## 10.4 使用BERT实现情感分析

在CV问题中，目前已经有了很多成熟的模型供大家使用，只需要结合特定的业务场景修改结尾的全连接层或添加softmax层即可满足需求，也就是我们常说的迁移学习。那么在NLP领域是否有这样泛化能力很强的模型呢，答案是肯定的。

2018年底，Google推出了一个打破11项NLP任务的模型BERT（Bidirectional Encoder Representation from Transformers），该模型一经问世就火遍AI领域并受到了广大开发者的青睐，可以说是NLP领域中具有里程碑意义的模型，目前BERT依旧是比赛中或者工业界首选的模型，各大公司也均基于BERT进行了更多的升级与优化。

BERT是一个预训练模型，该模型的训练阶段分为两个部分，预训练与微调，预训练阶段Google已经处理好，如果要使用该模型，只需要针对特定场景进行微调即可。在本章节中，我们会先介绍BERT的原理，再以一个实际的例子来讲解如何微调。

1. **Transformer模型**

在NLP任务中，常用的特种提取器有RNN及其变体、CNN搭配池化层、Transformer等，RNN类型的提取器有一个最大的优点能捕捉长依赖信息，但是其速度很慢，CNN搭配池化层能有效获取一些重要的特征并忽略没有意义的特征，但是却无法捕捉长依赖信息，Transformer兼具了RNN与CNN的优点，在保留长依赖信息的同时速度也很快，其中的attention机制也使其具有了类似最大池化层捕捉重要特征的能力。BERT的特征提取器实际上就是采用的Transformer的encoder层，Google提供了两个版本的BERT，其中base版本的是由12层的Transformer的encoder堆叠在一起，large版本的是由24层的Transformer的encoder堆叠在一起。

Transformer模型是基于encoder-decoder结构的，如图10.1-1所示。其中encoder层由6层图中所示的结构堆叠而成，每一个层又包括了两个子层，第一个子层是multi-head self-attention，第二个子层是一个全连接层，这两个子层均采用了residual connection来进行连接，并且还有一层layer normalization层。decoder和encoder的结构类似，也是由6层图中所示的结构堆叠而成，除了encoder提到的两层结构外，decoder层还有一层额外的masked multi-head attention。

attention是一种加权机制，针对候选值进行加权求和。Attention能表述为一个query与一个key-value集合的映射关系，其中query，keys，values都是向量，输出值是values的加权求和，这个权重是根据query与key来计算出来的。

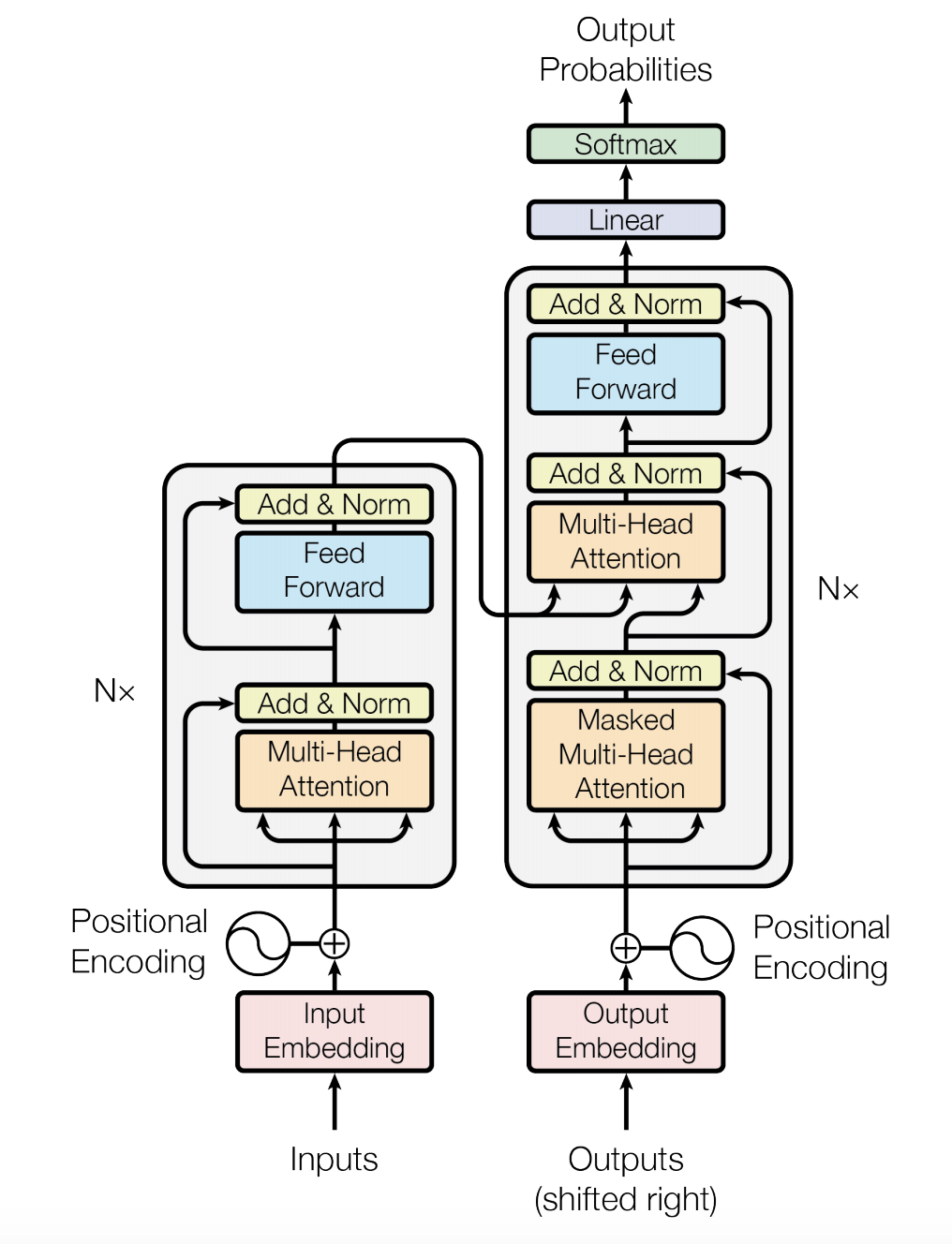


图10.1-1 Transformer结构

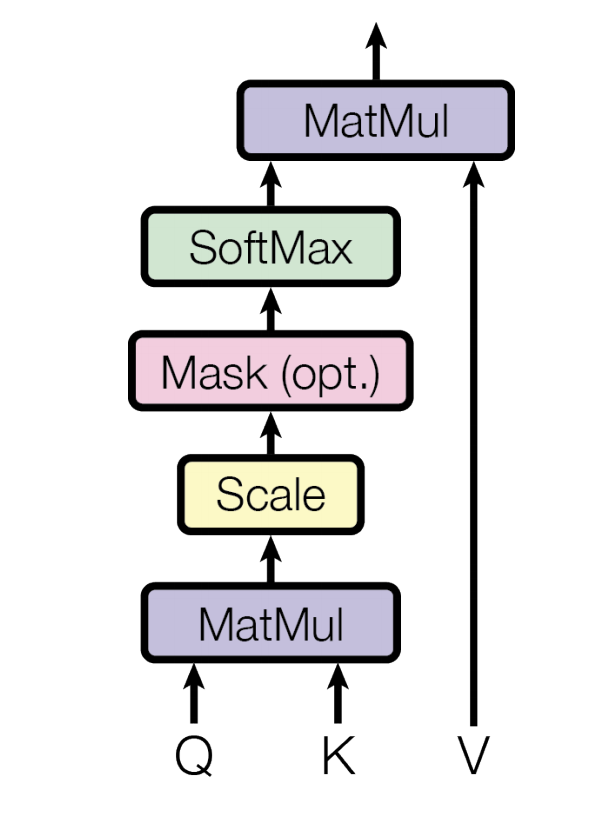


图10.1-2 Scaled Dot-Product Attention

在Transformer中也运用了attention机制，并称为Scaled Dot-Product Attention，如图10.1-2所示。其输入值是由维度为的queries、keys，维度为的values构成，首先对query与keys做一个点积的运算再除以，然后使用softmax计算出其权重并运用在values上。通常，计算attention值的时候会采用以集合的形式来降低运算次数，多个queries、keys与values组合在一起构成矩阵Q、K、V，最终计算attention值的时候即可以如下公式来表示。

attention的计算方法其实有很多种，最常用的有additive attention与dot-product attention，Transformer中提到的Scaled Dot-Product Attention其实就是dot-product attention的一个变体，添加了一个缩放因子，如果的值很小，那么这两种attention操作得到的结果差不多，但是当的值很大的时候，那么点积之后的结果某些值也会很大，softmax之后就很容易出现一个值全盘通吃的情况，而缩放因子就是为了解决这个问题而提出来的。

对于普通的Attention机制只能得到一个维度的加权求和值，transformer采用了一种叫做 Multi-Head Attention的方式，把Q,K,V做多种简单的线性变换再用Scaled Dot-Product Attention计算attention的值，最终的结果则是由多个attention的值拼接在一起并进行一个维度变换如图10.1-3所示。

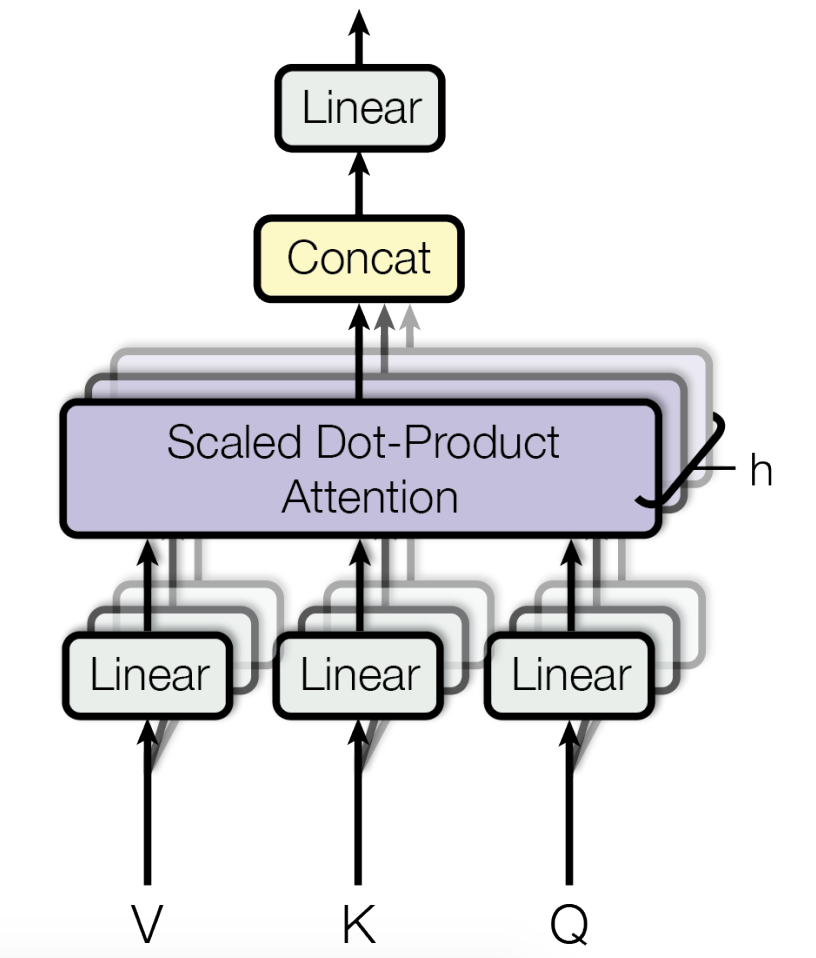


图10.1-3 Multi-Head Attention

Multi-head attention让模型能从多个不同的角度去获取文本的信息，其计算过程也可以用以下公式来表示：

其中对应的是Q,K,V的三个线性变换矩阵，的作用是用来对拼接后的结果进行降维，head的个数选择的是8。

1. **BERT预训练**

BERT的base版本是由12个transformer的encoder层堆叠在一起，没有用到decoder层，因此transformer的decoder这里就不再赘述，感兴趣的读者可自己阅读transformer的论文。接下来我们就一起来看下BERT的训练阶段是怎么做的。

BERT的预训练阶段采用了两个独有的非监督任务，一个是Masked Language Model，还有一个是Next Sentence Prediction。

Masked Language Model可以理解为完形填空，随机mask掉训练预料中15%的词，用其上下文来做预测，例如：

my dog is hairy → my dog is [MASK]

此处将hairy进行了mask处理，再采用非监督学习的方法预测mask位置的词是什么，但是该方法有一个问题，因为是mask掉15%的词，其数量已经很高了，这样就会导致某些词在微调阶段从未见过，为了解决这个问题，作者做了如下的处理。

* 80%的时间是采用[mask]，my dog is hairy → my dog is [MASK]
* 10%的时间是随机取一个词来代替mask的词，my dog is hairy -> my dog is apple
* 10%的时间保持不变，my dog is hairy -> my dog is hairy

那么为啥要以一定的概率使用随机词呢？这是因为transformer要保持对每个输入token分布式的表征，否则Transformer很可能会记住这个[MASK]就是"hairy"。至于使用随机词带来的负面影响，Google认为所有其他的token(即非"hairy"的token)共享15%\*10% = 1.5%的概率，其影响是可以忽略不计的。

BERT另一个预训练阶段的任务是Next Sentence Prediction，选一些句子对A与B，其中50%的数据B是A的下一条句子，剩余50%的数据B是语料库中随机选择的，学习其中的相关性，添加这样的预训练的目的是目前很多NLP的任务比如QA和NLI都需要理解两个句子之间的关系，从而能让预训练的模型更好的适应这样的任务。

1. **使用BERT进行文本分类**

接下来我们以一个情感分析的例子来介绍如何在keras中使用对bert进行微调。情感分析的应用场景很多，也是nlp领域一个最常见的任务，本文将会以IMDB数据集为例，该数据集是一个开源的电影评价数据集，我们的任务是根据文本信息判断评价是正面情绪还是负面情绪。

在keras中，如果需要使用BERT，只需要安装一个第三方库keras-bert，执行以下命令：pip install keras-bert

接下来需要下载Google官方的BERT预训练模型，下载好后解压到同级目录，下载地址请参考以下页面：<https://github.com/google-research/bert>

数据集的下载地址为：<http://ai.stanford.edu/~amaas/data/sentiment/>

接下来就是编码部分，首先定义一些需要用到的数据，包括BERT的位置，序列的最大长度，batch\_size的值。

|  |  |
| --- | --- |
| 1 | config\_path = 'uncased\_L-12\_H-768\_A-12/bert\_config.json' |
| 2 | checkpoint\_path = 'uncased\_L-12\_H-768\_A-12/bert\_model.ckpt' |
| 3 | vocab\_path = 'uncased\_L-12\_H-768\_A-12/vocab.txt' |
| 4 | max\_len = 32 |
| 5 | batch\_size = 64 |

定义使用正则表达式去除一些异常的HTML标签的函数

|  |  |
| --- | --- |
| 1 | def remove\_html(text): |
| 2 | r = re.compile(r'<[^>]+>') |
| 3 | return r.sub('', text) |

定义读取数据的函数，该函数能根据数据的格式做一些处理并读取到内存中

|  |  |
| --- | --- |
| 4 | def read\_file(filetype): |
| 5 | Path = './aclImdb/' |
| 6 | file\_list = [] |
| 7 | positive = path + filetype + '/pos/' |
| 8 | for f in os.listdir(positive): |
| 9 | file\_list += [positive + f] |
| 10 | negative = path + filetype + **'/neg/'** |
| 11 | **for** f **in** os.listdir(negative): |
| 12 | file\_list += [negative + f] |
| 13 | label = ([1] \* 12500 + [0] \* 12500) |
| 14 | text = [] |
| 15 | **for** f\_ **in** file\_list: |
| 16 | **with** open(f\_, encoding=**'utf8'**) **as** f: |
| 17 | text += [remove\_html(**''**.join(f.readlines()))] |
| 18 | **return** label, text |

将BERT官方提供的词典中的词添加到keras-bert的字典中，以便分词时使用。

|  |  |
| --- | --- |
| 19 | token\_dict = {} |
| 20 | with codecs.open(vocab\_path, 'r', 'utf8') as reader: |
| 21 | for line in reader: |
| 22 | token = line.strip() |
| 23 | token\_dict[token] = len(token\_dict) |
| 24 | tokenizer = Tokenizer(token\_dict) |

为了保证序列的长度一样，需要对小于最大长度的序列补0。

|  |  |
| --- | --- |
| 25 | def seq\_padding(text, padding=0): |
| 26 | max\_len = max([len(t) for t in text]) |
| 27 | return np.array([  np.concatenate([t,[padding]\*(max\_len - len(t))]) if len(t) < max\_len else t for t in text  ]) |

因为数据量比较大，这里新建一个生成器，根据batch\_size的大小来准备当前显存可用的数据。

|  |  |
| --- | --- |
| 28 | def data\_generator(data, batch\_size=64): |
| 29 | while True: |
| 30 | idx = list(range(len(data[0]))) |
| 31 | np.random.shuffle(idx) |
| 32 | X, S, Y = [], [], [] |
| 33 | for i in idx: |
| 34 | text = data[0][i][:max\_len] |
| 35 | x, s = tokenizer.encode(first=text) |
| 36 | y = data[1][i] |
| 37 | X.append(x) |
| 38 | S.append(s) |
| 39 | Y.append([y]) |
| 40 | if len(X) == batch\_size or i == idx[-1]: |
| 41 | X = seq\_padding(X) |
| 42 | S = seq\_padding(S) |
| 43 | Y = seq\_padding(Y) |
| 44 | yield [X, S], Y |
| 45 | [X, S, Y] = [], [], [] |

接下来调用keras-bert封装的方法，加载BERT模型到内存中，并把每一层设置为可训练，这样在我们微调的过程中参数才会进行更新。

|  |  |
| --- | --- |
| 46 | bert\_model=load\_trained\_model\_from\_checkpoint(config\_path, checkpoint\_path, seq\_len=None) |
| 47 | for l in bert\_model.layers: |
| 48 | l.trainable = True |

然后开始构建模型，模型的输入包括两个输入值，其中x对应的是分词在词典中对应的下标index，s表示的是BERT需要的段落向量，这里只要一个序列，因此s的值全为0。然后把x与s作为BERT的输入值，得到BERT的输出，这里我们只需要取句向量，也就是BERT中[CLS]对应的值，X[:0]表示的即是该值，最后接上一个全连接层与一个sigmoid层做二分类即可。

|  |  |
| --- | --- |
| 49 | x = Input(shape=(None,)) |
| 50 | s = Input(shape=(None,)) |
| 51 |  |
| 52 | out = bert\_model([x, s]) |
| 53 | out = Lambda(lambda x: x[:, 0])(out) |
| 54 | out = Dense(1, activation=’sigmoid’)(out) |
| 55 | model = Model([x, s], out) |

模型定义完成后，即可定义其训练所需参数，其中损失函数采用的是交叉熵损失，优化器采用的是Adam学习率为1e-4，评估指标为accuracy。然后调用上文定义好的生成器来生成与batch\_size一样长的训练数据与验证数据。最后执行fit\_generator方法即可开始训练。

|  |  |
| --- | --- |
| 56 | model.compile(  loss='binary\_crossentropy',  optimizer=Adam(1e-4),  metrics=['accuracy'] ) |
| 57 | model.summary() |
| 58 | train\_data = read\_file('train') |
| 59 | test\_data = read\_file('test') |
| 60 | train\_gen = data\_generator(train\_data, batch\_size) |
|  | test\_gen = data\_generator(test\_data, batch\_size) |
| 25 | model.fit\_generator(  train\_gen,  steps\_per\_epoch=len(train\_data[0]) // batch\_size,  epochs=5,  validation\_data=test\_gen,  validation\_steps=len(test\_data[0]) // batch\_size ) |

1. **小结**

本节主要讲解了BERT模型的原理，并采用keras微调BERT实现了情感分析。BERT作为一个目前热门的预训练模型，其效果突出，在文本特征提取阶段均可采用该模型，再根据具体的业务场景对损失函数进行修改即可实现对应的模型搭建。当然在使用keras-bert之前建议读者务必弄清楚其原理，毕竟知其然还需知其所以然。