChatGPT 及其使用方法

8.6.2 ChatGPT 的使用方法

通过审核后。再点击 Edge 浏览器右上角的【登录】按钮并按要求输入登录名和密码后，打开如图所示的界面。这表示，我们可以使用 Bing进行聊天了。

8.6 ChatGPT 及其使用方法

8.6.2 ChatGPT 的使用方法

在页面中点击【立即聊天】按钮，这时出现如下图所示的对话界面。

8.6 ChatGPT 及其使用方法

8.6.2 ChatGPT 的使用方法

这时，在页面底部的文本框中输入相应问题，回车后 Bing 立刻为你生成答案。

例子：在文本输入框中输入问题文本：如何才能学好深度学习？Bing 给出的回答如图所示。从图中可以看到，Bing 的回答已经非常中规中矩了。

8.7 本章小结

本章内容

- Seq2Seq结构与注意力机制
- Transformer及其在NLP中的应用
- BERT及其在NLP中的应用
- 基于GPT的文本生成
- 视觉Transformer(ViT)
- ChatGPT及其使用方法