# This Post Is All You Need（下卷）

——步步走进 BERT

# 目　录

# 修订记录

**2022 年 05 月 30 日，v1.2.0 修复数据预处理时未过滤换行符、保留分隔符等**
**2022 年 05 月 27 日，v1.1.0 修复 ignore_index 代码错误问题**
**2022 年 05 年 25 日，v1.0.0 初始版本发布**

# 第 1 节 BERT 原理与预训练任务

各位朋友大家好，欢迎来到月来客栈。前段时间掌柜在知乎上看到了这么一个提问“为什么 BERT 这么难理解”，而下面也有不少回答给出了如何学习 BERT 的建议。在掌柜看来， BERT 之所以难于理解是因为绝大多数客官真的是直接在看 BERT，而忘记了 BERT 本质上就是 Transformer 中的 Encoder 部分。同时，又由于 BERT 论文中对于这部分内容也只是一笔带过，而大多数人又根本没去看，所以最后看不懂 BERT 就变得很正常了。所以，想要弄懂 BERT 模型的前提就是一定要清楚 Transformer 模型，不然一切都是白谈。

不过好在掌柜已经对这部分内容从原理和实践上进行了详细地介绍，各位客官直接参考文章 **This Post Is All You Need（上卷）——层层剥开 Transformer** 即可。公众号后台回复“Transformer”可获取最新版本全文高清 PDF 讲解内容。

## This Post Is All You Need（上卷）

## ——层层剥开 Transformer

在本篇文章中，掌柜将会以前面介绍的 Transformer 为基础来介绍整个 BERT 网络模型的原理。当然，为了使得大家能够更加清晰的理解 BERT 模型，掌柜同样也会以原理与实践相结合的方式来进行介绍，包括 BERT 模型的实现、常见下游任务的微调以及从零实现 NSP 和 MLM 预训练任务等。下面，让我们首先进入到第一部分原理的介绍。

### 1.1 引言

经过对 Transformer 的学习，我们总算是对 Transformer 有了比较清晰的认知。不过说起 Transformer 模型，其实在它发表之初并没有引起太大的反响，直到它的后继者 BERT[1]的出现才使得大家再次回过头来仔细研究 Transformer。因此，在接下来内容中，掌柜将主要从 BERT 的基本原理、BERT 模型的实现、BERT 预训练模型在下游任务中的运用、Mask LM 和 NSP 的实现与运用这几个方面来详细介绍 Bert。总的来说就是先介绍 BERT 模型的整体原理，再介绍如何在下游任务中运用预训练的 BERT 模型，最后介绍如何利用 Mask LM 和 NSP 这两个任务来训练 BERT 模型。

关于 Bert 所取得的成就这里就不再细说，用论文里面作者的描述来说就是：BERT 不仅在概念上很简单，并且从经验上来看它也非常强大，以至于直接刷新了 11 项 NLP 记录。

*BERT is conceptually simple and empirically powerful. It obtains new state-of-the-art re- sults on eleven natural language processing tasks.*

作者之所以说 BERT 在概念上很简单，掌柜猜测这是因为 BERT 本质上是根据 Transformer 中 Encoder 堆叠而来所以说它简单；而作者说 BERT 从经验上来看非常强大，是因为 BERT 在训练过程中所基于的 Mask LM 和 NSP 这两个任务所以说其强大。

由于 BERT 是基于多层 Transformer 堆叠而来，因此在整篇论文中关于 BERT 网络结构的细节之处作者并没有提及。同时，由于论文配了一张极具迷惑性的网络结构图，使得在不看源码的基础上你几乎很难弄清整个网络结构的细节之处。

*BERT's model architec- ture is a multi-layer bidirectional Transformer en- coder based on the original implementation de- scribed in Vaswani et al. (2017) and released in the tensor2tensor library.*

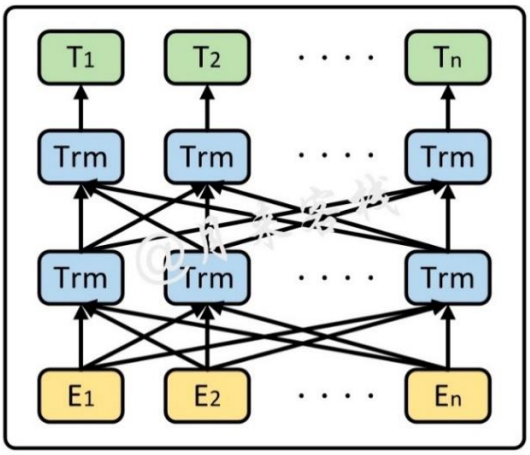



图 1-1. 原始 BERT 网络结构图

如图 1-1 所示就是论文中所展示的 BERT 网络结构图。看完论文后真的不知道作者为什么要画这么一个结构图，难道就是为了凸显“bidirectional”？一眼看去，对于同一层的 Trm 来说它到底代表什么？ 是类似于 time step 的展开，还是每个 Trm 都有着不同的权重？这些你都不清楚，当然论文也没有介绍。不过在看完这部分的源码实现后你就会发现，其实真正的 BERT 网络结构大体上应该是如图 1-2 所示的模样。

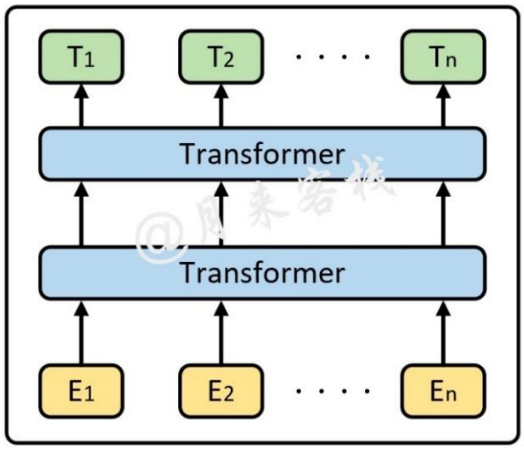



图 1-2. 原始 BERT 网络结构图

虽然从图 1-2 来看，BERT 的网络结构确实不太复杂，但是其中的 Transformer 模块相较于原始 Transformer[2]中的实现依旧有着不小的差别，不过这并不影响我们从整体上来认识 BERT。

接下来，掌柜希望通过这篇文章能够让大家对 BERT 的原理与运用有一个比较清楚的认识与理解。下面，就让我们正式走进对这篇论文的解读中。

## 1.2 动机

### 1.2.1 面临问题

在论文的介绍部分作者谈到，预训练语言模型（Language model pre-training）对于下游很多自然语言处理任务都有着显著的改善。但是作者继续说到，现有预训练模型的网络结构限制了模型本身的表达能力，其中最主要的限制就是没有采用双向编码的方法来对输入进行编码。

*Language model pre-training has been shown to be effective for improving many natural language processing tasks. We argue that current techniques restrict the power of the pre-trained representations, especially for the fine-tuning approaches. The major limitation is that standard language models are unidirectional.*

例如在 OpenAI GPT 模型中，它使用了从左到右（left-to-right）的网络架构，这就使得模型在编码过程中只能够看到当前时刻之前的信息，而不能够同时捕捉到当前时刻之后的信息。

*For example, in OpenAI GPT, the authors use a left-to- right architecture, where every token can only at- tend to previous tokens in the self-attention layers of the Transformer.*

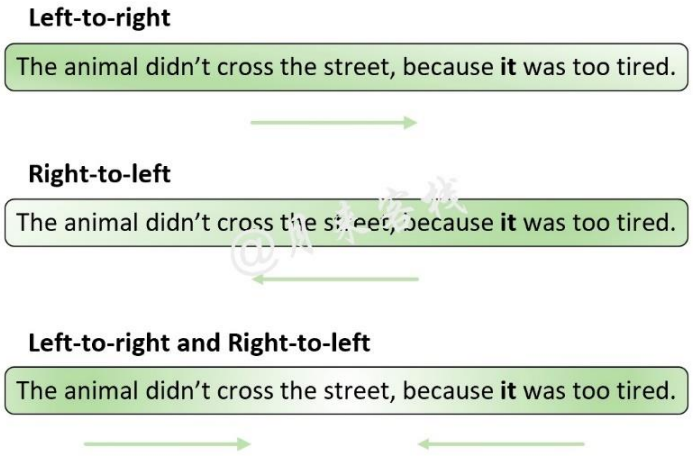



图 1-3. 不同方向编码示意图

如图 1-3 所示，对于这句样本来说，无论模型采用的是 left-to-right 的编码方式还是采用 right-to-left 的编码方式，模型在对“it”进行编码的时候都不能很好地捕捉到其具体所指代的信息。这就类似于我们人在阅读这句话时一样，在没有看到“tired”这个词之前我们同样也无法判断“it”具体所指代的事物。例如：如果我们把“tired”这个词换成“wide”，则“it”指代的就变成了“street”。所

以，如果模型采用的是双向编码的方式，那么从理论上来看就能够很好的避免这个问题，如图 1-4 所示。

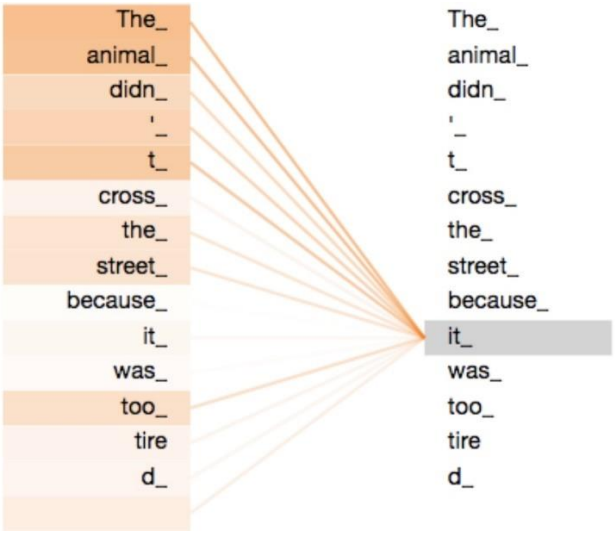



图 1-4. 注意力机制可视化图[3]

在图 1-4 中，橙色线条表示“it”应该将注意力集中在哪些位置上，颜色越深则表示注意力权重越大。通过这幅图可以发现，模型在对“it”进行编码时，将大部分注意力都集中在了“The animal”上，而这也符合我们预期的结果。

### 1.2.2 解决思路

在论文中，作者提出了采用 BERT（Bidirectional Encoder Representations from Transformers）这一网络结构来实现模型的双向编码学习能力。同时，为了使得模型能够有效的学习到双向编码的能力，BERT 在训练过程中使用了基于掩盖的语言模型（Masked Language Model, MLM），即随机对输入序列中的某些位置进行遮蔽，然后通过模型来对其进行预测。

*In this paper, we improve the fine-tuning based approaches by proposing BERT: Bidirectional Encoder Representations from Transformers. BERT alleviates the previously mentioned unidi- rectionality constraint by using a “masked lan- guage model” (MLM) pre-training objective.*

作者继续谈到，由于 MLM 预测任务能够使得模型编码得到的结果同时包含上下文的语境信息，因此有利于训练得到更深的 BERT 网络模型。除此之外，在训练BERT 的过程中作者还加入了下句预测任务（Next Sentence Prediction, NSP），即同时输入两句话到模型中，然后预测第 2 句话是不是第 1 句话的下一句话。

## 1.3 技术实现

对于技术实现这部分内容，掌柜将会分为三个大的部分来进行介绍。第一部分主要介绍 BERT 的网络结构原理以及 MLM 和 NSP 这两种任务的具体原理；第二部分将主要介绍如何实现 BERT 以及 BERT 预训练模型在下游任务中的使用；第三部分则是介绍如何利用 MLM 和 NSP 这两个任务来训练 BERT 模型（可

以是从头开始，也可以是基于开源的 BERT 预训练模型开始）。下面，掌柜将先对第一部分的内容进行介绍。

### 1.3.1 BERT 网络结构

正如图 1-2 所示，BERT 网络结构整体上就是由多层的 Transformer Encoder 堆叠所形成。关于 Transformer 部分的具体解读可以参见文章（This post is all you need——层层剥开 Transformer）[4]，这里就不再赘述。

*BERT's model architec- ture is a multi-layer bidirectional Transformer en- coder based on the original implementation de- scribed in Vaswani et al. (2017) [2].*

当然，如果图 1-2 中的结果再细化一点便能得到如图 1-5 所示的网络结构图。

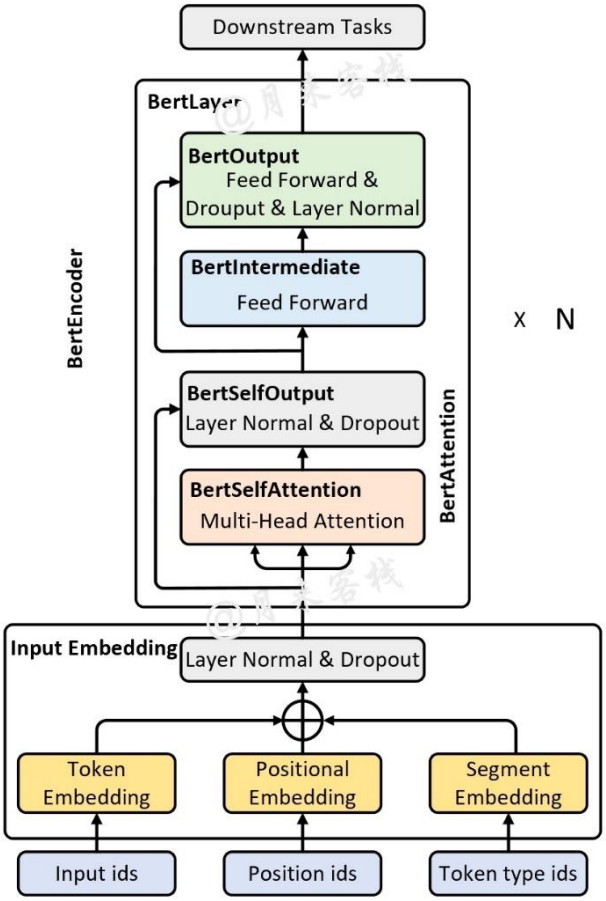



图 1-5. BERT 网络模型细节图

如图 1-5 所示便是一个详细版的 BERT 网络结构图，可以发现其上半部分的结构与之前介绍的 Transformer Encoder 差不多，只不过在 Input 部分多了一个 Segment Embedding。下面就开始分别对其中的各个部分进行介绍。

### 1.3.2 Input Embedding

如图 1-6 所示，在 BERT 中 Input Embedding 模块主要包含三个部分：Token Embedding、Positional Embedding 和 Segment Embedding。虽然前面两种

Embedding 在 Transformer 中已经介绍过，但这里需要注意的是 BERT 中的 Positional Embedding 对于每个位置的编码并不是采用公式计算得到，而是类似普通的词嵌入一样为每一个位置初始化了一个向量，然后随着网络一起训练得到。

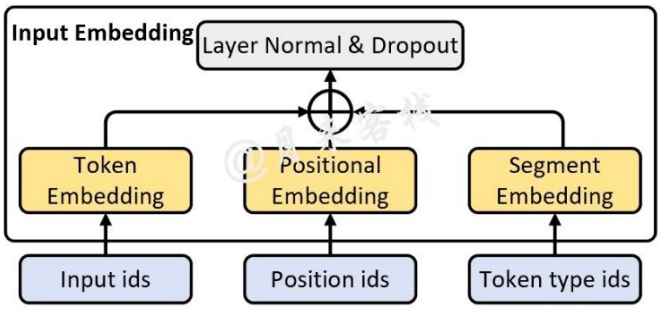



图 1-6. Input Embedding 结构图

当然，最值得注意的一点就是 **BERT 开源的预训练模型最大只支持 512 个字符的长度，这是因为其在训练过程中（位置）词表的最大长度只有 512。**

*To speed up pretraing in our experiments, we pre-train the model with sequence length of 128 for 90% of the steps. Then, we train the rest 10% of the steps of sequence of 512 to learn the positional embeddings.*

除此之外，第三部分则是 BERT 中所引入的 Segment Embedding。由于 BERT 的主要目的是构建一个通用的预训练模型，因此难免需要兼顾到各种 NLP 任务场景下的输入。因此 Segment Embedding 的作用便是用来区分输入序列中的不同部分，其本质就是通过一个普通的词嵌入来区分每一个序列所处的位置。例如在 NSP 任务中，对于任意一个句子（一共两个）来说，其中的每一位置都将用同一个向量来进行表示，以此来区分哪部分字符是第 1 句哪部分字符是第 2 句，即此时 Segment 词表的长度为 2。

最后，再将这三部分 Embedding 后的结果相加（并进行标准化）便得到了最终的 Input Embedding 部分的输出，如图 1-7 所示。

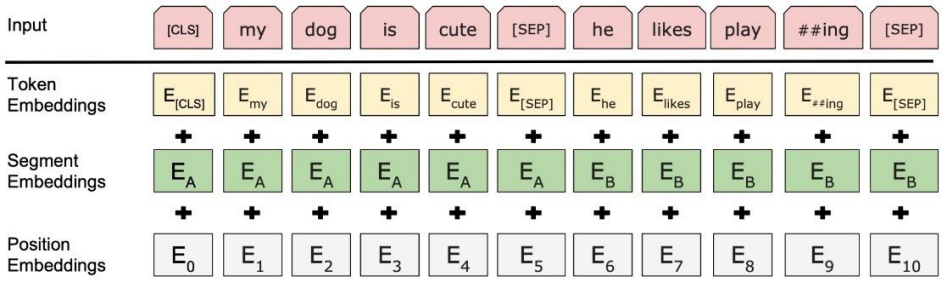



图 1-7. BERT 输入图

如图 1-7 所示，最上面的 Input 表示原始的输入序列，其中第一个字符“[CLS]”是一个特殊的分类标志，如果下游任务是做文本分类的话，那么在 BERT 的输出结果中可以只取“[CLS]”对应的向量进行分类即可（不过实验表明，取所有位置向量的均值往往有着更好的效果）；而其中的“[SEP]”字符则是用来作为将两句话分开的标志。Segment Embedding 层则同样是用来区分两句话所在的不同位置，对于每句话来说其内部各自的位置向量都是一样的，当然如果原始输入只有

一句话，那么 Segment Embedding 层中对应的每个 Token 的位置向量都相同。最后，Positional Embedding 则是用来标识句子中每个 Token 各自所在的位置，使得模型能够捕捉到文本“有序”这一特性。具体细节之处见后续代码实现。

*Sentence pairs are packed together into a single sequence. We differentiate the sentences in two ways. First, we separate them with a special token ([SEP]). Second, we add a learned embed- ding to every token indicating whether it belongs to sentence A or sentence B.*

### 1.3.3 BertEncoder

如图 1-8 所示便是 Bert Encoder 的结构示意图，其整体由多个 BertLayer（也就是论文中所指代的 Transformer blocks）所构成。

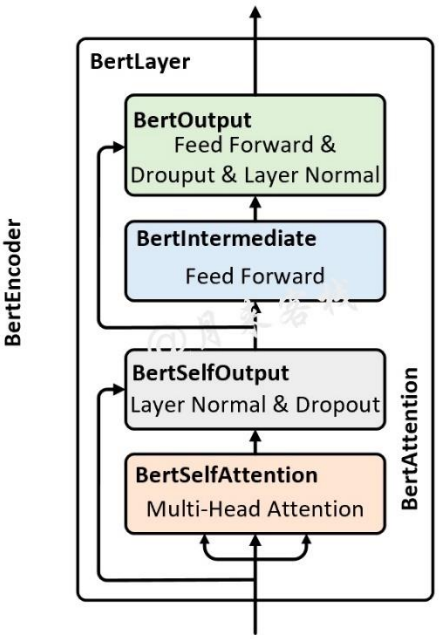



图 1-8. BertEncoder 结构图

具体的，在论文中作者分别用 $L$ 来表示 BertLayer 的层数，即 BertEncoder 是由 $L$ 个 BertLayer 所构成；用 $H$ 来表示模型的维度；用 $A$ 来表示多头注意力中多头的个数。同时，在论文中作者分别就 $\text{BERT}_{\text{BASE}}(L=12, H=768, A=12)$ 和 $\text{BERT}_{\text{LARGE}}(L=24, H=1024, A=16)$ 这两种尺寸的 BERT 模型进行了实验对比。由于这部分类似的内容在 Transformer 中已经进行了详细介绍，所以这里就不再赘述，细节之处见后续代码实现。

### 1.3.4 MLM 与 NSP 任务

为了能够更好训练 BERT 网络，论文作者在 BERT 的训练过程中引入两个任务，MLM 和 NSP。对于 MLM 任务来说，其做法是随机掩盖掉输入序列中15%的 Token（即用“[MASK]”替换掉原有的 Token），然后在 BERT 的输出结果中取对应掩盖位置上的向量进行真实值预测。

*In order to train a deep bidirectional representation, we simply mask some percentage of the input tokens at random, and then predict those masked tokens. In this case, the final hidden vectors corresponding to the mask tokens are fed into an output softmax over the vocabulary, as in a standard LM. In all of our experiments, we mask 15% of all WordPiece tokens in each sequence at random.*

接着作者提到，虽然 MLM 的这种做法能够得到一个很好的预训练模型，但是仍旧存在不足之处。由于在 fine-tuning 时，由于输入序列中并不存在“[MASK]”这样的 Token，因此这将导致 pre-training 和 fine-tuning 之间存在不匹配不一致的问题（GAP）。

*Although this allows us to obtain a bidirec- tional pre-trained model, a downside is that we are creating a mismatch between pre-training and fine-tuning, since the [MASK] token does not ap- pear during fine-tuning.*

为了解决这一问题，作者在原始 MLM 的基础了做了部分改动，即先选定15%的 Token，然后将其中的80% 替换为“[MASK]”、10% 随机替换为其它 Token、剩下的10% 不变。最后取这15% 的 Token 对应的输出做分类来预测其真实值。

*The training data generator chooses 15% of the token positions at random for prediction. If the i-th token is chosen, we replace the i-th token with (1) the [MASK] token 80% of the time (2) a random token 10% of the time (3) the unchanged i-th token 10% of the time.*

最后，作者还给出了一个不同掩盖策略下的对比结果，如图 1-9 所示。

| Masking Rates | | | Dev Set Results | | |
|---|---|---|---|---|---|
| MASK | SAME | RND | MNLI | NER | |
| | | | Fine-tune | Fine-tune | Feature-based |
| 80% | 10% | 10% | 84.2 | 95.4 | 94.9 |
| 100% | 0% | 0% | 84.3 | 94.9 | 94.0 |
| 80% | 0% | 20% | 84.1 | 95.2 | 94.6 |
| 80% | 20% | 0% | 84.4 | 95.2 | 94.7 |
| 0% | 20% | 80% | 83.7 | 94.8 | 94.6 |
| 0% | 0% | 100% | 83.6 | 94.9 | 94.6 |

图 1-9. 不同 MASK 策略对比结果图

由于很多下游任务需要依赖于分析两句话之间的关系来进行建模，例如问题回答等。为了使得模型能够具备有这样的能力，作者在论文中又提出了二分类的下句预测任务。

*Many important downstream tasks such as Ques- tion Answering (QA) and Natural Language Infer- ence (NLI) are based on understanding the relationship between two sentences. In order to train a model that understands sentence relationships, we pre-train for a binarized next sen- tence prediction task.*

具体地，对于每个样本来说都是由 A 和 B 两句话构成，其中50%的情况 B 确实为 A 的下一句话（标签为 IsNext），另外的50%的情况是 B 为语料中其它的随机句子（标签为 NotNext），然后模型来预测 B 是否为 A 的下一句话。

*Specifically, when choosing the sentences A and B for each pretraining example, 50% of the time B is the actual next sentence that follows A (labeled as IsNext), and 50% of the time it is a random sentence from the corpus (labeled as NotNext).*

如图 1-10 所示便是 ML 和 NSP 这两个任务在 BERT 预训练时的输入输出示意图，其中最上层输出的 $C$ 在预训练时用于 NSP 中的分类任务；其它位置上的 $T_i, T_j'$ 则用于预测被掩盖的 Token。

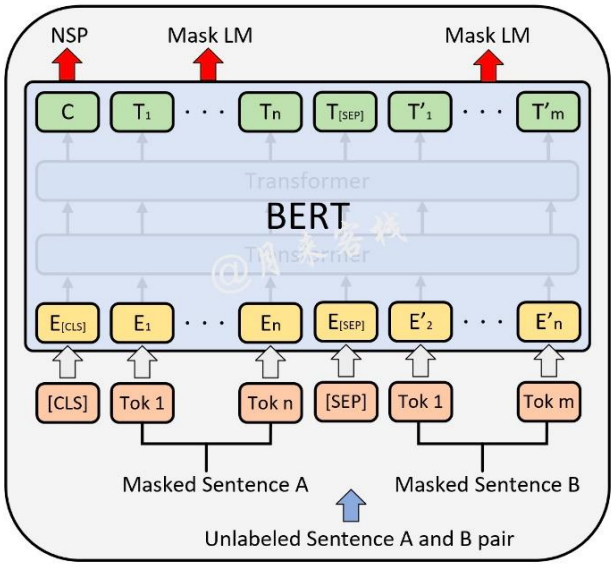



图 1-10. ML 和 NSP 任务网络结构图

到此，对于 BERT 模型的原理以及 NSP、MLM 这两个任务的内容就介绍完了。总的来说，如果单从网络结构上来看 BERT 并没有太大的创新，这也正如作者所说“BERT 整体上就是由多层的 Transformer Encoder 堆叠而来”，并且所谓的“bidirectional”其实指的也就是 Transformer 中的 self-attention 机制。同时，在掌柜看来真正让 BERT 表现出色的应该是基于 MLM 和 NSP 这两种任务的预训练过程，使得训练得到的模型具有强大的表征能力。

在下一节中，掌柜将会详细介绍如何动手来实现 BERT 模型，以及如何载入现有的模型参数并运用在下游任务中。

# 第 2 节 BERT 实现

经过第 1.3.1 节内容的介绍，相信大家对于 BERT 模型的整体结构已经有了一定的了解。根据图 1-8 可知，本质上来说 BERT 就是由多个不同的 Transformer 结构堆叠而来，同时在 Embedding 部分多加入了一个 Segment Embedding。在接下来的代码实现过程中，掌柜将会以图 1-5 中黑色加粗字体所示的部分为一个类来分别进行实现。不过在正式介绍 BERT 模型的代码实现前，掌柜先来介绍一下整个代码的工程结构，以及学习一下 BERT 模型中的参数管理。

## 2.1 工程结构

由于整个项目涉及到的代码模块较多，所以掌柜有必要在这里先进行说明，这样也便于大家在阅读后续文章时能够快速地定位到相应的代码部分。同时，掌柜也强烈建议大家在阅读的同时能够结合着代码一起阅读，并动手实践。

工程代码可从仓库：**https://github.com/moon-hotel/BertWithPretrained** 获取！

如若发现代码可能存在问题，请上 github 拉取最新版本代码或提交 issue！

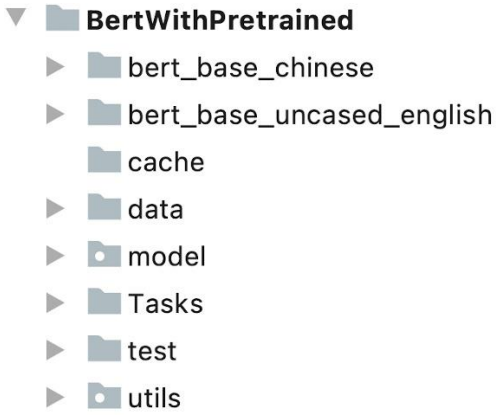



图 2-1. 代码工程结构图

如图 2-1 所示是整个工程的目录结构，包含了两个预训练模型、缓存目录、数据集目录、模型目录、下游任务目录、测试案例目录和工具模块目录。

1) 预训练模型目录

在图 2-1 中，最上面的两个文件夹分别是一个常用的中文以及英文的开源预训练模型。每个文件夹里面都包含有 3 个文件，分别是模型参数、模型超参数配置和对应的词表文件，以 bert_base_chinese 为例，如图 2-2 所示。

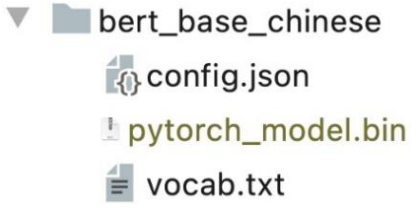



图 2-2. 预训练模型目录结构

在图 2-2 中，config.json 是 BERT 模型的默认配置内容；pytorch_model.bin 是 huggingface[8]开源的 BERT 中文预训练模型；vocab.txt 是词表。一般来说大多数情况下我们都不需要去修改这些配置，唯一可能改动的就是 config.json 中的部分参数。

2) 缓存目录

在图 2-1 中，cache 目录是用来存放训练过程中所保存下来的模型。

3) 数据集目录

在图 2-1 中，data 目录是用来存放各个下游任务的相关数据集，目前来说一共有 6 个数据集，如图 2-3 所示。

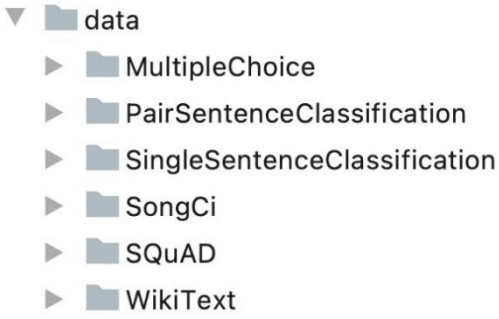



图 2-3. 数据集目录结构

如图 2-3 所示，MultpleChoice、PairSentenceClassification、SingleSentenceClassification 和 SQuAD 这 4 个目录分别对应的是 4 个下游任务的数据集；SongCi 和 WikiText 是后面用于 NSP 和 MLM 两个任务的数据集，后面实际使用时掌柜会仔细进行介绍。

4) 模型目录

在图 2-1 中，model 目录中存放的是整个 BERT 模型的实现代码，以及下游任务的相关实现，如图 2-4 所示。

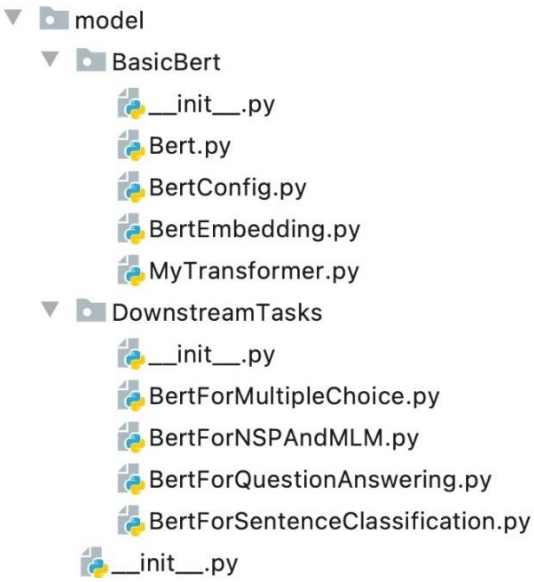



图 2-4. 模型目录结构

如图 2-4 所示，BasicBert 目录中是 BERT 模型各个部分的实现，其中 MyTransformer.py 是整个 Transformer 结构的实现，这里我们需要引用到其中的 MultiheadAttention 模块；BertEmbedding.py 是 BERT 模型中 InputEmbedding 模块的实现；BertConfig.py 是 BERT 模型的配置导入实现；Bert.py 是整个 BERT 模型的实现。

在图 2-4 中，DownstreamTasks 目录中是 BERT 模型各个下游任务的实现，包括针对单文本和多文本分类的 BertForSentenceClassification.py 模块；针对多问题选择的 BertForMultipleChoice.py 模块；针对 SQuAD 问题回答的 BretForQuestionAnswering.py 模块；以及针对 NSP 和 MLM 预训练过程的 BertForNSPAndMLM.py 模块。

5) 下游任务目录

在图 2-1 中，Tasks 目录中存放的是 model 中各个任务的模型训练代码，如图 2-5 所示。

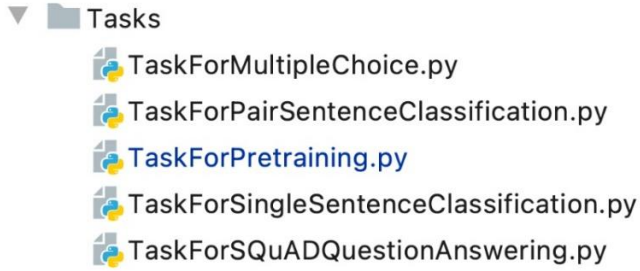



图 2-5. 下游任务目录结构

在图 2-5 中， Tasks 目录是 BERT 模型各个任务训练部分的实现，包括针对单文本分类的 TaskForSingleSentenceClassification.py 模块；针对多文本分类的 TaskForPairSentenceClassification.py 模块；针对多问题选择的 TaskForMultipleChoice.py 模块；针对 SQuAD 问题回答的 TaskForSQuADQuestionAnswering.py 模块；以及针对 NSP 和 MLM 预训练过程的 TaskForPretraining.py 训练模块。

6) 测试案例目录

在图 2-1 中，test 目录中存放的是各个模块测试案例，用于在现实过程中的验证，后续内容中的相关使用示例都能在其中找到。

7) 工具模块目录

在图 2-1 中，utils 目录中存放的是一些辅助工具模块，如图 2-6 所示。

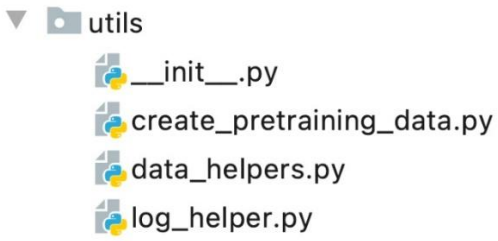



图 2-6. 工具模块目录结构

如图 2-6 所示，log_helper.py 是日志输出内容的相关定义；data_helpers.py 是所有下游任务的数据集构造实现；create_pretraining_data.py 是 NSP 和 MLM 任务的数据集构造实现。

## 2.2 参数管理

在模型的实现过程中，我们通常都会定义一个 Config 类并用类的成员变量来保存模型的各个参数值。当然，如果你需要从本地文件中读取参数（例如 bert 的 config.json 文件），那么只需要将读取后的结果赋值到 Config 类的各个成员变量即可。通过类对象来管理参数最大的一个好处就是容易扩展新的参数。因为使用类对象来管理参数时传递给模型的都是一个实例化对象，这样在新增加参数时就不需要在相关接口处再定义新的形参。此时只需要在 Config 类中增加一个成员变量，然后在模型的相关地方以访问类成员的方式来使用参数。

### 2.2.1 初始化类成员

如下是定义的一个 BertConfig 类，里面的成员变量就是模型中的各个超参数：

```
class BertConfig(object):
    def __init__(self, vocab_size=21128,
                 hidden_size=768,
                 num_hidden_layers=12,
                 num_attention_heads=12,
                 intermediate_size=3072,
                 pad_token_id=0,
                 hidden_act="gelu",
                 hidden_dropout_prob=0.1,
                 attention_probs_dropout_prob=0.1,
                 max_position_embeddings=512,
                 type_vocab_size=2,
                 initializer_range=0.02):

        self.vocab_size = vocab_size
        self.hidden_size = hidden_size
        self.num_hidden_layers = num_hidden_layers
        self.num_attention_heads = num_attention_heads
        self.hidden_act = hidden_act
        self.intermediate_size = intermediate_size
        self.pad_token_id = pad_token_id
        self.hidden_dropout_prob = hidden_dropout_prob
        self.attention_probs_dropout_prob = attention_probs_dropout_prob
        self.type_vocab_size = type_vocab_size
        self.max_position_embeddings = max_position_embeddings
        self.initializer_range = initializer_range
```

在上述代码中，第 16-27 行便是 BERT 模型默认的所有超参数变量，同时在类的初始化方法中也给了相应的默认值。

### 2.2.2 导入本地参数

进一步，我们还需要实现一个方法来从本地文件中导入 config.json 这个默认配置文件，并重新赋值给 BertConfig 类的各个成员变量中，实现代码如下：

```
1     @classmethod
2     def from_dict(cls, json_object):
3         """Constructs a BertConfig from a Python dictionary of parameters."""
4         config = BertConfig(vocab_size=None)
5         for (key, value) in six.iteritems(json_object):
6             config.__dict__[key] = value
7         return config
8 
9     @classmethod
10    def from_json_file(cls, json_file):
11        """Constructs a BertConfigfrom a json file of parameters."""
12        """从 json 配置文件读取配置信息"""
13        with open(json_file, 'r') as reader:
14            text = reader.read()
15        logging.info(f"成功导入 BERT 配置文件 {json_file}")
16        return cls.from_dict(json.loads(text))
```

在上述代码中，第 10-15 行是载入本地配置文件得到一个 json 对象；第 16 行是通过 from_dict 方法来将 json 对象的每个参数值赋值到 BertConfig 类对象中。这里值得一提的是，在 Python 中@classmethod 修饰器的作用是在不实例化类对象之前也能够使用对应的类方法，即方法中的第 1 个形参 cls 便是为实例化的类对象。

### 2.2.3 使用示例

在实现完上述代码后，便可以通过如下方式来进行使用：

```
1 if __name__ == '__main__':
2     json_file = '../bert_base_chinese/config.json'
3     config = BertConfig.from_json_file(json_file)
4     print(config.hidden_size)
5     print(config.vocab_size)
6 
7 # 768
8 # 21128
```

在上述代码中，第 2 行是配置文件所在的路径；第 3 行是根据 config.json 中的参数来实例化 BertConfig 类对象并返回该对象；第 4-5 行是以访问类成员的方式来使用模型的相关参数。

## 2.3 Input Embedding 实现

首先，我们先来看 Input Embedding 的实现过程。为了复用之前在介绍 Transformer 实现时所用到的这部分代码，我们直接在这基础上再加一个 Segment Embedding 即可。

### 2.3.1 Token Embedding

Token Embedding，也叫做词嵌入，算是 NLP 中将文本表示为稠密向量的一个基本操作，其原理掌柜就不再赘述，具体实现如下：

```
1  class TokenEmbedding(nn.Module):
2      def __init__(self, vocab_size, hidden_size, pad_token_id=0,
                    initializer_range=0.02):
3          super(TokenEmbedding, self).__init__()
4          self.embedding = nn.Embedding(vocab_size, hidden_size,
                                         padding_idx=pad_token_id)
5          self._reset_parameters(initializer_range)
6
7      def forward(self, input_ids):
8          """
9          :param input_ids: shape : [input_ids_len, batch_size]
10         :return: shape: [input_ids_len, batch_size, hidden_size]
11         """
12         return self.embedding(input_ids)
13
14     def _reset_parameters(self, initializer_range):
15         for p in self.parameters():
16             if p.dim() > 1:
17                 normal_(p, mean=0.0, std=initializer_range)
```

在上述代码中，第 4 行中的 padding_idx 是用来指定序列中用于 padding 处理的索引编号，一般来说默认都是 0。在指定 padding_idx 后，如果输入序列中有 0，那么对应位置的向量就会全是 0。当然，这一步掌柜认为不做也可以，因为在计算自主力权重的时候会通过 padding_mask 向量来去掉这部分内容，具体可参见文章[6]第 3 节中的内容。第 5 行是用给定的方式来初始化参数，当然这几乎不会用到。因为不管是在下游任务中微调，还是继续通过 NSL 和 MLM 这两个任务来训练模型参数，我们大多数情况下都会再开源的 BERT 模型参数上进行，而不是从头再来。第 7-12 行是 Token Embedding 的前向传播过程；第 14-17 行是上面提到的参数初始化方法。

### 2.3.2 Positional Embedding

对于 Positional Embedding 来说，其作用便是用来解决自注意力机制不能捕捉到文本序列内部各个位置之间顺序的问题。关于这部分内容原理的介绍，可以参见文章[6]第 2 节中的内容。不同于 Transformer 中 Positional Embedding 的实现方式，在 BERT 中 Positional Embedding 并没有采用固定的变换公式来计算每个位置上的值，而是采用了类似普通 Embedding 的方式来为每个位置生成一个向量，然后随着模型一起训练。因此，这一操作就限制了在使用预训练的中文 BERT 模型时，最大的序列长度只能是 512，因为在训练时只初始化了 512 个位置向量。具体地，其实现代码如下：

```
1  class PositionalEmbedding(nn.Module):
2      """
3      # Since the position embedding table is a learned variable, we create it
4      # using a (long) sequence length max_position_embeddings. The actual
5      # sequence length might be shorter than this, for faster training of
6      # tasks that do not have long sequences.
7
8      https://github.com/google-research/bert/blob/master/modeling.py
9      """
10
11     def __init__(self, hidden_size, max_position_embeddings=512,
                    initializer_range=0.02):
12         super(PositionalEmbedding, self).__init__()
13         # 因为 BERT 预训练模型的长度为 512
14         self.embedding = nn.Embedding(max_position_embeddings, hidden_size)
15         self._reset_parameters(initializer_range)
16
17     def forward(self, position_ids):
18         """
19         :param position_ids: [1,position_ids_len]
20         :return: [position_ids_len, 1, hidden_size]
21         """
22         return self.embedding(position_ids).transpose(0, 1)
```

从上述代码可以看出，Positional Embedding 本质上也就是一个普通的 Embedding 层，只是在这一场景下作者赋予了它特俗的含义，即序列中的每一个位置有自己独属的向量表示。同时，在默认配置中，第 14 行中的 max_position_embeddings 值为 512，也就是只支持最大 512 个 token 的输入。

### 2.3.3 Segment Embedding

Segment Embedding 的原理及目的掌柜在文章[6]中已经详细介绍过，总结起来就是为了满足下游任务中存在需要两句话同时输入到模型中的场景，即可以看

成是对输入的两个序列分别赋予一个位置向量用以区分各自所在的位置。这一点可以和上面的 Positional Embedding 进行类比。具体地，其实现代码如下：

```
class SegmentEmbedding(nn.Module):
    def __init__(self, type_vocab_size, hidden_size,
                 initializer_range=0.02):
        super(SegmentEmbedding, self).__init__()
        self.embedding = nn.Embedding(type_vocab_size, hidden_size)
        self._reset_parameters(initializer_range)

    def forward(self, token_type_ids):
        """
        :param token_type_ids:  shape: [token_type_ids_len, batch_size]
        :return: shape: [token_type_ids_len, batch_size, hidden_size]
        """
        return self.embedding(token_type_ids)
```

在上述代码中，type_vocab_size 的默认值为 2，即只用于区分两个序列的不同位置。

### 2.3.4 Bert Embeddings

在完成 Token、Positional、Segment Embedding 这 3 个部分的代码之后，只需要将每个部分的结果相加即可得到最终的 Input Embedding 作为模型的输入，如图 2-1 所示。

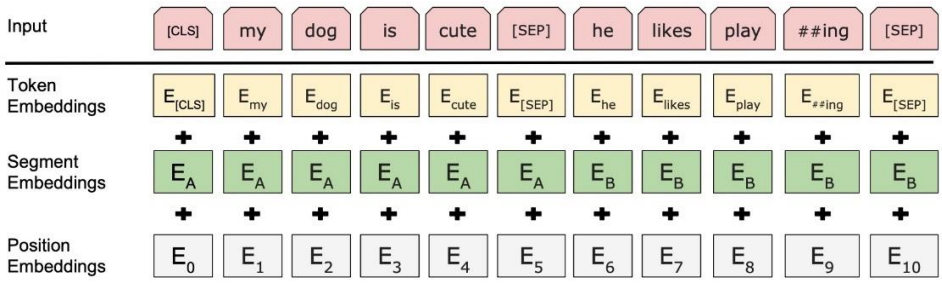



图 2-1. BERT 输入图

具体地，其代码实现为：

```
class BertEmbeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.word_embeddings = TokenEmbedding(config.vocab_size,
                                              config.hidden_size,
                                              config.pad_token_id,
                                              config.initializer_range)
        # return shape [src_len,batch_size,hidden_size]
        self.position_embeddings =
                            PositionalEmbedding(config.max_position_embeddings,
```

```
10                                              config.hidden_size,
11                                              config.initializer_range)
12          # return shape [src_len,1,hidden_size]
13          self.token_type_embeddings=SegmentEmbedding(config.type_vocab_size,
14                                                      config.hidden_size,
15                                                     config.initializer_range)
16          # return shape  [src_len,batch_size,hidden_size]
17          self.LayerNorm = nn.LayerNorm(config.hidden_size)
18          self.dropout = nn.Dropout(config.hidden_dropout_prob)
19          self.register_buffer("position_ids",
                  torch.arange(config.max_position_embeddings).expand((1, -1)))
20          # shape: [1, max_position_embeddings]
```

在上述代码中，config 是传入的一个配置类，里面各个类成员就是 BERT 中对应的模型参数。第 4、9、13 行便是用来分别定义图 2-1 中的 3 部分 Embedding。第 19 行代码是用来生成一个默认的位置 id，即[0,1,....,512]，在后续可以通过 self.position_ids 来进行调用。

进一步，其前向传播过程代码为：

```
1     def forward(self,
2                 input_ids=None,
3                 position_ids=None,
4                 token_type_ids=None):
5
6         src_len = input_ids.size(0)
7         token_embedding = self.word_embeddings(input_ids)
8         # shape:[src_len,batch_size,hidden_size]
9
10        if position_ids is None:
11            position_ids = self.position_ids[:, :src_len]  # [1,src_len]
12        positional_embedding = self.position_embeddings(position_ids)
13        # [src_len, 1, hidden_size]
14
15        if token_type_ids is None:
16            token_type_ids = torch.zeros_like(input_ids,
17                    device=self.position_ids.device)  # [src_len, batch_size]
18        segment_embedding = self.token_type_embeddings(token_type_ids)
19        # [src_len,batch_size,hidden_size]
20
21        embeddings = token_embedding+positional_embedding+segment_embedding
22        embeddings = self.LayerNorm(embeddings)
23        embeddings = self.dropout(embeddings)
24        return embeddings # [src_len,batch_size,hidden_size]
```

在上述代码中，第 7 行 input_ids 表示输入序列的原始 token id，即根据词表映射后的索引，其形状为[src_len, batch_size]；第 10 行 position_ids 是位置序列，本质就是 [0,1,2,3,...,src_len-1]，其形状为[1,src_len]，在实际建模时这个参数其实可以不用传值，因为当其为空时会自动从 self.position_ids 截取一段；第 15 行 token_type_ids 用于不同序列之间的分割，例如[0,0,0,0,1,1,1,1]用于区分前后不同的两个句子，形状为[src_len,batch_size]，如果输入模型的只有一个序列，那么这个参数也不用传值。第 21-23 行代码则是用来将 3 部分 Embeeding 的结果相加。

### 2.3.5 使用示例

在完成上述 Embedding 部分的代码实现后，便可以通过如下方式来进行使用：

```
from model.BasicBert.BertEmbedding import BertEmbeddings
from model.BasicBert.BertConfig import BertConfig
import torch

if __name__ == '__main__':
    json_file = '../bert_base_chinese/config.json'
    config = BertConfig.from_json_file(json_file)
    src = torch.tensor([[1,3,5, 7, 9], [2, 4, 6, 8, 10]], dtype=torch.long)
    src = src.transpose(0, 1)  # [src_len, batch_size]
    token_type_ids = torch.LongTensor([[0, 0, 0, 1, 1], [0, 0, 1, 1,
                                                         1]]).transpose(0, 1)
    bert_embedding = BertEmbeddings(config)
    bert_embedding_result=bert_embedding(src,token_type_ids=token_type_ids)
    print(" --------- 测试 BertEmbedding ------------")
    print("bert embedding shape [src_len, batch_size, hidden_size]: ",
                                                    bert_embedding_result.shape)
```

上述代码运行结束后将会输出如下结果：

```
 --------- 测试 BertEmbedding ------------
bert embedding shape [src_len,batch_size,hidden_size]: torch.Size([5,2,768])
```

## 2.4 BertModel 实现

在实现完 Input Embedding 部分的代码后，下面就可以着手来实现构成 BERT 模型的第 2 个重要组成部分 BertEncoder 了。如图 2-2 所示，整个 BertEncoder 由多个 BertLayer 堆叠形成；而 BertLayer 又是由 BertOutput、BertIntermediate 和 BertAttention 这 3 个部分组成；同时 BertAttention 是由 BertSelfAttention 和 BertSelfOutput 所构成。

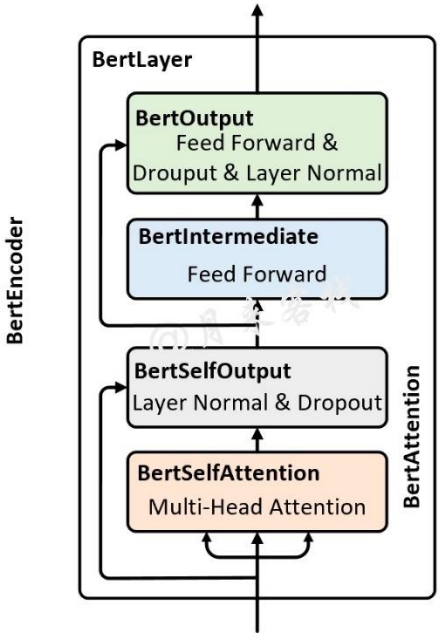



图 2-2. BertEncoder 结构图

接下来，掌柜就以图 2-2 中从下到上的顺序来依次对每个部分进行实现。

### 2.4.1 BertAttention 实现

对于 BertAttention 来说，需要明白的是其核心就是在 Transformer 中所提出来的 self-attention 机制，也就是图 2-2 中的 BertSelfAttention 模块；其次再是一个残差连接和标准化操作。对于 BertSelfAttention 的实现，其代码如下：

```
1 class BertSelfAttention(nn.Module):
2     def __init__(self, config):
3         super(BertSelfAttention, self).__init__()
4         if 'use_torch_multi_head' in config.__dict__ and
                                                config.use_torch_multi_head:
5             MultiHeadAttention = nn.MultiheadAttention
6         else:
7             MultiHeadAttention = MyMultiheadAttention
8         self.multi_head_attention = MultiHeadAttention(config.hidden_size,
9                                                 config.num_attention_heads,
10                                      config.attention_probs_dropout_prob)
11
12     def forward(self,query,key,value, attn_mask=None,key_padding_mask=None):
13         return self.multi_head_attention(query, key, value,
14                                          attn_mask=attn_mask,
15                                          key_padding_mask=key_padding_mask)
```

在上述代码中，第 4-10 行是实例化一个多头注意力机制对象，并且这里掌柜提供了两种多头实现：一种是之前 Transformer 中我们自己的实现，另一种则是 PyTorch 中的实现；可以通过设置参数 use_torch_multi_head = True 来使用 PyTorch

中的实现。第 12-15 行则是多头注意力的前向传播过程，其返回包含两个部分：多头注意力的线性组合以及多头注意力权重的均值。

如上便是 BertSelfAttention 的实现代码，其对应的就是 GoogleResearch[7]代码中的 attention_layer 方法。正如前面所说，BertSelfAttention 本质上就是 Transformer 模型中的 self-attention 模块，具体原理可参见文章[6]，这里掌柜就不再赘述。

对于 BertSelfOutput 的实现，其主要就是层 Dropout、标准化和残差连接 3 个操作，代码如下：

```
class BertSelfOutput(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=1e-12)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, hidden_states, input_tensor):
        """
        :param hidden_states: [src_len, batch_size, hidden_size]
        :param input_tensor: [src_len, batch_size, hidden_size]
        :return: [src_len, batch_size, hidden_size]
        """
        hidden_states = self.dropout(hidden_states)
        hidden_states = self.LayerNorm(hidden_states + input_tensor)
        return hidden_states
```

上述代码便是 BertSelfOutput 的实现，其过程也十分简单，掌柜就不再赘述。

接下来就是对 BertAttention 部分的实现，其由 BertSelfAttention 和 BertSelfOutput 这两个类构成，代码如下：

```
class BertAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.self = BertSelfAttention(config)
        self.output = BertSelfOutput(config)

    def forward(self,
                hidden_states,
                attention_mask=None):
        """
        :param hidden_states: [src_len, batch_size, hidden_size]
        :param attention_mask: [batch_size, src_len]
        :return: [src_len, batch_size, hidden_size]
        """
        self_outputs = self.self(hidden_states,
```

```
                                    hidden_states,
                                    hidden_states,
                                    attn_mask=None,
                                    key_padding_mask=attention_mask)
        # self_outputs[0] shape: [src_len, batch_size, hidden_size]
        attention_output = self.output(self_outputs[0], hidden_states)
        return attention_output
```

在上述代码中，第 8 行 hidden_states 是 Input Embedding 处理后的结果；第 9 行的 attention_mask 就是同一个 batch 中不同长度序列的 padding 信息，具体可以参见文章[6]第 3 节的内容；第 15 行是自注意力机制的输出结果；第 21 行便是执行 BertSelfOutput 中的 3 个操作。

### 2.4.2 BertLayer 实现

根据图 2-2 可知，BertLayer 里面还有 BertOutput 和 BertIntermediate 这两个模块，因此下面先来实现这两个部分。

对于 BertIntermediate 来说也就是一个普通的全连接层，因此实现起来也非常简单，代码如下：

```
class BertIntermediate(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.hidden_size, config.intermediate_size)
        if isinstance(config.hidden_act, str):
            self.intermediate_act_fn = get_activation(config.hidden_act)
        else:
            self.intermediate_act_fn = config.hidden_act

    def forward(self, hidden_states):
        """
        :param hidden_states: [src_len, batch_size, hidden_size]
        :return: [src_len, batch_size, intermediate_size]
        """
        hidden_states = self.dense(hidden_states)
         # [src_len, batch_size, intermediate_size]
        if self.intermediate_act_fn is None:
            hidden_states = hidden_states
        else:
            hidden_states = self.intermediate_act_fn(hidden_states)
        return hidden_states
```

在上述代码中，第 6 行用来根据指定参数获取激活函数，其代码实现如下所示：

```
def get_activation(activation_string):
    act = activation_string.lower()
    if act == "linear":
        return None
    elif act == "relu":
        return nn.ReLU()
    elif act == "gelu":
        return nn.GELU()
    elif act == "tanh":
        return nn.Tanh()
    else:
        raise ValueError("Unsupported activation: %s" % act)
```

进一步，对于 BertOutput 来说，其包含有其包含有一个全连接层和残差连接，实现代码如下：

```
class BertOutput(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.intermediate_size, config.hidden_size)
        self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=1e-12)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, hidden_states, input_tensor):
        """
        :param hidden_states: [src_len, batch_size, intermediate_size]
        :param input_tensor: [src_len, batch_size, hidden_size]
        :return: [src_len, batch_size, hidden_size]
        """
        hidden_states = self.dense(hidden_states)
        # [src_len, batch_size, hidden_size]
        hidden_states = self.dropout(hidden_states)
        hidden_states = self.LayerNorm(hidden_states + input_tensor)
        return hidden_states
```

在上述代码中，第 8 行里 hidden_states 指的就是 BertIntermediate 模块的输出，而 input_tensor 则是 BertAttention 部分的输出。

在实现完这两个部分的代码后，便可以通过 BertAttention、BertIntermediate 和 BertOutput 这 3 个部分来实现组合的 BertLayer 部分，代码如下：

```
class BertLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.bert_attention = BertAttention(config)
        self.bert_intermediate = BertIntermediate(config)
```

```
        self.bert_output = BertOutput(config)

    def forward(self,
                hidden_states,
                attention_mask=None):
        """
        :param hidden_states: [src_len, batch_size, hidden_size]
        :param attention_mask: [batch_size, src_len] mask 掉 padding 部分的内容
        :return: [src_len, batch_size, hidden_size]
        """
        attention_output=self.bert_attention(hidden_states, attention_mask)
        # [src_len, batch_size, hidden_size]
        intermediate_output = self.bert_intermediate(attention_output)
        # [src_len, batch_size, intermediate_size]
        layer_output=self.bert_output(intermediate_output, attention_output)
        # [src_len, batch_size, hidden_size]
        return layer_output
```

从上述代码中可以发现，对于 BertLayer 的实现来说其整体逻辑也并不太复杂，就是根据 BertAttention、BertOutput 和 BertIntermediate 这三部分构造而来；同时每个部分输出后的维度掌柜也都进行了标注以便大家进行理解。

到此，对于 BertLayer 部分的实现就介绍完了，下面继续来看如何对 BertEncoder 进行实现。

### 2.4.3 BertEncoder 实现

根据图 2-2 所示可知，BERT 主要由 Input Embedding 和 BertEncoder 这两部分构成；而 BertEncoder 是有多个 BertLayer 堆叠所形成，因此需要先实现 BertEncoder，代码如下：

```
class BertEncoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.bert_layers = nn.ModuleList([BertLayer(config) for _ in
                                  range(config.num_hidden_layers)])

    def forward(
            self,
            hidden_states,
            attention_mask=None):
        """
        :param hidden_states: [src_len, batch_size, hidden_size]
        :param attention_mask: [batch_size, src_len]
```

```
15          :return:
16          """
17          all_encoder_layers = []
18          layer_output = hidden_states
19          for i, layer_module in enumerate(self.bert_layers):
20              layer_output = layer_module(layer_output, attention_mask)
21              #  [src_len, batch_size, hidden_size]
22              all_encoder_layers.append(layer_output)
23          return all_encoder_layers
```

在上述代码中，第 5-6 行便是用来定义多个 BertLayer；第 18-22 行用来循环计算多层 BertLayer 堆叠后的输出结果。最后，只需要按需将 BertEncoder 部分的输出结果输入到下游任务即可。

进一步，在将 BertEncoder 部分的输出结果输入到下游任务前，需要将其进行略微的处理，代码如下：

```
1 class BertPooler(nn.Module):
2     def __init__(self, config):
3         super().__init__()
4         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
5         self.activation = nn.Tanh()
6         self.config = config
7 
8     def forward(self, hidden_states):
9         """
10        :param hidden_states:  [src_len, batch_size, hidden_size]
11        :return: [batch_size, hidden_size]
12        """
13        if self.config.pooler_type == "first_token_transform":
14            token_tensor = hidden_states[0, :].reshape(-1,
                                                       self.config.hidden_size)
15        elif self.config.pooler_type == "all_token_average":
16            token_tensor = torch.mean(hidden_states, dim=0)
17        pooled_output = self.dense(token_tensor) #[batch_size, hidden_size]
18        pooled_output = self.activation(pooled_output)
19        return pooled_output  # [batch_size, hidden_size]
```

在上述代码中，第 13-14 行代码用来取 BertEncoder 输出的第一个位置（[cls]位置），例如在进行文本分类时可以取该位置上的结果进行下一步的分类处理；第 15-16 行是掌柜自己加入的一个选项，表示取所有位置的平均值，当然我们也可以根据自己的需要在添加下面添加其它的方式；最后，17-19 行就是一个普通的全连接层。

### 2.4.4 BertModel 实现

到此，对于 BERT 模型中的各个基础模块就已经实现完毕了。如图 2-3 所示，只需要再将各个部分的代码组合到一起便完成了对于 BERT 模型的实现。

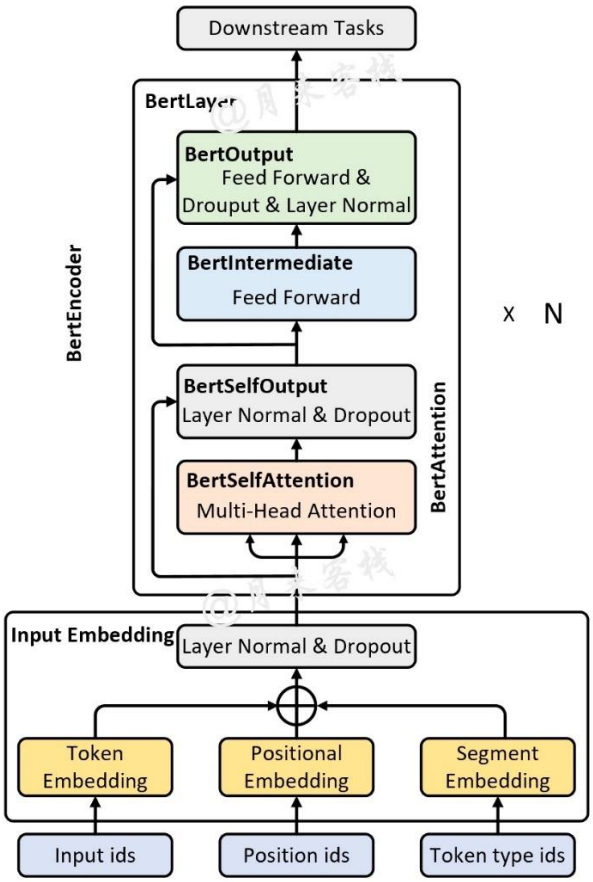



图 2-3. BERT 网络模型细节图

基于上述所有实现便可以搭建完成整个 BERT 的主体结构，代码如下：

```
class BertModel(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.bert_embeddings = BertEmbeddings(config)
        self.bert_encoder = BertEncoder(config)
        self.bert_pooler = BertPooler(config)
        self.config = config
        self._reset_parameters()

    def forward(self,
                input_ids=None,
                attention_mask=None,
                token_type_ids=None,
                position_ids=None):
```

```
15          """
16          :param input_ids:  [src_len, batch_size]
17          :param attention_mask: [batch_size, src_len] mask 掉 padding 部分的内容
18          :param token_type_ids: [src_len, batch_size]
                    如果输入模型的只有一个序列，那么这个参数也不用传值
19          :param position_ids: [1,src_len] 在实际建模时这个参数其实可以不用传值
20          :return:
21          """
22          embedding_output = self.bert_embeddings(input_ids=input_ids,
23                                                  position_ids=position_ids,
24                                                 token_type_ids=token_type_ids)
25          all_encoder_outputs = self.bert_encoder(embedding_output,
26                                                 attention_mask=attention_mask)
27          sequence_output = all_encoder_outputs[-1]
28          pooled_output = self.bert_pooler(sequence_output)
29           return pooled_output, all_encoder_outputs
```

如上代码所示便是整个 BERT 部分的实现，可以发现在厘清了整个思路后这部分代码理解起来就相对容易了。第 22-24 行便是 Input Embedding 后的输出结果，其形状为[src_len, batch_size, hidden_size]；第 25-26 行是整个 BERT 编码部分的输出，其中 all_encoder_outputs 为一个包含有 num_hidden_layers 个层的输出；第 27 行是处理得到整个 BERT 网络的输出，这里取了最后一层的输出，形状为[src_len, batch_size, hidden_size]；第 28 行默认是最后一层的第 1 个 token 即[cls]位置经 dense + tanh 后的结果，其形状为[batch_size, hidden_size]。

到此，对于整个 BERT 主体部分的代码实现就介绍完了。需要提醒各位客官的是，在阅读本文的过程中最好是结合着每个部分的输出结果（包括形状和意义）来进行理解，以便能更加清晰的认识 BERT 模型。

### 2.4.5 使用示例

在完成上述整个BERT模型的代码实现后，便可以通过如下方式来进行使用：

```
1   json_file = '../bert_base_chinese/config.json'
2   config = BertConfig.from_json_file(json_file)
3   config.__dict__['use_torch_multi_head'] = True
4   src = torch.tensor([[1, 3, 5, 7, 9, 2, 3],
                        [2, 4, 6, 8, 10, 0, 0]], dtype=torch.long)
5   src = src.transpose(0, 1)  # [src_len, batch_size]
6   print(f"input shape [src_len,batch_size]: ", src.shape)
7   token_type_ids = torch.LongTensor([[0, 0, 0, 1, 1, 1, 1],
                                       [0, 0, 1, 1, 1, 0, 0]]).transpose(0, 1)
8   attention_mask = torch.tensor([[True,True,True,True,True,True,True],
9                                  [True,True,True,True,True,False,False]])
10  position_ids = torch.arange(src.size()[0]).expand((1, -1)) # [1,src_len]
```

```
    bert_model = BertModel(config)
    bert_model_output = bert_model(input_ids=src,
                                   attention_mask=attention_mask,
                                   token_type_ids=token_type_ids,
                                   position_ids=position_ids)[0]
    print(f"BertModel's pooler output shape [batch_size, hidden_size]: ",
                                                    bert_model_output.shape)
```

上述代码运行结束后便会得到类似如下所示的结果：

```
- INFO: 成功导入 BERT 配置文件 ../bert_base_chinese/config.json
input shape [src_len,batch_size]:  torch.Size([7, 2])
BertModel's pooler output shape [batch_size,hidden_size]: [2, 768])
  =======  BertMolde 参数: ========
bert_embeddings.position_ids    torch.Size([1, 518])
bert_embeddings.word_embeddings.embedding.weight torch.Size([21128, 768])
bert_embeddings.position_embeddings.embedding.weight torch.Size([518, 768])
bert_embeddings.token_type_embeddings.embedding.weight torch.Size([2, 768])
bert_embeddings.LayerNorm.weight      torch.Size([768])
bert_embeddings.LayerNorm.bias    torch.Size([768])
……
bert_encoder.bert_layers.0.bert_intermediate.dense.weight
torch.Size([3072, 768])
bert_encoder.bert_layers.0.bert_intermediate.dense.bias  torch.Size([3072])
bert_encoder.bert_layers.0.bert_output.dense.weight  torch.Size([768, 3072])
bert_encoder.bert_layers.0.bert_output.dense.bias     torch.Size([768])
bert_encoder.bert_layers.0.bert_output.LayerNorm.weight    torch.Size([768])
bert_encoder.bert_layers.0.bert_output.LayerNorm.bias     torch.Size([768])
……
bert_encoder.bert_layers.1.bert_intermediate.dense.bias    torch.Size([3072])
bert_encoder.bert_layers.1.bert_output.dense.weight  torch.Size([768, 3072])
bert_encoder.bert_layers.1.bert_output.dense.bias     torch.Size([768])
bert_encoder.bert_layers.1.bert_output.LayerNorm.weight     torch.Size([768])
bert_encoder.bert_layers.1.bert_output.LayerNorm.bias     torch.Size([768])
……
```

# 第 3 节 模型的保存与迁移

在正式介绍 BERT 模型的下游微调任务之前，掌柜先来带着各位客官了解一下 PyTorch 框架中模型保持与迁移的使用方法，以便后续更好的理解 BERT 下游任务中预训练模型的加载过程。

## 3.1 运用场景

通常，对于模型的保存与加载会出现在以下 3 个场景中：

1) 模型推理过程；一个网络模型在完成训练后通常都需要对新样本进行推理预测，此时只需要构建模型的前向传播过程，然后载入已训练好的参数初始化网络即可。
2) 模型再训练过程；模型在一批数据上训练完成之后需要将其保存到本地，并且可能过了一段时间后又收集到了一批新的数据，因此这个时候就需要将之前的模型载入进行在新数据上进行增量训练（或者是在整个数据上进行全量训练）。
3) 模型迁移学习；这个时候就是将别人已经训练好的预模型拿过来，作为你自己网络模型参数的一部分进行初始化。例如：你在 BERT 模型的基础上加了几个全连接层来做分类任务，那么你就需要将原始 BERT 模型中的参数载入并以此来初始化你的网络中的 BERT 部分的权重参数。

接下来，掌柜就以上述 3 个场景为例来介绍如何利用 PyTorch 框架来完成上述过程。

## 3.2 查看网络参数

所谓查看网络参数指的是在网络模型定义好之后，我们将整个网络中参数信息打印出来的过程。之所以要介绍这部分内容是因为在后续载入 BERT 与训练模型的时候，我们需要将本地预训练模型的参数名与网络中的参数名一一对应起来才能够完成参数的初始化工作。

由于 BERT 模型参数较多，为了简洁下面掌柜将以之前介绍的 LeNet5 网络模型为例来分别进行介绍。

### 3.2.1 查看参数

首先需要定义好 LeNet5 的网络模型结构，如下代码所示：

```
1 class LeNet5(nn.Module):
2     def __init__(self, ):
3         super(LeNet5, self).__init__()
4         self.conv = nn.Sequential(
5             nn.Conv2d(1, 6, 5, padding=2),
```

```
            nn.ReLU(),  # [n,6,24,24]
            nn.MaxPool2d(2, 2),
            nn.Conv2d(6, 16, 5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2))
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(16  5  5, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, 10))
    def forward(self, img):
        output = self.conv(img)
        output = self.fc(output)
        return output
```

在定义好 LeNet5 网络结构之后，只要我们完成了这个类的实例化操作，那么网络中对应的权重参数也就完成了相应的初始化工作，即有了一个初始值。同时，我们可以通过如下方式来查看：

```
# Print model's state_dict
print("Model's state_dict:")
for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())
```

其输出的结果为：

```
conv.0.weight    torch.Size([6, 1, 5, 5])
conv.0.bias      torch.Size([6])
conv.3.weight    torch.Size([16, 6, 5, 5])
....
....
```

可以发现，网络模型中的参数 model.state_dict()其实是以字典的形式（实质上是 collections 模块中的 OrderedDict）在进行保存。当然，我们也可以直接输出网络中各个参数的名称：

```
print(model.state_dict().keys())
# odict_keys(['conv.0.weight', 'conv.0.bias',
# 'conv.3.weight', 'conv.3.bias', 'fc.1.weight',
# 'fc.1.bias', 'fc.3.weight', 'fc.3.bias', 'fc.5.weight', 'fc.5.bias'])
```

这样，我们便得到了网络中每个参数的相关信息，包括名称和维度。在后续迁移学习时就可以根据参数名称或形状来匹配相应有用的参数。

### 3.2.2 自定义参数前缀

同时，这里值得注意的地方有两点：①参数名中的 fc 和 conv 前缀是根据你在上面定义 nn.Sequential()时的变量名所确定的；②参数名中的数字表示每个 Sequential()中网络层所在的位置。例如将网络结构定义成如下形式：

```
class LeNet5(nn.Module):
    def __init__(self, ):
        super(LeNet5, self).__init__()
        self.moon = nn.Sequential(
            nn.Conv2d(1, 6, 5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(6, 16, 5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Flatten(),
            nn.Linear(16  5  5, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, 10))
```

那么其参数名输出结果为：

```
print(model.state_dict().keys())
odict_keys(['moon.0.weight', 'moon.0.bias', 'moon.3.weight',
    'moon.3.bias', 'moon.7.weight', 'moon.7.bias', 'moon.9.weight',
    'moon.9.bias', 'moon.11.weight', 'moon.11.bias'])
```

理解了这一点对于后续我们去解析和载入一些预训练模型很有帮助。在介绍完模型参数的查看方法后，就可以进入到模型复用阶段的内容介绍了。

## 3.3 模型推理过程

模型推理一般涉及到两个过程，一是训练结束后模型的保存；二是推理时模型的加载。下面掌柜逐一进行介绍。

### 3.3.1 模型保存

在 PyTorch 中，对于模型的保存来说比较简单，通常来说通过如下两行代码便可以实现[11]：

```
model_save_path = os.path.join(model_save_dir, 'model.pt')
torch.save(model.state_dict(), model_save_path)
```

在指定保存的模型名称时 PyTorch 官方建议的后缀为.pt 或者.pth（当然也不是强制，例如.bin 也有后缀）。最后，只需要在合适的地方加入第 2 行代码即可完成模型的保存。同时，如果想要在训练过程中保存某个条件下的最优模型，那么应该通过下面方法一中的形式：

```
1 # 方法一
2 best_model_state = deepcopy(model.state_dict())
3 torch.save(best_model_state, model_save_path)
4 # 方法二
5 best_model_state = model.state_dict()
6 torch.save(best_model_state, model_save_path)
```

因为方法二中 best_model_state 得到只是 model.state_dict() 的引用，它依旧会随着训练过程而发生改变。

### 3.3.2 模型复用

在推断过程中，首先需要完成网络的初始化，然后再载入已有的模型参数来覆盖网络中的权重参数即可，示例代码如下：

```
1 def inference(data_iter, device, model_save_dir='./MODEL'):
2     model = LeNet5()  #
3     model.to(device)
4     model_save_path = os.path.join(model_save_dir, 'model.pt')
5     if os.path.exists(model_save_path):
6         loaded_paras = torch.load(model_save_path)
7         model.load_state_dict(loaded_paras)
8         model.eval() # 注意不要忘记
9     with torch.no_grad():
10        acc_sum, n = 0.0, 0
11        for x, y in data_iter:
12            x, y = x.to(device), y.to(device)
13            logits = model(x)
14            acc_sum += (logits.argmax(1) == y).float().sum().item()
15            n += len(y)
16        print("Accuracy in test data is :", acc_sum / n)
```

在上述代码中，第 2 行是实例化一个 LeNet5 网络模型，同时初始化现有模型的权重参数；第 4-8 行是载入本地已有模型，并用其来重新初始化网络中的权重参数。这样，便可以进行后续的推断工作：

```
1 Accuracy in test data is : 0.8851
```

## 3.4 模型再训练过程

在介绍完模型的保存与复用之后，对于网络的追加训练就很简单了。最简便的一种方式就是在训练过程中只保存网络权重，然后在后续进行追加训练时只载入网络权重参数初始化网络进行训练即可，示例如下（完整代码参见[12]）：

```
def train(self):
        #......
        model_save_path = os.path.join(self.model_save_dir, 'model.pt')
        if os.path.exists(model_save_path):
            loaded_paras = torch.load(model_save_path)
            self.model.load_state_dict(loaded_paras)
            print("#### 成功载入已有模型，进行追加训练...")
        optimizer = torch.optim.Adam(self.model.parameters(),
                                     lr=self.learning_rate)
       #......
        for epoch in range(self.epochs):
            for i, (x, y) in enumerate(train_iter):
                x, y = x.to(device), y.to(device)
                logits = self.model(x)
                # ......
            print("Epochs[{}/{}]--acc {:.4}".format(epoch,self.epochs,
                            self.evaluate(test_iter, self.model, device)))
            torch.save(self.model.state_dict(), model_save_path)
```

这样，便完成了模型的追加训练。如下为相关输出结果：

```
#### 成功载入已有模型，进行追加训练...
Epochs[0/5]---batch[938/0]---acc 0.9062---loss 0.2926
Epochs[0/5]---batch[938/100]---acc 0.9375---loss 0.1598
......
```

除此之外，也可以在保存参数的时候，将优化器参数、损失值等一同保存下来，然后在恢复模型的时候连同其它参数一起恢复，示例如下：

```
model_save_path = os.path.join(model_save_dir, 'model.pt')
torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,
            ...
            }, model_save_path)
```

恢复模型时的载入方式如下：

```
checkpoint = torch.load(model_save_path)
model.load_state_dict(checkpoint['model_state_dict'])
```

```
3 optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
4 epoch = checkpoint['epoch']
5 loss = checkpoint['loss']
```

## 3.5 模型迁移学习

到目前为止，对于前面两种应用场景的介绍就算完成了，可以发现总体上并不复杂。但是对于第 3 种场景的应用来说就会略微复杂一点，而这也是后面我们载入 BERT 与训练模型的关键。

### 3.5.1 定义新模型

假设现在有一个 LeNet6 网络模型，它是在 LeNet5 的基础最后多加了一个全连接层，其定义如下：

```
1 class LeNet6(nn.Module):
2     def __init__(self, ):
3         super(LeNet6, self).__init__()
4         self.conv = nn.Sequential(
5             nn.Conv2d(1, 6, 5, padding=2),
6             nn.ReLU(),
7             nn.MaxPool2d(2, 2),
8             nn.Conv2d(6, 16, 5),
9             nn.ReLU(),
10            nn.MaxPool2d(2, 2))
11        self.fc = nn.Sequential(
12            nn.Flatten(),
13            nn.Linear(16  5  5, 120),
14            nn.ReLU(),
15            nn.Linear(120, 84),
16            nn.ReLU(),
17            nn.Linear(84, 64),
18            nn.ReLU(),
19            nn.Linear(64, 10) )
```

接下来，我们需要将在 LeNet5 上训练得到的权重参数迁移到 LeNet6 网络中去。从上面 LeNet6 的定义可以发现，此时尽管只是多加了一个全连接层，但是倒数第 2 层参数的维度也发生了变换。因此，对于 LeNet6 来说只能复用 LeNet5 网络前面 4 层的权重参数。

### 3.5.2 读取可用参数

所谓读取模型参数指的是将本地模型载入后输出其中相关参数信息的过程。例如对于开源的 BERT 中文预训练模型 bert_base_chinese[10]来说，我们拿到的将是一个名为 pytorch_model.bin 的文件。因此首先我们需要读取这个模型并输出

相关的参数信息，然后再根据相应的规则解析并将其重新初始化我们自己的网络模型中。

在这里，首先我们可以将 LeNet5 模型载入，然后再来查看相关参数的信息：

```
model_save_path = os.path.join('./MODEL', 'model.pt')
loaded_paras = torch.load(model_save_path)
for param_tensor in loaded_paras:
    print(param_tensor, "\t", loaded_paras[param_tensor].size())

#---- 可复用部分
conv.0.weight     torch.Size([6, 1, 5, 5])
conv.0.bias       torch.Size([6])
conv.3.weight     torch.Size([16, 6, 5, 5])
conv.3.bias       torch.Size([16])
fc.1.weight       torch.Size([120, 400])
fc.1.bias     torch.Size([120])
fc.3.weight       torch.Size([84, 120])
fc.3.bias     torch.Size([84])
#----- 不可复用部分
fc.5.weight       torch.Size([10, 84])
fc.5.bias     torch.Size([10])
```

同时，对于 LeNet6 网络的参数信息为：

```
model = LeNet6()
for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())
#
conv.0.weight     torch.Size([6, 1, 5, 5])
conv.0.bias       torch.Size([6])
conv.3.weight     torch.Size([16, 6, 5, 5])
conv.3.bias       torch.Size([16])
fc.1.weight       torch.Size([120, 400])
fc.1.bias     torch.Size([120])
fc.3.weight       torch.Size([84, 120])
fc.3.bias     torch.Size([84])
#------ 新加入部分
fc.5.weight       torch.Size([64, 84])
fc.5.bias     torch.Size([64])
fc.7.weight       torch.Size([10, 64])
fc.7.bias     torch.Size([10])
```

在理清楚了新旧模型的参数后，下面就可以将 LeNet5 中我们需要的参数给取出来，然后再换到 LeNet6 的网络中。

### 3.5.3 模型迁移学习

虽然本地载入的模型参数（上面的 loaded_paras）和模型初始化后的参数（上面的 model.state_dict()）都是一个字典的形式，但是我们并不能够直接改变 model.state_dict()中的权重参数。这里需要先构造一个 state_dict 然后通过 model.load_state_dict()方法来重新初始化网络中的参数。

同时，在这个过程中我们还需要筛选掉本地模型中不可复用的部分，具体代码如下：

```
1 def para_state_dict(model, model_save_dir):
2     state_dict = deepcopy(model.state_dict())
3     model_save_path = os.path.join(model_save_dir, 'model.pt')
4     if os.path.exists(model_save_path):
5         loaded_paras = torch.load(model_save_path)
6         for key in state_dict:  # 在新的网络模型中遍历对应参数
7             if key in loaded_paras and state_dict[key].size() ==
                                                loaded_paras[key].size():
8                 print("成功初始化参数:", key)
9                 state_dict[key] = loaded_paras[key]
10    return state_dict
```

在上述代码中，第 2 行的作用是先拷贝网络中（LeNet6）原有的参数；第 6-9 行则是用本地的模型参数（LeNet5）中可以复用的部分替换掉 LeNet6 中的对应部分，其中第 7 行就是判断可用的条件。同时需要注意的是在不同的情况下筛选的方式可能不一样，因此具体情况需要具体分析，但是整体逻辑是一样的。

最后，我们只需要在模型训练之前调用该函数，然后重新初始化 LeNet6 中的部分权重参数即可[12]：

```
1 state_dict = para_state_dict(self.model, self.model_save_dir)
2 self.model.load_state_dict(state_dict)
```

此时，训练时的相关信息输出如下：

```
1 成功初始化参数: conv.0.weight
2 成功初始化参数: conv.0.bias
3 成功初始化参数: conv.3.weight
4 成功初始化参数: conv.3.bias
5 成功初始化参数: fc.1.weight
6 成功初始化参数: fc.1.bias
7 成功初始化参数: fc.3.weight
8 成功初始化参数: fc.3.bias
9 #### 成功载入已有模型，进行追加训练...
10 Epochs[0/5]---batch[938/0]---acc 0.1094---loss 2.512
11 Epochs[0/5]---batch[938/100]---acc 0.9375---loss 0.2141
12 Epochs[0/5]---batch[938/200]---acc 0.9219---loss 0.2729
```

```
13 Epochs[0/5]---batch[938/300]---acc 0.8906---loss 0.2958
14 ......
15 Epochs[0/5]---batch[938/900]---acc 0.8906---loss 0.2828
16 Epochs[0/5]--acc on test 0.8808
```

可以发现，在大约 100 个 batch 之后，模型的准确率就提升上来了。

# 第 4 节 基于 BERT 预训练模型的文本分类任务

经过前面第 1、2 节内容的介绍，相信各位客官对于 BERT 的原理以及实现过程已经有了比较清晰的理解。同时，我们都知道 BERT 是一个强大的预训练模型，它可以基于谷歌发布的预训练参数在各个下游任务中进行微调。因此，在本节内容中，掌柜将会介绍第一个下游微调场景，即如何在文本分类场景中基于 BERT 预训练模型进行微调。

## 4.1 任务构造原理

总的来说，基于 BERT 的文本分类（准确的是单文本，也就是输入只包含一个句子）模型就是在原始的 BERT 模型后再加上一个分类层即可，类似的结构掌柜在文章[6]（基于 Transformer 的分类模型）中也介绍过，大家可以去看一下。同时，对于分类层的输入（也就是原始 BERT 的输出），默认情况下取 BERT 输出结果中[CLS]位置对于的向量即可，当然也可以修改为其它方式，例如所有位置向量的均值等（见 2.4.3 节内容，将配置文件 config.json 中的 pooler_type 字段设置为"all_token_average"即可）。

因此，对于基于 BERT 的文本分类模型来说其输入就是 BERT 的输入，输出则是每个类别对应的 logits 值。接下来，掌柜首先就来介绍如何构造文本分类的数据集。

## 4.2 数据预处理

数据预处理完整代码见仓库 BertWithPretrained/utils/data_helper.py 文件。

### 4.2.1 输入介绍

在构建数据集之前，我们首先需要知道的是模型到底应该接收什么样的输入，然后才能构建出正确的数据形式。在上面我们说到，基于 BERT 的文本分类模型的输入就等价于 BERT 模型的输入，同时根据第 2 节内容的介绍可以知道 BERT 模型的输入如图 4-1 所示。

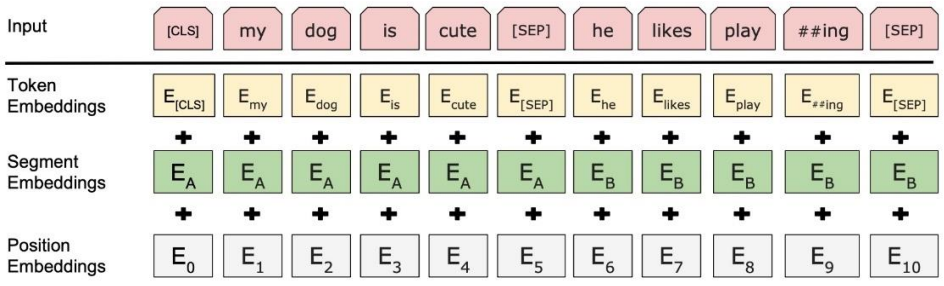



图 4-1. BERT 输入图

由于对于文本分类这个场景来说其输入只有一个序列，所以在构建数据集的时候并不需要构造 Segment Embedding 的输入，直接默认使用全为 0 即可（见文

章[6] 第 2.2.4 节内容）；同时，对于 Position Embedding 来说在任何场景下都不需要对其指定输入，因为我们在代码实现时已经做了相应默认时的处理（同样见文章[6]第 2.2.4 节内容）。

因此，对于文本分类这个场景来说，只需要构造原始文本对应的 Token 序列，并在首尾分别再加上一个[CLS]符和[SEP]符作为输入即可。

### 4.2.2 语料介绍

在这里，我们使用到的数据集是今日头条开放的一个新闻分类数据集[13]，一共包含有 382688 条数据，15 个类别。同时掌柜已近将其进行了格式化处理，以 7:2:1 的比例划分成了训练集、验证集和测试集 3 个部分。如下所示便是部分示例数据：

```
1 千万不要乱申请网贷，否则后果很严重_!_4
2 10 年前的今天，纪念 5.12 汶川大地震 10 周年_!_11
3 怎么看待杨毅在一 NBA 直播比赛中说詹姆斯的球场统治力已经超过乔丹、伯德和科比？_!_3
4 戴安娜王妃的车祸有什么谜团？_!_2
```

其中_!_左边为新闻标题，也就是后面需要用到的分类文本，右边为类别标签。

### 4.2.3 数据集预览

同样，在正式介绍如何构建数据集之前我们先通过一张图来了解一下整个构建的流程，以便做到心中有数，不会迷路。假如我们现在有两个样本构成了一个 batch，那么其整个数据的处理过程则如图 4-2 所示。

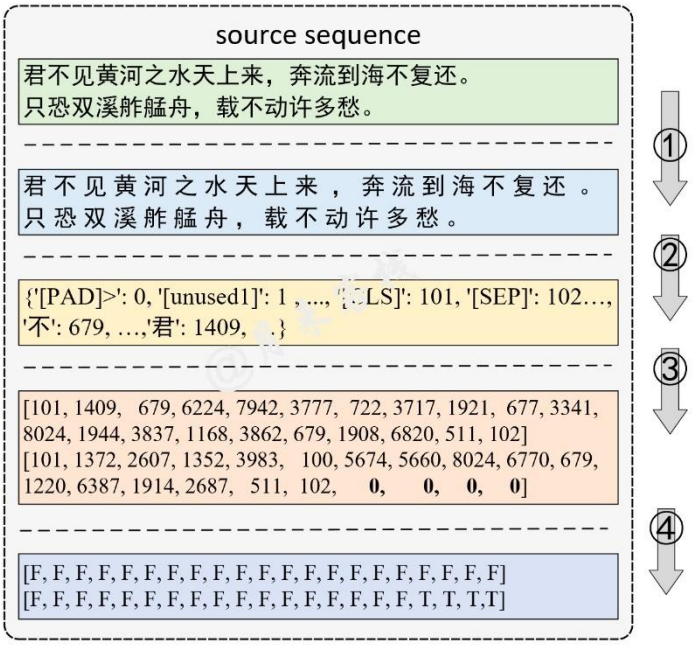



图 4-2. 文本分类数据集构建流程图

如图 4-2 所示，第 1 步需要将原始的数据样本进行分字（tokenize）处理；第 2 步再根据 tokenize 后的结果构造一个字典，不过在使用 BERT 预训练时并不需

要我们自己来构造这个字典，直接载入谷歌开源的 vocab.txt 文件构造字典即可，因为只有 vocab.txt 中每个字的索引顺序才与开源模型中每个字的 Embedding 向量一一对应的。第 3 步则是根据字典将 tokenize 后的文本序列转换为 Token 序列，同时在 Token 序列的首尾分别加上[CLS]和[SEP]符号，并进行 Padding。第 4 步则是根据第 3 步处理后的结果生成对应的 Padding Mask 向量。

最后，在模型训练时只需要将第 3 步和第 4 步处理后的结果一起喂给模型即可。

### 4.2.4 数据集构建

**第 1 步：定义 tokenize**

第 1 步需要完成的就是将输入进来的文本序列 tokenize 到字符级别。对于中文语料来说就是将每个字和标点符号都给切分开。在这里，我们可以借用 transformers 包中的 BertTokenizer 方法来完成，如下所示：

```
1 if __name__ == '__main__':
2     model_config = ModelConfig()
3     tokenizer = BertTokenizer.from_pretrained(model_config.
                                                pretrained_model_dir).tokenize
4     print(tokenizer("青山不改，绿水长流，我们月来客栈见！"))
5     print(tokenizer("10 年前的今天，纪念 5.12 汶川大地震 10 周年"))
6
7 # ['青', '山', '不', '改', '，', '绿', '水', '长', '流', '，', '我', '们', '月', '来', '客
  ', '栈', '见', '！']
8 # ['10', '年', '前', '的', '今', '天', '，', '纪', '念', '5', '.', '12', '汶', '川', '大',
  '地', '震', '10', '周', '年']
```

在上述代码中，第 2-3 行就是根据指定的路径（BERT 预训练模型的路径）来载入一个分字模型；第 7-8 行便是 tokenize 后的结果。

**第 2 步：建立词表**

由于 BERT 预训练模型中已经有了一个给定的词表（vocab.txt），因此我们并不需要根据自己的语料来建立一个词表。当然，也不能够根据自己的语料来建立词表，因为相同的字在我们自己构建的词表中和 vocab.txt 中的索引顺序肯定会不一样，而这就会导致后面根据 token id 取出来的向量是错误的。

进一步，我们只需要将 vocab.txt 中的内容读取进来形成一个词表即可，代码如下：

```
1 class Vocab:
2     UNK = '[UNK]'
3     def __init__(self, vocab_path):
4         self.stoi = {}
5         self.itos = []
```

```
        with open(vocab_path, 'r', encoding='utf-8') as f:
            for i, word in enumerate(f):
                w = word.strip('\n')
                self.stoi[w] = i
                self.itos.append(w)

    def __getitem__(self, token):
        return self.stoi.get(token, self.stoi.get(Vocab.UNK))

    def __len__(self):
        return len(self.itos)
```

接着便可以定义一个方法来实例化一个词表：

```
def build_vocab(vocab_path):
    return Vocab(vocab_path)

if __name__ == '__main__':
    vocab = build_vocab()
```

在经过上述代码处理后，我们便能够通过 vocab.itos 得到一个列表，返回词表中的每一个词；通过 vocab.itos[2]返回得到词表中对应索引位置上的词；通过 vocab.stoi 得到一个字典，返回词表中每个词的索引；通过 vocab.stoi['月']返回得到词表中对应词的索引；通过 len(vocab)来返回词表的长度。如下便是建立后的词表：

```
{'[PAD]': 0, '[unused1]': 1, '[unused2]': 2, '[unused3]': 3, '[unused4]': 4,
'[unused5]': 5,'[unused6]': 6, '[unused7]': 7, '[unused8]': 8,'[unused9]': 9,
'[unused10]': 10, '[unused11]': 11, '[unused12]': 12, '[unused13]': 13,  ...
'[unused42]': 42, ....'乐': 727, '乒': 728, '乓': 729, '乔': 730, '乖': 731,
 '乘': 732, '乘': 733, '乙': 734, '乜': 735, '九': 736, '乞': 737, '也': 738,
 '习':739,'乡': 740,'书': 741,'乩': 742,'买': 743, '乱': 744,'乳': 745, ....}
```

此时，我们就需要定义一个类，并在类的初始化过程中根据训练语料完成字典的构建等工作，代码如下：

```
class LoadSingleSentenceClassificationDataset:
    def __init__(self,
                 vocab_path='./vocab.txt',  #
                 tokenizer=None,
                 batch_size=32,
                 max_sen_len=None,
                 split_sep='\n',
                 max_position_embeddings=512,
                 pad_index=0,
                 is_sample_shuffle=True):
```

```
11
12          self.tokenizer = tokenizer
13          self.vocab = build_vocab(vocab_path)
14          self.PAD_IDX = pad_index
15          self.SEP_IDX = self.vocab['[SEP]']
16          self.CLS_IDX = self.vocab['[CLS]']
17          self.batch_size = batch_size
28          self.split_sep = split_sep
19          self.max_position_embeddings = max_position_embeddings
20          if isinstance(max_sen_len, int) and max_sen_len >
                                                    max_position_embeddings:
23              max_sen_len = max_position_embeddings
24          self.max_sen_len = max_sen_len
25          self.is_sample_shuffle = is_sample_shuffle
```

在上述代码中，第 3 行 vocab_path 表示本地词表的路径。第 6 行 max_sen_len 表示最大样本长度，当 max_sen_len = None 时，即以每个 batch 中最长样本长度为标准，对其它进行 padding；当 max_sen_len = 'same'时，以整个数据集中最长样本为标准，对其它进行 padding；当 max_sen_len = 50， 表示以某个固定长度符样本进行 padding，多余的截掉。第 7 行 split_sep 表示样本与标签之间的分隔符。is_sample_shuffle 表示是否打乱数据集。第 14-16 行为建立词表并取对应特殊字符的索引。第 19 行中 max_position_embeddings 为最大样本长度，最大为 512。第 20-23 行则是用来判断传入的最大样本长度。

**第 3 步：转换为 Token 序列**

在得到构建的字典后，便可以通过如下方法来分别将训练集、验证集和测试集转换成 Token 序列，其作用是将每一句话中的每一个词根据字典转换成索引的形式，同时返回所有样本中最长样本的长度。实现代码如下：

```
1     def data_process(self, filepath):
2         raw_iter = open(filepath, encoding="utf8").readlines()
3         data = []
4         max_len = 0
5         for raw in tqdm(raw_iter, ncols=80):
6             line = raw.rstrip("\n").split(self.split_sep)
7             s, l = line[0], line[1]
8             tmp = [self.CLS_IDX] + [self.vocab[token] for token in
                                                    self.tokenizer(s)]
9             if len(tmp) > self.max_position_embeddings - 1:
10                tmp = tmp[:self.max_position_embeddings - 1]
11            tmp += [self.SEP_IDX]
12            tensor_ = torch.tensor(tmp, dtype=torch.long)
13            l = torch.tensor(int(l), dtype=torch.long)
```

```
            max_len = max(max_len, tensor_.size(0))
            data.append((tensor_, 1))
        return data, max_len
```

在上述代码中，第 6-7 行便是用来取得文本和标签；第 8 行则是首先对序列进行 tokenize，然后转换成 Token 序列并在最前面加上分类标志位[CLS]。第 9-11 行则是用来对 Token 序列进行截取，最长为 max_position_embeddings 个字符，默认为 512，并同时在末尾加上[SEP]符号。不过掌柜认为其实末尾不加[SEP]应该也不会有影响，因为这本来是单个序列的分类。第 14 行则是用来保存最长序列的长度。

在处理完成后，4.2.2 节中的 4 个样本将会被转换成类似如下形式：

```
tensor([[101, 1283,  674,  679, 6206,  744, 4509, 6435, 5381,..,  0,    0],
        [101, 8108, 2399, 1184, 4638,  791, 2399, 8024, 5279,..,  0,    0],
        [101, 2582,  720, 4692, 2521, 3342, 3675, 1762,  671,..,8043, 102],
        [101, 2785, 2128, 2025, 4374, 1964, 4638, 6756, 1730,..,  0,    0]])
torch.Size([39, 4])
```

从上面的输出结果可以看出，101 就是[CLS]在词表中的索引位置，102 则是[SEP]在词表中的索引；其它非 0 值就是 tokenize 后的文本序列转换成的 Token 序列。同时可以看出，这里的结果是以第 3 个样本的长度 39 对其它样本进行 padding 的，并且 padding 的 Token ID 为 0。因此，下面我们就来介绍样本的 padding 处理。

**第 4 步：padding 处理与 mask**

从第 3 步的输出结果看出，在对原始文本序列 tokenize 转换为 Token ID 后还需要对其进行 padding 处理。对于这一处理过程可以通过如下代码来完成：

```
def pad_sequence(sequences,batch_first=False,max_len=None, padding_value=0):
    if max_len is None:
        max_len = max([s.size(0) for s in sequences])
    out_tensors = []
    for tensor in sequences:
        if tensor.size(0) < max_len:
            tensor = torch.cat([tensor, torch.tensor([padding_value]
                                        (max_len - tensor.size(0)))], dim=0)
        else:
            tensor = tensor[:max_len]
        out_tensors.append(tensor)
    out_tensors = torch.stack(out_tensors, dim=1)
    if batch_first:
        return out_tensors.transpose(0, 1)
    return out_tensors
```

在上述代码中，第 1 行 sequences 为待 padding 的序列所构成的列表，其中的每一个元素为一个样本的 Token 序列；batch_first 表示是否将 batch_size 这个维度放在第 1 个；max_len 表示指定最大序列长度，当 max_len = 50 时，表示以某个固定长度对样本进行 padding 多余的截掉，当 max_len=None 时表示以当前 batch 中最长样本的长度对其它进行 padding。第 2-3 行用来获取 padding 的长度；第 5-11 行则是遍历每一个 Token 序列，根据 max_len 来进行 padding。第 12-13 行是将 batch_size 这个维度放到最前面。

如下为使用示例：

```
1 if __name__ == '__main__':
2     a = torch.tensor([1, 2, 3])
3     b = torch.tensor([4, 5, 6, 7, 8])
4     c = torch.tensor([9, 10])
5     d = pad_sequence([a, b, c], max_len=None).size()
6         # torch.Size([5, 3])
```

进一步，我们需要定义一个方法来对每个 batch 的 Token 序列进行 padding 处理：

```
1     def generate_batch(self, data_batch):
2         batch_sentence, batch_label = [], []
3         for (sen, label) in data_batch:
4             batch_sentence.append(sen)
5             batch_label.append(label)
6         batch_sentence = pad_sequence(batch_sentence,
7                                       padding_value=self.PAD_IDX,
8                                       batch_first=False,
9                                       max_len=self.max_sen_len)
10        batch_label = torch.tensor(batch_label, dtype=torch.long)
11        return batch_sentence, batch_label
```

上述代码的作用就是对每个 batch 的 Token 序列进行 padding 处理。

最后，对于每一序列的 attention_mask 向量，我们只需要判断其是否等于 padding_value 便可以得到这一结果，可见第 5 步中的使用示例。

**第 5 步：构造 DataLoade 与使用示例**

经过前面 4 步的操作，整个数据集的构建就算是已经基本完成了，只需要再构造一个 DataLoader 迭代器即可，代码如下：

```
1     def load_train_val_test_data(self, train_file_path=None,
2                                  val_file_path=None,
3                                  test_file_path=None,
4                                  only_test=False):
5         test_data, _ = self.data_process(test_file_path)
```

```
6         test_iter = DataLoader(test_data, batch_size=self.batch_size,
7                                shuffle=False,
                                 collate_fn=self.generate_batch)
8         if only_test:
9             return test_iter
10        train_data, max_sen_len = self.data_process(train_file_path)
11        if self.max_sen_len == 'same':
12            self.max_sen_len = max_sen_len
13        val_data, _ = self.data_process(val_file_path)
14        train_iter = DataLoader(train_data, batch_size=self.batch_size,
15                                shuffle=self.is_sample_shuffle,
                                  collate_fn=self.generate_batch)
16        val_iter = DataLoader(val_data, batch_size=self.batch_size,
17                              shuffle=False, collate_fn=self.generate_batch)
18        return train_iter, test_iter, val_iter
```

在上述代码中，第 5-7 行用来得到预处理后的数据并构造对应的 DataLoader，其中 generate_batch 将作为一个参数传入来对每个 batch 的样本进行处理；第 8-9 行则判断是否只返回测试集；同理，第 10-18 行则是用来构造相应的训练集和验证集。在完成类 LoadSingleSentenceClassificationDataset 所有的编码过程后，便可以通过如下形式进行使用：

```
1 from Tasks.TaskForSingleSentenceClassification import ModelConfig
2 from utils.data_helpers import LoadSingleSentenceClassificationDataset
3 from transformers import BertTokenizer
4
5 if __name__ == '__main__':
6     model_config = ModelConfig()
7     tokenizer= BertTokenizer.from_pretrained(
                              model_config.pretrained_model_dir).tokenize
8     load_dataset = LoadSingleSentenceClassificationDataset(
9         vocab_path=model_config.vocab_path,
10        tokenizer=tokenizer,
11        batch_size=model_config.batch_size,
12        max_sen_len=model_config.max_sen_len,
13        split_sep=model_config.split_sep,
14        max_position_embeddings=model_config.max_position_embeddings,
15        pad_index=model_config.pad_token_id,
16        is_sample_shuffle=model_config.is_sample_shuffle)
17
18    train_iter, test_iter, val_iter = \
19        load_dataset.load_train_val_test_data(model_config.train_file_path,
```

```
                                                model_config.val_file_path,
                                                model_config.test_file_path)
    for sample, label in train_iter:
        print(sample.shape)  # [seq_len,batch_size]
        print(sample.transpose(0, 1))
        padding_mask = (sample == load_dataset.PAD_IDX).transpose(0, 1)
        print(padding_mask)
        print(label)
        break
```

在上述代码中，第 6 行是载入配置参数。第 7 行是实例化一个 tokenizer。第 8-16 行是实例化数据集载入对象。第 18-21 行是返回训练集、测试机和验证集的迭代器；第 22-27 行是遍历每个 batch 的样本。

执行完上述代码后便可以得到如下所示的结果：

```
torch.Size([39, 4])
tensor([[ 101, 1283,  674,  679, 6206, ... 7028,  102, ...  ,0,    0,    0],
          ...
        ])
tensor([[False, False, False,False,False, ..., False,... True, True,  True],
        ...
        ])
tensor([ 4,...])
```

到此，对于整个数据集构建部分的内容就算是介绍完了，接下来我们再来看如何加载预训练模型进行微调。

## 4.3 加载预训练模型

在介绍模型微调之前，我们先来看看当我们拿到一个开源的模型参数后怎么读取以及分析。下面掌柜就以 huggingface 开源的 PyTorch 训练的 bert-base-chinese 模型参数[10]为例进行介绍。

### 4.3.1 查看模型参数

在第 3 节内容中，尽管掌柜已经大致介绍了如何通过 PyTorch 来读取和加载模型参数，但是这里仍旧有必要以 bert-base-chinese 参数为例再进行一次详说明。根据第 3.2 节内容的介绍可知，我们可以通过如下方式来查看本地模型中的参数情况：

```
import torch
loaded_paras = torch.load('./pytorch_model.bin')
print(type(loaded_paras))
print(len(list(loaded_paras.keys())))
print(list(loaded_paras.keys()))
```

执行完上述代码后，便可以得到如下输出结果：

```
<class 'collections.OrderedDict'>
207
['bert.embeddings.word_embeddings.weight',
'bert.embeddings.position_embeddings.weight',
'bert.embeddings.token_type_embeddings.weight',
.....
'bert.encoder.layer.11.output.dense.bias',
'bert.encoder.layer.11.output.LayerNorm.gamma',
'bert.encoder.layer.11.output.LayerNorm.beta',.....]
```

从上面的输出结果可以看到，参数 pytorch_model.bin 被载入后变成了一个有序的字典 OrderedDict，并且其中一共有 207 个参数，其名字分别就是列表中的各个元素。进一步，我们还可以将各个参数的形状打印出来看一看：

```
for name in loaded_paras.keys():
    print(f"### 参数:{name},形状:{loaded_paras[name].size()}")

# 参数:bert.embeddings.word_embeddings.weight,形状:torch.Size([21128,768])
# 参数:bert.embeddings.position_embeddings.weight,形状:torch.Size([512, 768])
# 参数:bert.embeddings.token_type_embeddings.weight,形状:torch.Size([2, 768])
# 参数:bert.embeddings.LayerNorm.gamma,形状:torch.Size([768])
......
# 参数:bert.encoder.layer.11.output.dense.weight,形状:torch.Size([768, 3072])
# 参数:bert.encoder.layer.11.output.dense.bias,形状:torch.Size([768])
# 参数:bert.encoder.layer.11.output.LayerNorm.gamma,形状:torch.Size([768])
# 参数:bert.encoder.layer.11.output.LayerNorm.beta,形状:torch.Size([768])
# 参数:bert.pooler.dense.weight,形状:torch.Size([768, 768])
# 参数:bert.pooler.dense.bias,形状:torch.Size([768])
# 参数:cls.predictions.bias,形状:torch.Size([21128])
# 参数:cls.predictions.transform.dense.weight,形状:torch.Size([768, 768])
# 参数:cls.predictions.transform.dense.bias,形状:torch.Size([768])
# 参数:cls.predictions.transform.LayerNorm.gamma,形状:torch.Size([768])
# 参数:cls.predictions.transform.LayerNorm.beta,形状:torch.Size([768])
# 参数:cls.predictions.decoder.weight,形状:torch.Size([21128, 768])
# 参数:cls.seq_relationship.weight,形状:torch.Size([2, 768])
# 参数:cls.seq_relationship.bias,形状:torch.Size([2])
```

同样，我们还可以直接打印出某个参数具体的值。不过这里并不需要分析所以就不用打印。

到此，对于本地的模型参数的信息就分析完了。不过想要将它迁移到自己所搭建的模型上还要进一步的来分析自己所搭建网络的参数信息。

### 4.3.2 载入并初始化

在第 2 节内容中，掌柜已经详细地介绍了如何实现整个 BERT 模型，但是对于如何载入已有参数来初始化网络中的参数还并未介绍。在将本地参数迁移到一个新的模型之前，除了像上面那样分析本地参数之外，我们还需要将网络的参数信息也打印出来看一下，以便将两者一一对应上。

```
1 json_file = '../bert_base_chinese/config.json'
2 config = BertConfig.from_json_file(json_file)
3 bert_model = BertModel(config)
4 print("\n  =======  BertMolde 参数: ========")
5 print(len(bert_model.state_dict()))
6 for param_tensor in bert_model.state_dict():
7     print(param_tensor, "\t", bert_model.state_dict()[param_tensor].size())
```

在执行完上述代码后，便可以得到如下输出结果：

```
 1  =======  BertMolde 参数: ========
 2 200
 3 # bert_embeddings.position_ids  torch.Size([1, 512])
 4 # bert_embeddings.word_embeddings.embedding.weight torch.Size([21128,768])
 5 # bert_embeddings.position_embeddings.embedding.weight torch.Size([512,768])
 6 # bert_embeddings.token_type_embeddings.embedding.weight torch.Size([2,768])
 7 ......
 8 # bert_encoder.bert_layers.11.bert_output.dense.bias    torch.Size([768])
 9 # bert_encoder.bert_layers.11.bert_output.LayerNorm.weight torch.Size([768])
10 # bert_encoder.bert_layers.11.bert_output.LayerNorm.bias    torch.Size([768])
11 # bert_pooler.dense.weight      torch.Size([768, 768])
12 # bert_pooler.dense.bias    torch.Size([768])
```

从上面的输出结果可以发现，BertMolde 一共有 200 个参数，而 bert-base-chinese 一共有 207 个参数。这里需要注意的是 BertMolde 模型中的 position_ids 这个参数并不是模型中需要训练的参数，只是一个默认的初始值。最后，经分析（两者一一进行对比）后发现 bert-base-chinese 中除了最后的 8 个参数以外，其余的 199 个参数和 BertMolde 模型中的 199 个参数一样且顺序也一样。

因此，最后我们可以通过在 BertMolde 类（Bert.py 文件中）中再加入一个如下所示的方法来用 bert-base-chinese 中的参数初始化 BertMolde 中的参数：

```
1     @classmethod
2     def from_pretrained(cls, config, pretrained_model_dir=None):
3         model = cls(config)
4         pretrained_model_path = os.path.join(pretrained_model_dir,
                                               "pytorch_model.bin")
5         loaded_paras = torch.load(pretrained_model_path)
6         state_dict = deepcopy(model.state_dict())
```

```
7          loaded_paras_names = list(loaded_paras.keys())[:-8]
8          model_paras_names = list(state_dict.keys())[1:]
9          for i in range(len(loaded_paras_names)):
10             state_dict[model_paras_names[i]] =
                                          loaded_paras[loaded_paras_names[i]]
11             logging.info(f"成功将参数{loaded_paras_names[i]}赋值给
                                                  {model_paras_names[i]}")
12         model.load_state_dict(state_dict)
13         return model
```

在上述代码中，第 3 行是初始化模型，cls 为未实例化的对象，即一个未实例化的 BertModel 对象；第 4-5 行用来载入本地的 bert-base-chinese 参数；第 6 行用来拷贝一份 BertModel 中的网络参数，这是因为我们无法直接修改里面的值；第 7-10 行则是根据我们上面的分析，将 bert-base-chinese 中的参数赋值到 state_dict 中；第 12 行是用 state_dict 中的参数来初始化 BertModel 中的参数。

最后，我们只需要通过如下方式便可以返回一个通过 bert-base-chinese 初始化的 BERT 模型：

```
1 bert = BertModel.from_pretrained(config, bert_pretrained_model_dir)
```

当然，如果你需要冻结其中某些层的参数不参与模型训练，那么可以通过类似如下所示的代码来进行设置：

```
1 for para in bert_model.parameters():
2     if xxxx:
3         para.requires_grad = False
```

到此，对于整个预训练模型的加载过程就介绍完了，接下来让我们正式进入到基于 BERT 预训练模型的文本分类场景中。

## 4.4 文本分类

### 4.4.1 前向传播

在介绍完如何分析和载入本地 BERT 预训练模型后，接下来我们首先要做的就是实现文本分类的前向传播过程。如图 2-4 所示，在 BertForSentenceClassification.py 文件中，我们通过定义如下一个类来完成整个前向传播的过程：

```
1 from ..BasicBert.Bert import BertModel
2 import torch.nn as nn
3 class BertForSentenceClassification(nn.Module):
4     def __init__(self, config, bert_pretrained_model_dir=None):
5         super(BertForSentenceClassification, self).__init__()
6         self.num_labels = config.num_labels
7         if bert_pretrained_model_dir is not None:
```

```
8            self.bert = BertModel.from_pretrained(config,
                                                   bert_pretrained_model_dir)
9        else:
10           self.bert = BertModel (config)
11       self.dropout = nn.Dropout(config.hidden_dropout_prob)
12       self.classifier = nn.Linear(config.hidden_size, self.num_labels)
```

在上述代码中，第 4 行代码分别就是用来指定模型配置和预训练模型的路径；第 7-10 行代码则是用来定义一个 BERT 模型，可以看到如果预训练模型的路径存在则会返回一个由 bert-base-chinese 参数初始化后的 BERT 模型，否则则会返回一个随机初始化参数的 BERT 模型；第 12 行则是定义最后的分类层。

最后，整个前向传播的实现代码如下所示：

```
1    def forward(self, input_ids, # [src_len, batch_size]
2                attention_mask=None, # [batch_size, src_len]
3                token_type_ids=None, # [src_len, batch_size]
4                position_ids=None, # [1,src_len]
5                labels=None): # [batch_size,]
6        pooled_output, _ = self.bert(input_ids=input_ids,
7                                     attention_mask=attention_mask,
8                                     token_type_ids=token_type_ids,
9                                     position_ids=position_ids)
10       pooled_output = self.dropout(pooled_output)
11       logits = self.classifier(pooled_output)  # [batch_size, num_label]
12       if labels is not None:
13           loss_fct = nn.CrossEntropyLoss()
14           loss = loss_fct(logits.view(-1,self.num_labels),labels.view(-1))
15           return loss, logits
16       else:
17           return logits
```

在上述代码中，第 6-9 行返回的就是原始 BERT 网络的输出，其中 pooled_output 为 BERT 第 1 个位置的向量经过一个全连接层后的结果，第 2 个参数是 BERT 中所有位置的向量；第 10-11 行便是用来进行文本分类的分类层；第 12-17 行则是用来判断返回损失值还是返回 logits 值。

### 4.4.2 模型训练

如图 2-5 所示，我们将在 Tasks 目录下新建一个名为 TaskForSingleSentenceClassification.py 的模块来完成分类模型的微调训练任务。

首先，我们需要在其中定义一个 ModelConfig 类来对分类模型中的超参数进行管理，代码如下所示：

```
1 class ModelConfig:
2     def __init__(self):
```

```
3         self.project_dir =
                  os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
4         self.dataset_dir = os.path.join(
                    self.project_dir, 'data', 'SingleSentenceClassification')
5         self.pretrained_model_dir = os.path.join(
                                        self.project_dir, "bert_base_chinese")
6         self.vocab_path=os.path.join(self.pretrained_model_dir, 'vocab.txt')
7         self.device = torch.device('cuda:0'
                                     if torch.cuda.is_available() else 'cpu')
8         self.train_file_path =
                          os.path.join(self.dataset_dir,'toutiao_train.txt')
9         self.val_file_path =
                          os.path.join(self.dataset_dir, 'toutiao_val.txt')
10        self.test_file_path =
                          os.path.join(self.dataset_dir, 'toutiao_test.txt')
11        self.model_save_dir = os.path.join(self.project_dir, 'cache')
12        self.logs_save_dir = os.path.join(self.project_dir, 'logs')
13        self.split_sep = '_!_'
14        self.is_sample_shuffle = True
15        self.batch_size = 64
16        self.max_sen_len = None
17        self.num_labels = 15
18        self.epochs = 10
19        self.model_val_per_epoch = 2
20        logger_init(log_file_name='single', log_level=logging.INFO,
21                    log_dir=self.logs_save_dir)
22        if not os.path.exists(self.model_save_dir):
23            os.makedirs(self.model_save_dir)
24
25        # 把原始 bert 中的配置参数也导入进来
26        bert_config_path =
                      os.path.join(self.pretrained_model_dir, "config.json")
27        bert_config = BertConfig.from_json_file(bert_config_path)
28        for key, value in bert_config.__dict__.items():
29            self.__dict__[key] = value
30        logging.info(" ### 将当前配置打印到日志文件中 ")
31        for key, value in self.__dict__.items():
32            logging.info(f"###   {key} = {value}")
```

在上述代码中，第 2-23 行则是分别用来定义模型中的一些数据集目录、超参数和初始化日志打印类等；第 25-29 行则是将原始 bert_base_chinese 配置文件，即 config.json 中的参数也导入到类 ModelConfig 中；第 31-32 行则是将所有的超

参数配置情况一同打印到日志文件中方便后续分析，更多关于日志的内容可以参加文章[14]。

最后，我们只需要再定义一个 train()函数来完成模型的训练即可，代码如下：

```
def train(config):
    model = BertForSentenceClassification(config,
                                          config.pretrained_model_dir)
    #......
    optimizer = torch.optim.Adam(model.parameters(), lr=5e-5)
    model.train()
    tokenizer = BertTokenizer.from_pretrained(config.pretrained_model_dir)
    data_loader = LoadSingleSentenceClassificationDataset(
                    vocab_path=config.vocab_path,
                    tokenizer=tokenizer.tokenize,
                    batch_size=config.batch_size,
                    max_sen_len=config.max_sen_len,
                    split_sep=config.split_sep,
                    max_position_embeddings=config.max_position_embeddings,
                    pad_index=config.pad_token_id)
    train_iter, test_iter, val_iter = data_loader.load_train_val_test_data(
        config.train_file_path,config.val_file_path,config.test_file_path)
    max_acc = 0
    for epoch in range(config.epochs):
        losses = 0
        start_time = time.time()
        for idx, (sample, label) in enumerate(train_iter):
            sample = sample.to(config.device)  # [src_len, batch_size]
            label = label.to(config.device)
            padding_mask = (sample == data_loader.PAD_IDX).transpose(0, 1)
            loss, logits = model(input_ids=sample,
                                 attention_mask=padding_mask,
                                 token_type_ids=None,
                                 position_ids=None,
                                 labels=label)
                         #.........
            acc = (logits.argmax(1) == label).float().mean()
            #.........
        if (epoch + 1) % config.model_save_per_epoch == 0:
            acc = evaluate(val_iter, model, config.device)
            logging.info(f"Accuracy on val {acc:.3f}")
            #.........
```

在上述代码中，第 2-3 行用来初始化一个基于 BERT 的文本分类模型；第 8-17 行则是载入相应的数据集；第 19-36 行则是整个模型的训练过程，完整示例代码可参见[6]，掌柜也在代码中进行了详细的注释。

紧接着，可以通过如下方式来进行模型训练：

```
1 if __name__ == '__main__':
2     model_config = ModelConfig()
3     train(model_config)
```

如下便是网络的部分训练结果：

```
 1 -- INFO: Epoch: 0, Batch[0/4186], Train loss :2.862, Train acc: 0.125
 2 -- INFO: Epoch: 0, Batch[10/4186], Train loss :2.084, Train acc: 0.562
 3 -- INFO: Epoch: 0, Batch[20/4186], Train loss :1.136, Train acc: 0.812
 4 -- INFO: Epoch: 0, Batch[30/4186], Train loss :1.000, Train acc: 0.734
 5 ...
 6 -- INFO: Epoch: 0, Batch[4180/4186], Train loss :0.418, Train acc: 0.875
 7 -- INFO: Epoch: 0, Train loss: 0.481, Epoch time = 1123.244s
 8 ...
 9 -- INFO: Epoch: 9, Batch[4180/4186], Train loss :0.102, Train acc: 0.984
10 -- INFO: Epoch: 9, Train loss: 0.100, Epoch time = 1130.071s
11 -- INFO: Accurcay on val 0.884
12 -- INFO: Accurcay on test 0.888
```

### 4.4.3 模型推理

在完成模型的训练过程后，便可以将训练过程中保存好的模型用于任务的推理场景中。这部分代码实现起来也比较容易，只需要按照第 3.3 节内容中介绍的方式使用即可，具体代码实现如下：

```
 1 def inference(config):
 2     model = BertForSentenceClassification(config,
 3                                           config.pretrained_model_dir)
 4     model_save_path = os.path.join(config.model_save_dir, 'model.pt')
 5     if os.path.exists(model_save_path):
 6         loaded_paras = torch.load(model_save_path)
 7         model.load_state_dict(loaded_paras)
 8         logging.info("## 成功载入已有模型，进行预测......")
 9     model = model.to(config.device)
10     tokenizer = BertTokenizer.from_pretrained(config.pretrained_model_dir)
11     data_loader = LoadSingleSentenceClassificationDataset(
12                     vocab_path=config.vocab_path,
13                     tokenizer=tokenizer.tokenize,
14                     batch_size=config.batch_size,
15                     max_sen_len=config.max_sen_len,
16                     split_sep=config.split_sep,
```

```
                    max_position_embeddings=config.max_position_embeddings,
                    pad_index=config.pad_token_id,
                    is_sample_shuffle=config.is_sample_shuffle)
    train_iter, test_iter, val_iter = data_loader.load_train_val_test_data(
          config.train_file_path,config.val_file_path,config.test_file_path)
    acc = evaluate(test_iter,model,config.device, data_loader.PAD_IDX)
    logging.info(f"Acc on test:{acc:.3f}")
```

在上述代码中，第 2-3 行是用来实例化一个文本分类模型；第 4-8 行是载入本地模型参数来重新初始化网络中的参数；第 11-21 行是载入新的测试数据；第 22 行是返回模型在测试集上的准确率，当然在实际情况中也可以按照需求进行改动。

同时，测试集上准确率的实现过程如下所示：

```
def evaluate(data_iter, model, device, PAD_IDX):
    model.eval()
    with torch.no_grad():
        acc_sum, n = 0.0, 0
        for x, y in data_iter:
            x, y = x.to(device), y.to(device)
            padding_mask = (x == PAD_IDX).transpose(0, 1)
            logits = model(x, attention_mask=padding_mask)
            acc_sum += (logits.argmax(1) == y).float().sum().item()
            n += len(y)
        model.train()
        return acc_sum / n
```

紧接着，可以通过如下方式来进行模型的推理过程：

```
if __name__ == '__main__':
    model_config = ModelConfig()
    inference(model_config)
```

到此，对于简单的单文本分类任务就介绍完了。在下一节内容中，掌柜将会介绍如何在文本蕴含任务中，即输入两个句子来进行分类的场景下，进行 BERT 预训练模型的微调

# 第 5 节 基于 BERT 预训练模型的文本蕴含任务

经过第 4 节内容的介绍，相信大家对于如何利用 BERT 预训练模型来进行单文本分类的整体流程已经有了较为清晰的认识。当然 BERT 的能力显然远不止于此，因此在这节内容中，掌柜将会介绍第 2 个下游任务的微调场景，即如何基于 BERT 预训练模型来完成文本蕴含（文本对的分类）任务。

## 5.1 任务构造原理

所谓文本对分类指的就是同时给模型输入两句话，然后让模型来判断两句话之间的关系，所以本质上也就变成了一个文本分类任务。同时，鉴于第 1 个任务场景用到的是中文语料，因此在第 2 个场景中我们将会用英文语料进行示例。这样便于大家两者都能掌握。

总的来说，基于 BERT 的文本蕴含任务同第 4 节中介绍的单文本分类任务本质上没有任何不同，最终都是对一个文本序列进行分类。只是按照 BERT 模型的思想，文本对分类任务在数据集的构建过程中需要通过 Segment Embedding 来区分前后两个不同的序列，并且两个句子之间需要通过一个[SEP]符号来进行分割，因此本节内容的核心就在于如何构建数据集。

总结起来，文本对的分类任务除了在模型输入上发生了变换，其它地方均与单文本分类任务一样，同样也是取最后一层的[CLS]向量进行分类。接下来，掌柜首先就来介绍如何构造文本分类的数据集。

## 5.2 数据预处理

### 5.2.1 输入介绍

由于在文本对分类任务这个场景中模型的输入包含两个序列，因此在构建数据集的时候不仅仅需要进行 Token Embedding 操作，同时还要对两个序列进行 Segment Embedding 操作。对于 Position Embedding 来说在任何场景下都不需要对其指定输入，因为我们在代码实现时已经做了相应默认时的处理。

因此，对于文本对分类这个场景来说构建模型所需的训练集需要完成两个步骤：①需要构造原始文本对应的 Token 序列，然后在最前面加上一个[CLS]符，两个序列之间以及整个序列的末尾分别再加上一个[SEP]符；②根据两个序列各自的长度再构建一个类似[0,0,0,...,1,1,1,...]的 token_type_ids 向量。最后将两者均作为模型的输入即可。

### 5.2.2 语料介绍

在这里，我们使用到的是论文中所提到的 MNLI（The Multi-Genre Natural Language Inference Corpus，多类型自然语言推理数据库）自然语言推断任务数据

集[3]。也就是给定前提（premise）语句和假设（hypothesis）语句，任务是预测前提语句是否包含假设（蕴含，entailment），与假设矛盾（矛盾，contradiction）或者两者都不（中立，neutral）。

如下所示便是原始示例数据中的两个样本：

```
{"annotator_labels": ["entailment", "neutral", "entailment", "neutral",
"entailment"],"genre": "oup","gold_label": "entailment", "pairID": "82890e",
"promptID": "82890", "sentence1": " From Home Work to Modern Manufacture",
"sentence1_binary_parse": "( From ( ( Home Work ) ( to ( Modern
Manufacture ) ) ) )" ...",
"sentence1_parse": "(ROOT (PP (IN From) (NP (NP (NNP Home) (NNP Work)) (PP
(TO to) (NP (NNP Modern)...",
"sentence2": "Modern manufacturing has changed over time.",
"sentence2_binary_parse": "( ( Modern manufacturing ) ( ( has ( ....",
"sentence2_parse": "(ROOT (S (NP (NNP Modern) (NN manufacturing)) (VP ..." }

{"annotator_labels": ["neutral", "neutral", "entailment", "neutral",
"neutral"],"genre":"nineeleven","gold_label": "neutral", "pairID": "16525n",
"promptID": "16525", "sentence1": "They were promptly executed.",
"sentence1_binary_parse": "( They ( ( were ( promptly executed ) ) . ) )",
"sentence1_parse": "(ROOT (S (NP (PRP They)) (VP (VBD were) (VP (ADVP (RB
not) (RB only)) (VBD earned) ...",
"sentence2": "They were executed immediately upon capture.",
"sentence2_binary_parse": "( They ( ( were ( ( executed immediately apparent
(nor why the item is ...",
"sentence2_parse": "(ROOT (S (NP (PRP They)) (VP (VBD were) (VP (VBN called)
(NP (NP (DT a) (JJ floppy))..."}
```

由于该数据集同时也可用于其它任务中，因此除了我们需要的前提和假设两个句子和标签之外，还有每个句子的语法解析结构等等。在这里，下载完成数据后只需要执行项目中的 format.py 脚本即可将原始数据划分成训练集、验证集和测试集。格式化后的数据形式如下所示：

```
From Home Work to Modern Manufacture_!_Modern manufacturing has changed over
time._!_1
They were promptly executed._!_They were executed immediately upon
capture._!_2
```

下一步，我们只需要在格式化后的数据上进行数据集的构建即可。

### 5.2.3 数据集预览

同样，在正式介绍如何构建数据集之前我们先通过一张图来了解一下整个构建流程，以便做到心中有数。假如我们现在有两个样本构成了一个 batch，那么其整个数据的处理过程则如图 5-1 所示。

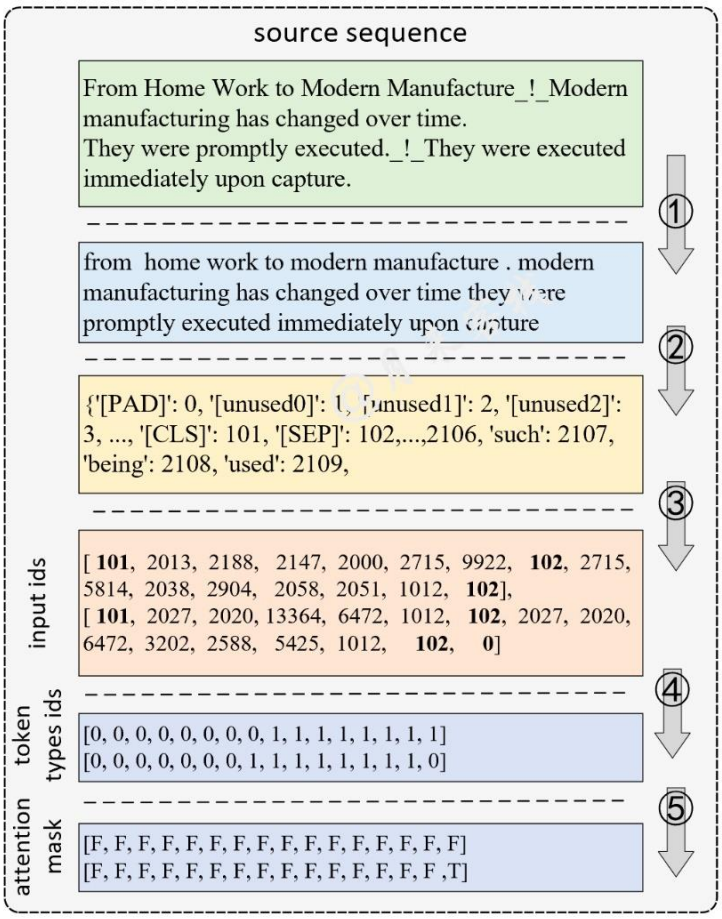



图 5-1. 文本对分类数据集处理流程图

如图 5-1 所示，第 1 步需要将原始的数据样本进行分词（tokenize）处理；第 2 步再根据 tokenize 后的结果构造一个字典，不过在使用 BERT 预训练时并不需要我们自己来构造这个字典，直接使用相应开源模型中的 vocab.txt 文件构造字典即可，因为只有 vocab.txt 中每个字的索引顺序才与开源模型中每个字的 Embedding 向量一一对应的。第 3 步则是根据字典将 tokenize 后的文本序列转换为 Token id 序列，同时在 Token id 序列的起始位置加上[CLS]，在两个序列之间以及整个序列的末尾加上[SEP]符号，并进行 Padding。第 4、5 步则是根据第 3 步处理后的结果分别生成对应的 token types ids 和 attention mask 向量。

最后，在模型训练时只需要将第 3、4 和 5 步处理后的结果一起喂给模型进行训练即可。

### 5.2.4 数据集构建

**第 1 步：定义 tokenize**

第 1 步需要完成的就是将输入进来的文本序列 tokenize 到单词级别，对于英文语料简单来说就是将每个单词和标点符号分开。不过具体的处理方式使用到的是一种叫做 WordPiece 的处理方式。在这里，我们可以借用 transformers 包中的 BertTokenizer 方法来完成，示例如下：

```
1 from transformers import BertTokenizer
2 model_name = '../bert_base_uncased_english'
3 tokenizer = BertTokenizer.from_pretrained(model_name)
4 r = tokenizer.tokenize("From Home Work to Modern Manufacture. Modern
                                        manufacturing has changed over time.")
5 print(r)
6 ['from', 'home', 'work', 'to', 'modern', 'manufacture', '.', 'modern',
                        'manufacturing', 'has', 'changed', 'over', 'time', '.']
```

在上述代码中，第 2-3 行就是根据指定的路径（BERT 预训练模型的路径）来载入一个 tokenize 模型；第 6 行便是 tokenize 后的结果。

同时，对于一些新词 WordPiece 也会将其拆分成合理的部分：

```
1 r = tokenizer.tokenize("huggingface")
2 print(r)
3 ['hugging', '##face']
```

**第 2 步：建立词表**

由于 BERT 预训练模型中已经有了一个给定的词表（vocab.txt），因此我们并不需要根据自己的语料来建立一个词表。当然，也不能够根据自己的语料来建立词表，因为相同的字在我们自己构建的词表中和 vocab.txt 中的索引顺序肯定会不一样，而这就会导致后面根据 token id 取出来的向量是错误的。

进一步，我们只需要将 vocab.txt 中的内容读取进来形成一个词表即可。这部分代码同第 4.2.2 节中的相同，所以掌柜在这里就不在赘述。

接着便可以定义一个方法来实例化一个词表：

```
1 def build_vocab(vocab_path):
2     return Vocab(vocab_path)
3
4 if __name__ == '__main__':
5     vocab = build_vocab()
```

在经过上述代码处理后，我们便能够通过 vocab.itos 得到一个列表，返回词表中的每一个词；通过 vocab.itos[2]返回得到词表中对应索引位置上的词；通过 vocab.stoi 得到一个字典，返回词表中每个词的索引；通过 vocab.stoi['good']返回得到词表中对应词的索引；通过 len(vocab)来返回词表的长度。如下便是建立后的词表：

```
1 {'[PAD]': 0, '[unused0]': 1, '[unused1]': 2, '[unused2]': 3, ..., '[CLS]':
  101, '[SEP]': 102,...,2106, 'such': 2107, 'being': 2108, 'used': 2109,
  'state':2110, 'people': 2111, 'part': 2112, 'know': 2113, 'against': 2114,
  'your': 2115, 'many': 2116, 'second': 2117, 'university': 2118, 'both': 2119,
  'national': 2120, '##er': 2121, 'these': 2122, 'don': 2123, 'known': 2124,
  'off': 2125, 'way': 2126, 'until': 2127, 're': 2128, 'how': 2129, ...}
```

此时，我们就需要定义一个类，并在类的初始化过程中根据训练语料完成字典的构建等工作，代码如下：

```
1 class LoadPairSentenceClassificationDataset(
                                    LoadSingleSentenceClassificationDataset):
2     def __init__(self, kwargs):
3         super(LoadPairSentenceClassificationDataset, self).__init__(kwargs)
4         pass
```

由于在第 4 节中的单文本分类场景中已近实现了数据集构建整个流程的代码，所以我们在这里只需要继承 LoadSingleSentenceClassificationDataset 这个类，然后再重写里面的 data_process()和 generate_batch()方法即可，其它地方不用修改。

**第 3 步：转换为 Token 序列**

在得到构建的字典后，便可以通过如下方法来将训练集、验证集和测试集转换成 Token 序列：

```
 1     def data_process(self, filepath):
 2         raw_iter = open(filepath).readlines()
 3         data = []
 4         max_len = 0
 5         for raw in tqdm(raw_iter, ncols=80):
 6             line = raw.rstrip("\n").split(self.split_sep)
 7             s1, s2, l = line[0], line[1], line[2]
 8             token1 = [self.vocab[token] for token in self.tokenizer(s1)]
 9             token2 = [self.vocab[token] for token in self.tokenizer(s2)]
10             tmp = [self.CLS_IDX] + token1 + [self.SEP_IDX] + token2
11             if len(tmp) > self.max_position_embeddings - 1:
12                 tmp = tmp[:self.max_position_embeddings - 1]
13             tmp += [self.SEP_IDX]
14             seg1 = [0]  (len(token1) + 2)
15             seg2 = [1]  (len(tmp) - len(seg1))
16             segs = torch.tensor(seg1 + seg2, dtype=torch.long)
17             tensor_ = torch.tensor(tmp, dtype=torch.long)
18             l = torch.tensor(int(l), dtype=torch.long)
19             max_len = max(max_len, tensor_.size(0))
20             data.append((tensor_, segs, l))
21         return data, max_len
```

在上述代码中，第 6-7 行便是用来取得文本和标签；第 8-9 行是分别对两个序列 s1 和 s2 转换为词表中对应的 Token；第 10-13 行则是将两个序列拼接起来，并在序列的开始加上[CLS]符号，在两个序列之间及末位加上[SEP]符号；第 14-16 行则是构造得到 Segment Embedding 的输入向量，其中第 14 行里 2 表示[CLS]

和中间的[SEP]这两个字符；第 17-20 行则是整合得到对应的样本数据，包括 Token Embedding 的输入、Segment Embedding 的输入以及每个样本对应的标签。

在处理完成后，第 5.2 节中的 2 个样本将会被转换成如下形式：

```
tensor([[  101,  2013,  2188,  2147,  2000,  2715,  9922,   102,  2715,  5814,
          2038,  2904,  2058,  2051,  1012,   102],
        [  101,  2027,  2020, 13364,  6472,  1012,   102,  2027,  2020,  6472,
          3202,  2588,  5425,  1012,   102,     0]])
torch.Size([2, 16])
tensor([[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0]])
```

从上面的输出结果可以看出，101 就是[CLS]在词表中的索引位置，102 则是[SEP]在词表中的索引；其它非 0 值就是 tokenize 后的文本序列转换成的 Token 序列。同时可以看出，这里的结果是以第 1 个样本的长度 16 对第 2 个样本进行 padding 的，并且 padding 的 Token ID 为 0。可以发现，除了对原始的文本的 Token 序列进行 Padding 外，还需要对 Segment Embedding 的输入进行 Padding。

因此，下面我们就来介绍样本的 padding 处理。

第 4 步：padding 处理与 mask

从第 3 步的输出结果看出，在对原始文本序列 tokenize 转换为 Token id 后还需要对其进行 padding 处理。对于这一处理过程掌柜在前面第 4.2.4 节中已经介绍过了，这里就不再赘述。

进一步，我们需要定义一个方法来对每个 batch 的 Token 序列进行 padding 处理：

```
    def generate_batch(self, data_batch):
        batch_sentence, batch_seg, batch_label = [], [], []
        for (sen, seg, label) in data_batch: # 对一个 batch 中每个样本进行处理。
            batch_sentence.append(sen)
            batch_seg.append((seg))
            batch_label.append(label)
        batch_sentence = pad_sequence(batch_sentence, # [batch_size,max_len]
                                      padding_value=self.PAD_IDX,
                                      batch_first=False,
                                      max_len=self.max_sen_len) #[max_len,batch_size]
        batch_seg = pad_sequence(batch_seg,  # [batch_size,max_len]
                                 padding_value=self.PAD_IDX,
                                 batch_first=False,
                                 max_len=self.max_sen_len)# [max_len,batch_size]
        batch_label = torch.tensor(batch_label, dtype=torch.long)
        return batch_sentence, batch_seg, batch_label
```

上述代码的作用就是对每个 batch 的 Token Embedding 输入序列以及 Segment Embedding 输入进行 padding 处理并同时返回标签。

最后，对于每一序列的 attention_mask 向量，我们只需要判断其是否等于 padding_value 便可以得到这一结果，具体可见下面的使用示例。

**第 5 步：构造 DataLoade 与使用示例**

经过前面 4 步的处理，整个数据集的构建就算是已经基本完成了，只需要再构造一个 DataLoader 迭代器即可。由于这部分代码同第 4.2.4 节中的相同，掌柜这里就不再赘述。

在完成类 LoadPairSentenceClassificationDataset 所有的编码过程后，便可以通过如下形式进行使用：

```
from Tasks.TaskForPairSentenceClassification import ModelConfig
from utils.data_helpers import LoadPairSentenceClassificationDataset
from transformers import BertTokenizer

if __name__ == '__main__':
    config = ModelConfig()
    tokenizer = BertTokenizer.from_pretrained(config.pretrained_model_dir)
    load_dataset = LoadPairSentenceClassificationDataset(
        vocab_path=config.vocab_path,
        tokenizer=tokenizer.tokenize,
        batch_size=config.batch_size,
        max_sen_len=config.max_sen_len,
        split_sep=config.split_sep,
        max_position_embeddings=config.max_position_embeddings,
        pad_index=config.pad_token_id,
        is_sample_shuffle=config.is_sample_shuffle)

    train_iter, test_iter, val_iter = \
        load_dataset.load_train_val_test_data(config.train_file_path,
                                              config.val_file_path,
                                              config.test_file_path)
    for sample, seg, label in train_iter:
        print(sample.shape)  # [seq_len,batch_size]
        print(sample.transpose(0, 1))  # [batch_size,seq_len]
        padding_mask = (sample == load_dataset.PAD_IDX).transpose(0, 1)
        print(padding_mask.shape)
        print(label.shape)
        print(seg.shape)  # [seq_len,batch_size]
        print(label)  # [batch_size,]
        print(seg.transpose(0, 1))
        break
```

执行完上述代码后便可以得到如下所示的结果：

```
torch.Size([16, 2])
tensor([[  101,  2013,  2188,  2147,  2000,  2715,  9922,  102,  2715,  5814,
          2038,  2904,  2058,  2051,  1012,   102],
        [  101,  2027,  2020, 13364,  6472,  1012,  102,  2027,  2020,  6472,
          3202,  2588,  5425,  1012,   102,     0]])
torch.Size([2, 16])
torch.Size([2])
torch.Size([16, 2])
tensor([1, 2])
tensor([[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0]])
```

## 5.3 文本蕴含

### 5.3.1 前向传播

如同掌柜在第 5.1 节内容中介绍的那样，文本对（文本蕴含）分类任务本质上还是一个序列分类的问题，因此这里直接使用 DownstreamTasks 目录中的 BertForSentenceClassification.py 模块即可。同时，由于这个模块的内容在第 4.1.1 节中已经介绍过了，所以掌柜在这里就不再赘述。

### 5.3.2 模型训练

进一步，对于模型的训练过程来说，我们需要在 Tasks 目录（如图 2-5 所示）下新建一个名为 TaskForPairSentenceClassification.py 的模块来完成文本对分类模型的微调训练任务。

首先，我们需要定义一个 ModelConfig 类来对分类模型中的超参数进行管理，代码如下所示：

```
class ModelConfig:
    def __init__(self):
        self.project_dir = os.path.dirname(
                                os.path.dirname(os.path.abspath(__file__)))
        self.dataset_dir = os.path.join(self.project_dir, 'data',
                                        'PairSentenceClassification')
        self.pretrained_model_dir = os.path.join(self.project_dir,
                                        "bert_base_uncased_english")
        self.vocab_path=os.path.join(self.pretrained_model_dir, 'vocab.txt')
        self.device = torch.device('cuda:0'
                                if torch.cuda.is_available() else 'cpu')
        self.train_file_path = os.path.join(self.dataset_dir, 'train.txt')
        self.val_file_path = os.path.join(self.dataset_dir, 'val.txt')
        self.test_file_path = os.path.join(self.dataset_dir, 'test.txt')
```

```
11          self.model_save_dir = os.path.join(self.project_dir, 'cache')
12          self.logs_save_dir = os.path.join(self.project_dir, 'logs')
13          self.split_sep = '_!_'
14          self.is_sample_shuffle = True
15          self.batch_size = 16
16          self.max_sen_len = None
17          self.num_labels = 3
18          self.epochs = 10
19          self.model_val_per_epoch = 2
20          logger_init(log_file_name='pair', log_level=logging.INFO,
21                      log_dir=self.logs_save_dir)
22          if not os.path.exists(self.model_save_dir):
23              os.makedirs(self.model_save_dir)
24
25          config_path = os.path.join(self.pretrained_model_dir, "config.json")
26          bert_config = BertConfig.from_json_file(config_path)
27          for key, value in bert_config.__dict__.items():
28              self.__dict__[key] = value
29          logging.info(" ### 将当前配置打印到日志文件中 ")
30          for key, value in self.__dict__.items():
31              logging.info(f"### {key} = {value}")
```

在上述代码中，第 2-23 行则是分别用来定义模型中的一些数据集目录、超参数和初始化日志打印类等；第 25-28 行则是将原始 bert_base_uncased_english 配置文件，即 config.json 中的参数也导入到类 ModelConfig 中；第 31-33 行则是将所有的超参数配置情况一同打印到日志文件中。

最后，我们只需要再定义一个 train()函数来完成模型的训练即可。由于这部分代码同 4.4.2 节中的大同小异，所以掌柜这里就不再赘述，大家可以直接阅读项目中的源码。

### 5.3.3 模型推理

文本对分类模型中的推理部分同 4.4.3 单文本分类模型中的内容几乎一样，所以掌柜在这里也不再赘述，大家可以直接参考项目源码。

到此，对于第 2 个基于 BERT 预训练模型的文本蕴含任务就介绍完了。在下一节内容中，掌柜将会介绍如何在问题选择任务（即输入一个问题和四个选项让模型选择其中最合理的一个答案）场景下进行 BERT 预训练模型的微调。

# 第 6 节 基于 BERT 预训练模型的 SWAG 选择任务

经过前面几节内容的介绍，相信大家对于 BERT 预训练模型的使用已经有了一定的认识。不过为了满足不同人群的学习需求，在这篇文章中掌柜将会介绍基于 BERT 预训练模型的第 3 个下游任务场景，即如何完成推理问答选择任务。所谓问答选择指的就是同时给模型输入一个问题和若干选项的答案，最后需要模型从给定的选项中选择一个最符合问题逻辑的答案。可问题在于我们应该怎么来构建这个模型呢？

通常来说，在 NLP 领域的很多场景中模型最后所做的基本上都是一个分类任务，虽然表面上看起来不是。例如：文本蕴含任务其实就是将两个序列拼接在一起，然后预测其所属的类别；基于神经网络的序列生成模型（翻译、文本生成等）本质就是预测词表中下一个最有可能出现的词，此时的分类类别就是词表的大小。因此，从本质上来说本文介绍的问答选择任务以及在后面将要介绍的问题回答任务其实都是一个分类任务，而关键的地方就在于如何构建模型的输入和输出。

## 6.1 任务构造原理

正如前面所说，对于问答选择这个任务场景来说其本质上依旧可以归结为分类任务，只是关键在于如何构建这个任务以及整个数据集。对于问答选择这个场景来说，其整体原理如图 6-1 所示。

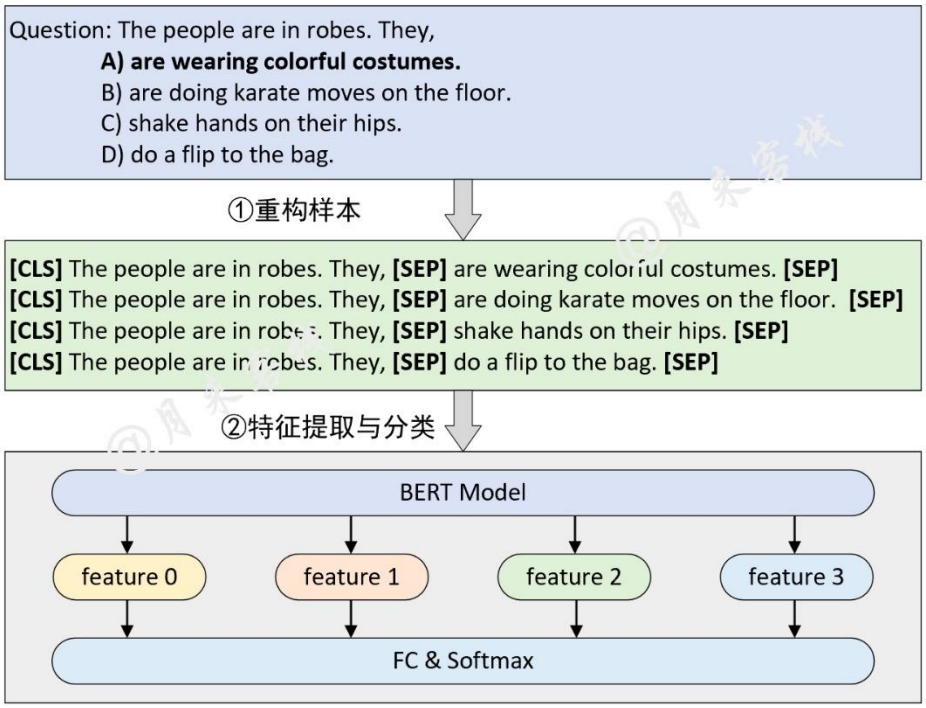



图 6-1. 问答选择原理图

如图 6-1 所示，是一个基于 BERT 预训练模型的四选一问答选择模型的原理图。从图中可以看出，原始数据的形式是一个问题和四个选项，模型需要做的就是从四个选项中给出最合理的一个，于是也就变成了一个四分类任务。同时，构建模型输入的方式就是将原始问题和每一个答案都拼接起来构成一个序列中间用[SEP]符号隔开，然后再分别输入到 BERT 模型中进行特征提取得到四个特征向量形状为[4,hidden_size]，最后再经过一个分类层进行分类处理得到预测选项。值得一提的是，通常情况下这里的四个特征都是直接取每个序列经 BERT 编码后的[CLS]向量。

到此，对于问答选择整个模型的原理我们算是清楚了，下面首先来看如何构造数据集。

## 6.2 数据预处理

### 6.2.1 输入介绍

根据第 6.1 节内容的介绍，对于问答任务来说其接受的输入也分为两个部分：一是由问题和每个选项这两个句子所组成的 Token 序列，并且需要在两个句子的开始位置加上一个[CLS]符号，以及两个句子之间和结尾分别加上一个[SEP]符号；二是 Segment Embedding 部分的输入，用于确定两个句子的所属部分。最后将两者均作为模型的输入即可。

同时，这里需要注意的是，虽然对于 BERT 模型来说“问题+一个选项”构成的序列就是一个样本，但是我们在构造数据集的时候，还是需要将“问题+四个选项”看成一个整体，然后在输入模型之前在变形为对应的形状。

### 6.2.2 语料介绍

在这里，我们使用到的也是论文中所提到的 SWAG（The Situations With Adversarial Generations ）数据集[16] [17]，即给定一个情景（一个问题或一句描述），任务是模型从给定的四个选项中预测最有可能的一个。

如下所示便是部分原始示例数据：

```
1 ,video-id,fold-ind,startphrase,sent1,sent2,gold-source, ending0,ending1,
  ending2,ending3,label
2 0,anetv_NttjvRpSdsI,19391,The people are in robes. They,The people are in
  robes.,They,gold,are wearing colorful costumes.,are doing karate moves on the
  floor.,shake hands on their hips.,do a flip to the bag.,0
3 1,lsmdc3057_ROBIN_HOOD-27684,16344,She smirks at someone and rides off.
  He,She smirks at someone and rides off.,He,gold,smiles and falls
  heavily.,wears a bashful smile.,kneels down behind her.,gives him a playful
  glance.,1
```

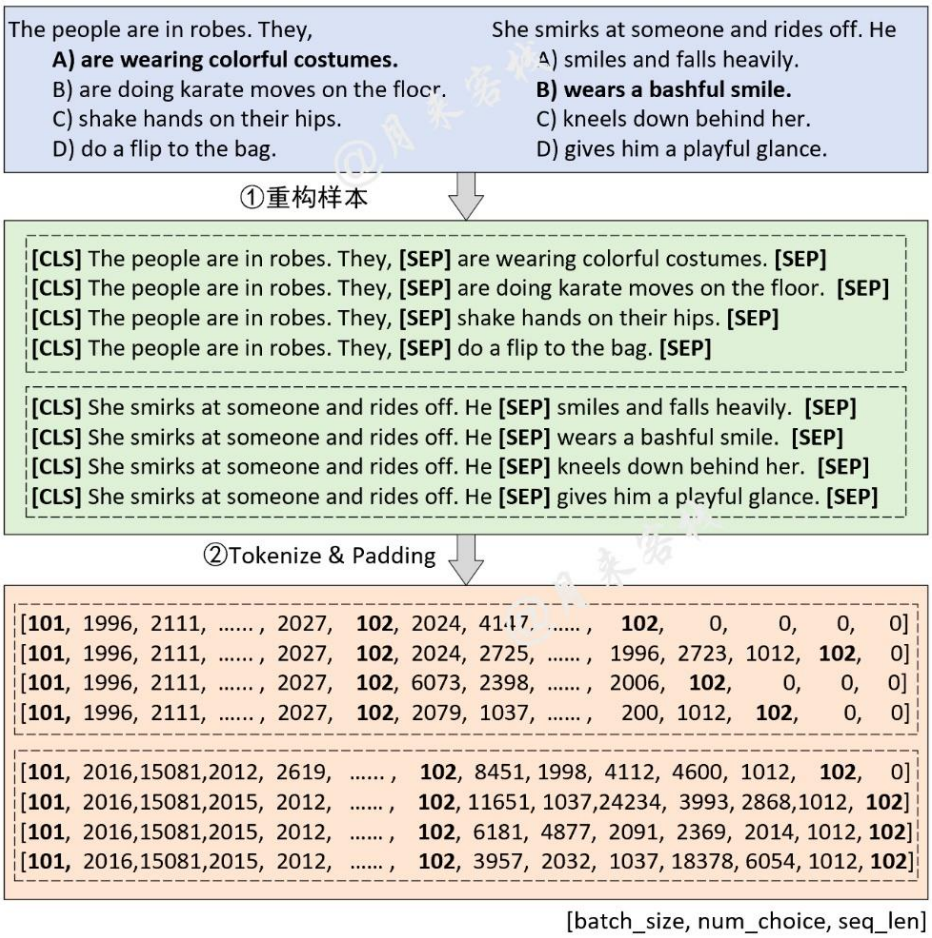
在上述示例中，数据集中一共有 12 个字段（第 1 行）包含两个样本（第 2-3 行），我们这里需要用到的就是 sent1,ending0,ending1,ending2,ending3,label 这 6 个字段。例如对于第一个样本来说，其形式如下：

```
1 The people are in robes. They
2   A) wearing colorful costumes. # 正确选项
3   B) are doing karate moves on the floor.
4   C) shake hands on their hips.
5   D) do a flip to the bag.
```

同时，由于该数据集已经做了训练集、验证集和测试集（没有标签）的划分，所以后续我们也就不需要来手动划分了。

### 6.2.3 数据集预览

同样，在正式介绍如何构建数据集之前我们先通过一张图来了解一下整个大致构建的流程。假如我们现在有两个样本构成了一个 batch，那么其整个数据的处理过程则如图 6-2 所示。



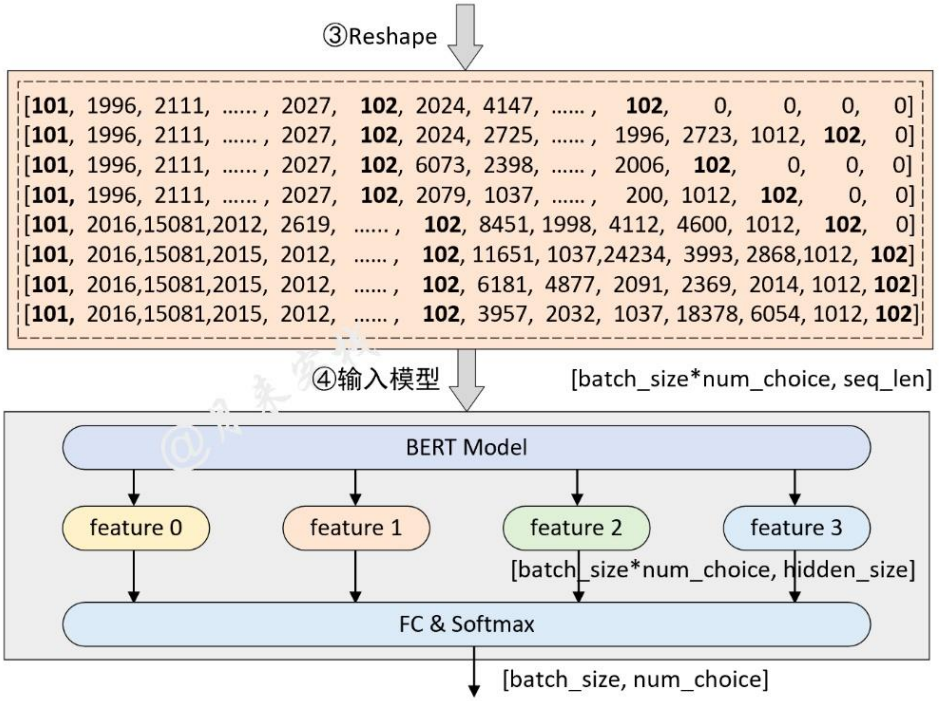



图 6-2. 问答选择数据集构建流程图

如图 6-2 所示，首先对于原始数据的每个样本（一个问题和四个选项），需要将问题同每个选项拼接在一起构造成为 4 个序列并添加上对应的分类符[CLS]和分隔符[SEP]，即图中的第①步重构样本。紧接着需要将第①步构造得到的序列转换得到 Token id 并进行 padding 处理，此时便得到了一个形状为[batch_size, num_choice, seq_len]的 3 维矩阵，即图 6-2 中第 2 步处理完成后形状为[2,4,19]的结果。同时，在第②步中还要根据每个序列构造得到相应的 attention_mask 向量和 token_types_ids 向量（图中未画出），并且两者的形状也是[batch_size, num_choice, seq_len]。

其次是将第②步处理后的结果变形成[batch_sizenum_choice, seq_len]的 2 维形式，因为 BERT 模型接收输入形式便是一个二维的矩阵。在经过 BERT 模型进行特征提取后，将会得到一个形状为[batch_sizenum_choice, hidden_size]的 2 维矩阵，最后再乘上一个形状为[hidden_size,1]的矩阵并变形成[batch_size, num_choice]即可完成整个分类任务。

## 6.2.4 数据集构造

在说完数据集构造的整理思路后，下面我们就来正式编码实现整个数据集的构造过程。同样，对于数据预处理部分我们可以继续继承之前文本分类处理的这个类 LoadSingleSentenceClassificationDataset，然后再稍微修改其中的部分方法即可。同时，由于在前两个示例[3] [4]中已经就 tokenize 和词表构建等内容做了详细的介绍，所以这部分内容就不再赘述。

**第 1 步：重构样本和 Tokenize**

如图 6-2 处理流程所示，需要对原始样本进行重构以及转换得到每个序列对应的 Token id，下面首先是在 data_process()方法中来定义如何读取原始数据：

```
1 class LoadMultipleChoiceDataset(LoadSingleSentenceClassificationDataset):
2     def __init__(self, num_choice=4, kwargs):
3         super(LoadMultipleChoiceDataset, self).__init__(kwargs)
4         self.num_choice = num_choice
5 
6     def data_process(self, filepath):
7         data = pd.read_csv(filepath)
8         questions = data['startphrase']
9         answers0, answers1 = data['ending0'], data['ending1']
10        answers2, answers3 = data['ending2'], data['ending3']
11        labels = [-1]  len(questions)
12        if 'label' in data:
13            labels = data['label']
14        all_data = []
15        max_len = 0
16        for i in tqdm(range(len(questions)), ncols=80):
17            t_q = [self.vocab[token] for token in
                                              self.tokenizer(questions[i])]
18            t_q = [self.CLS_IDX] + t_q + [self.SEP_IDX]
19            t_a0 = [self.vocab[token] for token in
                                              self.tokenizer(answers0[i])]
20            t_a1 = [self.vocab[token] for token in
                                              self.tokenizer(answers1[i])]
21            t_a2 = [self.vocab[token] for token in
                                              self.tokenizer(answers2[i])]
22            t_a3 = [self.vocab[token] for token in
                                              self.tokenizer(answers3[i])]
23            max_len = max(max_len,
                    len(t_q) + max(len(t_a0), len(t_a1), len(t_a2), len(t_a3)))
24            seg_q = [0]  len(t_q)
25            seg_a0 = [1]  (len(t_a0) + 1)
26            seg_a1 = [1]  (len(t_a1) + 1)
27            seg_a2 = [1]  (len(t_a2) + 1)
28            seg_a3 = [1]  (len(t_a3) + 1)
29            all_data.append((t_q, t_a0, t_a1, t_a2, t_a3, seg_q,
30                             seg_a0, seg_a1, seg_a2, seg_a3, labels[i]))
31        return all_data, max_len
```

在上述代码中，第 1-4 行用于继承之前的 LoadSingleSentenceClassificationDataset 类以及添加一个新的参数 num_choice 也就是分类数。第 7-10 行则是根据文件路径来读取原始数据并按对应字段取得问题和答案。第 11-13 行则是用来判断

是否存在正确标签，因为测试集中不含有标签。第 15 行 max_len 则是用来保存数据集中最长序列的长度。第 16 行为一个循环用来遍历每一个问题以及对应的答案；第 17-18 行是将原始问题根据词表转换为对应的 Token id，同时在起止位置分别加上[CLS]和[SEP]符号；第 19-23 行是分别将每个问题对应的 4 个选项转换为对应的 Token id，以及保存最大序列的长度；第 24-28 行是用来构造对应的 token_type_ids 向量；第 29-30 行是分别将每一个问题以及对应的 4 个选项处理后的结果保存和返回最后的结果。

注意，这里还没有将每个问题同对应的 4 个选项进行拼接。

**第 2 步：拼接与 padding**

在处理得到每个问题及对应选项的 Token ids 和 token_type_ids 后，再定义一个 generate_batch()方法对每个 batch 中的数据拼接和 padding 处理，代码如下：

```
1     def generate_batch(self, data_batch):
2         batch_qa, batch_seg, batch_label = [], [], []
3
4         def get_seq(q, a):
5             seq = q + a
6             if len(seq) > self.max_position_embeddings - 1:
7                 seq = seq[:self.max_position_embeddings - 1]
8             return torch.tensor(seq + [self.SEP_IDX], dtype=torch.long)
9
10        for item in data_batch:
11            tmp_qa = [get_seq(item[0], item[1]),get_seq(item[0], item[2]),
12                      get_seq(item[0], item[3]),get_seq(item[0], item[4])]
13            tmp_seg = [torch.tensor(item[5] + item[6], dtype=torch.long),
14                       torch.tensor(item[5] + item[7], dtype=torch.long),
15                       torch.tensor(item[5] + item[8], dtype=torch.long),
16                       torch.tensor(item[5] + item[9], dtype=torch.long)]
17            batch_qa.extend(tmp_qa)
18            batch_seg.extend(tmp_seg)
19            batch_label.append(item[-1])
20        batch_qa = pad_sequence(batch_qa, padding_value=self.PAD_IDX,
21                                batch_first=True,max_len=self.max_sen_len)
22        batch_mask = (batch_qa == self.PAD_IDX).view(
23                                 [-1, self.num_choice, batch_qa.size(-1)])
24        batch_qa = batch_qa.view([-1, self.num_choice, batch_qa.size(-1)])
25        batch_seg = pad_sequence(batch_seg,padding_value=self.PAD_IDX,
26                                 batch_first=True,max_len=self.max_sen_len)
27        batch_seg = batch_seg.view([-1,self.num_choice, batch_seg.size(-1)])
28        batch_label = torch.tensor(batch_label, dtype=torch.long)
29        return batch_qa, batch_seg, batch_mask, batch_label
```

在上述代码中，第 4-8 行中的 get_seq()方法是用于根据传入的问题 Token id 和答案 Token id 拼接得到一个完整的 Token id，并将超过长度的部分进行截取处理。第 11-12 行是将每个问题分别与其对应的 4 个选项进行拼接。第 13-16 行是分别构造得到每个问题与其对应的 4 个选项所形成的 token_type_ids 向量。第 17-19 行是保存每个 Batch 所有样本处理好的结果。第 20-28 行是对各个输入进行 padding 或者变形等以得到对应形状的输入，最后处理结束后 batch_qa、batch_seq 和 batch_mask 的维度均为[batch_size, num_choice, src_len]，batch_label 的形状为[batch_size,]。

**第 3 步：使用示例**

在完成上述两个步骤之后，整个数据集的构建就算是已经基本完成了，可以通过如下代码进行数据集的载入：

```
from utils.data_helpers import LoadMultipleChoiceDataset
from Tasks.TaskForMultipleChoice import ModelConfig
from transformers import BertTokenizer

if __name__ == '__main__':
    config = ModelConfig()
    tokenizer = BertTokenizer.from_pretrained(config.pretrained_model_dir)
    load_dataset = LoadMultipleChoiceDataset(vocab_path=config.vocab_path,
                tokenizer=tokenizer.tokenize,batch_size=2,max_sen_len=None,
           max_position_embeddings=512,pad_index=0,is_sample_shuffle=False,
                        num_choice=model_config.num_labels)
    train_iter, test_iter, val_iter = \
        load_dataset.load_train_val_test_data(model_config.train_file_path,
                                              model_config.val_file_path,
                                              model_config.test_file_path)
    for qa, seg, mask, label in test_iter:
        print(" ### input ids:")
        print(qa.shape)  # [batch_size,num_choice, max_len]
        print(qa[0])
        print(" ### attention mask:")
        print(mask.shape)  # [batch_size,num_choice, max_len]
        print(mask[0])
        print(" ### token type ids:")
        print(seg.shape)  # [batch_size,num_choice, max_len]
        print(seg[0])
        print(label.shape)  # [batch_size]
        trans_to_words(qa[0], load_dataset.vocab.itos)
        break
```

上述代码运行结束后将会看到类似如下所示的结果：

```
### input ids: torch.Size([2, 4, 22])
tensor([[101,  1996,  2111,  2024,  1999, 17925,  1012,  2027,   102,  2024,
         4147, 14231, 12703,  1012,   102,     0,     0,     0,     0,     0,
            0,     0],
        [101,  1996,  2111,  2024,  1999, 17925,  1012,  2027,   102,  2024,
         2725, 16894,  5829,  2006,  1996,  2723,  1012,   102,     0,     0,
            0,     0],
        [101,  1996,  2111,  2024,  1999, 17925,  1012,  2027,   102,  6073,
         2398,  2006,  2037,  6700,  1012,   102,     0,     0,     0,     0,
            0,     0],
        [101,  1996,  2111,  2024,  1999, 17925,  1012,  2027,   102,  2079,
         1037, 11238,  2000,  1996,  4524,  1012,   102,     0,     0,     0,
            0,     0]])
### attention mask: torch.Size([2, 4, 22])
tensor([[False, False,.....,True,  True,  True,  True,  True, True,  True],
        [False, False,.....,False, False, False, True,  True, True,  True],
        [False, False,.....,False,  True,  True, True,  True, True,  True],
        [False, False,.....,False, False,  True, True,  True, True, True]])
### token type ids: torch.Size([2, 4, 22])
tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]])
torch.Size([2]) Question and Answer:
[CLS] the people are in robes . they [SEP] are wearing colorful costumes .
    [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[CLS] the people are in robes . they [SEP] are doing karate moves on the
    floor . [SEP] [PAD] [PAD] [PAD] [PAD]
[CLS] the people are in robes . they [SEP] shake hands on their hips . [SEP]
    [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[CLS] the people are in robes . they [SEP] do a flip to the bag . [SEP]
    [PAD] [PAD] [PAD] [PAD] [PAD]
```

在上述结果中，其中第 29-32 行为根据 Token id 再转换为字符串后的结果。

到此，对于整个数据集的构建过程就介绍完了，下面掌柜开始继续介绍问答选择模型的实现内容。

## 6.3 选择任务

### 6.3.1 前向传播

正如第 6.2 节内容所介绍，我们只需要在原始 BERT 模型的基础上再加一个分类层即可，因此这部分代码相对来说也比较容易理解。如图 2-4 所示，在

BertForMultipleChoice.py 文件中，首先需要定义一个类以及相应的初始化函数，如下：

```
1 from ..BasicBert.Bert import BertModel
2 import torch.nn as nn
3
4 class BertForMultipleChoice(nn.Module):
5     def __init__(self, config, bert_pretrained_model_dir=None):
6         super(BertForMultipleChoice, self).__init__()
7         self.num_choice = config.num_labels
8         if bert_pretrained_model_dir is not None:
9             self.bert = BertModel.from_pretrained(config,
                                                    bert_pretrained_model_dir)
10        else:
11            self.bert = BertModel(config)
12        self.dropout = nn.Dropout(config.hidden_dropout_prob)
13        self.classifier = nn.Linear(config.hidden_size, 1)
```

在上述代码中，第 8-11 行便是根据相应的条件返回一个 BERT 模型，第 13 行则是定义了一个分类层。

然后再是定义完成整个前向传播过程，代码如下：

```
1     def forward(self, input_ids,
2                 attention_mask=None,
3                 token_type_ids=None,
4                 position_ids=None,
5                 labels=None):
6         flat_input_ids=input_ids.view(-1,input_ids.size(-1)).transpose(0, 1)
7         flat_token_type_ids=token_type_ids.view(-1,token_type_ids.size(-
                                                  1)).transpose(0, 1)
8         flat_attention_mask=attention_mask.view(-1,token_type_ids.size(-1))
9         pooled_output, _ = self.bert(
10            flat_input_ids, # [src_len,batch_size*num_choice]
11            flat_attention_mask, # [batch_size*num_choice,src_len]
12            flat_token_type_ids, # [src_len,batch_size*num_choice]
13            position_ids=position_ids)
14        pooled_output = self.dropout(pooled_output)
15        logits = self.classifier(pooled_output) # [batch_size*num_choice, 1]
16        shaped_logits = logits.view(-1, self.num_choice)
17        if labels is not None:
18            loss_fct = nn.CrossEntropyLoss()
19            loss = loss_fct(shaped_logits, labels.view(-1))
20            return loss, shaped_logits
21        else:
22            return shaped_logits
```

在上述代码中，第 6-8 行用于将 3 维的输入变成 2 维的输入（也就是图 6-2 中的第③步），这是因为 BERT 所接收的输入形式便是两个维度。同时根据需要还将 src_len 这个维度放到了最前面。第 9-13 行则是通过原始的 BERT 模型提取得到每个序列（指的是每个问题和其中一个选项所构成的序列，即图 6-2 中第③步后的每一行）的特征表示，输出形状为[batch_size*num_choice, hidden_size]。第 15-16 行是先进行分类处理，然后再变形得到每个问题所对应预测选项的 logits 值，最后输出形状为[batch_size, num_choice]。第 17-22 行是根据相应的判断条件返回损失或者 logits 值。

### 6.3.2 模型训练

首先，如图 2-5 所示在 Tasks 目录下新建一个名为 TaskForMultipleChoice.py 的模块，然后定义一个 ModelConfig 类来对分类模型中的超参数以及其它变量进行管理。部分代码如下所示：

```
1  class ModelConfig:
2      def __init__(self):
3          self.project_dir =
                   os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
4          self.dataset_dir =
                     os.path.join(self.project_dir, 'data', 'MultipleChoice')
5          self.pretrained_model_dir =
                   os.path.join(self.project_dir, "bert_base_uncased_english")
6          self.vocab_path=os.path.join(self.pretrained_model_dir, 'vocab.txt')
7          self.device = torch.device('cuda:0'
                                      if torch.cuda.is_available() else 'cpu')
8          self.train_file_path = os.path.join(self.dataset_dir, 'train.csv')
9          self.val_file_path = os.path.join(self.dataset_dir, 'val.csv')
10         self.test_file_path = os.path.join(self.dataset_dir, 'test.csv')
11         self.model_save_dir = os.path.join(self.project_dir, 'cache')
12         self.logs_save_dir = os.path.join(self.project_dir, 'logs')
13         self.is_sample_shuffle = True
14         self.batch_size = 16
15         self.max_sen_len = None
16         self.num_labels = 4  # num_choice
17         self.learning_rate = 2e-5
18         self.epochs = 10
19         self.model_val_per_epoch = 2
20         logger_init(log_file_name='choice', log_level=logging.INFO,
21                                                 log_dir=self.logs_save_dir)
```

在上述代码中，第 3-12 行分别用来获取各个文件的所在路径；第 13-19 行则是设置模型对应的超参数；第 20-31 行是初始化日志打印的相关信息。

同时，为了展示训练时的预测结果，这里我们需要写一个函数来进行格式化：

```
def show_result(qas, y_pred, itos=None, num_show=5):
    count = 0
    num_samples, num_choice, seq_len = qas.size()
    qas = qas.reshape(-1)
    strs = np.array([itos[t] for t in qas]).reshape(-1, seq_len)
    for i in range(num_samples):
        s_idx = i  num_choice
        e_idx = s_idx + num_choice
        sample = strs[s_idx:e_idx]
        if count == num_show:
            return
        count += 1
        for j, item in enumerate(sample):  # 每个样本的四个答案
            q, a, _ = " ".join(item[1:]).replace(" .", ".").replace(" ##",
                                                                    "").split('[SEP]')
            if y_pred[i] == j:
                a += " ## True"
            else:
                a += " ## False"
            logging.info(f"[{count + 1}/{num_show}] ### {q + a}")
        logging.info("\n")
```

在上述函数调用结束可以输出类似如下所示的结果：

```
- the people are in robes. they are wearing colorful costumes. ## False
- the people are in robes. they are doing karate moves on the floor. ## True
- the people are in robes. they shake hands on their hips. ## False
- the people are in robes. they do a flip to the bag. ## False
```

最后，我们便可以通过如下方法完成整个模型的微调，关键代码如下所示：

```
def train(config):
    model = BertForMultipleChoice(config,config.pretrained_model_dir)
    ......
    optimizer=torch.optim.Adam(model.parameters(), lr=config.learning_rate)
    model.train()
    tokenizer = BertTokenizer.from_pretrained(config.pretrained_model_dir)
    data_loader = LoadMultipleChoiceDataset(
         vocab_path=config.vocab_path,tokenizer=tokenizer.tokenize,
         batch_size=config.batch_size,max_sen_len=config.max_sen_len,
         max_position_embeddings=config.max_position_embeddings,
         pad_index=config.pad_token_id,
         is_sample_shuffle=config.is_sample_shuffle,
         num_choice=config.num_labels)
    train_iter, test_iter, val_iter = \
        data_loader.load_train_val_test_data(config.train_file_path,
```

```
15                          config.val_file_path,config.test_file_path)
16      max_acc = 0
17      for epoch in range(config.epochs):
18          losses = 0
19          start_time = time.time()
20          for idx, (qa, seg, mask, label) in enumerate(train_iter):
21              loss, logits = model(input_ids=qa,attention_mask=mask,
22                                   token_type_ids=seg,position_ids=None,
23                                   labels=label)
24              optimizer.zero_grad()
25              loss.backward()
26              optimizer.step()
27              losses += loss.item()
28              acc = (logits.argmax(1) == label).float().mean()
29              if idx % 10 == 0:
30                  logging.info(
                        f"Epoch:{epoch}, Batch[{idx}/{len(train_iter)}],"
31                      f"Train loss :{loss.item():.3f}, Train acc: {acc:.3f}")
32              if idx % 100 == 0:
33                  y_pred = logits.argmax(1).cpu()
34                  show_result(qa, y_pred, data_loader.vocab.itos, num_show=1)
35          end_time = time.time()
36          train_loss = losses / len(train_iter)
37          logging.info(f"Epoch: {epoch}, Train loss: "
38          f"{train_loss:.3f}, Epoch time = {(end_time - start_time):.3f}s")
39          if (epoch + 1) % config.model_val_per_epoch == 0:
40              acc, _ = evaluate(val_iter,model,config.device, inference=False)
41              logging.info(f"Accuracy on val {acc:.3f}")
```

在上述代码中，第 2 行是根据指定预训练模型的路径初始化一个基于 BERT 的问答任务模型；第 7-15 行是载入相应的数据集；第 17-36 行则是整个模型的训练过程，完整示例代码可在项目仓库中进行获取。

如下便是网络的训练时的输出结果：

```
1 - INFO: Epoch: 0, Batch[0/4597], Train loss :1.433, Train acc: 0.250
2 - INFO: Epoch: 0, Batch[10/4597], Train loss :1.277, Train acc: 0.438
3 - INFO: Epoch: 0, Batch[20/4597], Train loss :1.249, Train acc: 0.438
4         ......
5 - INFO: Epoch: 0, Batch[4590/4597], Train loss :0.489, Train acc: 0.875
6 - INFO: Epoch: 0, Batch loss :0.786, Epoch time = 1546.173s
7 - INFO: Epoch: 0, Batch[0/4597], Train loss :1.433, Train acc: 0.250
8 - INFO: He is throwing darts at a wall. A woman, squats alongside flies side
to side with his gun.   ## False
```

```
 9 - INFO: He is throwing darts at a wall. A woman, throws a dart at a
dartboard.    ## False
10 - INFO: He is throwing darts at a wall. A woman, collapses and falls to the
floor.    ## False
11 - INFO: He is throwing darts at a wall. A woman, is standing next to him.
## True
12 - INFO: Accuracy on val 0.794
```

### 6.3.3 模型推理

在完成模型的训练过程后，便可以将训练过程中保存好的模型用于任务的推理场景中。具体代码实现如下：

```
1 def inference(config):
2     model = BertForMultipleChoice(config,
3                                   config.pretrained_model_dir)
4     model_save_path = os.path.join(config.model_save_dir, 'model.pt')
5     if os.path.exists(model_save_path):
6         loaded_paras = torch.load(model_save_path)
7         model.load_state_dict(loaded_paras)
8         logging.info("## 成功载入已有模型，进行预测.....")
9     model = model.to(config.device)
10    tokenizer = BertTokenizer.from_pretrained(config.pretrained_model_dir)
11    data_loader = LoadMultipleChoiceDataset(vocab_path=config.vocab_path,
12                tokenizer=tokenizer.tokenize,batch_size=config.batch_size,
13              max_sen_len=config.max_sen_len,pad_index=config.pad_token_id,
14                max_position_embeddings=config.max_position_embeddings,
15                is_sample_shuffle=config.is_sample_shuffle)
16    test_iter = data_loader.load_train_val_test_data(
                                    config.test_file_path,only_test=True)
17    y_pred = evaluate(test_iter, model, config.device, inference=True)
18    logging.info(f"预测标签为：{y_pred.tolist()}")
```

在上述代码中，第 2-3 行是用来实例化一个问答任务模型；第 4-8 行是载入本地模型参数来重新初始化网络中的参数；第 11-16 行是载入新的测试数据；第 17 行是返回模型在测试集上的准确率。

到此，对于整个基于 BERT 预训练模型的 SWAG 数据集的问答模型就介绍完了。总的来讲，对于问答选择这一任务场景来说，只需要将每个问题与其对应的各个选项看成两个拼接在一起的序列，再输入到 BERT 模型中进行特征提取最后进行分类即可。

# 第 7 节 自定义学习率动态调整

## 7.1 引言

在深度学习模型训练过程中，当训练出的模型效果不那么尽如人意时，相信大家第一时间想到的策略就是动态调整学习率。或者是在模型搭建之初就想好后面要通过动态调整学习率来训练模型。之所以要在这里介绍这部分内容是因为在下一个 BERT 微调任务场景（问题回答）中论文作者就用到了学习了动态调整的方式。同时，例如在 Transformer 论文中，也采用了如下公式来动态调整学习率：

$$lr\_rate=d_{model}^{-0.5}\cdot\min(step\_num^{-0.5},step\_num\cdot warmup\_steps^{-1.5}) \quad (7.1)$$

根据公式(7.1)可知，模型训练过程中学习率的变化过程将类似图 7-1 所示。

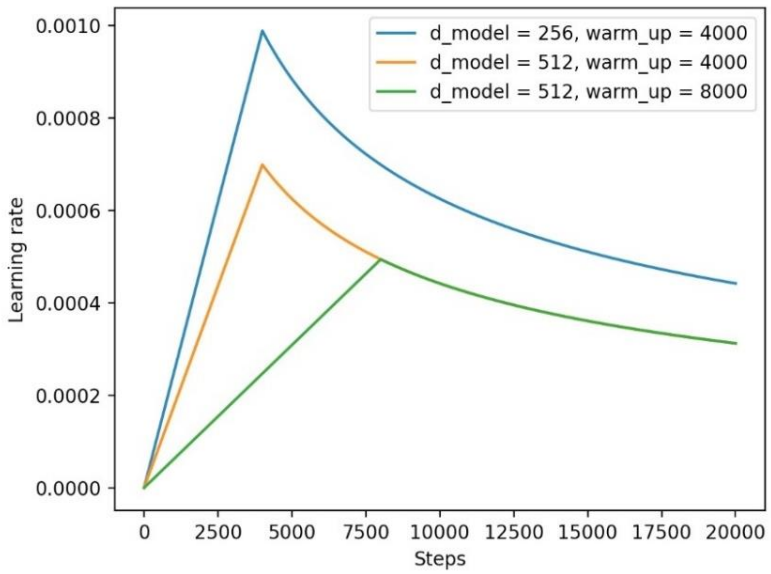



图 7-1. Transformer 动态学习率变化图

由此可见，动态学习了调整是深度学习中一种场景的训练策略。同时，对于公式(7.1)所示的学习率调整策略，我们还可以通过如下代码来实现模型学习率在训练过程中的动态调整，如下所示：

```
class CustomSchedule(object):
    def __init__(self, d_model, warmup_steps=4000, optimizer=None):
        super(CustomSchedule, self).__init__()
        self.d_model = torch.tensor(d_model, dtype=torch.float32)
        self.warmup_steps = warmup_steps
        self.steps = 1.
        self.optimizer = optimizer

    def step(self):
```

```
10          arg1 = self.steps  -0.5
11          arg2 = self.steps  (self.warmup_steps  -1.5)
12          self.steps += 1.
13          lr = (self.d_model  -0.5)  min(arg1, arg2)
14          for p in self.optimizer.param_groups:
15              p['lr'] = lr
16          return lr
17
18 def train(config):
19      optimizer = torch.optim.Adam(translation_model.parameters(),lr=0.,
20                      betas=(config.beta1, config.beta2),eps=config.epsilon)
21      lr_scheduler = CustomSchedule(config.d_model, optimizer=optimizer)
22      loss.backward()
23      lr_scheduler.step()
24      optimizer.step()
25      ...
```

在上述代码中，第 1-16 行是整个自定义学习率的实现部分，其中 warmup_steps 表示学习率在达到最大值前的一个“热身步数”（例如图 7-1 中的直线部分）；第 23 行则是在每个训练的 step 中对学习率进行更新；第 24 行则是采用更新后的学习率对模型参数进行更新。

当然，对于这类复杂或并不常见的学习率动态调整方式确实需要手动来实现，但是对于一些常见的常数、线性、余弦变换等学习率调整，我们可以直接借助 Transformers 框架中的 optimization 模块来实现。同时，即使是不常见的调整策略我们也可以借助 PyTorch 中相关模块来实现，以便更好的利用相关内置方法。

在接下来的内容中，掌柜将会先来介绍如何直接使用 Transformers 框架中的 optimization 模块来快速实现学习率动态调整的目的；然后再来简单介绍一下各个方法背后的实现逻辑以及如何模仿来实现自定义的方法。

## 7.2 动态学习率使用

在 Transformers 框架中，我们可以通过如下方式来导入 optimization 模块：

```
1 from transformers import optimization
```

在 optimization 模块中，一共包含了 6 种常见的学习率动态调整方式，包括 constant、constant_with_warmup、linear、polynomial、cosine 和 cosine_with_restarts，其分别通过一个函数来返回对应的实例化对象。

下面掌柜就开始依次对这 6 种动态学习率调整方式进行介绍。

### 7.2.1 constant 使用

在 optimization 模块中可以通过 get_constant_schedule 函数来返回对应的常数动态学习率调整方法。顾名思义，常数学习率动态调整就是学习率是一个恒定不

变的常数，也就是说相当于没用。为了方便后续对学习率的变化进行可视化，这里我们先随便定义一个网络模型，代码如下：

```
1 import torch
2 import torch.nn as nn
3
4 class Model(nn.Module):
5     def __init__(self):
6         super(Model, self).__init__()
7         self.fc = nn.Linear(5, 10)
8
9     def forward(self, x):
10        out = self.fc(x).sum()
11        return out
```

进一步，在模型训练的过程中可以通过以下方式来进行使用：

```
1 from transformers import optimization
2
3 if __name__ == '__main__':
4     x = torch.rand([8, 5])
5     model = Model()
6     model.train()
7     steps = 1000
8     optimizer = torch.optim.Adam(model.parameters(), lr=1.0)
9     scheduler = optimization.get_constant_schedule(optimizer, last_epoch=-1)
10    lrs = []
11    for _ in range(steps):
12        loss = model(x)
13        optimizer.zero_grad()
14        loss.backward()
15        optimizer.step()
16        scheduler.step()
17        lrs.append(scheduler.get_last_lr())
```

在上述代码中，第 9 行便是用来得到对应的常数学习率变化的实例化对象，其中 last_epoch 用于在恢复训练时指定上次结束时的 epoch 数量，因为有些方法学习率的变化会与 epoch 数有关，如果不考虑模型恢复的话指定为-1 即可，这部分内容掌柜将在本节最后进行详细介绍；第 16 行则是对学习率进行更新；第 17 行则是取出对应的学习率便于可视化。

在模型训练结束后（或训练过程中采用 tensorboard 工具）便可以对学习率的变化过程进行可视化了，代码如下：

```
1     plt.figure(figsize=(7, 4))
2     plt.plot(range(steps), lrs, label=name)
```

```
3     plt.legend(fontsize=13)
4     plt.show()
```

上述方法的可视化结果如图 7-2 所示。

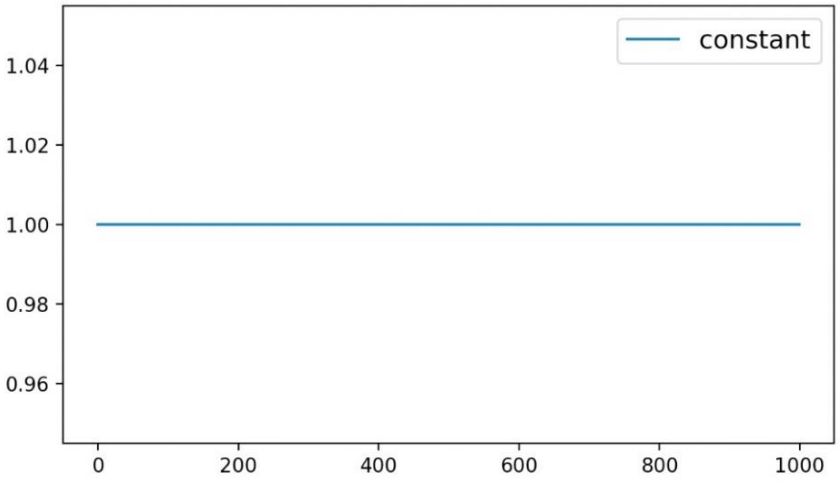



图 7-2. constant 学习率变化图

如图 7-2 所示，模型在整个训练过程中的学习率并没有发生变化，都是保持着 1.0 的初始值。

### 7.2.2 constant_with_warmup 使用

在 optimization 模块中，我们可以通过 get_constant_schedule_with_warmup 函数来返回得到对应的动态学习率调整实例化方法。从名字可以看出来，该方法最终得到的是一个带 warmup 的常数学习率变化过程。在模型训练的过程中，我们可以通过以下方式来进行使用：

```
1 scheduler = optimization.get_constant_schedule_with_warmup(
2                     optimizer, num_warmup_steps=300)
```

其中 num_warmup_steps 表示 warmpup 的数量。

最后，该方法的可视化结果如图 7-3 所示.

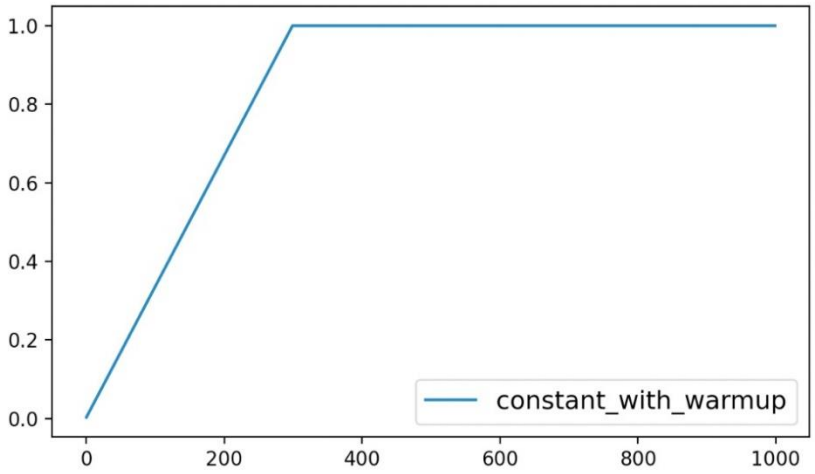



图 7-3. constant_with_warmup 学习率变化图

从图 7-3 可以看出 constant_with_warmup 仅仅只是在最初的 300 个 steps 中以线性的方式进行增长，之后便是同样保持为常数。

### 7.2.3 linear 使用

在 optimization 模块中可以通过 get_linear_schedule_with_warmup 函数来返回对应的动态学习率调整的实例化方法。从名字可以看出，该方法最终得到的是一个带 warmup 的常数学习率变化。在模型训练的过程中，我们可以通过以下方式来进行使用：

```
1 scheduler = optimization.get_linear_schedule_with_warmup(optimizer,
2                         num_warmup_steps=300, num_training_steps=steps)
```

其中 num_training_steps 表示整个模型训练的 step 数。

最后，该方法的可视化结果如下所示：

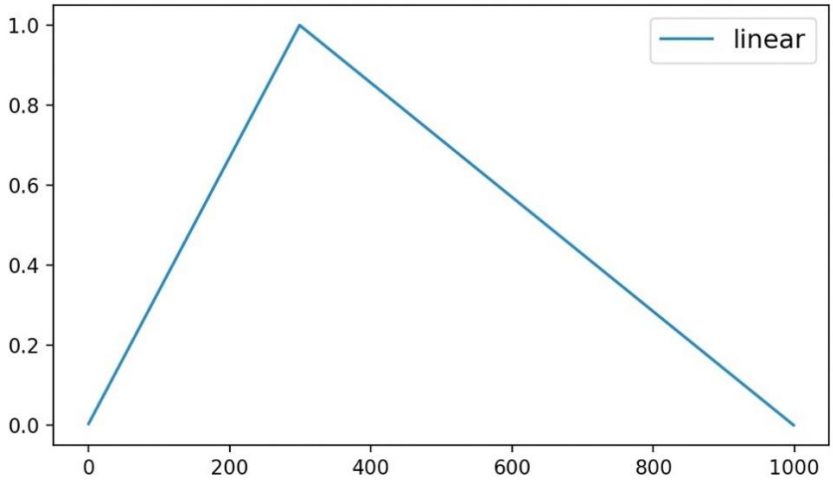



图 7-4. linear 学习率变化图

从图 7-4 可以看出 linear 动态学习率调整先是在最初的 300 个 steps 中以线性的方式进行增长，之后便是同样以线性的方式进行递减，直到衰减到 0 为止。

### 7.2.4 polynomial 使用

在 optimization 模块中可以通过 get_polynomial_decay_schedule_with_warmup 函数来返回对应的动态学习率调整的实例化方法。从名字可以看出，该方法最终得到的是一个基于多项式的学习率动态调整策略。在模型训练的过程中，我们可以通过以下方式来进行使用：

```
1 scheduler = optimization.get_polynomial_decay_schedule_with_warmup(optimizer,
2         num_warmup_steps=300,num_training_steps=steps,lr_end = 1e-7,power=3)
```

其中 power 表示多项式的次数，当 power=1 时（默认）类似于 get_linear_schedule_with_warmup 函数；lr_end 表示学习率衰减到的最小值。

最后，该方法的可视化结果如图 7-5 所示。

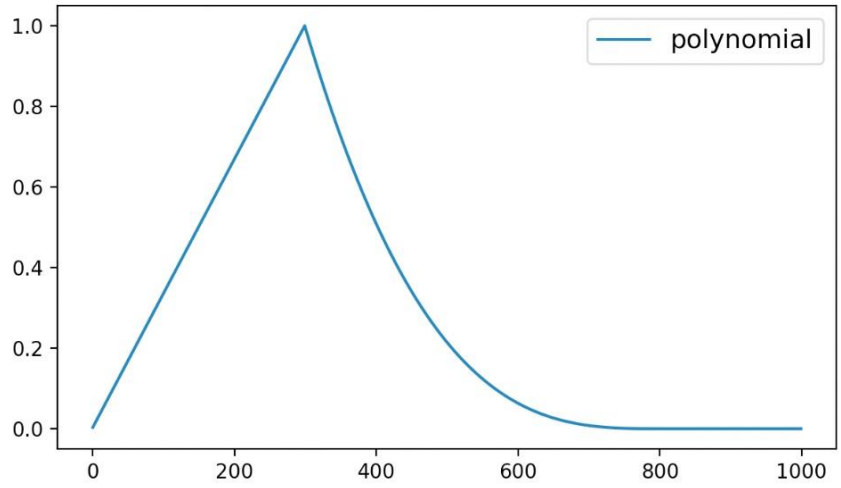



图 7-5. polynomial 学习率变化图（power=3）

从图 7-5 可以看出 polynomial 动态学习率调整先是在最初的 300 个 steps 中以线性的方式进行增长，之后便是多项式的方式进行递减，直到衰减到 lr_end 后保持不变。

### 7.2.5 cosine 使用

在 optimization 模块中可以通过 get_cosine_schedule_with_warmup 来返回基于 cosine 函数的动态学习率调整方法。在模型训练过程中我们可以通过如下方式来进行调用：

```
1 scheduler = optimization.get_cosine_schedule_with_warmup(optimizer,
2             num_warmup_steps=300,num_training_steps=steps,num_cycles=2)
```

其中 num_cycles 表示循环的次数。

最后，该方法的可视化结果如图 7-6 所示。

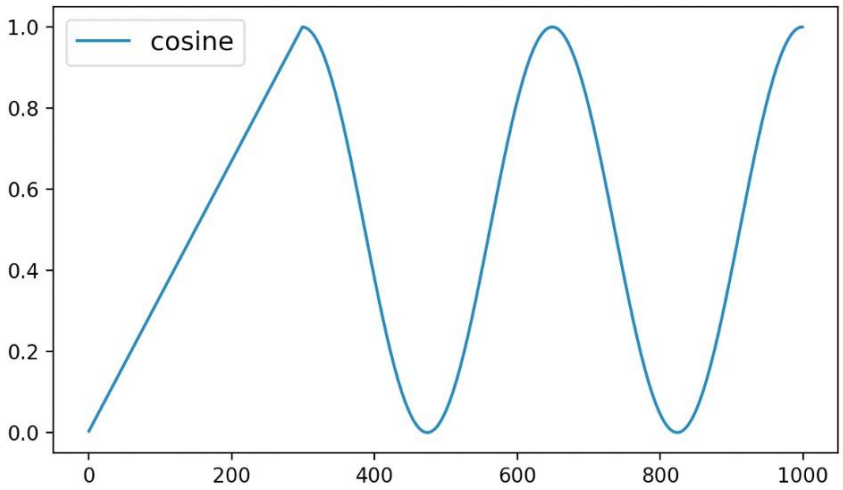



图 7-6. cosine 学习率变化图（num_cycles=2）

从图 7-6 可以看出 cosine 动态学习率调整方法先是在最初的 300 个 steps 中以线性的方式进行增长，之后便是以余弦函数的方式进行周期性变换。

### 7.2.6 cosine_with_restarts 使用

在 optimization 模块中可以通过 get_cosine_with_hard_restarts_schedule_with_warmup 来返回基于 cosine 函数的硬重启动态学习率调整方法。所谓硬重启就是学习率衰减到 0 之后直接变回到最大值的方式。在模型训练过程中我们可以通过如下方式来进行调用：

```
1 scheduler = optimization.get_cosine_with_hard_restarts_schedule_with_warmup(
        optimizer, num_warmup_steps=300,num_training_steps=steps,num_cycles=2)
```

最后，该方法的可视化结果如图 7-7 所示。

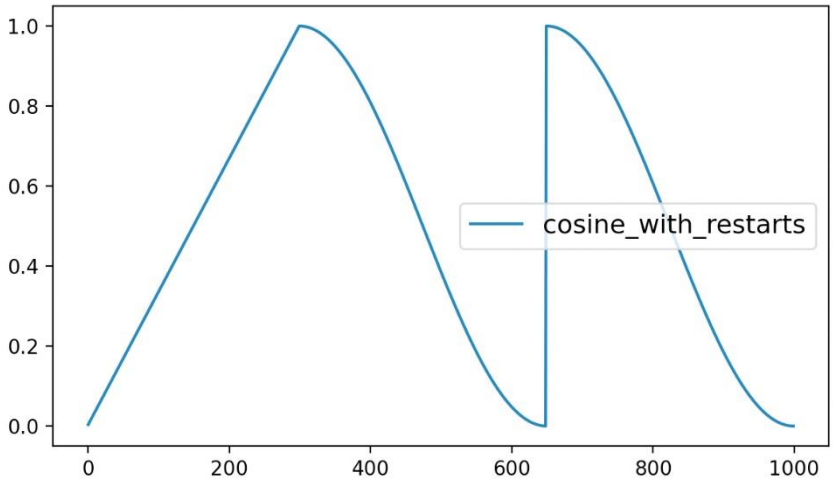



图 7-7. cosine_with_restarts 学习率变化图（num_cycles=2）

从图 7-7 可以看出 cosine_with_restarts 动态学习率调整方法先是在最初的 300 个 steps 中以线性的方式进行增长，之后便是以余弦函数的方式进行周期性衰减，当达到最小值时再直接恢复到初始学习率。

### 7.2.7 get_scheduler 使用

通过上述 6 个函数便能够返回得到相应的动态学习率调整方法。当然，如果你并不需要修改一些特定的参数，例如多项式中的 power 和余弦变换中的 num_cycles 等，那么还可以使用一个更加简单的统一接口来调用上述 6 个方法：

```
1 from transformers import get_scheduler
2 def get_scheduler(
3     name: Union[str, SchedulerType],
4     optimizer: Optimizer,
5     num_warmup_steps: Optional[int] = None,
6     num_training_steps: Optional[int] = None):
```

在上述代码中，第 3 行 name 表示指定学习率调整的方式，可选项就是上面介绍的 6 种，并且通过 constant、constant_with_warmup、linear、polynomial、cosine 和 cosine_with_restarts 这 6 个关键字就能够返回得到对应的方法；而对于其它特

定的参数则会保持每个方法对应的默认值。例如通过 get_scheduler 函数返回 get_cosine_with_hard_restarts_schedule_with_warmup 时，num_cycles 则为 1。

例如：

```
scheduler = optimization.get_cosine_with_hard_restarts_schedule_with_warmup(
        optimizer,num_warmup_steps=300,num_training_steps=steps,num_cycles=1)
scheduler = get_scheduler(name="cosine_with_restarts", optimizer=optimizer,
                          num_warmup_steps=300, num_training_steps=steps)
```

在上述代码中，两种方式返回得到的学习调整方式都是一样的；但是如果想要返回 num_cycles=2 的情况那就不能通过 get_scheduler 函数获得。

到此，对于 Transformes 框架中常见的 6 种学习率动态调整方法及其使用示例就介绍完了。

## 7.3 动态学习率实现

对于 Transformers 框架中实现的这 6 种学习率动态调整方法本质上也是基于 PyTorch 框架中的 LambdaLR 类而来。

```
from torch.optim.lr_scheduler import LambdaLR
class LambdaLR(_LRScheduler):
        def __init__(self, optimizer, lr_lambda, last_epoch=-1):
        pass
```

通过这个接口，我们只需要指定优化器、学习率系数的计算方式（函数）以及 last_epoch 参数来实例化类 LambdaLR 便可以返回得到相应的实例化对象。下面掌柜就来依次进行一个简单的介绍。

### 7.3.1 constant 实现

对于 constant 的计算过程来说比较简单， 只需要传入一个返回值始终为 1.0 的匿名函数即可。因为返回的 1 将会作为一个系数乘以我们初始设定的学习率。实现代码如下：

```
def get_constant_schedule(Optimizer, last_epoch = -1):
    return LambdaLR(optimizer, lambda _: 1, last_epoch=last_epoch)
```

在上述代码中，lambda _:1 就是对应返回值为 1 的匿名函数，也就是无论传入和值，它的返回值都是 1.

### 7.3.2 constant_with_warmup 实现

对于 constant_with_warmup 的计算过程来说同样也比较简单。整体逻辑便是在 num_warmup_steps 之前系数保持线性增长，在 num_warmup_steps 之后保持为 1.0 不变即可，即：

$$\text{lr\_coef}=\begin{cases}\dfrac{\text{current\_step}}{\text{num\_warmup\_steps}}, & \text{current\_step}<\text{num\_warmup\_steps}\\ 1.0, & \text{current\_step}\geq\text{num\_warmup\_steps}\end{cases} \tag{7.2}$$

根据公式(7.2)，最终实现代码如下所示：

```
def get_constant_schedule_with_warmup(Optimizer, num_warmup_steps,
                                                                last_epoch = -1):
    def lr_lambda(current_step):
        if current_step < num_warmup_steps:
            return float(current_step) / float(max(1.0, num_warmup_steps))
        return 1.0

    return LambdaLR(optimizer, lr_lambda, last_epoch=last_epoch)
```

这里掌柜需要再次提醒大家的是，lr_lambda()返回的是学习率的变换系数，该系数乘以初始的学习率才是最终模型用到的学习率。例如上述代码中当current_step 大于等于 num_warmup_steps 时返回的系数就是 1，这样就能保证在这之后学习率就会保持初始设定的学习率不变。

### 7.3.3 linear 实现

对于 linear 的系数计算过程来说只需要分别在 num_warmup_steps 之前和之后分别保持线性增加和线性减少即可，即：

$$\text{lr\_coef}=\begin{cases}\dfrac{\text{current\_step}}{\text{num\_warmup\_steps}}, & \text{current\_step}<\text{num\_warmup\_steps}\\ \dfrac{\text{num\_training\_steps-current\_step}}{\text{num\_training\_steps-num\_warmup\_steps}}, & \text{current\_step}\geq\text{num\_warmup\_steps}\end{cases} \tag{7.3}$$

根据公式(7.3)，最终实现代码如下所示：

```
def get_linear_schedule_with_warmup(optimizer, num_warmup_steps,
                                    num_training_steps, last_epoch=-1):
    def lr_lambda(current_step: int):
        if current_step < num_warmup_steps:
            return float(current_step) / float(max(1, num_warmup_steps))
        return max( 0.0, float(num_training_steps - current_step) /
          float(max(1, num_training_steps - num_warmup_steps)))

    return LambdaLR(optimizer, lr_lambda, last_epoch)
```

### 7.3.4 polynomial 实现

对于 polynomial 的系数计算过程来说则稍微复杂了一点，其整体逻辑便是在 num_warmup_steps 之前系数保持线性增长，在 num_warmup_steps 之后保持为定值不变，在两者之间则以对应的多项式函数进行变换，计算公式式如下：

$$\text{lr\_coef}=\begin{cases}\dfrac{\text{current\_step}}{\text{num\_warmup\_steps}} \\ \dfrac{\text{lr\_end}}{\text{lr\_init}} \\ \dfrac{(\text{lr\_init}-\text{lr\_end})\cdot\left[1-\dfrac{\text{current\_step}-\text{num\_warmup\_steps}}{\text{num\_training\_steps}-\text{num\_warmup\_steps}}\right]^{\text{power}}+\text{lr\_end}}{\text{lr\_init}}\end{cases} \tag{7.4}$$

$$\begin{cases}\text{current\_step}<\text{num\_warmup\_steps} \\ \text{current\_step}>\text{num\_training\_steps} \\ \text{num\_warmup\_steps}\le\text{current\_step}\le\text{num\_training\_steps}\end{cases}$$

其中 lr_init 表示初始设定的学习率。

根据公式(7.4)，最终实现代码如下所示：

```
def get_polynomial_decay_schedule_with_warmup(optimizer,
            num_warmup_steps, num_training_steps, lr_end=1e-7,
            power=1.0, last_epoch=-1):
    lr_init = optimizer.defaults["lr"]
    assert lr_init > lr_end, f"lr_end ({lr_end}) must be
                                    smaller than initial lr ({lr_init})"
    def lr_lambda(current_step):
        if current_step < num_warmup_steps:
            return float(current_step) / float(max(1, num_warmup_steps))
        elif current_step > num_training_steps:
            return lr_end / lr_init
        else:
            lr_range = lr_init - lr_end
            decay_steps = num_training_steps - num_warmup_steps
            pct_remaining = 1-(current_step-num_warmup_steps) / decay_steps
            decay = lr_range  pct_remaining  power + lr_end
            return decay / lr_init

    return LambdaLR(optimizer, lr_lambda, last_epoch)
```

### 7.3.5 cosine 实现

对于 cosine 学习率动态变换的系数计算过程来说就稍微更复杂了，其整体逻辑便是在 num_warmup_steps 之前系数保持线性增长，在 num_warmup_steps 之后则以对应的余弦函数进行变换，计算公式如下：

$$\text{lr\_coef}=\begin{cases}\dfrac{\text{current\_step}}{\text{num\_warmup\_steps}} \\ \dfrac{1}{2}\cdot\left(1+\cos\left(2\cdot\pi\cdot\text{num\_cycles}\cdot\dfrac{\text{current\_step-num\_warmup\_steps}}{\text{num\_training\_steps-num\_warmup\_steps}}\right)\right)\end{cases}\begin{matrix}\text{current\_step}<\text{num\_warmup\_steps} \\ \text{current\_step}\geq\text{num\_warmup\_steps}\end{matrix}\tag{7.5}$$

根据公式(7.5)，最终实现代码如下所示：

```
def get_cosine_schedule_with_warmup(optimizer, num_warmup_steps,
                        num_training_steps, num_cycles = 0.5, last_epoch = -1):
    def lr_lambda(current_step):
        if current_step < num_warmup_steps:
            return float(current_step) / float(max(1, num_warmup_steps))
        progress = float(current_step - num_warmup_steps) /
                           float(max(1, num_training_steps - num_warmup_steps))
        return max(0.0, 0.5  (1.0 + math.cos(math.pi  float(num_cycles)  2.0
                                                    progress)))
    return LambdaLR(optimizer, lr_lambda, last_epoch)
```

### 7.3.6 cosine_with_restarts 实现

对于 cosine_with_restarts 学习率动态变换的系数计算过程来说，总体上与 cosine 方式的实现过程类似，仅仅只是多增加了一个条件判断，具体计算公式如下：

$$\text{lr\_coef}=\begin{cases}\dfrac{\text{current\_step}}{\text{num\_warmup\_steps}} \\ 0.0 \\ \dfrac{1}{2}\cdot\left(1+\cos\left(\pi\cdot\left(\text{num\_cycles}\cdot\dfrac{\text{current\_step-num\_warmup\_steps}}{\text{num\_training\_steps-num\_warmup\_steps}}\right)\%1.0\right)\right)\end{cases}\begin{matrix}\text{current\_step}<\text{num\_warmup\_steps} \\ \text{current\_step}>\text{num\_training\_steps} \\ \text{num\_warmup\_steps}\leq\text{current\_step}\leq\text{num\_training\_steps}\end{matrix}\tag{7.6}$$

其中%表示取余。

根据公式(7.6)，最终实现代码如下所示：

```
def get_cosine_with_hard_restarts_schedule_with_warmup(optimizer,
        num_warmup_steps,num_training_steps, num_cycles = 1,last_epoch = -1):
    def lr_lambda(current_step):
        if current_step < num_warmup_steps:
            return float(current_step) / float(max(1, num_warmup_steps))
        progress = float(current_step - num_warmup_steps) /
                          float(max(1, num_training_steps - num_warmup_steps))
        if progress >= 1.0:
```

```
            return 0.0
        return max(0.0, 0.5 * (1.0 + math.cos(math.pi
                               * ((float(num_cycles) * progress) % 1.0))))
    return LambdaLR(optimizer, lr_lambda, last_epoch)
```

### 7.3.7 transfromer 实现

经过上述几种动态学习率调整方法实现的介绍，对于公式(7.1)也就是 Transformer 论文中学习率的调整，我们也可以模仿上述的方式来进行实现：

```
def get_customized_schedule_with_warmup(optimizer, num_warmup_steps,
                                        d_model=1.0, last_epoch=-1):
    def lr_lambda(current_step):
        current_step += 1
        arg1 = current_step ** -0.5
        arg2 = current_step * (num_warmup_steps ** -1.5)
        return (d_model ** -0.5) * min(arg1, arg2)

    return LambdaLR(optimizer, lr_lambda, last_epoch)
```

由于公式(7.1)中计算学习率的方法并不涉及到初始学习率的，所以在后面初始化 Adam()时参数 lr 需要赋值为 1.0，这样 get_customized_schedule_with_warmup 返回后的结果就直接是我们需要的学习率了。当然，也可以直接在上述代码第 6 行的返回值中再加上除以初始学习率，这样后续就不用有学习率必须设置为 1 的限制了，大家理解便是。

进一步我们就可以通过上述类似方式来使用该方法：

```
optimizer = torch.optim.Adam(model.parameters(), lr=1.0)
scheduler = get_customized_schedule_with_warmup(optimizer,
                                num_warmup_steps=200,d_model=728)
```

最终同样会得到如图 7-8 所示的学习率变化曲线。

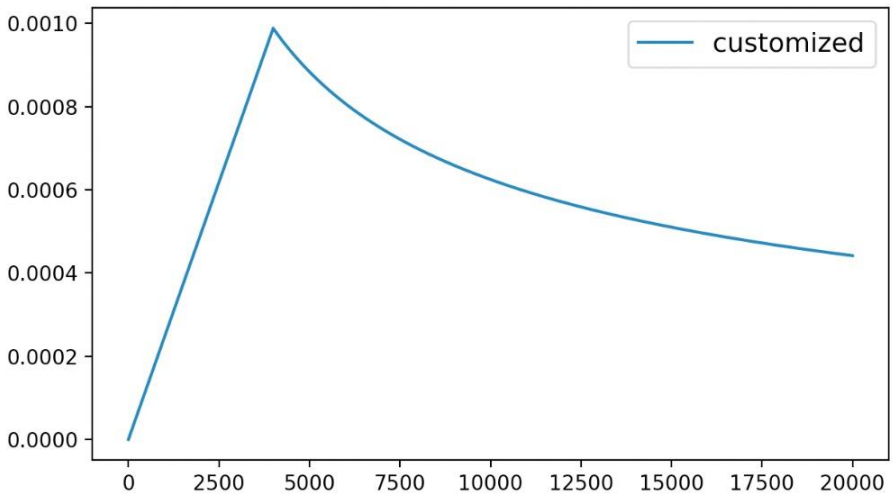



图 7-8. 自定义学习率动态调整图

## 7.4 LambdaLR 原理

在介绍完上述几种动态学习率调整及自定义的用法后，我们再来大致看看底层 LambdaLR 的实现逻辑，这样更有利于我们灵活的使用上述方法。当然，如果有客官暂时只想停留在对上述 6 种方式的使用层面，那么后续内容可以先行略过，等有需要再来查阅。

### 7.4.1 实现逻辑

翻阅 LambdaLR 类的实现代码可以发现，类 LambdaLR 是继承自类 _LRScheduler，两者之中各自重要的类方法和类成员变量如下：

```
class _LRScheduler(object):

    def __init__(self, optimizer, last_epoch=-1):
        if last_epoch == -1:
            for group in optimizer.param_groups:
                group.setdefault('initial_lr', group['lr'])
        ......
        self.base_lrs = list(map(lambda group: group['initial_lr'],
                                                    optimizer.param_groups))
        self.last_epoch = last_epoch
                ......
        self.step()

    def get_lr(self):
        raise NotImplementedError

    def step(self, epoch=None):
                ......
        self._step_count += 1
        with _enable_get_lr_call(self):
            if epoch is None:
                self.last_epoch += 1
                values = self.get_lr()
            else:
                self.last_epoch = epoch
                if hasattr(self, "_get_closed_form_lr"):
                    values = self._get_closed_form_lr()
                else:
                    values = self.get_lr()
        for param_group, lr in zip(self.optimizer.param_groups, values):
            param_group['lr'] = lr
        ......
```

```
32 ......
33 class LambdaLR(_LRScheduler):
34
35     def __init__(self, optimizer, lr_lambda, last_epoch=-1):
36         self.optimizer = optimizer
37         self.last_epoch = last_epoch
38         super(LambdaLR, self).__init__(optimizer, last_epoch)
39
40     def get_lr(self):
41         return [base_lr  lmbda(self.last_epoch)
42                 for lmbda, base_lr in zip(self.lr_lambdas, self.base_lrs)]
```

注：上述代码并非完整部分，掌柜只是对其中的关键部分进行摘取。

要理解整个动态学习率的计算过程最重要的就是弄清楚 get_lr()和 step()这两个方法。从第 7.3 节中的使用示例可以发现，模型在训练过程中是通过 step()这个方法来实现学习率更新的，因此这里我们就从 step()方法入手来进行研究。

从上述代码第 16 行可以发现，其实 step()方法在调用时还会接受一个 epoch 参数，但我们在前面的使用过程中并没有传入，那它又有什么用呢？进一步，从第 20-22 行可以当 epoch 为 None 时，那么 self.last_epoch 就会累计加 1；而如果 epoch 不为 None 那么 self.last_epoch 就会直接取 epoch 的值；接着便是通过 self.get_lr()函数来获取当前的学习率。在得到当前学习率的计算结果后，再通过第 29-30 行代码将其传入到优化器中便实现了学习率的动态调整。

接着再来看 LambdaLR 中 get_lr()部分的实现代码。从第 40-42 行代码可知，self.lr_lambdas 就是 LambdaLR 实例化时传入的参数 lr_lambda，也就是第 7.3 节中介绍的学习率系数的计算函数；而 self.last_epoch 就是前面对应的 current_step 参数。从这里我们就可以发现，LambdaLR 中 epoch 这个概念不仅仅有我们平常训练时所说的迭代“轮”数，也可以理解成训练时参数更新的次数。

从第 20-28 行的逻辑可以看出，如果在使用过程中需要学习率在每个 batch 参数更新时都发生变化，那么最简单的做法就是调用 step()方法时不指定 epoch；如果仅仅是需要在每个 epoch（轮）后学习率才发生变化，那么在调用 step()方法时指定 epoch 为当前的轮数即可，例如：

```
1 for epoch in epoches:
2     for data in data_iter:
3         optimizer.step()
4         scheduler.step(epoch=epoch)
```

通常来说，前一种方式（在每 batch 参数更新后学习率都发生改变）用到的时候更多，也就是第 7.3 节中介绍到的示例。

同时，根据上述代码第 3-8 行可知，当 last_epoch=-1 时，_LRScheduler 就默认当前为模型刚开始训练时的状态，并把 optimizer 中的 lr 参数作为初始学习率 initial_lr，也就是后续的 self.base_lrs，就被用于在第 42 行中计算当前的学习率。

当 last_epoch 不为-1 时，也就意味着此时是的模型可能需要恢复到之前的某个时刻继续进行训练，那么此时学习率也就需要恢复到之前结束的那一刻。

到此，对于类 LambdaLR 的实现逻辑就算是基本介绍完了。下面掌柜再来介绍最后一个示例，即如何通过指定 last_epoch 来恢复到学习率之前的状态继续进行追加训练。

### 7.4.2 学习率恢复

假如某位客官正在采用 cosine 方法作为学习率动态调整策略来训练模型，并且在训练 3 个 epoch 后便结束了训练。同时也得到了如图 7-9 所示的学习率变化曲线。

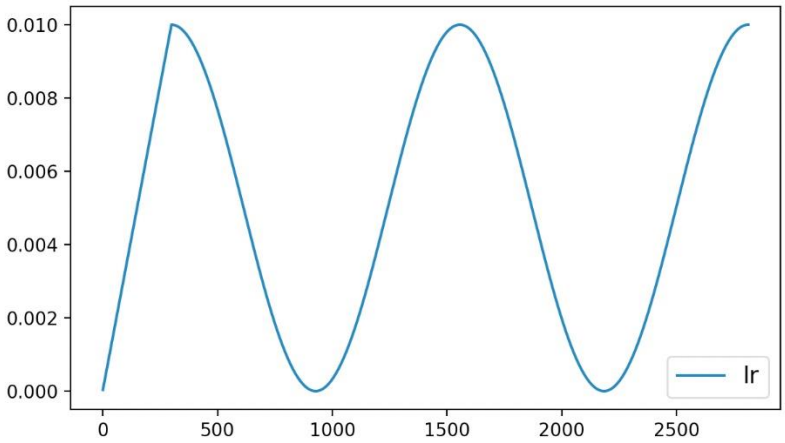



图 7-9. 学习率变化曲线

在这位客官认真分析完训练产生的相关数据后认为，模型如果继续进行训练应该还能获得更好的结果于是就打算对之前保存的模型进行追加训练。但是学习率要怎么样才能恢复到之前结束时的状态呢？也就是说模型在进行追加训练时学习率应该接着之前的状态继续进行，而不是像图 7-9 那样又从头开始。

此时，我们便可以通过如下代码来实现上述目的：

```
        last_epoch = -1
        if os.path.exists('./model.pt'):
            checkpoint = torch.load('./model.pt')
            last_epoch = checkpoint['last_epoch']
            self.model.load_state_dict(checkpoint['model_state_dict'])

        num_training_steps = len(train_iter)  self.epochs
        optimizer = torch.optim.Adam([{"params": self.model.parameters(),
                                       "initial_lr": self.learning_rate}])
        scheduler = get_cosine_schedule_with_warmup(optimizer,
                                    num_warmup_steps=300,
                                    num_training_steps=num_training_steps,
                                    num_cycles=2, last_epoch=last_epoch)
        for epoch in range(self.epochs):
```

```
14              for i, (x, y) in enumerate(train_iter):
15                  loss, logits = self.model(x, y)
16                  optimizer.zero_grad()
17                  loss.backward()
18                  optimizer.step()
19                  scheduler.step()
20                  lrs.append(scheduler.get_last_lr())
21          ......
22          torch.save({'last_epoch': scheduler.last_epoch,
23                      'model_state_dict': self.model.state_dict()},
24                     './model.pt')
```

在上述代码中，第 2-5 行用来判断本地是否存在模型，如果存在则获取对应的参数值；第 7-12 行则分别用来定义和实例化相关方法，当本地不存在模型时 last_epoch 将作为-1 被传递到 get_cosine_schedule_with_warmup 中，即此时学习率从头开始变换；第 22-24 行则是对训练结束后的模型参数进行保存，同时也保存了 last_epoch 的值。

这里需要注意一点的是，只要在优化器中指定了 initial_lr 参数，那么 LambdaLR 在动态计算学习率时的 base_lr 就是 initial_lr 对应的值，与优化器中的指定的 lr 参数也就没有了关系。

当后续再对模型进行追加训练时，第 4 行代码便获取得到了 last_epoch 上一次训练结束后的值，接着后续训练时学习率就可以接着上一次结束时的状态继续进行。最终我们也可以得到如图 7-10 所示的学习率变化曲线。

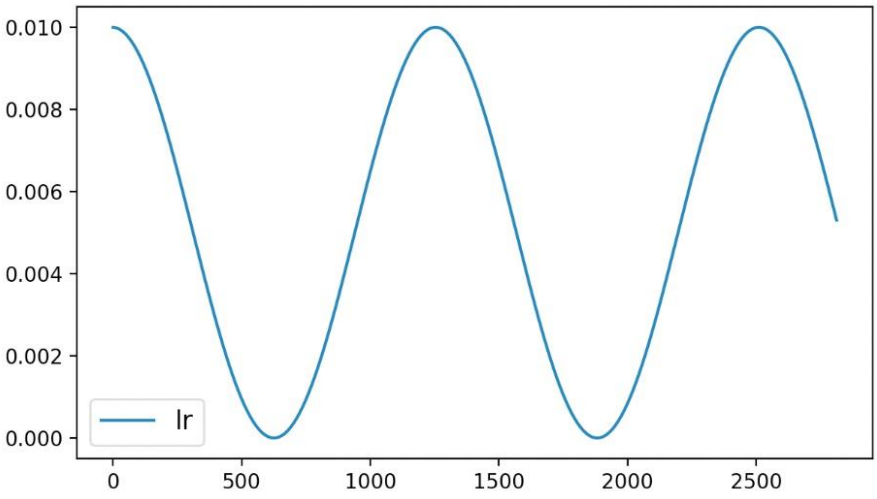



图 7-10. 学习率恢复曲线

从图 7-10 可以看出，学习率的初始值就是接着图 7-9 中学习率的结束值开始进行的变换。

到此，对于动态学习率的使用方法掌柜就介绍完毕了。在下一节内容中，掌柜将会详细介绍如何在问题回答任务（即输入一段文本描述和一个问题，让模型给出答案在文本中的起止位置）场景下进行 BERT 预训练模型的微调。

# 第 8 节 基于 BERT 预训练模型的 SQuAD 问答任务

经过前面几节内容的介绍，我们已经清楚了 BERT 模型的基本原理、如何从零实现 BERT 模型、如何基于 BERT 预训练模型来完成文本分类任务、文本蕴含任务以及问答选择任务这些下游微调任务。在接下来的这节内容中，掌柜将会继续介绍基于 BERT 预训练模型的第 4 个下游任务微调场景，即如何完成问题回答任务。

所谓问题回答指的就是同时给模型输入一个问题和一段描述，最后需要模型从给定的描述中预测出问题答案所在的位置（text span)。

例如：

描述：苏轼是北宋著名的文学家与政治家，眉州眉山人。

问题：苏轼是哪里人？

标签：眉州眉山人

对于这样一个问题问答任务我们应该怎么来构建这个模型呢？

在做这个任务之前首先需要明白的就是：①最终问题的答案一定是在给定的描述中；②问题再描述中的答案一定是一段连续的字符，不能有间隔。例如对于上面的描述内容来说，如果给出的问题是“苏轼生活在什么年代以及他是哪里人？”，那么模型最终并不会给出类似“北宋”和“眉州眉山人”这两个分离的答案，最好的情况下便是给出“北宋著名的文学家与政治家，眉州眉山人”这一个连续的答案。

在有了这两个限制条件后，对于这类问答任务的本质也就变成了需要让模型预测得到答案在描述中的起始位置（start position）以及它的结束位置（end position）。所以，问题最终又变成了如何在 BERT 模型的基础上再构建一个分类器来对 BERT 最后一层输出的每个 Token 进行分类，判断它们是否属于 start position 或者是 end position。

## 8.1 任务构造原理

正如上面所说，尽管对于看似复杂的问题回答任务场景来说，其本质上依旧可以归结为一个普普通的分类任务，只是解决这个问题的关键在于如何构建这个任务以及整个数据集。

如图 8-1 所示，是一个基于 BERT 预训练模型的问题回答模型的原理图。从图中可以看出，构建模型输入的方式就是将原始问题和上下文描述拼接成一个序列中间用[SEP]符号隔开，然后再分别输入到 BERT 模型中进行特征提取。在 BERT 编码完成后，再取最后一层的输出对每个 Token 进行分类即可得到 start position 和 end position 的预测输出。

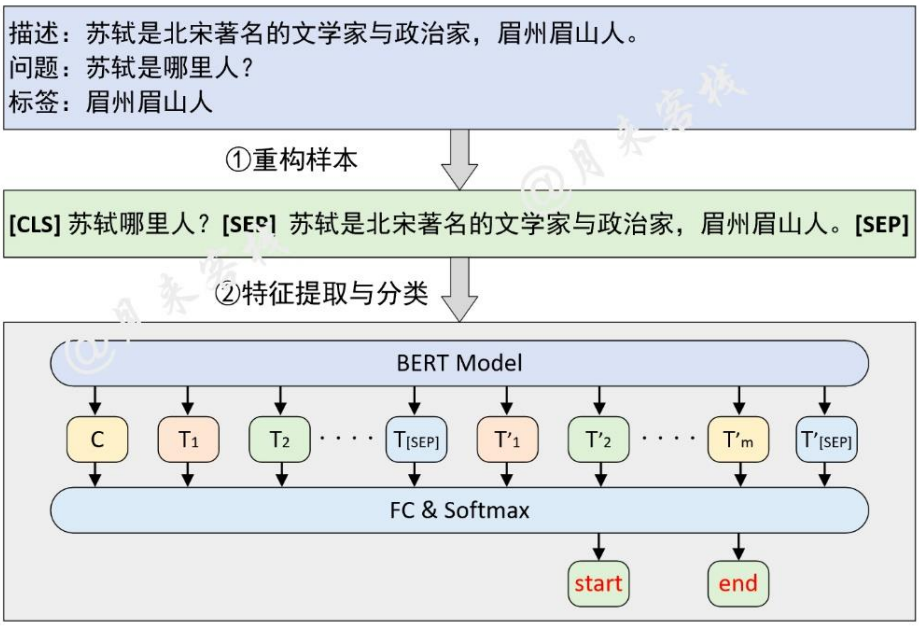



图 8-1. 问题回答原理图

值得注意的是在问题回答场景中是将问题放在上下文描述前面的，即 Sentence A 为问题，Sentence B 为描述[1]。而在上一个问题选择任务场景中是将答案放在描述之后。 掌柜猜测这是因为在 SWAG 这一推理数据集中，每个选项其实都可以看作是问题（描述）的下半句，两者具有强烈的先后顺序算是一种逻辑推理，因此将选项放在了描述了后面。在问题回答这一场景中，论文中将问题放在描述前面掌柜猜测是因为：①两者并没有强烈的先后顺序；②问题相对较短放到前面可能好处理一点。所以基于这样的考虑，在问答任务中将问题放在了描述前面。不过后续大家依旧可以尝试交换一下顺序看看效果。

到此，对于问题回答模型的原理我们算是大致清楚了，下面首先来看如何构造数据集。

## 8.2 数据预处理

### 8.2.1 输入介绍

在正式介绍如何构建数据集之前我们先来看这么一个问题以及其对应的常见解决方案。如果在建模时碰到上下文过长时的情况该怎么办？是直接采取截断处理吗？如果问题答案恰巧是在被截断的那部分里面呢，还能直接截断吗？显然，认真想一想截断这种做法在这里肯定是不行的，因此在论文中作者也采用了另外一种方法来解决这一问题，那就是滑动窗口。

在问题回答这个任务场景中（其它场景也可以参考借鉴），当原始的上下文过长超过给定长度或者是 512 个字符时，可以采取滑动窗口的方法来构造整个模型的输入序列，如图 8-2 所示。

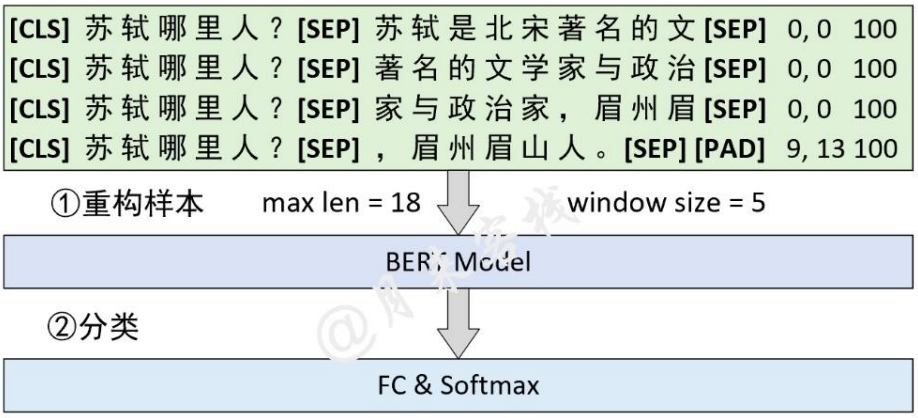



图 8-2. 问题回答滑动窗口训练时处理流程

在图 8-2 所示的这一场景中，第①步需要做的是根据指定最大长度和滑动窗口大小将原始样本进行滑动窗口处理并得到多个子样本。不过这里需要注意的是，sentence A，也就是问题部分不参与滑动处理。同时，图 8-2 中样本右边的 3 列数字分别表示每个子样本的起始结束索引和原始样本对应的 ID。紧接着第②步便是将所有原始样本滑动处理后的结果作为训练集来训练模型。

总的来说，在这一场景中训练程并没有太大的问题，因为每个子样本也都有其对应的标签值，因此和普通的训练过程并没有什么本质上的差异。因此，最关键的地方在于如何在推理过程中也使用滑动窗口。

### 8.2.2 结果筛选

一种最直观的做法就是直接取起始位置预测概率值加结束位置预测概率值最大的子样本对应的结果，作为整个原始样本对应的预测结果。不过下面掌柜将来介绍另外一种效果更好的处理方式（这也是论文中所采取的方式），其整个处理流程如图 8-3 所示。

如图 8-3 所示，在推理过程中第①步要做的仍旧是需要根据指定最大长度和滑动窗口大小将原始样本进行滑动窗口处理。接着第②步便是根据 BERT 分类的输出取前 K 个概率值最大的结果。在图 3 中这里的 K 值为 4，因此对于每个子样本来说其 start position 和 end position 分别都有 4 个候选结果。例如，第②步中第 1 行的 7:0.41,10:02,9:0.12,2:01 表示函数就是对于第 1 个子样本来说，start position 为索引 7 的概率值为 0.41，其它同理。

这样对于每一个子样本来说，在分别得到 start position 和 end position 的 K 个候选值后便可以通过组合来得到更多的候选预测结果，然后再根据一些规则来选择最终原始样本对应的预测输出。

根据图 8-3 中样本重构后的结果可以看出：(1)最终的索引预测结果肯定是大于 8 的，因为答案只可能在上下文中出现；(2)在进行结果组合的过程中，起始索引肯定是小于等于结束索引的。因此，根据这两个条件在经过步骤③的处理后，便可以得到进一步的筛选结果。例如，对于第 1 个子样本来说，start position 中

7 和 2 是不满足条件(1)的，所以可以直接去掉；同时，为了满足第(2)个条件所以在 end position 中 8，6，7 均需要去掉。

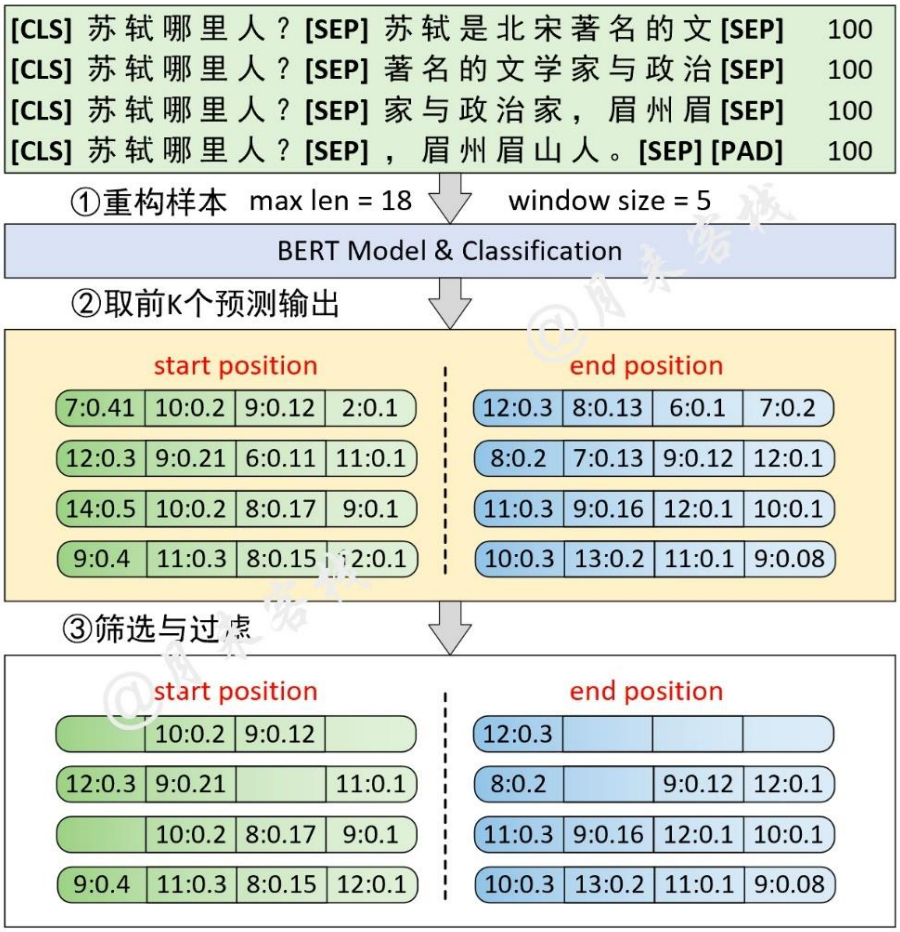



图 8-3. 问题回答滑动窗口推理时处理流程（一）

进一步，将第③步处理后的结果在每个子样本内部进行组合，并按照 start position 加 end position 值的大小进行排序，便可以得到如图 8-4 所示的结果。


图 8-4. 问题回答滑动窗口推理时处理流程（二）

如图 8-4 所示表示根据概率和排序后的结果。例如第 1 列 9，13，0.65 的含义便是最终原始样本预测结果为 9，13 的概率值为 0.65。因此，最终该原始样本对应的预测值便可以取 9 和 13。

### 8.2.3 语料介绍

由于没有找到类似的高质量中文数据集，所以在这里掌柜使用到的也是论文中所提到的 SQuAD（The Stanford Question Answering Dataset 1.1 ）数据集[19]，即给定一个问题和描述需要模型从描述中找出答案的起止位置（找不到数据集的客官也可以找掌柜要）。

从第 8.1 节内容的介绍来看，在问题回答这个任务场景中模型的整体原理并不复杂，只不过需要通过滑动窗口来重构输入，所以后续我们在预处理数据集的时候还需要费点功夫。同时，SQuAD 数据集本身的结构也略显复杂，所以这也相应加大了整个数据集构建的工作难度。如果不太想了解 SQuAD 数据集的构建流程也可以直接跳到对应预处理完成后的内容。下面掌柜就带着各位客官来一起梳理数据集的结构信息。

如图 8-5 所示便是数据集 SQuAD1.1 的结构形式。它整体由一个 json 格式的数据组成，其中的数据部分就在字段“data”中。可以看到 data 中存放的是一个列表，而列表中的每个元素可以看成是一篇文章，并以字典进行的存储。进一步，对于每一篇文章来说，其结构如图 8-6 所示。

如图 8-6 所示，对于 data 中的每一篇文章来说，由“title”和“paragraphs”这两个字段组成。可以看到 paragraphs 是一个列表，其中的每一个元素为一个字典，可以看做是每一篇文章的其中一个段落，即后续需要使用到的上下文描述 context。对于每一个段落来说，其包含有一段描述（“context”字段）和若干个问题与答案对（“qas”字段），其结构如图 8-7 所示。

```
{
  "data": [
    {"title": "Super_Bowl_50"...},
    {"title":"welcome to moon hotel","paragraphs": []},
    {"title": "...","paragraphs": []}
  ],
  "version": "1.1"
}
```

图 8-5. SQuAD 数据集结构图（一）

如图 8-7 所示，对于每个段落对应的问题组 qas 来说，其每一个元素都是答案（“answers”字段）、问题（“question”字段）和 ID（“id”字段）的字典形式。同时，这里的“answer_start”只是答案在 context 中字符层面的索引，而不是每个单词在 context 中的位置，所以后面还需要进行转换。而我们所需要完成的便是从数据集中提取出对应的 context、question、start pos、end pos 以及 id 信息。


图 8-6. SQuAD 数据集结构图（二）


图 8-7. SQuAD 数据集结构图（三）

到此，对于数据集 SQuAD1.1 的基本信息就介绍完了。下面掌柜就开始来一步步介绍如何构建数据集。

### 8.2.4 数据集预览

在正式介绍如何构建数据集之前我们同样先通过一张图来了解一下整个大致构建的流程。假如我们现在有两个样本构成了一个 batch，那么其整个数据的处理过程则如图 8-8 所示。由于英文样例普遍较长画图不太方便，所以这里就以中文进行了示例，不过两者原理都一样。

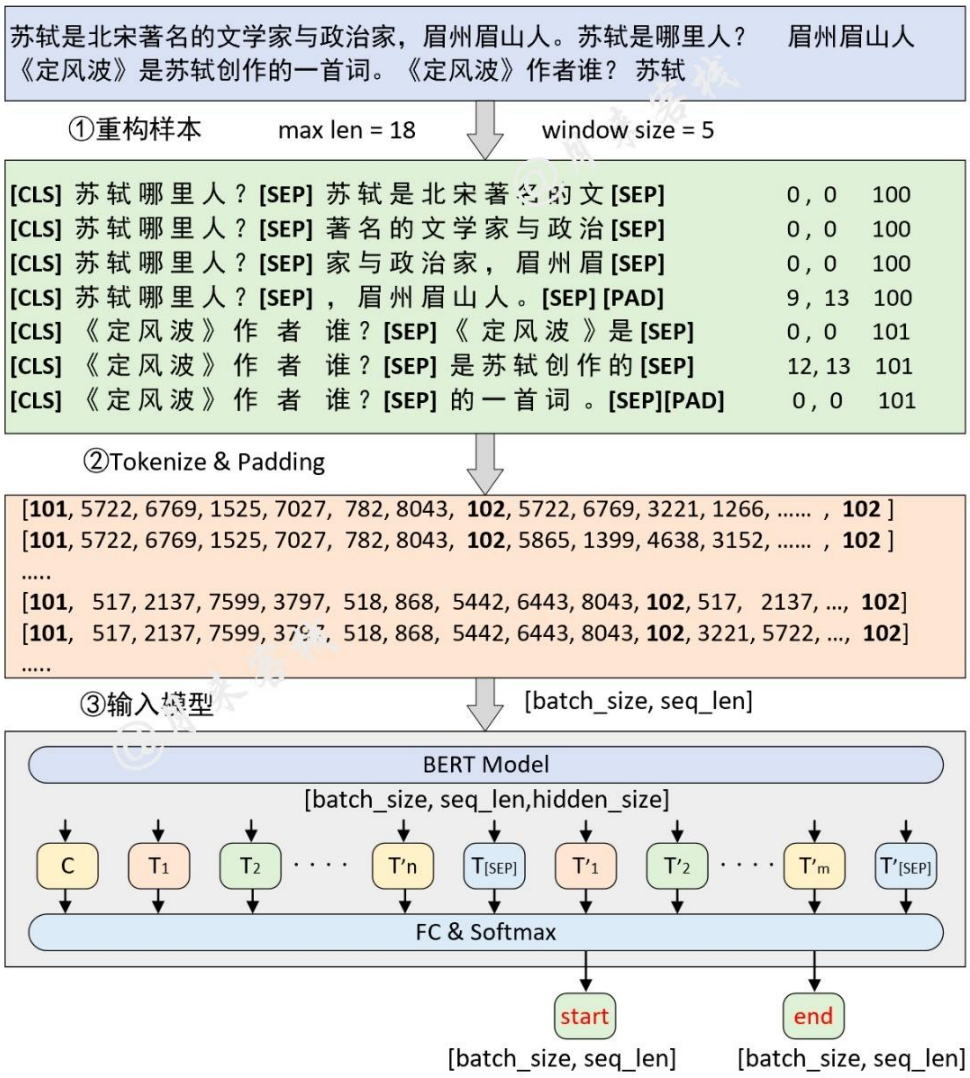



图 8-8. 问题回答模型数据集构造流程图

如图 8-8 所示，首先对于原始数据中的上下文按照指定的最大长度和滑动窗口大小进行滑动处理，然后再将问题同上下文拼接在一起构造成为一个序列并添加上对应的分类符[CLS]和分隔符[SEP]，即图中的第①步重构样本。紧接着需要将第①步构造得到的序列转换得到 Token id 并进行 padding 处理，此时便得到了一个形状为[batch_size,seq_len]的 2 维矩阵，即图 8 中第②步处理完成后形状为[7,18]的结果。同时，在第②步中还要根据每个序列构造得到相应的 attention_mask 向量和 token_types_ids 向量（图中未画出），并且两者的形状也是[batch_size,seq_len]。

最后，将第②步处理后的结果输入到 BERT 模型中，在经过 BERT 特征提取后将会得到一个形状为[batch_size,seq_len,hidden_size]的 3 维矩阵，最后再乘上一个形状为[hidden_size,2]的矩阵并变形成[batch_size,seq_len,2]的形状，即是对每个 Token 进行分类。

### 8.2.4 数据集构造

在说完数据集构造的整理思路后，下面我们就来正式编码实现整个数据集的构造过程。同样，对于数据预处理部分我们可以继续继承之前文本分类处理的这个类 LoadSingleSentenceClassificationDataset，然后再稍微修改其中的部分方法即可。同时，由于在前几个示例中已经就 tokenize 和词表构建等内容做了详细的介绍，所以这部分内容就不再赘述。

**(1) 规整化原始数据**

以下数据预处理代码主要参考自 https://github.com/google-research/bert 中的 run_squad.py 脚本。

**第 1 步：格式化文本**

从图 8-7 可以看出，SQuAD 原始数据集给出的每个问题的答案并不是标准的 start pos 和 end pos，因此我们需要自己编写一个函数来根据 answer_start 和 text 字段来获得答案在原始上下文中的 start pos 和 end pos（单词级别），实现代码如下所示：

```
    @staticmethod
    def get_format_text_and_word_offset(text):

        def is_whitespace(c):
            if c == " " or c == "\t" or c == "\r"
                                        or c == "\n" or ord(c) == 0x202F:
                return True
            return False

        doc_tokens = []
        char_to_word_offset = []
        prev_is_whitespace = True
        # 以下这个 for 循环的作用就是将原始 context 中的内容进行格式化
        for c in text:  # 遍历 paragraph 中的每个字符
            if is_whitespace(c):  # 判断当前字符是否为空格（各类空格）
                prev_is_whitespace = True
            else:
                if prev_is_whitespace:  # 如果前一个字符是空格
                    doc_tokens.append(c)
                else:
```

```
20                      doc_tokens[-1] += c  # 在 list 的最后一个元素中继续追加字符
21                  prev_is_whitespace = False
22              char_to_word_offset.append(len(doc_tokens) - 1)
23          return doc_tokens, char_to_word_offset
```

如上代码所示便是该函数的整体实现，其主要思路为：①第 12-13 行，依次遍历原始字符串中的每一个字符，然后判断其前一个字符是否为空格；②第 15-17 行，如果前一个字符是空格那么表示当前字符是一个新单词的开始，因此可以将前面的多个字符 append 到 list 中作为一个单词；③第 18-19 行，如果不是空格那么表示当前的字符仍旧属于上一个单词的一部分，因此将当前字符继续追加到上一个单词的后面；④同时，第 21 行用来记录当前当前字符所属单词的索引偏移量，这样根据 answer_start 字段来取 char_to_word_offset 中对应位置的值便得到了问题答案在上下文中的起始位置。

例如对于如下文本来说：

```
1 text = "Architecturally, the school has a Catholic character. "
```

函数 get_format_text_and_word_offset()返回的结果便是：

```
1 doc_tokens = ['Architecturally,', 'the', 'school', 'has', 'a', 'Catholic',
2               'character.'],
3 char_to_word_offset = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
4                        1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5, 5,
5                        5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6]
```

例如单词 school 在原始 text 中的起始位置为 21（字符级别），那么根据 char_to_word_offset[21]便可以返回的到 school 在原始文本中的位置为 2。

此时，细心的客官可能发现，返回的 doc_tokens 中有的单词里还包含有符号，并且很多单词在经过 tokenize 后还会被拆分成多个部分，因此这里只是暂时得到一个初步的起始位置，后续还会进行修正。

**第 2 步：读取数据**

根据图 8-5 到图 8-7 的展示，下面我们需要定义一个函数来对原始数据进行读取得到每个样本原始的字符串形式。

```
1 class LoadSQuADQuestionAnsweringDataset(
                                    LoadSingleSentenceClassificationDataset):
2
3     def __init__(self, doc_stride=64,
4                  max_query_length=64,
5                  n_best_size=20,
6                  max_answer_length=30,
7                  kwargs):
8         super(LoadSQuADQuestionAnsweringDataset, self).__init__(kwargs)
9         self.doc_stride = doc_stride
```

```
10          self.max_query_length = max_query_length
11          self.n_best_size = n_best_size
12          self.max_answer_length = max_answer_length
13
14      def preprocessing(self, filepath, is_training=True):
15          with open(filepath, 'r') as f:
16              raw_data = json.loads(f.read())
17              data = raw_data['data']
18          examples = []
19          ……
```

在上述代码中，首先定义了类 LoadSQuADQuestionAnsweringDataset 并继承自之前的 LoadSingleSentenceClassificationDataset 类；第 9-q 行分别表示滑动窗口的长度，以及问题的最大长度；第 14-19 行便是定义 preprocessing 函数用来对原始 json 数据进行读取，其中第 17 行返回的便是整个 json 格式的原始数据。

下面开始遍历 json 数据中的每个 paragraph：

```
1       def preprocessing(self, filepath, is_training=True):
2           # 接上面代码
3           for i in tqdm(range(len(data)), ncols=80, desc="正在遍历每一个段落"):
4               paragraphs = data[i]['paragraphs']
5               for j in range(len(paragraphs)):
6                   context = paragraphs[j]['context']
7                   context_tokens, word_offset =
                                self.get_format_text_and_word_offset(context)
8                   qas = paragraphs[j]['qas']
9                   for k in range(len(qas)):
10                      question_text = qas[k]['question']
11                      qas_id = qas[k]['id']
12                      if is_training:
13                          answer_offset = qas[k]['answers'][0]['answer_start']
14                          orig_answer_text = qas[k]['answers'][0]['text']
15                          answer_length = len(orig_answer_text)
16                          start_position = word_offset[answer_offset]
17                          end_position = word_offset[answer_offset +
                                                       answer_length - 1]
18                          actual_text = " ".join(context_tokens
19                                        [start_position:(end_position + 1)])
20                          cleaned_answer_text =
                                  " ".join(orig_answer_text.strip().split())
21                          if actual_text.find(cleaned_answer_text) == -1:
22                              logging.warning("Could not find answer: '%s' vs.
                                    '%s'", actual_text, cleaned_answer_text)
23                              continue
```

```
                    else:
                        start_position = None
                        end_position = None
                        orig_answer_text = None
                    examples.append([qas_id, question_text,
                                     orig_answer_text, " ".join(context_tokens),
                                     start_position, end_position])
        return examples
```

在上述代码中，第 3-4 行用来遍历原始数据中的每一篇文章；第 5-8 行用来遍历每一篇文章中的每个 paragraph，并取相应的上下文 context 和问题-答案对；第 9-11 行是用来遍历取每个 paragraph 中对应的多个问题和问题 id；第 12-23 行表示，如果当前处理的是训练集，那么再取问题对应的 answer_offset、orig_answer_text 并以此获取原始答案对应的起始位置 start_positin 和结束位置 end_position；第 18-22 行则是判断真实答案和根据起止位置从 context 中截取的答案是否相同，不同则跳过该条样本；第 25-27 行是用来处理验证集或测试集。

最后，该函数将会返回一个 2 维列表，内层列表中的各个元素分别为：

```
['问题 ID','原始问题文本','答案文本','context 文本','答案在 context 中的开始位置',
 '答案在 context 中的结束位置']
```

例如：

```
[['5733be284776f41900661182', 'To whom did the Virgin Mary allegedly appear
  in .... France?', 'Saint Bernadette Soubirous', 'Architecturally, the
  school has a Catholic character.....', 90, 92],
 ['5733be284776f4190066117f', ....]]
```

**第 3 步：修正起止位置**

在上面掌柜提到，此时得到的起止位置会因为不同的 tokenize 而产生变化，因此需要进一步进行修正。同时，原始文本中”(1895-1943)“这类形式的字符串也会被认为是一个单词，但是在 tokenize 后也会发生变化，所以需要一同进行处理。实现代码如下所示：

```
    @staticmethod
    def improve_answer_span(context_tokens,
                            answer_tokens,
                            start_position,
                            end_position):
        new_end = None
        for i in range(start_position, len(context_tokens)):
            if context_tokens[i] != answer_tokens[0]:
                continue
            for j in range(len(answer_tokens)):
                if answer_tokens[j] != context_tokens[i + j]:
```

```
                    break
                new_end = i + j
            if new_end - i + 1 == len(answer_tokens):
                return i, new_end
        return start_position, end_position
```

在上述代码中，context_tokens 为已经 tokenize 后的原始上下文；answer_tokens 为原始答案文本经过 tokenize 后的结果，例如：

```
context = "Virgin mary reputedly appeared to Saint Bernadette Soubirous in
          1858"
answer_text = "Saint Bernadette Soubirous"
start_position = 5
end_position = 7
context_tokens: ['virgin', 'mary', 'reputed', '##ly', 'appeared', 'to',
'saint','bern', '##ade', '##tte', 'so', '##ub', '##iro', '##us', 'in','1858']
answer_tokens: ['saint', 'bern', '##ade', '##tte','so','##ub','##iro', '##us'
```

则其返回后新的起止位置为[6,13]。

这段代码的主要思路是以原始起始位置开始，依次将后续的每个片段同 answer_tokens 进行对比，如果 context_tokens 中的子片段等同于 answer_tokens，那么就返回新的起止位置；否则还是返回传入的起止位置。

同样，对于下面的示例：

```
context = "The leader was John Smith (1895-1943).
answer_test = "1985"
answer_tokens: ["1895"]
start_position: 5
end_position: 5
context_tokens: ['the', 'leader', 'was', 'john', 'smith',
                 '(', '1895', '-', '1943', ')', '.']
```

返回新的起止位置便为[6,6]。

### (2) 重构输入样本

在通过预处理函数 preprocessing()后，我们便可以来进一步地采用滑动窗口方式来构造模型的输入。由于这部分代码稍微有点长，所以掌柜下面就分块进行介绍。

```
    @cache
    def data_process(self, filepath, is_training=False, postfix='cache'):
        logging.info(f"## 使用窗口滑动滑动，doc_stride = {self.doc_stride}")
        examples = self.preprocessing(filepath, is_training)
        all_data = []
        example_id, feature_id = 0, 1000000000
        for example in tqdm(examples,ncols=80,desc="正在遍历每个问题（样本）"):
```

```
8             question_tokens = self.tokenizer(example[1])
9             if len(question_tokens) > self.max_query_length:# 问题过长进行截取
10                question_tokens = question_tokens[:self.max_query_length]
11            question_ids = [self.vocab[token] for token in question_tokens]
12            question_ids = [self.CLS_IDX] + question_ids + [self.SEP_IDX]
13            context_tokens = self.tokenizer(example[3])
14            context_ids = [self.vocab[token] for token in context_tokens]
15            logging.debug(f"<<<<<<<<<  进入新的 example  >>>>>>>>>")
16            logging.debug(f"## 正在预处理数据 {__name__}
                                              is_training = {is_training}")
17            logging.debug(f"## 问题 id: {example[0]}")
18            logging.debug(f"## 原始问题 text: {example[1]}")
19            logging.debug(f"## 原始描述 text: {example[3]}")
20            start_position, end_position, answer_text = -1, -1, None
21            if is_training:
22                start_position, end_position = example[4], example[5]
23                answer_text = example[2]
24                answer_tokens = self.tokenizer(answer_text)
25                start_position, end_position = self.improve_answer_span(
26                   context_tokens,answer_tokens,start_position,end_position)
27            rest_len = self.max_sen_len - len(question_ids) - 1
28            context_ids_len = len(context_ids)
29            logging.debug(f"## 上下文长度为：{context_ids_len},剩余长度
                                  rest_len 为：{rest_len}")
```

在上述代码中，第 1 行@cache 的作用是将 data_process()处理后的结果进行缓存，以方便下次直接从本地载入节约时间，具体原理可以参见文章 [27]；第 7 行则是开始遍历 preprocessing()函数返回的每一条原始数据；第 8-14 行则是取对应我们后续所需要的成分；第 21-26 行则是用来获取得到训练集中 start_position、end_position 以及 answer_text ；第 27-28 行是用来计算 context 的长度判读是否需要进行滑动窗口处理。

以下代码则是滑动窗口处理部分：

```
1             if context_ids_len > rest_len: # 长度超过 max_sen_len,需要进行滑动窗口
2                 logging.debug(f"## 进入滑动窗口 …… ")
3                 s_idx, e_idx = 0, rest_len
4                 while True:
5                     tmp_context_ids = context_ids[s_idx:e_idx]
6                     tmp_context_tokens = [self.vocab.itos[item] for item
7                                                 in tmp_context_ids]
8                     input_ids = torch.tensor(question_ids +
9                                         tmp_context_ids + [self.SEP_IDX])
10                    input_tokens = ['[CLS]'] + question_tokens +
11                                ['[SEP]'] +tmp_context_tokens + ['[SEP]']
```

```
12                    seg = [0]  len(question_ids) + [1]  (len(input_ids)
13                                                  - len(question_ids))
14                    seg = torch.tensor(seg)
15                    if is_training:
16                        new_start_position, new_end_position = 0, 0
17                        if start_position >=s_idx and end_position <= e_idx:
18                            logging.debug(f"## 滑动窗口中存在答案 -----> ")
19                            new_start_position = start_position - s_idx
20                            new_end_position = new_start_position +
21                                          (end_position - start_position)
22                            new_start_position += len(question_ids)
23                            new_end_position += len(question_ids)
24                        all_data.append([example_id, feature_id, input_ids,
25                                 seg, new_start_position,new_end_position,
26                                   answer_text, example[0], input_tokens])
27                        logging.debug(f"## start pos:{new_start_position}")
28                        logging.debug(f"## end pos:{new_end_position}")
29                    else:
30                        all_data.append([example_id, feature_id, input_ids,
31                              seg, start_position,end_position, answer_text,
32                              example[0], input_tokens])
33                        logging.debug(f"## start pos:{start_position}")
34                        logging.debug(f"## end pos:{end_position}")
35                    token_to_orig_map =
                                self.get_token_to_orig_map(input_tokens,
36                              example[3], self.tokenizer)
37                    all_data[-1].append(token_to_orig_map)
38                    feature_id += 1
39                    if e_idx >= context_ids_len:
40                        break
41                    s_idx += self.doc_stride
42                    e_idx += self.doc_stride
```

在上述代码中，第 4 代码开始便是进入滑动窗口循环处理中；第 5-14 行则是构造得到相应的所需部分，包括 input ids、input tokens 和 segment ids 等；第 15-32 行是分别取训练和预测时输入序列对应答案所在的索引位置，并同其余部分形成一个原始样本存入 all_data 当中；第 32-36 行则是返回得 input_tokens 中每个 Token 在原始单词中所对应的位置索引，这一结果将会在最后推理过程中得到最后预测结果时用到。

例如现在有如下 Token：

```
1 input_tokens = ['[CLS]', 'to', 'whom', 'did', 'the', 'virgin', '[SEP]',
2 'architectural', '##ly', ',', 'the', 'school', 'has', 'a', 'catholic',
3 'character', '.', '[SEP']
```

那么上下文 Token 在原始上下文中的索引映射表则为：

```
origin_context = "Architecturally, the Architecturally, test, Architecturally, the school has a Catholic character. Welcome moon hotel"
orig_map = {7: 4, 8: 4, 9: 4, 10: 5, 11: 6, 12: 7, 13: 8, 14: 9,15:10,16: 10}
```

其含义表示，input_tokens[7]为 origin_context 中的第 4 个单词 Architecturally，同理 input_tokens[10]为 origin_context 中的第 5 个单词 the。

以下代码则是非滑动窗口处理部分

```
            else:
                input_ids = torch.tensor(question_ids + context_ids + [self.SEP_IDX])
                input_tokens = ['[CLS]'] + question_tokens + ['[SEP]'] + context_tokens + ['[SEP]']
                seg = [0]  len(question_ids) + [1]  (len(input_ids) - len(question_ids))
                seg = torch.tensor(seg)
                if is_training:
                    start_position += (len(question_ids))
                    end_position += (len(question_ids))
                token_to_orig_map = self.get_token_to_orig_map(input_tokens, example[3], self.tokenizer)
                all_data.append([example_id, feature_id, input_ids, seg, start_position,
                                 end_position, answer_text, example[0], input_tokens, token_to_orig_map])
                logging.debug(f"## input_tokens: {input_tokens}")
                logging.debug(f"## input_ids:{input_ids.tolist()}")
                logging.debug(f"## segment ids:{seg.tolist()}")
                logging.debug(f"## orig_map:{token_to_orig_map}")
                logging.debug("====================\n")
                feature_id += 1
            example_id += 1
        data = {'all_data': all_data, 'max_len': self.max_sen_len, 'examples': examples}
        return data
```

其中 all_data 中的每个元素分别为：原始样本 id、训练特征 id、input_ids、seg、开始位置、结束位置、答案文本、问题 id、input_tokens 和 ori_map。

### (3) 构造 DataLoader

在完成前面两部分内容后，只需要在构造每个 batch 时对输入序列进行 padding 并返回的相应的 DataLoader 整个数据集就算是构造完成了。为此，首先需要定义一个函数来构造每一个 batch，代码如下：

```
1     def generate_batch(self, data_batch):
2         batch_input, batch_seg, batch_label, batch_qid = [], [], [], []
3         batch_example_id, batch_feature_id, batch_map = [], [], []
4         for item in data_batch:
5             # item: [原始样本 Id,训练特征 id,input_ids, seg, 开始,
6             #         结束, 答案文本, 问题 id,input_tokens,ori_map]
7             batch_example_id.append(item[0])  # 原始样本 Id
8             batch_feature_id.append(item[1])  # 训练特征 id
9             batch_input.append(item[2])  # input_ids
10            batch_seg.append(item[3])  # seg
11            batch_label.append([item[4], item[5]])  # 开始, 结束
12            batch_qid.append(item[7])  # 问题 id
13            batch_map.append(item[9])  # ori_map
14
15        batch_input = pad_sequence(batch_input,  # [batch_size,max_len]
16                                   padding_value=self.PAD_IDX,batch_first=False,
17                                   max_len=self.max_sen_len)  # [max_len,batch_size]
18        batch_seg = pad_sequence(batch_seg,  # [batch_size,max_len]
19                                 padding_value=self.PAD_IDX,batch_first=False,
20                                 max_len=self.max_sen_len)  # [max_len, batch_size]
21        batch_label = torch.tensor(batch_label, dtype=torch.long)
22        # [max_len,batch_size] , [max_len, batch_size] , [batch_size,2],
           [batch_size,], [batch_size,]
23        return batch_input, batch_seg, batch_label, batch_qid,
                  batch_example_id, batch_feature_id, batch_map
```

在上述代码中，第 1 行中的 data_batch 便是 data_process()处理后返回的 all_data；第 4-13 行便是构造每个 batch 中所包含的向量；第 15-21 行则是根据指定的参数 max_len 来进行 padding 处理（关于 pad_sequence 函数的详细介绍见第 4.2.4 节内容）；第 23 行是用来返回每个 batch 处理后的结果

进一步，构造得到数据集对应的 DataLoader，代码如下：

```
1     def load_train_val_test_data(self, train_file_path=None,
2                                  val_file_path=None,
3                                  test_file_path=None,
4                                  only_test=True):
5         doc_stride = str(self.doc_stride)
6         max_sen_len = str(self.max_sen_len)
7         max_query_length = str(self.max_query_length)
8         postfix = doc_stride + '_' + max_sen_len + '_' + max_query_length
9         data = self.data_process(filepath=test_file_path,
10                                 is_training=False,postfix=postfix)
11        test_data, examples = data['all_data'], data['examples']
12        test_iter = DataLoader(test_data, batch_size=self.batch_size,
```

```
13                      shuffle=False,collate_fn=self.generate_batch)
14          if only_test:
15              return test_iter, examples
16
17          data = self.data_process(filepath=train_file_path,
18                                   is_training=True,postfix=postfix)
19          train_data, max_sen_len = data['all_data'], data['max_len']
20          _, val_data = train_test_split(train_data, test_size=0.3,
                                                        random_state=2021)
21          if self.max_sen_len == 'same':
22              self.max_sen_len = max_sen_len
23          train_iter = DataLoader(train_data, batch_size=self.batch_size,
24              shuffle=self.is_sample_shuffle, collate_fn=self.generate_batch)
25          val_iter = DataLoader(val_data, batch_size=self.batch_size,
26                                shuffle=False, collate_fn=self.generate_batch)
27          logging.info(f"## 成功返回训练集样本（{len(train_iter.dataset)}）个、"
28                       f"开发集样本（{len(val_iter.dataset)}）个"
29                       f"测试集样本（{len(test_iter.dataset)}）个.")
30          return train_iter, test_iter, val_iter
```

在上述代码中，第 5-8 行则是根据对应的超参数来构造一个数据预处理结果缓存的名称；第 9-15 行便是用来构造得到对应测试集的 DataLoader；在第 20 行中，掌柜将原始的训练集又划分成了两个部分，并在 23-26 行中分别返回了两者对应的 DataLoader。

同时，这里特别需要注意的一点是，为了方便后续对预测结果进行处理，掌柜直接固定了验证集和测试集中每个样本的顺序，即 shuffle=False。因为当 shuffle 为 True 时，每次通过 for 循环遍历 data_iter 时样本的顺序都不一样。

到此，对于整个 SQuAD1.1 版数据集的构建流程就介绍了。

### (4) 使用示例

在完成上述 3 个步骤之后，可以通过如下代码进行数据集的载入：

```
1 from utils.data_helpers import LoadSQuADQuestionAnsweringDataset
2 from transformers import BertTokenizer
3 from Tasks.TaskForSQuADQuestionAnswering import ModelConfig
4
5 if __name__ == '__main__':
6     config = ModelConfig()
7     tokenizer = BertTokenizer.from_pretrained(config.pretrained_model_dir)
8     data_loader = LoadSQuADQuestionAnsweringDataset(
9                     vocab_path=config.vocab_path,
10                    tokenizer=tokenizer.tokenize,batch_size=5,max_sen_len=120,
11                    max_position_embeddings=512,pad_index=0,
12                    is_sample_shuffle=False,
```

```
13                  doc_stride=8,max_query_length=5)
14
15      train_iter, test_iter, val_iter = data_loader. \
16          load_train_val_test_data(test_file_path=config.test_file_path,
17                                   train_file_path=config.train_file_path,
18                                   only_test=False)
19      for b_input, b_seg, b_label, b_qid, b_example_id, b_feature_id,
                                                        b_map in train_iter:
20          print("====================>")
21          print(f"intput_ids shape: {b_input.shape}") # [max_len, batch_size]
22          print(f"token_type_ids shape: {b_seg.shape}")# [max_len, batch_size]
23          print(b_seg.transpose(0, 1).tolist())
24          print(b_label)  # [batch_size,2]
25          print(b_map)  # [batch_size]
26          for i in range(b_input.size(-1)):
27              sample = b_input.transpose(0, 1)[i]
28              start_pos, end_pos = b_label[i][0], b_label[i][1]
29              strs = [data_loader.vocab.itos[s] for s in sample]  # 原始 tokens
30              answer = " ".join(strs[start_pos:(end_pos+1)]).replace(" ##","")
31              strs = " ".join(strs).replace(" ##", "").split('[SEP]')
32              question, context = strs[0], strs[1]
33              print(f"问题 ID：{b_qid[i]}")
34              print(f"问题：{question}")
35              print(f"正确答案：{answer}")
36              print(f"答案起止：{start_pos, end_pos}")
37              print(f"example ID：{b_example_id[i]}")
38              print(f"feature ID：{b_feature_id[i]}")
```

在上述代码中，第 15-18 行便是返回训练集、验证集和测试集对应的 DataLoader；第 19-25 行是用来遍历训练集中每个 batch 的数据；第 27-38 行是遍历一个 batch 中的每个样本，并转换为对应的文本形式。

最后，上述代码输入结果将类似如下所示：

```
-INFO: 导入配置~/BertWithPretrained/bert_base_uncased_english/config.json
-INFO: ### 将当前配置打印到日志文件中
-INFO: ### project_dir = ~/BertWithPretrained
-INFO: ### dataset_dir = ~/BertWithPretrained/data/SQuAD
-INFO: ### train_file_path = ~/BertWithPretrained/data/SQuAD/train-v1.1.json
......
-INFO: ### model_save_path = ~/BertWithPretrained/cache/model.pt
-INFO: ### n_best_size = 10
-INFO: ### max_answer_len = 30
......
-INFO: 缓存~/BertWithPretrained/data/SQuAD/dev-v1_8_120_5.pt 不存在，重新处理!
```

```
-INFO: ## 使用窗口滑动滑动，doc_stride = 8
正在遍历每一个段落: 100%|██████████████| 48/48 [00:00<00:00, 60.80it/s]
正在遍历每个问题（样本）:  0%|                | 0/10570 [00:00<?, ?it/s]
-DEBUG: <<<<<<<<<  进入新的 example  >>>>>>>>>
-DEBUG: ## 正在预处理数据 utils.data_helpers is_training = False
-DEBUG: ## 问题 id: 56be4db0acb8001400a502ec
-DEBUG: ## 原始问题 text:Which NFL team represented the AFC at Super Bowl 50?
-DEBUG: ## 原始描述 text: Super Bowl 50 was an American football game to
  determine the champion of the National Football League (NFL) for the 2015
  season. The American Football Conference (AFC) champion  ......
-DEBUG: ## 上下文长度为：157, 剩余长度 rest_len 为 :  112
-DEBUG: ## 滑动窗口范围：(0, 112),example_id: 0, feature_id: 1000000000
-DEBUG: ## start pos:-1 end pos:-1 end pos:-1 end pos:-1
-DEBUG: ## input_tokens: ['[CLS]', 'which', 'nfl', 'team', 'represented',
  'the', '[SEP]', 'super', 'bowl', '50', 'was', 'an', 'american', 'football',
  'game', 'to', 'determine', 'the', 'champion', ......, '[SEP]']
-DEBUG: ## input_ids:[101, 2029, 5088, 2136, 3421, 1996, 102, 3565, 4605,
      2753, 2001, 2019, 2137, 2374, 2208, 2000, 5646, 1996, 3410, 1997, 1996,
      2120, 2374, 2223, 1006, 5088, 1007, 2005, 1996, 2325, ......, 102]
-DEBUG: ## segment ids:[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
          1, 1, 1, 1, 1, 1, 1, 1, 1,  ......1]
-DEBUG: ## orig_map:{7: 0, 8: 1, 9: 2, 10: 3, 11: 4, 12: 5, 13: 6, 14: 7,
          15: 8, 16: 9, 17: 10, 18: 11, 19: 12, 20: 13, 21: 14, 22: 15, 23: 16,
          24: 17, 25: 17, 26: 17, 27: 18, ......}
-DEBUG: ====================
问题 ID：5733bf84d058e614000b61c1
问题：[CLS] in what year did the
正确答案：1987
答案起止：(tensor(60), tensor(60))
example ID：9
feature ID：1000000125
问题 ID：5733bf84d058e614000b61c1
问题：[CLS] in what year did the
正确答案：1987
......
```

到此，对于整个数据集的介绍就完了。如果是正在研究这类问题回答模型的客官，掌柜强烈建议你结合本文一行一行地去阅读本项目中的原始代码，你一定会受益匪浅。

下面掌柜开始继续介绍问题回答模型的实现部分。

## 8.3 问答任务

### 8.3.1 前向传播

正如第 8.1 节内容所介绍，我们只需要在原始 BERT 模型的基础上取最后一层的输出结果，然后再加一个分类层即可。因此这部分代码相对来说也比较容易理解。

如图 2-4 所示，在 BertForMultipleChoice.py 模块中，首先需要定义一个类以及相应的初始化函数，代码如下：

```
1 from ..BasicBert.Bert import BertModel
2 import torch.nn as nn
3
4 class BertForQuestionAnswering(nn.Module):
5
6     def __init__(self, config, bert_pretrained_model_dir=None):
7         super(BertForQuestionAnswering, self).__init__()
8         if bert_pretrained_model_dir is not None:
9             self.bert = BertModel.from_pretrained(config,
                                                    bert_pretrained_model_dir)
10        else:
11            self.bert = BertModel(config)
12        self.qa_outputs = nn.Linear(config.hidden_size, 2)
```

在上述代码中，第 8-11 行便是根据相应的条件返回一个 BERT 模型，第 12 行则是定义了一个分类层。

然后再是定义完成整个前向传播过程，代码如下：

```
1     def forward(self, input_ids,attention_mask=None,
2                 token_type_ids=None,position_ids=None,
3                 start_positions=None,end_positions=None):
4         _, all_encoder_outputs = self.bert(
5             input_ids=input_ids,attention_mask=attention_mask,
6             token_type_ids=token_type_ids,position_ids=position_ids)
7         sequence_output = all_encoder_outputs[-1]  # 取 Bert 最后一层的输出
8         # sequence_output: [src_len, batch_size, hidden_size]
9         logits = self.qa_outputs(sequence_output)  # [src_len, batch_size,2]
10        start_logits, end_logits = logits.split(1, dim=-1)
11        # [src_len,batch_size,1]  [src_len,batch_size,1]
12        start_logits = start_logits.squeeze(-1).transpose(0, 1)
13        end_logits = end_logits.squeeze(-1).transpose(0, 1)
14        if start_positions is not None and end_positions is not None:
15            ignored_index = start_logits.size(1)  # 取输入序列的长度
16            start_positions.clamp_(0, ignored_index)
```

```
17              end_positions.clamp_(0, ignored_index)
18              loss_fct = nn.CrossEntropyLoss(ignore_index=ignored_index)
19              start_loss = loss_fct(start_logits, start_positions)
20              end_loss = loss_fct(end_logits, end_positions)
21              return (start_loss + end_loss) / 2, start_logits, end_logits
22          else:
23              return start_logits, end_logits  # [batch_size,src_len]
```

在上述代码中，第 4-7 行便是根据输入返回原始 BERT 模型的输出结果，需要注意的是这里要取 BERT 输出整个最后 1 层的输出结果，而不是像之前一样只取最后 1 层第 1 个位置[CLS]对应的向量。第 9 行便是分类层的输出结果，形状为[src_len, batch_size,2]。第 10-13 行则是得到对应的 start_logits 和 end_logits，两者的形状均是[batch_size,src_len]。第 14-23 行则是根据是否有标签返回对应的损失或者 logits 值；第 15-17 行是用来处理当给定的 start_positions 和 end_positions 在[0,max_len]这个范围之外时，强制将其改为 0 或 max_len；例如某个样本的起始位置为 520，而序列最大长度为 512（即此时 ignore_index=512），那么 clamp_()方法便会将 520 改变成 512，当然根据前面我们的数据处理流程来最后生成的数据集并不存在这样的情况，这里只是以防万一。在第 18 行中，之所以要将 ignored_index 作为损失计算时的忽略值，是因为这些位置并不能算是模型预测错误的（只能看做是没有预测），而是答案超出了范围所以需要忽略掉这些情况.最后，第 19-20 行是分别计算。

到此，对于问题回答任务的模型定义就介绍完了，可以看出这一过程也并不复杂。掌柜这里依旧建议各位客官在阅读代码时能够带着对应维度进行理解，以便对每一步变换有着清晰的认识。

### 8.3.2 模型训练

首先，如图 2-5 所示在 Tasks 目录下新建一个名为 TaskForSQuADQuestionAnswering.py 的模块，然后定义一个 ModelConfig 类来对分类模型中的超参数以及其它变量进行管理。部分代码如下所示：

```
1  class ModelConfig:
2      def __init__(self):
3          self.project_dir =
                  os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
4          self.dataset_dir = os.path.join(self.project_dir, 'data', 'SQuAD')
5          self.vocab_path=os.path.join(self.pretrained_model_dir, 'vocab.txt')
6          self.train_file_path=os.path.join(self.dataset_dir,'train-1.1.json')
7          self.test_file_path =os.path.join(self.dataset_dir, 'dev-v1.1.json')
8          self.model_save_dir = os.path.join(self.project_dir, 'cache')
9          self.logs_save_dir = os.path.join(self.project_dir, 'logs')
10         self.model_save_path = os.path.join(self.model_save_dir, 'model.pt')
11         self.n_best_size = 10
```

```
12          self.max_answer_len = 30
13          self.is_sample_shuffle = True
14          self.use_torch_multi_head = False
15          self.max_sen_len = 384
16          self.max_query_len = 64  #
17          self.doc_stride = 128
18          self.epochs = 2
19          self.model_val_per_epoch = 1
20          logger_init(log_file_name='qa', log_level=logging.DEBUG,
21                      log_dir=self.logs_save_dir)
```

在上述代码中，第 11 行是对预测出的答案进行后处理时，选取的候选答案数量，这个将在下一小节中进行介绍。第 12 行是在对候选进行筛选时，对答案最大长度的限制。第 13 行表示是否对训练集进行打乱。第 14 行表示是否使用 PyTorch 中的 multihead 实现，如果为 False 则使用项目 MyTransformer.py 中的多头注意力实现，如果是 True 则会使用 torch.nn.MultiheadAttention 中的多头注意力实现。第 15 行是最大句子长度，即[CLS] + question ids + [SEP] + context ids + [SEP]的长度。第 16 行是问题的最大长度，超过长度则进行截取。第 17 行是#滑动窗口一次滑动的长度。

紧接着，我们便可以定义函数 train()来完成模型的训练：

```
1  def train(config):
2      model = BertForQuestionAnswering(config,config.pretrained_model_dir)
3      if os.path.exists(config.model_save_path):
4          loaded_paras = torch.load(config.model_save_path)
5          model.load_state_dict(loaded_paras)
6          logging.info("## 成功载入已有模型，进行追加训练.....")
7      tokenize = BertTokenizer.from_pretrained(config.pretrained_model_dir)
8      data_loader = LoadSQuADQuestionAnsweringDataset(
9                  vocab_path=config.vocab_path,
10                 tokenizer=tokenize.tokenize,batch_size=config.batch_size,
11                 max_sen_len=config.max_sen_len,
12                 max_query_length=config.max_query_len,
13                 max_position_embeddings=config.max_position_embeddings,
14                 pad_index=config.pad_token_id,
15                 is_sample_shuffle=config.is_sample_shuffle,
16                 doc_stride=config.doc_stride)
17     train_iter, test_iter, val_iter = \
18         data_loader.load_train_val_test_data(
19                     train_file_path=config.train_file_path,
20                     test_file_path=config.test_file_path,only_test=False)
21     lr_scheduler = get_scheduler(name='linear',optimizer=optimizer,
22                      num_warmup_steps=int(len(train_iter)  0),
```

```
23                      num_training_steps=int(config.epochs  len(train_iter)))
24      max_acc = 0
25      for epoch in range(config.epochs):
26          for idx, (batch_input, batch_seg, batch_label, _, _, _, _)
                                                    in enumerate(train_iter):
27              padding_mask=(batch_input== data_loader.PAD_IDX).transpose(0, 1)
28              loss, start_logits, end_logits = model(input_ids=batch_input,
29                      attention_mask=padding_mask, token_type_ids=batch_seg,
30                      position_ids=None, start_positions=batch_label[:, 0],
31                      end_positions=batch_label[:, 1])
32              lr_scheduler.step()
33              ......
34              if idx % 100 == 0:
35                  y_pred = [start_logits.argmax(1), end_logits.argmax(1)]
36                  y_true = [batch_label[:, 0], batch_label[:, 1]]
37                  show_result(batch_input,data_loader.vocab.itos,y_predy_true)
38          if (epoch + 1) % config.model_val_per_epoch == 0:
39              acc = evaluate(val_iter,model,config.device,
                                                    data_loader.PAD_IDX,False)
40              logging.info(f"## Acc on val:{round(acc, 4)} max :{max_acc}")
```

在上述代码中，第 2 行是返回整个问答模型。第 3-6 行是载入已有模型进行追加训练。第 8-20 行是根据对应参数返回训练集、验证集和测试集。第 21-23 行是定义动态学习率（这个技巧在训练过程中尤其重要）。第 25-40 行是正式开始进入模型的训练过程中，其中第 37 行中的 show_result 函数是用来展示训练时的预测结果。第 39 行是计算当前模型在验证集上的准去率。

最后，模型训练过程中将会输出类似如下信息：

```
1 - INFO: Epoch:0, Batch[810/7387] Train loss: 0.998, Train acc: 0.708
2 - INFO: Epoch:0, Batch[820/7387] Train loss: 1.130, Train acc: 0.708
3 - INFO: Epoch:0, Batch[830/7387] Train loss: 1.960, Train acc: 0.375
4 - INFO: Epoch:0, Batch[840/7387] Train loss: 1.933, Train acc: 0.542
5 ......
6 - INFO: ### Quesiotn:[CLS] when was the first university in switzerland
  founded..
7 - INFO:     ## Predicted answer: 1460
8 - INFO:     ## True answer: 1460
9 - INFO:     ## True answer idx: (tensor(46, tensor(47))
10 - INFO: ### Quesiotn:[CLS] how many wards in plymouth elect two councillors?
11 - INFO:     ## Predicted answer: 17 of which elect three .....
12 - INFO:     ## True answer: three
13 - INFO:     ## True answer idx: (tensor(25, tensor(25))
```

到此，掌柜算是把整个模型的训练部分给介绍完了。下面，我们就正式进入具有挑战的模型推理部分的介绍。

## 8.4 模型推理

掌柜这里之所有把模型的推理与评估单独作为一个小节来进行介绍，是因为这部分内容对于 SQuAD 这个任务来说非常重要并且内容也很多。如果这部分内容处理不好，那么最终得到的结果和原始论文中的结果可能就会产生 10 个点的差距。因此，掌柜在复现这部分代码时为了达到和论文中相当的表现结果，也算是费了九牛二虎之力。

### 8.4.1 模型评估

首先，我们需要在 TaskForSQuADQuestionAnswering.py 文件中定义一个评估函数来对模型的表现结果进行评估。在推理时通过该函数便可以返回每个子样本对应的 logits 预测输出，然后再按照图 8-3 和图 8-4 的方法确定得到每个原始样本的预测结果；而在非推理阶段时，返回的则是所有子样本预测结果对应的准确率。代码实现如下：

```
def evaluate(data_iter, model, device, PAD_IDX, inference=False):
    model.eval()
    with torch.no_grad():
        acc_sum, n = 0.0, 0
        all_results = collections.defaultdict(list)
        for batch_input, batch_seg, batch_label, batch_qid, _,
                                            batch_feature_id, _ in data_iter:
            batch_input = batch_input.to(device)
            batch_seg = batch_seg.to(device)
            batch_label = batch_label.to(device)
            padding_mask = (batch_input == PAD_IDX).transpose(0, 1)
            start_logits, end_logits = model(input_ids=batch_input,
                       attention_mask=padding_mask,token_type_ids=batch_seg,
                       position_ids=None)

            all_results[batch_qid[0]].append([batch_feature_id[0],
                        start_logits.cpu().numpy().reshape(-1),
                        end_logits.cpu().numpy().reshape(-1)])
            if not inference:
                acc_sum_start = (start_logits.argmax(1) ==
                                        batch_label[:, 0]).float().sum().item()
                acc_sum_end = (end_logits.argmax(1) ==
                                        batch_label[:, 1]).float().sum().item()
                acc_sum += (acc_sum_start + acc_sum_end)
                n += len(batch_label)
```

```
26          model.train()
27          if inference:
28              return all_results
29          return acc_sum / (2  n)
```

在上述代码中，第 5 行的作用是用来定义一个默认 value 值为空 list 的字典，详细介绍可参见文章[20]。第 12-14 行则是返回一个 batch 所有样本对应的开始和结束位置的 logits；第 16-18 行则是将同一个问题下所有子样本的预测结果保存到一个 list 中，并且只有当 batch_size=1 时保存的结果才是正确形式。第 20-25 行则是统计 start position 和 end position 预测正确的数量；第 27-29 行则是根据不同的条件返回 logits 或准确率。有了这个评估函数，我们可以在训练时返回模型在验证集上的准确率，在推理时返回模型在测试集上的 logits。

### 8.4.2 模型推理

在完成评估函数定义后，我们只需要再在 TaskForSQuADQuestionAnswering.py 模块中定义一个用于推理的函数便可以在模型训练结束后对测试集进行推理预测并返回相应的 logits 结果。实现代码如下所示：

```
1  def inference(config):
2      tokenize = BertTokenizer.from_pretrained(config.pretrained_model_dir).
3      data_loader = LoadSQuADQuestionAnsweringDataset(config.vocab_path,
4          tokenizer=tokenize.tokenize,batch_size=1,
           max_sen_len=config.max_sen_len,
5          doc_stride=config.doc_stride,max_query_length=config.max_query_len,
6          max_answer_length=config.max_answer_len,
7          max_position_embeddings=config.max_position_embeddings,
8          pad_index=config.pad_token_id,n_best_size=config.n_best_size)
9      test_iter, all_examples = data_loader.load_train_val_test_data(
10             test_file_path=config.test_file_path,only_test=True)
11     model = BertForQuestionAnswering(config,config.pretrained_model_dir)
12     if os.path.exists(config.model_save_path):
13         loaded_paras = torch.load(config.model_save_path)
14         model.load_state_dict(loaded_paras)
15         logging.info("## 成功载入已有模型，开始进行推理.....")
16     else:
17         raise ValueError(f"## 模型{config.model_save_path}不存在，"
18                          f"请检查路径或者先训练模型.....")
19     model = model.to(config.device)
20     all_result_logits = evaluate(test_iter, model, config.device,
21                                  data_loader.PAD_IDX, inference=True)
22     data_loader.write_prediction(test_iter, all_examples,
23                                  all_result_logits, config.dataset_dir)
```

在上述代码中，第 3-10 行是根据相应的超参数来返回对应测试集，其中需要注意的一点便是第 4 行中的 batch_size 只能是 1，因为从 evaluate 函数中的第 16 行代码可以看出，我们在保存开始和结束位置的 logits 时是一条记录一条记录进行保存的（当然你也可以进行修改）；第 11-15 行是用训练好保存在本地的参数来重新初始化模型；第 20-21 行是得到根据评估函数 evaluate()返回每个原始样本下所有子样本的预测 logits 结果；第 22-23 行则是根据返回的 logits 根据图 8-3 和图 8-4 中的原理来筛选得到最终的预测结果。

### 8.4.3 结果筛选

对于整个推理过程来说，最重要的就是最后这一步结果筛选。由于这部分代码稍微有点长，所以掌柜同样也分块进行介绍。首先，我们需要在类 LoadSQuADQuestionAnsweringDataset 中再添加一个 write_prediction()方法用于将筛选后的预测结果写入到本地文件中。

```
1    def write_prediction(self,test_iter,all_examples,logits_data,output_dir):
2        qid_to_example_context = {}
3        for example in all_examples:
4            context = example[3]
5            context_list = context.split()
6            qid_to_example_context[example[0]] = context_list
7        _PrelimPrediction = collections.namedtuple(
8            "PrelimPrediction",
9            ["text", "start_index","end_index", "start_logit", "end_logit"])
10       prelim_predictions = collections.defaultdict(list)
```

在上述代码中，第 2-6 行用于获取得到每个 qid 对应的上下文 Token，这样后面根据 qid 便能获取得到每个问题对应上下文的 Token，类似于

```
1 {'5733be284776f41900661181':['Architecturally,', 'the', 'school', 'has', 'a',
  'Catholic', 'character.', 'Atop', 'the', 'Main', "Building's", 'gold'],...}
```

第 7-10 行则是分别定义一个命名体元组和默认 value 为列表的字典便于后续使用[20]。

进一步，我们遍历测试集中的每个输入特征（子样本），并取其对应的 logits 预测结果进行筛选，实现代码如下：

```
1        for b_input, _, _, b_qid, _, b_feature_id, b_map in
2                tqdm(test_iter, ncols=80, desc="正在遍历候选答案"):
3            all_logits = logits_data[b_qid[0]]
4            for logits in all_logits:
5                if logits[0] != b_feature_id[0]:
6                    continue
7                start_indexes=self.get_best_indexes(logits[1],
                                                     self.n_best_size)
```

```
8                 end_indexes=self.get_best_indexes(logits[2],self.n_best_size)
```

在上述代码中，第 1 行用来得到该问题 ID 对应的所有 logits 结果，因为有了滑动窗口所以原始一个 context 可以构造得到多个训练子样本。第 4-6 行是用来只取当前子样本对应的 logits 预测结果，非当前子样本对应的 logits 忽略。第 7-8 两行则是用来返回前 n_best_size 个概率值最大候选索引，也就是图 8-3 中的第②步。例如 start_index 的候选值可能是[23,45,33,24]，end_indexes 的候选值可能是[19,35,28,56]。

在得到多个候选的开始结束索引后，便需要对其按条件进行过滤和筛选，实现代码如下：

```
1               for start_index in start_indexes:
2                   for end_index in end_indexes:  # 遍历所有存在的结果组合
3                       if start_index >= b_input.size(0):
4                           continue  # 起始索引大于 token 长度，忽略
5                       if end_index >= b_input.size(0):
6                           continue  # 结束索引大于 token 长度，忽略
7                       if start_index not in b_map[0]:
8                           continue
9                       if end_index not in b_map[0]:
10                          continue
11                      if end_index < start_index:
12                          continue
13                      length = end_index - start_index + 1
14                      if length > self.max_answer_length:
15                          continue
```

在上述代码中，第 1-2 行两个 for 循环用来依次遍历每个[start_index,end_index]组合；第 3-6 行则是过滤掉开始和结束位置大于输入序列长度的情况。第 7-10 行则是用来过滤开始和结束位置不在当前子样本对应的映射表 b_map 中的情况，因为 b_map 中 key 是 b_input 里上下文 Token 在 b_input 中的索引。

例如对于如下结果来说：

```
1 input_tokens = ['[CLS]', 'what', 'is', 'in', 'front', 'of', 'the', 'notre',
  'dame', 'main', 'building', '[SEP]', 'character', '.', 'atop', 'the', ... ]
2 orig_map = {12: 6, 13: 6, 14: 7, 15: 8, 16: 9, ....}
```

orig_map 里 12:6 中的 12 表示单词 character 在 input_tokens 中的索引是 12，14: 7 中的 14 表示单词 atop 在 input_tokens 中的索引是 14，因此对于所有预测得到的开始和结束位置都应该在 orig_map 的 key 当中。

同时，第 11-12 行是用来过滤掉结束位置大于开始位置的情况；第 13-15 行则是用来过滤答案长度大于设定最大长度的情况。

在完成这一步之后，我们便可以根据筛选得到的候选结果取到对应的预测答案文本，代码如下：

```
                    token_ids = b_input.transpose(0, 1)[0]
                    strs = [self.vocab.itos[s] for s in token_ids]
                    tok_text=" ".join(strs[start_index:(end_index + 1)])
                    tok_text=tok_text.replace(" ##","").replace("##","")
                    tok_text = tok_text.strip()
                    tok_text = " ".join(tok_text.split())
                    orig_doc_start = b_map[0][start_index]
                    orig_doc_end = b_map[0][end_index]
                    orig_tokens = qid_to_example_context[b_qid[0]]
                                  [orig_doc_start:(orig_doc_end + 1)]
                    orig_text = " ".join(orig_tokens)
                    final_text=self.get_final_text(tok_text, orig_text)
                    prelim_predictions[b_qid[0]].append(
                       _PrelimPrediction(text=final_text,
                              start_index=int(start_index),
                              end_index=int(end_index),
                              start_logit=float(logits[1][start_index]),
                              end_logit=float(logits[2][end_index])))
```

在上述代码中，第 1-6 行用于根据预测得到的开始结束位置直接从 input_token 中获取最后的答案.第 7-11 行则是先根据预测得到的开始结束位置在 b_map 中取到对应在原始上下文中的开始结束位置，然后从原始的上下文中获取最后的答案。第 12 行则是根据 get_final_text 函数来对比两种方式获取的答案来返回最终的答案。第 13-18 行是将对应的结果以命名体元组的形式保存到字典中。

最后，只需要将保存到 prelim_predictions 中的结果进行排序然后写入到本地即可，代码如下：

```
        for k, v in prelim_predictions.items():
            prelim_predictions[k] = sorted(prelim_predictions[k],
                 key=lambda x: (x.start_logit + x.end_logit),reverse=True)
        best_results, all_n_best_results = {}, {}
        for k, v in prelim_predictions.items():
            best_results[k] = v[0].text  # 取最好的第一个结果
            all_n_best_results[k] = v  # 保存所有预测结果
        with open(os.path.join(output_dir, f"best_result.json"), 'w') as f:
            f.write(json.dumps(best_results, indent=4) + '\n')
        with open(os.path.join(output_dir, f"best_n_result.json"),'w') as f:
            f.write(json.dumps(all_n_best_results, indent=4) + '\n')
```

在上述代码中，第 1-3 行代码是对每个 qid 对应的所有预测答案按照 start_logit+end_logit 的大小进行排序.第 5-7 行是取最好的预测结果与最好的前 n 个预测结果（可用于分析）。第 8-11 行是将这两个结果写入到本地。

在得到预测结果后，只需要运行如下代码即可得到最终的评价结果：

```
python evaluate-v1.1.py dev-v1.1.json best_result.json
```

```
2
3 "exact_match": {80.879848628193, "f1": 88.338575234135}
```

到此，对于整个 SQuAD 任务的细节之处就介绍完了，不过掌柜依旧强烈建议各位客官在阅读完文章的同时再去看看整个项目的源码实现，会理解得更透彻。

总的来说，基于 BERT 模型的 SQuAD 问答任务从原理上来讲并不难，难就难在如何通过一些技巧来预处理样本，以及如何一步步地筛选得到最后的结果。好在掌柜已经替各位客官踩完了所有的坑，你们只需要跟着掌柜的足迹一步一步向前走便可。

## 8.5 样本过长问题

在做 NLP 的相关任务中，最常见的一个问题便是当输入序列过长时应该如何进行处理。例如在谷歌开源的预训练模型中，最大长度只支持 512 个 Token。对于这样的问题应该怎么进行处理呢？是简单的进行截断处理吗？

总的来说，对于这类问题可以有两种方式来进行解决：第 1 种是从模型入手，例如消除最大长度为 512 的限制；第 2 种是从输入入手，改变输入序列的长度重新构造任务。下面，掌柜就依次来介绍这两种方法的具体做法。

### 8.5.1 消除长度限制

根据第 2.3.2 节内容的介绍可知，原始的 BERT 网络模型模型只支持最大长度为 512 的 token 序列（包括[CLS]、[SEP]等特俗 token）。因此，除非是我们自己从零开始训练一个模型，否则如果是直接使用谷歌开源的预训练模型，那么这个词表的大小将会被限制在 512。 当然，我们依旧有办法可以突破这个限制，那就是重新初始化 Positional Embedding 中的向量，并将前 512 个向量用已有的进行替换，超出部分就使用随机初始化的权重在语料上进行微调或训练。这样，对于通过消除长度限制的办法来解决的输入样本过长的问题原理就算是清楚了，下面继续来看如何通过代码进行实现。

根据第 4.3.2 节中对方法 from_pretrained 的介绍可知，如果想要完成对于 Positional Embedding 层参数的修改，那么在逐层初始化参数的过程中就需要先找到这一层，然后再对其进行修改。因此，整体代码实现如下：

```
1     def replace_512_position(init_embedding, loaded_embedding):
2         logging.info(f"模型参数 max_positional_embedding> 512，采用替换处理！")
3         init_embedding[:512, :] = loaded_embedding[:512, :]
4         return init_embedding
5
6     @classmethod
7     def from_pretrained(cls, config, pretrained_model_dir=None):
8         model = cls(config)
9         pretrained_model_path = os.path.join(pretrained_model_dir,
                                               "pytorch_model.bin")
```

```
10          if not os.path.exists(pretrained_model_path):
11              raise ValueError(f"<模型:{pretrained_model_path}不存在! >")
12          loaded_paras = torch. load(pretrained_model_path)
13          state_dict = deepcopy(model.state_dict())
14          loaded_paras_names = list(loaded_paras.keys())[:-8]
15          model_paras_names = list(state_dict.keys())[1:]
16          for i in range(len(loaded_paras_names)):
17              logging.debug(f"## 成功将参数:{loaded_paras_names[i]}赋值给
                    {model_paras_names[i]},"
18                  f"参数形状为:{state_dict[model_paras_names[i]].size()}")
19              if "position_embeddings" in model_paras_names[i]:
20                  if config.max_position_embeddings > 512:
21                      new_embedding = replace_512_position(
22                                      state_dict[model_paras_names[i]],
23                                      loaded_paras[loaded_paras_names[i]])
24                      state_dict[model_paras_names[i]] = new_embedding
25                      continue
26              state_dict[model_paras_names[i]] =
                                          loaded_paras[loaded_paras_names[i]]
27          model.load_state_dict(state_dict)
28          return model
```

在上述代码中，第 1-4 行是定义一个函数来对 Positional Embedding 层中的参数进行替换。第 19 行是判断当前参数是否为 Positional Embedding 层中的参数。第 20 行是判断如果 max_position_embeddings 超过 512 时，则将网络模型中 Positional Embedding 层前 512 个向量用预训练模型中的 Positional Embedding 参数进行替换。

通过上述代码，我们在载入预训练模型的同时就成功将网络层中随机初始化 Positional Embedding 中的前 512 个向量替换为了预训练模型中 Positional Embedding 中的参数。在这之后，剩余部分的参数可以通过下游任务来进行微调，也可以在一些语料上根据 NSP 和 MLM 任务进行训练。

此时，我们可以通过以下方式来进行验证：

```
1 from model.BasicBert.Bert import BertModel
2 from model.BasicBert.BertConfig import BertConfig
3
4 if __name__ == '__main__':
5     json_file = '../bert_base_chinese/config.json'
6     config = BertConfig.from_json_file(json_file)
7     config.max_position_embeddings = 567
8     model = BertModel.from_pretrained(config,
9                                       pretrained_model_dir="../bert_base_chinese")
10    for param_tensor in model.state_dict():
```

```
11          print(param_tensor, "\t", model.state_dict()[param_tensor].size())
```

上述代码运行结束后输出结果如下：

```
1 bert_embeddings.position_ids      torch.Size([1, 567])
2 bert_embeddings.word_embeddings.embedding.weight    torch.Size([21128, 768])
3 bert_embeddings.position_embeddings.embedding.weight    torch.Size([567, 768])
4 bert_embeddings.token_type_embeddings.embedding.weight    torch.Size([2, 768])
5 bert_embeddings.LayerNorm.weight    torch.Size([768])
6 bert_embeddings.LayerNorm.bias    torch.Size([768])
```

从上述结果第 3 行可以看出，此时 Positional Embedding 层参数的形状已经变成了[567,768]。

到此，对于第 1 种通过消除模型长度限制来解决输入序列过长的方法就介绍完了。接下来我们继续来看如何从输入的角度来解决这一问题。

### 8.5.2 重构模型输入

所谓重构模型输入就是通过某种方式来改变原始的输入序列，使得其能够满足长度小于 512 的长度且同时能够完成原本的建模任务。例如最简单粗暴的方式便是直接将序列长度超过 512 的部分直接去掉保留前 512 个 Token 或者后 512 个 Token。当然，除此之外另外一种更常见的做法便是采用滑动窗口进行处理，例如我们在第 8.2 节所看到的 SQuAD 问题回答任务中的处理方式。下面掌柜就再以文本分类为例来介绍一下采用滑动窗口的处理过程

对于文本分类这个场景来说，我们可以将原始样本以滑动窗口的形式进行采样构造得到多个子样本；然后将这些子样本作为训练集来训练模型；最后在推理阶段同样采取这样的方式对原始样本进行处理，并选择各个子样本中概率值最大的标签作为原始样本的标签。

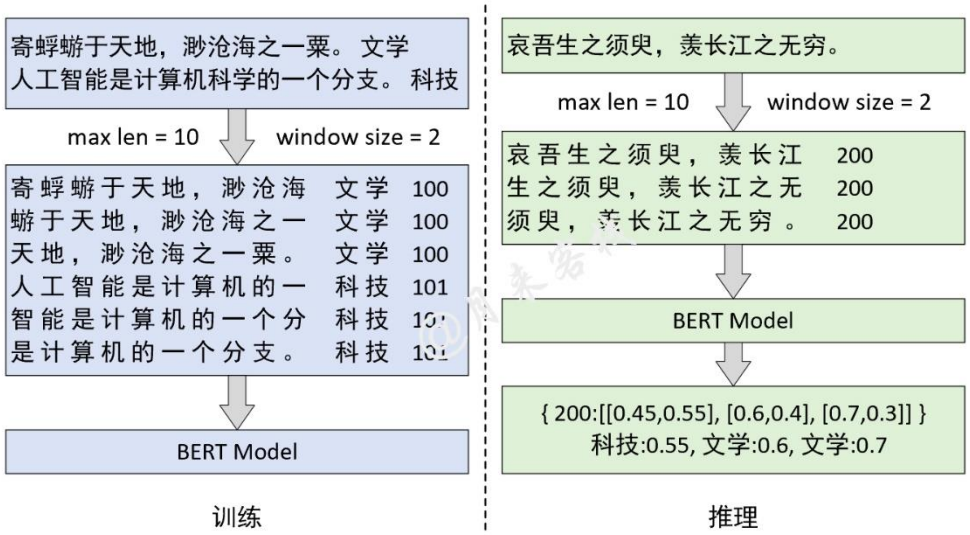



图 8-9. 文本分类滑动窗口处理流程

如图 8-9 所示便是整体的处理流程，其中左边为训练部分，右边为推理部分。在模型训练阶段时，需要先将每个原始样本按照固定长度和窗口进行滑动得到相

应的子样本。例如图 8-9 左边的原始样本“寄蜉蝣于天地，渺沧海之一粟”就重构成了 3 个子样本，并且标签也同原始样本。同时，为了区分不同原始样本之间的子样本，在构造子样本时还分别加上了原始样本对应的 ID。最后，将所有原始样本构造得到的子样本作为训练集来训练模型即可。

在推理阶段，对于每个原始样本来说同样要先按照训练集构造时的方式进行处理。在得到多个子样本后再分别将其进行分类并根据样本 ID 将同一个 ID 对应的所有分类结果放到一起，最后取概率最大的标签作为原始样本的预测结果即可。例如图 8-9 右边，“哀吾生之须臾，羡长江之无穷”这个原始样本就重构得到了 3 个子样本，其经过 BERT 分类模型后 3 个子样本分别被分进了“科技、文学、文学”这 3 个类别，最后可直接选择概率值最大的标签（文学:0.7）作为原始样本的预测值。

当然，除此之外还以将每个子样本对应的前 K 个概率值最大的预测结果都输出，然后再以集成模型的思想选择最终原始样本的预测结果。例如某个原始样本经过滑动窗口处理后得到了 5 个子样本，对于每个子样本我们都输出前 K 个概率值最大的预测结果；然后可以再通过以投票的方式来决定原始样本的预测类别。

如下所示便是一段简单的滑动窗口处理示例代码可供大家参考：

```
import numpy as np
np.random.seed(2021)
def data_process(samples, labels, window_size, max_len):
    data,uid = [],100
    for sample, label in zip(samples, labels):
        if len(sample) <= max_len:
            data.append([uid, sample, label])
            continue
        s_idx, e_idx = 0, max_len
        while True:
            s = sample[s_idx:e_idx]
            data.append([uid, s, label])
            if e_idx >= len(sample):
                break
            s_idx += window_size
            e_idx += window_size
        uid += 1
    return data

if __name__ == '__main__':
    samples = np.random.randint(0, 100, [5, 20])
    labels = [0, 0, 2, 1, 3]
    print(samples)
    data = data_process(samples, labels,window_size=6,max_len=10)
    print(data)
```

在没有使用滑动窗口处理前，此时 samples 的输出结果如下所示：

```
[[85 57  0 94 86 44 62 91 29 21 93 24 12 70 70 33  7  1 97 26]
 [66 48 99 63 49 16 50 54 52 93  5 49 38 14 71 85 70 41 21 25]
 [10 36 19 57 82 90 15 40 76 53 11 19 33 78 17 89 50  7 27 63]
 [51  9 25 71 84 27 75 27 19 31 50 89 27 18 53 32 20 95 87  3]
 [97 20 18 70 38 90 53 62 93 26 47 91 60  7 93 33 89 37 95 48]]
```

在经过 data_process()函数预处理之后则会变成：

```
[[100, array([85, 57,  0, 94, 86, 44, 62, 91, 29, 21]), 0],
 [100, array([62, 91, 29, 21, 93, 24, 12, 70, 70, 33]), 0],
 [100, array([12, 70, 70, 33,  7,  1, 97, 26]), 0],
 [101, array([66, 48, 99, 63, 49, 16, 50, 54, 52, 93]), 0],
 [101, array([50, 54, 52, 93,  5, 49, 38, 14, 71, 85]), 0],
 [101, array([38, 14, 71, 85, 70, 41, 21, 25]), 0],
 [102, array([10, 36, 19, 57, 82, 90, 15, 40, 76, 53]), 2],
 [102, array([15, 40, 76, 53, 11, 19, 33, 78, 17, 89]), 2], ...]
```

到此，对于 BERT 中文本过长时的两种处理方法掌柜就介绍完毕了。在下一节内容中，掌柜将会详细介绍如何在 PyTorch 中来使用 Tensorboard 这一利器，以便后续更加方便的观察 BERT 的训练过程。

# 第 9 节 PyTorch 中使用 Tensorboard

在网络模型的训练过程中，通常我们都需要观察其损失值或准确率的变化趋势来确定模型的优化方向，例如学习率的动态调整、惩罚项系数等等。同时，对于图像处理方向的朋友来说可能还希望能够可视化模型的特征图或者是样本分类类别在空间中的分布情况等。虽然这些结果我们可以在网络训练结果后取对应的变量通过 matplotlib 进行可视化，但是我们更希望的是在模型训练的过程中就对其各种状态进行可视化。

因此，对于上述需求我们可以借助谷歌开源的 Tensorboard 工具来进行实现。在本篇文章中，掌柜将会详细介绍如何在 PyTorch 中通过 Tensorboard 来对各类变量及指标进行可视化。

## 9.1 安装与调试

如果需要在 PyTorch 中使用 Tensorboard 除了需要安装 Tensorboard 工具本身之外，还需要安装的便是 TensorFlow 本身。因为 Tensorboard 在使用中会依赖于 TensorFlow 框架。

### 9.1.1 安装启动

对于 TensorFlow 和 Tensorboard 的安装，我们只需要执行安装 TensorFlow 的命令便可以同时完成两者的安装：

```
1 pip install tensorflow
```

同时，由于我们只是借助于 Tensorboard 来进行可视化，因此在安装 TensorFlow 的时候不用区分是 GPU 还是 CPU 版本的，掌柜测试了都可以。也就是说假如你在某台主机上装了 GPU 版本的 PyTorch，而不管你是装的 GPU 版还是 CPU 版的 TensorFlow，Tensorboard 都可以正常使用。

在安装成功之后通过可以通过如下命令来进行测试：

```
1 tensorboard --logdir=runs
```

并会出现如下提示：

```
1 TensorBoard 1.15.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

也就是说此时我们便可以通过 127.0.0.1:6006 这个链接来访问 Tensorboard 的可视化页面，如图 9-1 所示。

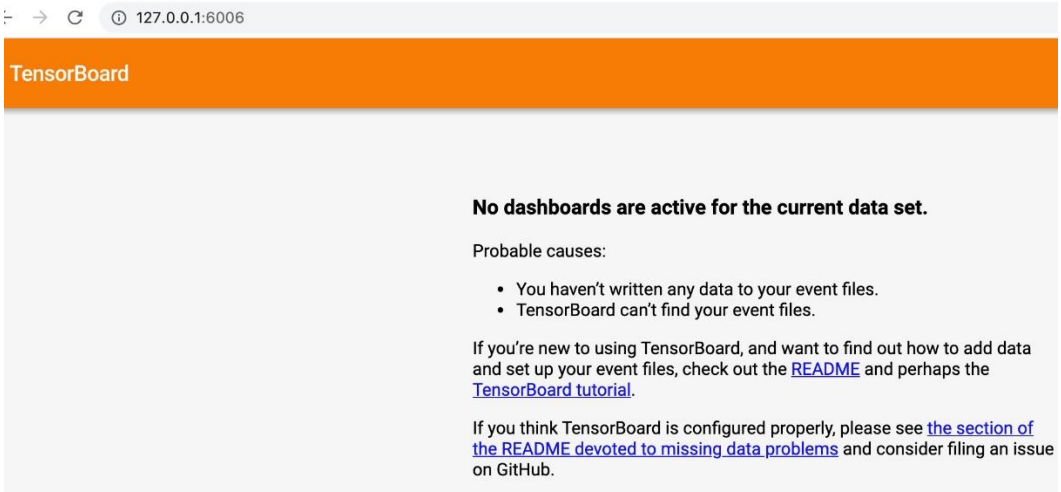



图 9-1. Tensorboard 启动成功图

如果你发现打不开这里地址，那么可以尝试通过如下命令来进行启动，然后再通过 127.0.0.1:6006 这个链接来访问。

```
1 tensorboard --logdir=runs --host 0.0.0.0
```

命令中的--logdir 用来指定可视化文件的目录地址，后续掌柜会详细介绍。

### 9.1.2 远程连接

上面掌柜介绍了如何在本地安装与启动 Tensoboard，但是更常见的一种场景便是在远程主机上运行代码，但是需要在本地电脑上查看可视化的运行结果。如果需要实现这种目的通常来说有两种方法，下面掌柜就来分别进行介绍。

**(1) 通过 IP 直接访问**

在通过 IP 直接访问的方案中，不管你是在类似于腾讯云或阿里云上租用的主机还是实验室的专用主机，在根据 9 第.1.1 节中的步骤完成 Tensoboard 安装并启动后，你在自己电脑上都可以通过主机 IP 来进行访问，例如：

```
1 http://10.67.25.113:6006
```

需要注意的是，上面的 IP 对于公网主机（如腾讯云）来说指的是主机的公网 IP，对于实验室或学校的主机来说指的则是局域网的内网 IP。同时，如果在主机上启动 Tensoboard 后发现在本地并不能够打开，那么可以通过如下方式来进行排查：

**情况一：互联网主机**

- 在后台的安全策略里面查看一下 6006 这个端口有没有被打开，如果没有则需要打开；
- 查看 IP 是否为公网 IP，在主机的后台管理页面可以看到。

**情况二：局域网主机**

- 查看本地电脑是否和主机处于同一网段；

➢ 查看主机的 6006 端口是否打开，如果没有可以通过如下命令打开

```
1 firewall-cmd --zone=public --list-ports # 查看已开放端口
2 firewall-cmd --zone=public --add-port=5672/tcp --permanent #开放 5672 端口
3 firewall-cmd --zone=public --remove-port=5672/tcp --permanent #关闭 5672
4 firewall-cmd --reload   # 配置立即生效
```

**(2) SSH 转发访问**

当然，除了通过直接 IP 访问之外，我们还可以借助 SSH 反向隧道技术来进行访问，例如服务器只开了 22 端口而且你没有权限打开其它端口的情况。在这种情况下，你可以通过下面两种方式来进行远程连接：

**方式一：命令行终端**

如果你的命令行终端支持 SSH 命令（例如较新的 Windows10 的 CMD 或者 Linux 等）的话，那么可以直接通过下面这一条命令来进行连接：

```
1 ssh -L 16006:127.0.0.1:6006 username@ip
```

这条命令的含义就是将服务器上 6006 端口的信息通过 SSH 转发到本地的 16006 端口，其中 16006 是本地的任意端口（无限制），只要不和本地应用有冲突就行；后面则是对应的用户名和 IP。

上述命令连接成功并根据 9.1.1 节中的命令在远程主机上启动 Tensorboard 之后，在本地通过浏览器打开如下地址即可：

```
1 http://127.0.0.1:16006
```

**方式二：Xshell 工具**

如果你的电脑终端不支持 SSH 命令的话，那么你还可以通过 Xshell 工具来实现 SSH 反向代理。首先你需要安装好 Xshell 工具，安装完成后再按照如下步骤配置。

**第 1 步：新建连接**

如图 2 所示，点击新建连接。


图 9-2. 新建连接（一）

然后再根据图 9-3 所示的界面配置主机信息。

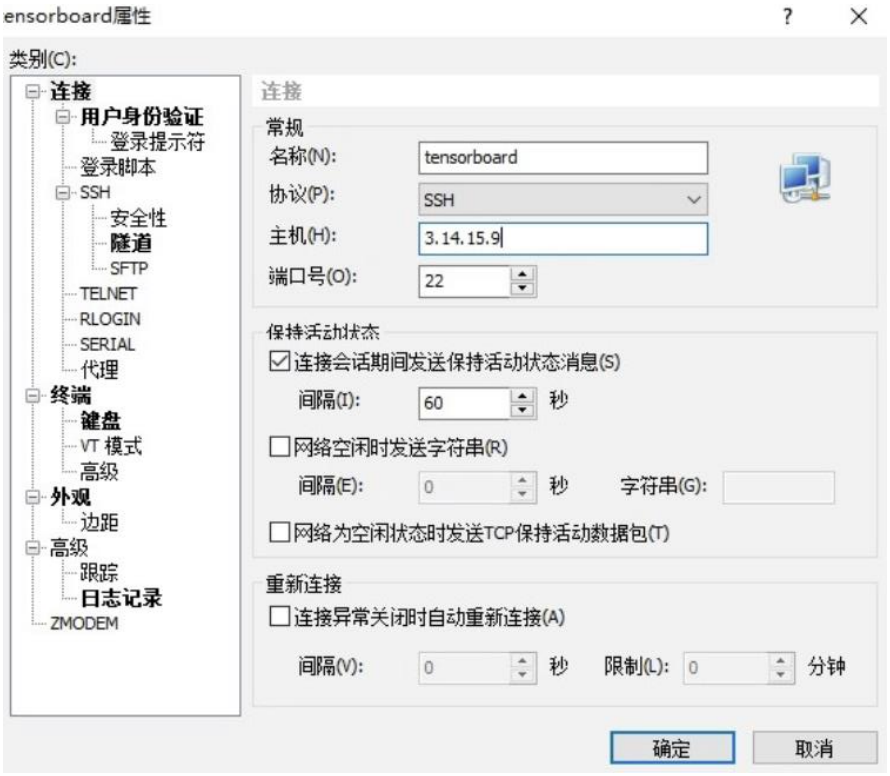



图 9-3. 新建连接（二）

## 第 2 步：配置代理

进一步，如图 9-4 所示点击侧边栏的隧道，并点击右侧的添加按钮。

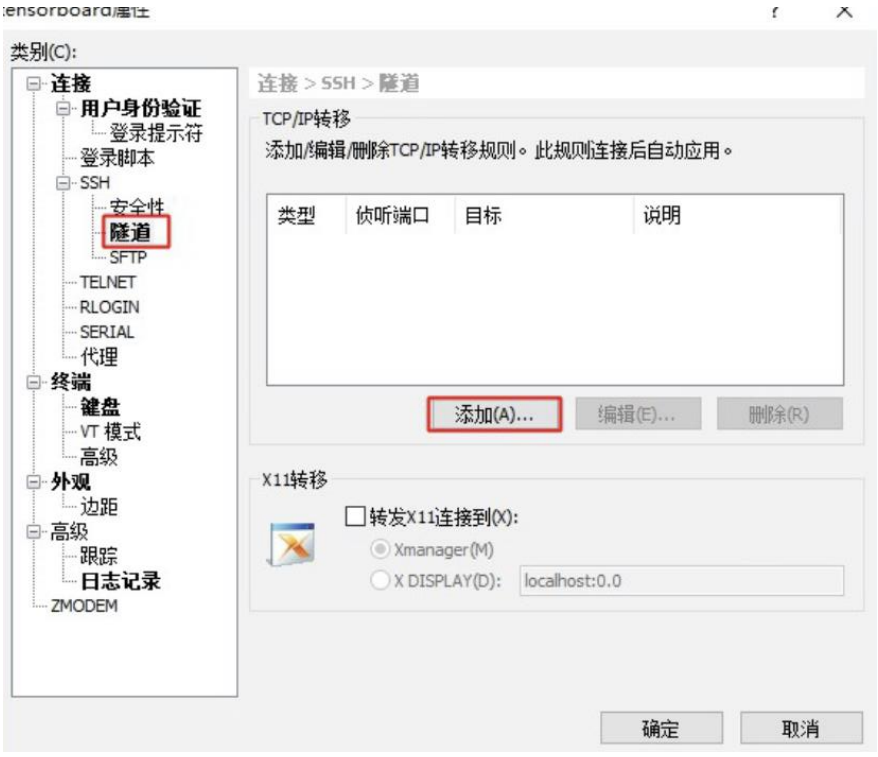



图 9-4. 配置代理（一）

接着再根据图 9-5 所示的示例进行端口代理配置。

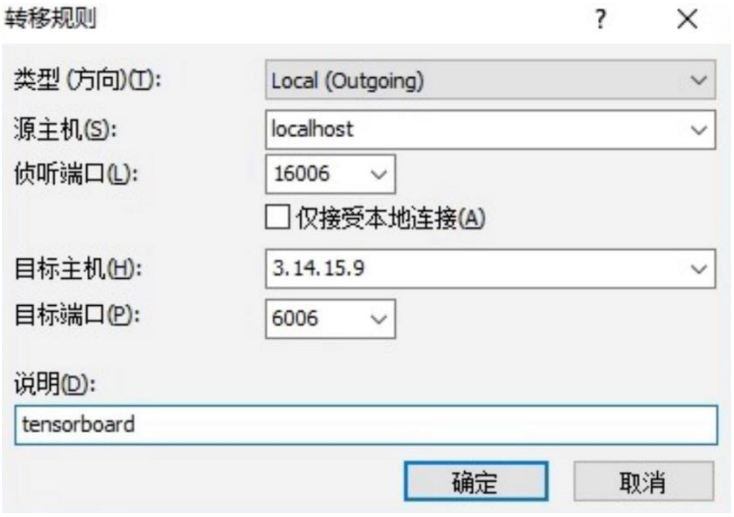



图 9-5. 配置代理（二）

配置完成后点击图 9-6 中的确认即可。

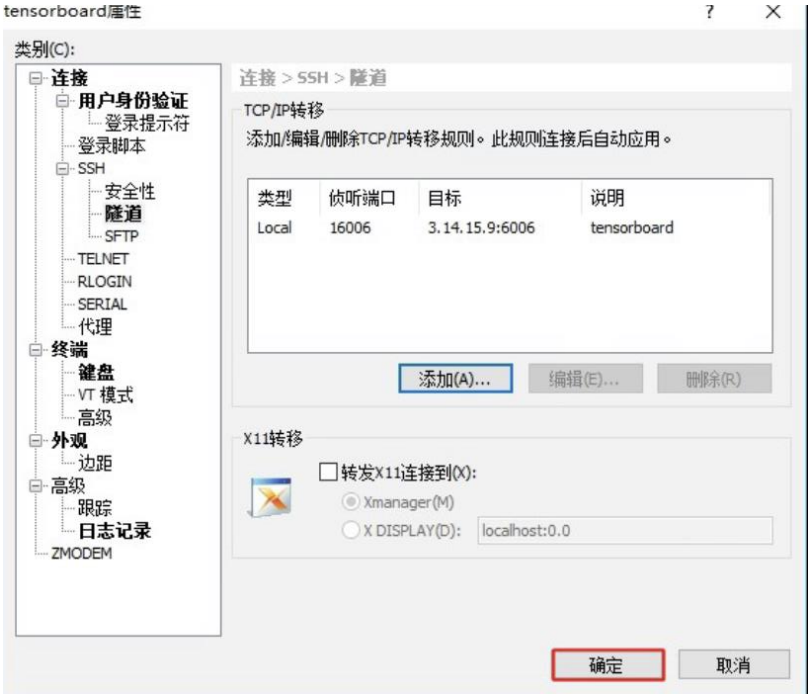



图 9-6. 配置代理（三）

完成上述两步配置之后，再双击刚刚这个新建的连接输入用户名和密码之后即可登录到主机并对相应的端口进行了监听与转发。最后我们同样只需要先在当前的命令行中启动 Tensorboard，再在本地浏览器中通过如下地址进行访问即可：

```
1 http://127.0.0.1:16006
```

## 9.2 使用 Tensoboard

下面，掌柜将先通过一个简单的示例来展示如何可视化标量数据并在 Tensoboard 中进行显示；然后再来依次介绍一些常用的可视化方法。本节所有示例代码可在[12]处获取。

### 9.2.1 add_scalar 方法

这个方法通常用来可视化网络训练时的各类标量参数，例如损失、学习率和准确率等。如下便是 add_scalar 方法的使用示例：

```
1 from torch.utils.tensorboard import SummaryWriter
2 if __name__ == '__main__':
3     writer = SummaryWriter(log_dir="runs/result_1", flush_secs=120)
4     for n_iter in range(100):
5         writer.add_scalar(tag='Loss/train',
6                           scalar_value=np.random.random(),
7                           global_step=n_iter)
8         writer.add_scalar('Loss/test', np.random.random(), n_iter)
9     writer.close()
```

在上述代码中，第 1 行用来导入相关的可视化模块；第 3 行是实例化一个可视化类对象，log_dir 用于指定可视化数据的保存路径，flush_secs 表示指定多少秒将数据写入到本地一次（默认为 120 秒）；第 5-7 行则是利用 add_scalar 方法来对相关标量进行可视化，其中 tag 表示对应的标签信息。

在上述代码运行之前，先进入到该代码文件所在的目录，然后运行如下命令来启动 Tensoboard：

```
1 tensorboard --logdir=runs
2
3 TensorBoard 1.15.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

可以看出，logdir 后面的参数就是上面代码第 3 行里的参数。同时，根据提示在浏览器中打开上述链接便可以看到如图 9-1 所示的界面。

接着开始运行上述程序，此时便会在当前目录中生成如图 9-7 所示的文件（夹），其中 result_1 便是前面所指定的子目录，而以 events.out 开始的文件则是生成的相关可视化事件。

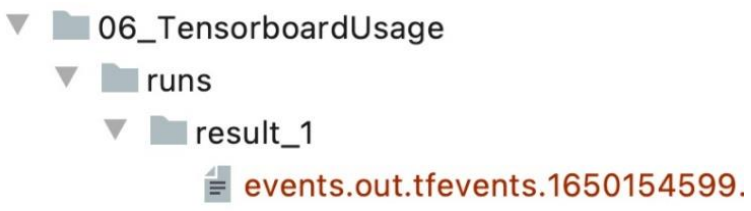



图 9-7. 可视化数据文件图

当程序运行时 Tensoboard 便会加载图 9-7 中所示的文件并在网页端进行渲染，如图 9-8 所示。

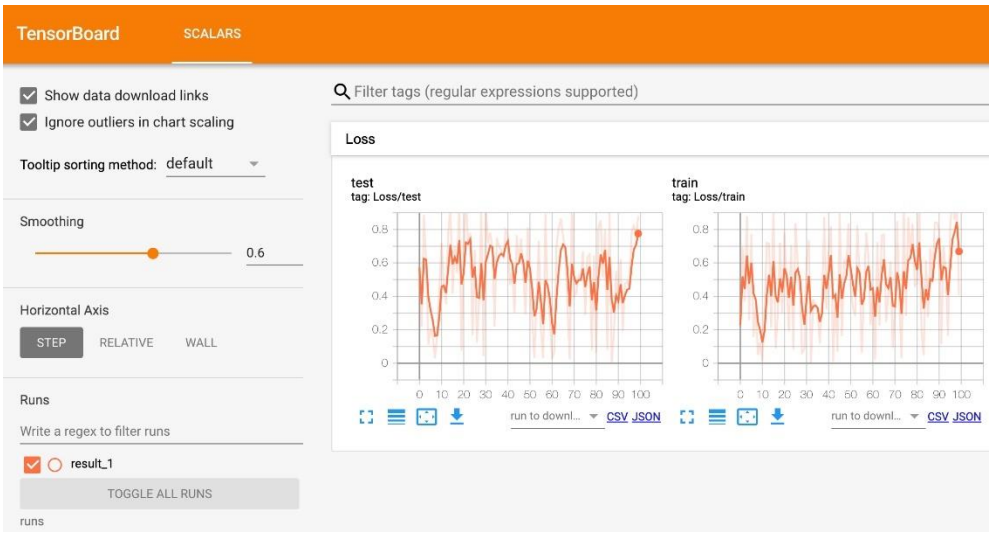



图 9-8. Tensoboard 可视化结果图

如图 9-8 所示为 Tensoboard 的可视化结果图，其中右边的 Loss 标签就是上面第 5 行代码中指定的 Loss/train 参数的前缀部分，也就是说如果想把若干个图放到一个标签下，那么就要保持其前缀一致，例如这里的 Loss/train 和 Loss/test 这两个图都将被放在 Loss 这个标签下。同时，在勾选左上角的"Show data download links"后，还能点击图片下方的按钮来分别下载 SVG 矢量图、原始图片的 CSV 或 JSON 数据。

在图 9-8 的左边部分，Smoothing 参数用来调整右侧可视化结果的平滑度；Horizontal Axis 用来切换不同的显示模式；Runs 下面用来勾选需要可视化的结果，例如后续在初始化 SummaryWriter()时指定 log_dir="runs/result_2"，那么在 result_1 下方便会再出现一个 result_2 的选项，这时我们可以选择多个结果同时可视化展示。

下面，掌柜开始逐一介绍几个常用的可视化方法。

## 9.2.2 add_graph 方法

从名字可以看出 add_graph 方法是用于可视化模型的网络结构图，其用法示例如下：

```
1 import torchvision
2 def add_graph(writer):
3     img = torch.rand([1, 3, 64, 64], dtype=torch.float32)
4     model = torchvision.models.AlexNet(num_classes=10)
5     writer.add_graph(model, input_to_model=img) #类似于 TensorFlow 1.x 中的 feed
```

为了示例简洁，掌柜这里又把 SummaryWriter()中的 add_graph 方法写成了一个函数。在上述代码中，第 4 行用于返回一个网络模型；第 5 行则是对网络结构图进行可视化，其中 input_to_model 参数为模型所接收的输入，这类似于 TensorFlow 1.x 版本中的 feed_dict 参数。

上述代码开始运行之后，便可以在网页端看到如图 9-9 所示的可视化结果。

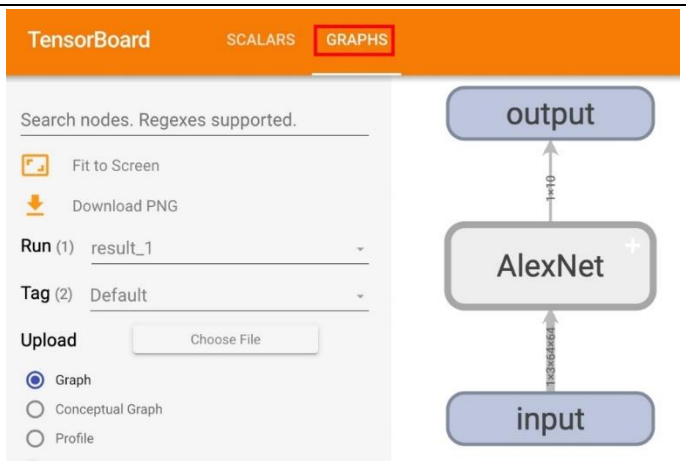



图 9-9. 网络结构图

如图 9-9 所示便是可视化后的网络结构图，对于右侧网络结构中的每个模块都可以双击进行展开，而左边则是相关模式的切换。

### 9.2.3 add_scalars 方法

这个方法与 add_scalar 的差别在于 add_scalars 在一张图中可以绘制多个曲线，此时只需我们要以字典的形式传入需要可视化的参数即可，如下为 add_scalars 方法的使用示例：

```
def add_scalars(writer):
    r = 5
    for i in range(100):
        writer.add_scalars(main_tag='scalars1/P1',
                           tag_scalar_dict={'xsinx': i  np.sin(i / r),
                                            'xcosx': i  np.cos(i / r),
                                            'tanx': np.tan(i / r)},
                           global_step=i)
        writer.add_scalars('scalars1/P2',
                           {'xsinx': i  np.sin(i / (2  r)),
                            'xcosx': i  np.cos(i / (2  r)),
                            'tanx': np.tan(i / (2  r))}, i)
        writer.add_scalars(main_tag='scalars2/Q1',
                           tag_scalar_dict={'xsinx':i  np.sin((2  i) / r),
                                            'xcosx':i  np.cos((2  i) / r),
                                            'tanx':np.tan((2  i) / r)},
                           global_step=i)
        writer.add_scalars('scalars2/Q2',
                           {'xsinx': i  np.sin(i / (0.5  r)),
                            'xcosx': i  np.cos(i / (0.5  r)),
                            'tanx': np.tan(i / (0.5  r))}, i)
```

在上述代码中，掌柜一共画了 4 个图，分别对应代码中的 4 个 add_scalars；同时在每张图里面都都对应了 3 条曲线，也即 add_scalars 方法里的 tag_scalar_dict 参数；并且掌柜这里一共用了 2 个标签来进行分隔，即 scalars1 和 scalars2。可视化的结果如图 9-10 所示。

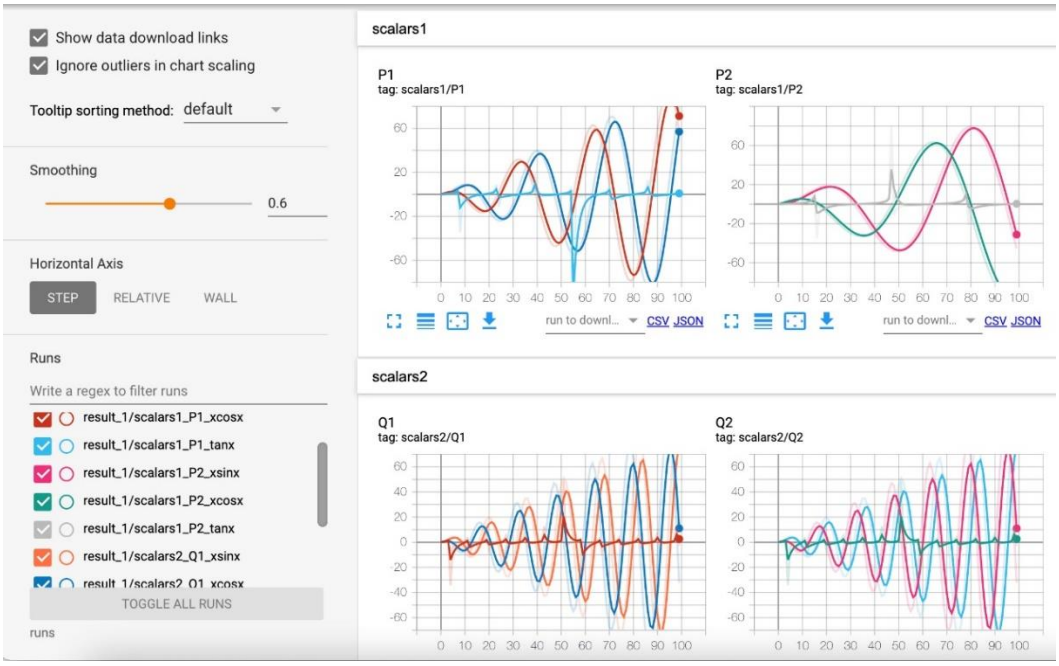



图 9-10. add_scalars 可视化结果图

## 9.2.4 add_histogram 方法

直方图一般被用于可视化网络层中参数的分布情况，其使用示例也比较简单，代码如下所示：

```
1 def add_histogram(writer):
2     for i in range(10):
3         x = np.random.random(1000)
4         writer.add_histogram('distribution centers/p1', x + i, i)
5         writer.add_histogram('distribution centers/p2', x + i  2, i)
```

上述代码运行结束后，其可视化后结果如图 9-11 所示。

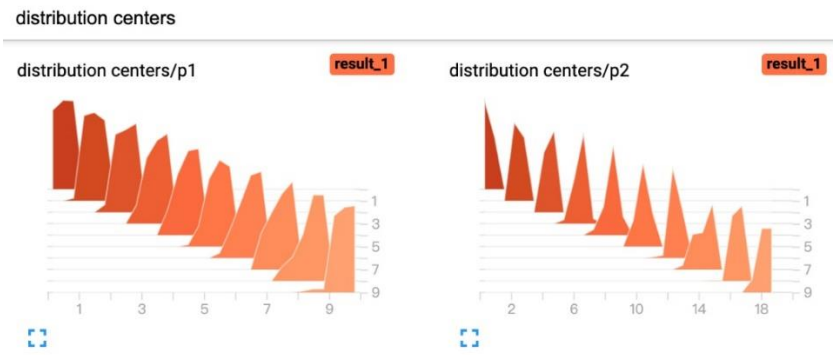



图 9-11. add_histogram 可视化结果图

### 9.2.5 add_image 方法

add_image 方法通常是用来可视化相应的像素矩阵，例如本地图片，或者是特征图等，其示例代码如下所示：

```
1 def add_image(writer):
2     from PIL import Image
3     img1 = np.random.randn(1, 100, 100)
4     writer.add_image('img/imag1', img1)
5     img2 = np.random.randn(100, 100, 3)
6     writer.add_image('img/imag2', img2, dataformats='HWC')
7     img = Image.open('./dufu.png')
8     img_array = np.array(img)
9     writer.add_image('local/dufu', img_array, dataformats='HWC')
```

在上述代码中，第 3-4 行用于生成一个形状为[C,H,W]的 3 维矩阵并进行可视化；第 5-6 行则是生成形状为[H,W,C]的 3 维矩阵并可视化，同时需要在 add_image 中指定矩阵的维度信息，因此可以看出 add_image 方法接受的默认格式为[C,H,W]；第 7-9 行则是先从本地读取一张图片，然后再对其进行可视化，如图 9-12 所示。

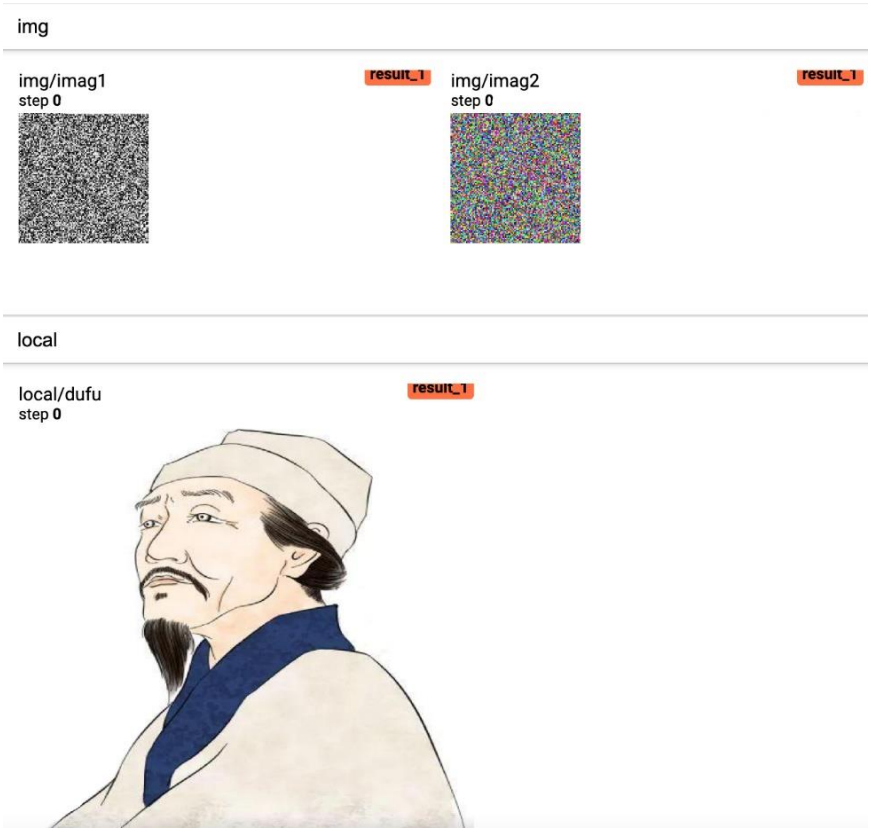



图 9-12. add_image 可视化结果图

如图 9-12 所示，上面部分则是两个大小为100×100的像素矩阵可视化结果。下面部分则是本地图片的可视化结果。

### 9.2.6 add_images 方法

从名字可以看出，该方法是一次性可视化多张像素矩阵图，使用示例如下：

```
1 def add_images(writer):
2     img1 = np.random.randn(8, 100, 100, 1)
3     writer.add_images('imgs/imags1', img1, dataformats='NHWC')
4     img2 = np.zeros((16, 3, 100, 100))
5     for i in range(16):
6         img2[i, 0] = np.arange(0, 10000).reshape(100, 100) / 10000 / 16  i
7         img2[i, 1] = (1 - np.arange(0, 10000).reshape(100, 100)/10000)/16  i
8     writer.add_images('imgs/imags2', img2)  # Default is :math:(N, 3, H, W)
```

在上述代码中，第 2-3 行用于生成 8 张通道数为 1 的像素矩阵并进行可视化。第 4-8 行则是生成 16 张通道数为 3 的像素矩阵并进行可视化。最后可视化的结果如图 9-13 所示。

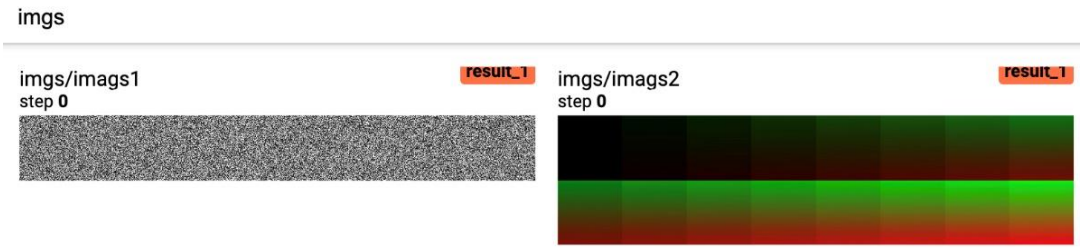



图 9-13. add_images 可视化结果图

如图 9-13 所示，左边是 8 张通道数为 1 的像素矩阵可视化结果。右边是 16 张通道数为 3 的像素矩阵可视化结果。

### 9.2.7 add_figure 方法

这个方法的作用是用来将 matplotlib 包中的 figure 对象可视化到 Tensoboard 的网页端，用于展示一些较为复杂的图片，其示例用法如下：

```
1 def add_figure(writer):
2     fig = plt.figure(figsize=(5, 4))
3     ax = fig.add_axes([0.12, 0.1, 0.85, 0.8])
4     xx = np.arange(-5, 5, 0.01)
5     ax.plot(xx, np.sin(xx), label="sin(x)")
6     ax.legend()
7     fig.suptitle('Sin(x) figure\n\n', fontweight="bold")
8     writer.add_figure("figure", fig, 4)
```

在上述代码中，第 2-7 行为根据 matplotlib 包绘制相应的图像，其中第 3 行用来指定；第 8 行则是将其在 Tensoboard 中进行可视化；第 9 行则是直接在程序里进行可视化。最后可视化的结果如图 9-14 所示。

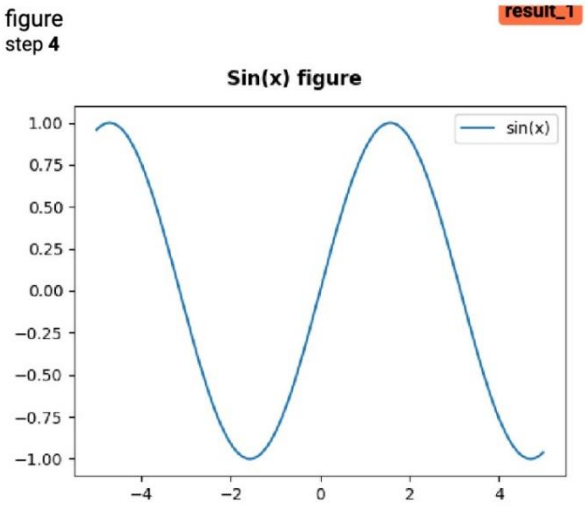



图 9-14. add_figure 可视化结果图

如果要一次在 Tensoboard 中可视化一组图像的话，可以通过如下方式实现：

```
1 def add_figures(writer, images, labels):
 2     text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
 3                    'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
 4     labels = [text_labels[int(i)] for i in labels]
 5     fit, ax = plt.subplots(len(images)//5,5,figsize=(10,2len(images)//5))
 6     for i, axi in enumerate(ax.flat):
 7         image, label = images[i].reshape([28, 28]).numpy(), labels[i]
 8         axi.imshow(image)
 9         axi.set_title(label)
10         axi.set(xticks=[], yticks=[])
11     writer.add_figure("figure", fit)
```

在上述代码中，掌柜选择是的 FashionMNIST 数据集进行的可视化；第 5 行代码用来生成一个包含有若干个子图的画布；第 6-10 行是分别用来画出每一个子图；第 10 行用来去掉横纵坐标的信息；第 11 行则是将其在 Tensoboard 中进行展示。最终可视化后的结果如图 9-15 所示。


图 9-15. add_figure 可视化结果图

### 9.2.8 add_pr_curve 方法

add_pr_curve 这个方法是用来在训练过程中可视化 Precision-Recall 曲线，即观察在不同阈值下精确率与召回率的平衡情况。更多关于 Precision-Recall 曲线内容的介绍可以参考文章[22]。用法示例如下所示：

```
1 def add_pr_curve(writer):
2     from sklearn.linear_model import LogisticRegression
3     from sklearn.preprocessing import label_binarize

4     def get_dataset():
5         from sklearn.datasets import load_iris
6         from sklearn.model_selection import train_test_split
7         x, y = load_iris(return_X_y=True)
8         random_state = np.random.RandomState(2020)
9         n_samples, n_features = x.shape
10        # 为数据增加噪音维度以便更好观察 pr 曲线
11        x = np.concatenate([x, random_state.randn(
12                        n_samples, 100  n_features)], axis=1)
13        x_train, x_test, y_train, y_test = train_test_split(
14              x, y, test_size=0.5,random_state=random_state)
15        return x_train, x_test, y_train, y_test
16
17    x_train, x_test, y_train, y_test = get_dataset()
18    model = LogisticRegression(multi_class="ovr")
19    model.fit(x_train, y_train)
20    y_scores = model.predict_proba(x_test)  # shape: (n,3)
21
22    b_y = label_binarize(y_test, classes=[0, 1, 2])  # shape: (n,3)
23    for i in range(3):
24        writer.add_pr_curve(f"pr_curve/label_{i}", b_y[:, i],
25                            y_scores[:, i], global_step=1)
```

在上述代码中，第 2-19 行代码用来根据逻辑回归生成预测结果，其中第 11 行用来给原始数据加入噪音，目的是为了可视化得到更加真实的 PR 曲线；第 21 行用来将原始标签转化为 one-hot 编码形式的标签；第 22-23 行则是分别根据每个类别的预测结果画出对应的 PR 曲线。运行上述代码将会得到类似如图 9-16 所示的结果。

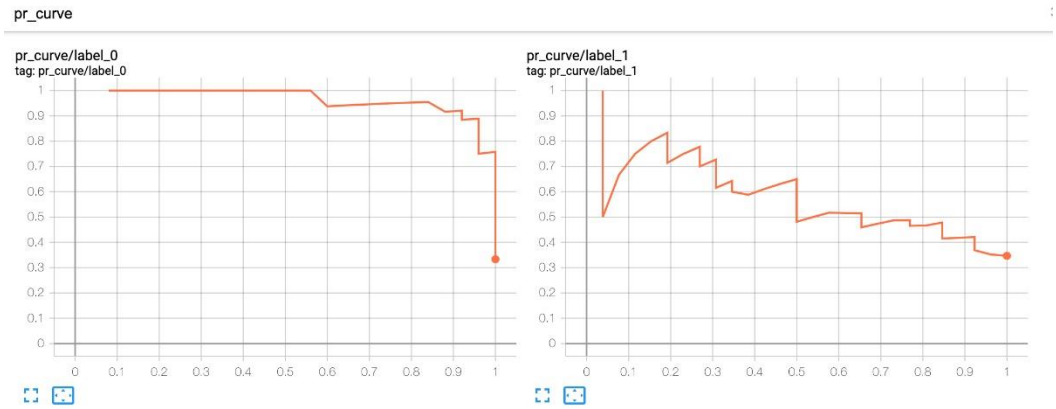



图 9-16. add_pr_curve 可视化结果图

## 9.2.9 add_embedding 方法

这个方法作用是在三维空间中对高维向量进行可视化，默认情况下是对高维向量以 PCA 方法进行降维处理。add_embedding()方法主要有三个比较重要的参数 mat、metadata 和 label_img，下面掌柜依次来进行介绍。

mat：用来指定可视化结果中每个点的坐标，形状为$(N,D)$，不能为空，例如对词向量可视化时 mat 就是词向量矩阵，图片分类时 mat 可以是分类层的输出结果；

metadata：用来指定每个点对应的标签信息，是一个包含 $N$ 个元素的字符串列表，为空时则默认为['1','2',....,'N'];

label_img：用来指定每个点对应可视化信息，形状为$(N,C,H,W)$，可以为空，例如图片分类时 label_img 就是每一张真实图片的可视化结果。

进一步，我们便可以通过如下代码来进行 3 维空间的高维向量可视化：

```
def add_embedding(writer):
    import tensorflow as tf
    import tensorboard as tb
    tf.io.gfile = tb.compat.tensorflow_stub.io.gfile
    import keyword
    import torch
    # 随机生成 100 个标签信息
    meta = []
    while len(meta) < 100:
        meta = meta + keyword.kwlist  # get some strings
    meta = meta[:100]
    for i, v in enumerate(meta):
        meta[i] = v + str(i)
    # 随机生成 100 个标签图片
    label_img = torch.rand(100, 3, 10, 32)
    for i in range(100):
        label_img[i] = i / 100.0
```

```
18      data_points = torch.randn(100, 5)  # 随机生成 100 个点
19      writer.add_embedding(mat=data_points, metadata=meta,
20          label_img=label_img, global_step=1)
```

在上述代码中，第 2-4 行用于解决 TensorFlow1.x 版本的兼容性问题；第 8-13 行则是随机生成 100 个字符串标签信息；第 15-17 行则是生成标签对应的图片；第 18 行则是随机生成需要可视化的高维向量。上述代码运行结束后便会得到如图 9-17 所示的结果。

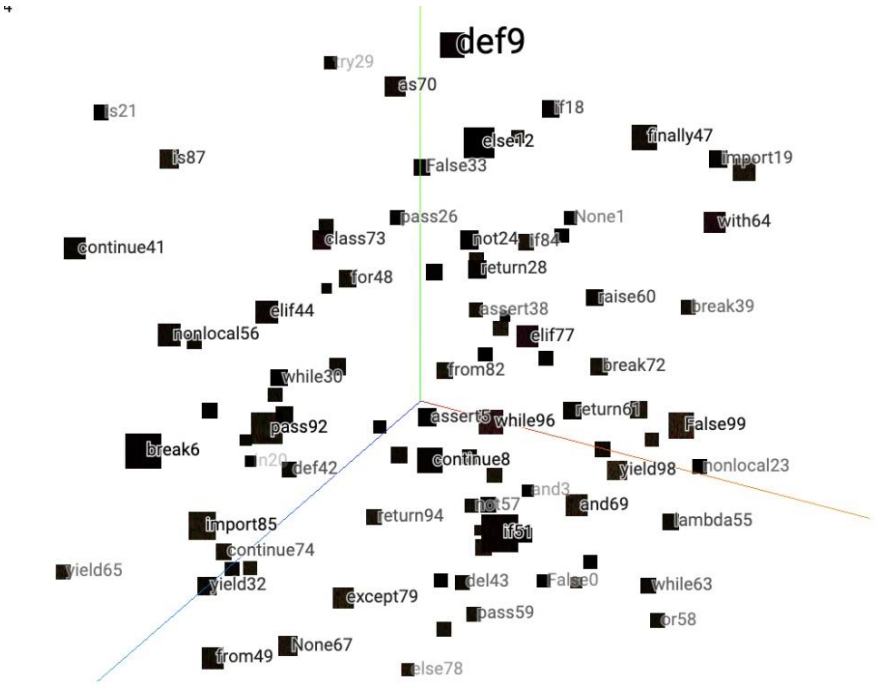



图 9-17. add_embedding 可视化结果图

如图 9-17 所示，字符串就是上面代码中对应的 metadata 参数，黑色方块就是对应的 label_img 参数，而方块背后的点（图中看不到）就是对应的 mat 参数。这里掌柜只是用了随机数据生成了上面这张图，在下一小节中掌柜将会用一个实际的例子来进行展示。

## 9.3 使用实例

### 9.3.1 定义模型

这里，掌柜以 LeNet5 网络模型进行示例，所以首先需要定义模型的前向传播过程，代码如下：

```
1 import torch.nn as nn
2
3 class LeNet5(nn.Module):
4     def __init__(self, ):
5         super(LeNet5, self).__init__()
```

```
6           self.conv = nn.Sequential(
7               nn.Conv2d(1, 6, 5, padding=2),
8               nn.ReLU(),
9               nn.MaxPool2d(2, 2),
10              nn.Conv2d(6, 16, 5),
11              nn.ReLU(),
12              nn.MaxPool2d(2, 2))
13
14          self.fc = nn.Sequential(
15              nn.Flatten(),
16              nn.Linear(16  5  5, 120),
17              nn.ReLU(),
18              nn.Linear(120, 84),
19              nn.ReLU(),
20              nn.Linear(84, 10))
21
22      def forward(self, img, labels=None):
23          output = self.conv(img)
24          logits = self.fc(output)
25          if labels is not None:
26              loss_fct = nn.CrossEntropyLoss(reduction='mean')
27              loss = loss_fct(logits, labels)
28              return loss, logits
29          else:
30              return logits
```

在上述代码中，第 25-30 行为根据不同的输入情况返回不同的结果。其余部的代码相对较为简单掌柜这里就不再赘述了。

## 9.3.2 定义分类模型

**第 1 步：数据集构造**

首先，我们需要构造训练模型时使用到的数据集，这里掌柜还是以 FashionMNIST 为例进行示例。构造代码如下所示：

```
1 text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
2                'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
3
4 def load_dataset(batch_size=64):
5     mnist_train = torchvision.datasets.FashionMNIST(download=True,
6                           root='~/Datasets/FashionMNIST',train=True,
7                                   transform=transforms.ToTensor())
8     mnist_test = torchvision.datasets.FashionMNIST(download=True,
9                         root='~/Datasets/FashionMNIST',train=False,
```

```
10                                      transform=transforms.ToTensor())
11     train_iter = torch.utils.data.DataLoader(mnist_train,
12                   batch_size=batch_size,shuffle=True,num_workers=1)
13     test_iter = torch.utils.data.DataLoader(mnist_test,
14                   batch_size=batch_size,shuffle=True,num_workers=1)
15     return train_iter, test_iter
```

在上述代码中，第 1-2 行为定义的每个标签序号所对应的标签名，用于在使用 add_embedding 可视化预测结果时展示每个样本的标签名称；第 5-15 行则是分别构造训练集和测试集的 DataLoader 对象。

**第 2 步：定义分类模型**

进一步，我们需要定义一个类来实现模型的整个训练和推理过程。首先定义这个类的初始化方法，代码如下：

```
1 class MyModel:
2     def __init__(self,
3                  batch_size=64,
4                  epochs=3,
5                  learning_rate=0.01):
6         self.batch_size = batch_size
7         self.epochs = epochs
8         self.learning_rate = learning_rate
9         self.model_save_path = 'model.pt'
10        self.device = torch.device('cuda:0'
11                    if torch.cuda.is_available() else 'cpu')
12        self.model = LeNet5()
```

**第 2 步：定义评估方法**

在这里，我们先定义一个评估函数，用来计算在测试集上模型的准确率。同时返回我们在使用 Tensoboard 可视化时需要用到的相关变量，代码如下：

```
1     @staticmethod
2     def evaluate(data_iter, net, device):
3         net.eval()
4         all_logits = []
5         y_labels = []
6         images = []
7         with torch.no_grad():
8             acc_sum, n = 0.0, 0
9             for x, y in data_iter:
10                x, y = x.to(device), y.to(device)
11                logits = net(x)
12                acc_sum += (logits.argmax(1) == y).float().sum().item()
13                n += len(y)
```

```
                all_logits.append(logits)
                y_pred = logits.argmax(1).view(-1)
                y_labels += (text_labels[i] for i in y_pred)
                images.append(x)
            net.train()
            return acc_sum / n, torch.cat(all_logits, dim=0),
                                y_labels, torch.cat(images, dim=0)
```

在上述代码中，第 1 行表示将这个方法声明为静态方法，因为函数里面没有使用到相关类成员；第 4-6 行分别定义了 3 个列表用于保存所有样本的预测 logits 向量、标签文本和原始表示；第 10-13 行用来累计预测正确样本的个数；第 14-17 分别用来处理得到在使用 add_embedding 时所需要用到的变量；第 19 行则是返回所有需要用到的结果。

**第 3 步：定义训练过程**

在完成上述步骤后便可以来实现模型训练部分的代码，同时对需要可视化的变量进行记录。由于这部分代码较长，掌柜下面就分块进行介绍。

```
    def train(self):
        train_iter, test_iter = load_dataset(self.batch_size)
        last_epoch = -1
        if os.path.exists('./model.pt'):
            checkpoint = torch.load('./model.pt')
            last_epoch = checkpoint['last_epoch']
            self.model.load_state_dict(checkpoint['model_state_dict'])

        num_training_steps = len(train_iter)  self.epochs
        optimizer = torch.optim.Adam([{"params": self.model.parameters(),
                                       "initial_lr": self.learning_rate}])
        scheduler = get_cosine_schedule_with_warmup(optimizer,
                                    num_warmup_steps=300,
                                    num_training_steps=num_training_steps,
                                    num_cycles=2, last_epoch=last_epoch)
```

在上述代码中，第 4-7 行用来判断本地是否存在之前保存的模型，如果存在则直接载入，而 last_epoch 是用来获取得到之前模型结束训练时的状态，目的是能够使得 Tensoboard 在可视化的时候能够接着之前的数据集进行可视化。第 10-14 行则是用来分别定义优化器和学习率动态调整策略。

进一步，实现模型训练时损失、准确率和学习率的可视化过程，代码如下：

```
        writer = SummaryWriter("runs/nin")
        max_test_acc = 0
        for epoch in range(self.epochs):
            for i, (x, y) in enumerate(train_iter):
                x, y = x.to(self.device), y.to(self.device)
```

```
                loss, logits = self.model(x, y)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
                scheduler.step()
                if i % 50 == 0:
                    acc = (logits.argmax(1) == y).float().mean()
                    print(f"# Epochs [{epoch + 1}/{ self.epochs}]-batch"
                    f"[{len(train_iter)}/{i}]-acc {acc}-loss {loss.item()}")
                    writer.add_scalar('Training/Accuracy', acc,
                                      scheduler.last_epoch)
                writer.add_scalar('Training/Loss', loss.item(),
                                      scheduler.last_epoch)
                writer.add_scalar('Training/Learning Rate',
                                  scheduler.get_last_lr()[0],
                                  scheduler.last_epoch)
```

在上述代码中，第 1 行用来实例化 SummaryWriter 用于后续可视化操作；第 5-14 行则是用来执行整个模型的训练过程；第 15-18 行则是用来分别对训练集上的准确率、损失和学习率进行可视化。

最后一步便是通过 add_embedding 来对预测结果进行可视化，代码如下：

```
            acc, all_logits, y_labels, label_img =
                        self.evaluate(test_iter,self.model, self.device)
            writer.add_scalar('Testing/Accuracy',acc, scheduler.last_epoch)
            writer.add_embedding(mat=all_logits,  # 所有点
                                 metadata=y_labels,  # 标签名称
                                 label_img=label_img,  # 标签图片
                                 global_step=scheduler.last_epoch)
```

在上述代码中，第 1-2 行主要用来获取 add_embedding 所需要的遍历；第 3-7 行则是分别对测试集上的准确率和预测结果进行可视化。

### 9.3.3 可视化展示

在完成所有部分的编码工作后，便可以通过如下代码来运行整个模型：

```
if __name__ == '__main__':
    model = MyModel()
    model.train()
```

在程序运行开始后，便可以通过第 9.1 节中的方式启动 Tensoboard 前端界面，并可以看到如图 9-18 所示的可视化结果。


图 9-18. LeNet5 训练可视化结果图

如图 9-18 所示便是模型在训练集上的准确率、学习率和损失的变化结果。进一步，我们可以将分类后的结果在空间中进行展示，如图 9-19 所示。

图 9-19. LeNet5 模型预测标签可视化结果图（一）

如图 19 所示便是模型在测试集上的预测结果经过 add_embedding 方法可视化后的结果，其中每个小方块都表示一个原始样本，每种颜色代表一个类别。进一步，点击任意方块便可以查看该样本的相关信息，如图 9-20 所示。


图 9-20. LeNet5 模型预测标签可视化结果图（二）

如图 9-20 所示便是 ankle boot 的可视化结果，并且可以发现只要点击其中一个样本，与它类别相同的样本也会被标记出来。当然，该页面还有其它相应的功能大家可以试着去探索，这里掌柜就不进行介绍了。

到此，对于 Tensorboard 的使用方法就介绍完了。在下一节内容中，掌柜将会介绍如何从零开始训练一个 BERT 模型，并同时通过 Tensoboard 来可视化网络在训练过程中各个参数的变化情况。

# 第 10 节 从零实现 NSP 和 MLM 预训练任务

经过前面几节内容的介绍，我们已经清楚了 BERT 模型的基本原理、如何从零实现 BERT、如何基于 BERT 预训练模型来完成文本分类任务、文本蕴含任务、问答选择任务（SWAG）以及问题回答任务（SQuAD），算是完成了 BERT 模型前面两部分内容的介绍。在接下来的这一节内容中，掌柜将开始就 BERT 模型的第三部分内容，即如何利用 MLM 和 NSP 这两个预训练任务来训练 BERT 模型进行介绍。

## 10.1 引言

通常来说，我们既可以通过 MLM 和 NSP 这两个任务来从头训练一个 BERT 模型，当然也可以在开源预训练模型的基础上再次通过 MLM 和 NSP 任务来在特定语料中对模型进行追加训练，以使得模型参数更加符合这一场景。并且一般来说更加倾向于第二种做法。

在第 1.1 节 BERT 模型的基本原理中掌柜已经就 MLM 和 NSP 这两个任务的原理做了详细的介绍，所以这里就不再赘述。一句话概括，如图 10-1 所示 MLM 就是随机掩盖掉部分 Token 让模型来预测，而 NSP 则是同时输入模型两句话让模型判断后一句话是否真的为前一句话的下一句话，最终通过这两个任务来训练 BERT 中的权重参数。

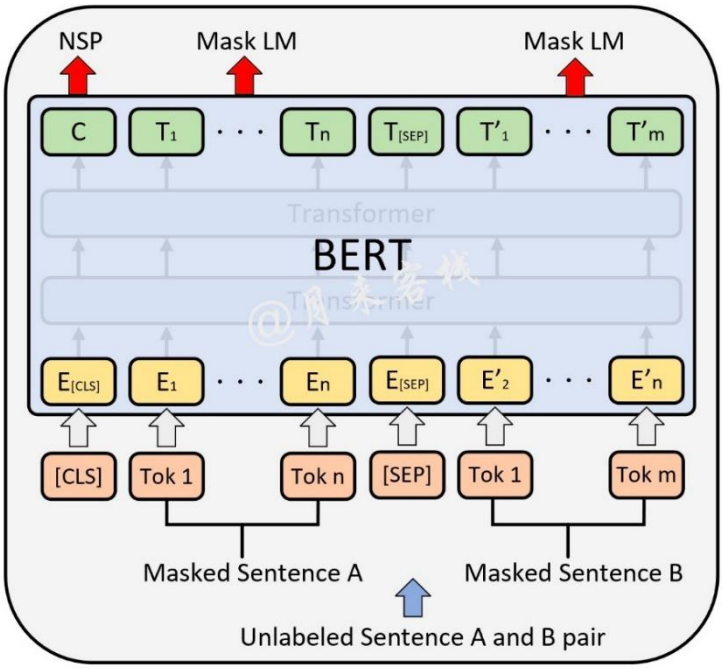



图 10-1. MLM 和 NSP 任务网络结构图

## 10.2 数据预处理

在正式介绍数据预处理之前，我们还是依照老规矩先通过一张图来大致了解一下整个处理流程，以便做到心中有数不会迷路。

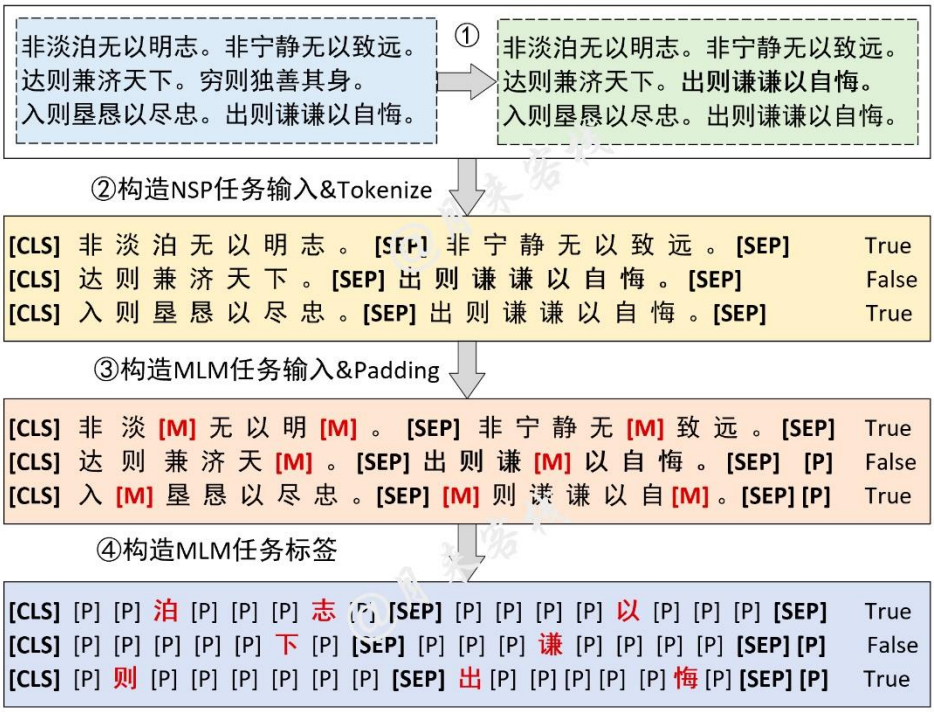



图 10-2. MLM 和 NSP 任务数据集构造流程图

如图 10-2 所示便是整个 NSP 和 MLM 任务数据集的构建流程。第①②步是根据原始语料来构造 NSP 任务所需要的输入和标签；第③步则是随机 MASK 掉部分 Token 来构造 MLM 任务的输入，并同时进行 padding 处理；第④步则是根据第③步处理后的结果来构造 MLM 任务的标签值，其中[P]表示 Padding 的含义，这样做的目的是为了忽略那些不需要进行预测的 Token 在计算损失时的损失值。在大致清楚了整个数据集的构建流程后，我们下面就可以一步一步地来完成数据集的构建了。

同时，为了能够使得整个数据预处理代码具有通用性，同时支持构造不同场景语料下的训练数据集，因此我们需要为每一类不同的数据源定义一个格式化函数来完成标准化的输入。这样即使是换了不同的语料只需要重写一个针对该数据集的格式化函数即可，其余部分的代码都不需要进行改动。

### 10.2.1 英文数据格式化

这里首先以英文维基百科数据 wiki2 [23]为例来介绍如何得到格式化后的标准数据。如下所示便是 wiki2 中的原始文本数据：

```
1 The development of [UNK] powder , based on [UNK] or [UNK] , by the French
  inventor Paul [UNK] in 1884 was a further step allowing smaller charges of
  propellant with longer barrels . The guns of the pre @-@ [UNK] battleships
```

```
    of the 1890s tended to be smaller in calibre compared to the ships of the
    1880s , most often 12 in ( 305 mm ) , but progressively grew in length of
    barrel , making use of improved [UNK] to gain greater muzzle velocity .
2
3     = = = [UNK] of armament = = =
4
5   The nature of the projectiles also changed during the ironclad period .
    Initially , the best armor @-@ piercing [UNK] was a solid cast @-@ iron
    shot . Later , shot of [UNK] iron , a harder iron alloy , gave better armor
    @-@ piercing qualities . Eventually the armor @-@ piercing shell was
    developed .
```

在上述示例数据中，每一行都表示一个段落，其由一句话或多句话组成。如图 2-6 所示，我们需要在目录 utils 下新建 create_pretraining_data.py 模块，然后定义一个函数来对其进行预处理：

```
1 def read_wiki2(filepath=None,seps='.'):
2     with open(filepath, 'r') as f:
3         lines = f.readlines()
4     paragraphs = []
5     for line in tqdm(lines, ncols=80, desc=" ## 正在读取原始数据"):
6         if len(line.split(' . ')) < 2:
7             continue
8         line = line.strip()
9         paragraphs.append([line[0]])
10        for w in line[1:]:
11            if paragraphs[-1][-1][-1] in seps:
12                paragraphs[-1].append(w)
13            else:
14                paragraphs[-1][-1] += w
15    random.shuffle(paragraphs)  # 将所有段落打乱
16    return paragraphs
```

在上述代码中，第 1 行 seps 用于指定句子与句子之间的分隔符。第 2-3 行用于一次读取所有原始数据，每一行为一个段落。第 5-14 行用于遍历每一个段落，并进行相应的处理。第 6-7 行用于过滤掉段落中只有一个句子的情况，因为后续我们要构造 NSP 任务所需的数据集所以只有一句话的段落需要去掉。第 8 行用于去掉整个段落两端的空格或换行符。第 9-14 行开始遍历段落中的每一句话并进行分割，同时保留了分隔符在句子中。第 15 行则是将所有的段落给打乱，注意不是句子。

最终，经过 read_wiki2 函数处理后，我们便能得到一个标准的 2 维列表，格式形如：

```
1 [ [sentence a1, sentence a2, ...], [sentence b1, sentence b2,...],...,[] ]
```

例如上述语料处理后的结果为：

```
[['the development of [unk] powder , based on [unk] or [unk] , by the french inventor paul [unk] in 1884 was a further step allowing smaller charges of propellant with longer barrels', 'the guns of the pre @-@ [unk] battleships of the 1890s tended to be smaller in calibre compared to the ships of the 1880s , most often 12 in ( 305 mm ) , but progressively grew in length of barrel , making use of improved [unk] to gain greater muzzle velocity .'], ['the nature of the projectiles also changed during the ironclad period', 'initially , the best armor @-@ piercing [unk] shot .'],[],[]...[]]
```

这种格式就是后续代码处理所接受的标准格式，如果需要引入自己的数据那么务必需要处理成这样的格式。

### 10.2.2 中文数据格式化

在介绍完英文数据集的格式化过程后我们再来看一个中文原始数据的格式化过程。如下所示便是我们后续所需要用到的中文宋词数据集：

```
红酥手，黄縢酒，满城春色宫墙柳。东风恶，欢情薄。一怀愁绪，几年离索。错错错。春如旧，人空瘦，泪痕红鲛绡透。桃花落。闲池阁。山盟虽在，锦书难托。莫莫莫。
十年生死两茫茫。不思量。自难忘。千里孤坟，无处话凄凉。纵使相逢应不识，尘满面，鬓如霜。夜来幽梦忽还乡。小轩窗。正梳妆。相顾无言，惟有泪千行。料得年年断肠处，明月夜，短松冈。
```

在上述示例中，每一行表示一首词，句与句之间通过句号进行分割。下面我们同样需要定义一个函数来对其进行预处理并返回指定的标准格式：

```
def read_songci(filepath=None, seps='。'):
    with open(filepath, 'r', encoding='utf-8') as f:
        lines = f.readlines()
    paragraphs = []
    for line in tqdm(lines, ncols=80, desc=" ## 正在读取原始数据"):
        if "□" in line or "……" in line or len(line.split('。')) < 2:
            continue
        paragraphs.append([line[0]])
        line = line.strip()  # 去掉换行符和两边的空格
        for w in line[1:]:
            if paragraphs[-1][-1][-1] in seps:
                paragraphs[-1].append(w)
            else:
                paragraphs[-1][-1] += w
    random.shuffle(paragraphs)  # 将所有段落打乱
    return paragraphs
```

在上述代码中，第 1 行 seps 用于指定句子与句子之间的分隔符。第 2-3 行用于一次读取所有原始数据，每一行为一首词（段落）。第 5-14 行用于遍历每一个

段落，并进行相应的处理。第 6-7 行用于过滤掉字符乱码以及段落中只有一个句子的情况。第 8-14 行开始遍历段落中的每一句话并进行分割，同时保留了分隔符在句子中。第 15 行则是将所有的段落给打乱，注意不是句子。

例如上述语料处理后的结果为：

```
1 [['五花心里看抛球', '香腮红嫩柳烟稠'], ['若论风流，无过圆社，拐蹬蹊搭齐全', '门庭富贵，曾到御帘前', '灌口二郎为首，赵皇上、下脚流传', '人都道、齐云一社，三锦独争先', '花前', '并月下，全身绣带，偷侧双肩', '更高而不远，一搭打秋千', '球落处、圆光拐，双佩剑、侧蹊相连', '高人处，翻身估料，天下总呼圆
'], [], [].... []]
```

可以看到， 预处理完成后的结果同上面 wiki2 数据预处理完后的格式一样。

### 10.2.3 构造 NSP 任务数据

在正式构造 NSP 任务数据之前，我们需要在 create_pretraining_data.py 先定义一个类并定义相关的类成员变量以方便在其它成员方法中使用，代码如下：

```
1 class LoadBertPretrainingDataset(object):
2     def __init__(self,
3                  vocab_path='./vocab.txt',
4                  tokenizer=None,
5                  batch_size=32,
6                  max_sen_len=None,
7                  max_position_embeddings=512,
8                  pad_index=0,
9                  is_sample_shuffle=True,
10                 random_state=2021,
11                 data_name='wiki2',
12                 masked_rate=0.15,
13                 masked_token_rate=0.8,
14                 masked_token_unchanged_rate=0.5,
15                 seps='。'):
16         self.tokenizer = tokenizer
17         self.vocab = build_vocab(vocab_path)
18         self.PAD_IDX = pad_index
19         self.SEP_IDX = self.vocab['[SEP]']
20         self.CLS_IDX = self.vocab['[CLS]']
21         self.MASK_IDS = self.vocab['[MASK]']
22         self.batch_size = batch_size
23         self.max_sen_len = max_sen_len
24         self.max_position_embeddings = max_position_embeddings
25         self.pad_index = pad_index
26         self.is_sample_shuffle = is_sample_shuffle
27         self.data_name = data_name
```

```
28          self.masked_rate = masked_rate
29          self.masked_token_rate = masked_token_rate
30          self.masked_token_unchanged_rate = masked_token_unchanged_rate
31          self.random_state = random_state
32          self.seps = seps
33          random.seed(random_state)
```

由于后续会有一系列的随机操作，所以上面代码第 33 行加入了随机状态用于固定随机结果。

紧接着，需要定义一个成员函数来封装格式化原始数据集的函数，代码如下：

```
1     def get_format_data(self, filepath,):
2         if self.data_name == 'wiki2':
3             return read_wiki2(filepath, self.seps)
4         elif self.data_name == 'songci':
5             return read_songci(filepath, self.seps)
6         elif self.data_name == 'custom':
7             return read_custom(filepath)
8         else:
9             raise ValueError(f"数据 {self.data_name} 不存在对应的格式化函数，"
10                              f"请参考 read_wiki(filepath)实现对应格式化函数！")
```

从上述代码可以看出，该函数的作用就是给出了一个标准化的格式化函数调用方式，可以根据指定的数据集名称返回相应的格式化函数。但是需要注意的是，格式化函数返回的格式需要同 read_wiki2()函数返回的样式保持一致。

进一步，我们便可以来定义构造 NSP 任务数据的处理函数，用来根据给定的连续两句话和对应的段落返回 NSP 任务中的句子对和标签，具体代码如下：

```
1     @staticmethod
2     def get_next_sentence_sample(sentence, next_sentence, paragraphs):
3         if random.random() < 0.5:  # 产生[0,1)之间的一个随机数
4             is_next = True
5         else:
6             next_sentence = random.choice(random.choice(paragraphs))
7             is_next = False
8         return sentence, next_sentence, is_next
```

在上述代码中，第 3 行用于根据均匀分布产生[0,1)之间的一个随机数作为概率值。第 6 行则是先从所有段落中随机出一个段落，再从随机出的一个段落中随机出一句话，以此来随机选择下一句话。第 8 行则是返回构造好的一条 NSP 任务样本。最后，由于该方法只是功能性的函数没有引用到类中的其它成员，所以通过第 1 行代码将其申明为了静态方法。

到此，对于 NSP 任务样本的构造就介绍完了，后续我们只需要调用 get_next_sentence_sample()函数即可。

### 10.2.4 构造 MLM 任务数据

为了方便后续构造 MLM 任务中的数据样本，我们这里需要先定义一个辅助函数，其作用是根据给定的 token_ids、候选 mask 位置以及需要 mask 的数量来返回被 mask 后的 token_ids 和标签 label 信息，代码如下：

```
1    def replace_masked_tokens(self, token_ids, candidate_pred_positions,
                                                                num_mlm_preds):
2        pred_positions = []
3        mlm_input_tokens_id = [token_id for token_id in token_ids]
4        for mlm_pred_position in candidate_pred_positions:
5            if len(pred_positions) >= num_mlm_preds:
6                break
7            masked_token_id = None
8            if random.random() < self.masked_token_rate:
9                masked_token_id = self.MASK_IDS
10           else:
11               if random.random() < self.masked_token_unchanged_rate:
12                   masked_token_id = token_ids[mlm_pred_position]
13               else:
14                   masked_token_id=random.randint(0,len(self.vocab.stoi)-1)
15           mlm_input_tokens_id[mlm_pred_position] = masked_token_id
16           pred_positions.append(mlm_pred_position)
17       mlm_label = [self.PAD_IDX if idx not in pred_positions
18                    else token_ids[idx] for idx in range(len(token_ids))]
19       return mlm_input_tokens_id, mlm_label
```

在上述代码中，第 1 行里 token_ids 表示经过 get_next_sentence_sample()函数处理后的上下句，且已经转换为 ids 后的结果，candidate_pred_positions 表示所有可能被 maks 掉的候选位置，num_mlm_preds 表示根据15% 的比例计算出来的需要被 mask 掉的位置数量。第 4-6 行为依次遍历每一个候选 Token 的索引，如果已满足需要被 mask 的数量则跳出循环。第 8-9 行则表示将其中80% 的 Token 替换为[MASK]（注意，这里其实就是15% 里面的80%）。第 10-14 行则是分别保持10% 的 Token 不变以及将另外10% 替换为随机 Token。第 15-16 则是对 Token 进行替换，以及记录下哪些位置上的 Token 进行了替换。第 17-18 行则是根据已记录的 Token 替换信息得到对应的标签信息，其做法便是如果该位置没出现在 pred_positions 中则表示该位置不是需要被预测的对象，因此在进行损失计算时需要忽略掉这些位置（即为 PAD_IDX）。而如果其出现在 mask 的位置，则其标签为原始 token_ids 对应的 id，即正确标签。

例如以下输入：

```
token_ids = [101, 1031, 4895, 2243, 1033, 10029, 2000, 2624, 1031,....]
candidate_pred_positions = [2,8,5,9,7,3...]
num_mlm_preds = 5
```

经过函数 replace_masked_tokens()处理后的结果则类似为：

```
mlm_input_tokens_id = [101,1031,103,2243,1033, 10029, 2000, 103,  1031, ...]
mlm_label = [ 0,    0,    4895,  0,    0,    0,    0,    2624,  0,...]
```

在这之后，我们便可以定义一个函数来构造 MLM 任务所需要用到的训练数据，代码如下：

```
    def get_masked_sample(self, token_ids):
        candidate_pred_positions = []  # 候选预测位置的索引
        for i, ids in enumerate(token_ids):
            # 在遮蔽语言模型任务中不会预测特殊词元，所以如果该位置是特殊词元
            # 那么该位置就不会成为候选 mask 位置
            if ids in [self.CLS_IDX, self.SEP_IDX]:
                continue
            candidate_pred_positions.append(i)
            # 保存候选位置的索引， 例如可能是 [ 2,3,4,5, ....]
        random.shuffle(candidate_pred_positions) # 将候选位置打乱，更利于随机
        # 被掩盖位置的数量，BERT 模型中默认将 15%的 Token 进行 mask
        num_mlm_preds = max(1, round(len(token_ids) * self.masked_rate))
        logging.debug(f" ## Mask 数量为: {num_mlm_preds}")
        mlm_input_tokens_id, mlm_label = self.replace_masked_tokens(
            token_ids, candidate_pred_positions, num_mlm_preds)
        return mlm_input_tokens_id, mlm_label
```

在上述代码中，第 1 行 token_ids 便是传入的模型输入序列的 Token ID（一个样本）。第 3-8 行是用来记录所有可能进行掩盖的 Token 的索引，并同时排除掉特殊 Token。第 10 行是将所有候选位置打乱，更利于后续随机抽取。第 12 行则是用来计算需要进行掩盖的 Token 的数量，例如原始论文中是15% 。第 14-15 行便是上面介绍到的 replace_masked_tokens()功能函数。第 16 行则是返回最终 MLM 任务和 NSP 任务的输入 mlm_input_tokens_id 和 MLM 任务的标签 mlm_label。

### 10.2.5 构造整体任务数据

在分别介绍完 MLM 和 NSP 两个任务各自的样本构造方法后，下面我们再通过一个方法将两者组合起来便得到了最终整个样本数据的构建，代码如下：

```
    @cache
    def data_process(self, filepath, postfix='cache'):
        paragraphs = self.get_format_data(filepath)
```

```
4           # 返回的是一个二维列表，每个列表可以看做是一个段落（其中每个元素为一句话）
5           data = []
6           max_len = 0
7           # max_len 用来记录整个数据集中最长序列的长度，后续可将其作为 padding 长度的标准
8           desc = f" ## 正在构造 NSP 和 MLM 样本({filepath.split('.')[1]})"
9           for paragraph in tqdm(paragraphs, ncols=80, desc=desc):
10              for i in range(len(paragraph) - 1):  # 遍历一个段落中的每一句话
11                  sentence, next_sentence, is_next
12                      = self.get_next_sentence_sample(paragraph[i],
13                      paragraph[i + 1], paragraphs)  # 构造 NSP 样本
14                  logging.debug(f" ## 当前句文本：{sentence}")
15                  logging.debug(f" ## 下一句文本：{next_sentence}")
16                  logging.debug(f" ## 下一句标签：{is_next}")
17                  if len(next_sentence) < 2 :
18                      logging.warning(f"句子{sentence}的下一句为空，请检查！")
19                      continue
20                  token_a_ids = [self.vocab[token] for token in
21                                                  self.tokenizer(sentence)]
22                  token_b_ids = [self.vocab[token] for token in
23                                                  self.tokenizer(next_sentence)]
24                  token_ids = [self.CLS_IDX] + token_a_ids +
25                                                 [self.SEP_IDX] + token_b_ids
26                  if len(token_ids) > self.max_position_embeddings - 1:
27                      token_ids = token_ids[:self.max_position_embeddings - 1]
28                  token_ids += [self.SEP_IDX]
29                  seg1 = [0] * (len(token_a_ids) + 2)
30                  seg2 = [1] * (len(token_ids) - len(seg1))
31                  segs = torch.tensor(seg1 + seg2, dtype=torch.long)
32                  nsp_lable = torch.tensor(int(is_next), dtype=torch.long)
33                  mlm_input_tokens_id, mlm_label =
34                                          self.get_masked_sample(token_ids)
35                  token_ids=torch.tensor(mlm_input_tokens_id,dtype=torch.long)
36                  mlm_label = torch.tensor(mlm_label, dtype=torch.long)
37                  max_len = max(max_len, token_ids.size(0))
38                  logging.debug(f"## Mask 之后 token ids:{token_ids.tolist()}")
39                  logging.debug(f"## Mask 之后词元结果：{[self.vocab.itos[t]
40                                                  for t in token_ids.tolist()]}")
41                  logging.debug(f"## Mask 之后 label ids:{mlm_label.tolist()}")
42                  logging.debug(f"## 当前样本构造结束================== \n\n")
43                  data.append([token_ids, segs, nsp_lable, mlm_label])
44          all_data = {'data': data, 'max_len': max_len}
45          return all_data
```

在上述代码中，第 1 行中的@cache 修饰器用于保存或直接载入已预处理完成后的结果，具体原理可以参见文章[27]。第 6 行中的 max_len 用来记录整个数据集中最长序列的长度，在后续可将其作为 padding 长度的标准。从第 9-10 行开始，便是依次遍历每个段落以及段落中的每个句子来构造 MLM 和 NSP 任务样本。第 11-13 行用于构建 NSP 任务数据样本；第 17-19 行用于过滤掉 NSP 任务中下一句过段或为空等数据预处理中没有考虑到的情况；第 20-28 行则是将得到的 Token 序列转换为 token_ids，其中 26-27 行用于判断序列长度，对于超出部分进行截取。

紧接着，第 29-32 行则是分别构造 Segment Embedding 输入和 NSP 任务的真实标签；第 33-36 行是分别构造 MLM 任务的输入和标签。第 43 行则是将每个构造完成的样本保存到 data 这个列表中；第 43-45 行是返回最终生成的结果。

例如在处理宋词语料时，上述代码便会输出如下类似结果：

```
1 - DEBUG:  ## 当前句文本：风住尘香花已尽，日晚倦梳头
2 - DEBUG:  ## 下一句文本：锦书欲寄鸿难托
3 - DEBUG:  ## 下一句标签：False
4 - DEBUG:  ## Mask 之前词元结果：['[CLS]', '风', '住', '尘', '香', '花', '已', '尽',
', ', '日', '晚', '倦', '梳', '头','[SEP]', '锦', '书', '欲','寄', '鸿','难', '托','[SEP]']
5 - DEBUG:  ## Mask 之前 token ids:[101, 7599, 857, 2212, 7676, 5709, 2347,
 2226, 8024, 3189, 3241, 958, 3463, 1928, 102, 7239, 741, 3617, 2164, 7896,
 7410, 2805, 102]
6 - DEBUG:  ## segment ids:[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
 1, 1, 1, 1, 1, 1],序列长度为 23
7 - DEBUG:  ## Mask 数量为: 3
8 - DEBUG:  ## Mask 之后 token ids:[101, 7599, 857, 2212, 103, 5709, 2347, 103,
 8024, 3189, 3241, 103, 3463, 1928, 102, 7239, 741, 3617, 2164, 7896, 7410,
 2805, 102]
9 - DEBUG:  ## Mask 之后词元结果：['[CLS]','风', '住', '尘','[MASK]','花','已','[MASK]',
 ', ', '日','晚','[MASK]', '梳','头','[SEP]', '锦','书', '欲','寄','鸿','难','托', '[SEP]']
10 - DEBUG:  ## Mask 之后 label ids:[0, 0, 0, 0, 7676, 0, 0, 2226, 0, 0, 0, 958,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
11 - DEBUG:  ## 当前样本构造结束==================
```

如果在构造数据集时不想输出上述结果，只需要将日志等级设置为 log_level =logging.INFO 即可。

### 10.2.6 构造训练数据集

在整个数据预处理结束后，便可以构造得到最终模型训练时所需要的 DataLoader 了。如下代码便是训练集、验证集和测试集 3 部分 DataLoader 的构建过程：

```
1     def load_train_val_test_data(self,
2                 train_file_path=None,val_file_path=None,
3                 test_file_path=None,only_test=False):
```

```
4         postfix = f"_ml{self.max_sen_len}_rs{self.random_state}"
5                   f"_mr{str(self.masked_rate)[2:]}"
6                   f"_mtr{str(self.masked_token_rate)[2:]}"
7                   f"_mtur{str(self.masked_token_unchanged_rate)[2:]}"
8         test_data = self.data_process(filepath=test_file_path,
9                                       postfix='test' + postfix)['data']
10        test_iter = DataLoader(test_data, batch_size=self.batch_size,
11                               shuffle=False, collate_fn=self.generate_batch)
12        if only_test:
13            logging.info(f"# 返回测试集，包含样本{len(test_iter.dataset)}个")
14            return test_iter
15        data = self.data_process(filepath=train_file_path,
                                                  postfix='train' + postfix)
16        train_data, max_len = data['data'], data['max_len']
17        if self.max_sen_len == 'same':
18            self.max_sen_len = max_len
19        train_iter = DataLoader(train_data, batch_size=self.batch_size,
20                                shuffle=self.is_sample_shuffle,
21                                collate_fn=self.generate_batch)
22        val_data = self.data_process(filepath=val_file_path,
23                                           postfix='val' + postfix)['data']
24        val_iter = DataLoader(val_data, batch_size=self.batch_size,
25                              shuffle=False,collate_fn=self.generate_batch)
26        logging.info(f"# 返回训练集样本（{len(train_iter.dataset)}）个、"
27                     f"开发集样本（{len(val_iter.dataset)}）个"
28                     f"测试集样本（{len(test_iter.dataset)}）个.")
29        return train_iter, test_iter, val_iter
```

在上述代码中，第 4-7 行是根据传入的相关参数来构建一个数据预处理结果的缓存名称，因为不同的参数会处理得到不同的结果，最终缓存后的数据预处理结果名称将会类似如下所示：

```
1 songci_test_mlNone_rs2021_mr15_mtr8_mtur5.pt
```

这样在每次载入数据集时如果已经有相应的预处理缓存则直接载入即可。

第 8-14 行便是用来构造测试集所对应的 DataLoader。第 15-21 行则用于构建训练集所对应的 DataLoader，其中如果 self.max_sen_len 为 same，那么在对样本进行 padding 时会以整个数据中最长样本的长度为标准进行 padding，该参数默认情况下为 None，即以每个 batch 中最长的样本为标准进行 padding，更多相关内容可以参见 4.2.4 节第 4 步中的介绍；第 22-25 行则是构造验证集所对应的 DataLoader。

到此，对于整个 BERT 模型预训练的数据集就算是构建完成了。

### 10.2.7 使用示例

在整个数据集的 DataLoader 构建完毕后，便可以通过如下方式来进行使用：

```
class ModelConfig:
    def __init__(self):
        self.project_dir = os.path.dirname(os.path.dirname(
                                                os.path.abspath(__file__)))
        # ========== wiki2 数据集相关配置
        # self.dataset_dir=os.path.join(self.project_dir,'data','WikiText')
        # self.pretrained_model_dir = os.path.join(self.project_dir,
        #                                          "bert_base_uncased_english")
        # self.train_file_path = os.path.join(self.dataset_dir,
        #                                                  'wiki.train.tokens')
        # self.val_file_path = os.path.join(self.dataset_dir,
        #                                                  'wiki.valid.tokens')
        # self.test_file_path = os.path.join(self.dataset_dir,
        #                                                   'wiki.test.tokens')
        # self.data_name = 'wiki2'
        # self.seps = '.'

        # ========== songci 数据集相关配置
        self.dataset_dir = os.path.join(self.project_dir, 'data', 'SongCi')
        self.pretrained_model_dir = os.path.join(self.project_dir,
                                                 "bert_base_chinese")
        self.train_file_path = os.path.join(self.dataset_dir,
                                                  'songci.train.txt')
        self.val_file_path = os.path.join(self.dataset_dir,
                                                  'songci.valid.txt')
        self.test_file_path = os.path.join(self.dataset_dir,
                                                   'songci.test.txt')
        self.data_name = 'songci'
        self.seps = '。'
        self.vocab_path=os.path.join(self.pretrained_model_dir, 'vocab.txt')
        self.model_save_dir = os.path.join(self.project_dir, 'cache')
        self.logs_save_dir = os.path.join(self.project_dir, 'logs')
        self.is_sample_shuffle = True
        self.batch_size = 16
        self.max_sen_len = None
        self.max_position_embeddings = 512
        self.pad_index = 0
        self.is_sample_shuffle = True
        self.random_state = 2021
        self.masked_rate = 0.15
```

```
40          self.masked_token_rate = 0.8
41          self.masked_token_unchanged_rate = 0.5
42          ......
```

在上述代码中，第 5-16 行为 wiki2 数据集的相关路径，而 17-28 行则是 songci 数据集的相关路径，可以根据需要直接进行切换。第 29-42 行则是其它数据预处理的相关数据。

最后，我们便可通过如下方式来实例化类 LoadBertPretrainingDataset 并输出相应的结果：

```
1 if __name__ == '__main__':
2     config = ModelConfig()
3     data_loader = LoadBertPretrainingDataset(
4         vocab_path=config.vocab_path,
5         tokenizer=BertTokenizer.from_pretrained(
6             config.pretrained_model_dir).tokenize,
7         batch_size=config.batch_size,
8         max_sen_len=config.max_sen_len,
9         max_position_embeddings=config.max_position_embeddings,
10        pad_index=config.pad_index,
11        is_sample_shuffle=config.is_sample_shuffle,
12        random_state=config.random_state,
13        data_name=config.data_name,
14        masked_rate=config.masked_rate,
15        masked_token_rate=config.masked_token_rate,
16        masked_token_unchanged_rate=config.masked_token_unchanged_rate,
17        seps=config.seps)
18    test_iter = data_loader.load_train_val_test_data(
19        test_file_path=config.test_file_path,only_test=True)
20    for b_token_ids, b_segs, b_mask, b_mlm_label, b_nsp_label in test_iter:
21        print(b_token_ids.shape)  # [src_len,batch_size]
22        print(b_segs.shape)  # [src_len,batch_size]
23        print(b_mask.shape)  # [batch_size,src_len]
24        print(b_mlm_label.shape)  # [src_len,batch_size]
25        print(b_nsp_label.shape)  # [batch_size]
26        break
```

输出结果如下：

```
1 - INFO: 缓存文件 ~/BertWithPretrained/data/SongCi/songci_test_
2             mlNone_rs2021_mr15_mtr8_mtur5.pt 存在，直接载入缓存文件!
3 - INFO: ## 成功返回测试集，一共包含样本 6249 个
4 torch.Size([42, 16])
5 torch.Size([42, 16])
6 torch.Size([16, 42])
```

```
7 torch.Size([42, 16])
8 torch.Size([16])
```

到此，对 BERT 模型中 NSP 和 MLM 这两个预训练任务数据集的构造过程掌柜就介绍完了。接下来，让我来看如何实现这两个任务对应的前向传播过程。

## 10.3 预训练任务实现

根据第 1 节内容可知，BERT 的预训练过程包括两个任务：NSP 和 MLM。为了使得大家能够对这两部分的代码实现有着更加清晰的认识与理解，掌柜将先分别来实现这两个任务，最后再将两者结合到一起来实现 BERT 的预训练任务。

### 10.3.1 NSP 任务实现

由于 NSP 任务实现起来较为简单，所以掌柜这里就直接贴出代码：

```
class BertForNextSentencePrediction(nn.Module):

    def __init__(self, config, bert_pretrained_model_dir=None):
        super(BertForNextSentencePrediction, self).__init__()
        if bert_pretrained_model_dir is not None:
            self.bert = BertModel.from_pretrained(config,
                                                  bert_pretrained_model_dir)
        else:
            self.bert = BertModel(config)
        self.classifier = nn.Linear(config.hidden_size, 2)

    def forward(self,
                input_ids,  # [src_len, batch_size]
                attention_mask=None,  # [batch_size, src_len]
                token_type_ids=None,  # [src_len, batch_size]
                position_ids=None,
                next_sentence_labels=None):  # [batch_size,]
        pooled_output, _ = self.bert(
            input_ids=input_ids,attention_mask=attention_mask,
            token_type_ids=token_type_ids,position_ids=position_ids)
        # pooled_output: [batch_size, hidden_size]
        seq_relationship_score = self.classifier(pooled_output)
        # seq_relationship_score: [batch_size, 2]
        if next_sentence_labels is not None:
            loss_fct = nn.CrossEntropyLoss()
            loss = loss_fct(seq_relationship_score.view(-1, 2),
                                              next_sentence_labels.view(-1))
            return loss
        else:
            return seq_relationship_score
```

上述代码便是整个 NSP 任务的实现，可以看到其本质上就是一个文本分类任务，仅仅只用取 BERT 模型最后一层输出的[CLS]做一个分类任务即可，这里掌柜就不再赘述了。

## 10.3.2 MLM 任务实现

相比较于 NSP，对于实现 MLM 任务来说则稍微复杂了一点点。它需要将 BERT 模型整个最后一层的输出进行一次变换和标准化，然后再做 Token 级的分类任务来预测被掩盖部分对应的 Token 值，这个网络结构如图 10-3 所示。

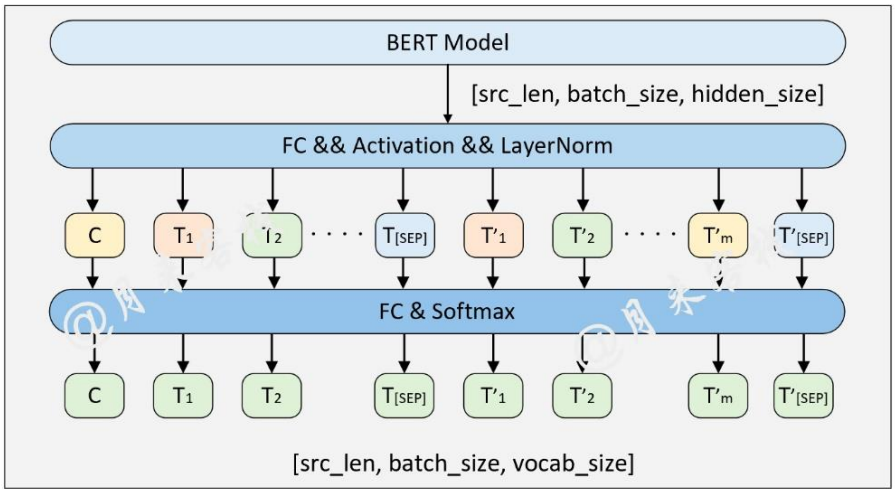



图 10-3. MLM 任务网络结构图

如图 10-3 所示便是构造 MLM 任务的流程示意。首先取 BERT 模型最后一层的输出，形状为[src_len,batch_size,hidden_size]；然后再经过一次（非）线性变换和标准化，形状同样为[src_len,batch_size,hidden_size]；最后再经过一个分类层对每个 Token 进行分类处理便得到了最后的预测结果，形状为[src_len,batch_size,vocab_size]。

此时我们便可以定义类 BertForLMTransformHead 来完成上述 3 个步骤：

```
class BertForLMTransformHead(nn.Module):
    def __init__(self, config, bert_model_embedding_weights=None):
        super(BertForLMTransformHead, self).__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        if isinstance(config.hidden_act, str):
            self.transform_act_fn = get_activation(config.hidden_act)
        else:
            self.transform_act_fn = config.hidden_act
        self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=1e-12)
        self.decoder = nn.Linear(config.hidden_size, config.vocab_size)
        if bert_model_embedding_weights is not None:
            self.decoder.weight = nn.Parameter(bert_model_embedding_weights)
        # [hidden_size, vocab_size]
```

```
14          self.decoder.bias = nn.Parameter(torch.zeros(config.vocab_size))
15      def forward(self, hidden_states):
16          hidden_states = self.dense(hidden_states)
17          hidden_states = self.transform_act_fn(hidden_states)
18          hidden_states = self.LayerNorm(hidden_states)
19          hidden_states = self.decoder(hidden_states)
20          return hidden_states # [src_len, batch_size, vocab_size]
```

在上述代码中，第 4-8 行用来定义相应的（非）线性变换。第 9-10 行则是用来定义对应的标准化和最后的分类层。第 11-12 则是用来判断最后分类层中的权重参数是否复用 BERT 模型 Token Embedding 中的权重参数，因为 MLM 任务最后的预测类别就等于 Token Embedding 中的各个词，所以最后分类层中的权重参数可以复用 Token Embedding 中的权重参数[26]。第 15 行开始则是对应的前向传播过程。第 16-18 行处理后的结果形状均为[src_len, batch_size, hidden_size]。

同时，还有一个细节的地方在于，在 Token Embedding 中词表的形状为[vocab_size,hidden_size]，而最后一个分类层权重参数的形状为[hidden_size,vocab_size]，因此按道理说这里的权重 bert_model_embedding_weights 应该要转置后才能复用。不过在第 12 行代码中我们却直接将其赋值给了最后一层的权重参数，这是为什么呢？原来在 PyTorch 的 nn.Linear()中，权重参数默认的初始化方式就是

```
1 self.weight = Parameter(torch.Tensor(out_features, in_features))
```

也就是将两个维度进行了交换，所以刚好在复用时就不用做其它任何操作了。

紧接着，我们便可以通过如下代码来实现 MLM 任务：

```
1  class BertForMaskedLM(nn.Module):
2      def __init__(self, config, bert_pretrained_model_dir=None):
3          super(BertForMaskedLM, self).__init__()
4          if bert_pretrained_model_dir is not None:
5              self.bert = BertModel.from_pretrained(config,
6                                                    bert_pretrained_model_dir)
7          else:
8              self.bert = BertModel(config)
9          weights = None
10         if config.use_embedding_weight:
11             weights = self.bert.bert_embeddings.
12                                     word_embeddings.embedding.weight
13
14         self.classifier = BertForLMTransformHead(config, weights)
15         self.config = config
16     def forward(self,input_ids, attention_mask=None,token_type_ids=None,
17                 position_ids=None, masked_lm_labels=None):
18         _, all_encoder_outputs = self.bert(
19             input_ids=input_ids,attention_mask=attention_mask,
20             token_type_ids=token_type_ids,position_ids=position_ids)
```

```
21          sequence_output = all_encoder_outputs[-1]  # 取 Bert 最后一层的输出
22          prediction_scores = self.classifier(sequence_output)
23          if masked_lm_labels is not None:
24              loss_fct = nn.CrossEntropyLoss(ignore_index=0)
25              masked_lm_loss = loss_fct(prediction_scores.reshape(-1,
26                                                  self.config.vocab_size),
27                                        masked_lm_labels.reshape(-1))
28              return masked_lm_loss
29          else:
30              return prediction_scores  # [src_len, batch_size, vocab_size]
```

在上述代码中，第 4-8 行用于返回得到原始的 BERT 模型。第 9-12 行则是取 Token Embedding 中的权重参数。第 14 行则是返回得到 MLM 任务实例化后的类对象。第 18-21 行是返回得到 BERT 模型的所有层输出，并只取最后一层，此时的形状为[src_len, batch_size, hidden_size]。第 22 行则是完成最后 MLM 中分类任务的输出，形状为[src_len, batch_size, vocab_size]。第 23-30 行则是根据标签是否为空来返回不同的输出结果。

到此，对于 MLM 任务的实现掌柜就介绍完了。

### 10.3.3 前向传播

经过上面两节内容的介绍，此时再来整体实现 NSP 和 MLM 任务那就变得十分容易了。整体实现代码如下所示：

```
1  class BertForPretrainingModel(nn.Module):
2      def __init__(self, config, bert_pretrained_model_dir=None):
3          super(BertForPretrainingModel, self).__init__()
4          if bert_pretrained_model_dir is not None:
5              self.bert = BertModel.from_pretrained(config,
6                                                    bert_pretrained_model_dir)
7          else:  # 如果没有指定预训练模型路径，则随机初始化整个网络权重
8              self.bert = BertModel(config)
9          weights = None
10         if 'use_embedding_weight' in config.__dict__ and
11                                                 config.use_embedding_weight:
12             weights = self.bert.bert_embeddings.
13                                         word_embeddings.embedding.weight
14
15         self.mlm_prediction = BertForLMTransformHead(config, weights)
16         self.nsp_prediction = nn.Linear(config.hidden_size, 2)
17         self.config = config
18
19     def forward(self, input_ids,  # [src_len, batch_size]
20                 attention_mask=None,  # [batch_size, src_len]
```

```
21                 token_type_ids=None,  # [src_len, batch_size]
22                 position_ids=None,
23                 masked_lm_labels=None,  # [src_len,batch_size]
24                 next_sentence_labels=None):  # [batch_size]
25         pooled_output, all_encoder_outputs = self.bert(
26             input_ids=input_ids,attention_mask=attention_mask,
27             token_type_ids=token_type_ids,position_ids=position_ids)
28         sequence_output = all_encoder_outputs[-1]  # 取 Bert 最后一层的输出
29         mlm_prediction_logits = self.mlm_prediction(sequence_output)
30         nsp_pred_logits = self.nsp_prediction(pooled_output)
31         if masked_lm_labels is not None and next_sentence_labels is not None
32             loss_fct_mlm = nn.CrossEntropyLoss(ignore_index=0)
33             loss_fct_nsp = nn.CrossEntropyLoss()
34             mlm_loss = loss_fct_mlm(mlm_prediction_logits.reshape(
35                 -1,self.config.vocab_size),masked_lm_labels.reshape(-1))
36             nsp_loss = loss_fct_nsp(nsp_pred_logits.reshape(-1, 2),
37                                     next_sentence_labels.reshape(-1))
38             total_loss = mlm_loss + nsp_loss
39             return total_loss, mlm_prediction_logits, nsp_pred_logits
40         else: # [src_len, batch_size, vocab_size], [batch_size, 2]
41             return mlm_prediction_logits, nsp_pred_logits
```

在上述代码中，第 15-16 行分别是返回得到实例化后的 MLM 和 NSP 任务模型。第 25-27 行则是返回 BERT 模型的所有输出。第 27-30 行则是分别取 BERT 模型输出的不同部分来分别进行后续的 MLM 和 NSP 任务，此时 sequence_output 的形状为[src_len, batch_size, hidden_size]，mlm_prediction_logits 的形状为[src_len, batch_size, vocab_size]，nsp_pred_logits 的形状为[batch_size, 2]。第 31-41 行则是根据是否有标签输入来返回不同的输出结果，同时需要注意的是第 37 行返回的是 NSP+MLM 两个任务的损失和作为整体模型的损失值；第 39-41 行是根据条件返回模型不同的结果。

到此对于 NSP 和 MLM 任务模型的实现部分就介绍完了。不过掌柜在这里同样要提醒大家的是，在逐行阅读代码的时候最好是将各个变量的维度一起带进去，弄清楚每一步计算后各个变量维度的变化，这样才能更好的理解整个模型。

## 10.4 模型训练与微调

在实现完整个 NSP 和 MLM 部分的代码后便可以开始进行模型的训练。同时，经过训练完成之后的模型参数又可以继续在下游任务中进行微调。

### 10.4.1 模型训练

对于整个模型训练部分的代码其实和掌柜在前面几个微调任务中介绍的差不多，只是为了能更加清楚地知道训练模的训练过程型掌柜在这里加入了一些通

过 Tensorboard 可视化的代码。由于这部分代码较长掌柜就分两部分来进行介绍。第一部分代码如下：

```
1 def train(config):
2     model = BertForPretrainingModel(config,config.pretrained_model_dir)
3     last_epoch = -1
4     if os.path.exists(config.model_save_path):
5         checkpoint = torch.load(config.model_save_path)
6         last_epoch = checkpoint['last_epoch']
7         loaded_paras = checkpoint['model_state_dict']
8         model.load_state_dict(loaded_paras)
9         logging.info("## 成功载入已有模型，进行追加训练.....")
10    model.train()
11    tokenize = BertTokenizer.from_pretrained(config.pretrained_model_dir)
12    data_loader=LoadBertPretrainingDataset(vocab_path=config.vocab_path,...)
13    train_iter, test_iter, val_iter = \
14        data_loader.load_train_val_test_data(config.test_file_path, ...)
15    no_decay = ["bias", "LayerNorm.weight"]
16    optimizer_grouped_parameters = [
17        {
18            "params": [p for n, p in model.named_parameters()
19                       if not any(nd in n for nd in no_decay)],
20            "weight_decay": config.weight_decay,
21            "initial_lr": config.learning_rate
22        },
23        {
24            "params": [p for n, p in model.named_parameters()
25                       if any(nd in n for nd in no_decay)],
26            "weight_decay": 0.0,
27            "initial_lr": config.learning_rate
28        },
29    ]
30    optimizer = AdamW([{"params": model.parameters(),
31                        "initial_lr": config.learning_rate}])
32    scheduler = get_polynomial_decay_schedule_with_warmup(optimizer,
33      config.num_warmup_steps,config.num_train_steps,last_epoch=last_epoch)
```

在上述代码中，第 2 行是实例化模型对象；第 4-9 行是查看本地是否存在相关模型（指对训练过程中保存的模型进行追加训练），这里值得一说的是之所以也要保存 last_epoch 这个参数是为了同时能够恢复学习率、以及 Tensoboard 中相关可视化变量在上一次模型保存是时的状态；第 11-14 行是载入训练模型时所需要用到的数据集；第 15-29 行是筛选模型中哪些参数需要进行权重衰减（也就是 $L_2$ 正则化），哪些参数不需要进行权重衰减，同时根据筛选条件可知所有 bias 和 LayerNorm 相关的参数都不需要进行正则化处理；第 30-31 行是定义优化器，并

且是通过 initial_lr 来指定的学习率的，因为后续需要用到动态学习率调整的策略；第 32-33 行是指定动态学习率的调整策略。

进一步，实现模型的迭代正反向传播过程，代码如下：

```
1     for epoch in range(config.epochs):
2         losses = 0
3         for idx, (b_token_ids, b_segs, b_mask, b_mlm_label, b_nsp_label)
4                                                 in enumerate(train_iter):
5           #......
6             loss, mlm_logits, nsp_logits = model(input_ids=b_token_ids,
7                     attention_mask=b_mask,token_type_ids=b_segs,
8                     masked_lm_labels=b_mlm_label,
9                     next_sentence_labels=b_nsp_label)
10            optimizer.zero_grad()
11            loss.backward()
12            optimizer.step()
13            scheduler.step()
14            losses += loss.item()
15            mlm_acc, _, _, nsp_acc, _, _ = accuracy(mlm_logits, nsp_logits,
16                                b_mlm_label,b_nsp_label, data_loader.PAD_IDX)
17            if idx % 20 == 0:
18                config.writer.add_scalar('Training/Loss', loss.item(),
19                                         scheduler.last_epoch)
20                config.writer.add_scalar('Training/Learning Rate',
21                                         scheduler.get_last_lr()[0],
22                                         scheduler.last_epoch)
23                config.writer.add_scalars(main_tag='Training/Accuracy',
24                                          tag_scalar_dict={'NSP': nsp_acc,
25                                                           'MLM': mlm_acc},
26                                          global_step=scheduler.last_epoch)
27        train_loss = losses / len(train_iter)
28        if (epoch + 1) % config.model_val_per_epoch == 0:
29            mlm_acc, nsp_acc = evaluate(config, val_iter, model,
                                                        data_loader.PAD_IDX)
30            config.writer.add_scalars(main_tag='Testing/Accuracy',
31                                      tag_scalar_dict={'NSP': nsp_acc,
                                                         'MLM': mlm_acc},
32                                      global_step=scheduler.last_epoch)
33            if mlm_acc > max_acc:
34                max_acc = mlm_acc
35                state_dict = model.state_dict()
36            torch.save({'last_epoch': scheduler.last_epoch,
37                        'model_state_dict': state_dict},
38                       config.model_save_path)
```

在上述代码中，第 6-9 行是模型的前向传播过程；第 10-13 行是反向传播及相关参数的更新过程；第 14-16 行分别是损失的累计和两个预训练任务准确率的计算；第 18-26 行是对训练过程中模型的损失、学习率和准确率进行可视化；第 29-32 行是计算模型在测试集上的准确率，并同时进行可视化处理；第 33-38 行则是保存在测试集上 MLM 任务取得最大准确率时所对应的模型。

到此，对于模型训练部分的内容就介绍完了。同时，模型在训练过程中将会有类似如下所示的输出：

```
1 - INFO: Epoch: [1/120], Batch[ 0/7836], Train loss : 1.897, Train mlm acc:
0.0, nsp acc: 0.537
2 - INFO:  Epoch: [1/120], Batch[20/7836], Train loss : 1.786, Train mlm acc:
0.0, nsp acc: 0.562
3 - INFO: Epoch: [1/120], Batch[40/7836], Train loss : 1.812, Train mlm acc:
0.0, nsp acc: 0.489
```

最终，模型的在宋词数据集上损失值和准确率的变化如图 10-4 所示。

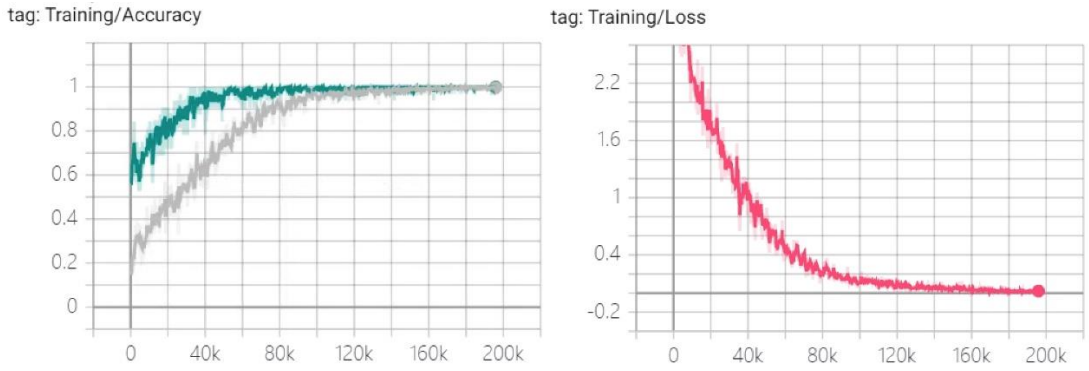



图 10-4. 模型损失和准确率变化图

如图 10-4 所示，左边为 NSP 和 MLM 两个预训练任务整体损失的变化情况；左边绿色曲线和灰色曲线分别为 NSP 和 MLM 这两个任务在训练集上的准确率的变化情况。不过虽然模型看似在训练集上有着不错的训练效果，但是在测试集上的结果却显得不那么尽如人意，如图 10-5 所示。

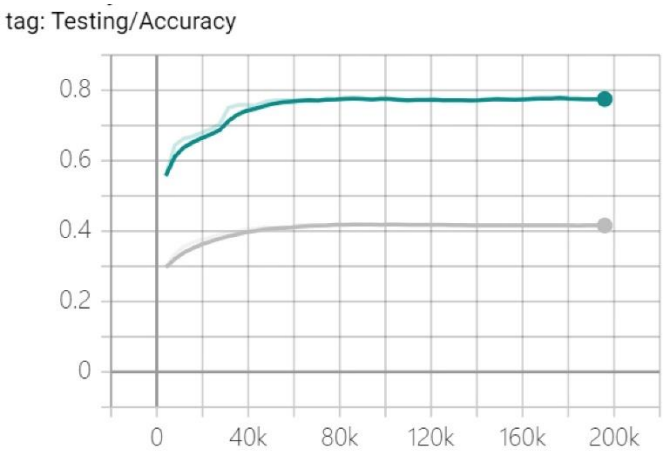



图 10-5. 模型测试集准确率变化图

如图 10-5 所示便是模型在测试集上的表现情况，其中绿色曲线和灰色曲线分别表示 NSP 和 MLM 这两个任务在测试集上的准确率变化情况。从图中可以发现，模型大约在 6 万次迭代后准确率就没有发生明显地变换了。在调整过多次参数组合后依旧没有得到一个好的效果，掌柜猜测这可能是由于使用的宋词数据集比较特别，不像普通的白话文那样测试集和训练集的句式比较类似；以及原始数据集每句话的分割方式导致句子过短等。同时 NSP 和 MLM 任务的最后两层参数也随机初始化的并没有使用开源的模型参数。当然，也不排除代码中存在未知 Bug 的情况。不过后续掌柜也会继续尝试调整模型，并将相应的结果推送到代码仓库中。

### 10.4.2 模型推理

在模型训练部分的内容介绍完毕后，下面我们再来看模型推理部分的实现。对于推理部分的实现总体思路为：①将测试样本构造为模型所接受的输入格式；②通过模型前向传播得到预测结果输出；③对模型输出结果进行格式化处理得到最终的预测结果。

首先，对于模型预测部分的实现代码如下：

```
1 def inference(config, sentences=None, masked=False, language='en'):
2     tokenize = BertTokenizer.from_pretrained(config.pretrained_model_dir)
3     data_loader = LoadBertPretrainingDataset(vocab_path=config.vocab_path,
4                 tokenizer=tokenize.tokenize,pad_index=config.pad_index,
5                 random_state=config.random_state,masked_rate=0.15)# 15% Mask
6     token_ids, pred_idx, mask = data_loader.make_inference_samples(
7                                 sentences,masked=masked,language=language)
8     model = BertForPretrainingModel(config,config.pretrained_model_dir)
9     if os.path.exists(config.model_save_path):
10        checkpoint = torch.load(config.model_save_path)
11        loaded_paras = checkpoint['model_state_dict']
12        model.load_state_dict(loaded_paras)
13        logging.info("## 成功载入已有模型进行推理.....")
14    else:
15        raise ValueError(f"模型 {config.model_save_path} 不存在！")
16    model.eval()
17    with torch.no_grad():
18        mlm_logits, _ = model(input_ids=token_ids,attention_mask=mask)
19    pretty_print(token_ids, mlm_logits, pred_idx,
20                 data_loader.vocab.itos, sentences, language)
```

在上述代码中，第 3-5 行为初始化类 LoadBertPretrainingDataset，同时需要说明的是由于是预测场景，所以构造样本时 masked_rate 可以是任意值，不用局限 15%。第 6-7 行则是将传入的测试样本转换为模型所接受的形式，其中 masked 参数是用来指定输入的测试样本有没进行 mask 操作，如果没有则自动按

masked_rate 的比例进行 mask 操作；language 参数是指定测试样本的语种类型。第 9-13 行则是载入本地保存好的模型来初始化完了。第 17-18 行是得到模型前向传播的输出结果。第 19-20 行是根据模型的前向传播输出结果来格式化得到最终的输出形式。

最终，可以通过如下方式来完成模型的推理过程，代码如下：

```
1 if __name__ == '__main__':
2     config = ModelConfig()
3     train(config)
4     sentences = ["十年生死两茫茫。不思量。自难忘。千里孤坟，无处话凄凉。",
5                  "红酥手。黄藤酒。满园春色宫墙柳。"]
6     inference(config, sentences, masked=False, language='zh')
```

上述代码运行结束后将会看到类似如下所示的结果：

```
1 - INFO: ## 成功载入已有模型进行推理……
2 - INFO:  ### 原始：我住长江头，君住长江尾。
3 - INFO:   ## 掩盖：我住长江头，[MASK]住长[MASK]尾。
4 - INFO:   ## 预测：我住长江头，君住长河尾。
5 - INFO: ====================
6 - INFO:  ### 原始：日日思君不见君，共饮长江水。
7 - INFO:   ## 掩盖：日日思君不[MASK]君，共[MASK]长江水。
8 - INFO:   ## 预测：日日思君不见君，共饮长江水。
9 #   ......
```

需要注意的是，由于目前模型在测试集上的结果并不理想，所以上述展示的是模型在训练集上的推理结果。

### 10.4.3 模型微调

在介绍完整个预训练模型实现过程后，最后一步就是如何将训练得到的模型继续运用在下游任务中。当然，实现这一目的也非常简单，只需要将保存好的模型重新命名为 pytorch_model.bin，然后替换掉之前的文件即可。这样就可以想前面介绍的几个下游任务一样对模型进行微调了。

# 总结

在本篇文章中，掌柜首先详细地介绍了 BERT 模型的基本原理以及其所提出的动机，并且明白了 BERT 模型本质上就是 Transformer 模型的 Encoder 部分；接着，掌柜一步一步详细地介绍了如何从零来实现 BERT 模型，包括导入 Input Embedding 的实现、BertModel 的实现以及如何完成预训练模型的加载和模型的迁移等内容；然后介绍了基于 BERT 预训练模型的四大微调任务场景，包括单文本分类如任务、文本对分类任务、问题选择任务和问题回答任务，同时也详细讲解了在对模型进行训练时的相关技巧，如动态学习了调整和通过 Tensorboard 进行模型变量可视化等；最后，掌柜细致的介绍了 BERT 模型中 NSP 和 MLM 这两个任务的实现过程，以及构建一个通用的数据预处理模块，大家只需要根据自己的需要定义一个数据格式化函数便可以快捷地在自己的数据集上完成整个模型的训练工作。当然，掌柜在后期也会根据实际情况继续补充一些基于 BERT 预训练模型的其它微调场景，如关系抽取、命名体识别等任务。

本次内容就到此结束，感谢您的阅读！如果你觉得上述内容对你有所帮助，欢迎分享至一位你的朋友！若有任何疑问与建议，请添加掌柜微信'nulls8'或加群进行交流。**青山不改，绿水长流，我们月来客栈见！**

**扫码关注@月来客栈可获得更多优质内容！**

# 引用

[1] BERT: Pre-training of Deep Bidirectional Transformers for Language Under standing.
[2] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need
[3] http://jalammar.github.io/illustrated-transformer/
[4] This post is all you need
[5] The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learni ng) https://jalammar.github.io/illustrated-bert/
[6] This Post Is All You Need（上卷）——层层剥开 Transformer
[7] https://github.com/google-research/bert/blob/master/modeling.py#L558
[8] BERT https://huggingface.co/transformers/model_doc/bert.html#bertmodel
[9] https://github.com/codertimo/BERT-pytorch
[10]bert_base_chinese 下载地址：https://huggingface.co/bert-base-chinese/tree/main
[11]SAVING AND LOADING MODELS https://pytorch.org/tutorials/beginner/saving_loading_models.html
[12]https://github.com/moon-hotel/DeepLearningWithMe
[13]https://github.com/aceimnorstuvwxz/toutiao-text-classfication-dataset
[14]训练模型时如何便捷保存训练日志
[15]https://cims.nyu.edu/~sbowman/multinli/
[16]https://rowanzellers.com/swag/
[17]https://github.com/rowanz/swagaf/tree/master/data
[18]https://huggingface.co/docs/transformers/main_classes/optimizer_schedules
[19]https://rajpurkar.github.io/SQuAD-explorer/
[20]Python 中的默认字典与命名体元组你会用吗？
[21]https://pytorch.org/docs/master/tensorboard.html#torch-utils-tensorboard
[22]详解机器学习中的 Precision-Recall 曲线
[23]https://s3.amazonaws.com/research.metamind.io/wikitext/wikitext-2-v1.zip
[24]https://docs.python.org/3.6/library/random.html?highlight=random
[25]动手学深度学习，李沐
[26]https://github.com/google-research/bert/
[27]如何用@修饰器来缓存数据与处理结果？