未来能引领人工智能世界的是Transformer而非ChatGPT

Transformer变形金刚红书 <packt>

ChatGPT红得发紫，
强得让人类心悸，
但在它的背后，
还隐藏着一位真正的大佬：
它的名字叫作
——Transformer

Transformer到底是什么，
让ChatGPT如此迷恋？
这只变形金刚，
到底是一只什么怪兽？
——用于自然语言处理（NLP）的
神经网络架构

精通 Transformer

从零开始构建最先进的NLP模型

[土] 萨瓦斯·伊尔蒂利姆（Savaş Yıldırım） [伊朗] 梅萨姆·阿斯加里－切纳格卢（Meysam Asgari-Chenaghlu）◎著 江红 余青松 余靖◎译

MASTERING TRANSFORMERS

BUILD STATE-OF-THE-ART MODELS FROM SCRATCH WITH ADVANCED NATURAL LANGUAGE PROCESSING TECHNIQUES

北京理工大学出版社
BEIJING INSTITUTE OF TECHNOLOGY PRESS

Transformer

AI领域的基石王者，那些还沉迷于CNN和RNN的工程师被警告：

放弃战斗吧，向Transformer投降！

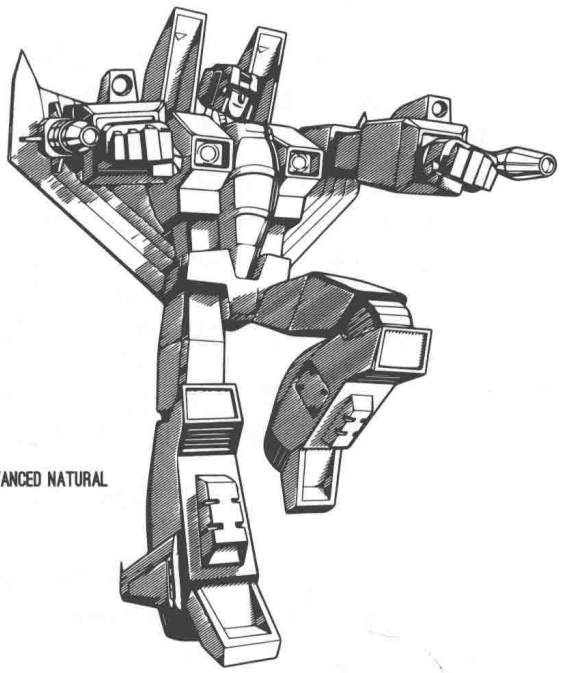
上架指导：计算机/编程开发

ISBN 978-7-5763-2227-9

9 787576 322279 >

定价：99.00元

<packt>



MASTERING TRANSFORMERS

BUILD STATE-OF-THE-ART MODELS FROM SCRATCH WITH ADVANCED NATURAL LANGUAGE PROCESSING TECHNIQUES

精通 Transformer

从零开始构建最先进的NLP模型

[土] 萨瓦斯·伊尔蒂利姆 (Savaş Yıldırım)　[伊朗] 梅萨姆·阿斯加里－切纳格卢 (Meysam Asgari-Chenaghlu) ◎著

江红　余青松　余靖◎译

北京理工大学出版社
BEIJING INSTITUTE OF TECHNOLOGY PRESS

版权专有　侵权必究

图书在版编目(CIP)数据

精通 Transformer：从零开始构建最先进的 NLP 模型 /（土）萨瓦斯·伊尔蒂利姆，（伊朗）梅萨姆·阿斯加里－切纳格卢著；江红，余青松，余靖译. -- 北京：北京理工大学出版社，2023.4

书名原文：Mastering Transformers Build state－of－the－art models from scratch with advanced natural language processing techniques

ISBN 978－7－5763－2227－9

Ⅰ. ①精… Ⅱ. ①萨… ②梅… ③江… ④余… ⑤余… Ⅲ. ①自然语言处理 Ⅳ. ①TP391

中国国家版本馆 CIP 数据核字(2023)第 056068 号

北京市版权局著作权合同登记号　图字：01－2023－1318 号

Title：Mastering Transformers，by Savaş Yıldırım and Meysam Asgari－Chenaghlu

Copyright © 2021 Packt Publishing

First published in the English language under the title－'Mastering Transformers －（9781801077651）' by Packt Publishing.

Simplified Chinese edition copyright © 2023 by Beijing Jie Teng Culture Media Co.，Ltd.

All rights reserved. Unauthorized duplication or distribution of this work constitutes copyright infringement.

出版发行 / 北京理工大学出版社有限责任公司
社　　址 / 北京市海淀区中关村南大街 5 号
邮　　编 / 100081
电　　话 /（010）68914775（总编室）
（010）82562903（教材售后服务热线）
（010）68944723（其他图书服务热线）
网　　址 / http：//www.bitpress.com.cn
经　　销 / 全国各地新华书店
印　　刷 / 文畅阁印刷有限公司
开　　本 / 787 毫米×1092 毫米　1/16
印　　张 / 17.5
字　　数 / 388 千字
版　　次 / 2023 年 4 月第 1 版　2023 年 4 月第 1 次印刷
定　　价 / 99.00 元

责任编辑 / 钟　博
文案编辑 / 钟　博
责任校对 / 刘亚男
责任印制 / 施胜娟

图书出现印装质量问题，请拨打售后服务热线，本社负责调换

作者简介
Author Profile

萨瓦斯·伊尔蒂利姆（Savaş Yıldırım）毕业于伊斯坦布尔技术大学计算机工程系，拥有自然语言处理（Natural Language Processing，NLP）专业的博士学位。目前，他是土耳其伊斯坦布尔比尔基大学的副教授，也是加拿大瑞尔森大学的访问研究员。他是一位积极热情的讲师和研究员，具有20多年教授机器学习、深度学习和自然语言处理等课程的丰富经验。他开发了大量的开源软件和资源，为土耳其自然语言理解社区做出了重大贡献。他还为人工智能公司的研究开发项目提供全面的咨询服务。在业余时间，他还创作和导演电影短片，并喜欢练习瑜伽。

“首先，我要感谢我亲爱的合作伙伴，Aylin Oktay，感谢她在撰写本书的漫长过程中不断的支持、耐心和鼓励。我还要感谢伊斯坦布尔比尔基大学计算机工程系同事们的支持。”

梅萨姆·阿斯加里 – 切纳格卢（Meysam Asgari – Chenaghlu）是碳咨询公司（Carbon Consulting）的人工智能经理，同时在大不里士大学攻读博士学位。他曾担任土耳其几家领先的电信和银行公司的顾问，还参与了自然语言理解和语义搜索等各种项目。

“首先（也是最重要的），我要感谢我亲爱的充满耐心的妻子Narjes Nikzad – Khasmakhi，感谢她对我的支持和理解。还要感谢我的父亲对我的支持，愿他的灵魂安息。非常感谢碳咨询公司和我的同事们。”

译者序
Preface

在大数据和人工智能时代，机器学习（Machine Learning，ML）和深度学习（Deep Learning，DL）已经成为各行各业解决问题的有效方法，自然语言处理（Natural Language Processing，NLP）是深度学习的重要应用领域之一。在过去的二十年中，自然语言处理经历了翻天覆地的变化：从传统的自然语言处理方法（*n*－gram 语言模型、基于 TF－IDF 的信息检索模型、独热编码文档术语矩阵等）到深度学习方法（RNN、CNN、FFNN、LSTM 等），再到 Transformer。

目前，基于 Transformer 的语言模型主导了自然语言处理领域的研究，已经成为一种新的范式。Transformer 模型是谷歌公司于 2017 年推出的自然语言处理经典模型。在自然语言处理任务中，Transformer 的表现超越了 RNN 和 CNN，只需编码器/解码器就能达到很好的效果，并且可以实现高效的并行化。Transformer 社区提供的大量预训练模型为自然语言处理的研究和生成部署提供了最前沿的基准。

本书涵盖了 Transformer 深度学习体系结构的理论知识和实践指南。借助 Hugging Face 社区的 Transformer 库，本书循序渐进地提供了各种自然语言处理问题的解决方案。本书采用理论和实践相结合的方式，系统地阐述了自然语言处理的相关理论和技术，详细介绍了使用 Transformer 训练、微调和部署自然语言处理解决方案的流程。

通过本书的学习，读者可以利用 Transformer 库探索最先进的自然语言处理解决方案；使用 Transformer 体系结构训练任何语言模型；微调预训练的语言模型以执行多个下游任务；监控训练过程、可视化解释 Transformer 内部表示以及部署生产环境。

本书的读者对象主要包括深度学习研究人员、自然语言处理从业人员、教师和学生。本书要求读者具有良好的 Python 基础知识以及机器学习和深度学习的基本知识。

本书由华东师范大学江红、余青松和余靖共同翻译。衷心感谢北京颉腾文化传媒有限公司李华君和本书的编辑老师，他们积极帮助我们筹划翻译事宜并认真审阅翻译稿件。翻译也是一种再创造，同样需要艰辛的付出，感谢朋友、家人以及同事的理解和支持。感谢我们的研究生刘映君、余嘉昊、刘康、钟善毫、方宇雄、唐文芳、许柯嘉等同学对本译稿的认真通读和指正。在本书翻译的过程中我们力求忠于原著，但由于时间和学识有限，并且本书涉及各个领域的专业知识，故书中的不足之处在所难免，敬请诸位同行、专家和读者指正。

江红　余青松　余靖

2023 年 2 月

前言
Forward

在过去的二十年中，自然语言处理研究领域发生了翻天覆地的变化。在这段时间里，自然语言处理经历了不同的处理范式，并最终进入了一个由神奇的 Transformer 体系结构主导的新时代。Transformer 深度学习架构是通过继承许多方法而产生的，其中包括上下文词嵌入、多头注意力机制、位置编码、并行体系结构、模型压缩、迁移学习、跨语言模型等。在各种基于神经的自然语言处理方法中，Transformer 架构逐渐演变为基于注意力的“编码器 – 解码器”体系结构，并持续发展到今天。现在，我们在文献中看到了这种体系结构的新的成功变体。目前研究已经发现了只使用 Transformer 架构中编码器部分的出色模型，如 BERT（Bidirectional Encoder Representations from Transformers，Transformers 双向编码表示）；或者只使用 Transformer 架构中解码器部分的出色模型，如 GPT（Generated Pre – trained Transformer，生成式的预训练 Transformer）。

本书涵盖了这些自然语言处理方法。基于 Hugging Face 社区的 Transformer 库，我们能够轻松地使用 Transformer。本书将循序渐进地提供各种自然语言处理问题的解决方案：从文档摘要到问题回答系统。我们将看到，基于 Transformer，可以取得最先进的成果。

读者对象

本书面向深度学习研究人员、自然语言处理从业人员、机器学习/自然语言处理教育者，以及希望开启 Transformer 体系结构学习之旅的学生群体。为了充分掌握本书的内容，要求读者具有初级水平的机器学习知识，以及良好的 Python 基础知识。

本书涵盖的内容

第 1 章 从词袋到 Transformer。本章简要介绍了自然语言处理的历史，并将传统方法、深度学习模型（如 CNN、RNN 和 LSTM）与 Transformer 模型进行了比较分析。

第 2 章 Transformer 的实践操作入门。本章深入探讨了如何使用 Transformer 模型，并通过实际例子阐述了分词器和模型，如 BERT。

第 3 章 自编码语言模型。本章讨论了如何从零开始在任何给定语言上训练自编码语言模型。训练过程将包括模型的预训练和特定任务的训练。

第 4 章 自回归和其他语言模型。本章讨论了自回归语言模型的理论细节，并引导读者如何在自己的语料库中对模型进行预训练。读者将学习如何在自己的文本上预训练任何语言模型（如GPT – 2），并在各种任务（如语言生成）中使用该模型。

第 5 章 微调文本分类语言模型。在本章中，读者将学习如何配置文本分类的预训练模型，以及如何微调文本分类下游任务的模型（如情感分析或多类别分类）。

第 6 章 微调标记分类语言模型。本章讲述如何微调标记分类任务的语言模型［如命名实体识别（NER）、词性标注（POS）和问题回答（QA）系统］。

第 7 章 文本表示。在本章中，读者将学习文本表示技术以及如何有效地利用 Transformer 体系结构，特别是对于无监督任务，如聚类、语义搜索和主题建模。

第 8 章 使用高效的 Transformer。本章展示了如何使用提炼、剪枝和量化方法，从预训练模型中生成高效的模型。然后，读者将获得有关高效稀疏 Transformer 的知识，如 Linformer 和 BigBird，以及如何使用这些模型。

第 9 章 跨语言和多语言建模。在本章中，读者将学习多语言和跨语种语言模型预训练以及单语言和多语言预训练之间的差异。本章涉及的其他主题包括因果语言建模和翻译语言建模。

第 10 章 部署 Transformer 模型。本章将详细介绍如何在 CPU/GPU 环境中，为基于 Transformer 的自然语言处理解决方案提供服务。本章还将描述如何使用 TensorFlow 扩展（TFX）部署机器学习系统。

第 11 章 注意力可视化与实验跟踪。本章涵盖两个不同的技术概念：注意力可视化与实验跟踪。我们将使用诸如 exBERT 和 BertViz 之类的复杂工具进行实验练习。

充分利用本书的资源

为了阅读本书，读者需要具备 Python 程序设计语言的基本知识，还需要了解自然语言处理的基础知识、深度学习的基础知识，以及深度神经网络的工作原理。

重要提示

本书中的所有代码都是在 Python 3.6 版本中执行的，因为针对 Python 3.9 版本中的一些库尚处于开发阶段。

本书涵盖的软件和硬件以及对操作系统的要求请参见表 1。

表 1　本书涵盖的软件和硬件以及对操作系统的要求

本书涵盖的软件和硬件	操作系统要求
Transformer	Windows、macOS 或 Linux
TensorFlow 和 PyTorch	Windows、macOS 或 Linux
Python 3.6x	Windows、macOS 或 Linux
Jupyter Notebook	Windows、macOS 或 Linux
Google Colaboratory	Windows、macOS 或 Linux
Docker	Windows、macOS 或 Linux
Locust. io	Windows、macOS 或 Linux
Git	Windows、macOS 或 Linux

如果读者使用的是本书的数字版本，建议读者自己输入代码或者从本书的 GitHub 存储库中访问代码（具体请参考下面提供的链接地址），这种方法可以帮助读者避免复制和粘贴代码所产生的潜在错误。

下载示例代码文件

读者可以从 GitHub 存储库中下载本书的示例代码文件，具体网址为 https://github.com/PacktPublishing/Mastering-Transformers。如果书中代码有更新，GitHub 存储库中的代码将随之更新。

我们还提供了大量其他书籍和视频教程，读者可以从 GitHub 存储库中下载对应的代码包，具体请访问网址 https://github.com/PacktPublishing/。

本书约定

本书使用了以下约定。

代码的格式（code in text）：表示代码、数据库表名称、文件夹名称、文件名称、文件扩展名、路径名、统一资源定位符 URL、用户输入和 Twitter 句柄。下面是一个示例："长度小于 max_sen_len（最大句子长度）的序列使用一个 PAD 值填充，直到它们的长度达到 max_sen_len。"

代码块设置如下所示。

```
max_sen_len = max([len(s.split()) for s in sentences])
words = ["PAD"] + list(set([w for s in sentences for w in s.split()]))
word2idx = {w:i for i,w in enumerate(words)}
max_words = max(word2idx.values()) + 1
idx2word = {i:w for i,w in enumerate(words)}
train = [list(map(lambda x:word2idx[x], s.split())) for s in sentences]
```

提示或重要提示：

相关内容将出现在方框中。

目录
Contents

第 1 部分　导论：相关领域的最新发展概述、环境安装和 Hello World 应用程序

第 1 章　从词袋到 Transformer

1.1　技术需求//004
1.2　自然语言处理到 Transformer 的演变历程//005
1.3　理解分布式语义//007
1.3.1　词袋技术的实现//008
1.3.2　克服维度问题//009
1.3.3　语言建模与生成//010
1.4　利用深度学习//012
1.4.1　学习单词嵌入//012
1.4.2　循环神经网络概述//014
1.4.3　长短期记忆网络和门控循环单元//015
1.4.4　卷积神经网络概述//018
1.5　Transformer 体系结构概述//021
1.5.1　注意力机制//021
1.5.2　多头注意力机制//023
1.6　在迁移学习中结合使用 Transformer//027
1.7　本章小结//029

第 2 章　Transformer 的实践操作入门

2.1　技术需求//032
2.2　使用 Anaconda 安装 Transformer//032
2.2.1　在 Linux 操作系统中安装 Anaconda//032
2.2.2　在 Windows 操作系统中安装 Anaconda//033
2.2.3　在 macOS 操作系统中安装 Anaconda//034
2.2.4　安装 TensorFlow、PyTorch 和 Transformer//035
2.2.5　使用 Google Colab 安装环境//037
2.3　使用语言模型和分词器//037

2.4 使用社区提供的模型//039
2.5 使用基准测试和数据集//042
2.5.1 重要的基准测试//042
2.5.2 使用应用程序编程接口访问数据集//044
2.6 速度和内存的基准测试//052
2.7 本章小结//056

第 2 部分 Transformer 模型：从自编码模型到自回归模型

第 3 章 自编码语言模型
3.1 技术需求//060
3.2 BERT：一种自编码语言模型//060
3.2.1 BERT 语言模型预训练任务//061
3.2.2 对 BERT 语言模型的深入研究//062
3.3 适用于任何语言的自编码语言模型训练//064
3.4 与社区共享模型//073
3.5 了解其他自编码模型//074
3.5.1 Albert 模型概述//074
3.5.2 RoBERTa 模型//077
3.5.3 ELECTRA 模型//078
3.6 使用分词算法//079
3.6.1 字节对编码//081
3.6.2 WordPiece 分词算法//082
3.6.3 SentencePiece 分词算法//082
3.6.4 tokenizers 库//083
3.7 本章小结//088

第 4 章 自回归和其他语言模型
4.1 技术需求//090
4.2 使用自回归语言模型//091
4.2.1 生成式预训练模型的介绍与训练//091
4.2.2 Transformer - XL 模型//093
4.2.3 XLNet 模型//094
4.3 使用序列到序列模型//094
4.3.1 T5 模型//095
4.3.2 BART 概述//096
4.4 自回归语言模型训练//098

4.5　使用自回归模型的自然语言生成//103
4.6　使用 simpletransformers 进行总结和机器翻译微调//105
4.7　本章小结//108

第5章　微调文本分类语言模型
5.1　技术需求//110
5.2　文本分类导论//110
5.3　微调 BERT 模型以适用于单句二元分类//111
5.4　使用原生 PyTorch 训练分类模型//118
5.5　使用自定义数据集对多类别分类 BERT 模型进行微调//122
5.6　微调 BERT 模型以适用于句子对回归//128
5.7　使用 run_glue. py 对模型进行微调//133
5.8　本章小结//134

第6章　微调标记分类语言模型
6.1　技术需求//136
6.2　标记分类概述//136
6.2.1　理解命名实体识别//137
6.2.2　理解词性标注//138
6.2.3　理解问题回答系统//138
6.3　微调语言模型以适用于命名实体识别任务//139
6.4　基于标记分类的问题回答系统//146
6.5　本章小结//155

第7章　文本表示
7.1　技术需求//157
7.2　句子嵌入概述//158
7.2.1　交叉编码器与双向编码器//158
7.2.2　句子相似性模型的基准测试//159
7.2.3　使用 BART 模型进行零样本学习//163
7.3　使用 FLAIR 进行语义相似性实验//165
7.3.1　平均词嵌入//168
7.3.2　基于循环神经网络的文档嵌入//168
7.3.3　基于 Transformer 的 BERT 嵌入//169
7.3.4　Sentence - BERT 嵌入//169
7.4　基于 Sentence - BERT 的文本聚类//171
7.4.1　基于 paraphrase - distilroberta - base - v1 的主题建模//171

7.4.2 基于 BERTopic 的主题建模//174
7.5 基于 Sentence－BERT 的语义搜索//176
7.6 本章小结//179

第 3 部分 高级主题

第 8 章 使用高效的 Transformer
8.1 技术需求//184
8.2 高效、轻便、快速的 Transformer 概述//185
8.3 模型规模缩减的实现//186
8.3.1 使用 DistilBERT 进行知识提炼//186
8.3.2 剪枝//188
8.3.3 量化//190
8.4 使用高效的自注意力机制//192
8.4.1 固定模式下的稀疏注意力机制//192
8.4.2 可学习的模式//202
8.4.3 低秩因子分解、核函数和其他方法//207
8.5 本章小结//207

第 9 章 跨语言和多语言建模
9.1 技术需求//209
9.2 翻译语言建模与跨语言知识共享//210
9.3 跨语言的语言模型和来自 Transformer 的多语言双向编码器表示//211
9.3.1 mBERT//212
9.3.2 XLM//213
9.4 跨语言相似性任务//216
9.4.1 跨语言文本相似性//216
9.4.2 可视化跨语言文本相似性//218
9.5 跨语言分类//222
9.6 跨语言零样本学习//226
9.7 多语言模型的基本局限性//229
9.8 微调多语言模型的性能//230
9.9 本章小结//232

第 10 章 部署 Transformer 模型
10.1 技术需求//234
10.2 FastAPI Transformer 模型服务//235

10.3　容器化 API//237
10.4　使用 TFX 提供更快的 Transformer 模型服务//238
10.5　使用 Locust 进行负载测试//241
10.6　本章小结//243

第 11 章　注意力可视化与实验跟踪
11.1　技术需求//245
11.2　解读注意力头//246
11.2.1　使用 exBERT 对注意力头进行可视化//246
11.2.2　使用 BertViz 实现注意力头的多尺度可视化//251
11.2.3　使用探测分类器理解 BERT 的内部结构//259
11.3　跟踪模型度量指标//259
11.3.1　使用 TensorBoard 跟踪模型训练过程//260
11.3.2　使用 W&B 及时跟踪模型训练过程//263
11.4　本章小结//266

第 1 部分
导论：相关领域的最新发展概述、环境安装和 Hello World 应用程序

在第 1 部分中，读者将初步学习有关 Transformer 各个方面的内容。通过加载社区提供的预先训练好的语言模型，用户将编写第一个使用 Transformer 的程序 Hello World，并在配置图形处理单元（GPU）或无 GPU 的环境下运行相关代码。在第 1 部分中，还将详细阐述有关 tensorflow、pytorch、conda、transformer 和 sentenceTransformer 库的安装和使用方法。

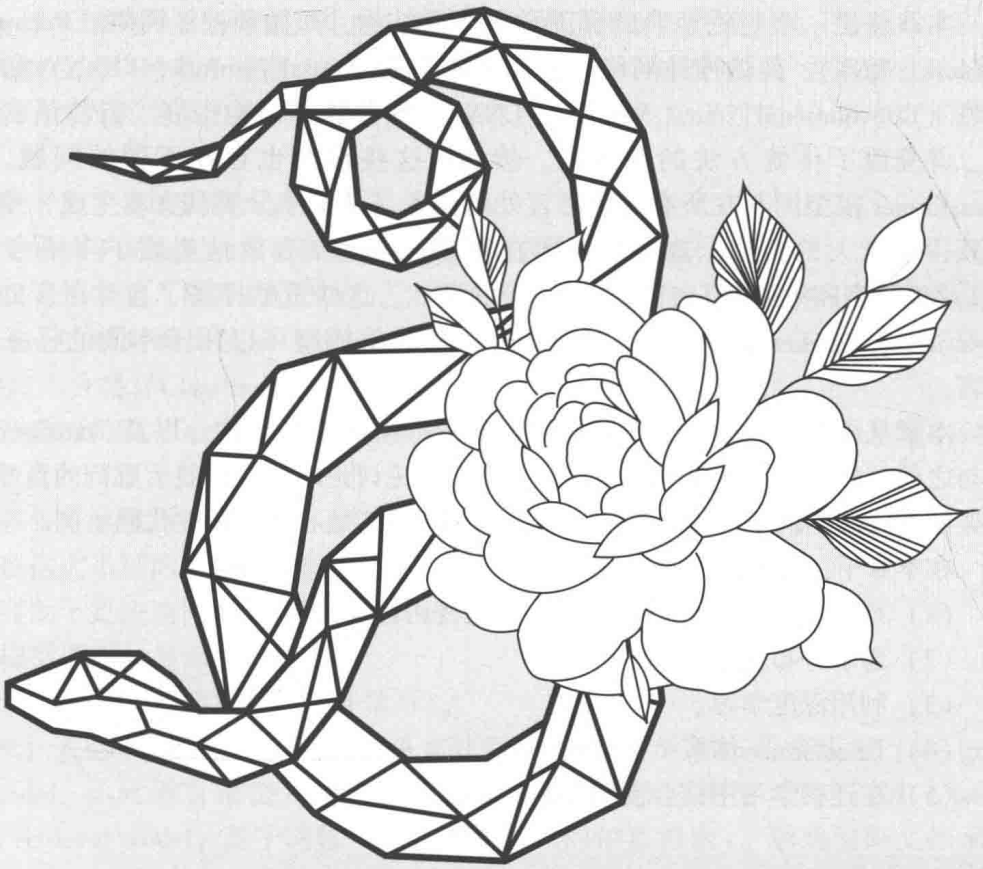
第 1 部分包括以下章节内容。

- 第 1 章：从词袋到 Transformer。
- 第 2 章：Transformer 的实践操作入门。

1

第1章

从词袋到Transformer



本章将回顾自然语言处理过去 20 年的发展历程。在过去的 20 年中，自然语言处理经历了不同的范式，最终进入 Transformer 体系结构时代。所有这些范式都将有助于用户更好地表达用于解决问题的单词和文档。分布式语义学使用向量表示法描述单词或文档的含义，通过一组文档查看分布式证据。向量表示法可以解决监督学习和非监督学习管道中的许多问题。对于语言生成问题，多年来 n - gram 语言模型一直被作为一种传统方法来使用。然而，这些传统方法存在许多缺点，本章将讨论这些缺点。

本章将进一步讨论经典的深度学习体系结构，如循环神经网络（Recurrent Neural Network，RNN）、前馈神经网络（Feed - Forward Neural Network，FFNN）和卷积神经网络（Convolutional Neural Network，CNN）。这些体系结构提高了自然语言处理的性能，并克服了传统方法的局限性。然而，这些模型也存在不同的问题。近年来，Transformer 模型因其在所有自然语言处理任务（从文本分类到文本生成）中的有效性而获得了极大的关注。这主要原因在于 Transformer 有效地提高了多语言和多任务（以及单语言和单任务）自然语言处理的性能。这些贡献提高了自然语言处理中迁移学习（Transfer Learning，TL）的可行性，从而使模型可以用于不同的任务或不同的语言。

本章从注意力机制开始，简要讨论 Transformer 体系结构，以及 Transformer 体系结构与之前自然语言处理模型的差异；在进行理论讨论的同时，展示流行的自然语言处理框架的实际应用例子；为了简洁起见，选择尽可能短小的介绍性代码示例。

在本章中，将介绍以下主题。

（1）自然语言处理到 Transformer 的演变历程。

（2）理解分布式语义。

（3）利用深度学习。

（4）Transformer 体系结构概述。

（5）在迁移学习中结合使用 Transformer。

1.1 技术需求

本章使用 Jupyter Notebook 运行编码练习，要求 Python 版本至少为 3.6.0，并需要使

用 pip install 命令安装以下软件包。

（1）sklearn；

（2）nltk（3.5.0 版本）；

（3）genism（3.8.3 版本）；

（4）fasttext；

（5）keras（不低于 2.3.0 版本）；

（6）Transformers（不低于 4.0.0 版本）。

可以通过以下 GitHub 链接获得本章中所有编码练习的 Jupyter Notebook：

https://github.com/PacktPublishing/Advanced-Natural-LanguageProcessing-with-Transformers/tree/main/CH01。

1.2　自然语言处理到 Transformer 的演变历程

在过去的 20 年中，自然语言处理经历了巨大的变化。在此期间，自然语言处理经历了不同的范式，并最终进入一个主要由 Transformer 体系结构主导的新时代。Transformer 体系结构并不是凭空产生的。基于神经的自然语言处理方法，Transformer 体系结构逐渐演变为一种基于注意力的编码器-解码器类型的体系结构，并且仍在不断发展。该体系结构及其变体之所以取得了很大的成功，主要归功于过去 10 年中以下技术的发展。

（1）基于上下文的词嵌入。

（2）更好的子词级别的标记算法，用于处理不可见词或稀有词。

（3）向句子中注入额外的记忆标记。例如，在 Doc2vec 中注入段落 ID，或者在基于 Transformer 的双向编码表示（Bidirectional Encoder Representations from Transformers，BERT）中注入分类（Classification，CLS）标记。

（4）注意力机制（Attention mechanism）克服了强制输入句子将所有信息编码到一个上下文向量的问题。

（5）多头自注意力（Multi-head self-attention）机制。

（6）根据大小写词序进行位置编码。

（7）有助于更快地训练和微调的可并行化体系结构。

（8）模型压缩（提炼、量化等）。

（9）迁移学习（跨语言、多任务学习）。

多年来，人们一直在使用传统自然语言处理方法，如 n-gram 语言模型（n-gram language model，n 元语言模型）、基于 TF-IDF 的信息检索模型（TF-IDF-based information retrieval model，基于词频-逆向文档频率的信息检索）、独热编码文档术语矩阵（one-hot encoded document-term matrice）等。所有这些方法为解决诸如序列分类（sequence classification）、语言生成（language generation）、语言理解（language understanding）等自然语言处理问题做出了巨大贡献。另外，这些传统的自然语言处理

方法有其自身的弱点。例如，在解决稀疏性、不可见词表示、跟踪长期依赖性等问题方面存在不足。为了克服这些弱点，人们开发了基于深度学习的方法，例如循环神经网络、卷积神经网络、前馈神经网络，以及它们的其他一些变体。

2013 年，作为一个两层前馈神经网络词编码器模型，Word2vec 通过生成短而密集的词表示（称为词嵌入）来解决维度问题。早期的模型设法产生快速有效的静态词嵌入。该模型使用上下文预测目标词或者基于滑动窗口预测相邻词，将无监督文本数据转换为有监督文本数据（自监督学习）。另一个被广泛使用并且也很流行的模型 GloVe 则认为，基于计数的模型可能比神经模型更好。基于计数的模型可以同时利用语料库的全局统计信息和局部统计信息，来学习基于单词共现统计信息的嵌入。这种模型在一些语法和语义任务上表现良好，如图 1－1 所示。图 1－1 表明，术语之间的嵌入偏移有助于面向向量推理的应用。该模型可以学习性别关系的泛化，这是一种语义关系（semantic relation），即来自 Man（男人）和 Woman（女人）之间的偏移量（Man→Woman），然后，可以将表示术语 Actor（男演员）的向量与之前计算的偏移量相加，利用算术方法来估计表示术语 Actress（女演员）的向量。同样，该模型可以学习诸如单词复数形式的语法关系（syntactic relations）。例如，如果给定 Actor、Actors 和 Actress 各自对应的向量，则可以通过以下方式估算 Actresses 的向量。

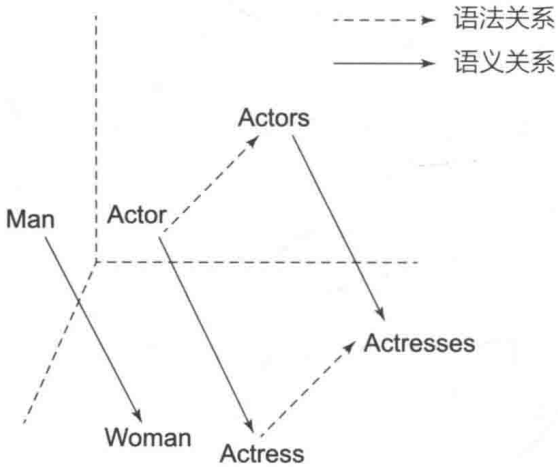


图 1－1　用于关系提取的单词嵌入偏移

循环神经网络和卷积神经网络结构已经用于序列到序列（Sequence－to－Sequence，Seq2Seq）问题中的编码器和解码器。这些早期模型所面临的主要挑战是多义词。由于每个词都有一个固定的表示形式，所以忽略了词的意义，这对于多义词和句子语义来说尤其是一个严重的问题。

与静态词嵌入不同，更先进的神经网络模型，如通用语言模型微调（Universal Language Model Finetuning，ULMFit）和基于语言模型的嵌入（Embedding from Language Model，ELMo）成功地编码了句子级别的信息，并最终缓解了多义问题。这两种重要的方法都是基于长短期记忆神经网络（Long Short Term Memory，LSTM）实现的。这些模

型还引入了预训练和微调的概念，非常有助于应用迁移学习的方法，也就是使用一些预训练过的模型，并基于大量文本数据集的一般任务进行训练。然后，可以通过在监督模式下继续针对目标任务的预先训练网络的训练来轻松执行微调。这些表示不同于传统的单词嵌入，因此每个单词表示都是整个输入句子的函数。现代 Transformer 体系结构充分利用了这一理念。

与此同时，注意力机制的思想在自然语言处理领域产生了深刻影响，并取得了重大成功，特别是在 Seq2Seq 问题上。早期的方法会将从整个输入序列获得的最后一个状态［称为上下文向量（context vector）或思维向量（thought vector）］传递到输出序列，而无须链接（linking）或消除（elimination）。注意力机制能够通过将输入序列中确定的标记链接到输出序列中的特定标记来构建更复杂的模型。例如，假设读者正在完成一项从英语到土耳其语的翻译任务，输入句子中有一个关键词短语“Government of Canada（加拿大政府）”。在输出句子中，Kanada Hükümeti 标记与输入短语建立强连接，并与输入中的其余单词建立弱连接，如图 1－2 所示。

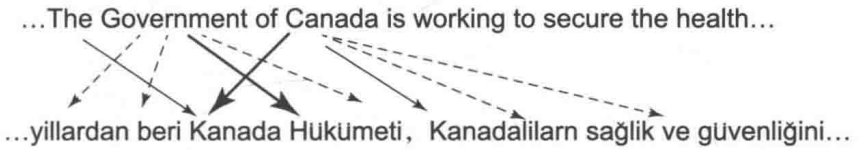


图 1－2　注意力机制的可视化示意

因此，注意力机制使模型在诸如语言翻译、问题回答和文本摘要等 Seq2Seq 问题上更为成功。

2017 年，研究人员提出了基于 Transformer 的编码器－解码器模型，并取得了成功。该设计基于前馈神经网络，摒弃了循环神经网络中的循环，仅使用注意力机制。到目前为止，基于 Transformer 的模型已经克服了其他方法所面临的诸多困难，并已成为一种新的文本处理范式。本书将探索和理解基于 Transformer 模型的工作原理。

1.3　理解分布式语义

分布式语义学使用向量表示法描述单词的含义，倾向于查看单词的分布证据，而不是查看单词预定义的词典定义。该理论认为，在相似的环境中同时出现的单词往往具有相似的含义。该理论首先由学者哈里斯提出［*Distributional Structure Word*（分布式结构单词），1954 年］。例如，类似的单词 dog 和 cat，大多同时出现在同一个上下文中。分布式方法的优点之一是可以帮助研究人员理解和监控词汇在时间和领域上的语义演变，因此也称为词汇语义变化问题（lexical semantic change problem）。

多年来，传统的方法使用词袋（Bag－of－Words，BoW）和 n－gram 语言模型来构建单词和句子的表示。在词袋方法中，单词和文档通过独热编码（one－hot encoding）表示为稀疏矩阵的方式，也称为向量空间模型（Vector Space Model，VSM）。

一直以来，人们通过这一热门编码技术解决了许多自然语言处理问题，如文本分类、词语相似性、语义关系抽取、词义消歧等。另外，n - gram 语言模型将概率分配给单词序列，这样既可以计算一个单词序列属于语料库的概率，也可以基于给定的语料库生成随机单词序列。

1.3.1 词袋技术的实现

词袋是一种通过统计文档中单词出现的次数来表示文档的技术。该技术的主要数据结构是文档术语矩阵（document - term matrix）。接下来讨论使用 Python 实现词袋技术的简单方法。以下代码片段演示了如何使用 Python sklearn 库为一个包含 3 个句子的简单语料库构建文档术语矩阵。

```
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np
import pandas as pd
toy_corpus = ["the fat cat sat on the mat",
              "the big cat slept",
              "the dog chased a cat"]
vectorizer = TfidfVectorizer()
corpus_tfidf = vectorizer.fit_transform(toy_corpus)
print(f"The vocabulary size is \
              {len(vectorizer.vocabulary_.keys())} ")
print(f"The document - term matrix shape is \
              {corpus_tfidf.shape}")
df = pd.DataFrame(np.round(corpus_tfidf.toarray(),2))
df.columns = vectorizer.get_feature_names()
```

上述代码片段的输出结果是一个文档术语矩阵，如图 1 - 3 所示。该矩阵大小为 (3×10)，但在实际应用场景中，文档术语矩阵可能非常巨大，如 $10^4 \times 10^7$。

The vocabulary size is 10
The document-term matrix shape is （3，10）

	big	cat	chased	dog	fat	mat	on	sat	slept	the
0	0.00	0.25	0.00	0.00	0.42	0.42	0.42	0.42	0.00	0.49
1	0.61	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.61	0.36
2	0.00	0.36	0.61	0.61	0.00	0.00	0.00	0.00	0.00	0.36

图 1 - 3 文档术语矩阵

图 1 - 3 中的数据显示了一个基于计数的数学矩阵，其中单元格的值通过词频 - 逆向文档频率（Term Frequency - Inverse Document Frequency，TF - IDF）加权模式进行转换。这种方法不关心单词的位置。由于词序是确定句子含义的重要因素，所以有时忽视词序会导致丢失句子的含义。这是词袋方法中的一个常见问题，最终可以通过循环神经网络中的循环机制和 Transformer 中的位置编码来解决此问题。

矩阵中的每一列表示词汇表中单词的向量，每一行表示文档的向量。用户可以使用语义相似性度量计算单词和文档的相似性或相异性。在大多数情况下，使用诸如 cat_sat 和 the_street 之类的 bigram 来丰富文档的表示。例如，将参数 ngram_range = (1,2) 传递给 TfidfVectorizer 时，将构建一个包含 unigrams(big,cat,dog) 和 bigrams(big_cat,big_dog) 的向量空间。因此，这样的模型也称为 ngrams 包，这是词袋的自然拓展。

如果一个单词经常出现在每个文档中，则可以将其视为高频词，如 and 和 the。相反，有些词很少出现在文档中，则称之为低频（或者罕见）词。由于高频词和低频词可能会妨碍模型的正常工作，所以这里使用词频 - 逆向文档频率作为解决方案，词频 - 逆向文档频率是最重要的，也是众所周知的加权机制之一。

逆向文档频率（Inverse Document Frequency，IDF）是一种统计权重，用于衡量文档中某个词的重要性。例如，虽然单词 the 没有辨别力，但是单词 chased 可以提供大量的信息，并提供有关文本主题的线索。这是因为高频词［停用词（stopword）、虚词（functional word，又称为功能词）］在理解文档时几乎没有什么辨别力。

术语的辨别性也取决于其所涉及的领域。例如，在有关深度学习方面的文章中，很可能几乎在每个文档中都包含 network 这个单词。逆向文档频率可以使用所有术语（单词）的文档频率（Document Frequency，DF）来减小术语的权重，其中单词的文档频率基于出现术语的文档数。术语频率（Term Frequency，TF）是文档中术语的原始计数。

基于词频 - 逆向文档频率的词袋模型的优、缺点如表 1 - 1 所示。

表 1 - 1　基于词频 - 逆向文档频率的词袋模型的优、缺点

优点	缺点
• 易于实现 • 结果易于解释 • 适配于应用领域	• 会出现维度灾难 • 无法解决不可见单词的问题 • 难以捕捉语义关系，如属于（is - a）关系、包含（has - a）关系、同义词等 • 忽略词序信息 • 对于大型词汇表，速度较慢

1.3.2　克服维度问题

为了克服词袋模型的维度灾难问题，潜在语义分析（Latent Semantic Analysis，LSA）被广泛用于在低维空间获取语义。这是一种线性方法，可以捕获术语之间的成对相关性。基于潜在语义分析的概率方法仍然可以看作一层隐藏的主题变量。然而，当前的深度学习模型包含多个隐藏层，具有数十亿个参数。此外，研究表明基于 Transformer 的模型比传统模型更能发现潜在的表示。

对于自然语言理解（Natural Language Understanding，NLU）任务，传统的管道首先

是预处理步骤，如标记化（tokenization）、词干提取（stemming）、名词短语检测（noun phrase detection）、组块分析（chunking）、停用词消除（stop - word elimination）等；然后，使用任一加权模式（其中词频 - 逆向文档频率是最常用的一种）构造文档术语矩阵；最后，文档术语矩阵被用作机器学习管道、情感分析、文档相似性、文档聚类，或者对查询与文档之间的相关性得分之类的表格式数据输入进行度量。类似地，文档术语表示为表格式矩阵，然后可以作为标记分类问题的输入，从而实现命名实体识别、语义关系抽取等任务。

分类阶段包括有监督的机器学习算法的直接实现，如支持向量机（Support Vector Machine，SVM）、随机森林、逻辑回归、朴素贝叶斯和多学习器［提升法（Boosting）或装袋法（Bagging）］。实际上，可以简单地通过以下代码片段实现这种管道。

```
from sklearn.pipeline import make_pipeline
from sklearn.svm import SVC
labels = [0,1,0]
clf = SVC()
clf.fit(df.to_numpy(), labels)
```

如上面的代码片段所示，借助 sklearn 应用程序编程接口（Application Programming Interface，API），可以轻松地应用 fit 操作。为了将学习模型应用于训练数据，可以执行以下代码片段。

```
clf.predict(df.to_numpy())
Output: array([0, 1, 0])
```

下面继续下一小节的学习。

1.3.3 语言建模与生成

对于语言生成问题，传统的方法是利用 n - gram 语言模型来实现。这种方法也称为马尔可夫过程（Markov process），这是一种随机模型，其中每个单词（事件）取决于前面单词（称为 unigram、bigram 或 n - gram）的子集。

（1）unigram（所有单词都是独立的，不存在链）：用于估算单词在一个词汇表中的概率，即简单地计算该单词出现的频数与单词总数之比。

（2）bigram（一阶马尔可夫过程）：用于估算概率 $P(\text{word}_i \mid \text{word}_{i-1})$。$\text{word}_i$ 的概率取决于 word_{i-1}，即简单地计算概率 $P(\text{word}_i, \text{word}_{i-1})$ 与 $P(\text{word}_{i-1})$ 的比率。

（3）n - gram（n 阶马尔可夫过程）：用于估算概率 $P(\text{word}_i \mid \text{word}_0, \cdots, \text{word}_{i-1})$。

接下来，使用自然语言工具软件包（Natural Language Toolkit，NLTK）构造一个简单的语言模型进行实现。在下面的实现过程中，训练阶数 $n = 2$ 的最大似然估计器（Maximum Likelihood Estimator，MLE）。可以选择任意阶数的 n - gram。例如，$n = 1$ 表示 unigrams；$n = 2$ 表示 bigrams；$n = 3$ 表示 trigrams，依此类推。

```
import nltk
from nltk.corpus import gutenberg
from nltk.lm import MLE
from nltk.lm.preprocessing import padded_everygram_pipeline
nltk.download('gutenberg')
nltk.download('punkt')
macbeth = gutenberg.sents('shakespeare-macbeth.txt')
model, vocab = padded_everygram_pipeline(2, macbeth)
lm=MLE(2)
lm.fit(model,vocab)
print(list(lm.vocab)[:10])
print(f"The number of words is {len(lm.vocab)}")
```

nltk 软件包首先下载古腾堡语料库（gutenberg corpus），其中包括古腾堡项目（Project Gutenberg）电子文本档案中的一些文本，该项目的官网地址为 https://www.gutenberg.org。nltk 软件包还下载用于分词过程的 punkt 分词器工具。punkt 分词器使用无监督机器学习算法将原始文本划分为句子列表。nltk 软件包包含了一个经过预训练的英语 punkt 分词器模型，用于缩写词和常用语搭配。在使用之前，可以对任何语言的文本列表进行训练。在接下来的章节中，将讨论如何为 Transformer 模型训练不同且更高效的分词器。以下代码片段生成了语言模型迄今所学习到的内容。

```
print(f"The frequency of the term 'Macbeth' is {lm.
counts['Macbeth']}")
print(f"The language model probability score of 'Macbeth' is
{lm.score('Macbeth')}")
print(f"The number of times 'Macbeth' follows 'Enter' is {lm.
counts[['Enter']]['Macbeth']} ")
print(f"P(Macbeth | Enter) is {lm.score('Macbeth',
['Enter'])}")
print(f"P(shaking | for) is {lm.score('shaking', ['for'])}")
```

上述代码片段的输出结果如下所示。

```
The frequency of the term 'Macbeth' is 61
The language model probability score of 'Macbeth' is 0.00226
The number of times 'Macbeth' follows 'Enter' is 15
P(Macbeth | Enter) is 0.1875
P(shaking | for) is 0.0121
```

n-gram 语言模型保持 n-gram 计数并计算句子生成的条件概率。lm = MLE(2) 表示最大似然估计，根据每个标记概率生成最可能的句子。以下代码片段基于给定 <s> 作为起始条件，生成一个由 10 个单词组成的随机句子。

```
lm.generate(10, text_seed=['<s>'], random_seed=42)
```

其输出结果显示在以下代码片段中。

```
['My','Bosome','franchis',"'",'s','of','time',',','We','are']
```

可以通过 text_seed 参数指定一个特定的起始条件，以生成基于前面的上下文条件。在上述示例中，前面的上下文是 <s>，这是一个特殊的标记，表示句子的开头。

到目前为止，本节讨论了基于传统自然语言处理模型的范式，并提供了使用流行框架的非常简单的实现过程。接下来讨论深度学习部分，探讨神经网络语言模型如何影响自然语言处理领域，以及如何克服传统模型的局限性。

1.4 利用深度学习

深度学习体系结构得到了广泛和成功的应用，自然语言处理是其中的一个应用领域。几十年来，我们见证了各类成功的体系结构，特别是在单词和句子表示方面。本节讨论这些方法的发展历程，而这些方法均基于常用的框架来实现。

1.4.1 学习单词嵌入

基于神经网络的语言模型有效地解决了特征表示和语言建模问题，因为这种模型可以在更大的数据集上训练更复杂的神经结构，以构建简短而密集的表示。2013 年，Word2vec 模型（一种流行的单词嵌入技术）使用一种简单有效的体系结构来学习连续单词的高质量表示。Word2vec 模型在各种句法和语义语言任务［如情感分析（sentiment analysis）、释义检测（paraphrase detection）、关系抽取（relation extraction）等］方面都优于其他模型。该模型流行的另一个关键原因是其具有较低的计算复杂度。该模型最大限度地提高了当前单词在任何上下文中出现的概率，反之亦然。

下面的代码片段演示了如何为戏剧 *Macbeth*（麦克白）的句子训练单词向量。

```
from gensim.models import Word2vec
model = Word2vec(sentences = macbeth, size = 100, window = 4, min_count = 10,
workers = 4, iter = 10)
```

上述代码片段通过一个滑动的长度为 5 的上下文窗口来训练向量大小为 100 的单词嵌入。为了可视化单词嵌入，需要通过应用主成分分析（Principal Component Analysis，PCA）将维度减少到 3。代码如下所示。

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import random
np.random.seed(42)
words = list([e for e in model.wv.vocab if len(e) > 4])
random.shuffle(words)
words3d = PCA(n_components = 3,random_state = 42).fit_
```

```
transform(model.wv[words[:100]])
def plotWords3D(vecs, words, title):
...
plotWords3D(words3d, words, "Visualizing Word2vec Word
Embeddings using PCA")
```

输出结果如图 1 -4 所示。

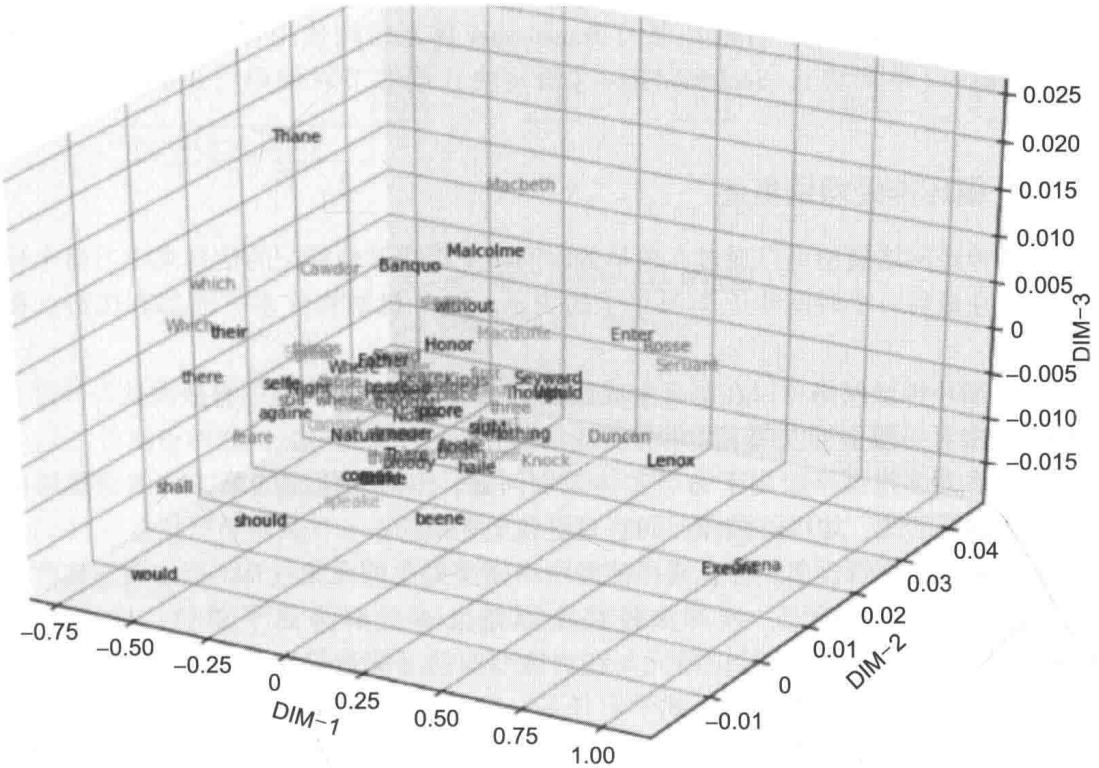


图 1 -4　基于 PCA 单词嵌入的可视化

在图 1 -4 中，莎士比亚戏剧中的主要角色（Macbeth、Malcolm、Banquo、Macduff 等）相互之间非常接近。同样，助动词（shall、should 和 would）在图 1 -4 的左下角彼此靠近。还可以使用嵌入函数集来捕捉类比关系，例如 man - woman = uncle - aunt。有关此主题的更多有趣的视觉示例，请参阅以下项目 https://projector. tensorflow. org/。

类似 Word2vec 的模型使用基于预测的神经结构来学习单词嵌入。这些模型对一些目标函数和附近的单词预测采用梯度下降法。传统方法采用基于计数的方法，而神经模型为分布式语义设计了基于预测的体系结构。对于基于计数的方法还是基于预测的方法，哪种方法最适合分布式单词表示呢？GloVe 方法回答了这个问题，认为这两种方法并没有显著的区别。Jeffrey Penington 等甚至支持以下观点：基于计数的方法可以捕获全局统计数据，因此更加有效。他们指出，GloVe 方法在单词类比、单词相似性和命名实体识别（Named Entity Recognition，NER）任务方面优于其他神经网络语言模型。

然而，这两种范式并没有为不可见词和词义问题提供有用的解决方案。这些范式没有利用子词信息，因此无法学习稀有词和不可见词的嵌入。

FastText（另一个广泛使用的模型）提出了一种新的使用子词信息的扩展方法，其中每个词都表示为一袋字符 n - gram。该模型为每个字符 n - gram 设置一个常量向量，并将单词表示为其子向量之和，该思想是 Hinrich Schütze（*Word Space*，1993 年）首次提出的。该模型甚至可以计算不可见词的表示，并学习单词的内部结构，如后缀/词缀，这对于形态丰富的语言（如芬兰语、匈牙利语、土耳其语、蒙古语、韩语、日语、印度尼西亚语等）尤为重要。目前，现代 Transformer 体系结构使用各种子词标记化方法，如 WordPiece（单词块）、SentencePiece（语句块）或字节对编码（BytePair Encoding，BPE）。

1.4.2 循环神经网络概述

循环神经网络模型可以通过在较早的时间步上汇总其他标记的信息来学习每个标记表示，并在最后一个时间步上学习句子的表示。这一机制在许多方面都有优势，概述如下。

（1）循环神经网络可以在语言生成或音乐生成的一对多模型中重新设计。

（2）多对一模型可以用于文本分类或情感分析。

（3）多对多模型可以用于命名实体识别问题。多对多模型的第二个用途是解决编码器 - 解码器问题，如机器翻译、问答系统和文本摘要。

与其他神经网络模型一样，循环神经网络模型采用词元化（tokenization，又称为标记化）算法生成的标记，该算法将整个原始文本分解为原子单位，也称为标记（tokens）。此外，该算法将标记单元与在训练期间学习的数值向量（令牌嵌入）关联。作为替代方案，可以事先将嵌入式学习任务分配给著名的单词嵌入算法，如 Word2vec 或 FastText。

下面是一个简单的循环神经网络体系结构示例，用于处理语句“The cat is sad.”。其中，$\boldsymbol{x}_0$ 是 The 的向量嵌入；$\boldsymbol{x}_1$ 是 cat 的向量嵌入，依此类推。图 1 - 5 显示了循环神经网络被展开为一个全深度神经网络（Deep Neural Network，DNN）。

展开（Unfolding）意味着将每个单词与一个层关联。对于语句系列“The cat is sad.”，处理由 5 个单词组成的一个系列。每层中的隐藏状态充当网络的存储器。它对所有前面的时间步和当前时间步中发生的情况进行编码，如图 1 - 5 所示。

循环神经网络体系结构的优点如下。

（1）可变长度输入：处理可变长度输入的能力，与输入的句子大小无关。可以在不改变参数的情况下为网络提供 3 ~ 300 个单词的句子。

（2）关注词序：按顺序逐字处理序列，关注单词的位置。

（3）适用于各种模式（多对多、一对多）：可以使用相同的循环范式训练机器翻译模型或情感分析。这两种体系结构都基于循环神经网络。

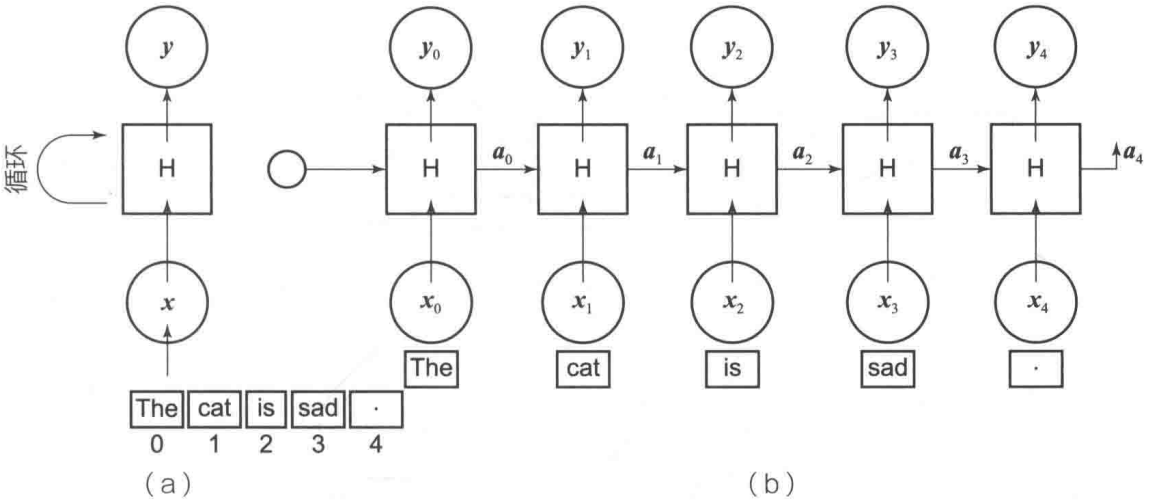


图 1－5　一个循环神经网络体系结构

（a）折叠表示；（b）展开表示

循环神经网络体系结构的缺点如下。

（1）长期依赖性问题：当处理一个很长的文档并尝试将彼此相距很远的术语相链接时，需要关注这些术语之间所有不相关的其他术语，并对其编码。

（2）容易出现梯度爆炸或梯度消失的问题：在处理长文档时，更新第一个单词的权重非常重要，因为权重的设置会使模型由于梯度消失问题而无法训练。

（3）难以应用并行化训练：并行化将主要问题分解为一个较小的问题，并同时执行解决方案，但循环神经网络遵循经典的顺序方法。每一层都强烈地依赖前面的层，这使并行化变得不可能实现。

（4）当序列较长时计算速度会变慢：循环神经网络对于短文本问题可能非常有效。除了长期依赖性问题外，循环神经网络处理较长文档的速度也非常慢。

尽管从理论上讲，循环神经网络可以处理许多时间步的信息，但在现实世界中，诸如长文档和长期依赖性之类的问题是不可能轻易被发现的。

1.4.3　长短期记忆网络和门控循环单元

长短期记忆网络和门控循环单元（Gated Recurrent Units，GRU）是循环神经网络的新变体，它们解决了长期依赖性问题，并引起了广泛的关注。长短期记忆网络是专门为解决长期依赖性问题而开发的。长短期记忆网络模型的优点是使用附加的单元状态，即长短期记忆网络单元顶部的水平序列线。该单元状态由用于遗忘、插入或更新操作的专用门来控制。长短期记忆网络体系结构的复杂单元如图 1－6 所示。

长短期记忆网络体系结构可以确定以下事项。

（1）在单元状态中存储何种信息。

（2）哪些信息将被遗忘或删除。

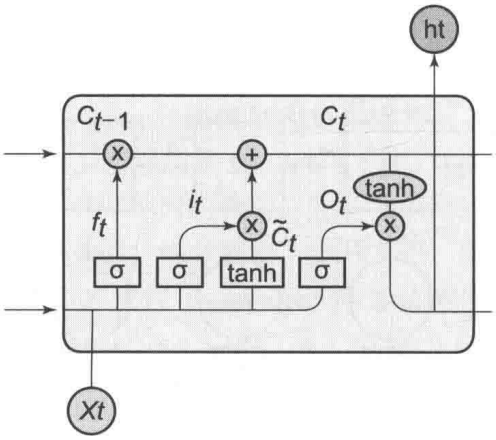


图 1-6 长短期记忆网络体系结构的复杂单元

在原始的循环神经网络中，为了学习标记 I 的状态，将在 $timestep_0$ 和 $timestep_{i-1}$[①] 之间重复处理先前标记的整个状态。携带早期时间步的全部信息会导致梯度消失的问题，这使模型无法训练。长短期记忆网络中的门机制允许体系结构在特定的时间步跳过一些不相关的标记，或者记住远程状态，以便学习当前所标记的状态。

门控循环单元在许多方面与长短期记忆网络相似，主要区别在于门控循环单元不使用单元状态。门控循环单元通过将单元状态的功能转移到隐藏状态，简化了体系结构。门控循环单元只包括两个门：更新门（update gate）和重置门（reset gate）。更新门确定向前推送上一个和当前时间步中的信息量。该特性有助于模型保留来自过去的相关信息，从而最大限度地降低梯度消失问题的风险。重置门检测不相关的数据并使模型忘记这些数据。

下面介绍如何使用 Keras 简单地实现长短期记忆网络。

首先需要从通用语言理解评估（General Language Understanding Evaluation，GLUE）基准下载 Stanford Sentiment Treebank（SST-2）情感数据集。可以运行以下代码来下载该数据集。

```
$ wget https://dl.fbaipublicfiles.com/glue/data/SST-2.zip
$ unzip SST-2.zip
```

重要提示

SST-2 是一个完全标记的解析树，可以使用英语进行完整的情感分析。该语料库最初由约 12 000 个从电影评论中提取的单个句子组成。使用斯坦福解析器进行解析，包含超过 20 万个独特的短语，每个短语都由三位人类评判员进行标注。

① 此处原著有误，译者做了更正。——译者注

在下载 SST－2 情感数据集后，将其读取为 pandas 对象。实现代码如下所示。

```
import tensorflow as tf
import pandas as pd
df = pd.read_csv('SST-2/train.tsv',sep = "\t")
sentences = df.sentence
labels = df.label
```

需要设置最大句子长度以构建词汇表和字典（word2idx、idx2words），最后将每个句子表示为索引列表而不是字符串。可以运行以下代码来实现上述功能。

```
max_sen_len = max([len(s.split()) for s in sentences])
words = ["PAD"] + \
        list(set([w for s in sentences for w in s.split()]))
word2idx = {w:i for i,w in enumerate(words)}
max_words = max(word2idx.values()) + 1
idx2word = {i:w for i,w in enumerate(words)}
train = [list(map(lambda x:word2idx[x], s.split())) \
        for s in sentences]
```

长度小于 max_sen_len（最大句子长度）的序列使用一个 PAD 值进行填充，直到序列的长度达到 max_sen_len。另外，较长的序列被截断，以使这些序列适合 max_sen_len。以下是具体的实施方案。

```
from keras import preprocessing
train_pad = preprocessing.sequence.pad_sequences(train,
                                    maxlen = max_sen_len)
print('Train shape:', train_pad.shape)
Output: Train shape: (67349, 52)
```

然后准备设计和训练长短期记忆网络模型。实现代码如下所示。

```
from keras.layers import LSTM, Embedding, Dense
from keras.models import Sequential
model = Sequential()
model.add(Embedding(max_words, 32))
model.add(LSTM(32))
model.add(Dense(1, activation = 'sigmoid'))
model.compile(optimizer = 'rmsprop',loss = 'binary_crossentropy',
metrics = ['acc'])
history = model.fit(train_pad,labels, epochs = 30, batch_size = 32,
validation_split = 0.2)
```

该模型训练 30 个 epoch（学习迭代次数，或者称为训练阶段、训练周期）。为了绘制长短期记忆网络模型迄今为止学习的结果，可以执行以下代码。

```
import matplotlib.pyplot as plt
def plot_graphs(history, string):
...
plot_graphs(history,'acc')
plot_graphs(history,'loss')
```

以上代码将生成图 1－7 所示的曲线图，该图展示了基于长短期记忆网络的文本分类的训练性能和验证性能。

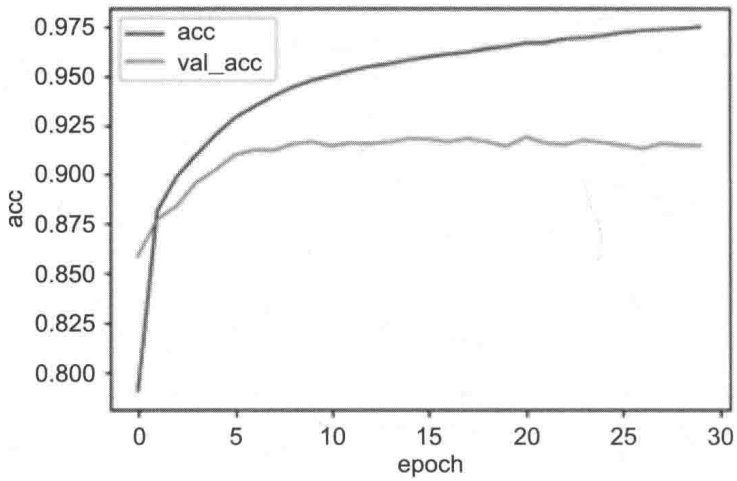


图 1－7　长短期记忆网络的分类性能

如前所述，基于循环神经网络的编码器－解码器模型的主要问题在于，它为序列生成单一的固定表示。然而，注意力机制允许循环神经网络在将输入标记映射到输出标记的特定部分时，将注意力集中在输入标记的特定部分上。注意力机制已经被发现是有用的，并且已经成为 Transformer 体系结构的基本思想之一。我们将在下一部分和整本书中讨论 Transformer 体系结构如何利用注意力机制。

1.4.4　卷积神经网络概述

卷积神经网络在计算机视觉领域取得成功后，被移植到自然语言处理中，用于句子建模或语义文本分类等任务。在许多实践中，卷积神经网络由卷积层和密集的神经网络组成。卷积层在数据上执行，以便提取有用的特征。与所有深度学习模型一样，卷积层扮演着自动特征提取的角色。在自然语言处理的情况下，该特征层由一个嵌入层提供，而嵌入层则以一个独热向量化格式将句子作为输入。独热向量由构成句子的每个单词的 token－id（标记 id）生成。图 1－8 中的左半部分显示了一个句子的独热表示方式。

	1	2	3	4	5
I	1	0	0	0	0
saw	0	1	0	0	0
a	0	0	1	0	0
cat	0	0	0	1	0
.	0	0	0	0	1

图 1－8　独热向量

每个标记（分别由一个独热向量来表示）被馈送到嵌入层。嵌入层可以初始化为

随机值或者使用预先训练好的词向量（如 GloVe、Word2vec 或 FastText）；然后，一个句子将被转换成 $N \times E$ 形状的密集矩阵（其中，N 是句子中的标记数量，E 是嵌入大小）。包含 5 个标记的句子的一维卷积神经网络如图 1－9 所示。

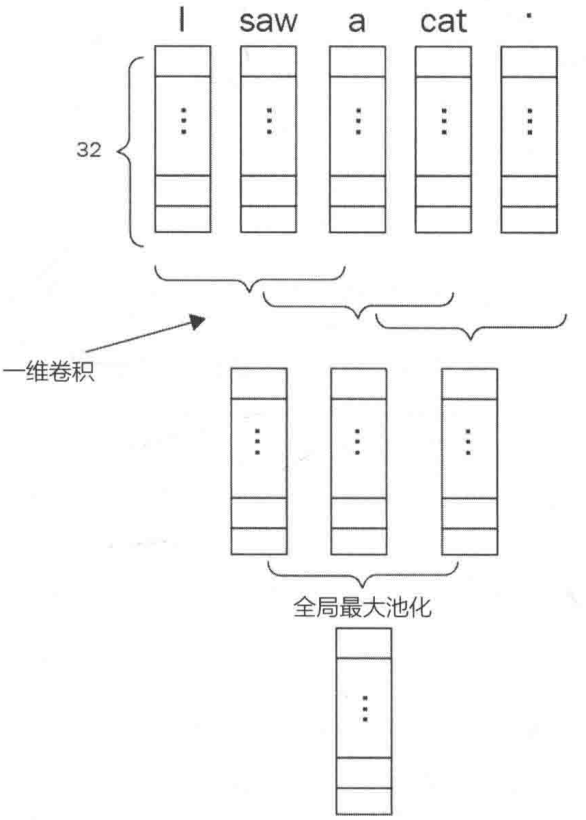


图 1－9　包含 5 个标记的句子的一维卷积神经网络

卷积将在不同层和核的操作上进行。卷积层的超参数是核大小和核的数量。同样值得注意的是，这里应用了一维卷积，其原因是标记嵌入不能被视为部分的，用户可能希望应用那些能够以连续顺序同时看到多个标记的核。可以将其视为具有指定窗口的 n－gram。将浅层迁移学习与卷积神经网络模型结合使用也是此类模型的另一个特色。如图 1－10 所示，还可以通过多种标记表示的组合来传播网络，这是 Yoon Kim 在 2014 年的研究“*Convolutional Neural Networks for Sentence Classification*（用于句子分类的卷积神经网络）”中所提出的思想。

例如，可以使用三个嵌入层而不是一个嵌入层，并为每个标记连接这些嵌套层。基于此设置，如果所有三种不同嵌入的嵌入大小均为 128，则像 fell 这样的标记，其向量大小将为 3×128。这些嵌入可以使用 Word2vec、GloVe 和 FastText 中预先训练的向量进行初始化。每个步骤的卷积运算将观察到 N 个字及其各自的三个向量（N 是卷积滤波器的大小）。这里使用的卷积类型是一维卷积。此处的维度表示进行操作时可能发生的移动。例如，二维卷积将沿两个轴移动，而一维卷积仅沿一个轴移动。表 1－2 所示为各个维度卷积之间的差异。

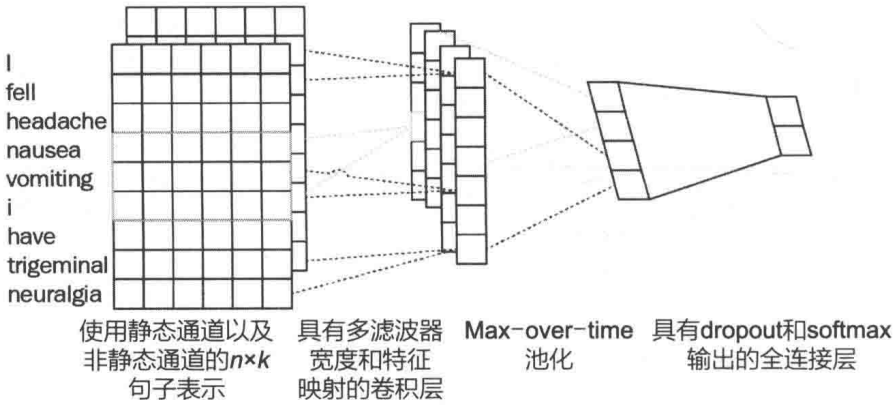


图 1－10　卷积神经网络中多种表示的组合

表 1－2　各个维度卷积之间差异

卷积方向	输入	滤波	输出
一个方向	3维	3维	2维
两个方向	4维	4维	3维
三个方向	5维	5维	4维

下面的代码片段是一维卷积神经网络的实现，处理长短期记忆网络管道中使用的相同数据，其中包括 Conv1D 和 MaxPooling 层，然后是 GlobalMaxPooling 层。可以通过调整参数以及添加更多层来优化模型，从而扩展管道。

```
from keras import layers
model = Sequential()
model.add(layers.Embedding(max_words, 32, input_length=max_sen_len))
model.add(layers.Conv1D(32, 8, activation='relu'))
model.add(layers.MaxPooling1D(4))
model.add(layers.Conv1D(32, 3, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1, activation='sigmoid')
model.compile(loss='binary_crossentropy', metrics=['acc'])
history = model.fit(train_pad,labels, epochs=15, batch_size=32,
validation_split=0.2)
```

结果表明，卷积神经网络模型的性能与长短期记忆网络模型的性能相当。虽然卷积神经网络已成为图像处理的标准，但用户也看到了卷积神经网络在自然语言处理中的许多成功应用案例。长短期记忆网络模型适用于识别跨时间的模式，而卷积神经网络模型则适用于识别跨空间的模式。

1.5　Transformer 体系结构概述

Transformer 模型因其在从文本分类到文本生成的一系列自然语言处理问题中的有效性而受到广泛关注。注意力机制是这些模型的重要组成部分，并且起着非常关键的作用。在 Transformer 模型之前，注意力机制被提出作为改进传统深度学习模型（如循环神经网络）的助手。为了更好地理解 Transformer 及其对自然语言处理的影响，本节首先研究注意力机制。

1.5.1　注意力机制

Bahdanau 等（2015 年）提出了注意力机制的最初变体之一。该机制基于循环神经网络模型（如门控循环单元或长短期记忆网络）在诸如神经机器翻译（Neural Machine Translation，NMT）等任务上具有信息瓶颈的事实。这些基于编码器 - 解码器的模型以 token - id 的形式获取输入，并以循环方式（编码器）对其进行处理，然后，处理后的中间表示被馈送到另一个循环单元（解码器）以提取结果。这种类似雪崩的信息就像一个滚动的雪球，吸收了所有的信息；而对解码器部分而言，滚动该雪球是非常困难的，因为解码器部分看不到所有的依赖项，只能获得中间表示（上下文向量）作为输入。

为了协调这一机制，Bahdanau 提出了一种注意力机制，在中间隐藏值上使用权重。这些权重调整了模型在每个解码步骤中必须在输入上使用的注意力数量。这种出色的指导可以帮助模型完成特定的任务，如神经机器翻译，而神经机器翻译是一项多对多的任务。图 1 - 11 所示为典型的注意力机制示意。

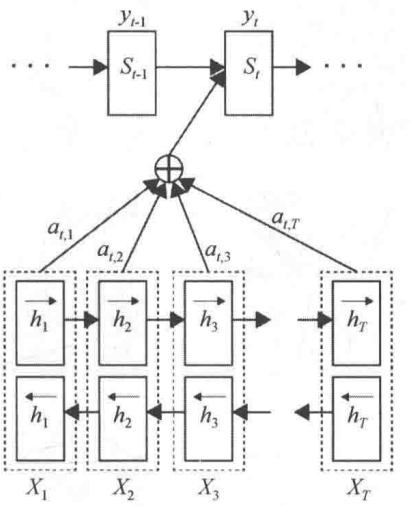


图 1 - 11　典型的注意力机制示意

研究人员提出了不同的注意力机制，并且分别进行了不同的改进。这些注意力机制系列包括加法型（additive）注意力机制、乘法型（multiplicative）注意力机制、通用型

（general）注意力机制、点积型（dot－product）注意力机制。点积型注意力机制是一个带有缩放参数的修改版本，被称为缩放点积注意力机制（scaled dot－product attention）。这个特定的注意力类型是 Transformer 模型的基础，被称为多头注意力机制（multi－head attention mechanism）。加法型注意力机制也是早期在神经机器翻译任务中引入的一个显著特点。注意力机制的类型如表 1－3 所示。

表 1－3　注意力机制的类型

名称	注意力评分函数
基于内容的注意力机制	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$
加法型注意力机制	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = v_a^{\mathrm{T}} \tanh(\boldsymbol{W}_a [\boldsymbol{s}_i; \boldsymbol{h}_i])$
基于位置的注意力机制	$\alpha_{t,i} = \text{softmax}(\boldsymbol{W}_a \boldsymbol{s}_t)$
通用型注意力机制	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^{\mathrm{T}} \boldsymbol{W}_a \boldsymbol{h}_i$
点积型注意力机制	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_i^{\mathrm{T}} \boldsymbol{h}_i$
缩放点积注意力机制	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_i^{\mathrm{T}} \boldsymbol{h}_i}{\sqrt{n}}$

注意力机制并不是自然语言处理特有的特点，它们也被用于计算机视觉、语音识别等不同领域的不同用例中。图 1－12 显示了用于神经图像字幕训练的多模式方法的可视化。

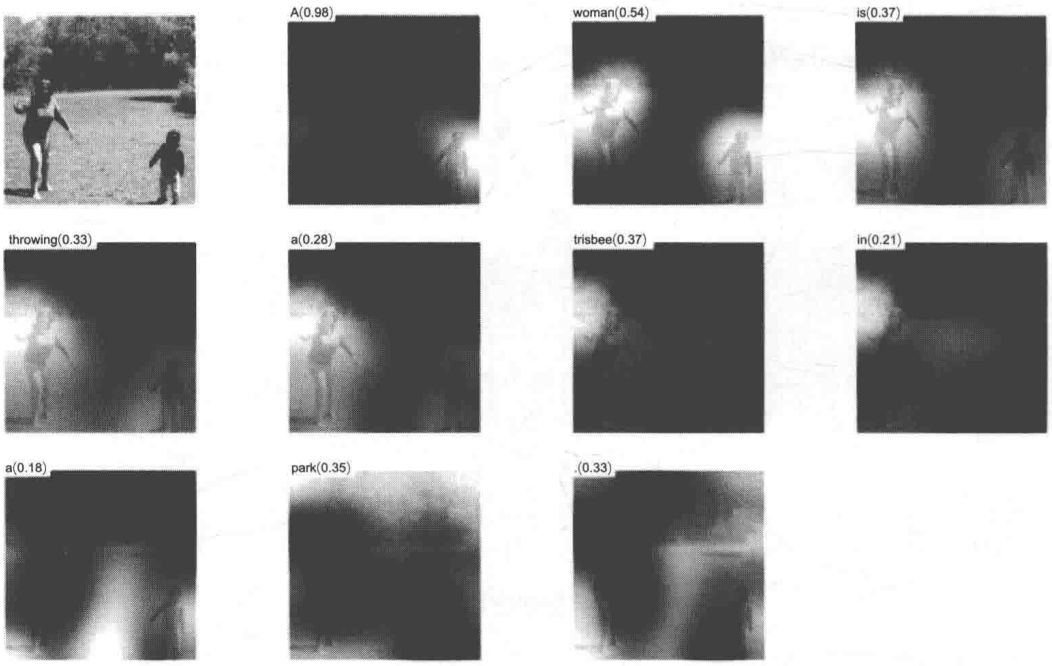


图 1－12　计算机视觉中的注意力机制

图 1－13 所示的多头注意力机制是 Transformer 体系结构的重要组成部分。

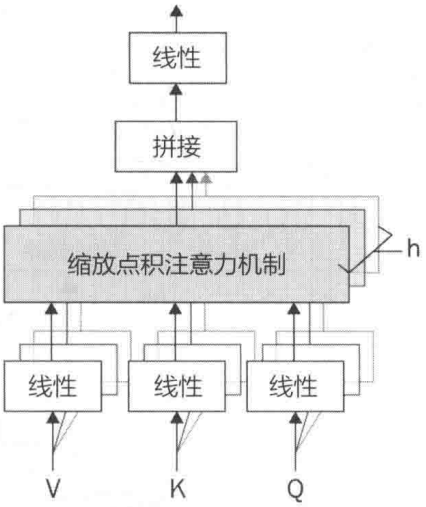


图 1－13　多头注意力机制

接下来讨论多头注意力机制。

1.5.2　多头注意力机制

在讨论缩放点积注意力机制之前，需要充分理解自注意力机制。如图 1－15 所示，自注意力机制是一种缩放自注意力机制的基本形式。该注意力机制使用输入矩阵（图 1－15 中表示为 $\boldsymbol{X}$），并在 $\boldsymbol{X}$ 中的各个项目之间生成注意力得分。将 $\boldsymbol{X}$ 视为一个 3×4 矩阵，其中，3 表示标记的数量，4 表示嵌入的大小。图 1－15 中的 $\boldsymbol{Q}$ 表示查询（query），$\boldsymbol{K}$ 表示键（key），$\boldsymbol{V}$ 表示值（value）。三种类型的矩阵（图 1－15 中分别为 $\boldsymbol{\theta}$，$\boldsymbol{\phi}$ 和 $\boldsymbol{g}$）在生成 $\boldsymbol{Q}$，$\boldsymbol{K}$ 和 $\boldsymbol{V}$ 之前乘以 $\boldsymbol{X}$。查询（$\boldsymbol{Q}$）和键（$\boldsymbol{K}$）之间的相乘结果生成注意力得分矩阵。这也可以看作一个数据库，在其中使用键进行查询，以找出在数值评估方面有多少不同的项是相关的。将注意力得分与 $\boldsymbol{V}$ 矩阵相乘，将得到这种注意力机制的最终结果。这种机制被称为自注意力机制的主要原因在于，其统一的输入为 $\boldsymbol{X}$，而 $\boldsymbol{Q}$，$\boldsymbol{K}$ 和 $\boldsymbol{V}$ 都是由 $\boldsymbol{X}$ 计算出来的。图 1－14 描述了上述内容。

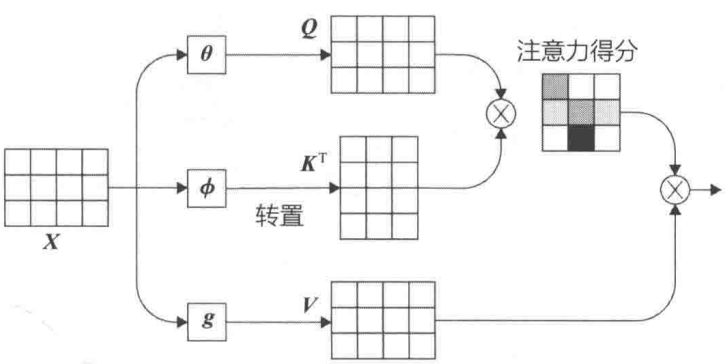


图 1－14　注意力机制的数学表示

缩放点积注意力机制与自注意力（点积）机制非常相似，其区别在于是否使用了缩放因子。另外，多头自注意力确保模型能够在所有级别下查看输入的各个方面。Transformer 模型关注编码器注解以及之前各个层中的隐藏值。Transformer 模型体系结构没有包含循环的分步流程；相反，它使用位置编码，以便获得关于输入序列中每个标记的位置信息。嵌入值（随机初始化）和位置编码的固定值拼接在一起作为输入，并馈入第一个编码器部分的各个层，然后沿着体系结构进行传播，如图 1－15 所示。

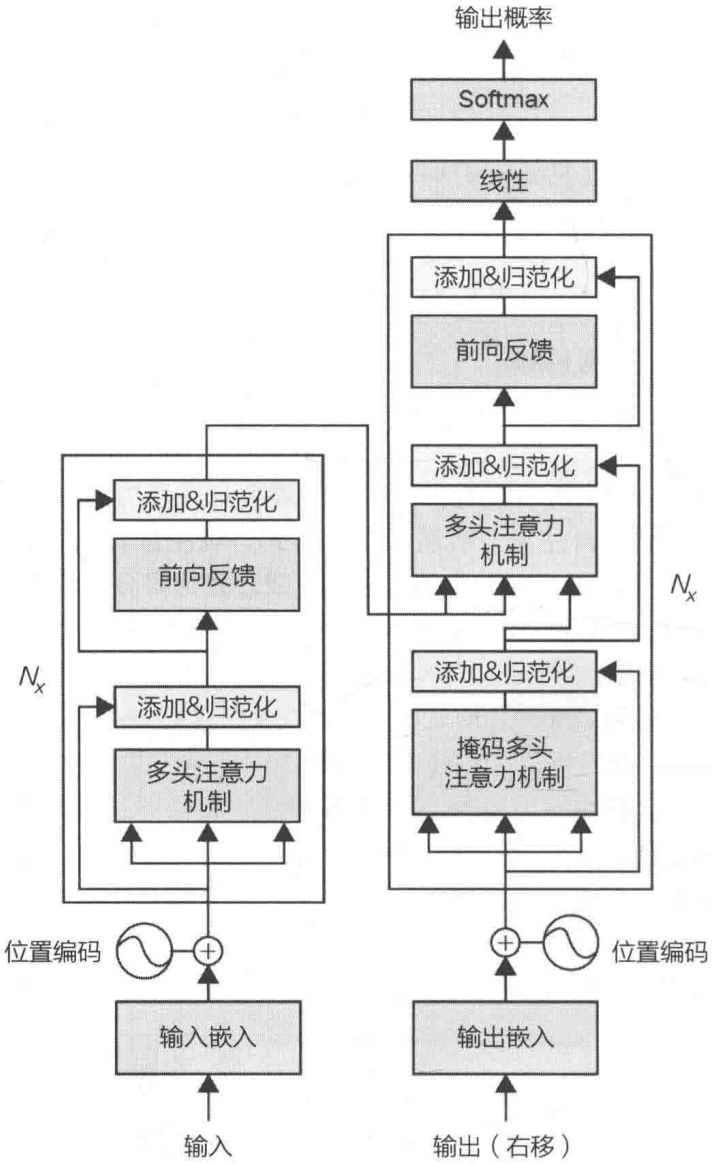


图 1－15　一个 Transformer 示例

通过计算不同频率下的正弦波和余弦波可以获得位置信息。图 1－16 所示为位置编码的示例。

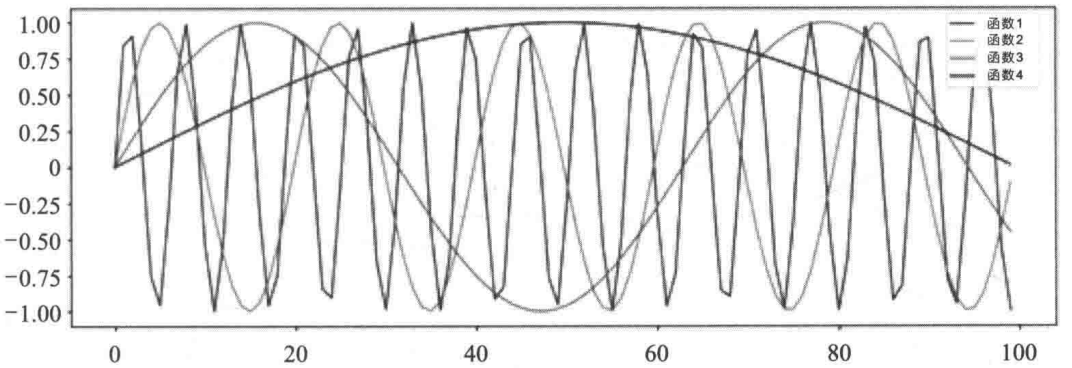


图 1－16　位置编码的示例

图 1－17 给出了 Transformer 体系结构和缩放点积注意力机制性能的经典示例。

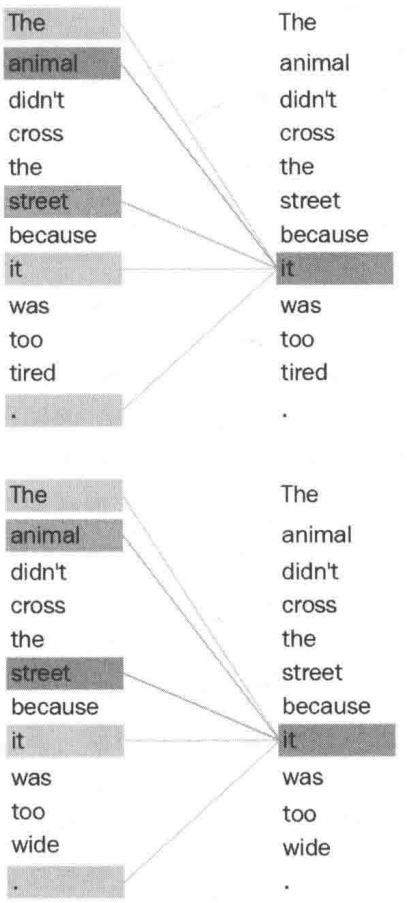


图 1－17　Transformer 的注意力映射

从图 1－17 中可以看出，单词 it 指示不同上下文中的不同实体。Transformer 体系结构的另一个改进是并行处理。传统的顺序递归模型（如长短期记忆网络和门控循环单元）并不具备并行处理能力，因为它们按逐个标记的方式处理输入。另外，前馈层的速度要快一点，因为单矩阵乘法远比循环单元快。多头注意力层的堆栈可以更好地理解复杂的句子。图 1－18 所示为多头注意力机制的一个经典视觉示例。

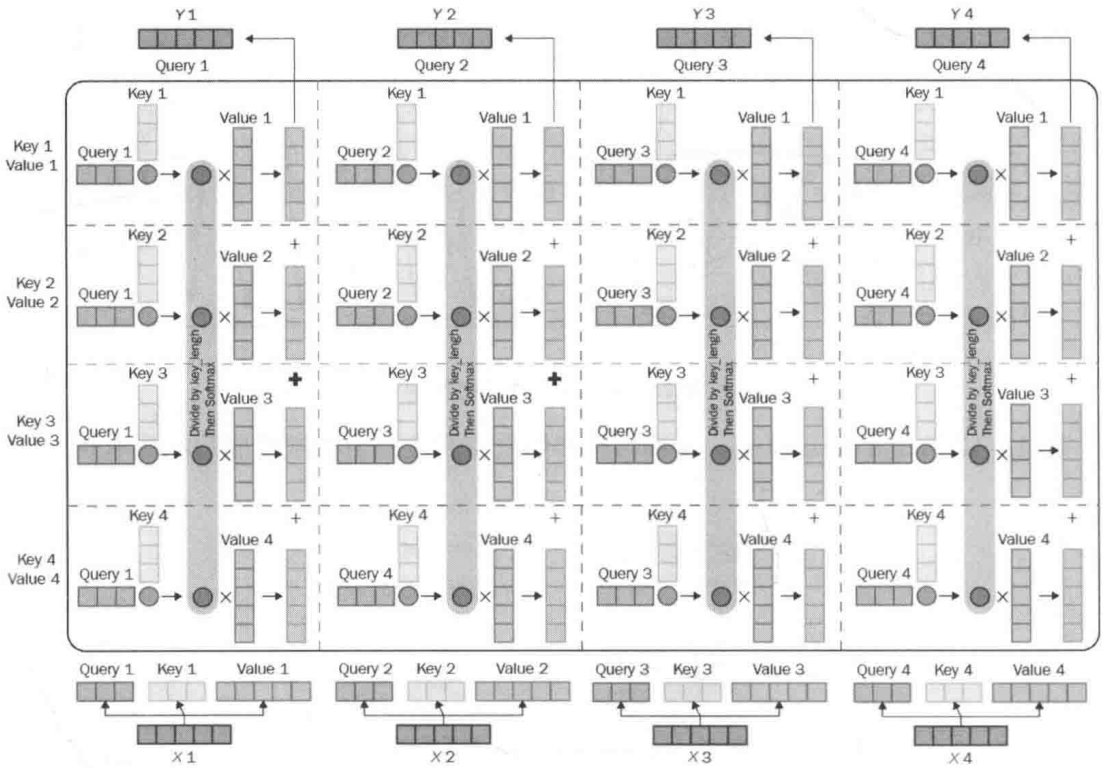


图 1－18　多头注意力机制的一个经典视觉示例

在注意力机制的解码器端，使用了与编码器非常相似的方法，只是稍加修改。多头注意力机制完全相同，但也使用编码器堆栈的输出。该编码被赋予第二个多头注意层中的每个解码器堆栈。这是一个小小的改进，在解码时引入了编码器堆栈的输出。这种改进使模型在解码时能够感知编码器的输出，同时在训练期间帮助模型在各个层具有更好的梯度流。解码器层末尾的最后一个 Softmax 层用于为各种用例（如神经机器翻译）提供输出，其中引入了原始的 Transformer 体系结构。

这种体系结构有两个输入，即输入和输出（右移）。其中一个在训练和推理中始终存在（输入），而另一个仅在模型产生的训练和推理中存在。在推理中不使用模型预测的原因是防止模型本身出错。但这意味着什么？想象一下，一个神经翻译模型试图在每一步将一个句子从英语翻译成法语，它对一个单词进行预测，并使用预测的单词预测下一个单词。但是，如果某一步出了问题，下面所有的预测也都将是错误的。为了防止模型出现这样的错误，本节提供了正确的单词作为一个右移版本。

图 1－19 所示为 Transformer 模型的可视化示例。该图显示了一个具有两个编码器和两个解码器层的 Transformer 模型。图 1－19 中的添加 & 规范化（Add&Normalize）层添加并规范化从前馈层获取的输入。

Transformer 体系结构的另一个主要改进是基于一个简单的通用文本压缩方案，以防止在输入端出现不可见的标记。这种改进使用不同的方法（例如，字节对编码以及句子片段编码）来实现，提高了 Transformer 处理不可见标记的性能。当模型遇到形态上近

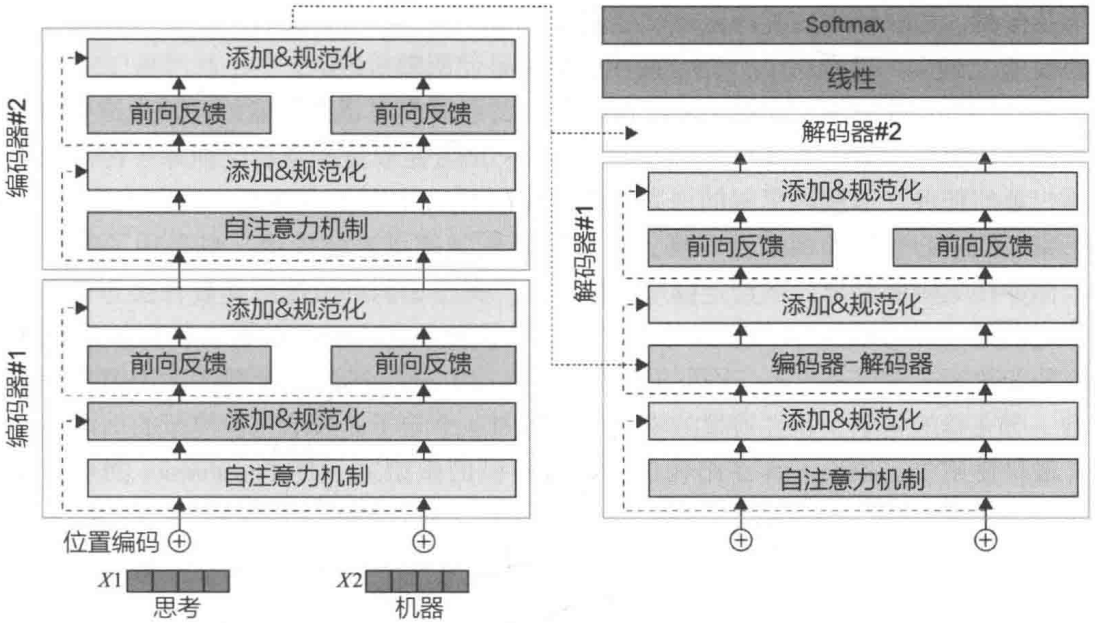


图 1－19　Transformer 模型的可视化示例

似的标记时，它也可以指导模型。这样的标记在过去是不可见的，在训练中很少被使用，但是在推理中有可能是可见的。在某些情况下，训练中会看到大量的信息；后者发生在形态丰富的语言中，如土耳其语、德语、捷克语和拉脱维亚语。例如，模型可能会看到单词 training，但不会看到 trainings。在这种情况下，模型可以将 trainings 标记为 training + s。当用户将这两个形态视为两个部分时，这两个部分是常见的形态。

基于 Transformer 的模型具有特别常见的特征。例如，它们都基于这个原始体系结构，区别在于模型所使用的步骤和不使用的步骤。在某些情况下，这些模型会产生微小的差异，例如对多头注意力机制的改进。

1.6　在迁移学习中结合使用 Transformer

迁移学习（Transfer Learning，TL）是人工智能（Artificial Intelligence，AI）和机器学习的一个领域，旨在使模型可以用于不同的任务。例如，在给定任务（如 A）上训练的模型可以用于不同的任务（如 B）。在自然语言处理领域，这可以使用类似 Transformer 的体系结构来实现，这种体系结构可以通过语言建模捕获对语言本身的理解。这些模型称为语言模型，它们为经过训练的语言提供了一个模型。迁移学习并不是一种新技术，它已被应用于计算机视觉等各个领域。ResNet、Inception、VGG 和 EfficientNet 都是此类模型的示例，这些模型都可以用作预先训练的模型，能够对不同的计算机视觉任务进行微调。

在自然语言处理中，也可以使用 Word2vec、GloVe 和 Doc2vec 等模型进行浅层迁移学习。之所以称为浅层，是因为这种迁移学习背后没有模型，而是使用了单词/标记的

预训练向量。可以使用这些标记或文档嵌入模型，然后使用分类器，或者将这些模型与其他模型（如循环神经网络）结合使用，而不是使用随机嵌入。

在自然语言处理中，使用 Transformer 模型的迁移学习也是可能的，因为这些模型可以在没有任何标记数据的情况下学习语言本身。语言建模是一项用于训练各种问题的可转移权重的任务。带掩码机制的语言建模是学习语言本身的方法之一。与 Word2vec 的基于窗口的中心标记预测模型一样，在带掩码机制的语言建模中，也采用了类似的方法，但存在关键的差异。在给定概率的情况下，每个单词都被掩蔽并被替换为一个特殊的标记，如［MASK］。语言模型（在本例中是基于 Transformer 的语言模型）必须预测被掩蔽的单词。与 Word2vec 不同的是，基于 Transformer 的语言模型不使用窗口，而是给出一个完整的句子，并且模型的输出必须是同一个句子，并填充被掩蔽的单词。

最早使用 Transformer 体系结构进行语言建模的模型之一是 Transformer 的双向编码表示（Bidirectional Encoder Representations form Transformer BART），它的实现基于 Transformer 体系结构的编码器部分。带掩码的语言建模由 BERT 使用训练语言模型之前和之后描述的相同方法完成。BERT 是一种迁移语言模型，用于不同的自然语言处理任务，如标记分类、序列分类，甚至问答系统。

一旦一个语言模型被训练出来，这些任务中的每一个都是 BERT 的一个微调任务。BERT 因其在基本 Transformer 编码器模型上的关键特征而闻名。通过改变这些特征，人们提出了小、极小、基本、大和超大的不同版本。上下文嵌入使模型能够根据给定的上下文确定每个单词的正确含义。例如，单词 Cold 可以在两个不同的句子中具有不同的含义：Cold - hearted killer（冷血杀手）和 Cold weather（寒冷天气）。编码器部分的层数、输入维度、输出嵌入维度和多头注意力机制的数量都是所谓的关键特征，如图 1 - 20 所示。

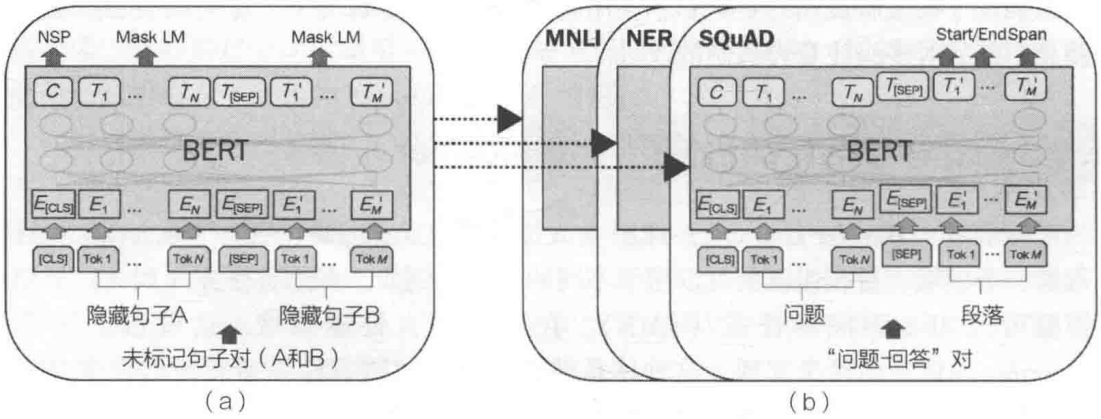


图 1 - 20 BERT 的预训练和微调过程

（a）预训练；（b）微调

在图 1 - 20 中，预训练（Pre - training）阶段还包括另一个目标，称为 next - sentence prediction（下一个句子预测）。正如我们所知道的，每个文档都是由相邻的句子组成的。训练一个模型捕获语言的另一个重要部分是理解句子之间的关系（无论这些

句子是否相关)。为了完成这些任务，BERT 引入了特殊的标记，如［CLS］和［SEP］。［CLS］标记最初是一个无意义的标记，用作所有任务的开始标记，并且包含有关句子的所有信息。在序列分类任务（如 NSP）中，使用位于该标记输出顶部的分类器（输出位置为 0)。该分类器也有助于评估句子的意义或者获取句子的语义。例如，当使用 Siamese BERT 模型时，通过余弦相似性等度量来比较不同句子的两个［CLS］标记非常有用。另外，［SEP］标记用于区分两个句子，它仅用于分隔两个句子。在预训练后，如果期望对序列分类任务（如情感分析）的 BERT 进行微调（这是一项序列分类任务)，那么将在［CLS］标记的输出嵌入上使用分类器。值得注意的是，所有迁移学习模型都可以在微调期间冻结或释放。冻结意味着将模型中的所有权重和偏差视为常量，并停止对其进行训练。在情感分析的例子中，如果模型冻结，那么只训练分类器，而不是训练模型。

1.7　本章小结

通过本章学习，读者了解到自然语言处理的各种方法及其演变历程：从 BoW 到 Transformer；研究了如何实现基于词袋的方法、基于循环神经网络的方法、基于卷积神经网络的方法，并了解了什么是 Word2vec，以及它如何帮助改善传统的基于深度学习的方法使用浅层的迁移学习；还以 BERT 作为示例，探讨了 Transformer 体系结构的基础。到本章结束时，读者了解了迁移学习，以及 BERT 是如何利用迁移学习的实现机制的。

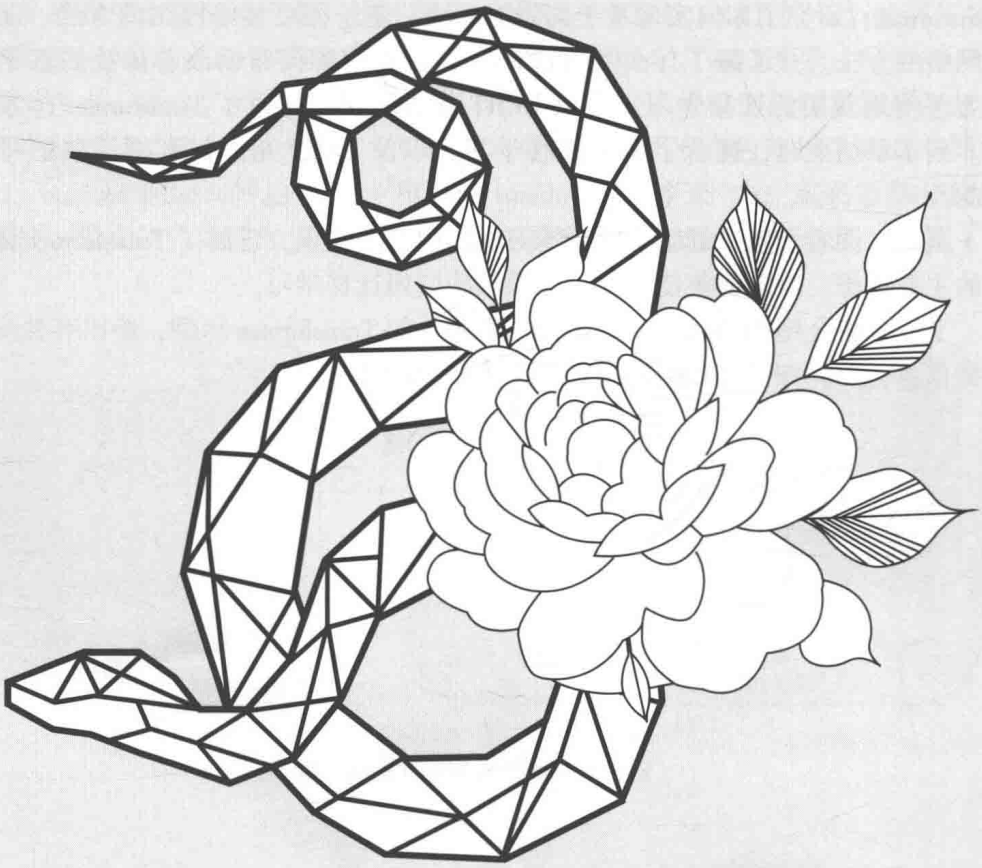
至此，读者了解了继续下一章学习所需的基本知识，理解了 Transformer 体系结构背后的主要思想，以及如何使用这种体系结构应用迁移学习。

下一章将介绍如何从头开始运行一个简单的 Transformer 示例，给出有关安装步骤的相关信息，并详细研究如何使用数据集和基准测试。

2

第2章

Transformer的实践操作入门



到目前为止，读者已经全面了解了基于深度学习方法的自然语言处理的发展历程。了解了一些有关 Transformer 及其相关体系结构的基本信息。在本章中，将深入探讨如何使用 Transformer 模型。本章将通过操作实例详细阐述分词器（Tokenizer）及其模型（例如 BERT）的技术细节，包括如何加载分词器/模型，以及如何使用社区提供的预训练模型。但在使用任何特定模型之前，将先讨论使用 Anaconda 提供必要环境所需的安装步骤。在安装步骤中，将介绍在各种操作系统（如 Linux、Windows 和 macOS）中安装库和程序的方法。本章还介绍了 PyTorch 和 TensorFlow 的两种版本［中央处理器（Central Processing Unit，CPU）和图形处理单元（Graphics Processing Unit，GPU）］的安装方法。本章还提供了在 Google Colaboratory（Google Colab）上快速安装 Transformer 库的方法。本章还包含一节，专门介绍在 PyTorch 和 TensorFlow 框架中使用模型的方法。

本章的另一个重要部分是 huggingface 模型库，其中讨论了寻找不同的模型和使用各种管道的步骤。例如，详细介绍了双向和自回归 Transformer（Bidirectional and Auto - Regressive Transformer，BART）、BERT 和表解析（TAble PArSing，TAPA）等模型，简要讨论了生成式预训练 Transformer 2（Generative Pre - trained Transformer 2，GPT - 2）的文本生成。然而，这仅是一个概述，本章的这一部分涉及安装环境和使用预先训练的模型。本章没有讨论模型训练，模型训练是后续章节的讨论重点。

一切准备就绪，并且读者已经了解了如何通过社区提供的模型使用 Transformer 库进行推理之后，将阐述 datasets 库。本章将介绍如何加载各种数据集、基准测试，以及如何使用各种度量；主要讨论如何加载一个特定的数据集并从中获取数据；讨论跨语言数据集以及如何使用 datasets 库加载本地文件；研究 map( ) 函数和 filter( ) 函数，这两个函数是 datasets 库在模型训练方面的重要函数。

本章是本书的重要部分，因为本章详细阐述了 datasets 库。了解如何使用社区提供的模型并为本书的后续章节准备好系统环境，这也是至关重要的。

本章将介绍以下主题。

（1）使用 Anaconda 安装 Transformer。

（2）使用语言模型和分词器。

（3）使用社区提供的模型。

（4）使用基准测试和数据集。

（5）速度和内存的基准测试。

2.1 技术需求

本章需要安装以下库和软件列表。尽管安装最新版本具有一定的优势，但必须安装彼此兼容的版本。

（1）Anaconda；

（2）Transformer 4.0.0；

（3）PyTorch 1.1.0；

（4）TensorFlow 2.4.0；

（5）Datasets 1.4.1。

可以通过以下 GitHub 链接获得本章中所有的源代码：

https://github.com/PacktPublishing/Mastering-Transformer/tree/main/CH02。

2.2 使用 Anaconda 安装 Transformer

Anaconda 是 Python 和 R 程序设计语言的一个发行版，简化了包的发行和部署，适用于科学计算。本章将介绍 Transformer 库的安装。但是，也可以在不借助 Anaconda 的情况下安装 Transformer 库。使用 Anaconda 的主要动机是更容易解释过程，并协调所使用的包。

如果要开始安装相关库，必须先安装 Anaconda。Anaconda 文档提供的官方指南提供了为常见操作系统（macOS、Windows 和 Linux）安装该软件的简单步骤。

2.2.1 在 Linux 操作系统中安装 Anaconda

目前有许多不同的 Linux 发行版可以供用户选择使用，但 Ubuntu 是其中的首选之一。本小节介绍在 Linux 操作系统中安装 Anaconda 的步骤。具体步骤如下。

（1）从 Anaconda 的官网下载 Linux 版本的 Anaconda 安装程序。在浏览器中打开网址 https://www.anaconda.com/products/individual#Downloads，然后跳转到 Linux 部分，如图 2-1 所示。

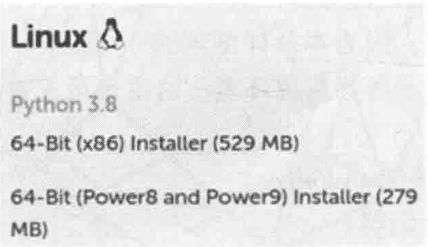


图 2-1 Linux 版本的 Anaconda 下载链接

（2）运行 bash 命令进行安装，并完成以下安装步骤。

（3）打开终端并运行以下命令。

```
bash Terminal./FilePath/For/Anaconda.sh
```

（4）按 Enter 键阅读许可协议，如果不想全部阅读，请按 Q 键。

（5）单击“Yes”按钮表示同意。

（6）单击“Yes”按钮使安装程序始终初始化 conda 根环境。

（7）从终端运行 Python 命令后，结果将会在 Python 版本信息后显示 Anaconda 提示符。

（8）可以从终端运行 anaconda - navigator 命令，以访问 Anaconda Navigator（Anaconda 导航器）。结果将显示 Anaconda 图形用户界面（Graphical User Interface，GUI），并开始加载相关模块，如图 2 - 2 所示。

图 2 - 2　Anaconda 导航器

2.2.2　在 Windows 操作系统中安装 Anaconda

在 Windows 操作系统中安装 Anaconda 的步骤如下。

（1）从 Anaconda 的官网下载 Windows 版本的 Anaconda 安装程序。在浏览器中打开网址 https://www.anaconda.com/products/individual#Downloads，然后跳转到 Windows 部分，如图 2 - 3 所示。

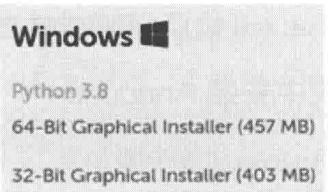


图 2 - 3　Windows 版本的 Anaconda 下载链接

（2）运行下载的安装程序，按照指南，单击“I Agree”按钮。

（3）选择安装位置，如图 2－4 所示。

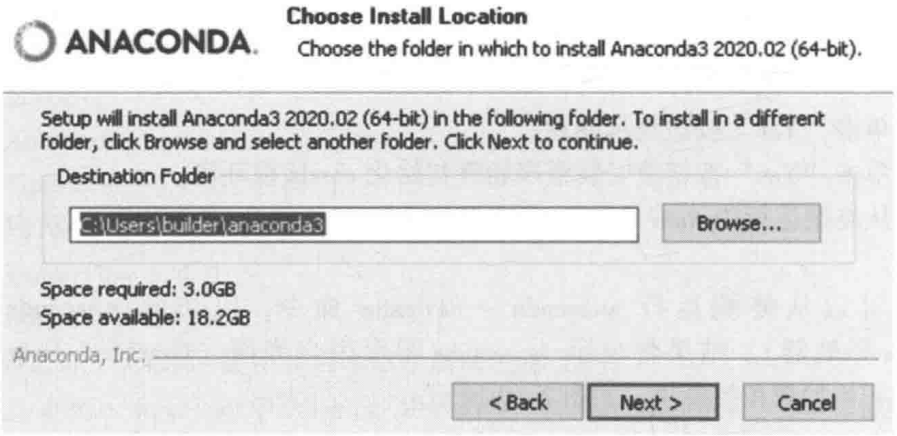


图 2－4　Windows 版本的 Anaconda 安装程序

（4）确认勾选“Add Anaconda3 to my PATH environment variable（添加 Anaconda3 到 PATH 环境变量）”复选框，如图 2－5 所示。如果没有勾选这个复选框，那么系统不会将 Anaconda 版本的 Python 添加到 Windows 环境变量中，从而无法在 Windows shell 或 Windows 命令行中直接运行 Python 命令。

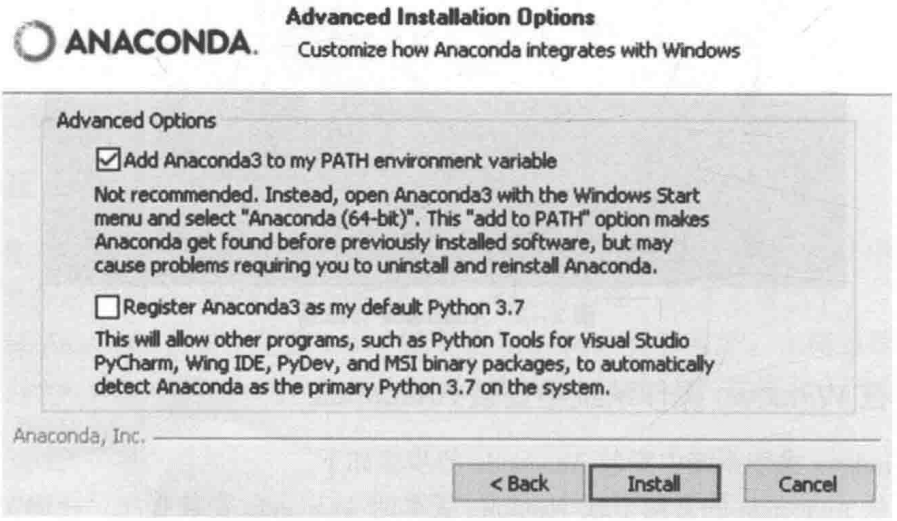


图 2－5　Anaconda 安装程序的高级选项

（5）按照其余的安装说明指示步骤完成安装。

至此，用户应该能够从“start（开始）”菜单中启动 Anaconda Navigator。

2.2.3　在 macOS 操作系统中安装 Anaconda

在 macOS 操作系统中安装 Anaconda 的步骤如下。

（1）从 Anaconda 的官网下载 macOS 版本的 Anaconda 安装程序。在浏览器中打开网

址 https://www.anaconda.com/products/individual#Downloads，然后跳转到 macOS 部分，如图 2－6 所示。

图 2－6　macOS 版本的 Anaconda 下载链接

（2）运行下载的安装程序。

（3）按照安装说明，单击“Install”按钮，在预定义位置安装 macOS 版本的 Anaconda，如图 2－7 所示。用户可以更改默认安装路径，但一般不建议修改。

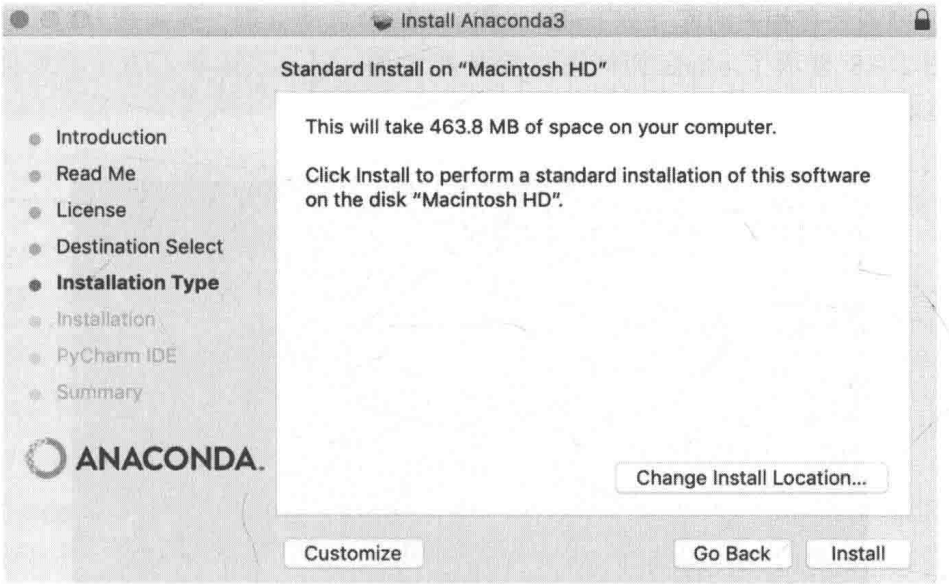


图 2－7　macOS 的 Anaconda 安装程序

完成安装后，用户应该能够访问 Anaconda Navigator。

2.2.4　安装 TensorFlow、PyTorch 和 Transformer

TensorFlow 和 PyTorch 是深度学习的两个主要库，可以通过 pip 或 conda 本身进行安装。conda 提供了一个命令行界面（Command－Line Interface，CLI），用于轻松地安装这些库。

为了安装一个干净的环境，并避免与其他环境冲突，最好为 huggingface 库创建一个 conda 环境。用户可以运行以下代码来创建一个虚拟环境。

```
conda create -n Transformer
```

上述命令将为安装其他库创建一个空的虚拟环境。创建虚拟环境之后，需要激活虚

拟环境。激活方法如下所示。

```
conda activate Transformer
```

运行以下命令，可以轻松安装 Transformer 库。

```
conda install -c conda-forge tensorflow
conda install -c conda-forge pytorch
conda install -c conda-forge Transformer
```

conda install 命令中的 -c 参数允许 Anaconda 使用其他通道搜索库。

请注意，必须安装 TensorFlow 和 PyTorch，因为 Transformer 库同时使用这两个库。另外一个注意事项是 conda 可以轻松处理 TensorFlow 的 CPU 和 GPU 版本。只需将 -gpu 放在 tensorflow 之后，conda 将自动安装 GPU 版本。若通过 cuda 库（GPU 版本）安装 PyTorch，则需要有相关的库（如 cuda），但 conda 会自动处理，无须额外的手动设置或安装。图 2-8 显示了 conda 如何通过安装相关的 cudatoolkit 和 cudnn 库来自动安装 PyTorch GPU 版本。

```
1: meysam@meysam-G3-3590: ~
(base) meysam@meysam-G3-3590:~$ conda install pytorch
Collecting package metadata (current_repodata.json): done
Solving environment: /
The environment is inconsistent, please check the package plan carefully
The following packages are causing the inconsistency:
  - defaults/linux-64::anaconda-navigator==1.9.12=py38_0
  - defaults/linux-64::ipykernel==5.3.4=py38h5ca1d4c_0
  - defaults/noarch::conda-verify==3.4.2=py_1
  - defaults/linux-64::notebook==6.1.4=py38_0
  - defaults/linux-64::nb_conda_kernels==2.2.3=py38_0
  - defaults/noarch::nbclient==0.5.1=py_0
  - conda-forge/linux-64::conda==4.9.2=py38h578d9bd_0
  - defaults/noarch::jupyter_client==6.1.7=py_0
  - defaults/linux-64::terminado==0.9.1=py38_0
  - defaults/linux-64::nbconvert==6.0.7=py38_0
  - defaults/linux-64::conda-build==3.18.11=py38_0
  - conda-forge/linux-64::widgetsnbextension==3.5.1=py38h32f6830_1
  - conda-forge/noarch::ipywidgets==7.5.1=pyh9f0ad1d_1
  - defaults/linux-64::_ipyw_jlab_nb_ext_conf==0.1.0=py38_0
  - defaults/linux-64::conda-package-handling==1.7.2=py38h03888b9_0
done

## Package Plan ##

  environment location: /home/meysam/anaconda3

  added / updated specs:
    - pytorch

The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    _openmp_mutex-4.5          |            1_gnu          22 KB
    _pytorch_select-0.1        |            cpu_0           3 KB
    anyio-2.2.0                |   py38h06a4308_1         125 KB
    babel-2.9.1                |     pyhd3eb1b0_0         5.5 MB
    ca-certificates-2021.7.5   |       h06a4308_1         113 KB
    certifi-2021.5.30          |   py38h06a4308_0         138 KB
```

图 2-8 在 conda 中安装 PyTorch 和相关 cuda 库

请注意，所有这些安装操作也可以在没有 conda 的情况下完成，但使用 Anaconda 的原因在于其易用性。在使用虚拟环境或者安装 GPU 版本的 TensorFlow 或 PyTorch 方面，Anaconda 的工作方式就像魔术一样，可以节省大量时间。

2.2.5　使用 Google Colab 安装环境

即使使用 Anaconda 可以节省时间并且非常有用，但在大多数情况下，并不是每个人都有良好的、合理的计算资源。在这种情况下，Google Colab 是一个很好的选择。可以使用以下命令在 Colab 中安装 Transformer 库。

```
!pip install Transformer
```

命令语句前面的感叹号（!）使代码可以在 Colab shell 中运行，这相当于在终端中运行代码，而不是使用 Python 解释器运行代码。该命令将自动安装 Transformer 库。

2.3　使用语言模型和分词器

本节介绍如何将 Transformer 库与语言模型及其相关的分词器（tokenizers）一起使用。为了使用指定的语言模型，首先需要导入对应的库，从 Google 提供的 BERT 模型开始，并使用其预训练版本。代码片段如下所示。

```
>>> from Transformer import BERTTokenizer
>>> tokenizer = \
BERTTokenizer.from_pretrained('BERT-base-uncased')
```

在上述代码片段中，第 1 行导入了 BERT 分词器，第 2 行下载了 BERT 基本版本的预训练分词器。请注意，不区分大、小写的版本使用不区分大、小写的字母进行训练，因此文本采用大写还是小写是无关紧要的。为了测试并查看输出，可以运行以下代码片段。

```
>>> text = "Using Transformer is easy!"
>>> tokenizer(text)
```

其输出结果如下所示。

```
{'input_ids': [101, 2478, 19081, 2003, 3733, 999, 102], 'token_type_ids': [0, 0,
0, 0, 0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1]}
```

input_ids 显示每个标记的标记 ID，token_type_ids 显示分隔第 1 个序列和第 2 个序列的标记类型，如图 2-9 所示。

```
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
| first sequence    | second sequence |
```

图 2-9　BERT 的序列分隔

注意，attention_mask 是一个由 0 和 1 组成的掩码，用于显示 Transformer 模型中序列的开始和结束，以避免不必要的计算。每个分词器都按自己的方式向原始序列添加特殊标记。对于 BERT 分词器，它将［CLS］标记添加到序列的开头，将［SEP］标记添加到序列的结尾，结果显示为 101 和 102。这些数字来自预训练分词器的标记 ID。

分词器可以分别用于基于 PyTorch 的 Transformer 模型以及基于 TensorFlow 的 Transformer 模型。为了输出对应模型所要求的输出，必须在命令参数 return_tensors（返回张量）中指定 pt 或 tf 关键字。例如，运行以下命令就可以使用分词器。

```
>>> encoded_input = tokenizer(text, return_tensors = "pt")
```

变量 encoded_input 中包含 PyTorch 模型所需要的分词文本。例如，为了运行模型（如 BERT 基础模型），可以使用以下代码，从 huggingface 库中下载模型。

```
>>> from Transformer import BERTModel
>>> model = BERTModel.from_pretrained("BERT - base - uncased")
```

通过以下代码行，可以将分词器的输出传递给下载的模型。

```
>>> output = model( * * encoded_input)
```

模型的输出结果为嵌入和交叉注意力输出的形式。

在加载和导入模型时，可以指定需要使用的模型版本。如果模型名称包含前缀 TF，Transformer 库将加载对应的 TensorFlow 版本。以下代码显示了如何加载和使用 BERT 基础模型的 TensorFlow 版本。

```
from Transformer import BERTTokenizer, TFBERTModel
tokenizer = \
BERTTokenizer.from_pretrained('BERT - base - uncased')
model = TFBERTModel.from_pretrained("BERT - base - uncased")
text = " Using Transformer is easy!"
encoded_input = tokenizer(text, return_tensors ='tf')
output = model( ** encoded_input)
```

对于特定的任务，如使用语言模型填充掩码，可以直接使用 huggingface 库设计的管道。例如，在以下代码段中，包含了填充掩码的任务。

```
>>> from Transformer import pipeline
>>> unmasker = \
pipeline('fill - mask', model ='BERT - base - uncased')
>>> unmasker("The man worked as a [MASK].")
```

上述代码片段将生成以下输出，显示得分以及可能放置在［MASK］标记中的标记。

```
[{'score': 0.09747539460659027, 'sequence': 'the man worked
as a carpenter.', 'token': 10533, 'token_str': 'carpenter'},
```

```
{'score': 0.052383217960596085, 'sequence': 'the man worked
as a waiter.', 'token': 15610, 'token_str': 'waiter'},
{'score': 0.049627091735601425, 'sequence': 'the man worked
as a barber.', 'token': 13362, 'token_str': 'barber'},
{'score': 0.03788605332374573, 'sequence': 'the man worked
as a mechanic.', 'token': 15893, 'token_str': 'mechanic'},
{'score': 0.03768084570765495, 'sequence': 'the man worked as
a salesman.', 'token': 18968, 'token_str': 'salesman'}]
```

为了使用 pandas 来显示整洁的视图，请运行以下代码。

```
>>> pd.DataFrame(unmasker("The man worked as a [MASK]."))
```

输出结果如图 2 - 10 所示。

	score	sequence	token	token_str
0	0.097475	the man worked as a carpenter.	10533	carpenter
1	0.052383	the man worked as a waiter.	15610	waiter
2	0.049627	the man worked as a barber.	13362	barber
3	0.037886	the man worked as a mechanic.	15893	mechanic
4	0.037681	the man worked as a salesman.	18968	salesman

图 2 - 10　BERT 掩码填充的输出结果

到目前为止，读者学习了如何加载和使用预训练的 BERT 模型，并且已经了解了分词器的基本知识，以及模型的 PyTorch 版本和 TensorFlow 版本的差异。在下一节中，将学习如何使用社区提供的模型加载不同的模型，阅读模型作者提供的相关信息，以及使用不同的管道，如文本生成管道、问题回答系统管道。

2.4　使用社区提供的模型

Hugging Face 拥有由大型人工智能和信息技术（Information Technology，IT）公司（如 Google 和 Facebook）的合作者提供的大量社区模型，还包括个人和大学研究人员提供的许多有趣的模型。访问和使用这些模型非常容易。首先，访问 Hugging Face 网站上的 Transformer models 目录（https://huggingface.co/models），如图 2 - 11 所示。

除了这些模型之外，自然语言处理任务还可以使用许多优秀的、实用的数据集。为了使用其中一些模型，可以通过关键字来搜索这些模型，或者通过指定主要的自然语言处理任务和管道来搜索这些模型。

例如，假设用户正在寻找一个表问答系统模型。在找到一个感兴趣的模型后，Hugging Face 网站将显示一个类似图 2 - 12 的页面（https://huggingface.co/google/tapas - base - finetuned - wtq）。

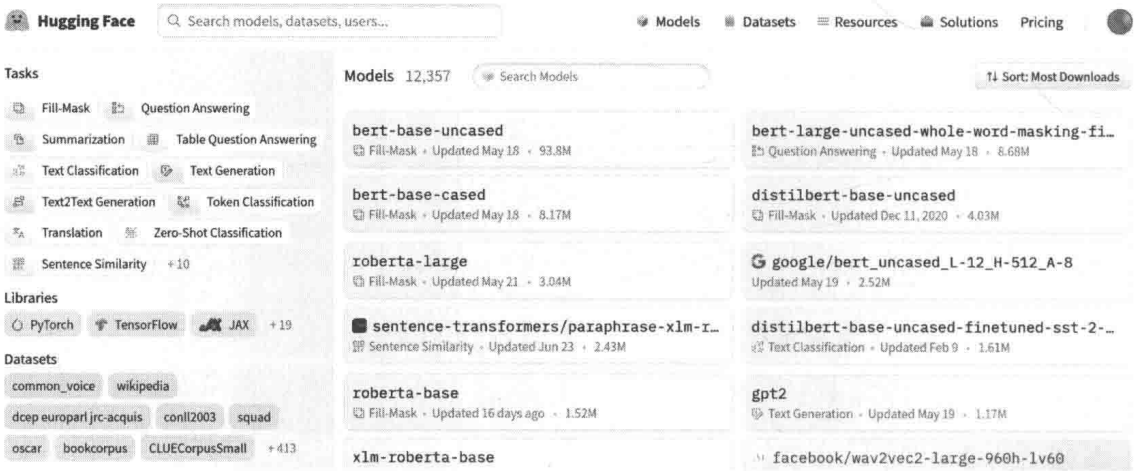


图 2－11　Hugging Face 网站中的模型存储库

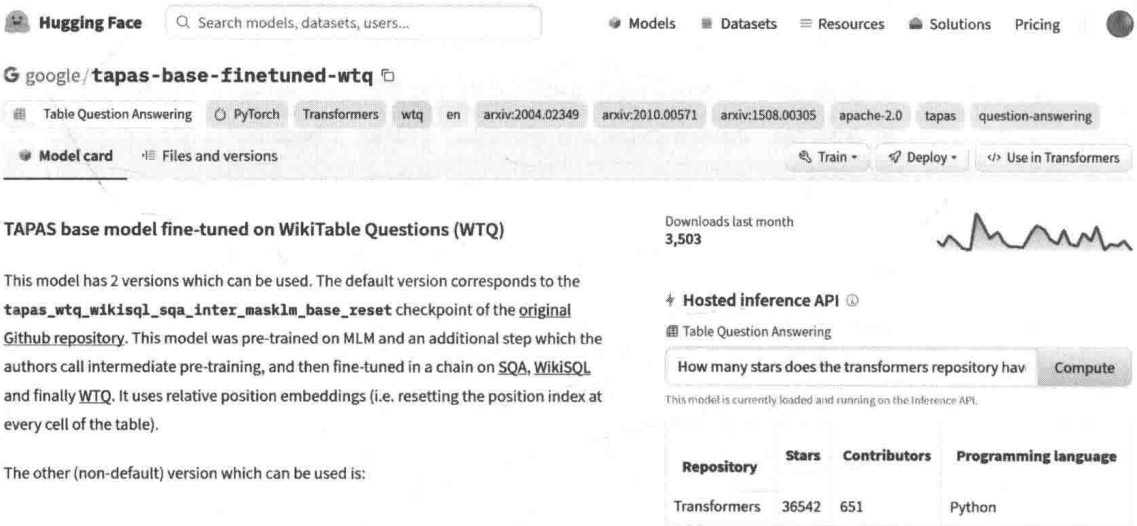


图 2－12　对应 TAPAS 模型的网页

在图 2－12 所示网页的右侧有一个面板，可以在其中测试此模型。请注意，这是一个表问答系统的模型，可以回答提供给模型的有关表的问题。如果用户问一个问题，模型会通过突出显示答案来回答。图 2－13 显示了用户获取输入并提供特定表的答案的过程。

每个模型都包含一个由模型作者提供的页面，也称为模型卡。可以通过模型页面中提供的示例使用模型。例如，可以访问 Hugging Face GPT－2 网页并查看作者提供的示例（https://huggingface. co/gpt2），如图 2－14 所示。

建议使用管道，因为 Transformer 库负责完成所有的复杂操作。作为另一个示例，假设需要一个现成的零样本（zero－shot）分类器。以下代码片段表明，实现和使用这种预训练模型非常简单。

Table Question Answering

How many stars does the transformers repository have? Compute

1 match: AVERAGE > 36542

Repository	Stars	Contributors	Programming language
Transformers	36542	651	Python
Datasets	4512	77	Python
Tokenizers	3934	34	Rust, Python and NodeJS

= Add row　|| Add col　Reset table

Computation time on cpu: cached.

图 2－13　使用 TAPAS 的表问答系统

How to use

You can use this model directly with a pipeline for text generation. Since the generation relies on some randomness, we set a seed for reproducibility:

```
>>> from transformers import pipeline, set_seed
>>> generator = pipeline('text-generation', model='gpt2')
>>> set_seed(42)
>>> generator("Hello, I'm a language model,", max_length=30, num_return_sequence

[{'generated_text': "Hello, I'm a language model, a language for thinking, a lan
 {'generated_text': "Hello, I'm a language model, a compiler, a compiler library
 {'generated_text': "Hello, I'm a language model, and also have more than a few
 {'generated_text': "Hello, I'm a language model, a system model. I want to know
 {'generated_text': 'Hello, I\'m a language model, not a language model"\n\nThe
```

图 2－14　Hugging Face GPT－2 网页中的文本生成代码示例

```
>>> from Transformers import pipeline
>>> classifier = pipeline("zero-shot-classification",
model="facebook/bart-large-mnli")
>>> sequence_to_classify = "I am going to france."
>>> candidate_labels = ['travel','cooking','dancing']
>>> classifier(sequence_to_classify, candidate_labels)
```

上述代码片段将产生以下输出结果。

```
{'labels': ['travel','dancing','cooking'],
'scores': [0.9866883158683777, 0.007197578903287649,
0.006114077754318714],'sequence': 'I am going to france.'}
```

到目前为止，本节完成了模型的安装以及 Hello World 应用程序的测试。本节详细介绍了模型的安装过程，完成了环境设置，并体验了第一个 Transformer 管道。在接下来的章节中，将介绍 datasets 库，它将是后续章节中实验内容不可或缺的实用工具。

2.5 使用基准测试和数据集

在介绍 datasets 库之前，有必要讨论一些重要的基准测试。例如，通用语言理解评估（General Language Understanding Evaluation[①]，GLUE）、多语言编码器的跨语言迁移评估（Cross - lingual Transfer Evaluation of Multilingual Encoders，XTREME）和斯坦福问答数据集（Stanford Question Answering Dataset，SQuAD）。基准测试对于在多任务和多语言环境中的迁移学习尤其重要。在自然语言处理中，用户主要关注特定的度量指标，即特定任务或数据集的性能得分。借助 Transformer 库，可以将从特定任务中学到的知识迁移到相关任务上，这称为迁移学习。通过在相关问题之间迁移表示，用户能够训练跨任务并且共享共同语言知识的通用模型，也称之为多任务学习（Multi - Task Learning，MTL）。迁移学习的另一个方面是跨自然语言（多语言模型）的迁移知识。

2.5.1 重要的基准测试

本小节介绍基于 Transformer 体系结构并且被广泛使用的重要基准测试。这些基准测试对多任务学习以及多语言和零样本学习（包括许多具有挑战性的任务）做出了很大贡献。本小节研究以下基准测试。

（1）GLUE；

（2）SuperGLUE；

（3）XTREME；

（4）XGLUE；

（5）SQuAD。

为了节约篇幅，仅讨论 GLUE 基准测试的任务细节。下面讨论该基准测试。

1. GLUE 基准测试

最近的研究成果表明，多任务训练方法作为一种特殊模式，与单任务学习方法相比，可以取得更好的效果。在这个研究方向上，为多任务学习引入了 GLUE 基准测试，它是一组工具和数据集，用于评估任务列表中多任务学习模型的性能。GLUE 基准测试提供了一个公共排行榜，用于监控基准测试上的提交性能，以及汇总了 11 项任务的单个数字指标。GLUE 基准测试包括许多基于现有任务的句子理解任务，这些任务涵盖不同大小、文本类型和难度级别的各种数据集。这些任务分为三大类，每个大类及其具体示例概述如下。

1）单句子任务

（1）CoLA：Corpus of Linguistic Acceptability（语言可接受性语料库）数据集。这项任务包括从语言学理论文章中得出的英语可接受性判断。

（2）SST - 2：Stanford Sentiment Treebank（斯坦福情绪树库）数据集。这项任务包

---

① 原著此处拼写错误，应该是 Evaluation。——译者注

括电影评论中的句子和用 pos/neg 标签对其情感进行的人工标注。

2）相似性和释义任务

（1）MRPC：Microsoft Research Paraphrase Corpus（微软研究释义语料库）数据集。这项任务主要检测两个句子在语义上是否等价。

（2）QQP：Quora Question Pairs（Quora 问题对）数据集。这项任务确定一对疑问句在语义上是否等价。

（3）STS－B：Semantic Textual Similarity Benchmark（语义文本相似性基准测试）数据集。这项任务是从新闻标题中提取的句子对集合，相似性得分在 1 和 5 之间。

3）推理任务

（1）MNLI：Multi－Genre Natural Language Inference（多体裁自然语言推理）语料库。它是一组具有文本蕴含（textual entailment）的句子对。这项任务是预测文本是否包含假设（蕴含关系），是否与假设冲突（矛盾关系），或者两者都不是（中性关系）。

（2）QNLI：Question Natural Language Inference（自然语言问题推理）数据集。这是 SQuAD 的转换版本。这项任务是检查一个句子是否包含一个问题的答案。

（3）RTE：Recognizing Textual Entailment（文本蕴含识别）数据集。这是一项具有挑战性的文本蕴含任务，需要组合来自不同来源的数据。该数据集类似之前的自然语言问题推理 QNLI 数据集，其任务是检查第一个文本是否包含第二个文本。

（4）WNLI：Winograd Natural Language Inference（Winograd 自然语言推理）模式挑战。它最初是一个代词解析任务，用于连接一个句子中的代词和短语。GLUE 将该问题转化为句子对的分类问题。

2. SuperGLUE 基准测试

与 GLUE 相类似，SuperGLUE 也是一个新的基准测试，采用了一组新的更难理解的语言任务，并利用现有数据，提供了一个由大约 8 个语言任务组成的公共排行榜，与 GLUE 等单一数字性能指标关联。其背后的动机是，在写这本书时，目前最先进的 GLUE 得分（90.8）超过了人类的表现（87.1）。因此，SuperGLUE 为通用语言理解技术提供了更具挑战性和多样性的任务。

在 gluebenchmark.com 网站上，用户可以访问 GLUE 和 SuperGLUE 的基准测试。

3. XTREME 基准测试

近年来，自然语言处理研究者越来越关注学习通用表征，而不是学习一个可以应用于许多相关任务的单一任务。构建通用语言模型的另一种方法是使用多语言任务。据观察，最近的多语种模型，如多语种 BERT（mBERT）和 XLM－R，在大量多语种语料库的预训练下，在将其迁移到其他语言时表现更好。因此，本节所提出的基准测试的主要优势在于，跨语言泛化使用户能够通过零样本跨语言迁移，在资源匮乏的语言中构建成功的自然语言处理应用程序。

基于这样的研究目的，本节设计了 XTREME 基准测试。它目前包括大约 40 种不同的语言，属于 12 个语系，包括 9 种不同的任务，需要对不同层次的语法或语义进行推理。然而，将一个模型扩展到 7 000 多种世界语言仍然是一个挑战，语言覆盖率和模型

能力之间存在着权衡。

4. XGLUE 基准测试

XGLUE 是另一个跨语言基准测试，用于评估和改进自然语言理解（Natural Language Understanding，NLU）和自然语言生成（Natural Language Generation，NLG）的跨语言预训练模型的性能。它最初由 19 种语言的 11 项任务组成。与 XTREME 的第一个区别是，每个任务的训练数据只有英文版本。它迫使语言模型只从英语文本数据中学习，并将这些知识迁移到其他语言中，这就是所谓的零样本跨语言迁移能力。第二个区别是，它同时具有自然语言理解和自然语言生成任务。

5. SQuAD 基准测试

SQuAD 是自然语言处理领域中广泛使用的问答系统数据集。它提供了一组“问题-回答”对，以对自然语言处理模型的阅读理解能力进行基准测试。它由一系列问题、一段阅读文章和一个答案组成，所提供的答案由众包工人（crowdworker）在一组维基百科文章中进行人工标注。问题的答案通过阅读文章中的一段文字来获取。最初的版本 SQuAD1.1 在收集数据集时，没有提供“unanswerable（无法回答）”选项，因此每个问题都有一个答案，可以在所阅读文章的某个地方找到。自然语言处理模型被迫回答这个问题，即使这似乎不可能。SQuAD2.0 是一个改进的版本，自然语言处理模型在可能的情况下回答问题，在不可能回答的情况下避免回答问题。SQuAD2.0 包含 50 000 个无法回答的问题，这些问题由众包工人以对抗的方式编写，看起来类似可回答的问题。此外，SQuAD2.0 还从 SQuAD1.1 中提取了 100 000 个问题。

2.5.2 使用应用程序编程接口访问数据集

datasets 库提供了一个非常有效的实用工具，通过 Hugging Face 中心与社区加载、处理和共享数据集。与 TensorFlow 数据集一样，datasets 库可以根据请求直接从原始数据集主机下载、缓存和动态加载数据集。该库还提供了评估指标以及所需要的数据。事实上，该中心并不持有或分发数据集。相反，它保留有关数据集的所有信息，包括所有者、预处理脚本、描述和下载链接。首先需要检查是否具有使用相应许可证下的数据集的权限。为了了解其他功能，请查看数据集的具体信息。

首先，需要安装 datasets[①] 库，安装命令如下所示。

```
pip install datasets
```

以下代码片段使用 Hugging Face 中心来自动加载 CoLA 数据集。如果数据尚未缓存，则 datasets.load_dataset()函数将从实际路径下载加载脚本。

```
from datasets import load_dataset
cola = load_dataset('glue','cola')
```

① 原著此处有误，应该是 datasets。——译者注

```
cola['train'][25:28]
```

重要提示

数据集的可重用性：当用户多次重新运行代码时，datasets 开始缓存用户的加载和操作请求。它首先存储数据集并开始缓存数据集上的操作，如数据拆分、选择和排序。用户将看到一条警告消息。

在前面的示例中，从 GLUE 基准测试下载了 CoLA 数据集，并从该数据集的 train 拆分中选择了几个样例。

目前，有 661 个自然语言处理数据集和 21 个不同任务的度量指标，如以下代码片段所示。

```
from pprint import pprint
from datasets import list_datasets, list_metrics
all_d = list_datasets()
metrics = list_metrics()
print(f"{len(all_d)} datasets and {len(metrics)} metrics exist
in the hub\n")
pprint(all_d[:20], compact = True)
pprint(metrics, compact = True)
```

输出结果如下所示。

```
661 datasets and 21 metrics exist in the hub.
['acronym_identification','ade_corpus_v2','adversarial_qa',
'aeslc','afrikaans_ner_corpus','ag_news','ai2_arc','air_
dialogue','ajgt_twitter_ar','allegro_reviews','allocine',
'alt','amazon_polarity','amazon_reviews_multi','amazon_us_
reviews','ambig_qa','amttl','anli','app_reviews','aqua_rat']
['accuracy','BERTscore','bleu','bleurt','comet','coval',
'f1','gleu','glue','indic_glue','meteor','precision',
'recall','rouge','sacrebleu','sari','seqeval','squad',
'squad_v2','wer','xnli']
```

一个数据集可能有多个配置。例如，作为一个聚合基准测试，GLUE 包含许多子集，如前面提到的 CoLA、SST－2 和 MRPC。为了访问每个 GLUE 基准数据集，需要传递两个参数：第一个参数是 GLUE；第二个参数是可以选择的样例数据集的特定数据集（CoLA 或 SST－2）。同样，Wikipedia 数据集为几种语言提供了多种配置。

数据集附带一个 DatasetDict 对象，包括几个 Dataset 实例。当使用数据拆分选项（split ='...'）时，将得到数据集实例。例如，CoLA 数据集附带一个 DatasetDict 对象，其中包含三个数据拆分部分，即训练（train）、验证（validation）和测试（test）。虽然

训练数据集和验证数据集包括两个标签（1 表示可接受；0 表示不可接受），但测试拆分数据集的标签值为 -1，这表示没有标签。

接下来，查看 CoLA 数据集对象的结构。代码如下所示。

```
>>> cola = load_dataset('glue','cola')
>>> cola
DatasetDict({
train: Dataset({
features: ['sentence','label','idx'],
            num_rows: 8551 })
validation: Dataset({
features: ['sentence','label','idx'],
             num_rows: 1043 })
test: Dataset({
          features: ['sentence','label','idx'],
          num_rows: 1063 })
})
cola['train'][12]
{'idx': 12,'label':1,'sentence':'Bill rolled out of the room.'}
>>> cola['validation'][68]
{'idx': 68,'label': 0,'sentence':'Which report that John was
incompetent did he submit?'}
>>> cola['test'][20]
{'idx': 20,'label': -1,'sentence':'Has John seen Mary?'}
```

数据集对象包含一些可能对用户有用的附加元数据信息，如 split（拆分）、description（描述）、citation（引用）、homepage（主页）、license（许可证）和 info（信息）等。接着运行以下代码。

```
>>> print("1#",cola["train"].description)
>>> print("2#",cola["train"].citation)
>>> print("3#",cola["train"].homepage)
1# GLUE, the General Language Understanding Evaluation
benchmark(https://gluebenchmark.com/) is a collection of
resources for training, evaluating, and analyzing natural
language understanding systems.2# @article{warstadt2018neural,
title = {Neural Network Acceptability Judgments},
author = {Warstadt, Alex and Singh, Amanpreet and Bowman,
Samuel R}, journal = {arXiv preprint arXiv:1805.12471},
year = {2018}}@inproceedings{wang2019glue, title = {{GLUE}:
A Multi-Task Benchmark and Analysis Platform for Natural
Language Understanding}, author = {Wang, Alex and Singh,
```

```
Amanpreet and Michael, Julian and Hill, Felix and Levy, Omer
and Bowman, Samuel R.}, note={In the Proceedings of ICLR.},
year={2019}}3# https://nyu-mll.github.io/CoLA/
```

如前所述，GLUE 基准测试提供了许多数据集。使用以下代码可以下载 MRPC 数据集。

```
>>> mrpc = load_dataset('glue','mrpc')
```

同样，为了访问其他 GLUE 任务，需要更改第二个参数。代码如下所示。

```
>>> load_dataset('glue','XYZ')
```

为了应用数据可用性的健全性检查，可以运行以下代码。

```
>>> glue=['cola','sst2','mrpc','qqp','stsb','mnli',
        'mnli_mismatched','mnli_matched','qnli','rte',
        'wnli','ax']
>>> for g in glue:
         _=load_dataset('glue',g)
```

XTREME（同时结合使用一个跨语言数据集）是本节讨论过的另一种流行的跨语言数据集。可以从 XTREME 集合中选取 MLQA 示例。多语言问题回答（MLQA）数据集是 XTREME 基准测试的一个子集，该基准测试旨在评估跨语言问题回答模型的性能，包括大约 5 000 个 SQuAD 格式的摘录问题回答实例，跨越 7 种语言（文字），即英语、德语、阿拉伯语、印地语、越南语、西班牙语和简体中文。

例如，MLQA. en. de 是一个英语 - 德语问题回答示例数据集，可以按照以下方式加载。

```
>>> en_de = load_dataset('xtreme','MLQA.en.de')
>>> en_de \
DatasetDict({
test: Dataset({features: ['id','title','context','question',
'answers'], num_rows: 4517
}) validation: Dataset({ features: ['id','title','context',
'question','answers'], num_rows: 512})})
```

使用 pandas DataFrame（数据帧，又被译为数据框），可以更方便地查看其内容。代码如下所示。

```
>>> import pandas as pd
>>> pd.DataFrame(en_de['test'][0:4])
```

上述代码片段的输出结果如图 2 - 15 所示。

	answers	context	id	question	title
0	{'answer_start': [31], 'text': ['cell']}	An established or immortalized cell line has a...	037e8929e7e4d2f949ffbabd10f0f860499ff7c9	Woraus besteht die Linie?	Cell culture
1	{'answer_start': [232], 'text': ['1885']}	The 19th-century English physiologist Sydney R...	4b36724f3cbde7c287bde512ff09194cbba7f932	Wann hat Roux etwas von seiner Medullarplatte ...	Cell culture
2	{'answer_start': [131], 'text': ['TRIPS']}	After the Uruguay round, the GATT became the b...	13e58403df16d88b0e2c665953e89575704942d4	Was muss ratifiziert werden, wenn ein Land ger...	TRIPS Agreement

图 2－15　英语－德语跨语言问题回答数据集

1. 使用 datasets 库进行数据操作

数据集附带有许多子集字典，其中 split 参数用于确定需要加载的子集或子集的一部分。当 split 参数为默认值 none 时，它将返回所有子集（训练、测试、验证或其他组合）的数据集字典。如果指定了 split 参数，则将返回单个数据集而不是字典。在以下示例中，仅检索 cola 数据集的训练数据拆分。

```
>>> cola_train = load_dataset('glue','cola', split ='train')
```

还可以得到训练子集和验证子集的混合数据。代码如下所示。

```
>>> cola_sel = load_dataset('glue','cola', split =
'train[:300]+validation[-30:]')
```

split 表达式表示获取前 300 个训练样例和最后 30 个验证样例，结果作为 cola_sel 返回。

可以应用各种拆分组合来得到不同训练子集和验证子集组成的混合数据，拆分示例如下所示。

（1）训练子集的前 100 个样例和验证子集的前 100 个样例。代码如下所示。

```
split ='train[:100]+validation[:100]'
```

（2）训练子集的前 50% 样例和验证子集的后 30% 样例。代码如下所示。

```
split ='train[:50%]+validation[-30%:]'
```

（3）训练子集的前 20% 样例和验证子集切片［30:50］后的样例。代码如下所示。

```
split ='train[:20%]+validation[30:50]'
```

2. 排序、索引和混排

以下代码片段执行调用 cola_sel 对象的 sort( ) 函数。可以看到前 15 个标签和后 15 个标签。

```
>>> cola_sel.sort('label')['label'][:15]
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

```
>>> cola_sel.sort('label')['label'][ -15:]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

到目前为止，用户已经熟练掌握了 Python 切片表示法。同样，也可以使用类似的切片表示法或索引列表访问多行数据。代码如下所示。

```
>>> cola_sel[6,19,44]
{'idx': [6, 19, 44],
'label': [1, 1, 1],
 'sentence':['Fred watered the plants flat.',
   'The professor talked us into a stupor.',
   'The trolley rumbled through the tunnel.']}
```

可以对数据集进行如下混排。

```
>>> cola_sel.shuffle(seed=42)[2:5]
{'idx': [159, 1022, 46],
'label': [1, 0, 1],
'sentence': ['Mary gets depressed if she listens to the
Grateful Dead.',
'It was believed to be illegal by them to do that.',
'The bullets whistled past the house.']}
```

重要提示

种子值：混排时，需要传递一个种子值来控制随机性，从而保证读者运行的结果与本书的运行结果一致。

3. 缓存和可重用性

通过使用缓存文件，可以使用快速后端通过内存映射（如果数据集在硬盘上）加载大型数据集。这种智能缓存有助于保存和重用在硬盘上执行的操作结果。为了查看有关数据集的缓存日志，请运行以下代码。

```
>>> cola_sel.cache_files
[{'filename':'/home/savas/.cache/huggingface...','skip':
0,'take': 300}, {'filename':'/home/savas/.cache/
huggingface...','skip': 1013,'take': 30}]
```

4. 数据集的 filter( ) 函数和 map( ) 函数

有时候人们可能希望使用数据集的特定选择。例如，仅检索 CoLA 数据集中包括单词 kick 在内的句子，如以下代码片段所示。datasets. Dataset. filter( ) 函数的作用是返回包括单词 kick 在内的句子，其中应用了匿名函数和 lambda 关键字。

```
>>> cola_sel = load_dataset('glue','cola',
split='train[:100%]+validation[ -30%:]')
```

```
>>> cola_sel.filter(lambda s: "kick" in s['sentence'])
["sentence"][:3]
['Jill kicked the ball from home plate to third base.','Fred
kicked the ball under the porch.','Fred kicked the ball behind
the tree.']
```

以下过滤操作用于从集合中获取正样例（也就是可接受的样例）。

```
>>> cola_sel.filter(lambda s: s['label'] = = 1 )["sentence"][:3]
["Our friends won't buy this analysis, let alone the next one
we propose.",
"One more pseudo generalization and I'm giving up.",
"One more pseudo generalization or I'm giving up."]
```

在某些情况下，可能不知道类标签所对应的整数编码。假设有很多类，而且无法判断在这 10 个类别中 culture 类所对应的编码。可以将 acceptable 标签传递给 str2int( ) 函数，而不是指定 acceptable 标签的整数编码为 1（如前面的示例所示）。代码如下所示。

```
>>> cola_sel.filter(lambda s: s['label'] == cola_sel.
features['label'].str2int('acceptable'))["sentence"][:3]
```

其输出结果与前一次执行的输出结果相同。

5. 使用 map( ) 函数处理数据

使用 datasets. Dataset. map( ) 函数遍历数据集，对集合中的每个样例应用一个处理函数，并修改样例的内容。以下代码片段演示了如何添加一个新的 len 特征（len 特征表示句子的长度）。

```
>>> cola_new = cola_sel.map(lambda e:{'len': len(e['sentence'])})
>>> pd.DataFrame(cola_new[0:3])
```

上述代码片段的输出结果如图 2 – 16 所示。

	idx	label	len	sentence
0	0	1	71	Our friends won't buy this analysis, let alone...
1	1	1	49	One more pseudo generalization and I'm giving up.
2	2	1	48	One more pseudo generalization or I'm giving up.

图 2 – 16　带有附加列的 cola 数据集

作为另一个示例，以下的代码片段将句子缩短为 20 个字符。代码中并没有创建新的特征，而是更新句子特征的内容。代码如下所示。

```
>>> cola_cut = cola_new.map(lambda e: {'sentence': e['sentence']
[:20] + '_'})
```

输出结果如图 2 – 17 所示。

	idx	label	len	sentence
0	0	1	71	Our friends won't bu_
1	1	1	49	One more pseudo gene_
2	2	1	48	One more pseudo gene_

图 2－17　更新后的 cola 数据集

6. 使用本地文件

为了从逗号分隔值（Comma－Separated Values，CSV）、文本（TXT）格式或 JavaScript 对象表示法（JavaScript Object Notation，JSON）格式的本地文件中加载数据集，可以将文件类型（csv、txt 或 json）传递给通用的 load_dataset( )加载脚本，实现方式如以下代码片段所示。假设在“../data/”文件夹下有三个 CSV 文件（a. csv、b. csv 和 c. csv），它们都是从 SST－2 数据集中随机选择的简单示例。可以加载单个文件，如 data1 对象所示；也可以合并多个文件，如 data2 对象所示；还可以进行数据集拆分，如 data3 对象所示。

```
from datasets import load_dataset
data1 = load_dataset('csv', data_files='../data/a.csv',
delimiter="\t")
data2 = load_dataset('csv', data_files=['../data/a.csv','../
data/b.csv','../data/c.csv'], delimiter="\t")
data3 = load_dataset('csv', data_files={'train':['../
data/a.csv','../data/b.csv'],'test':['../data/c.csv']},
delimiter="\t")
```

为了读取其他格式的文件，可以传递 json 或者 txt，而不是 csv 的文件类型。实现代码如下所示。

```
>>> data_json = load_dataset('json', data_files='a.json')
>>> data_text = load_dataset('text', data_files='a.txt')
```

到目前为止，本节讨论了如何加载、处理和操作托管在中心或本地硬盘上的数据集。接下来研究如何为 Transformer 模型训练准备数据集。

7. 为模型训练准备数据集

首先从分词过程开始。每个模型都有自己的分词模型，该模型在实际语言模型之前经过训练。分词模型将在下一章展开讨论。为了使用分词器，需要先安装 Transformer 库。下面的示例从预训练的 distilBERT－base－uncased 模型中加载分词器模型。使用 map( )函数和一个带有 lambda 的匿名函数将分词器应用于 data3 中的每个数据拆分。如果将 map( )函数的参数选项 batched 设置为 True，则会向分词器函数提供一批样例。默认情况下，batch_size 值为 1 000，这是传递给函数的每个批次的样例数。如果 map( )函数的参数选项 batched 并未设置为 True，则整个数据集将作为单个批次传递。代码片段如下所示。

```
from Transformer import DistilBERTTokenizer
tokenizer = \DistilBERTTokenizer.from_pretrained('distilBERT-base-uncased')
```

```
encoded_data3 = data3.map(lambda e: tokenizer( e['sentence'],
padding = True, truncation = True, max_length = 12), batched = True,
batch_size = 1000)
```

如以下输出所示，所得到的 data3 和 encoded_data3 之间存在着差异，两个额外的特征 attention_mask 和 input_ids 被添加到 encoded_data3 的训练数据集和测试数据集中。在本章前面的内容中，已经介绍过这两个特征。简而言之，input_ids 是对应于句子中每个标记的索引。这些特征是 Transformer 的 Trainer 类所需的预期特征，将在随后涉及微调语言模型的章节中加以讨论。

通常一次性地将几个句子［称为 batch（批次）］传递给分词器，然后将分词后的批次进一步传递给模型。为此，将每个句子填充到批次中的最大句子长度，或者填充到由参数 max_length 指定的特定最大长度（在该简单示例中为 12）。还可以截断较长的句子以适应最大长度。代码片段如下所示。

```
>>> data3
DatasetDict({
train: Dataset({
        features: ['sentence','label'], num_rows: 199 })
test: Dataset({
        features: ['sentence','label'], num_rows: 100 })})
>>> encoded_data3
DatasetDict({
train: Dataset({
        features: ['attention_mask','input_ids','label','sentence'],
        num_rows: 199 })
test: Dataset({
features: ['attention_mask','input_ids','label','sentence'],
        num_rows: 100 })})
>>> pprint(encoded_data3['test'][12])
{'attention_mask': [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
'input_ids': [101, 2019, 5186, 16010, 2143, 1012, 102, 0, 0, 0,
0, 0],'label': 0,'sentence':'an extremely unpleasant film .
'}
```

到目前为止，本节已经完成了对 datasets 库的讨论。本节讨论了有关 datasets 各个方面的功能，介绍了与 GLUE 相类似的基准测试，同时讨论了分类度量。在下一节中，将重点讨论如何在速度和内存方面对计算性能进行基准测试。

2.6 速度和内存的基准测试

仅比较大型模型在特定任务或基准测试上的分类性能不足以说明问题。现在必须考虑在给定的环境（RAM、CPU、GPU）下，特定模型在内存使用和速度方面的计算成本。训练和部署到生产环境进行推理的计算成本是需要衡量的两个主要指标值。

Transformer 库中的两个类 PyTorchBenchmark 和 TensorFlowBenchmark，可以对 TensorFlow 和 PyTorch 的模型进行基准测试。

在开始实验之前，需要执行以下命令来检查 GPU 的功能。

```
>>> import torch
>>> print(f"The GPU total memory is {torch.cuda.get_device_
properties(0).total_memory /(1024 **3)} GB")
The GPU total memory is 2.94921875 GB
```

输出结果来自 NVIDIA GeForce GTX 1050（3GB）。用户需要更强大的资源来完成高级实现。Transformer 库目前仅支持单设备基准测试。当在 GPU 上进行基准测试时，需要指出 Python 代码将在哪个 GPU 设备上运行，这是通过设置 CUDA_VISIBLE_DEVICES 环境变量来完成的。例如，export CUDA_VISIBLE_DEVICES = 0，0 表示将使用第一个 cuda 设备。

下面的代码示例探索了两个网格，比较了 4 个随机选择的预训练 BERT 模型，这 4 个模型的名称包含在 models 数组中。第二个要观察的参数是 sequence_lengths（序列长度）。在代码中将批次值设置为 4。如果用户的 GPU 容量更多，那么可以使用范围为 4 ~64 的批次值以及其他参数来扩展参数的搜索空间。

```
from Transformer import PyTorchBenchmark,
PyTorchBenchmarkArguments
models = ["BERT - base - uncased","distilBERT - base - uncased","distilroBERTa -
base", "distilBERT - base - german - cased"]
batch_sizes =[4]
sequence_lengths =[32,64, 128, 256,512]
args = PyTorchBenchmarkArguments(models =models, batch_
sizes =batch_sizes, sequence_lengths =sequence_lengths, multi_
process =False)
benchmark = PyTorchBenchmark(args)
```

重要提示

TensorFlow 的基准测试：本节中的代码示例用于 PyTorch 基准测试。对于 TensorFlow 基准测试，只需使用对应的类 TensorFlowBenchmarkArguments 和 TensorFlowBenchmark。

可以运行以下代码进行基准测试实验。

```
>>> results = benchmark.run()
```

运行需要一些时间，具体取决于用户的 CPU/GPU 容量和参数选择。如果用户面临内存不足的问题，那么可以采取以下措施来解决问题。

（1）重新启动内核或操作系统。

（2）启动前删除内存中所有不必要的对象。

（3）设置较低的批次大小，如 2，甚至 1。

以下输出显示推理速度性能，如图 2－18 所示。由于搜索空间有 4 个不同的模型和 5 个不同的序列长度，所以在结果中看到了 20 行内容。

```
====================        INFERENCE - SPEED - RESULT        ====================
--------------------------------------------------------------------------------
          Model Name             Batch Size     Seq Length     Time in s
--------------------------------------------------------------------------------
      bert-base-uncased              4               32            0.021
      bert-base-uncased              4               64            0.031
      bert-base-uncased              4              128            0.057
      bert-base-uncased              4              256             0.12
      bert-base-uncased              4              512            0.269
   distilbert-base-uncased           4               32            0.007
   distilbert-base-uncased           4               64            0.011
   distilbert-base-uncased           4              128            0.021
   distilbert-base-uncased           4              256            0.044
   distilbert-base-uncased           4              512            0.095
      distilroberta-base             4               32            0.009
      distilroberta-base             4               64            0.014
      distilroberta-base             4              128            0.025
      distilroberta-base             4              256            0.053
      distilroberta-base             4              512            0.118
 distilbert-base-german-cased        4               32            0.007
 distilbert-base-german-cased        4               64            0.012
 distilbert-base-german-cased        4              128            0.021
 distilbert-base-german-cased        4              256            0.044
 distilbert-base-german-cased        4              512            0.095
--------------------------------------------------------------------------------
```

图 2－18　推理速度性能

同样，观察到 20 种不同场景的推理内存使用情况，如图 2－19 所示。

```
====================        INFERENCE - MEMORY - RESULT        ====================
--------------------------------------------------------------------------------
          Model Name             Batch Size     Seq Length    Memory in MB
--------------------------------------------------------------------------------
      bert-base-uncased              4               32            1453
      bert-base-uncased              4               64            1487
      bert-base-uncased              4              128            1547
      bert-base-uncased              4              256            1661
      bert-base-uncased              4              512            1901
   distilbert-base-uncased           4               32            1908
   distilbert-base-uncased           4               64            1900
   distilbert-base-uncased           4              128            1900
   distilbert-base-uncased           4              256            1900
   distilbert-base-uncased           4              512            1900
      distilroberta-base             4               32            1907
      distilroberta-base             4               64            1900
      distilroberta-base             4              128            1900
      distilroberta-base             4              256            2098
      distilroberta-base             4              512            2492
 distilbert-base-german-cased        4               32            2499
 distilbert-base-german-cased        4               64            2492
 distilbert-base-german-cased        4              128            2492
 distilbert-base-german-cased        4              256            2491
 distilbert-base-german-cased        4              512            2491
--------------------------------------------------------------------------------
```

图 2－19　推理内存性能

为了观察不同参数之间的内存使用情况，将使用存储统计信息的 results 对象来绘制内存使用情况。以下代码片段绘制了跨模型和序列长度的时间推断性能。

```
import matplotlib.pyplot as plt
plt.figure(figsize=(8,8))
```

```
t = sequence_lengths
models_perf = [list(results.time_inference_result[m]['result']
[batch_sizes[0]].values()) for m in models]
plt.xlabel('Seq Length')
plt.ylabel('Time in Second')
plt.title('Inference Speed Result')
plt.plot(t, models_perf[0], 'rs --', t, models_perf[1], 'g - - .',
t, models_perf[2], 'b -- ^', t, models_perf[3], 'c -- o')
plt.legend(models)
plt.show()
```

如图 2 - 20 所示，两个 DistilBERT 模型的结果相近，并且性能优于其他两个模型。与其他模型相比，BERT - based - uncased 模型表现不佳，尤其是当序列长度增加时。

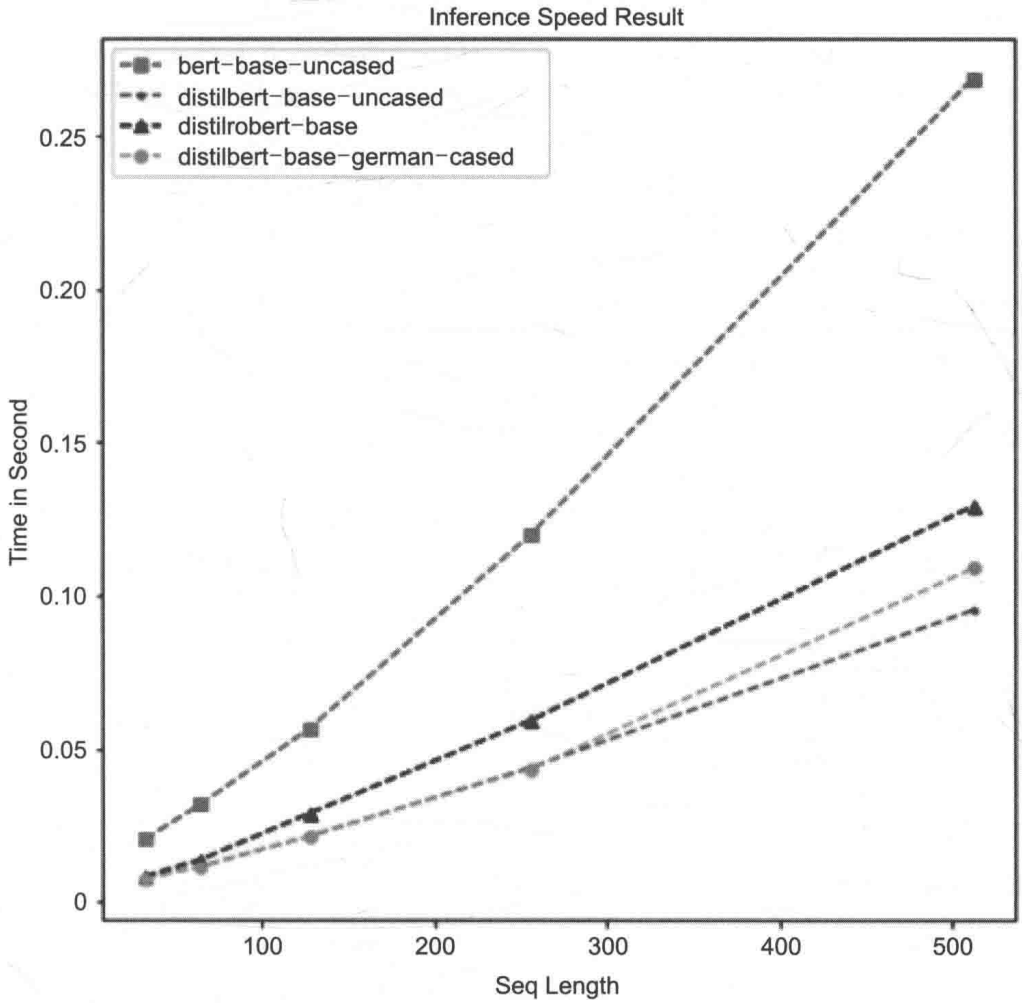


图 2 - 20　推理速度结果

为了绘制内存性能，需要使用 results 对象的 memory_inference_result 结果，而不是前面代码中显示的 time_inference_result。

至此结束了本节内容，并成功地完成了这一章的讨论。祝贺用户完成了安装、运行第一个 Hello World Transformer 程序、使用 datasets 库和基准测试！

2.7 本章小结

本章讨论了各种入门主题，还介绍了 Hello World Transformer 应用程序。另外，本章具有至关重要的作用，为将迄今所学知识应用于后续章节做准备。那么，到目前为止，读者学到了哪些知识点呢？首先，完成了环境设置和系统安装，其中，Anaconda 软件包管理器帮助用户为主流操作系统安装必要的模块；还研究了语言模型、社区提供的模型和分词过程；此外，引入了多任务（GLUE）和跨语言（XTREME）基准测试，这使这些语言模型变得更加强大和准确；讨论了 datasets 库，这有助于高效访问社区提供的自然语言处理数据集；最后，学习了如何根据内存使用和运行速度来评估特定模型的计算成本。Transformer 可以对 TensorFlow 和 PyTorch 模型进行基准测试。

本章中使用的模型由社区经过训练并共享。接下来，需要训练一种语言模型并向社区传播。

在下一章中，将学习如何训练 BERT 语言模型和分词器，并了解如何与社区共享这些语言模型和分词器。

第 2 部分 Transformer 模型：从自编码模型到自回归模型

在第 2 部分中，读者将了解自编码模型（如 BERT）和自回归模型（如 GPT）的体系结构，学习如何训练、测试和微调各种自然语言理解和自然语言生成问题的模型，还将学习如何与社区共享这些模型，以及如何微调社区共享的其他经过预训练的语言模型。

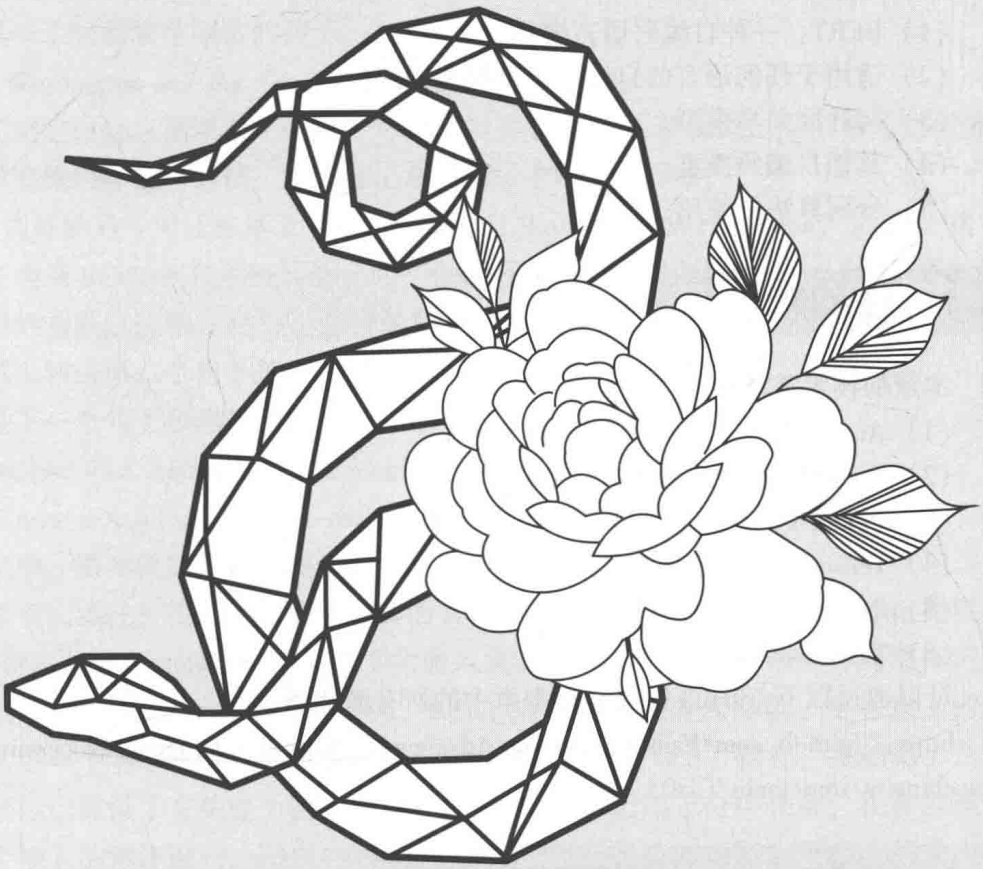
第 2 部分包括以下章节内容。

- 第 3 章：自编码语言模型。
- 第 4 章：自回归和其他语言模型。
- 第 5 章：微调文本分类语言模型。
- 第 6 章：微调标记分类语言模型。
- 第 7 章：文本表示。

3

第3章

自编码语言模型



上一章研究了如何通过 Hugging Face 的 Transformer 库使用典型的 Transformer 模型。到目前为止，所有的主题都包括了如何使用预定义的或预先构建的模型，很少涉及关于特定模型及其训练的信息。

在本章中，读者将了解如何从零开始在任何给定语言上训练自编码语言模型。这里的训练包括模型的预训练和特定任务的训练。首先，读者将了解有关 BERT 的基本知识及其工作原理，将使用一个简单的小语料库来训练语言模型；之后，将研究如何在 Keras 模型中使用该模型。

本章将介绍以下主题。

（1）BERT：一种自编码语言模型。

（2）适用于任何语言的自编码语言模型训练。

（3）与社区共享模型。

（4）其他自编码模型。

（5）分词算法的使用。

3.1 技术需求

本章的技术需求如下所示。

（1）Anaconda；

（2）Transformer（不低于 4.0.0 版本）；

（3）PyTorch（不低于 1.0.2 版本）；

（4）TensorFlow（不低于 2.4.0 版本）；

（5）datasets（不低于 1.4.1 版）；

（6）Tokenizers。

可以通过以下 GitHub 链接获得本章中的所有源代码：

https://github.com/PacktPublishing/Advanced-Natural-LanguageProcessing-with-Transformers/tree/main/CH03。

3.2 BERT：一种自编码语言模型

BERT 是第一批使用编码器 Transformer 堆栈（稍加修改后可用于语言建模）的自编

码语言模型之一。

BERT体系结构是基于Transformer库原始实现的多层Transformer编码器。Transformer模型本身最初用于机器翻译任务，但是BERT所做的主要改进是利用该体系结构的这一部分来提供更好的语言建模。这种语言模型经过预训练后，能够提供对所训练语言的全面理解。

3.2.1 BERT语言模型预训练任务

为了深入理解BERT所使用的带掩码机制的语言建模，首先明确定义其细节。带掩码机制的语言建模（Masked language modeling）是一种任务，用于训练一个输入模型（一个包含某些掩码标记的句子），并获得填充了掩码标记的整个句子的输出。掩码语言建模是如何以及为什么帮助模型在下游任务（如分类）上获得更好的结果？答案很简单：如果模型可以实现完形填空测试（一种通过填空来评估语言理解的语言测试），那么模型对语言本身就有一个大致的理解。对于其他任务，由于已经经过了预训练（通过语言建模），所以性能会更好。

下面是一个完形填空测试的例子。

George Washington was the first President of the ____ States.

用户期望空白处应该填补United。对于带掩码机制的语言模型，应用相同的任务，并且需要填充掩码标记。但是，掩码标记是从句子中随机选择的。

BERT训练的另一项任务是下一个句子预测（Next Sentence Prediction，NSP）。这个预训练任务确保BERT在预测掩码标记时不仅学习所有标记之间的关系，还帮助理解两个句子之间的关系。选择一对句子并传递给BERT，中间带有［SEP］拆分标记。从数据集中还可以知道第二个句子是否在第一个句子之后。

以下是下一个句子预测的示例。

It is required from reader to fill the blank.

Bitcoin price is way over too high compared to other altcoins.

在本例中，需要模型将其预测为否定（两个句子之间没有关联）。

这两项预训练任务使BERT能够理解语言本身。BERT标记嵌入为每个标记提供上下文嵌入（Contextual embedding）。上下文嵌入意味着每个标记都有一个与周围标记完全相关的嵌入。与Word2vec等模型不同，BERT为每个标记嵌入提供了更有意义的信息。另外，下一个句子预测任务使BERT能够更好地嵌入［CLS］标记。正如第1章所讨论的，该标记提供了有关整个输入的信息。［CLS］标记用于分类任务，在预训练部分学习整个输入的整体嵌入。图3-1显示了BERT模型的整体视图以及相应的输入和输出。

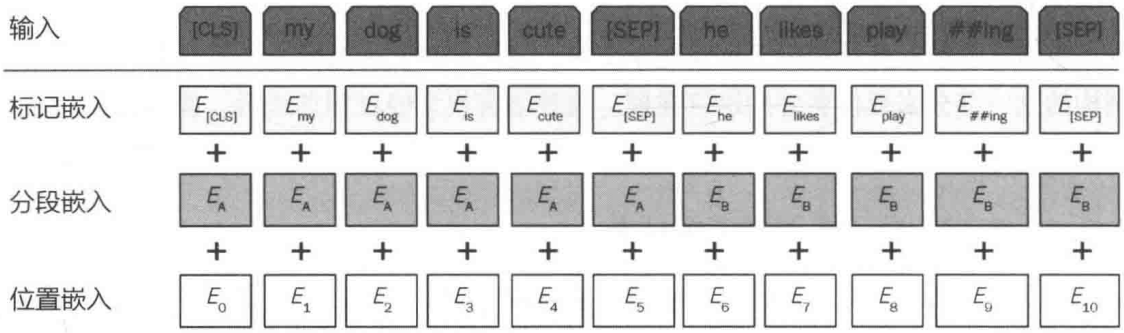


图 3－1　BERT 模型

3.2.2　对 BERT 语言模型的深入研究

分词器是许多自然语言处理应用程序在其各自管道中最重要的组成部分之一。BERT 使用 WordPiece 分词器。一般来说，WordPiece、SentencePiece 和 BytePairEncoding（BPE）是三种最广为人知的分词器，它们被用于不同的基于 Transformer 的体系结构，具体内容将在下一节中阐述。BERT 或者任何其他基于 Transformer 的体系结构使用子词进行分词的主要原因是，这些分词器具有处理未知标记的能力。

BERT 还使用位置编码来确保将标记的位置传递给模型。在第 1 章中，读者了解到 BERT 和类似的模型均使用非顺序操作（如密集的神经层）。传统模型，如基于长短期记忆神经网络和循环神经网络的模型，通过序列中标记的顺序获得位置。为了向 BERT 提供这些额外的信息，可以简单地使用位置编码。

BERT 的预训练（如自编码模型）为模型提供了语言方面的信息，但在实践中，在处理不同的问题（如序列分类、标记分类或问题回答系统）时，会使用模型输出的不同部分所包含的内容。

例如，在序列分类任务（如情绪分析或句子分类）的情况下，原始 BERT 文章建议必须使用最后一层的［CLS］嵌入。然而，还有其他一些研究借助 BERT 并且使用不同的技术进行分类（例如，使用所有标记的平均标记嵌入在最后一层部署长短期记忆网络，甚至在最后一层的顶部使用卷积神经网络）。用于序列分类的最后一个［CLS］嵌入可以被任何分类器使用，但建议（也是最常见的）是一个密集层，其输入大小等于最终标记嵌入大小，输出大小与 softmax( ) 激活函数的类别数量相同。当输出可能是多标签并且问题本身是多标签分类问题时，使用 sigmoid( ) 函数也是一种选择。

为了向读者提供有关 BERT 实际工作方式的更详细信息，图 3－2 显示了下一个句子预测任务的示例。注意，为了更好地理解，这里简化了分词过程。

BERT 模型具有不同的变体以及不同的设置。例如，输入的大小是可变的。在前面的示例中，输入的大小被设置为 512，模型作为输入可以获得的最大序列大小为 512。但是，该大小包括特殊标记［CLS］和［SEP］，因此最大序列将减小到 510。另外，使

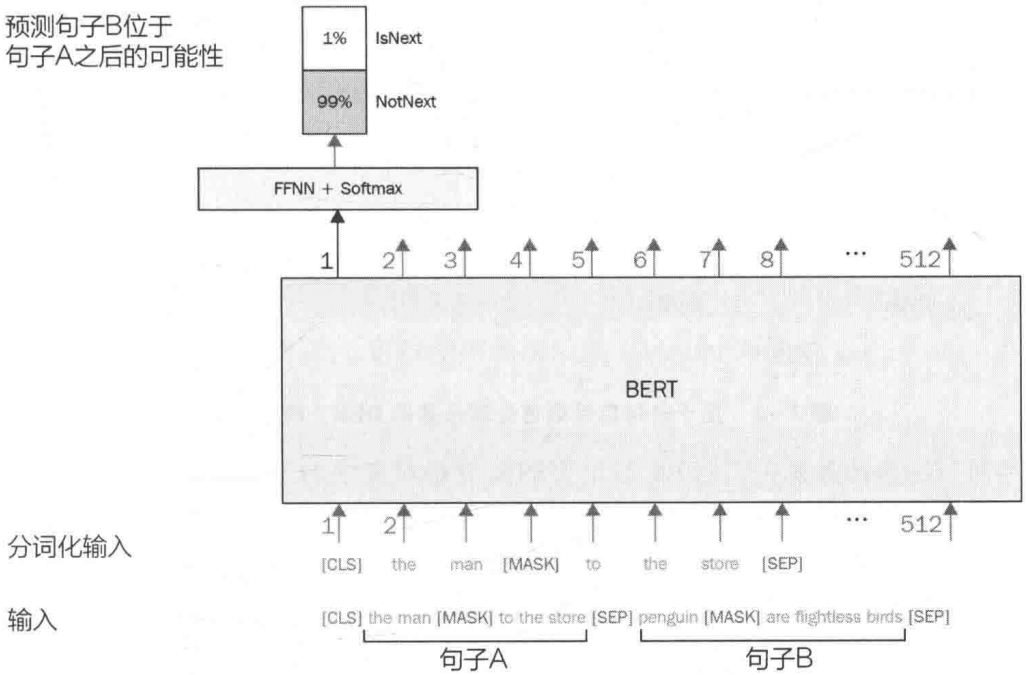


图 3－2　下一个句子预测任务的示例

用 WordPiece 作为分词器会产生子词标记，最初的序列包含更少的词。在分词之后，序列会增大，因为如果词在预训练语料库中不常见，分词器会将这些词分成子词。

图 3－3 所示为不同任务的 BERT 模型示例。对于一个命名实体识别任务，使用所有标记的输出，而不仅是［CLS］标记。在问题回答系统的情况下，使用［SEP］分隔符标记将问题和答案连接起来，并使用最后一层的 Start/End 以及 Span 输出来标注答案。在这种情况下，该段落是被提问的问题的上下文。

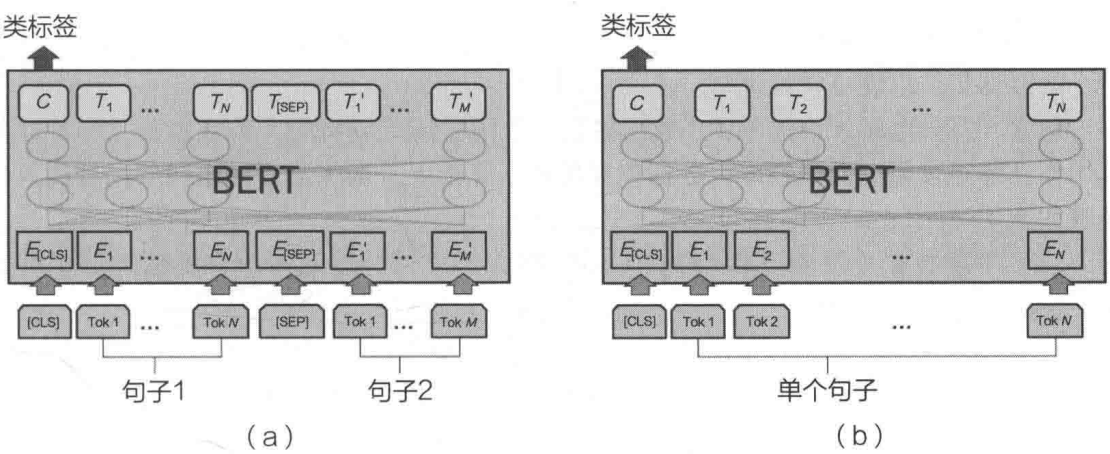
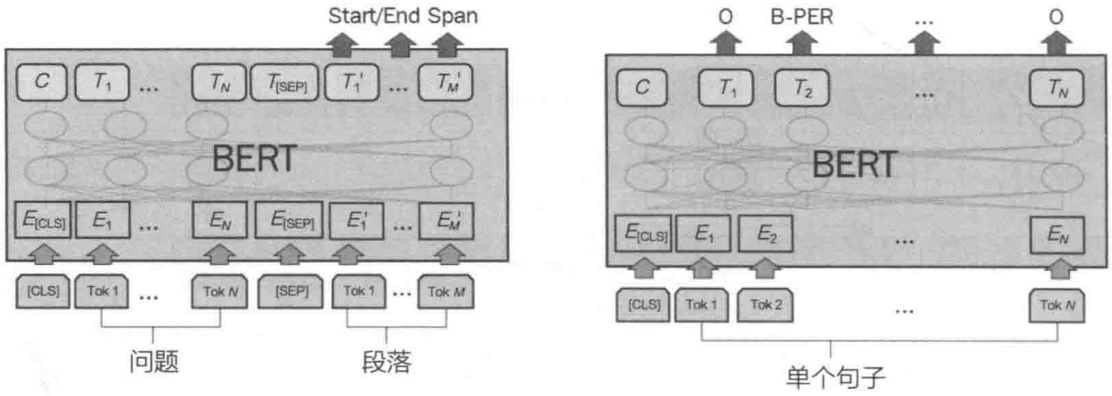


图 3－3　用于各种自然语言处理任务的 BERT 模型



（c）　　（d）

图 3－3　用于各种自然语言处理任务的 BERT 模型（续）

与所有这些任务无关，BERT 最重要的能力是对文本的上下文表示。BERT 在各种任务中成功的原因是 Transformer 编码器体系结构以密集向量的形式表示输入。可以通过非常简单的分类器将这些向量轻松地转换为输出。

到目前为止，本节介绍了 BERT 及其工作原理，并阐述了有关 BERT 可用于各种任务的详细信息以及相关体系结构的要点。

在下一节中，读者将学习如何预训练 BERT 并在训练后使用 BERT。

3.3　适用于任何语言的自编码语言模型训练

前面已经讨论了 BERT 的工作原理，并且读者已经可以使用 huggingface 库提供的预训练版本。在本节中，读者将学习如何使用 huggingface 库训练自己的 BERT。

在正式开始之前，必须拥有良好的训练数据，这些数据将用于语言建模。这些数据被称为语料库（corpus），通常是一大堆数据（有时是经过预处理和清洗的数据）。这些未标记的语料库必须适用于所训练语言模型的用例。例如，如果尝试为英语训练一个特殊的 BERT，虽然有很多巨大的、优秀的数据集，如 Common Crawl，这里仍倾向于选择较小的数据集，以便更快地训练。

包含 5 万条电影评论的 IMDB 数据集是一个用于情感分析的大型数据集，但如果将其用作训练语言模型的语料库，数据集规模并不算很大。训练语言模型和分词器的具体步骤如下。

（1）下载数据集并将其保存为“.txt”格式的文件，以用于语言模型和分词器训练。代码如下。

```
import pandas as pd
imdb_df = pd.read_csv("IMDB Dataset.csv")
reviews = imdb_df.review.to_string(index=None)
```

```
with open("corpus.txt", "w") as f:
    f.writelines(reviews)
```

（2）对分词器进行训练。tokenizers 库为 WordPiece 分词器提供了快速而简单的训练。为了在语料库中对其进行训练，需要运行以下代码。

```
>>> from tokenizers import BertWordPieceTokenizer
>>> bert_wordpiece_tokenizer =BertWordPieceTokenizer()
>>> bert_wordpiece_tokenizer.train("corpus.txt")
```

（3）访问训练后的词汇表。可以使用训练后的 tokenizer 对象的 get_vocab( )方法访问训练后的词汇表。代码如下所示。

```
>>> bert_wordpiece_tokenizer.get_vocab()
```

输出结果如下所示。

```
{'almod': 9111, 'events': 3710, 'bogart': 7647,
'slapstick': 9541, 'terrorist': 16811, 'patter': 9269,
'183': 16482, '##cul': 14292, 'sophie': 13109, 'thinki':
10265, 'tarnish': 16310, '##outh': 14729, 'peckinpah':
17156, 'gw': 6157, '##cat': 14290, '##eing': 14256,
'successfully': 12747, 'roomm': 7363, 'stalwart':
13347,...}
```

（4）保存分词器以供后面使用。使用 tokenizer 对象的 save_model( )方法并指定路径，保存分词器词汇表以供进一步使用。

```
>>> bert_wordpiece_tokenizer.save_model("tokenizer")
```

（5）调用 from_file( )函数重新加载。

```
>>> tokenizer = \BertWordPieceTokenizer.from_
file("tokenizer/vocab.txt")
```

（6）通过以下示例使用分词器。

```
>>> tokenized_sentence =\
tokenizer.encode("Oh it works just fine")
>>> tokenized_sentence.tokens
['[CLS]','oh','it','works','just','fine','[SEP]']
```

特殊标记［CLS］和［SEP］将自动添加到标记列表中，因为 BERT 需要这些标记来处理输入。

（7）使用分词器处理另一个句子。

```
>>> tokenized_sentence = \
tokenizer.encode("ohoh i thougt it might be workingg
well")
['[CLS]','oh','##o','##h','i','thoug','##t','it',
'might','be','working','##g','well','[SEP]']
```

（8）对于有噪声和拼写错误的文本，这似乎是一个不错的分词器。接下来可以保存分词器，并训练自己的 BERT。首先使用 Transformer 库中的 BertTokenizerFast，然后使用以下命令加载（4）中训练好的分词器。

```
>>> from Transformers import BertTokenizerFast
>>> tokenizer = BertTokenizerFast.from_pretrained("tokenizer")
```

由于 Hugging Face 文档建议使用 BertTokenizerFast，因此代码中使用了 BertTokenizerFast。还有一个分词器 BertTokenizer，根据库文档中的定义，它的实现速度不如 BertTokenizerFast 版本快。在大多数预训练模型的文档和卡片中，强烈建议使用 BertTokenizerFast 版本。

（9）准备用于快速训练的语料库，可以使用以下命令。

```
>>> from Transformers import LineByLineTextDataset
>>> dataset = LineByLineTextDataset(tokenizer=tokenizer,
                                    file_path="corpus.txt",
                                    block_size=128)
```

（10）提供一个用于带掩码机制的语言建模数据校对器（data collator）。

```
>>> from Transformers import DataCollatorForLanguageModeling
>>> data_collator = DataCollatorForLanguageModeling(
                     tokenizer=tokenizer,
                     mlm=True,
                     mlm_probability=0.15)
```

数据校对器获取数据并为训练做好准备。例如，上面的数据校对器获取数据，并以 0.15 的概率为带掩码机制的语言建模做准备。使用这种机制的目的在于即时进行预处理，从而减少计算资源的消耗。另外，这种机制会减慢训练过程，因为每个样本都必须在训练时即时进行预先处理。

（11）训练参数的同时为训练器提供训练阶段所需的信息，可以使用以下命令进行设置。

```
>>> from Transformers import TrainingArguments
>>> training_args =TrainingArguments(
                    output_dir="BERT",
                    overwrite_output_dir=True,
                    num_train_epochs=1,
                    per_device_train_batch_size=128)
```

（12）创建 BERT 模型，使用默认配置（包括注意力头的数量、Transformer 编码器的层数等）。

```
>>> from Transformers import BertConfig, BertForMaskedLM
>>> bert = BertForMaskedLM(BertConfig())
```

（13）创建一个训练器对象。

```
>>> from Transformers import Trainer
>>> trainer = Trainer(model = bert,
                      args = training_args,
                      data_collator = data_collator,
                      train_dataset = dataset)
```

（14）使用以下命令训练自己的语言模型。

```
>>> trainer.train()
```

训练器将显示一个进度条，如图 3 – 4 所示，指示训练的进度。

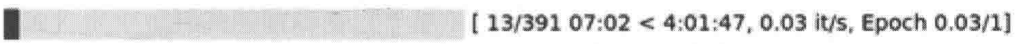


图 3 – 4　BERT 模型的训练进度

在模型训练期间，训练器将创建一个名为“runs”的日志目录，按步骤存储检查点，如图 3 – 5 所示。

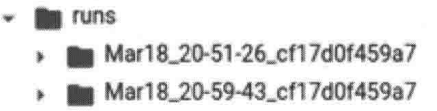


图 3 – 5　BERT 模型的检查点

（15）使用以下命令保存模型。

```
>>> trainer.save_model("MyBERT")
```

到目前为止，读者已经学会了如何从零开始针对任何特定语言来训练 BERT 模型，以及如何使用自己准备的语料库来训练分词器和 BERT 模型。

（16）为 BERT 提供的默认配置是训练过程中最重要的部分，这些配置定义了 BERT 的体系结构及其超参数。可以使用以下代码查看这些参数。

```
>>> from Transformers import BertConfig
>>> BertConfig()
```

输出结果如图 3 – 6 所示。

如果读者希望从原始 BERT 配置复制 Tiny（微型）、Mini（迷你型）、Small（小型）、Base（基础）以及相对模型，那么可以更改图 3 – 7 中的设置。

```
BertConfig {
  "attention_probs_dropout_prob": 0.1,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "transformers_version": "4.4.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 30522
}
```

图 3－6　查看 BERT 模型的配置

	H=128	H=256	H=512	H=768
L=2	2/128 (BERT-Tiny)	2/256	2/512	2/768
L=4	4/128	4/256 (BERT-Mini)	4/512 (BERT-Small)	4/768
L=6	6/128	6/256	6/512	6/768
L=8	8/128	8/256	8/512 (BERT-Medium)	8/768
L=10	10/128	10/256	10/512	10/768
L=12	12/128	12/256	12/512	12/768 (BERT-Base)

图 3－7　BERT 模型的配置参数

请注意，更改这些参数（特别是参数 max_position_embeddings、num_attention_heads、num_hidden_layers、intermediate_size 和 hidden_size）会直接影响训练时间。调大这些参数，会大幅增加大型语料库的训练时间。

（17）使用以下代码可以轻松地为一个微型版本的 BERT 进行全新的配置，以便更快速地进行训练。

```
>>> tiny_bert_config = BertConfig(max_position_
embeddings=512, hidden_size=128,
          num_attention_heads=2,
          num_hidden_layers=2,
          intermediate_size=512)
>>> tiny_bert_config
```

上述代码片段的输出结果如图 3－8 所示。

```
BertConfig {
  "attention_probs_dropout_prob": 0.1,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 128,
  "initializer_range": 0.02,
  "intermediate_size": 512,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 2,
  "num_hidden_layers": 2,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "transformers_version": "4.4.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 30522
}
```

图 3－8　微型 BERT 模型的配置参数

（18）使用以下配置创建一个微型 BERT 模型。

```
>>> tiny_bert = BertForMaskedLM(tiny_bert_config)
```

（19）使用相同的参数训练这个新的微型 BERT 模型。

```
>>> trainer = Trainer(model = tiny_bert, args = training_args,
                      data_collator = data_collator,
                      train_dataset = dataset)
>>> trainer.train()
```

其输出结果如图 3－9 所示。

[9/391 00:17 < 15:43, 0.40 it/s, Epoch 0.02/1]

图 3－9　微型 BERT 模型的训练进度①

很明显，训练时间大大缩短了。但需要注意的是，这是一个微型版本的 BERT，具有较少的层和参数，因此它比 BERT 基础模型的训练效果要差。

到目前为止，读者已经学会了如何从头开始训练自己的模型。但需要注意的是，在处理用于训练语言模型的数据集或者利用数据集执行特定于任务的训练时，使用 datasets 库是更好的选择。

（20）BERT 模型也可以作为嵌入层与任何深度学习模型相结合。例如，可以加载任何预训练的 BERT 模型或者在上一步中训练过的自定义的版本。以下代码显示了要在 Keras 模型中使用 BERT 模型，必须先按以下方法进行加载。

```
>>> from Transformers import TFBertModel, BertTokenizerFast
>>> bert = TFBertModel.from_pretrained("bert-base-uncased")
```

① 原著此处有误，应该是模型的训练进度而不是模型的配置。——译者注

```
>>> tokenizer = BertTokenizerFast.from_pretrained("bert-base-uncased")
```

（21）此处不需要整个模型，可以使用以下代码访问各个层。

```
>>> bert.layers
[<Transformers.models.bert.modeling_tf_bert.
TFBertMainLayer at 0x7f72459b1110>]
```

（22）正如所见，TFBertMainLayer 中只有一个层，用户可以在 Keras 模型中访问该层。在使用之前，最好先测试一下，观察其输出结果是什么。

```
>>> tokenized_text = tokenizer.batch_encode_plus(
                     ["hello how is it going with you",
                     "lets test it"],
                     return_tensors="tf",
                     max_length=256,
                     truncation=True,
                     pad_to_max_length=True)
>>> bert(tokenized_text)
```

输出结果如图 3－10 所示。

```
TFBaseModelOutputWithPooling([('last_hidden_state',
                               <tf.Tensor: shape=(2, 256, 768), dtype=float32, numpy=
                               array([[[ 1.00471362e-01,  6.77026287e-02, -8.33595246e-02, ...,
                                        -4.93304580e-01,  1.16539136e-01,  2.26647347e-01],
                                       [ 3.23623657e-01,  3.70719165e-01,  6.14685774e-01, ...,
                                        -6.27267540e-01,  3.79083097e-01,  7.05310702e-02],
                                       [ 1.99532971e-01, -8.75509441e-01, -6.47868365e-02, ...,
                                        -1.28077380e-02,  3.07651043e-01, -2.07325034e-02],
                                       ...,
                                       [-6.53299838e-02,  1.19046383e-01,  5.76846600e-01, ...,
                                        -2.95460820e-01,  2.49744654e-02,  1.13964394e-01],
                                       [-2.64715493e-01, -7.86386207e-02,  5.47280848e-01, ...,
                                        -1.37515247e-01, -5.94691373e-02, -5.17928638e-02],
                                       [-2.44958848e-01, -1.14799395e-01,  5.92173815e-01, ...,
                                        -1.56882048e-01, -3.39757390e-02, -8.46135616e-02]],

                                      [[ 2.94558890e-02,  2.30868042e-01,  2.92651534e-01, ...,
                                        -1.30421281e-01,  1.89659461e-01,  4.68427837e-01],
                                       [ 1.70523107e+00,  6.91360056e-01,  7.31509984e-01, ...,
                                         2.89302200e-01,  5.36758840e-01, -1.54553086e-01],
                                       [ 1.04596823e-01,  9.63676572e-02,  6.99661374e-02, ...,
                                        -4.15922999e-01, -1.18989825e-01, -6.72240376e-01],
                                       ...,
                                       [ 8.00909758e-01,  2.38983199e-01,  4.15492684e-01, ...,
                                         3.90530713e-02,  2.34373003e-01,  1.22278236e-01],
                                       [ 2.60862708e-01,  4.43267114e-02,  3.63648295e-01, ...,
                                        -7.53704458e-04,  3.84620279e-02, -2.14213312e-01],
                                       [-2.30111778e-01, -4.98388559e-01, -1.26496106e-02, ...,
                                         4.49867934e-01,  6.16019145e-02, -2.61357218e-01]]],
                                     dtype=float32)>),
                              ('pooler_output',
                               <tf.Tensor: shape=(2, 768), dtype=float32, numpy=
                               array([[-0.9204854 , -0.37138987, -0.6051259 , ..., -0.4473697 ,
                                       -0.64347583,  0.9423271 ],
                                      [-0.8854158 , -0.26547667,  0.21015054, ...,  0.17237163,
                                       -0.6402989 ,  0.8888342 ]], dtype=float32)>)])
```

图 3－10　BERT 模型的输出结果

从图 3－10 可以看出，BERT 模型有两个输出：一个输出显示最后一个隐藏状态；另一个输出显示池输出结果。最后一个隐藏状态提供了在开始和结束时分别来自 BERT

的所有标记，包括［CLS］标记和［SEP］标记。

（23）至此，用户已经了解了有关 BERT 的 TensorFlow 版本的更多信息，可以使用以下新嵌入来创建 Keras 模型。

```
from tensorflow import keras
import tensorflow as tf
max_length = 256
tokens = keras.layers.Input(shape=(max_length,),
                            dtype=tf.dtypes.int32)
masks = keras.layers.Input(shape=(max_length,),
                           dtype=tf.dtypes.int32)
embedding_layer = bert.layers[0]([tokens,masks])[0]
[:,0,:]
dense = tf.keras.layers.Dense(units=2,
          activation="softmax")(embedding_layer)
model = keras.Model([tokens,masks],dense)
```

（24）模型对象（Keras 模型）有两个输入：一个输入用于标记；另一个输入用于掩码。标记包含来自分词器输出的 token_ids，掩码将包含 attention_mask。接下来，尝试查看会发生什么现象。

```
>>> tokenized = tokenizer.batch_encode_plus(
["hello how is it going with you",
"hello how is it going with you"],
return_tensors="tf",
max_length= max_length,
truncation=True,
pad_to_max_length=True)
```

（25）在使用分词器时，设置参数 max_length、truncation 和 pad_to_max_length 非常重要。这些参数将输出填充到之前定义的最大长度 256，以确保输出为可用形状。现在，可以使用以下示例运行模型。

```
>>>model([tokenized["input_ids"],tokenized["attention_mask"]])
```

输出结果如图 3－11 所示。

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[0.45928752, 0.5407125 ],
       [0.45928752, 0.5407125 ]], dtype=float32)>
```

图 3－11　BERT 模型的分类输出

（26）在训练模型时，需要调用 compile( ) 函数进行编译。

```
>>> model.compile(optimizer="Adam",
loss="categorical_crossentropy",
```

```
metrics = ["accuracy"])
>>> model.summary()
```

输出结果如图 3－12 所示。

```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_tokens (InputLayer)       [(None, 256)]        0
__________________________________________________________________________________________________
input_masks (InputLayer)        [(None, 256)]        0
__________________________________________________________________________________________________
bert (TFBertMainLayer)          multiple             109482240   input_tokens[0][0]
                                                                 input_masks[0][0]
__________________________________________________________________________________________________
tf.__operators__.getitem_3 (Sli (None, 768)          0           bert[3][0]
__________________________________________________________________________________________________
output_layer (Dense)            (None, 2)            1538        tf.__operators__.getitem_3[0][0]
==================================================================================================
Total params: 109,483,778
Trainable params: 109,483,778
Non-trainable params: 0
__________________________________________________________________________________________________
```

图 3－12　BERT 模型的摘要信息

（27）从模型的摘要信息中可以看出，该模型包含 109 483 778 个可训练参数，包括 BERT。但是，如果对 BERT 模型进行预训练，并且希望在对应特定任务的训练中冻结模型，则可以使用以下命令。

```
>>> model.layers[2].trainable = False
```

由本节内容可知，嵌入层的层索引是 2，因此可以方便地冻结模型。如果重新调用 summary( ) 函数，结果发现可训练参数减少到 1 538，即最后一层参数的数量，如图 3－13 所示。

```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_tokens (InputLayer)       [(None, 256)]        0
__________________________________________________________________________________________________
input_masks (InputLayer)        [(None, 256)]        0
__________________________________________________________________________________________________
bert (TFBertMainLayer)          multiple             109482240   input_tokens[0][0]
                                                                 input_masks[0][0]
__________________________________________________________________________________________________
tf.__operators__.getitem_3 (Sli (None, 768)          0           bert[3][0]
__________________________________________________________________________________________________
output_layer (Dense)            (None, 2)            1538        tf.__operators__.getitem_3[0][0]
==================================================================================================
Total params: 109,483,778
Trainable params: 1,538
Non-trainable params: 109,482,240
__________________________________________________________________________________________________
```

图 3－13　具有较少可训练参数的 BERT 模型摘要信息

（28）如前所述，使用 IMDB 情感分析数据集来训练语言模型。现在，可以使用该数据集训练基于 Keras 的情感分析模型，但首先需要准备输入和输出。

```
import pandas as pd
imdb_df = pd.read_csv("IMDB Dataset.csv")
```

```
reviews = list(imdb_df.review)
tokenized_reviews = \
tokenizer.batch_encode_plus(reviews, return_tensors = "tf",
            max_length = max_length,
            truncation = True,
            pad_to_max_length = True)
import numpy as np
train_split = int(0.8 * len(tokenized_reviews["attention_mask"]))
train_tokens = tokenized_reviews["input_ids"][:train_split]
test_tokens = tokenized_reviews["input_ids"][train_split:]
train_masks = tokenized_reviews["attention_mask"][:train_split]
test_masks = tokenized_reviews["attention_mask"][train_split:]①
sentiments = list(imdb_df.sentiment)
labels = np.array([[0,1] if sentiment == "positive" else\
[1,0] for sentiment in sentiments])
train_labels = labels[:train_split]
test_labels = labels[train_split:]
```

（29）当数据准备就绪后，就可以拟合模型。

```
>>> model.fit([train_tokens,train_masks],train_labels, epochs = 5)
```

在拟合好模型后，模型就可以使用了。至此，读者已经了解了如何为分类任务执行模型训练，并学习了如何保存模型。在下一节中，读者将继续学习如何将训练好的模型提供给社区加以共享。

3.4 与社区共享模型

Hugging Face 提供了一种非常易于使用的模型共享机制。

（1）使用以下命令行工具，即可登录。

```
Transformers - cli login
```

（2）使用自己的账户信息登录后，可以创建存储库。

```
Transformers - cli repo create a - fancy - model - name
```

（3）为 a - fancy - model - name 参数输入任何模型名称，然后必须确保安装了 Git LFS。

```
git lfs install
```

Git LFS 是用于处理大型文件的 Git 扩展。Hugging Face 预训练模型通常是大型文件，Git 需要额外库（如 LFS）才能处理。

---

① 原著此处代码格式以及内容有误，译者已经根据源代码进行了勘误。——译者注

（4）克隆刚刚创建的存储库。

```
git clone https://huggingface.co/username/a-fancy-model-name
```

（5）根据需要从存储库中添加和删除数据，然后，与 Git 的用法一样，必须运行以下命令。

```
git add . && git commit -m "Update from $USER"
git push
```

自编码模型依赖原始 Transformer 的左侧编码器，在解决分类问题时非常高效。尽管 BERT 是自编码模型的一个典型示例，但相关文献中讨论了许多替代方案。接下来，继续讨论这些重要的替代方案。

3.5 了解其他自编码模型

在本节中，评述自编码模型的替代方案，这些方案对原始的 BERT 稍微进行了修改。这些重新实现的替代方案使用了许多改进方法，包括优化预训练流程以及层或头的数量、提高数据质量、设计更好的目标函数等，从而实现了更好的下游任务。这些替代方案改进的方法大致分为两部分：架构设计和预训练控制。

社区近期分享了许多有效的替代方案，因此本节不可能对其全部进行分析和阐述，只讨论文献中引用最多的一些模型，以及自然语言处理基准测试中使用最多的模型。接下来从 Albert 模型开始学习，它是对 BERT 的重新实现，该模型特别关注架构设计的选择。

3.5.1 Albert 模型概述

语言模型的性能会随着其规模的增大而提高。然而，由于内存的限制和较长的训练时间，所以训练大规模的模型变得更具挑战性。为了解决这些问题，Google 团队提出了 Albert 模型［一种用于语言表示的自监督学习的轻量型 BERT（A Lite BERT）］，这实际上是对 BERT 体系结构的重新实现，它利用了几种新技术来减少内存消耗并提高训练速度。新的设计使语言模型的可伸缩性比原来的 BERT 模型好得多。参数的数量减少为原来的 BERT 模型的 1/18，另外 Albert 的训练速度比原来的大型 BERT 模型快 1.7 倍。

Albert 模型主要包括对原始 BERT 的以下三处修改。

（1）分解嵌入参数化（Factorized embedding parameterization）。

（2）跨层参数共享（Cross-layer parameter sharing）。

（3）句间连贯性损失（Inter-sentence coherence loss）。

前两处修改是参数缩减方法，与原始 BERT 模型的大小和内存消耗问题相关。第三处修改对应于一个新的目标函数——句子顺序预测（Sentence-Order Prediction，SOP），它取代了原始 BERT 的下一个句子预测（Next Sentence Prediction，NSP）任务，这有助于模型的精简和性能的提高。

分解嵌入参数化用于将大的词汇表嵌入矩阵分解为两个小矩阵，从而将隐藏层的大小与词汇表的大小分开。这种分解将嵌入参数从 $O(V \times H)$ 减少到 $O(V \times E + E \times H)$。其中，$V$ 是词汇表；H 是隐藏层大小；E 是嵌入，如果满足 $H >> E$，则可以更有效地使用整个模型参数。

跨层参数共享可以防止参数总数随着网络的加深而增加。该技术被认为是提高参数效率的另一种方法，因为可以通过共享或复制来保持较小的参数数量。在最初的论文中，论文作者尝试了许多共享参数的方法。例如，仅跨层共享 FF 参数，或者仅共享关注的参数或整个参数。

Albert 模型的另一处改进是句间连贯性损失。如前所述，BERT 体系结构利用了两种损耗计算方法：带掩码机制的语言建模（Masked Language Modeling，MLM）的损失计算和下一个句子预测的损失计算。下一个句子预测附带了二进制交叉熵损失，用于预测原始文本中是否有两个片段出现在同一行中。通过从不同的文件中选择两个部分来获得负样例。然而，Albert 团队指出下一个句子预测是一个主题预测问题，这被认为是一个相对容易的问题。因此，研究小组提出了一个主要基于连贯性而非主题预测的损失函数。它们利用侧重于建模句子间连贯性而不是主题预测的句子顺序预测损失函数。句子顺序预测损失函数使用与 BERT 相同的正样例技术（即同一文档中的两个连续片段）；而作为负样例，使用相同的两个连续片段，但顺序交换。然后，该模型在语篇层面上学习连贯属性之间更细粒度的区别。

（1）将原始的 BERT 和 Albert 配置与 Transformer 库进行比较。下面的代码显示了如何配置基于 BERT 的初始模型。从输出结果可以看出，参数数量约为 11×10^7。

```
#BERT - BASE(L = 12, H = 768, A = 12, Total Parameters = 110M)
>> from Transformers import BertConfig, BertModel
>> bert_base = BertConfig()
>> model = BertModel(bert_base)
>> print(f"{model.num_parameters() /(10 * *6)}million parameters")
109.48224 million parameters
```

（2）使用下面的代码片段展示如何使用 Transformer 库中的两个类 AlbertConfig 和 AlbertModel 定义 Albert 模型。

```
# Albert - base Configuration
>>> from Transformers import AlbertConfig, AlbertModel
>>> albert_base = AlbertConfig(hidden_size = 768,
                        num_attention_heads = 12,
                        intermediate_size = 3072,)
>>> model = AlbertModel(albert_base)
>>> print(f"{model.num_parameters() /(10 **6)}million parameters")
11.683584 million parameters
```

因此，默认的 Albert 配置指向 Albert - xxlarge。需要设置隐藏大小、注意力头的数量和中间大小，以拟合 Albert 基础模型。以上代码片段显示 Albert 基础模型的参数数量

约为 11×10^6，是 BERT 基础模型的 1/10。关于 Albert 的原始论文报告的基准测试如图 3－14 所示。

Model		Parameters	SQuAD1.1	SQuAD2.0	MNLI	SST-2	RACE	Avg	Speedup
BERT	base	108M	90.4/83.2	80.4/77.6	84.5	92.8	68.2	82.3	4.7x
	large	334M	92.2/85.5	85.0/82.2	86.6	93.0	73.9	85.2	1.0
ALBERT	base	12M	89.3/82.3	80.0/77.1	81.6	90.3	64.0	80.1	5.6x
	large	18M	90.6/83.9	82.3/79.4	83.5	91.7	68.5	82.4	1.7x
	xlarge	60M	92.5/86.1	86.1/83.1	86.4	92.4	74.8	85.5	0.6x
	xxlarge	235M	94.1/88.3	88.1/85.1	88.0	95.2	82.3	88.7	0.3x

图 3－14　关于 Albert 的原始论文报告的基准测试

（3）基于以上操作步骤，同时为了从头开始训练 Albert 模型，需要使用统一的 Transformers 应用程序接口，安装前面章节的 BERT 训练中已经说明的类似阶段。这里不再重构相同的步骤。现在加载一个已经训练过的 Albert 模型。代码如下所示。

```
from Transformers import AlbertTokenizer, AlbertModel
tokenizer = \
AlbertTokenizer.from_pretrained("albert-base-v2")
model = AlbertModel.from_pretrained("albert-base-v2")
text = "The cat is so sad."
encoded_input = tokenizer(text, return_tensors='pt')
output = model(**encoded_input)
```

（4）上述代码片段从 Hugging Face 中心下载 Albert 模型权重及其配置，或者从本地缓存目录下载［如果之前已经缓存，就意味着用户曾经调用过 AlbertTokenizer. from_pretrained( )函数］。由于模型对象是一个预先训练过的语言模型，所以现在使用该模型做的事情是有限的。需要在下游任务中训练语言模型，以便能够使用该语言模型进行推理，这将是后面章节的主要主题。可以利用带掩码机制的语言模型目标，代码如下所示。

```
from Transformers import pipeline
fillmask = pipeline('fill-mask', model='albert-base-v2')
pd.DataFrame(fillmask("The cat is so [MASK]."))
```

输出结果如图 3－15 所示。

sequence	score	token	token_str
[CLS] the cat is so cute.[SEP]	0.281025	10901	_cute
[CLS] the cat is so adorable.[SEP]	0.094893	26354	_adorable
[CLS] the cat is so happy.[SEP]	0.042963	1700	_happy
[CLS] the cat is so funny.[SEP]	0.040976	5066	_funny
[CLS] the cat is so affectionate.[SEP]	0.024233	28803	_affectionate

图 3－15　Albert 模型版本 2 的填充掩码的输出结果

fill－mask（填充掩码）管道使用 softmax( )函数计算每个词汇表标记的得分，并对最可能的标记进行排序，其中 cute 是赢家，概率得分为 0.281。用户可能会注意到，

token_str 列中的项以下划线（_）字符开头，这是 Albert 分词器的元空间组件造成的。

接下来，讨论下一个替代方案：鲁棒优化的 BERT 预训练方法（Robustly Optimized BERT pre-training Approach，RoBERTa）。该方案侧重于预训练阶段。

3.5.2 RoBERTa 模型

RoBERTa 是另一种流行的 BERT 重新实现。RoBERTa 模型在训练策略方面的改进远远多于架构设计方面的改进。在 GLUE 上的几乎所有独立任务中，RoBERTa 的性能都优于 BERT。动态掩码是其原始设计选择之一。虽然静态掩码在某些任务中表现更好，但 RoBERTa 团队证明动态掩码在整体性能上表现良好。相对于 BERT 模型，RoBERTa 模型的改进总结如下。

RoBERTa 模型在架构设计上的改进如下所示。

（1）删除下一句预测训练目标。

（2）动态更改掩码模式，而不是静态掩码。这是通过在向模型提供序列时生成掩码模式来完成的。

（3）字节对编码 BPE 子词分词器。

RoBERTa 模型在预训练控制上的改进如下所示。

（1）控制训练数据：使用更多的数据（如 160GB），而原来在 BERT 中使用的数据规模是 16GB。研究中不仅考虑了数据的大小，还考虑了数据的质量和多样性。

（2）更长的迭代步骤：长达 5×10^5 的预训练步骤。

（3）更长的批次。

（4）更长的序列，从而减少填充。

（5）一个庞大的 50KB 对编码词汇表，而不是 30KB 对编码词汇表。

基于 Transformer 统一的应用程序接口，正如在前面所述的 Albert 模型管道中一样，对 RoBERTa 模型进行如下初始化。

```
>>> from Transformers import RobertaConfig, RobertaModel
>>> conf = RobertaConfig()
>>> model = RobertaModel(conf)
>>> print(f"{model.num_parameters() /(10 ** 6)}million parameters")
109.48224 million parameters
```

为了加载预先训练的模型，可以执行以下代码。

```
from Transformers import RobertaTokenizer, RobertaModel
tokenizer = \
RobertaTokenizer.from_pretrained('roberta-base')
model = RobertaModel.from_pretrained('roberta-base')
text = "The cat is so sad ."
encoded_input = tokenizer(text, return_tensors='pt')
output = model( ** encoded_input)
```

这些代码行演示了模型如何处理给定的文本。最后一层的输出表示目前没有作用。正如本节多次提到的，需要微调主要的语言模型。以下执行代码使用 RoBERTa 基础模型应用填充掩码的功能。

```
>>> from Transformers import pipeline
>>> fillmask = pipeline("fill-mask",model="roberta-base",
                        tokenizer=tokenizer)
>>> pd.DataFrame(fillmask("The cat is so <mask> ."))
```

输出结果如图 3-16 所示。

sequence	score	token	token_str
<s>The cat is so cute.</s>	0.191843	11962	Ġcute
<s>The cat is so sweet.</s>	0.051524	4045	Ġsweet
<s>The cat is so funny.</s>	0.033595	6269	Ġfunny
<s>The cat is so handsome.</s>	0.032893	19222	Ġhandsome
<s>The cat is so beautiful.</s>	0.032314	2721	Ġbeautiful

图 3-16 RoBERTa 基础模型填充掩码任务的结果

与前面所述的 Albert 填充掩码模型一样，此管道对合适的候选词进行排序。请忽略标记中的前缀 Ġ，这是字节级字节对编码分词器生成的编码空格字符，将在后面加以讨论。读者应该已经注意到，在 Albert 和 RoBERTa 管道中使用了［MASK］和 <mask> 标记，以便为掩码标记保留位置。这是基于分词器的配置。为了了解将使用哪个标记表达式，可以检查 tokenizer. mask_token。请参见以下代码的执行结果。

```
>>> tokenizer = \
AlbertTokenizer.from_pretrained('albert-base-v2')
>>> print(tokenizer.mask_token)
[MASK]
>>> tokenizer = \
RobertaTokenizer.from_pretrained('roberta-base')
>>> print(tokenizer.mask_token)
<mask>
```

为了确保正确使用掩码标记，可以在管道中添加 fillmask. tokenizer. mask_token 表达式。代码如下所示。

```
fillmask(f"The cat is very \
{fillmask.tokenizer.mask_token}.")
```

3.5.3 ELECTRA 模型

ELECTRA 模型（Kevin Clark 等于 2020 年提出）侧重于一种新的掩码语言模型，该模型利用替换的标记检测训练目标。在预训练期间，该模型被迫学习以区分真实输入标

记和合成生成的替换，其中合成负样例是从看似合理的标记，而不是随机采样的标记中采样的。Albert 模型指出 BERT 的下一个句子预测目标是一个主题预测问题，并使用了低质量的负样例。ELECTRA 模型训练两个神经网络——一个生成器和一个鉴别器，前者产生高质量的负样例，而后者区分原始标记和替换标记。用户从计算机视觉领域了解到生成式对抗网络（Generative Adversarial Networks，GAN），其中生成器 G 生成假图像并试图欺骗鉴别器 D，鉴别器网络试图避免被欺骗。ELECTRA 模型采用几乎相同的生成器-鉴别器方法，以高质量的负样例替换原始标记，这些样例看似合理，但实际上是以合成方式生成的。

为了避免在其他示例中重复相同的代码，本节仅为 ELECTRA 生成器提供一个简单的填充掩码示例。代码如下所示。

```
fillmask = \
pipeline("fill-mask", model="google/electra-small-generator")
fillmask(f"The cat is very {fillmask①.tokenizer.mask_token} .")
```

到此为止，终于完成了自编码模型所有相关的内容。接下来，继续讨论分词算法。分词算法对 Transformer 的成功具有重要影响。

3.6 使用分词算法

在本章的开头部分，使用了一个特定的分词器（即 BertWordPieceTokenizer）来训练 BERT 模型。现在有必要在这里详细讨论一下分词过程。分词是一种将文本输入拆分为标记并在输入神经网络架构之前为每个标记分配标识符的方法。最直观的方法是根据空间将序列分割成更小的块。然而，这种方法不能满足某些语言（如日语）的要求，还可能导致巨大的词汇问题。几乎所有的 Transformer 模型都利用子词（subword）分词法，它不仅可以降低维度，还可以对训练中未发现的罕见（或者未知）词进行编码。分词的基础是每个单词（包括稀有词或未知词）都可以被分解成有意义的小块，这些小块是训练语料库中常见的符号。

Moses 和 nltk 库中开发的一些传统分词器均应用了先进的基于规则的技术。但是，与 Transformer 库一起使用的分词算法是基于自监督学习的，并从语料库中提取规则。基于规则的分词最简单直观的解决方案是基于字符、标点符号或空格。基于字符的分词会导致语言模型失去输入意义。尽管这种方法可以减少词汇量（这是优点），但它使模型很难通过字符 c，a 和 t 的编码来捕获 cat 的含义。此外，输入序列的维数变得非常大。同样，基于标点符号的模型无法正确处理某些表达式，如 haven't 或 ain't。

最近，一些高级子词分词算法（如字节对编码 BPE）已成为 Transformer 体系结构的一个组成部分。这些现代分词过程包括两个阶段：第一个阶段是预分词阶段，使用空

① 原著此处有误，主要是续行符(\)使用有误，此行中应该删除续行符。——译者注

间或语言相关规则简单地将输入拆分为标记；第二个阶段是分词训练阶段，训练分词器基于标记构建合理大小的基础词汇表。在训练自定义的分词器之前，首先加载一个经过预训练的分词器。以下代码从 Transformer 库加载一个 BertTokenizerFast 类型的土耳其语分词器，其词汇表大小为 32×10^3。

```
>>> from Transformers import AutoModel, AutoTokenizer
>>> tokenizerTUR = AutoTokenizer.from_pretrained(
                    "dbmdz/bert-base-turkish-uncased")
>>> print(f"VOC size is: {tokenizerTUR.vocab_size}")
>>> print(f"The model is: {type(tokenizerTUR)}")
VOC size is: 32000
The model is: Transformers.models.bert.tokenization_bert_fast.
BertTokenizerFast
```

以下代码为 bert-base-uncased 模型加载一个英文 BERT 标记器。

```
>>> from Transformers import AutoModel, AutoTokenizer
>>> tokenizerEN = \
AutoTokenizer.from_pretrained("bert-base-uncased")
>>> print(f"VOC size is: {tokenizerEN.vocab_size}")
>>> print(f"The model is {type(tokenizerEN)}")
VOC size is: 30522
The model is ... BertTokenizerFast
```

接下来看一看模型是如何工作的。使用以下两个分词器对单词 telecommunication 进行分词。

```
>>> word_en = "telecommunication"
>>> print(f"is in Turkish Model ? \
{word_en in tokenizerTUR.vocab}")
>>> print(f"is in English Model ? \
{word_en in tokenizerEN.vocab}")
is in Turkish Model ? False
is in English Model ? True
```

word_en 标记出现在英语分词器的词汇表中，但不在土耳其语分词器的词汇表中。接下来看看土耳其语分词器会发生什么。

```
>>> tokens = tokenizerTUR.tokenize(word_en)
>>> tokens
['tel', '##eco', '##mm', '##un', '##ica', '##tion']
```

由于土耳其语分词器模型的词汇表中没有这样一个词，所以需要将该单词分成对其有意义的部分。所有这些分割标记都已存储在模型词汇表中。请注意以下代码的输出。

```
>>> [t in tokenizerTUR.vocab for t in tokens]
[True, True, True, True, True, True]
```

使用已加载的英语分词器对相同的单词进行分词。

```
>>> tokenizerEN.tokenize(word_en)
['telecommunication']
```

由于英语模型的基本词汇表中包含 telecommunication 一词，所以不需要将其分解成多个部分，而是将其作为一个整体。通过从语料库中学习，分词器能够将一个词转换成语法逻辑上的大部分子成分。首先从土耳其语中找出一个很难的例子。作为一种粘着语言，土耳其语允许在词干上添加许多后缀来构造很长的单词。这是一篇文章中使用的最长的土耳其语单词之一。

```
Muvaffakiyetsizle§tiricile§tiriveremeyebileceklerimizdenmi§sinizcesine
```

这个单词的英文含义是 *As though you happen to have been from among those whom we will not be able to easily/quickly make a maker of unsuccessful ones*。土耳其语的 BERT 分词可能在训练中没有看到这个词，但它已经看到了其片段；*muvaffak*（*succesful*）是一个词干，可以形成*##iyet*（*successfulness*）、*##siz*（*unsuccessfulness*）、*##le§*（*become unsuccessful*）等。将结果与 Wikipedia 文章进行比较时，土耳其语分词器提取的组件在语法上似乎符合土耳其语言的逻辑。

```
>>> print(tokenizerTUR.tokenize(long_word_tur))
['muvaffak', '##iyet', '##siz', '##les', '##tir', '##ici',
'##les', '##tir', '##iver', '##emeye', '##bilecekleri', '##mi',
'##z', '##den', '##mis', '##siniz', '##cesine']
```

土耳其语分词器是 WordPiece 算法的一个例子，因为它与 BERT 模型一起工作。几乎所有的语言模型，包括 BERT、DistilBERT 和 ELECTRA，都需要一个 WordPiece 分词器。

接下来，将讨论 Transformer 使用的分词方法。首先，讨论广泛使用的字节对编码、WordPiece 和 SentencePiece 分词算法；然后，使用 Hugging Face 的快速分词库对这些分词算法进行训练。

3.6.1 字节对编码

字节对编码（Byte Pair Encoding，BPE）是一种数据压缩技术。字节对编码扫描数据序列，并用单个符号迭代替换最频繁的字节对。这个方法首次在文献“*Neural Machine Translation of Rare Words with Subword Units*（基于字词对稀有词实施神经机器翻译）”（Sennrich 等，2015 年）中提出，用来解决机器翻译中未知词和稀有词的问题。

目前，字节对编码技术已经成功应用于 GPT－2（Generative Pre－Training，生成式预训练）模型和许多其他最先进的模型中。许多现代分词算法都基于这种压缩技术。

字节对编码将文本表示为字符 n－gram 的序列，也称为字符级子词。训练首先从语料库中看到的所有 Unicode 字符（或者符号）的词汇表开始。这对于英语来说可能很小，但对于日语等字符丰富的语言来说可能很大。然后，迭代计算字符 bigrams，并使用特殊的新符号替换最频繁的字符 bigrams。例如，t 和 h 是频繁出现的符号。使用 th 符号替换连续符号。这个过程一直迭代运行，直到词汇表达到所需的词汇表大小。最常见的词汇量约为 3×10^4。

字节对编码在表示未知词方面特别有效。但是，字节对编码可能无法保证处理稀有词和包含稀有子词的词。在这种情况下，字节对编码会将稀有字符与一个特殊符号（<UNK>）关联，这可能导致单词丢失部分含义。作为一种潜在的解决方案，用户已经提出了字节级字节对编码（Byte－Level BPE，BBPE），它使用 256 字节的词汇集而不是 Unicode 字符来确保词汇表中包含每个基本字符。

3.6.2 WordPiece 分词算法

WordPiece 是另一种流行的分词算法，广泛用于 BERT、DistilBERT 和 ELECTRA。Schuster 和 Nakajima 于 2012 年提出了该方法，用于解决日韩语音问题。这项工作背后的动机是，尽管分词对英语来说不是一个大问题，但对于许多亚洲语言来说分词是重要的预处理过程，因为在这些语言中很少使用空白符。因此，在亚洲语言的自然语言处理研究中经常遇到分词方法。与字节对编码类似，WordPiece 使用大型语料库来学习词汇和合并规则。字节对编码和字节级字节对编码基于共现统计量学习合并规则，而 WordPiece 算法使用最大似然估计从语料库中提取合并规则。WordPiece 首先使用 Unicode 字符（也称为词汇符号）初始化词汇表。它将训练语料库中的每个单词视为一个符号列表（最初为 Unicode 字符），然后根据最大似然率而非频率迭代生成一个新符号，将所有可能的候选符号对中的两个符号合并。这种生产管道继续运行，直至达到所需的词汇表大小。

3.6.3 SentencePiece 分词算法

前面所述的分词算法将文本视为空格分隔的单词列表。基于空格的拆分在某些语言中不起作用。在德语中，复合名词没有空格，如 menschenrechte（人权）。解决方案是使用特定于语言的预分词器。在德语中，自然语言处理管道利用复合拆分器（compound－splitter）模块检查单词是否可以细分为较小的单词。但是，东亚语言（如汉语、日语、韩语和泰语）在单词之间不使用空格分隔。SentencePiece 算法旨在克服这种没有空格的限制，这是 Kudo 等在 2018 年提出的一种简单且独立于语言的分词算法。这种算法将输入视为原始输入流，其中空格是字符集的一部分。使用 SentencePiece 算法的标记器生成“_”字符，这也是在前面的 Albert 模型示例的输出中看到“_”的原因。使用 SentencePiece 算法的其他流行语言模型有 XLNet、Marian 和文本到文本传输 Trans former

(Text - to - Text Transfer Transformer，T5)。

到目前为止，本节讨论了子词分词方法。接下来，使用 tokenizers 库尝试进行训练。

3.6.4　tokenizers 库

读者可能已经注意到，在前面的代码示例中，Transformer 库中包含预训练的土耳其语和英语分词器。另外，Hugging Face 团队提供了独立于 Transformer 库的 tokenizers 库，以加快训练速度并提供给用户更多选项。tokenizers 库最初是使用 Rust 语言编写的，结果使多核并行计算成为可能，并使用 Python 语言进行封装。

为了安装 tokenizers 库，可以使用以下命令。

```
$ pip install tokenizers
```

tokenizers 库提供了几个组件，因此可以构建一个端到端的分词器（从预处理原始文本到解码标记化单元 ID）。分词管道的过程如下：Normalizer（规范化器）→PreTokenizer（预分词器）→Model Training（模型训练）→Post - processing（后处理）→Decoder（解码器）。

分词管道示意如图 3 - 17 所示。

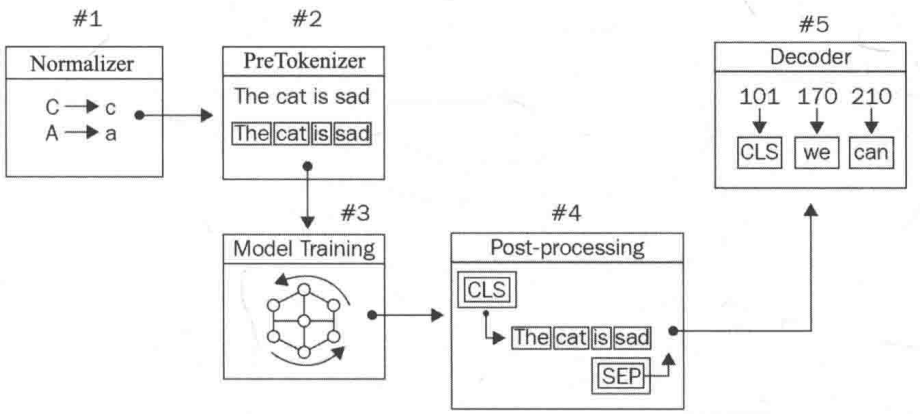


图 3 - 17　分词管道示意

(1) Normalizer（规范化器）：可以使用基本文本处理，如转换为小写、去空格、Unicode 规范化和删除重音符号。

(2) PreTokenizer（预分词器）：为下一个训练阶段准备语料库。预分词器根据规则（如空格）将输入拆分为标记。

(3) Model Training（模型训练）：一种子词分词算法（如字节对编码、字节级字节对编码和 WordPiece，前面已经讨论过）。模型训练发现子词/词汇表，并学习生成规则。

(4) Post - processing（后处理）：提供了与 Transformer 模型兼容（如 BertProcessors）的高级类构造。通常在馈入架构之前，向分词输入添加特殊的标记，如［CLS］和［SEP］标记。

（5）Decoder（解码器）：负责将标记 ID 转换回原始字符串。这只是为了检查正在发生的事情。

下面介绍如何训练 BPE 分词器和训练 WordPiece 模型。

1. 训练 BPE 分词器

下面使用莎士比亚的戏剧来训练 BPE 分词器。

（1）加载方式如下所示。

```
import nltk
from nltk.corpus import gutenberg
nltk.download('gutenberg')
nltk.download('punkt')
plays = ['shakespeare-macbeth.txt','shakespeare-hamlet.txt','shakespeare-
caesar.txt']
shakespeare = [" ".join(s) for ply in plays \
        for s in gutenberg.sents(ply)]
```

需要一个后处理器（TemplateProcessing）来处理前面的所有分词算法。需要定制后处理器，以方便特定的语言模型的输入。例如，下面的模板将适用于 BERT 模型，因为它需要在输入开始时嵌入［CLS］标记，在中间和末端嵌入［SEP］标记。

（2）模板的定义如下所示。

```
from tokenizers.processors import TemplateProcessing
special_tokens = ["[UNK]","[CLS]","[SEP]","[PAD]","[MASK]"]
temp_proc = TemplateProcessing(
      single = "[CLS] $A [SEP]",
      pair = "[CLS] $A [SEP] $B:1 [SEP]:1",
      special_tokens = [
        ("[CLS]", special_tokens.index("[CLS]")),
        ("[SEP]", special_tokens.index("[SEP]")),
      ],
)
```

（3）导入必要的组件以构建端到端分词管道。

```
from tokenizers import Tokenizer
from tokenizers.normalizers import \
    (Sequence,Lowercase, NFD, StripAccents)
from tokenizers.pre_tokenizers import Whitespace
from tokenizers.models import BPE
from tokenizers.decoders import BPEDecoder
```

（4）实例化 BPE 模型，代码如下所示。

```
tokenizer = Tokenizer(BPE())
```

(5) 预处理部分有两个组件：Normalizer（规范化器）和 Pre - Tokenizer（预分词器）。它可能有不止一个规范化器。因此，此处组成了规范化器组件的 Sequence（序列），该序列包括多个规范化器，其中，NFD( )是 Unicode 规范化器，StripeAccents( )用于删除重音。对于预分词，Whitespace( )会根据空格拆分文本。由于解码器组件必须与模型兼容，所以为 BPE 模型选择了 BPEDecoder( )。

```
tokenizer.normalizer = Sequence(
[NFD(),Lowercase(),StripAccents()])
tokenizer.pre_tokenizer = Whitespace()
tokenizer.decoder = BPEDecoder()
tokenizer.post_processor = temp_proc
```

(6) 至此已经准备好在数据上对 BPE 分词器进行训练。下面的代码片段对 BpeTrainer( )实例化，目的是通过设置超参数以帮助用户组织整个训练过程。将词汇大小参数设置为 5 000，因为莎士比亚语料库相对较小。对于大型的项目，会使用更大的语料库，通常将词汇大小设置为 30 000 左右。

```
>>> from tokenizers.trainers import BpeTrainer
>>> trainer = BpeTrainer(vocab_size = 5000,
                 special_tokens = special_tokens)
>>> tokenizer.train_from_iterator(shakespeare, trainer = trainer)
>>> print(f"Trained vocab size:{tokenizer.get_vocab_size()}" )
Trained vocab size: 5000
```

至此成功地完成了训练！

重要提示

从文件系统进行训练：为了开始训练过程，将内存中的 Shakespeare 对象作为字符串列表传递给 tokenizer. train_from_iterator( )。对于具有大型语料库的大型项目，需要设计一个 Python 生成器，主要使用文件系统中的文件而不是内存存储来生成字符串行信息。用户还应该检查 tokenizer. train( )，以从文件系统存储中进行训练。具体实现请参照上面的 BERT 训练部分的代码。

(7) 从戏剧 *Macbeth*（麦克白）中随机抽取一句话，命名为“sen”，并使用刚刚设计的新分词器进行分词。

```
>>> sen = "Is this a dagger which I see before me,\
the handle toward my hand?"
>>> sen_enc = tokenizer.encode(sen)
>>> print(f"Output: {format(sen_enc.tokens)}")
Output: ['[CLS]','is','this','a','dagger','which',
'i','see','before','me',',','the','hand','le',
'toward','my','hand','?','[SEP]']
```

（8）由于上面的后处理器功能，可以在适当的位置观察到额外的［CLS］和［SEP］标记。只有一个被拆分的词 handle（hand，le），因为给模型传递了一个来自 *Macbeth* 的句子，模型已经知道这个句子。此外，使用了一个小语料库，并且分词器不必使用压缩。接下来，传递一个富有挑战性的短语 Hugging Face，分词器可能不知道这个短语。

```
>>> sen_enc2 = tokenizer.encode("Macbeth and Hugging Face")
>>> print(f"Output: {format(sen_enc2.tokens)}")
Output: ['[CLS]','macbeth','and','hu','gg','ing','face','[SEP]']
```

（9）术语 Hugging 被转换为小写，并拆分为 hu，gg，ing 三部分，因为模型词汇表包含除 Hugging 以外的所有其他标记。接下来，再传递两个句子。

```
>>> two_enc = tokenizer.encode("I like Hugging Face!",
"He likes Macbeth!")
>>> print(f"Output: {format(two_enc.tokens)}")
Output: ['[CLS]','i','like','hu','gg','ing','face',
'!','[SEP]','he','li','kes','macbeth','!','[SEP]']
```

请注意，后处理器将［SEP］标记作为指示符注入。

（10）现在需要保存 BPE 模型。可以保存子单词分词模型，也可以保存整个分词管道。首先，仅保存 BPE 模型。

```
>>> tokenizer.model.save('.')
['./vocab.json','./merges.txt']
```

（11）BPE 模型保存了两个关于词汇表和合并规则的文件。“merges. txt”文件由 4 948 条合并规则组成。

```
$ wc -l ./merges.txt
4948 ./merges.txt
```

（12）排名前五的规则如下所示。其中，（t，h）是排名第一的规则，因为它是最常见的规则对。对于测试，BPE 模型扫描文本输入并尝试首先合并这两个符号（如果适用）。

```
$ head -3 ./merges.txt
t h
o u
a n
th e
r e
```

BPE 算法根据频率对规则进行排序。当用户在莎士比亚语料库中手动计算字符 bigrams 时，就会发现（t，h）是最常见的一对。

（13）现在，保存并加载整个分词管道。

```
>>> tokenizer.save("MyBPETokenizer.json")
>>> tokenizerFromFile = \
Tokenizer.from_file("MyBPETokenizer.json")
>>> sen_enc3 = \
tokenizerFromFile.encode("I like Hugging Face and
Macbeth")
>>> print(f"Output: {format(sen_enc3.tokens)}")
Output: ['[CLS]','i','like','hu','gg','ing','face',
'and','macbeth','[SEP]']
```

至此成功地重新加载了分词器！

2. 训练 WordPiece 模型

下面训练 WordPiece 模型。

（1）导入必要的模块。

```
from tokenizers.models import WordPiece
from tokenizers.decoders import WordPiece \
as WordPieceDecoder
from tokenizers.normalizers import BertNormalizer
```

（2）下面几行代码用于实例化一个空的 WordPiece 分词器，并为训练做好准备。BertNormalizer 是一个预定义的规范化器序列，包括清理文本、转换重音、处理汉字和转换为小写的过程。

```
tokenizer = Tokenizer(WordPiece())
tokenizer.normalizer = BertNormalizer()
tokenizer.pre_tokenizer = Whitespace()
tokenizer.decoder = WordPieceDecoder()
```

（3）现在，为 WordPiece( ) 实例化一个合适的训练器 WordPieceTrainer( ) 来组织训练过程。

```
>>> from tokenizers.trainers import WordPieceTrainer
>>> trainer = WordPieceTrainer(vocab_size=5000,\
special_tokens=["[UNK]", "[CLS]", "[SEP]",\
"[PAD]", "[MASK]"])
>>> tokenizer.train_from_iterator(shakespeare,
trainer=trainer)
>>> output = tokenizer.encode(sen)
>>> print(output.tokens)
['is','this','a','dagger','which','i','see',
'before','me',',','the','hand','##le','toward',
'my','hand','?']
```

（4）使用 WordPieceDecoder( ) 正确处理句子。

```
>>> tokenizer.decode(output.ids)
'is this a dagger which i see before me, the handle
toward my hand? '
```

（5）在输出中没有遇到任何［UNK］标记，因为分词器知道或者分割输入以进行编码。现在强制模型生成［UNK］标记，代码如下所示。它将一句土耳其语的句子传递给分词器。

```
>>> tokenizer.encode("Kralsın aslansın Macbeth!").tokens
['[UNK]', '[UNK]', 'macbeth', '! ']
```

以上代码得到了不错的结果！（得到了几个未知的标记）分词器从合并规则和基本词汇表中没有找到分解给定单词的方法。

到目前为止，我们设计了从规范化器组件到解码器组件的分词管道。另外，tokenizers 库提供了一个已经制作（未经训练）的空分词管道，其中包含适当的组件，可以构建用于生产的快速原型。以下是一些预先制作的分词器。

（1）CharBPETokenizer：原始 BPE。

（2）ByteLevelBPETokenizer：BPE 的字节级版本。

（3）SentencePieceBPETokenizer：与 SentencePiece 兼容的 BPE 实现。

（4）BertWordPieceTokenizer：著名的 BERT 分词器，使用 WordPiece。

以下代码用于导入这些管道。

```
>>> from tokenizers import (ByteLevelBPETokenizer,
        CharBPETokenizer,
        SentencePieceBPETokenizer,
        BertWordPieceTokenizer)
```

所有这些管道都已经设计完成。该过程的其余部分（如训练、保存模型和使用分词器）与之前的 BPE 和 WordPiece 训练过程相同。

恭喜读者取得了巨大的进步，并训练了第一个 Transformer 模型及其分词器。

3.7 本章小结

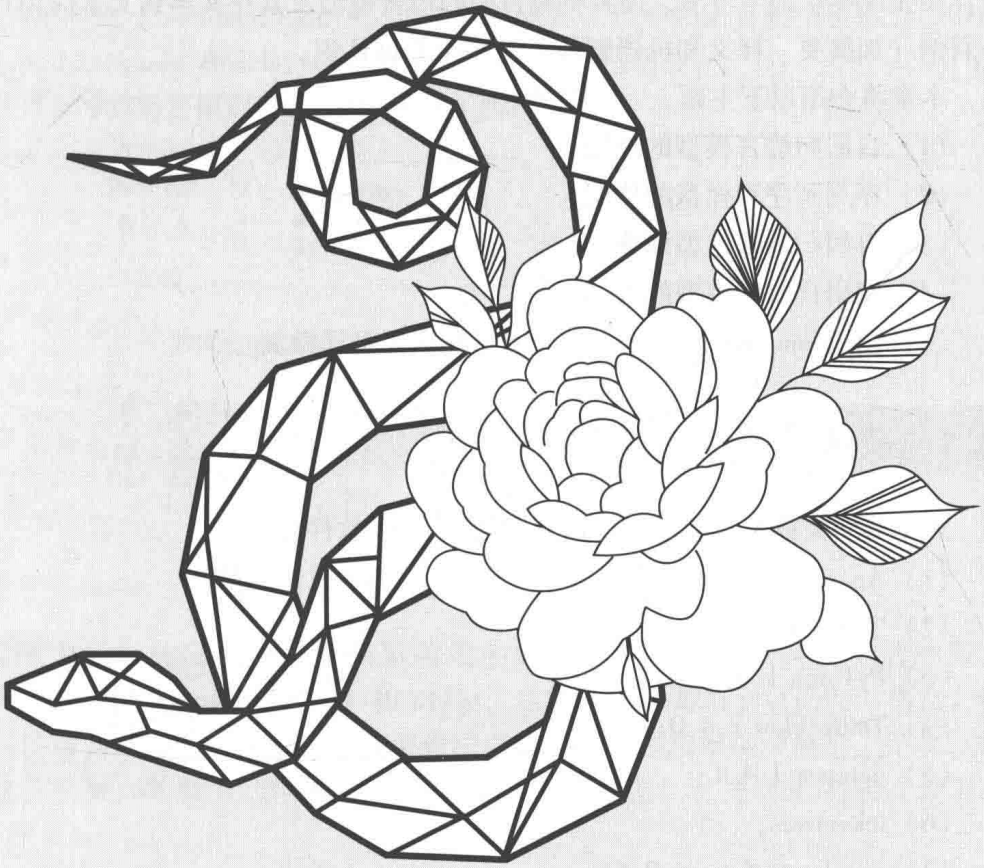
本章通过理论结合实践探索了自编码模型。首先讨论了 BERT 的基本知识，并从头开始训练了 BERT 模型以及相应的分词器，讨论了如何在其他框架内工作，如 Keras；除了 BERT，还回顾了其他自编码模型；为了避免过度的代码重复，没有提供其他模型训练的完整实现；在 BERT 训练过程中，训练了 WordPiece 分词算法；最后，研究了其他分词算法，因为这些算法都值得讨论和理解。

自编码模型使用原始 Transformer 的左侧解码器，并且大多针对分类问题进行了微调。在下一章中，将讨论和学习 Transformer 的右侧解码器部分，以实现语言生成模型。

4

第 4 章

自回归和其他语言模型



第 3 章详细介绍了自编码（Autoencoder，AE）语言模型，并研究了如何从零开始训练自编码语言模型。本章将讨论自回归（Autoregressive，AR）语言模型的理论细节，读者将学习如何在自己的语料库中预训练自回归语言模型，接着将学习如何在自己的文本上预训练任何语言模型，如 GPT－2 模型，并将其用于各种任务，如自然语言生成（Natural Language Generation，NLG），然后将学习 T5 模型的基础知识，并在自己的机器翻译（Machine Translation，MT）数据上训练多国语言版 T5（Multilingual T5，mT5）模型。在完成本章的学习后，读者将对自回归语言模型及其在文本到文本应用程序中的各种用例（如摘要、释义和机器翻译）有一个总体认识。

本章将介绍以下主题。

（1）自回归语言模型的使用。

（2）序列到序列模型的使用。

（3）自回归语言模型训练。

（4）使用自回归模型的 NLG。

（5）使用 simpletransformers 进行总结和机器翻译微调。

4.1 技术需求

为了成功完成本章的学习，需要安装以下库/软件包。

（1）Anaconda；

（2）Transformer 4.0.0；

（3）PyTorch 1.0.2；

（4）TensorFlow 2.4.0；

（5）datasets 1.4.1；

（6）tokenizers；

（7）simpletransformers 0.61。

可以通过以下 GitHub 链接获得本章中所有编码练习的 Jupyter Notebook：

https://github.com/PacktPublishing/Mastering－Transformers/tree/main/CH04。

4.2　使用自回归语言模型

Transformer 体系结构最初旨在对序列对序列的任务（如机器翻译或文档摘要）有效，但后来被用于各种自然语言处理问题（从标记分类到共指解析）。随后的工作开始分别更具创造性地使用 Transformer 体系结构的左、右部分。目标［也称为去噪目标（denoising objective）］是以双向方式从损坏的输入中完全恢复原始输入，如图 4－1（a）所示，稍后将详细介绍。从 BERT 体系结构中可以看出，这是自回归语言模型的一个显著示例，它们可以包含单词两侧的上下文。然而，第一个问题是，在预训练阶段使用的损坏的［MASK］符号在微调阶段的数据中不存在，从而导致预训练和微调差异。第二个问题是，BERT 模型假设掩码标记相互独立，而这是有争议的。

另外，自回归语言模型不依赖这种关于独立性的假设，因此不会自然地对预训练和微调差异产生影响，因为模型依赖以先前标记为条件预测下一个标记的目标，而不会掩蔽这些标记。自回归语言模型仅利用带掩码注意力机制的 Transformer 的解码部分。自回归语言模型阻止模型向前访问当前单词右侧的单词（或者向后访问当前单词左侧的单词），这称为单向性（unidirectionality）。由于其单向性，自回归语言模型也称为因果语言模型（Causal Language Models，CLM）。

自编码和自回归语言模型的对比如图 4－1 所示。

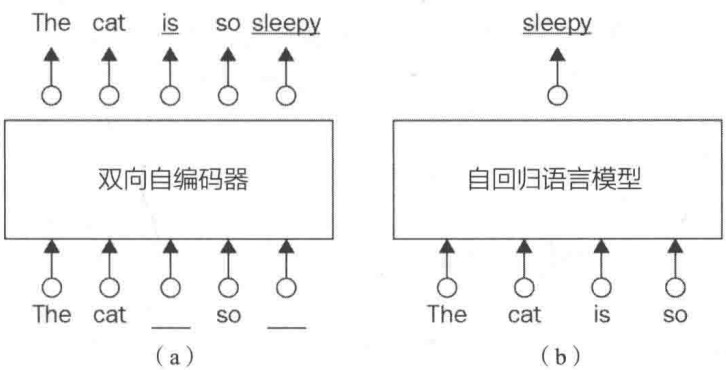


图 4－1　自编码和自回归语言模型的对比

相关文献中流行的自回归语言模型包括生成式预训练模型及其两个后续版本（GPT－2、GPT－3）、Transformer－XL 和 XLNet。尽管 XLNet 是基于自回归的模型，但在基于排列的语言目标的帮助下，该模型设法以双向方式利用单词两侧的上下文。接下来介绍这些模型，并展示如何通过各种实验来训练模型。首先讨论生成式预训练模型。

4.2.1　生成式预训练模型的介绍与训练

自回归语言模型由多个 Transformer 块组成。每个 Transformer 块包含一个带掩码的多

头自注意力层以及一个逐点前馈层。最后一个 Transformer 块中的激活被馈入 softmax( ) 函数，该函数生成整个词汇表中的单词概率分布，以预测下一个单词。

在生成式预训练的原始论文“*Improving Language Understanding by Generative Pre - Training*（通过生成式预训练提高对语言的理解）”（2018 年）中，作者解决了传统的基于机器学习的自然语言处理管道所面临的几个瓶颈问题。例如，首先，这些管道需要大量特定于任务的数据和体系结构。其次，在对预训练好模型的体系结构不做较大改动的情况下，很难应用任务感知的输入转换。最初的生成式预训练及其改进版本（GPT - 2 和 GPT - 3）由 OpenAI 团队设计，专注于缓解这些瓶颈的解决方案。原始生成式预训练研究的主要贡献在于，预训练的模型取得了令人满意的结果，不仅适用于单个任务，而且适用于多种任务。从未标记的数据中学习生成模型（称为无监督预训练）后，只需通过相对少量的特定于任务的数据［称为有监督的微调（supervised fine - tuning）］将模型微调到下游任务。这种“两阶段方案”广泛应用于其他 Transformer 模型中，其中无监督预训练之后是有监督微调。

为了提高生成式预训练体系结构的通用性，只需将输入转换为特定于任务的方式，而整个体系结构几乎保持不变。这种遍历式风格的方法根据任务将文本输入转换为有序序列，以便预训练的模型能够从中理解任务。图 4 - 2 的左侧部分说明了原始生成式预训练工作中使用的 Transformer 体系结构和训练目标；右侧部分显示了如何转换输入，以便在多个任务上进行微调。

简而言之，对于文本分类等单一序列任务，输入按照原样通过网络，线性层进行最后的激活以做出决策。对于文本蕴含等句子对任务，由两个序列组成的输入使用分隔符标记，如图 4 - 2 中的第二个示例所示。在这两种情况下，体系结构都认为统一的标记序列由预训练的模型处理。在这种转换中使用的分隔符有助于预训练的模型在文本蕴含的情况下，区分哪一部分是前提、哪一部分是假设。通过输入转换的方法，可以不必在整个任务中对结构进行实质性更改。

图 4 - 2 所示是输入转换的一种表示方式。

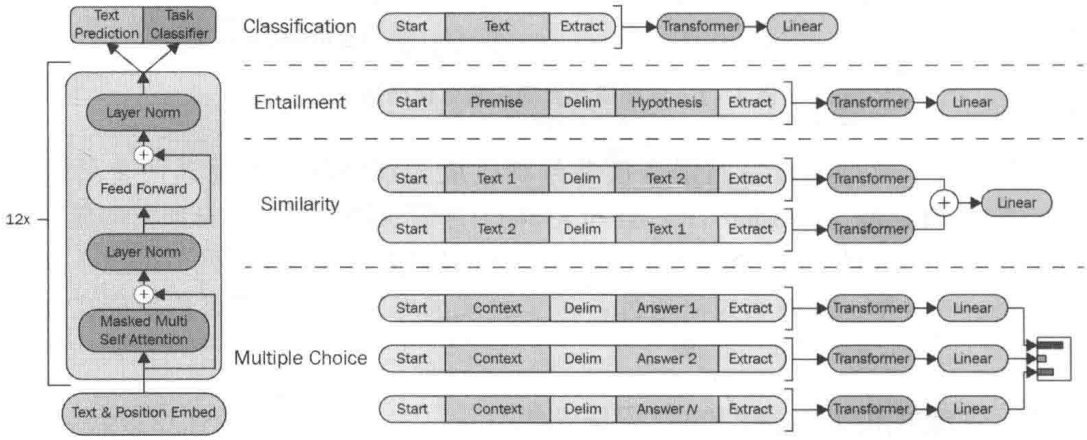


图 4 - 2　输入转换的一种表示方式（来源于原始论文）

生成式预训练模型及其两个改进版本大多专注于寻求特定的架构设计，不需要微调阶段。这些模型都基于这样一种理念，即模型可以非常熟练，可以在预训练阶段学习有关语言的大部分信息，而在微调阶段只剩下很少的工作。因此，微调过程可以在三个训练周期内完成，并且大多数任务的样例数相对较少。在极端情况下，零样本学习旨在禁用微调阶段。其基本思想是，模型可以在预训练期间学习有关该语言的大量信息。这对于所有基于 Transformer 的模型尤其如此。

下面介绍原始生成式预训练模型的改进版本。

GPT－2［具体请参见“*Language Models are Unsupervised Multitask Learners*（语言模型是无监督式多任务学习器）”）（2019 年）一文］是原 GPT－1 的改进版本，是一个比原 GPT－1 更大的模型，其使用的训练数据（称为 WebText）要多得多。在零样本设置下（没有应用微调，但在某些任务中取得了有限的成功），GPT－2 在 8 项任务中的 7 项任务中取得了最先进的结果。它在度量长期依赖性的较小数据集上取得了可比的结果。GPT－2 的作者认为，语言模型不一定需要明确的监督来学习任务。相反，当这些模型在一个庞大而多样的网页数据集上接受训练时，可以学习这些任务。GPT－2 被认为是一个通用系统，将原始生成式预训练中的学习目标 P（输出｜输入）替换为 P（输出｜输入，任务－i）。对于相同的输入，以特定任务为条件，模型产生不同的输出，即 GPT－2 通过训练相同的无监督模型来学习多个任务。一个单一的预训练模型仅通过学习目标来学习不同的能力。在其他研究中，在多任务和元任务设置中也可看到类似的公式。这种向多任务学习的转变使得为相同的输入执行许多不同的任务成为可能。但是，模型如何确定需要执行的任务呢？可以通过零样本任务迁移来实现这一目标。

与原始生成式预训练模型相比，GPT－2 没有特定于任务的微调，并且能够在零样本任务迁移设置下工作，其中所有下游任务都是预测条件概率的一部分。任务是在输入中以某种方式制定的，模型应该理解下游任务的性质并提供相应的答案。例如，从英语到土耳其语的机器翻译任务，不仅取决于输入，而且取决于任务。输入被安排为一个英语句子之后紧接着是一个土耳其语句子，并带有一个分隔符，模型根据分隔符来判断当前任务是英语到土耳其语的翻译。

OpenAI 团队使用 1 750 亿个参数训练了 GPT－3 模型［具体请参见“*Language models are few－shot Learners*（语言模型是少数样本学习器）”（2020 年）一文］，其参数的数量是 GPT－2 的 100 倍。GPT－2 和 GPT－3 的体系结构类似，主要区别通常在于模型大小和数据集的数量及质量。由于数据集中的数据量巨大，并且训练了大量的参数，因此无须任何基于梯度的微调，GPT－3 模型在零样本、1 样本和少数样本（$K=32$）设置下的许多下游任务中都取得了更好的效果。该团队声明，对于许多任务，包括机器翻译、问题回答系统和掩码标记的任务，GPT－3 模型的性能随着参数大小和样例数量的增加而提高。

4.2.2　Transformer－XL 模型

由于初始设计中缺乏重复性和上下文碎片，Transformer 模型受限于固定长度的上下

文，尽管这些模型能够学习长期依赖性。大多数 Transformer 模型将文档分成一个固定长度（大部分为 512）的段列表，其中不允许有任何跨段的信息流动。因此，语言模型无法捕获超出这个固定长度限制的长期依赖关系。此外，分段过程在不考虑句子边界的情况下构建分段。一个分段可以错误地由当前句子的后半部分和后一个句子的前半部分组合，因此语言模型在预测下一个标记时可能会错过必要的上下文信息。这个问题被研究者称为上下文碎片化（context fragmentation）问题。

为了解决和克服这些问题，Transformer - XL 模型的作者［具体情况请参见"*Transformer - XL: Attentive Language Models Beyond a Fixed - Length Context*（Transformer - XL：固定长度上下文之外的注意力语言模型）"（2019 年）一文］提出了一种新的 Transformer 体系结构，包括分段级别的循环机制和新的位置编码方案。这一方法启发了许多后续模型。Transformer - XL 模型不限于两个连续的分段，因为有效的上下文可以延伸到两个段之外。循环机制在每两个连续分段之间工作，导致在一定程度上跨越多个分段。模型可能涉及的最大依赖项长度受层数和分段长度的限制。

4.2.3 XLNet 模型

带掩码机制的语言建模（Masked Language Modeling，MLM）在基于 Transformer 体系结构的预训练阶段占主导地位。然而，由于带掩码的标记在预训练阶段存在，而在微调阶段不存在，这导致了预训练和微调之间的差异，所以带掩码机制的语言建模曾经受到许多非议。由于这种缺陷的存在，带掩码机制的语言模型可能无法使用在预训练阶段学习到的所有信息。XLNet［具体请参见"*XLNet: Generalized Autoregressive Pretraining for Language Understanding*（XLNet：通用自回归预训练的语言理解）"（2019 年）一文］采用带置换机制的语言建模（Permuted Language Modeling，PLM，又称为带排列机制的语言模型）取代带掩码机制的语言建模，这是一种输入标记的随机置换，以克服带掩码机制所带来的瓶颈。带置换机制的语言建模使每个标记位置利用所有位置的上下文信息，从而捕获双向的上下文信息。目标函数仅对因子分解的顺序进行排列，并定义标记预测的顺序，但不改变序列的自然位置。简而言之，该模型在排列后选择一些标记作为目标，并进一步尝试根据剩余标记和目标的自然位置进行预测。这使以双向方式使用自回归模型成为可能。

XLNet 同时利用了自编码模型和自回归模型。事实上，这是一个广义自回归模型。然而，正是归功于基于置换的语言建模方式，模型才可以处理来自左上下文和右上下文的标记。除了目标函数外，XLNet 还由两个重要机制组成：模型将 Transformer - XL 的分段级别的循环机制集成到其框架中，并且还包含了针对目标感知表示的双流注意力机制的精心设计。

下一节将讨论 Transformer 的另外两种模型的使用。

4.3 使用序列到序列模型

Transformer 的左侧编码器和右侧解码器部分通过交叉注意力连接在一起，这有助于

将每个解码器层的信息融入最终编码器层。这种方式自然会推动模型产生与原始输入密切相关的输出。作为原始 Transformer 的序列到序列（Seq2Seq）模型使用以下方案实现了这一点：

输入标记→嵌入→编码器→解码器→输出标记

Seq2Seq 模型保留 Transformer 的编码器和解码器部分。T5 模型、双向和自回归 Transformer（Bidirectional and Auto - Regressive Transformer，BART）模型，以及通过抽取间隙句的预训练生成式摘要序列模型（Pre - training with Extracted Gap - sentences for Abstractive Summarization Sequence - to - Sequence models，PEGASUS）都是流行的 Seq2Seq 模型。

4.3.1　T5 模型

大多数自然语言处理架构，从 Word2vec 到 Transformer，都是使用上下文（邻居）词来预测掩码以此来学习嵌入和其他参数。本节将自然语言处理问题视为单词预测问题。一些研究几乎将所有自然语言处理问题都归结为问题回答系统或标记分类。类似地，T5 模型［具体请参见论文“*Exploring the Limits of Transfer Learning with a Unified Text - to - Text Transformer*（探索使用统一的文本到文本 Transformer 框架进行迁移学习的局限性）”（2019 年）］提出了一个统一的框架，通过将许多任务转化为文本到文本的问题来解决问题。T5 的基本思想是将所有自然语言处理任务转换为文本到文本的问题，其中输入和输出都是一个标记列表，因为文本到文本的框架在将相同的模型应用于从问题回答系统到文本摘要的不同自然语言处理任务时十分有效。

图 4 - 3（来源于原论文）展示了 T5 如何在统一的框架内解决以下四个不同的自然语言处理问题：机器翻译、语言可接受性、语义相似性和文档摘要。

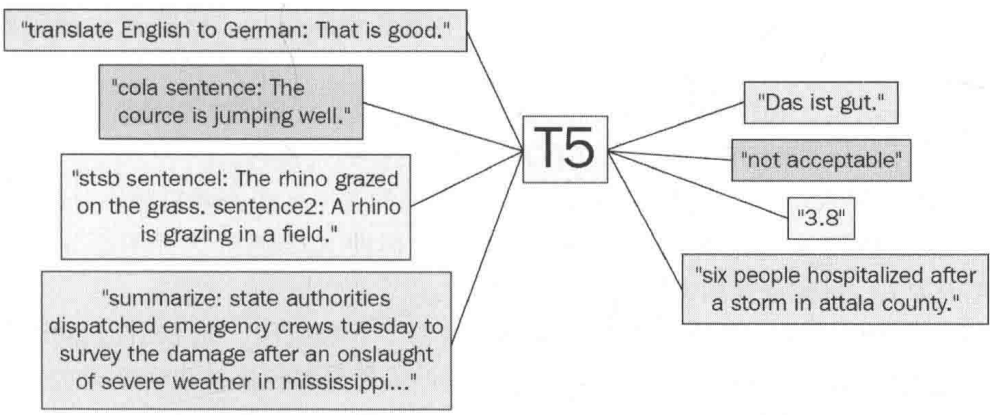


图 4 - 3　T5 体系结构

T5 模型大致遵循原始的编码器 - 解码器 Transformer 模型。修改是在层规范化和位置嵌入方案中完成的。T5 模型使用相对位置嵌入来代替正弦位置嵌入或学习嵌入，而相对位置嵌入在 Transformer 体系结构中越来越常见。T5 模型是一个单一的模型，可以处理多种任务，如语言生成。更重要的是，它将任务转换为文本到文本的格式。该模型

的输入是由任务前缀和附加到其上的输入所组成的文本。将带标签的文本数据集转换为{'inputs': '...', 'targets': ...'}格式，在此格式中将目标作为前缀插入输入。然后，使用带标记的数据训练模型，以使模型学会做什么以及如何做。在图 4－3 中，对于英语－德语的翻译任务，输入“translate English to German: That is good.”，将产生输出“Das ist gut.”。同样，带有“summarize:”前缀的输入将由模型汇总。

4.3.2 BART 概述

与 XLNet 一样，BART 模型［具体请参见论文“*BART: Denoising Sequence－to－Sequence Pre－training for Natural Language Generation, Translation, and Comprehension*（BART：自然语言生成、翻译和理解中对序列到序列的预训练去噪）”（2019 年）］利用了自编码和自回归模型的方案。BART 模型使用标准的 Seq2Seq 的 Transformer 体系结构，仅稍做修改。BART 是一个预训练好的模型，使用各种噪声干扰方法破坏文档。该研究对自然语言处理领域的主要贡献在于，它允许用户应用多种创造性的破坏方案，如图 4－4 所示。

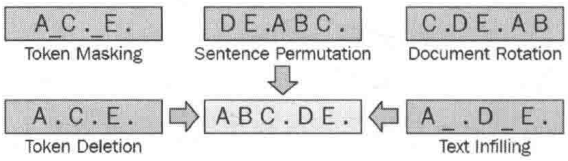


图 4－4 BART 图表（来源于原始 BART 论文）

接下来将讨论以下各个方案。

（1）标记掩码（Token Masking）：随机使用［MASK］符号屏蔽标记，与 BERT 模型相同。

（2）标记删除（Token Deletion）：从文档中随机删除标记。模型被迫确定删除的位置。

（3）文本填充（Text Infilling）：在 SpanBERT 之后，对多个文本跨距进行采样，然后将其替换为单个［MASK］标记。还存在［MASK］标记插入。

（4）句子排列（Sentence Permutation）：输入中的各个句子按随机顺序分割和排列。

（5）文档旋转（Document Rotation）：旋转文档使其以随机选择的标记开始，如图 4－4 中为 C。目标是找到文档的起始位置。

BART 模型可以通过多种方式进行微调，以用于下游应用，如 BERT。对于序列分类任务，将输入传输给编码器和解码器，解码器的最终隐藏状态被视为学习的表示。然后，可以使用一个简单的线性分类器进行预测。类似地，对于标记分类任务，整个文档作为输入被传输给编码器和解码器，最终解码器的最后状态是每个标记的表示。基于这些表示，可以解决标记分类问题，具体将在第 6 章中讨论。命名实体识别（Named－Entity Recognition，NER）和词性识别（Part－Of－Speech，POS）任务可以使用这种最终表示法来解决，其中 NER 可以识别文本中的实体，如人和组织；POS 将每个标记与其词汇类别（如名词、形容词等）关联。

对于序列生成，BART 模型的解码器块（自回归解码器）可以直接微调，以执行诸如抽象的问题回答系统或文档摘要之类的序列生成任务。BART 的作者（Lewis、Mike 等）使用两个标准文档摘要数据集（CNN/DailyMail 和 XSum）对模型进行了训练。BART 的作者还表明，可以同时使用编码器部分（处理源语言）和解码器部分（生成目标语言的单词，作为机器翻译的单个预训练解码器）。BART 模型使用新的随机初始化编码器替换编码器嵌入层，以便学习源语言中的单词。然后，模型以端到端的方式进行训练，训练新的编码器将外来词映射到输入中。模型还可以对目标语言进行去噪处理。新的编码器可以使用原始 BART 模型中的单独词汇表，包括外来语。

在 Hugging Face 平台中，可以使用以下代码行访问原始预训练的 BART 模型。

```
AutoModel.from_pretrained('facebook/bart-large')
```

当调用 Transformers 库的标准 summarization 管道时，将加载一个经过提炼的预训练 BART 模型。这个调用隐式加载“sshleifer/distilbart-cnn-12-6”模型及其相应的分词器。代码如下所示。

```
summarizer = pipeline("summarization")
```

下面的代码显式加载相同的模型及其相应的分词器。代码示例获取需要摘要信息的文本并输出结果。

```
from transformers import BartTokenizer, BartForConditionalGeneration,
BartConfig
from transformers import pipeline
model = \
BartForConditionalGeneration.from_pretrained('sshleifer/distilbart-cnn-
12-6')
tokenizer = BartTokenizer.from_pretrained('sshleifer/distilbart-cnn-12
-6')
nlp=pipeline("summarization", model=model, tokenizer=tokenizer)
text='''
We order two different types of jewelry from this
company the other jewelry we order is perfect.
However with this jewelry I have a few things I
don't like. The little Stone comes out of these
and customers are complaining and bringing them
back and we are having to put new jewelry in their
holes. You cannot sterilize these in an autoclave
as well because it heats up too much and the glue
does not hold up so the second group of these that
we used I did not sterilize them that way and the
stones still came out. When I use a dermal clamp
to put the top on the stones come out immediately.
DO not waste your money on this particular product
buy the three mm. that has the claws that hold the
```

```
jewelry in those are perfect. So now I'm stuck
with jewelry that I can't sell not good for
business.'''
q = nlp(text)
import pprint
pp = pprint.PrettyPrinter(indent = 0, width = 100)
pp.pprint(q[0]['summary_text'])
('The little Stone comes out of these little stones and
customers are complaining and bringing''them back and we are
having to put new jewelry in their holes . You cannot sterilize
these in an''autoclave because it heats up too much and the
glue does not hold up so the second group of''these that we
used I did not sterilize them that way and the stones still
came out .')
```

在下一节中，读者将动手实践，学习如何训练此类模型。

4.4 自回归语言模型训练

在本节中读者将学习如何训练自己的自回归语言模型。读者将从 GPT－2 开始，使用 Transformers 库深入了解其不同的训练函数。

可以找到任何特定的语料库来训练自己的 GPT－2，但在本示例中，使用了简·奥斯汀（Jane Austen）的 *Emma*（爱玛），这是一本浪漫小说。强烈建议在更大的语料库上进行训练，以得到更通用的语言生成模型。

在开始之前，请注意，使用了 TensorFlow 的原生训练功能，以演示所有 Hugging Face 模型都可以直接在 TensorFlow 或 PyTorch 上进行训练。请遵循以下操作步骤。

（1）使用以下命令下载小说 *Emma* 的原始文本。

```
wget https://raw.githubusercontent.com/teropa/nlp/master/
resources/corpora/gutenberg/austen-emma.txt
```

（2）在训练 GPT－2 的语料库上，训练用于 GPT－2 的 BytePairEncoding 分词器。以下代码将从 tokenizers 库导入 BPE 分词器。

```
from tokenizers.models import BPE
from tokenizers import Tokenizer
from tokenizers.decoders import ByteLevel as
ByteLevelDecoder
from tokenizers.normalizers import Sequence, Lowercase
from tokenizers.pre_tokenizers import ByteLevel
from tokenizers.trainers import BpeTrainer
```

（3）如读者所见，在本例中，计划通过添加更多功能［如 Lowercase（小写规范化）］来训练更高级的标记器。为了创建 tokenizer（分词器）对象，可以使用以下代码。

```
tokenizer = Tokenizer(BPE())
tokenizer.normalizer = Sequence([
        Lowercase()
])
tokenizer.pre_tokenizer = ByteLevel()
tokenizer.decoder = ByteLevelDecoder()
```

第一行代码从 BPE 分词器类创建一个分词器对象。对于规范化部分，添加了 Lowercase（小写字母），并且将 pre_tokenizer 属性设置为 ByteLevel()，以确保输入为字节。decoder 属性也必须设置为 ByteLevelDecoder()才能正确解码。

（4）使用 ByteLevel 提供的 50 000 个最大词汇量和初始字母表，对分词器进行训练。代码如下所示。

```
trainer = BpeTrainer(vocab_size=50000, inital_
alphabet=ByteLevel.alphabet(), special_tokens=[
            "<s>",
            "<pad>",
            "</s>",
            "<unk>",
            "<mask>"
          ])
tokenizer.train(["austen-emma.txt"], trainer)
```

（5）必须添加需要考虑的特殊标记。为了保存分词器对象，需要创建一个目录。代码如下所示。

```
!mkdir tokenizer_gpt
```

（6）可以运行以下命令来保存分词器对象。

```
tokenizer.save("tokenizer_gpt/tokenizer.json")
```

（7）保存分词器对象之后，可以使用保存好的分词器对语料库进行预处理，从而为 GPT-2 训练做好准备，但首先不能忘记导入相关的库。可以参照以下代码执行相关的导入操作。

```
from transformers import GPT2TokenizerFast, GPT2Config, TFGPT2LMHeadModel
```

（8）使用 GPT2TokenizerFast 加载分词器。代码如下所示。

```
tokenizer_gpt = GPT2TokenizerFast.from_pretrained("tokenizer_gpt")
```

（9）必须添加带有标记的特殊标记。代码如下所示。

```
tokenizer_gpt.add_special_tokens({
    "eos_token": "</s>",
    "bos_token": "<s>",
    "unk_token": "<unk>",
    "pad_token": "<pad>",
    "mask_token": "<mask>"
})
```

（10）还可以运行以下代码来再次检查是否一切正常。

```
tokenizer_gpt.eos_token_id
>> 2
```

此代码将输出句末（End - of - Sentence，EOS）标记的标识符（ID），对于当前分词器，其 EOS 标识为 2。

（11）还可以执行以下代码来测试句子。

```
tokenizer_gpt.encode("<s> this is </s>")
>> [0, 265, 157, 56, 2]
```

对于这个输出结果，0 表示句子的开头，265，157 和 56 与句子本身相关，EOS 标记为 2，即 </s>。

（12）创建 config 对象时必须使用这些配置。以下代码将创建 config 对象和 GPT - 2 模型的 TensorFlow 版本。

```
config = GPT2Config(
    vocab_size = tokenizer_gpt.vocab_size,
    bos_token_id = tokenizer_gpt.bos_token_id,
    eos_token_id = tokenizer_gpt.eos_token_id
)
model = TFGPT2LMHeadModel(config)
```

（13）在运行 config 对象时，可以看到字典格式的配置。代码如下所示。

```
config
>> GPT2Config { "activation_function": "gelu_new",
"attn_pdrop": 0.1, "bos_token_id": 0, "embd_pdrop":
0.1, "eos_token_id": 2, "gradient_checkpointing":
false, "initializer_range": 0.02, "layer_norm_
epsilon": 1e-05, "model_type": "gpt2", "n_ctx": 1024,
"n_embd": 768, "n_head": 12, "n_inner": null, "n_
layer": 12, "n_positions": 1024, "resid_pdrop": 0.1,
"summary_activation": null, "summary_first_dropout":
```

```
0.1, "summary_proj_to_labels": true, "summary_type":
"cls_index", "summary_use_proj": true, "transformers_
version": "4.3.2", "use_cache": true, "vocab_size":
11750}
```

如读者所见，其他设置未被修改。有趣的是，vocab_size 设置为 11 750。其背后的原因是此处将最大词汇量设置为 50 000，但是语料库的词汇量较少，其 BPE 标记创建了 11 750。

（14）现在，可以准备好语料库进行预训练。代码如下所示。

```
with open("austen-emma.txt", "r", encoding='utf-8') as f:
    content = f.readlines()
```

（15）变量 content 现在将包括原始文件中的所有原始文本，但需要删除每行后的换行符 '\n'，并删除那些少于 10 个字符的行。代码如下所示。

```
content_p = []
for c in content:
    if len(c) >10:
        content_p.append(c.strip())
content_p = " ".join(content_p) +tokenizer_gpt.eos_token
```

（16）删除短句将确保模型在长序列上进行训练，以便能够生成更长的序列。在前面代码段的末尾，content_p 的内容为连接后的原始文件，并将 eos_token 添加到末尾。用户也可以遵循不同的策略。例如，可以通过在每行中添加 </s> 来分隔每一行，这将有助于模型识别句子何时结束。然而，此处打算让模型在没有遇到 EOS 的情况下可以处理更长的序列。下面的代码片段演示了其实现。

```
tokenized_content = tokenizer_gpt.encode(content_p)
```

上述代码片段中的 GPT 分词器将标记整个文本，并使其成为一个完整的、长的标记 ID 序列。

（17）现在，可以创建训练样本了。代码如下所示。

```
sample_len = 100
examples = []
for i in range(0, len(tokenized_content)):
    examples.append(tokenized_content[i:i + sample_len])
```

（18）上面的代码使 examples 的大小为 100，每个样例从文本的给定部分开始，然后在 100 个标记后的位置结束。

```
train_data = []
labels = []
```

```
for example in examples:
    train_data.append(example[:-1])
    labels.append(example[1:])
```

在 train_data 中，包含一个大小为 99 的序列，标记从最开始到第 99 个，labels 标签将包含一个从 1 到 100 的标记序列。

（19）为了更快地进行训练，需要把数据转换为 TensorFlow 数据集的形式。代码如下所示。

```
import① tensorflow as tf
buffer = 500
batch_size = 16
dataset = tf.data.Dataset.from_tensor_slices((train_data, labels))
dataset = dataset.shuffle(buffer).batch(batch_size, drop_remainder=True)
```

在上面的代码中，buffer 用于混排数据的缓冲区大小，batch_size 用于训练的批次大小，drop_remainder 用于删除余数（如果小于 16）。

（20）现在，用户可以指定 optimizer（优化器）、loss（损失函数）和 metrics（度量指标）属性。代码如下所示。

```
optimizer = tf.keras.optimizers.Adam(learning_rate=3e-5,
epsilon=1e-08, clipnorm=1.0)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_
logits=True)
metric = tf.keras.metrics.
SparseCategoricalAccuracy('accuracy')
model.compile(optimizer=optimizer, loss=[loss, *[None] *
model.config.n_layer], metrics=[metric])
```

（21）模型已经编译好，可以按照用户希望的学习迭代次数进行训练。代码如下所示。

```
epochs = 10
model.fit(dataset, epochs=epochs)
```

读者将看到图 4-5 所示的输出结果。

接下来，讨论使用自回归模型的自然语言生成。现在已经保存了模型，下一节将使用该模型生成句子。

到目前为止，读者了解了如何为自然语言生成训练自己的模型。下一节将描述如何利用自然语言生成模型生成语言。

---

① 原著此处有误，Python 代码中 import 必须小写。——译者注

```
Epoch 1/10
WARNING:tensorflow:The parameters `output_attentions`, `output_hidden_states` and `use_cache` cannot be updated wh
en calling a model.They have to be set to True/False in the config object (i.e.: `config=XConfig.from_pretrained('
name', output_attentions=True)`).
WARNING:tensorflow:The parameter `return_dict` cannot be set in graph mode and will always be set to `True`.
WARNING:tensorflow:The parameters `output_attentions`, `output_hidden_states` and `use_cache` cannot be updated wh
en calling a model.They have to be set to True/False in the config object (i.e.: `config=XConfig.from_pretrained('
name', output_attentions=True)`).
WARNING:tensorflow:The parameter `return_dict` cannot be set in graph mode and will always be set to `True`.
166/166 [==============================] - 421s 3s/step - loss: 5.8450 - logits_loss: 5.8450 - logits_accuracy: 0.
1649 - past_key_values_1_accuracy: 0.0025 - past_key_values_2_accuracy: 0.0021 - past_key_values_3_accuracy: 0.001
7 - past_key_values_4_accuracy: 0.0031 - past_key_values_5_accuracy: 0.0024 - past_key_values_6_accuracy: 0.0026 -
past_key_values_7_accuracy: 0.0029 - past_key_values_8_accuracy: 0.0030 - past_key_values_9_accuracy: 0.0032 - pas
t_key_values_10_accuracy: 0.0028 - past_key_values_11_accuracy: 0.0015 - past_key_values_12_accuracy: 0.0039
Epoch 2/10
166/166 [==============================] - 421s 3s/step - loss: 2.6242 - logits_loss: 2.6242 - logits_accuracy: 0.
5361 - past_key_values_1_accuracy: 0.0025 - past_key_values_2_accuracy: 0.0023 - past_key_values_3_accuracy: 0.002
3 - past_key_values_4_accuracy: 0.0025 - past_key_values_5_accuracy: 0.0025 - past_key_values_6_accuracy: 0.0025 -
past_key_values_7_accuracy: 0.0024 - past_key_values_8_accuracy: 0.0024 - past_key_values_9_accuracy: 0.0027 - pas
t_key_values_10_accuracy: 0.0027 - past_key_values_11_accuracy: 0.0025 - past_key_values_12_accuracy: 0.0025
Epoch 3/10
 42/166 [======>.......................] - ETA: 5:07 - loss: 1.4982 - logits_loss: 1.4982 - logits_accuracy: 0.760
7 - past_key_values_1_accuracy: 0.0027 - past_key_values_2_accuracy: 0.0023 - past_key_values_3_accuracy: 0.0022 -
past_key_values_4_accuracy: 0.0026 - past_key_values_5_accuracy: 0.0025 - past_key_values_6_accuracy: 0.0023 - pas
t_key_values_7_accuracy: 0.0024 - past_key_values_8_accuracy: 0.0023 - past_key_values_9_accuracy: 0.0027 - past_k
ey_values_10_accuracy: 0.0027 - past_key_values_11_accuracy: 0.0025 - past_key_values_12_accuracy: 0.0023
```

图 4－5　使用 TensorFlow/Keras 训练 GPT－2

4.5　使用自回归模型的自然语言生成

在上一节中，学习了如何在自己的语料库上训练自回归模型。因此，用户已经训练了自己的 GPT－2 版本。但是，还缺少以下问题的答案：应该如何使用训练好的模型呢？为了回答这个问题，可以按照以下步骤进行实践。

（1）从刚刚训练的模型开始生成句子。代码如下所示。

```
def generate(start, model):
    input_token_ids = tokenizer_gpt.encode(start, return_tensors ='tf')
    output = model.generate(
        input_token_ids,
        max_length = 500,
        num_beams = 5,
        temperature = 0.7,
        no_repeat_ngram_size =2,
        num_return_sequences =1
    )
    return tokenizer_gpt.decode(output[0])
```

上述代码片段中定义的 generate( ) 函数接收一个字符串 start，并在该字符串之后生成序列。可以更改参数。例如，将 max_length 设置为较小的序列大小，或者将 num_return_sequences 设置为具有不同的代。

（2）可以使用一个空字符串进行测试。代码如下所示。

```
generate(" ", model)
```

输出结果如图 4－6 所示。

```
'  it was a nervous; and emma could not but it, and made it necessary to be cheerful. his spirits required with th
em; hating change of every kind.  he was by no means yet reconciled to his own daughter\'s marrying, was always di
sagreeable; but with compassion, as the origin of affection, though it had been entirely a mile from his habits of
being never able to part with miss taylor been a great must be accepted in herself as for herself; fond of gentle
selfishness, nor could ever speak of her own, only half a long october and from isabella\'s being now obliged to s
uppose that great deal happier if she was very much beyond her father and he could feel differently from such a go
od was now long ago for him from himself to have had done as sad a thing for her life at hartfield. emma was much
older man in their little too long evening, when and would not have been living together as her friend and a very
early from them, they had spent all the house of having been supplied by any means rank as cheerfully as friend wa
s her but emma smiled and bear the rest of emma had ceased to say her sister\'s marriage, "how she had died too mu
ch disposed to keep him not to think a little children of authority being settled in london was some satisfaction
in great danger of great comfort and chatted as large and with what a house from five years had said at any time.
weston was more than an excellent woman as he had many a man, the want of the with her temper had her advantages,
but little way and her daily but particularly was no companion for even half so unperceived, who had taught and wi
sh she dearly loved her through the actual disparity in mr. woodhouse had such an affection; you have never any re
straint as governess than any odd humours, it is such thoughts; highly her!" "i cannot that other-and you know fro
m a large.--and how was used to see her in affection for having great house; these were first with all his heart a
nd had hardly allowed her pleasant society again. how nursed her many now in consequence of a much have recommende
d him at this is the friendliness of sixteen years old and friend very mutually attached, being left to impose any
disagreeable consciousness were here to sit and november evening must go in the next visit from intellectual such
as few a match of both daughters, very good-people have more the advantage of his life in ways than a melancholy c
hange than her as great with you would be felt every body that her place had miss'
```

图 4－6　GPT－2 文本生成示例

从图 4－6 所示的输出结果可以看到生成了一个长文本，即使文本的语义不是很令人满意，但在许多情况下语法基本上是正确的。

（3）尝试不同的开始字符串，将 max_length 设置为较低的值（如 30）。代码如下所示。

```
generate("wetson was very good")
>> 'wetson was very good; but it, that he was a great
must be a mile from them, and a miss taylor in the house;'
```

这里的 wetson 是小说中的人物之一。

（4）可以使用以下代码保存模型，使该模型可以用于发布或者用于不同的应用程序。

```
model.save_pretrained("my_gpt-2/")
```

（5）为了验证模型是否已经正确保存，接着可以尝试加载模型。代码如下所示。

```
model_reloaded = TFGPT2LMHeadModel.from_pretrained("my_gpt-2/")
```

保存了两个文件——config. json 和 tf－model. h5，它们是 TensorFlow 版本的文件，如图 4－7 所示。

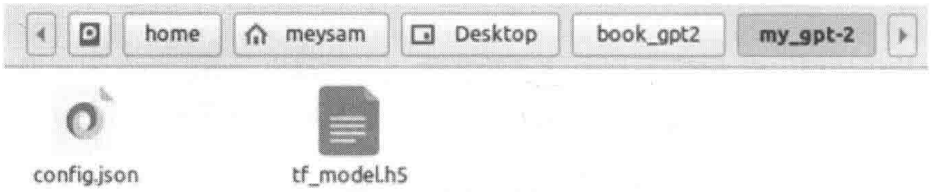


图 4－7　语言模型的 save_pretrained( ) 方法的输出结果

（6）Hugging Face 还包含一个必须遵守的文件命名标准。通过以下导入语句，可以使用这些标准文件名。

```
from transformers import WEIGHTS_NAME, CONFIG_NAME, TF2_WEIGHTS_NAME
```

然而，当使用 save_pretrained( )方法时，不需要指定文件名，只需指定目录。

（7）如前面章节所述，Hugging Face 还包含 AutoModel 和 AutoTokenizer 类。也可以使用该功能保存模型，但在此之前，仍然需要手动完成一些配置。第一件事是将分词器保存为 AutoTokenizer 所使用的正确格式。可以使用 save_pretrained( )方法执行此操作。代码如下所示。

```
tokenizer_gpt.save_pretrained("tokenizer_gpt_auto/")
```

输出结果如图 4 - 8 所示。

```
('tokenizer_gpt_auto/tokenizer_config.json',
 'tokenizer_gpt_auto/special_tokens_map.json',
 'tokenizer_gpt_auto/vocab.json',
 'tokenizer_gpt_auto/merges.txt',
 'tokenizer_gpt_auto/added_tokens.json')
```

图 4 - 8　分词器对象的 save_pretrained( )方法的输出结果

（8）文件列表显示在指定的目录中，但必须手动更改 tokenizer_config 才能使用。首先，应该将其重命名为“config. json”；其次，应该添加一个 JavaScript 对象表示法（JavaScript Object Notation，JSON）格式的属性，指示 model_type 属性为 gpt2。代码如下所示。

```
{"model_type":"gpt2",
...
}
```

（9）现在，一切都准备就绪，只需使用以下两行代码就可以加载模型和分词器。

```
model = AutoModel.from_pretrained("my_gpt-2/", from_tf=True)
tokenizer = AutoTokenizer.from_pretrained("tokenizer_gpt_auto")
```

但是，不要忘记将 from_tf 设置为 True，因为此处的模型是以 TensorFlow 格式保存的。

到目前为止，读者学习了如何使用 TensorFlow、Transformers 预训练和保存自己的文本生成模型，还学习了如何保存预训练的模型，并将其准备用作自动模型。在下一节中，读者将学习使用其他模型的基础知识。

4.6　使用 simpletransformers 进行总结和机器翻译微调

到目前为止，读者学习了训练语言模型的基础知识和高级方法，但是从零开始训练自己的语言模型并不总是可行的，因为有时会遇到诸如计算能力不足之类的障碍。在本

节中，读者将了解如何针对机器翻译和文档摘要的特定任务，在自己的数据集上微调语言模型。请按照以下步骤依次执行。

（1）安装 simpletransformers 库。代码如下所示。

```
pip install simpletransformers
```

（2）下载包含平行语料库的数据集。这个平行语料库可以是任何类型的 Seq2Seq 任务。对于本例，使用机器翻译示例，但用户还可以使用任何其他数据集执行其他任务，如释义、文档摘要等，甚至可以将文本转换为结构化查询语言（Structured Query Language，SQL）。

（3）下载并解压缩数据后，有必要为列标题添加 EN 和 TR，以方便使用。可以使用 pandas 加载数据集。代码如下所示。

```
import pandas as pd
df = pd.read_csv("TR2EN.txt",sep="\t").astype(str)
```

（4）需要向数据集添加特定于 T5 的命令，以使其了解正在处理的命令。可以使用以下代码执行这个操作。

```
data = []
for item in digitrons():
    data.append(["translate english to turkish", item[1].EN, item[1].TR])
```

（5）调整 DataFrame。代码如下所示。

```
df = pd.DataFrame(data, columns=["prefix", "input_text", "target_text"])
```

输出结果如图 4-9 所示。

（6）运行以下代码以导入所需的类。

```
from simpletransformers.t5 import T5Model, T5Args
```

（7）使用以下代码定义所训练的参数。

```
model_args = T5Args()
model_args.max_seq_length = 96
model_args.train_batch_size = 20
model_args.eval_batch_size = 20
model_args.num_train_epochs = 1
model_args.evaluate_during_training = True
model_args.evaluate_during_training_steps = 30000
model_args.use_multiprocessing = False
model_args.fp16 = False
model_args.save_steps = -1
```

	prefix	input_text	target_text
0	translate english to turkish	Hi.	Merhaba.
1	translate english to turkish	Hi.	Selam.
2	translate english to turkish	Run!	Kaç!
3	translate english to turkish	Run!	Koş!
4	translate english to turkish	Run.	Kaç!
5	translate english to turkish	Run.	Koş!
6	translate english to turkish	Who?	Kim?
7	translate english to turkish	Fire!	Ateş!
8	translate english to turkish	Fire!	Yangın!
9	translate english to turkish	Help!	Yardım et!
10	translate english to turkish	Jump.	Defol.
11	translate english to turkish	Stop!	Dur!
12	translate english to turkish	Stop!	Bırak!
13	translate english to turkish	Wait.	Bekle.

图 4－9　英语－土耳其语机器翻译平行语料库

```
model_args.save_eval_checkpoints = False
model_args.no_cache = True
model_args.reprocess_input_data = True
model_args.overwrite_output_dir = True
model_args.preprocess_inputs = False
model_args.num_return_sequences = 1
model_args.wandb_project = "MT5 English-TurkishTranslation"
```

（8）可以加载需要微调的模型。这是本文所选择的一个模型。

```
model = T5Model("mt5", "google/mt5-small", args=model_args, use_cuda=
False)
```

如果没有足够的计算统一设备体系结构（Compute Unified Device Architecture，CUDA）内存用于 mT5，请不要忘记将 use_cuda 设置为 False。

（9）可以使用以下代码拆分训练和评估 DataFrames。

```
train_df = df[:470000]
eval_df = df[470000:]
```

（10）使用以下代码开始训练。

```
model.train_model(train_df, eval_data=eval_df)
```

mT5 模型评估结果如图 4－10 所示。

```
(3,
 {'global_step': [3],
  'eval_loss': [28.536166508992512],
  'train_loss': [33.57326889038086]})
```

图 4－10　mT5 模型评估结果

该结果表示评估损失和训练损失。

（11）用户只需通过以下代码，就可以加载并使用模型。

```
model_args = T5Args()
model_args.max_length = 512
model_args.length_penalty = 1
model_args.num_beams = 10
model = T5Model("mt5", "outputs", args=model_args, use_cuda=False)
```

现在可以使用 model_predict( ) 函数将英语翻译成土耳其语。

simpletransformers 库可以使训练许多模型（从序列标签到 Seq2Seq 模型）变得非常容易并且实用。

至此，读者学习了如何训练自己的自回归模型，本章到此结束。

4.7　本章小结

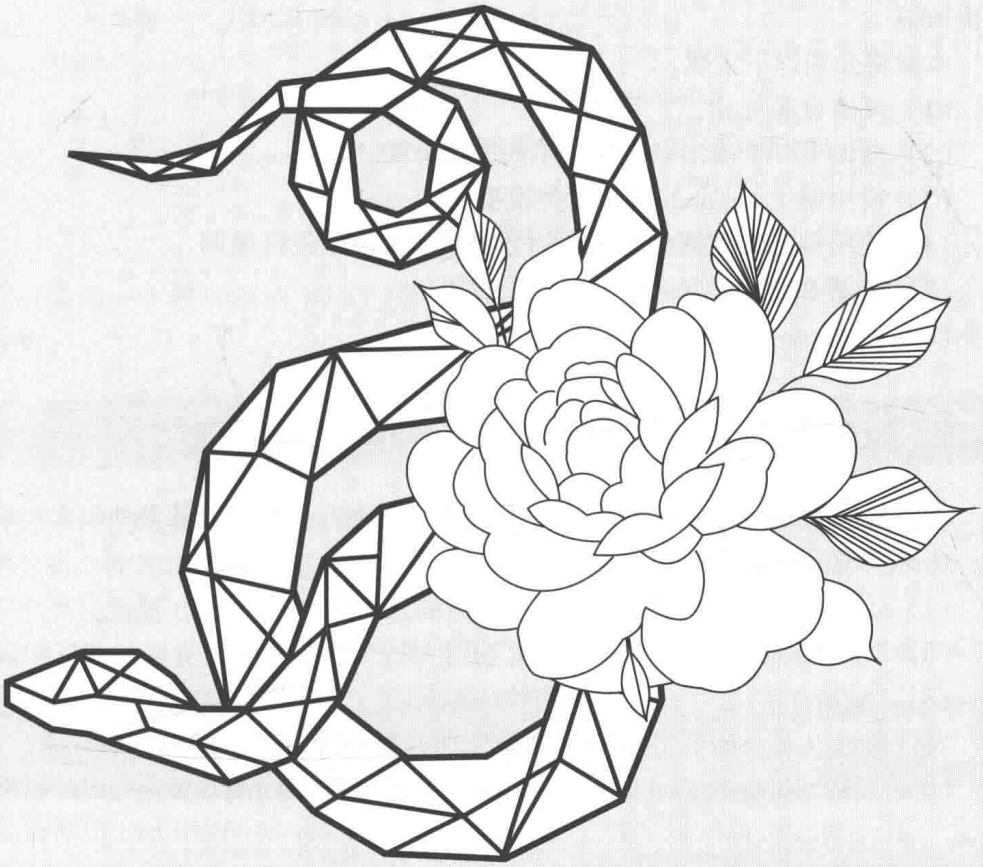
在本章中，读者学习了自回归语言模型各个方面的内容——从预训练到微调；通过训练生成式语言模型和类似机器翻译任务微调，来观察这些模型的最佳特性；了解了更复杂模型（如 T5）的基础知识，并使用这种模型执行机器翻译；使用了 simpletransformers 库；在自己的语料库上训练 GPT－2 模型，并使用该模型生成文本；学习了如何保存训练好的模型，并将其与 AutoModel 一起使用；深入了解了如何使用 tokenizers 库训练和使用字节对编码。

下一章将讨论如何微调文本分类语言模型。

5

第 5 章

微调文本分类语言模型



在本章中，读者将学习如何为文本分类配置预训练的模型，以及如何对其进行微调，以适应文本分类下游任务（如情感分析或多类别分类）。本章还将讨论如何通过一个具体的实现来处理句子对和回归问题。读者将使用著名的数据集，如 GLUE，以及本文的自定义数据集，然后利用 Trainer 类对训练和微调过程的复杂性进行处理。

首先，学习如何通过 Trainer 类微调单句二元情感分类；然后，在不使用 Trainer 类的情况下，使用原生 PyTorch 进行情绪分类训练，在多类别分类中，考虑两个以上的类别，执行对 7 个类进行分类的微调任务；最后，训练一个文本回归模型，使用句子对来预测数值。

本章将介绍以下主题。

（1）文本分类导论。

（2）微调 BERT 模型以适用于单句二元分类。

（3）使用原生 PyTorch 训练分类模型。

（4）使用自定义数据集对多类别分类 BERT 模型进行微调。

（5）微调 BERT 模型以适用于句子对回归。

（6）使用 run_glue. py 对模型进行微调。

5.1 技术需求

本章将使用 Jupyter Notebook 进行编码练习。为此，要求安装 Python 3.6 或以上的版本，还必须确保安装以下软件包。

（1）sklearn；

（2）Transformer（不低于 4.0.0 版本）；

（3）Datasets。

可以通过以下 GitHub 链接获得本章中所有编码练习的 Jupyter Notebook：

https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH05。

5.2 文本分类导论

文本分类（也称为文本归类）是一种将文档（句子、Twitter 帖子、图书章节、电

子邮件内容等）映射到预定义列表（类）中类别的方法。对于两个有正负标签的类别，本文称之为二元分类（binary classification），更具体地说，称之为情感分析（sentiment analysis）。对于两个以上的类，本文称之为多类别分类（multi - class classification），其中各个类别之间是互斥的；或者称之为多标签分类（multi - label classification），其中各个类别之间不是互斥的，这意味着一个文档可以接收多个标签。例如，一篇新闻文章的内容可能同时与体育和政治有关。除了这些分类方法外，可能希望对文档进行［-1，1］范围内的评分，或者对文档进行［1，5］范围内的排名。可以使用回归模型来解决这类问题，其中输出的类型是数值，而不是类别。

幸运的是，用户可以使用 Transformer 体系结构有效地解决这些问题。对于句子对任务（如文档相似性或文本蕴含），输入不是一个句子，而是两个句子，如图 5 - 1 所示。可以对两个句子的语义相似程度进行评分，也可以预测这两个句子是否在语义上相似。另一个句子对任务是文本蕴含（textual entailment），其中问题被定义为多类别分类。在这里，GLUE 基准测试（entail/contradict/neutral）中使用了图 5 - 1 所示的两个序列。

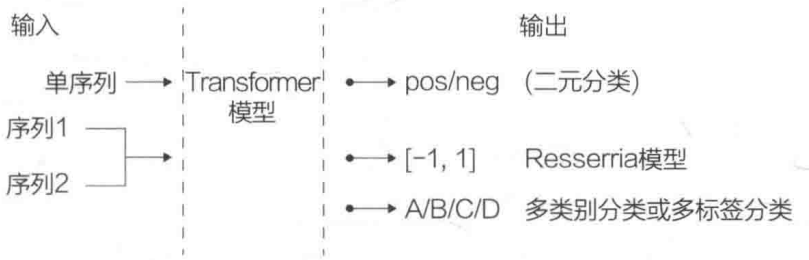


图 5 - 1　文本分类方案

本章通过微调一个预训练的 BERT 模型来开始训练过程，该模型针对一个常见问题：情感分析。

5.3　微调 BERT 模型以适用于单句二元分类

本节讨论如何使用流行的 IMDb 情感数据集对预训练好的 BERT 模型进行微调，以适用于情感分析。使用图形处理器 GPU 将加快学习过程，但如果用户没有 GPU 资源，也可以使用中央处理器 CPU 进行微调。具体操作步骤如下所示。

（1）为了了解并保存当前设备，可以执行以下代码。

```
from torch import cuda
device = 'cuda' if cuda.is_available() else 'cpu'
```

（2）此处使用 DistilBertForSequenceClassification 类，该类继承自 DistilBert 类，其头部有一个特殊的序列分类头。可以利用这个分类头（classification head）来训练分类模型，其中默认的分类数为 2。

```
from transformers import DistilBertTokenizerFast,
DistilBertForSequenceClassification
model_path = 'distilbert - base - uncased'
tokenizer = DistilBertTokenizerFast.from_pretrained①(model_path)
model = \
DistilBertForSequenceClassification.from_pretrained②(model_path,
id2label = {0:"NEG", 1:"POS"},
label2id = {"NEG":0, "POS":1})
```

（3）请注意，参数 id2label 和 label2id 被传递给模型，以在推理过程中使用。或者，可以实例化一个特定的 config（配置）对象并将其传递给模型。代码如下所示。

```
config = AutoConfig.from_pretrained③(...)
SequenceClassification.from_pretrained④(...
config = config)
```

（4）选择 IMDb 数据集的流行情感分类数据集。原始数据集由两组数据组成：25 000 个用于训练的样例和 25 个用于测试的样例。将数据集分成测试数据集和验证数据集。请注意，数据集前半部分的样例为正，而后半部分的样例均为负。按以下方式拆分样例。

```
from datasets import load_dataset
imdb_train = load_dataset('imdb', split = "train")
imdb_test = load_dataset('imdb', split = "test[:6250] + test[ -6250:]")
imdb_val = \
load_dataset('imdb', split = "test[6250:12500] + test[ -12500: -6250]")
```

（5）使用以下代码检查数据集的形状。

```
>>> imdb_train.shape, imdb_test.shape, imdb_val.shape
((25000, 2), (12500, 2), (12500, 2))
```

（6）根据计算资源获取数据集的一小部分。对于较小的部分，应该运行以下代码来选择 4 000 个样例进行训练。其中，1 000 个样例用于测试；1 000 个样例用于验证。代码如下所示。

```
imdb_train = load_dataset('imdb', split = "train[:2000] + train[ -2000:]")
imdb_test = load_dataset('imdb',
split = "test[:500] + test[ -500:]")
imdb_val = load_dataset('imdb', split = "test[500:1000] + test[ -1000: -500]")
```

① 原著此处有误，译者通过查看源码，做了修正，应该是 from_pretrained。——译者注
② 原著此处有误，译者通过查看源码，做了修正，应该是 from_pretrained。——译者注
③ 原著此处有误，译者通过查看源码，做了修正，应该是 from_pretrained。——译者注
④ 原著此处有误，译者通过查看源码，做了修正，应该是 from_pretrained。——译者注

（7）通过 tokenizer（分词器）模型处理这些数据集，以便为训练做好准备。

```
enc_train = imdb_train.map(lambda e: tokenizer(
e['text'], padding = True, truncation = True), batched = True,
batch_size = 1000)
enc_test = imdb_test.map(lambda e: tokenizer( e['text'],
padding = True, truncation = True), batched = True, batch_
size = 1000)
enc_val = imdb_val.map(lambda e: tokenizer( e['text'],
padding = True, truncation = True), batched = True, batch_
size = 1000)
```

（8）以下代码尝试查看训练数据集是什么样子。分词器将注意力掩码和输入 ID 添加到数据集中，以便 BERT 模型可以处理。

```
import pandas as pd
pd.DataFrame(enc_train)
```

输出结果如图 5 – 2 所示。

	attention_mask	input_ids	label	text
0	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 22953, 2213, 4381, 2152, 2003, 1037, 947...	1	Bromwell High is a cartoon comedy. It ran at t...
1	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 11573, 2791, 1006, 2030, 2160, 24913, 20...	1	Homelessness (or Houselessness as George Carli...
2	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 8235, 2058, 1011, 3772, 2011, 23920, 575...	1	Brilliant over-acting by Lesley Ann Warren. Be...
3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2023, 2003, 4089, 1996, 2087, 2104, 9250...	1	This is easily the most underrated film inn th...
4	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2023, 2003, 2025, 1996, 5171, 11463, 837...	1	This is not the typical Mel Brooks film. It wa...
...	...	...	...	...
24995	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2875, 1996, 2203, 1997, 1996, 3185, 1010...	0	Towards the end of the movie, I felt it was to...
24996	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2023, 2003, 1996, 2785, 1997, 3185, 2008...	0	This is the kind of movie that my enemies cont...
24997	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 1045, 2387, 1005, 6934, 1005, 2197, 2305...	0	I saw 'Descent' last night at the Stockholm Fi...
24998	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2070, 3152, 2008, 2017, 4060, 2039, 2005...	0	Some films that you pick up for a pound turn o...
24999	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	[101, 2023, 2003, 2028, 1997, 1996, 12873, 435...	0	This is one of the dumbest films, I've ever se...

25000 rows × 4 columns

图 5 – 2　编码后的训练数据集

此时，数据集已准备好进行训练和测试。Trainer 类（TensorFlow 的 TFTrainer）和 TrainingArguments 类（TensorFlow 的 TFTrainingArguments）将帮助用户完成大部分训练复杂性。在 TrainingArguments 类中定义参数集，然后将其传递给 Trainer 对象。

不同训练参数的定义如表 5 – 1 所示。

表 5 – 1　不同训练参数的定义

参数	定义
output_dir	输出目录，保存模型检查点以及训练结束后的预测
do_train 和 do_eval	用于监测训练期间模型性能的选项
logging_strategy	日志策略，可用选项为 no，epoch 和 steps（默认值）

续表

参数	定义
logging_steps （默认值 500）	两个日志之间的步骤数 （日志将保存到 logging_dir 目录中）
save_strategy	用于保存模型检查点的保存策略。 可用选项为 no， epoch 和 steps （默认值）
save_steps （默认值 500）	两个检查点之间的步骤数
fp16	用于混合精度， 可以混合使用 16 位和 32 位浮点数类型， 以提高模型训练速度， 同时减少使用的内存数量
load_best_model_at_end	在训练结束后， 根据验证损失， 载入最佳模型检查点
logging_dir	TensorBoard 日志目录

（9）有关更多信息，请查看 TrainingArguments 的 API 文档，或者在 Python 笔记本中执行以下代码。

```
TrainingArguments?
```

（10）尽管深度学习体系结构（如长短期记忆网络）需要许多学习迭代次数，有时甚至超过 50 个学习迭代次数，但基于 Transformer 的微调，由于迁移学习，通常经历 3 个学习迭代次数就会满足要求。在大多数情况下，3 个学习迭代次数足以进行微调，因为经过预训练的模型在预训练阶段（平均需要 50 个学习迭代次数）学习了大量有关语言的知识。为了确定正确的学习迭代次数，需要监控训练损失和验证损失。读者将在第 11 章中学习如何跟踪训练过程。

（11）这种方式足以解决许多下游任务问题，此处展开讨论。在训练过程中，每经过 200 个步骤，相应的模型检查点就被保存在“./MyIMDBModel”文件夹下。

```
from transformers import TrainingArguments, Trainer
training_args = TrainingArguments(
    output_dir='./MyIMDBModel',
    do_train=True,
    do_eval=True,
    num_train_epochs=3,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=64,
    warmup_steps=100,
    weight_decay=0.01,
    logging_strategy='steps',
    logging_dir='./logs',
    logging_steps=200,
    evaluation_strategy='steps',
```

```
    fp16 = cuda.is_available(),
    load_best_model_at_end = True
)
```

（12）在实例化 Trainer 对象之前，定义 compute_metrics( )方法。该方法帮助用户根据所需的特定指标［如精度、均方根误差、皮尔逊相关性、BLEU（BiLingual Evaluation Understudy，双语互译质量评估辅助工具）等］检测训练进度。文本分类问题（如情感分类或多类分类）大多采用微平均（micro - averaging）或宏平均（macro - averaging）F1 进行评估。宏平均方法为每个类别赋予相等的权重，而微平均方法为每个文本或每个标记分类决策赋予相等的权重。微平均值等于模型正确决策次数与已做出决策总数的比率。另外，宏平均方法计算每个类别的精度、召回率和 F1 的平均分数。对于用户的分类问题，宏平均值更便于评估，因为希望给每个标签赋予相等的权重。代码如下所示。

```
from sklearn.metrics import accuracy_score, Precision_
Recall_fscore_support
def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax( -1)
    Precision, Recall, f1, _ = \
    Precision_Recall_fscore_support(labels, preds, average ='macro')
    acc = accuracy_score(labels, preds)
    return {
      'Accuracy': acc,
      'F1': f1,
      'Precision': Precision,
      'Recall': Recall
    }
```

（13）至此，几乎准备好可以开始训练过程了。现在，可以实例化 Trainer 对象并启动该对象。Trainer 类是一个非常强大和优化的工具，用于组织 PyTorch 和 TensorFlow（TensorFlow 的 TFTrainer）的复杂训练和评估过程，当然，这得益于 Transformer 库。

```
trainer = Trainer(
    model = model,
    args = training_args,
    train_dataset = enc_train,
    eval_dataset = enc_val,
    compute_metrics = compute_metrics
)
```

（14）开始训练过程。

```
results = trainer.train()
```

调用上述函数后即开始记录度量指标（第 11 章将详细讨论度量指标）。整个 IMDb 数据集包括 25 000 个训练样例。当批次大小为 32 时，有 25 000/32≈782 个步骤，3 个学习迭代次数共 2 346（782×3）个步骤。Trainer 对象输出的结果如图 5－3 所示。

[2346/2346 21:13, Epoch 3/3]

Step	Training Loss	Validation Loss	Accuracy	F1	Precision	Recall	Runtime	Samples Per Second
200	0.417800	0.239647	0.900160	0.899943	0.903660	0.900160	58.657100	213.103000
400	0.251100	0.207064	0.918960	0.918960	0.918960	0.918960	58.724400	212.859000
600	0.237300	0.188785	0.926560	0.926554	0.926707	0.926560	58.727300	212.848000
800	0.209200	0.234559	0.923680	0.923621	0.924982	0.923680	58.750400	212.764000
1000	0.128500	0.248400	0.927280	0.927280	0.927286	0.927280	58.717100	212.885000
1200	0.137400	0.251818	0.920000	0.919869	0.922771	0.920000	58.713500	212.898000
1400	0.125900	0.186671	0.930720	0.930707	0.931054	0.930720	58.724900	212.857000
1600	0.111800	0.230385	0.932960	0.932959	0.932980	0.932960	58.695400	212.964000
1800	0.051300	0.255035	0.933440	0.933440	0.933440	0.933440	58.840300	212.440000
2000	0.045200	0.269209	0.934800	0.934795	0.934927	0.934800	58.819400	212.515000
2200	0.053700	0.242861	0.934640	0.934639	0.934661	0.934640	58.836100	212.455000

最小损失

图 5－3　Trainer 对象输出的结果

（15）在训练的最后，Trainer 对象保留验证损失最小的检查点。选择步骤 1 400 处的检查点，因为该步骤的验证损失最小。接下来，评估三个（训练、测试和验证）数据集上的最佳检查点。

```
>>> q=[trainer.evaluate(eval_dataset=data) for data in
[enc_train, enc_val, enc_test]]
>>> pd.DataFrame(q, index=["train","val","test"]).
iloc[:,:5]
```

输出结果如图 5－4 所示。

[391/391 10:03]

	eval_loss	eval_accuracy	eval_f1	eval_precision	eval_recall
train	0.057059	0.98320	0.983199	0.983259	0.98320
val	0.186671	0.93072	0.930707	0.931054	0.93072
test	0.213239	0.92616	0.926128	0.926904	0.92616

图 5－4　分类模型在训练/验证/测试数据集上的性能

（16）至此，读者成功地完成了训练阶段和测试阶段，获得了 92.6% 的准确率和 92.6 F1 的宏平均值。为了更详细地监测训练过程，可以调用 TensorBoard 等高级工具。用户使用这些工具解析日志能够跟踪各种指标以进行全面分析。此处已经在“./logs”文件夹下记录了性能指标和其他指标。只需在 Python 笔记本中运行 TensorBoard 工具就足够了，如以下代码片段所示（第 11 章将详细讨论 TensorBoard 和其他监控工具）。

```
%reload_ext tensorboard
```

```
%tensorboard --logdir logs
```

（17）现在，使用该模型进行推理，以检查模型是否正常工作。首先定义一个预测函数来简化预测步骤。代码如下所示。

```
def get_prediction(text):
    inputs = tokenizer(text, padding = True,truncation = True,
                max_length = 250, return_tensors = "pt").to(device)
    outputs = model①(inputs["input_ids"]. \
                to(device),inputs["attention_mask"].to(device))
    probs = outputs[0].softmax(1)
    return probs, probs.argmax()
```

（18）运行模型以进行推断。

```
>>> text = "I didn't like the movie it bored me "
>>> get_prediction(text)[1].item()
0
```

（19）结果为 0，表示一个负样例。之前已经定义了 ID 与标签的引用关系。此处可以使用此映射方案来获取标签。或者，可以简单地将所有步骤传递给一个专用的应用程序接口（即用户所熟悉的管道）。在实例化之前，首先保存最佳模型以供进一步推断。

```
model_save_path = "MyBestIMDBModel"
trainer.save_model(model_save_path)
tokenizer.save_pretrained②(model_save_path)
```

管道应用程序接口是使用预训练模型进行推理的一种简单方法。首先加载保存的模型，并将其传递给管道应用程序接口，由管道应用程序接口完成其余的工作。可以跳过保存模型的步骤，直接将内存中的模型和分词器对象传递给管道应用程序接口。两种方法的结果相同。

（20）如以下代码所示，在执行二元分类时，需要将管道的任务名称参数指定为 sentiment - analysis（情感分析）。

```
>>> from transformers import pipeline, \
③DistilBertForSequenceClassification, \
DistilBertTokenizerFast
>>> model = ④DistilBertForSequenceClassification. \
```

① 原著此处有误，误用了换行符(\)。——译者注
② 原著此处有误，译者通过查看源码做了修正，应该是 save_pretrained。——译者注
③ 原著此处有误，误用了换行符(\)。——译者注
④ 原著此处有误，误用了换行符(\)。——译者注

```
                from_pretrained①("MyBestIMDBModel")
>>> tokenizer = ②DistilBertTokenizerFast.\
                from_pretrained③("MyBestIMDBModel")
>>> nlp = pipeline("sentiment-analysis", model=model,
tokenizer=tokenizer)
>>> nlp("the movie was very impressive")
Out: [{'label': 'POS', 'score': 0.9621992707252502}]
>>> nlp("the text of the picture was very poor")
Out: [{'label': 'NEG', 'score': 0.9938313961029053}]
```

管道知道如何处理输入，并以某种方式了解哪个 ID 引用哪个标签（POS 或 NEG）。管道还会产生类别概率。

至此，使用 Trainer 类对 IMDb 数据集的情感预测模型进行了微调。在下一节中，读者将使用原生 PyTorch 进行相同的二元分类训练，还将使用不同的数据集进行训练。

5.4 使用原生 PyTorch 训练分类模型

Trainer 类非常强大，特别感谢 Hugging Face 团队提供了如此有用的工具。然而，在本节中，读者将从零开始微调预训练的模型，以查看模型的内部原理。操作步骤如下所示。

（1）加载需要微调的模型。这里将选择 DistilBERT，因为它是一个小型、快速、简单的 BERT 版本。

```
from transformers import DistilBertForSequenceClassification
model = DistilBertForSequenceClassification.from_pretrained④('distilbert-
base-uncased')
```

（2）为了微调模型，需要将模型置于训练模式。代码如下所示。

```
model.train()
```

（3）加载分词器。

```
from transformers import DistilBertTokenizerFast
tokenizer=DistilBertTokenizerFast.from_pretrained⑤('bert-base-uncased')
```

（4）由于 Trainer 类为用户组织了整个过程，因此在之前的 IMDb 情感分类练习中，

① 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注
② 原著此处有误，误用了换行符(\)。——译者注
③ 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注
④ 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注
⑤ 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注

并没有涉及优化和其他训练设置。现在，需要自己实例化优化器。此处必须选择 AdamW，它是 Adam 算法的一个实现，但具有权重衰减修正。最近的研究表明，使用 AdamW 训练的模型比使用 Adam 训练的模型会产生更好的训练损失和验证损失。因此，AdamW 是许多 Transformer 训练过程中广泛使用的优化器。

```
from transformers import AdamW
optimizer = AdamW(model.parameters(), lr =1e -3)
```

为了从零开始设计微调过程，必须了解如何实现单步前向传播和反向传播。可以通过 Transformer 层传递单个批次并获得输出，这被称为前向传播（forward propagation）。然后，必须使用输出和实际真值标签计算损失，并根据损失更新模型权重，这被称为反向传播（back propagation）。

以下代码片段在单个批次中接收与标签相关联的三句话，并执行前向传播。最后，模型自动计算损失。

```
import torch
texts = ["this is a good example","this is a bad
example","this is a good one"]
labels = [1,0,1]
labels = torch.tensor(labels).unsqueeze(0)
encoding = tokenizer(texts, return_tensors ='pt',
padding =True,
truncation =True, max_length =512)
input_ids = encoding['input_ids']
attention_mask = encoding['attention_mask']
outputs = \
model(input_ids, attention_mask =attention_mask,
labels =labels)
loss = outputs.loss
loss.backward()
optimizer.step()
Outputs
SequenceClassifierOutput(
[('loss', tensor(0.7178, grad_fn = <NllLossBackward >)),
('logits',tensor([[ 0.0664, -0.0161],[ 0.0738, 0.0665],[
0.0690, -0.0010]], grad_fn = <AddmmBackward >))])
```

该模型接收参数 input_ids 和 attention_mask（这两个都是分词器的输出结果），并使用实际真值标签计算损失。正如读者所看到的，输出由张量 loss 和 logits 组成。现在，loss. backward( )使用输入和标签来评估模型，以计算张量的梯度。optimizer. step( )执行单个优化步骤，并使用计算的梯度（称为反向传播）更新权重。当稍后将所有这些代码行放入一个循环中时，还将添加一个 optimizer. zero_grad( )函数，用于清除所有参数的梯度。必须在循环开始时调用该语句；否则，可能累计多个步骤的梯度。输出的第二个张量是 logits。在深度学习的背景下，术语 logits（logistic units 的缩写）是神经结构的

最后一层，由实数的预测值组成。在分类的情况下，需要调用 softmax( ) 函数将 logits 转换为概率。否则，这些值将被简单地标准化以进行回归。

（5）如果想手动计算 loss，那么不能将标签传递给模型。因此，该模型只生成 logits，不计算 loss。在以下示例中，手动计算交叉熵损失。

```
from torch.nn import functional
labels = torch.tensor([1,0,1])
outputs = model(input_ids, attention_mask = attention_mask)
loss = functional.cross_entropy(outputs.logits, labels)
loss.backward()
optimizer.step()
loss
Output: tensor(0.6101, grad_fn = <NllLossBackward >)
```

（6）通过上述步骤，了解了在一个单一的步骤中如何将批次输入沿前向馈送到网络中。现在，需要设计一个循环，该循环按批次迭代整个数据集，以训练具有多个学习迭代次数的模型。为此，将从设计自定义的 Dataset 类开始。该类是 torch. Dataset 的子类，继承父类的成员变量和函数，并实现__init__( )和__getitem__( )[①]抽象函数。

```
from torch.utils.data import Dataset
class MyDataset(Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels
    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key,
                        val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item
    def __len__(self):
        return len(self.labels)
```

（7）使用另一个称为 SST－2（即 Stanford Sentiment Treebank v2，斯坦福情感树库版本 2）的情感分析数据集来微调情感分析模型；还将加载 SST－2 的相应指标进行评估。代码如下所示。

```
import datasets
from datasets import load_dataset
sst2 = load_dataset("glue","sst2")
from datasets import load_metric
metric = load_metric("glue", "sst2")
```

（8）使用以下代码提取对应的句子和标签。

① 此处原著有误，应该是__getitem__( )。——译者注

```
texts = sst2['train']['sentence']
labels = sst2['train']['label']
val_texts = sst2['validation']['sentence']
val_labels = sst2['validation']['label']
```

（9）将数据集通过分词器 tokenizer 传递给 MyDataset( ) 构造函数，并实例化 MyDataset 对象，以使 BERT 模型与数据集一起工作。

```
train_dataset = MyDataset(tokenizer(texts,
truncation = True, padding = True), labels)
val_dataset = MyDataset(tokenizer(val_texts,
truncation = True, padding = True), val_labels)
```

（10）通过如下代码实例化一个 Dataloader 类的对象，该类提供了一个接口，可以通过加载的顺序来遍历数据样本。这个类还有助于批次和锁页内存（memory pinning）的相关执行。

```
from torch.utils.data import DataLoader
train_loader = DataLoader(train_dataset, batch_size = 16, shuffle = True)
val_loader = DataLoader(val_dataset, batch_size = 16, shuffle = True)
```

（11）使用以下代码检测设备，并按合适的方式定义 AdamW 优化器。

```
from transformers import AdamW
device = \
torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
model.to(device)
optimizer = AdamW(model.parameters(), lr = 1e - 3)
```

到目前为止，读者了解了如何实现前向传播，这属于处理一个批次样例的方式。在本例中，批次数据按前向馈入通过神经网络。在单个步骤中，从第一层到最后一层的每一层都由批次数据按照激活函数进行处理，并传递到后续层。为了在若干训练周期内遍历整个数据集，本节设计了两个嵌套循环：外部循环用于当前训练周期；内部循环用于每个批次的执行步骤。内部循环由两个语句块组成：一个语句块用于训练；另一个语句块用于评估每个训练周期。读者可能已经注意到，在第一个训练循环中调用了函数 model. train( )，然后当运行到第二个评估语句块时，调用了函数 model. eval( )。这一点很重要，因为本节将模型置于训练和推理模式。

（12）目前为止已经讨论了内部块。注意，可以通过相应的 metric 度量对象来跟踪模型的性能。代码如下所示。

```
for epoch in range(3):
    model.train()
    for batch in train_loader:
```

```
        optimizer.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
        outputs = \
        model(input_ids, attention_mask=attention_mask,
        labels=labels)
        loss = outputs[0]
        loss.backward()
        optimizer.step()
    model.eval()
    for batch in val_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
        outputs = \
                model(input_ids, attention_mask=attention_mask,
                      labels=labels)
        predictions=outputs.logits.argmax(dim=-1)
        metric.add_batch(
              predictions=predictions,
              references=batch["labels"],
        )
    eval_metric = metric.compute()
    print(f"epoch {epoch}: {eval_metric}")
OUTPUT:
epoch 0: {'accuracy': 0.9048165137614679}
epoch 1: {'accuracy': 0.8944954128440367}
epoch 2: {'accuracy': 0.9094036697247706}
```

通过对模型进行微调，精度达到 90.94% 左右。剩下的过程，如保存、加载和推断，将与 Trainer 类的训练方法类似。

至此，读者完成了二元分类的学习。下一节将实现一个多类别分类模型，以用于英语以外的语言。

5.5 使用自定义数据集对多类别分类 BERT 模型进行微调

本节将微调 Turkish（土耳其语）版本的 BERT，即 BERTurk，使用自定义数据集执行 7 个类别分类下游任务。该数据集由土耳其语报纸汇编而成，分为 7 个类别。本节将从获取数据集开始。可以从以下网址下载该数据集：https://www.kaggle.com/savasy/ttc4900。

（1）在 Jupyter Notebook 中，运行以下代码以获取数据。

```
!wget https://raw.githubusercontent.com/savasy/TurkishTextClassification/master/TTC4900.csv
```

（2）开始加载数据。

```
import pandas as pd
data = pd.read_csv("TTC4900.csv")
data = data.sample(frac = 1.0, random_state = 42)
```

（3）使用字典数据 id2label 和 label2id 组织 ID 和标签，使模型了解 ID 与标签之间的引用关系。还将向模型传递标签数量（NUM_LABELS），以指定 BERT 模型顶部的薄分类头层（thin classification head layer）的大小。

```
labels = ["teknoloji","ekonomi","saglik","siyaset",
"kultur","spor","dunya"]
NUM_LABELS = len(labels)
id2label = {i:l for i,l in enumerate(labels)}
label2id = {l:i for i,l in enumerate(labels)}
data["labels"] = data.category.map(lambda x: label2id[x.strip()])
data.head()
```

输出结果如图 5 - 5 所示。

	category	text	labels
4657	teknoloji	acıların kedisi sam çatık kaşlı kedi sam in i...	0
3539	spor	g saray a git santos van_persie den forma ala...	5
907	dunya	endonezya da çatışmalar 14 ölü endonezya da i...	6
4353	teknoloji	emniyetten polis logolu virüs uyarısı telefon...	0
3745	spor	beni türk yapın cristian_baroni yıldırım dan ...	5

图 5 - 5　文本分类数据集：TTC 4900

（4）以下代码使用 pandas 对象统计并绘制类别的数量。

```
data.category.value_counts().plot(kind='pie')
```

如图 5 - 6 所示，数据集的类别分布比较均匀。

（5）执行以下代码片段，将实例化一个序列分类模型，其中包含标签数（7 个）、标签 ID 映射和一个土耳其语 BERT 模型（dbmdz/bert - base - turkish - uncased），即 BERTurk。为了检查所创建的模型，请执行以下语句。

```
>>> model
```

（6）输出结果为模型的摘要，内容太长无法全部显示。作为替代，可以使用以下代码将注意力转向最后一层。

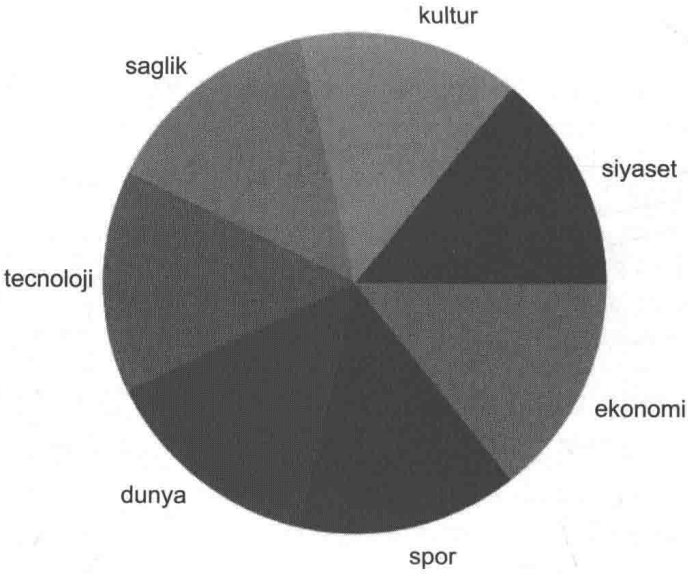


图 5－6 类别分布

```
(classifier): Linear(in_features=768, out_features=7, bias=True)
```

（7）读者可能已经注意到，以下代码中并没有选择 DistilBert，因为土耳其语中没有经过预先训练的 uncased（不区分大、小写）的 DistilBert。

```
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained①("dbmdz/
bert-base-turkish-uncased", max_length=512)
from transformers import BertForSequenceClassification
model = BertForSequenceClassification.from_
pretrained②("dbmdz/bert-base-turkish-uncased", num_
labels=NUM_LABELS, id2label=id2label, label2id=label2id)
model.to(device)
```

（8）现在，可以使用如下代码来准备训练数据集、验证数据集和测试数据集。

```
SIZE = data.shape[0]
## 句子
train_texts = list(data.text[:SIZE//2])
val_texts = list(data.text[SIZE//2:(3*SIZE)//4 ])
test_texts = list(data.text[(3*SIZE)//4:])
## 标签
train_labels = list(data.labels[:SIZE//2])
val_labels = list(data.labels[SIZE//2:(3*SIZE)//4])
```

① 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注

② 原著此处有误，译者通过查看源码做了修正，应该是 pretrained。——译者注

```
test_labels = list(data.labels[(3 * SIZE) //4:])
## 检测数据集的大小
len(train_texts), len(val_texts), len(test_texts)
(2450, 1225, 1225)
```

（9）以下代码对三个数据集中出现的语句及其标记进行分词，并将其转换为整数输入（input_ids），然后将其馈入 BERT 模型。

```
train_encodings = tokenizer(train_texts, truncation = True,
padding = True)
val_encodings = tokenizer(val_texts, truncation = True,
padding = True)
test_encodings = tokenizer(test_texts, truncation = True,
padding = True)
```

（10）在前面的章节中已经实现了 MyDataset 类（具体请参见 5.4 节[①]）。MyDataset 类重写了从抽象 Dataset 类中继承的__getitem__()方法和__len__()方法，这两个方法分别使用数据加载器返回数据项和数据集的大小。

```
train_dataset = MyDataset(train_encodings, train_labels)
val_dataset = MyDataset(val_encodings, val_labels)
test_dataset = MyDataset(test_encodings, test_labels)
```

（11）此处仍将保持批次大小为 16，因为数据集相对较小。请注意，TrainingArguments 的其他参数设置与之前的情感分析实验几乎相同。

```
from transformers import TrainingArguments, Trainer
training_args = TrainingArguments(
    output_dir = './TTC4900Model',
    do_train = True,
    do_eval = True,
    num_train_epochs = 3,
    per_device_train_batch_size = 16,
    per_device_eval_batch_size = 32,
    warmup_steps = 100,
    weight_decay = 0.01,
    logging_strategy = 'steps',
    logging_dir = './multi - class - logs',
    logging_steps = 50,
    evaluation_strategy = "steps",
    eval_steps = 50,
    save_strategy = "epoch",
```

① 原著此处关于页数的引用有误，建议改为“第 5.4 节”。——译者注

```
        fp16 = True,
        load_best_model_at_end = True
    )
```

（12）情感分析和文本分类是同一评价指标的对象。也就是说，统一采用宏平均值 F1、精度和召回率这些评价指标。因此，此处不再定义 compute_metric( ) 函数。以下是用于实例化 Trainer 对象的代码。

```
trainer = Trainer(
    model = model,
    args = training_args,
    train_dataset = train_dataset,
    eval_dataset = val_dataset,
    compute_metrics = compute_metrics
)
```

（13）开始训练过程。

```
trainer.train()
```

输出结果如图 5 - 7 所示。

[462/462 12:27, Epoch 3/3]

Step	Training Loss	Validation Loss	Accuracy	F1	Precision	Recall	Runtime	Samples Per Second
50	1.874100	1.706715	0.377143	0.379416	0.553955	0.383715	20.982000	58.383000
100	0.842900	0.327575	0.915102	0.913738	0.914565	0.915279	20.981700	58.384000
150	0.358200	0.281808	0.911020	0.910288	0.912012	0.911213	20.997000	58.342000
200	0.233500	0.366845	0.905306	0.905313	0.916948	0.903440	20.980800	58.387000
250	0.222700	0.292270	0.922449	0.921374	0.921567	0.923131	20.981300	58.385000
300	0.257700	0.280120	0.924898	0.923810	0.924427	0.924510	20.979800	58.390000
350	0.115200	0.292410	0.925714	0.924946	0.924752	0.925454	20.982700	58.381000
400	0.064900	0.322697	0.925714	0.924944	0.925674	0.925265	20.988300	58.366000
450	0.080400	0.297606	0.929796	0.929267	0.929170	0.929497	20.985100	58.375000

最小损失

图 5 - 7　用于文本分类的 Trainer 类的输出结果

（14）为了检查经过训练的模型，必须在三个数据集分割上评估经过微调的模型，代码片段如下所示。最佳模型在步骤 300 进行了微调，损失值为 0. 280 120。

```
q = [trainer.evaluate(eval_dataset = data) for data in
[train_dataset, val_dataset, test_dataset]]
pd.DataFrame(q, index = ["train","val","test"]).iloc[:,:5]
```

输出结果如图 5 - 8 所示。

[77/77 01:24]

	eval_loss	eval_Accuracy	eval_F1	eval_Precision	eval_Recall
train	0.091844	0.975510	0.97546	0.975942	0.975535
val	0.280120	0.924898	0.92381	0.924427	0.924510
test	0.280038	0.926531	0.92542	0.927410	0.925425

图 5－8　文本分类模型在训练/验证/测试数据集上的性能

分类精度约为 92.6%，而宏平均值 F1 约为 92.5。在文献中，许多方法已经在这个土耳其基准数据集上进行了测试。它们大多遵循词频－逆向文档频率和线性分类器、Word2vec 嵌入，或者基于长短期记忆网络的分类器，最多得到 90.0 F1 左右的分值。与这些方法相比，除了 Transformer，微调的 BERT 模型的性能也优于这些方法。

（15）与任何其他实验一样，可以通过 TensorBoard 跟踪实验。

```
%load_ext tensorboard
%tensorboard --logdir multi-class-logs/
```

（16）设计一个函数来运行模型进行推理。如果希望看到真实的标签而不是 ID，可以使用模型的 config 对象。predict( ) 函数的实现代码如下所示。

```
def predict(text):
    inputs = tokenizer(text, padding=True, truncation=True,
            max_length=512, return_tensors="pt").to("cuda")
    outputs = model(**inputs)
    probs = outputs[0].softmax(1)
    return probs, probs.argmax(), model.config.id2label[probs.argmax().
item()]
```

（17）调用 predict( ) 函数进行文本分类推断。以下代码对有关足球队的句子进行了分类。

```
text = "Fenerbahçeli futbolcular kısa paslarla hazırlık çalı§ması yaptılar"
predict(text)
(tensor([[5.6183e-04, 4.9046e-04, 5.1385e-04, 9.9414e-04,
3.4417e-04, 9.9669e-01, 4.0617e-04]], device='cuda:0',
grad_fn=<SoftmaxBackward>), tensor(5, device='cuda:0'),
'spor')
```

（18）正如所见，该模型正确地预测了句子为 sports(spor)。现在，需要保存模型并使用 from_pretrained( )[①]函数重新加载模型。代码如下所示。

① 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注

```
model_path = "turkish-text-classification-model"
trainer.save_model(model_path)
tokenizer.save_pretrained①(model_path)
```

（19）现在，可以使用 pipeline 类重新加载所保存的模型并运行推断。

```
model_path = "turkish-text-classification-model"
from transformers import pipeline, BertForSequenceClassification, BertTokenizerFast
model = BertForSequenceClassification.from_pretrained②(model_path)
tokenizer = BertTokenizerFast.from_pretrained③(model_path)
nlp = pipeline("sentiment-analysis", model=model,
tokenizer=tokenizer)
```

（20）读者可能已经注意到任务的名称是 sentiment - analysis（情感分析）。这个术语可能会让人困惑，但这个参数实际上会在末尾返回 TextClassificationPipeline。请使用以下代码运行管道。

```
>>> nlp("Sinemada hangi filmler oynuyor bugün")
[{'label':'kultur','score':0.9930670261383057}]
>>> nlp("Dolar ve Euro bugün yurtiçi piyasalarda
yükseldi")
[{'label':'ekonomi','score':0.9927696585655212}]
>>> nlp("Bayern Münih ile Barcelona bugün karsı karsıya
geliyor. Maçı Ingiliz hakem James Watts yönetecek!")
[{'label':'spor','score':0.9975664019584656}]
```

结果正是本文所期望的模型！预测成功！

到目前为止，本节实现了针对单个句子的两项任务，即情感分析和多类别分类。在下一节中，读者将学习如何处理句子对的输入，以及如何使用 BERT 设计回归模型。

5.6 微调 BERT 模型以适用于句子对回归

回归模型可以用于分类，但最后一层仅包含一个单元，因为结果并不是通过 softmax 逻辑回归进行处理，而是通过标准化进行处理。为了确定最终的模型并在顶部放置单个单元头层，可以直接将 num_labels = 1 参数传递给 BERT. from_pretrained( )④方法，或者通过 config 对象传递此信息。最初，这需要从预训练模型的 config 对象复制。代码如下所示。

---

① 原著此处有误，译者通过查看源码做了修正，应该是 save_pretrained。——译者注

② 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注

③ 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注

④ 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注

```
from transformers import DistilBertConfig, DistilBertTokenizerFast, Distil-
BertForSequenceClassification
model_path ='distilbert - base - uncased'
config = DistilBertConfig.from_pretrained①(model_path, num_labels =1)
tokenizer = DistilBertTokenizerFast.from_pretrained②(model_path)
model = \
DistilBertForSequenceClassification.from_pretrained③(model_path, config =
config)
```

由于指定了参数 num_labels = 1，因此预训练的模型有单个单元头层。现在，准备使用数据集对模型进行微调。此处，使用语义 - 文本相似性基准（Semantic Textual Similarity - Benchmark，STS - B），该基准是从各种内容（如新闻标题）中提取的若干句子对的集合。每一个句子对都标注了从 1 到 5 的相似性分数。本节的任务是对 BERT 模型进行微调以预测这些分数。在参考相关文献的同时，本节使用皮尔逊/斯皮尔曼（Pearson/Spearman）相关系数对模型进行评估。操作步骤如下。

（1）使用以下代码加载数据。原始数据被分成 3 个部分。但是，测试拆分没有标签，因此可以将验证数据分为两部分。代码如下所示。

```
import datasets
from datasets import load_dataset
stsb_train = load_dataset('glue','stsb', split = "train")
stsb_validation = load_dataset('glue','stsb', split = "validation")
stsb_validation = stsb_validation.shuffle(seed =42)
stsb_val = datasets.Dataset.from_dict(stsb_validation[:750])
stsb_test = datasets.Dataset.from_dict(stsb_validation[750:])
```

（2）使用 pandas 封装 stsb_train 训练数据，使其更加整洁。

```
pd.DataFrame(stsb_train)
```

STS - B 训练数据集如图 5 - 9 所示。

（3）运行以下代码，检查 3 个数据集的形状。

```
stsb_train.shape, stsb_val.shape, stsb_test.shape
((5749, 4), (750, 4), (750, 4))
```

（4）运行以下代码，对数据集进行分词。

```
enc_train = stsb_train.map(lambda e: tokenizer(
e['sentence1'],e['sentence2'], padding = True,
```

① 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注
② 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注
③ 原著此处有误，译者通过查看源码做了修正，应该是 from_pretrained。——译者注

	idx	label	sentence1	sentence2
0	0	5.00	A plane is taking off.	An air plane is taking off.
1	1	3.80	A man is playing a large flute.	A man is playing a flute.
2	2	3.80	A man is spreading shreded cheese on a pizza.	A man is spreading shredded cheese on an uncoo...
3	3	2.60	Three men are playing chess.	Two men are playing chess.
4	4	4.25	A man is playing the cello.	A man seated is playing the cello.
...	...	...	...	...
5744	5744	0.00	Severe Gales As Storm Clodagh Hits Britain	Merkel pledges NATO solidarity with Latvia
5745	5745	0.00	Dozens of Egyptians hostages taken by Libyan t...	Egyptian boat crash death toll rises as more b...
5746	5746	0.00	President heading to Bahrain	President Xi: China to continue help to fight ...
5747	5747	0.00	China, India vow to further bilateral ties	China Scrambles to Reassure Jittery Stock Traders
5748	5748	0.00	Putin spokesman: Doping charges appear unfounded	The Latest on Severe Weather: 1 Dead in Texas ...

5749 rows × 4 columns

图 5－9　STS－B 训练数据集

```
truncation=True), batched=True, batch_size=1000)
enc_val = stsb_val.map(lambda e: tokenizer(
e['sentence1'],e['sentence2'], padding=True,
truncation=True), batched=True, batch_size=1000)
enc_test = stsb_test.map(lambda e: tokenizer(
e['sentence1'],e['sentence2'], padding=True,
truncation=True), batched=True, batch_size=1000)
```

（5）分词器将两个句子与［SEP］分隔符合并，并为一个句子对生成单个 input_ids 和一个 attention_mask。代码如下所示。

```
pd.DataFrame(enc_train)
```

输出结果如图 5－10 所示。

	attention_mask	idx	input_ids	label	sentence1	sentence2
0	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	0	[101, 1037, 4946, 2003, 2635, 2125, 1012, 102,...	5.00	A plane is taking off.	An air plane is taking off.
1	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	1	[101, 1037, 2158, 2003, 2652, 1037, 2312, 8928...	3.80	A man is playing a large flute.	A man is playing a flute.
2	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	2	[101, 1037, 2158, 2003, 9359, 14021, 5596, 209...	3.80	A man is spreading shreded cheese on a pizza.	A man is spreading shredded cheese on an uncoo...
3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	3	[101, 2093, 2273, 2024, 2652, 7433, 1012, 102,...	2.60	Three men are playing chess.	Two men are playing chess.
4	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	4	[101, 1037, 2158, 2003, 2652, 1996, 10145, 101...	4.25	A man is playing the cello.	A man seated is playing the cello.
...	...	...	...	...	...	...
5744	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	5744	[101, 5729, 14554, 2015, 2004, 4040, 18856, 13...	0.00	Severe Gales As Storm Clodagh Hits Britain	Merkel pledges NATO solidarity with Latvia
5745	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	5745	[101, 9877, 1997, 23437, 19323, 2579, 2011, 19...	0.00	Dozens of Egyptians hostages taken by Libyan t...	Egyptian boat crash death toll rises as more b...
5746	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	5746	[101, 2343, 5825, 2000, 15195, 102, 2343, 8418...	0.00	President heading to Bahrain	President Xi: China to continue help to fight ...
5747	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	5747	[101, 2859, 1010, 2634, 19076, 2000, 2582, 177...	0.00	China, India vow to further bilateral ties	China Scrambles to Reassure Jittery Stock Traders
5748	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...	5748	[101, 22072, 14056, 1024, 23799, 5571, 3711, 4...	0.00	Putin spokesman: Doping charges appear unfounded	The Latest on Severe Weather: 1 Dead in Texas ...

5749 rows × 6 columns

图 5－10　编码后的训练数据集

与其他实验类似，本节在 TrainingArguments 类和 Trainer 类中遵循几乎相同的方案。实现代码如下所示。

```
from transformers import TrainingArguments, Trainer
```

```
training_args = TrainingArguments(
    output_dir='./stsb-model',
    do_train=True,
    do_eval=True,
    num_train_epochs=3,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=64,
    warmup_steps=100,
    weight_decay=0.01,
    logging_strategy='steps',
    logging_dir='./logs',
    logging_steps=50,
    evaluation_strategy="steps",
    save_strategy="epoch",
    fp16=True,
    load_best_model_at_end=True
)
```

（6）当前回归任务和以前分类任务之间的另一个重要区别是对 compute_metrics( ) 函数的设计。此处所使用的评估指标将基于皮尔逊相关系数（Pearson Correlation Coefficient）和斯皮尔曼等级相关性（Spearman's Rank Correlation），遵循相关文献中提供的常见做法。本节还提供了常用的（尤其是针对回归模型）均方误差（Mean Squared Error，MSE）、均方根误差（Root Mean Square Error，RMSE）和平均绝对误差（Mean Absolute Error，MAE）度量指标。

```
import numpy as np
from scipy.stats import pearsonr
from scipy.stats import spearmanr
def compute_metrics(pred):
    preds = np.squeeze(pred.predictions)
    return {"MSE": ((preds - pred.label_ids) **2).mean().item(),
            "RMSE": (np.sqrt (( (preds - pred.label_ids) ** 2).mean())).
item(),
            "MAE": (np.abs(preds - pred.label_ids)).mean().item(),
            "Pearson" : pearsonr(preds,pred.label_ids)[0],
            "Spearman's Rank":spearmanr(preds,pred.label_ids)[0]
          }
```

（7）使用以下代码实例化 Trainer 对象。

```
trainer = Trainer(
          model=model,
          args=training_args,
          train_dataset=enc_train,
          eval_dataset=enc_val,
```

```
            compute_metrics = compute_metrics,
            tokenizer = tokenizer
)
```

执行训练，代码如下所示。

```
train_result = trainer.train()
```

训练结果如图 5－11 所示。

Step	Training Loss	Validation Loss	Mse	Rmse	Mae	Pearson	Spearman's rank	Runtime	Samples Per Second
50	4.973200	2.242550	2.242550	1.497515	1.261815	0.140489	0.138228	0.943900	794.538000
100	1.447300	0.801587	0.801587	0.895314	0.735321	0.808588	0.809430	0.933300	803.602000
150	0.940400	0.693730	0.693730	0.832904	0.675787	0.843234	0.842421	0.930700	805.838000
200	0.736300	0.679696	0.679696	0.824437	0.662136	0.846722	0.843393	0.934700	802.407000
250	0.585400	0.590002	0.590002	0.768116	0.618677	0.859470	0.854824	0.931600	805.067000
300	0.513800	0.584674	0.584674	0.764640	0.610141	0.861033	0.856779	0.942900	795.438000
350	0.488000	0.604512	0.604512	0.777504	0.611338	0.865844	0.861726	0.939600	798.174000
400	0.362900	0.555219	0.555219	0.745130	0.582900	0.868366	0.863372	0.938200	799.379000
450	0.298500	0.544973	0.544973	0.738223	0.576751	0.868145	0.864209	0.938200	799.407000
500	0.270100	0.546966	0.546966	0.739571	0.575326	0.867538	0.864035	0.941900	796.240000

图 5－11　文本回归的训练结果

（8）在步骤 450 计算的最佳验证损失为 0.544 973。接下来，在该步骤中评估最佳检查点模型。代码如下所示。

```
q = [trainer.evaluate(eval_dataset = data) for data in [enc_train, enc_val,
enc_test]]
pd.DataFrame(q, index = ["train","val","test"]).iloc[:,:5]
```

输出结果如图 5－12 所示。

[90/90 00:30]

	eval_loss	eval_MSE	eval_RMSE	eval_MAE	eval_Pearson	eval_Spearman's Rank
train	0.232471	0.232471	0.482152	0.372915	0.944844	0.935578
val	0.544973	0.544973	0.738223	0.576751	0.868145	0.864209
test	0.537752	0.537752	0.733316	0.567489	0.875409	0.872858

图 5－12　训练/验证/测试数据集上的回归性能

在测试数据集上，皮尔逊相关分数和斯皮尔曼相关分数的值分别约为 87.54 和 87.28。虽然没有得到 SoTA 结果，但确实得到了基于通用语言理解评估（GLUE）基准排行榜的语义文本相似性基准测试（STS－B）任务的可比结果。

（9）现在已经准备好，可以运行模型进行推断。通过以下代码将两个意思相同的句子传递给模型。

```
s1,s2 = "A plane is taking off.","An air plane is taking off."
encoding = tokenizer(s1,s2, return_tensors ='pt',
padding = True, truncation = True, max_length = 512)
input_ids = encoding['input_ids'].to(device)
attention_mask = encoding['attention_mask'].to(device)
outputs = model(input_ids, attention_mask = attention_mask)
outputs.logits.item()
OUTPUT: 4.033723831176758
```

（10）以下代码使用否定句子对，这意味着这些句子在语义上是不同的。

```
s1,s2 = "The men are playing soccer.","A man is riding a motorcycle."
encoding = tokenizer("hey how are you there","hey how are you",
return_tensors ='pt', padding = True, truncation = True, max_length = 512)
input_ids = encoding['input_ids'].to(device)
attention_mask = encoding['attention_mask'].to(device)
outputs = model(input_ids, attention_mask = attention_mask)
outputs.logits.item()
OUTPUT: 2.3579328060150146
```

（11）运行以下代码保存模型。

```
model_path = "sentence - pair - regression - model"
trainer.save_model(model_path)
tokenizer.save_pretrained①(model_path)
```

非常棒，读者可以祝贺一下自己，因为读者已经成功地完成了 3 项任务：情感分析、多类别分类和句子对回归。

5.7　使用 run_glue. py 对模型进行微调

到目前为止，本章使用原生 PyTorch 和 Trainer 类从零开始设计了一个微调架构。Hugging Face 社区还提供了另一个名为“run_glue. py”的强大脚本，用于语义文本相似性基准测试 GLUE 和类似 GLUE 的分类下游任务。该脚本可以为用户处理和组织整个训练过程和验证过程。如果用户想快速构建原型，那么应该使用这个脚本。该脚本可以对 Hugging Face hub 上任何预训练的模型进行微调。用户也可以把自己的数据以任何格式提供给这个脚本。

读者可以访问以下链接以获取该脚本并了解其更多的信息：

https://github. com/huggingface/transformers/tree/master/examples。

该脚本可以执行 9 种不同的语义文本相似性基准测试任务。有了这个脚本，就可以

① 原著此处有误，译者通过查看源码做了修正，应该是 save_pretrained。——译者注

完成迄今为止在 Trainer 类中所做的一切。任务名称可以是以下语义文本相似性基准测试任务之一：cola，sst2，mrpc，stsb，qqp，mnli，qnli，rte 或 wnli。

以下是用于对模型进行微调的脚本方案。

```
export TASK_NAME = "My - Task - Name"
python run_glue.py \
  --model_name_or_path bert - base - cased \
  --task_name $TASK_NAME \
  --do_train \ --do_eval \
  --max_seq_length 128 \
  --per_device_train_batch_size 32 \
  --learning_rate 2e - 5 \
  --num_train_epochs 3 \
  --output_dir /tmp/ $TASK_NAME/
```

社区提供了另一个名为“run_glue_no_trainer. py”的脚本。原始脚本与此脚本之间的主要区别在于，此 no - trainer 脚本为用户提供了更多机会来更改优化器的选项，或者添加用户想要执行的任何定制。

5.8 本章小结

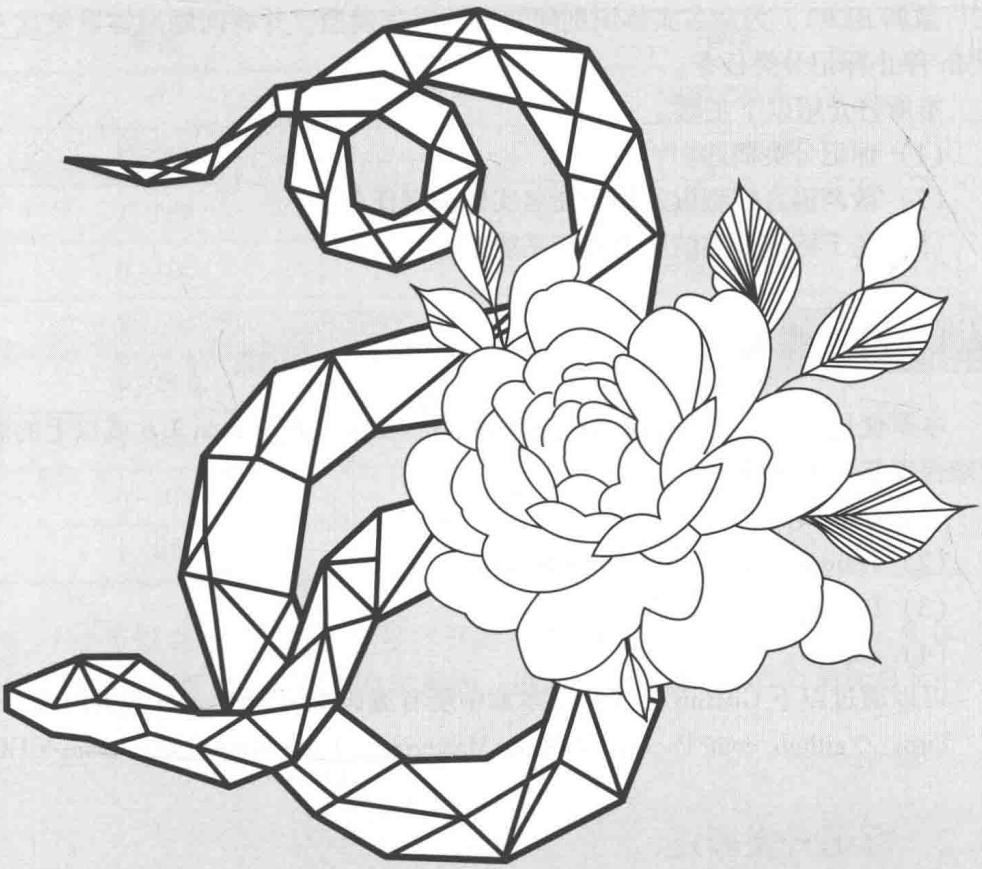
本章讨论了如何为文本分类下游任务微调预训练的模型，使用情感分析、多类别分类和句子对分类（更具体地说，是句子对回归）对模型进行了微调，使用了一个著名的 IMDb 数据集和自己的定制数据集来训练这些模型。虽然使用了 Trainer 类应对训练过程和微调过程的复杂性，但读者也学习了如何使用原生库从零开始训练，以理解 Transformers 库的前向传播和反向传播。综上所述，本章讨论并执行了以下几种微调：使用 Trainer 的单句分类微调、没有使用 Trainer 的原生 PyTorch 情感分类微调、单句多类别分类微调和句子对回归微调。

在下一章中，读者将学习如何对一个预训练好的模型进行微调，以用于标记分类下游任务，如 POS 或 NER。

6

第6章

微调标记分类语言模型



在本章中，读者将学习如何对标记分类（token classification）语言模型进行微调。本章探讨命名实体识别、词性标注和问题回答系统等任务。读者将学习如何在这些任务中对特定的语言模型进行微调。与其他语言模型相比，本章更多地关注 BERT。读者将学习如何使用 BERT 实现词性标注、命名实体识别和问题回答系统：熟悉这些任务的理论细节，例如，这些任务各自的数据集以及如何处理数据集。完成本章的学习之后，读者将能够使用 Transformers 执行所有的标记分类任务。

本章将为以下任务对 BERT 进行微调：为标记分类问题（如命名实体识别和词性标注）微调 BERT，为命名实体识别问题微调语言模型，并将问题回答系统这一问题视为开始/停止标记分类任务。

本章将介绍以下主题。

（1）标记分类概述。

（2）微调语言模型以适用于命名实体识别任务。

（3）基于标记分类的问题回答系统。

6.1 技术需求

本章使用 Jupyter Notebook 运行编码练习，要求安装 Python 3.6 或以上的版本，还必须确保安装以下软件包。

（1）sklearn；

（2）Transformer（不低于 4.0.0 版本）；

（3）Datasets；

（4）seqeval。

可以通过以下 GitHub 链接获得本章中所有编码练习的 Jupyter Notebook：

https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH06。

6.2 标记分类概述

对标记序列中的每个标记进行分类的任务称为标记分类。此任务要求特定模型必须能够将每个标记分类为一个类。词性标注和命名实体识别是这个标准下最著名的两个任

务。然而，问题回答系统也是另一个适合这一类别的主要自然语言处理任务。本章讨论这 3 项任务的基础知识。

6.2.1　理解命名实体识别

标记分类类别中的一个众所周知的任务是命名实体识别，即识别每个标记是否为实体，并识别每个检测到的实体所属的类型。例如，文本可以同时包含多个实体，包括人名、位置、组织以及其他类型的实体。以下文本是命名实体识别的一个明确示例。

George Washington is one the presidents of the United States of America.

George Washington（乔治·华盛顿）是人名，而 the United States of America（美利坚合众国）是地名。序列标记模型期望以标签的形式标记每个单词，每个标签包含关于标记的信息。BIO 标签［B - begin（开始）、I - inside（内部）、O - outside（外部）］是标准命名实体识别任务中普遍使用的标签。

表 6 - 1 列出了 BIO 标签及其说明。

表 6 - 1　BIO 标签及其说明

标签	说明
O	实体的外部
B - PER	开始人名实体
I - PER	人名实体的内部
B - LOC	开始地名实体
I - LOC	地名实体的内部
B - ORG	开始组织实体
I - ORG	组织实体的内部
B - MISC	开始其他实体
I - MISC	其他实体的内部

从表 6 - 1 中可以看出，B 表示标签的开始；I 表示标签的内部；O 表示实体的外部。这就是这种类型的标注被称为 BIO 的原因。例如，可以使用 BIO 对前面显示的句子进行标注。

```
[B - PER |George] [I - PER |Washington] [O |is] [O |one] [O |the]
[O |presidents] [O |of] [B - LOC |United] [I - LOC |States] [I - LOC |of]
[I - LOC |America] [O |.]
```

因此，序列必须以 BIO 格式标记。CONLL2003 数据集的格式如图 6 - 1 所示。

```
SOCCER NN B-NP O
- : O O
JAPAN NNP B-NP B-LOC
GET VB B-VP O
LUCKY NNP B-NP O
WIN NNP I-NP O
, , O O
CHINA NNP B-NP B-PER
IN IN B-PP O
SURPRISE DT B-NP O
DEFEAT NN I-NP O
. . O O

Nadim NNP B-NP B-PER
Ladki NNP I-NP I-PER

AL-AIN NNP B-NP B-LOC
, , O O
United NNP B-NP B-LOC
Arab NNP I-NP I-LOC
Emirates NNPS I-NP I-LOC
1996-12-06 CD I-NP O
```

图 6－1　CONLL2003 数据集的格式

除了看到的命名实体识别标签外，此数据集中还包含词性标注标签。

6.2.2　理解词性标注

词性标注（或者称为语法标注），是指在给定的文本中，根据词性来标注单词。举个简单的例子，在一个给定的文本中，识别每个单词的角色属于名词、形容词、副词和动词中的哪个类别，这被认为是识别单词的词性。然而，从语言学的角度来看，除了这四种词性之外，还有许多其他的角色。

在有词性标注标签的情况下，存在一些变化，但宾州树库（Penn Treebank）词性标注标签集是最著名的标签集之一。

图 6－2 所示为宾州树库词性标注的标签及其说明。

词性标注标签的标注（见图 6－1）在特定自然语言处理应用程序中非常有用，并且是许多其他方法的构建块之一。Transformer 和许多先进的模型可以在某种程度上理解其复杂架构中单词之间的关系。

6.2.3　理解问题回答系统

问题回答（或者称为阅读理解）任务包括一组阅读理解文本，其中包含相应的问题。此范围内的示例性数据集是斯坦福问题回答数据集（Stanford Question Answering Dataset，SQuAD）。该数据集由维基百科文本和相关问题组成。与问题相关的答案以原始维基百科文本的片段形式给出。

图 6－3 所示是该数据集的一个样例。

1. CC	Coordinating conjunction	25. TO	*to*
2. CD	Cardinal number	26. UH	Interjection
3. DT	Determiner	27. VB	Verb, base form
4. EX	Existential *there*	28. VBD	Verb, past tense
5. FW	Foreign word	29. VBG	Verb, gerund/present participle
6. IN	Preposition/subordinating conjunction	30. VBN	Verb, past participle
7. JJ	Adjective	31. VBP	Verb, non-3rd ps. sing. present
8. JJR	Adjective, comparative	32. VBZ	Verb, 3rd ps. sing. present
9. JJS	Adjective, superlative	33. WDT	*wh*-determiner
10. LS	List item marker	34. WP	*wh*-pronoun
11. MD	Modal	35. WP$	Possessive *wh*-pronoun
12. NN	Noun, singular or mass	36. WRB	*wh*-adverb
13. NNS	Noun, plural	37. #	Pound sign
14. NNP	Proper noun, singular	38. $	Dollar sign
15. NNPS	Proper noun, plural	39. .	Sentence-final punctuation
16. PDT	Predeterminer	40. ,	Comma
17. POS	Possessive ending	41. :	Colon, semi-colon
18. PRP	Personal pronoun	42. (	Left bracket character
19. PP$	Possessive pronoun	43.)	Right bracket character
20. RB	Adverb	44. "	Straight double quote
21. RBR	Adverb, comparative	45. ‘	Left open single quote
22. RBS	Adverb, superlative	46. “	Left open double quote
23. RP	Particle	47. ’	Right close single quote
24. SYM	Symbol (mathematical or scientific)	48. ”	Right close double quote

图 6－2　宾州树库词性标注的标签及其说明

Article: Endangered Species Act
Paragraph: "*... Other legislation followed, including the Migratory Bird Conservation Act of 1929, a 1937 treaty prohibiting the hunting of right and gray whales, and the Bald Eagle Protection Act of 1940. These later laws had a low cost to society—the species were relatively rare—and little opposition was raised.*"

Question 1: "*Which laws faced significant opposition?*"
Plausible Answer: *later laws*

Question 2: "*What was the name of the 1937 treaty?*"
Plausible Answer: *Bald Eagle Protection Act*

图 6－3　SQuAD 数据集样例

突出显示的部分是答案，每个问题的重要部分也进行了突出显示。一个好的自然语言处理模型需要根据问题对文本进行划分，这种划分可以通过序列标签的形式完成。模型将划分的起点和终点标记为答案的起点和终点部分。

到目前为止，读者学习了有关现代自然语言处理序列标记任务（如问题回答系统、命名实体识别和词性标注）的基础知识。在下一节中，读者将学习如何对 BERT 模型进行微调以适用于这些特定任务，并使用 datasets 库中的相关数据集。

6.3　微调语言模型以适用于命名实体识别任务

在本节中，读者将学习如何对 BERT 模型进行微调以适用于命名实体识别任务。首先从 datasets 库开始，加载 CONLL2003 数据集。

可以通过以下网址访问数据集卡片：https://huggingface. co/datasets/conll2003。图 6－4 所示的屏幕截图显示了 Hugging Face 网站上的 CONLL2003 数据集。

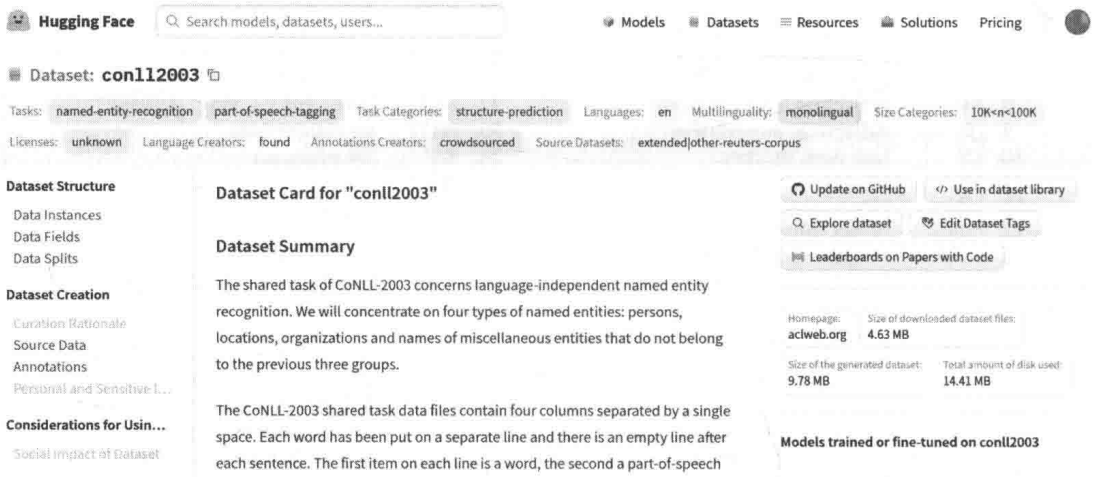


图 6－4　Hugging Face 网站上的 CONLL2003 数据集

从图 6－4 所示的屏幕截图可以看出，在这个数据集上训练的且当前可用的模型列举在右侧的面板中。网页上还包含数据集的描述，如数据集的大小和特征。

（1）为了加载数据集，请使用以下命令。

```
import datasets
conll2003 = datasets.load_dataset("conll2003")
```

在执行命令后，将出现下载进度条，当完成下载和缓存后，数据集就准备就绪。图 6－5 所示的屏幕截图显示了进度条。

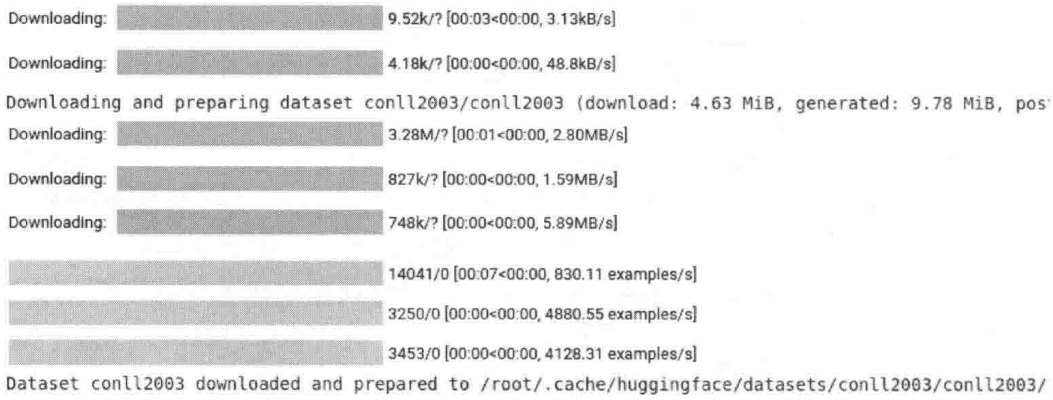


图 6－5　下载和准备数据集

（2）使用以下命令访问训练样本，以快速检查数据集。

```
>>> conll2003["train"][0]
```

输出结果如图 6－6 所示。

```
{'chunk_tags': [11, 21, 11, 12, 21, 22, 11, 12, 0],
 'id': '0',
 'ner_tags': [3, 0, 7, 0, 0, 0, 7, 0, 0],
 'pos_tags': [22, 42, 16, 21, 35, 37, 16, 21, 7],
 'tokens': ['EU',
  'rejects',
  'German',
  'call',
  'to',
  'boycott',
  'British',
  'lamb',
  '.']}
```

图 6－6　datasets 库中的 CONLL2003 训练样本

（3）词性标注和命名实体识别的相应标签显示在前面的屏幕截图中。此处仅使用命名实体识别标记。可以使用以下命令获取此数据集中可用的命名实体识别标记。

```
>>> conll2003["train"].features["ner_tags"]
```

（4）结果如图 6－7 所示。其中显示了所有的 BIO 标签，共有 9 个标签。

```
>>> Sequence(feature=ClassLabel(num_classes=9,
names=['O','B-PER','I-PER','B-ORG','I-ORG','B-LOC',
'I-LOC','B-MISC','I-MISC'], names_file=None, id=None),
length=-1, id=None)
```

（5）加载 BERT 分词器。

```
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained("bert-base-uncased")
```

（6）分词器类也可以处理空白标记化句子。需要启用分词器来处理空白标记化句子，因为命名实体识别任务对每个标记都有一个基于标记的标签。此任务中的标记通常是空白标记字，而不是字节对编码或任何其他分词器标记。根据上面的阐述，接下来看一看如何将分词器与空白标记化句子一起使用。

```
>>> tokenizer(["Oh","this","sentence","is","tokenized","and",
"splitted","by","spaces"], is_split_into_words=True)
```

正如所见，只需将 is_split_into_words 设置为 True，问题就解决了。

（7）在使用数据进行训练之前，需要对数据进行预处理。为此，必须使用以下函数并映射到整个数据集。

```
def tokenize_and_align_labels(examples):
    tokenized_inputs = tokenizer(examples["tokens"],
                truncation=True, is_split_into_words=True)
```

```
        labels = []
        for i, label in enumerate(examples["ner_tags"]):
            word_ids = tokenized_inputs.word_ids(batch_index=i)
            previous_word_idx = None
            label_ids = []
            for word_idx in word_ids:
                if word_idx is None:
                    label_ids.append(-100)
                elif word_idx != previous_word_idx:
                    label_ids.append(label[word_idx])
                else:
                    label_ids.append(label[word_idx] if label_all_tokens
                    else -100)
            previous_word_idx = word_idx
    labels.append(label_ids)
    tokenized_inputs["labels"] = labels
    return tokenized_inputs
```

（8）以上函数将确保标记和标签正确对齐。这种对齐是必需的，因为标记是分若干块进行标记的，但是单词必须位于一个块中。为了测试并查看此函数的工作方式，可以向该函数提供单个示例来运行函数。

```
q = tokenize_and_align_labels(conll2003['train'][4:5])
print(q)
```

结果如下所示。

```
>>> {'input_ids': [[101, 2762, 1005, 1055, 4387, 2000,
1996, 2647, 2586, 1005, 1055, 15651, 2837, 14121, 1062,
9328, 5804, 2056, 2006, 9317, 10390, 2323, 4965, 8351,
4168, 4017, 2013, 3032, 2060, 2084, 3725, 2127, 1996,
4045, 6040, 2001, 24509, 1012, 102]], 'token_type_ids':
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0]], 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1]], 'labels': [[-100, 5, 0,
-100, 0, 0, 0, 3, 4, 0, -100, 0, 0, 1, 2, -100, -100, 0,
0, 0, 0, 0, 0, 0, -100, -100, 0, 0, 0, 0, 5, 0, 0, 0, 0,
0, 0, 0, -100]]}
```

（9）但是结果的可读性很差，因此可以运行以下代码以获得可读性较好的版本。

```
for token, label in zip(tokenizer.convert_ids_to_tokens(q["input_ids"]
[0]),q["labels"][0]):
    print(f"{token:_<40} {label}")
```

结果如图 6－7 所示。

```
[CLS]                    -100
germany                  5
'                        0
s                        -100
representative           0
to                       0
the                      0
european                 3
union                    4
'                        0
s                        -100
veterinary               0
committee                0
werner                   1
z                        2
##wing                   -100
##mann                   -100
said                     0
on                       0
wednesday                0
consumers                0
should                   0
buy                      0
sheep                    0
##me                     -100
##at                     -100
from                     0
countries                0
other                    0
than                     0
britain                  5
until                    0
the                      0
scientific               0
advice                   0
was                      0
clearer                  0
.                        0
[SEP]                    -100
```

图 6－7　分词和对齐函数的运行结果

（10）使用 datasets 库的 map( ) 函数将此函数映射到数据集。

```
>>> tokenized_datasets = conll2003.map( \①
            tokenize_and_align_labels, batched = True)
```

（11）使用相应数量的标签加载 BERT 模型。

```
from transformers import ②AutoModelForTokenClassification
model = AutoModelForTokenClassification.from_
pretrained( "bert - base - uncased", num_labels = 9)
```

（12）模型已经被加载，并准备好接受训练。首先必须准备 Trainer 和 TrainingArguments。

```
from transformers import TrainingArguments, Trainer
args = TrainingArguments(
```

① 原著此处误用换行符，译者做了适当的调整。——译者注

② 原著此处误用换行符，译者做了适当的调整。——译者注

```
    "test-ner",
    evaluation_strategy = "epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
)
```

（13）需要准备数据校对器（data collator）。在训练数据集上应用批处理操作，以使用更少的内存并更快地执行。可以按以下方式进行操作。

```
from transformers import DataCollatorForTokenClassification
data_collator = DataCollatorForTokenClassification(tokenizer)
```

（14）为了能够评估模型性能，Hugging Face 的 datasets 库中为许多任务提供了可用的度量指标。此处使用命名实体识别的序列评估度量指标。seqeval 是一个很好的 Python 框架，用于评估序列标记算法和模型。首先必须安装 seqeval 库。

```
pip install seqeval
```

（15）加载度量指标。

```
>>> metric = datasets.load_metric("seqeval")
```

（16）使用以下代码，可以很容易地了解度量指标的工作原理。

```
example = conll2003['train'][0]
label_list = conll2003["train"]①.features["ner_tags"].feature.names
labels = [label_list[i] for i in example["ner_tags"]]
metric.compute(predictions=[labels], references=[labels])
```

输出结果如图 6－8 所示。

```
{'MISC': {'f1': 1.0, 'number': 2, 'precision': 1.0, 'recall': 1.0},
 'ORG': {'f1': 1.0, 'number': 1, 'precision': 1.0, 'recall': 1.0},
 'overall_accuracy': 1.0,
 'overall_f1': 1.0,
 'overall_precision': 1.0,
 'overall_recall': 1.0}
```

图 6－8　seqeval 度量指标的输出结果

可以为样本输入计算各种度量指标，如准确率、F1、精度和召回率。

（17）以下函数用于计算度量指标。

① 原著此处误用换行符，译者做了适当的调整。——译者注

```
import numpy as np
def ①compute_ metrics (p):
  predictions, labels = p
  predictions = np.argmax (predictions, axis =2)
  true_ predictions = [ [label_ list [p] for (p, l) in zip (prediction,
label) if l! = -100]
  for prediction, label in zip (predictions, labels) ]
  true_ labels = [ [label_ list [l] for (p, l) in zip (prediction, label) if l !
= -100]
  for prediction, label in zip (predictions, labels)]
  results = metric.compute (predictions = true_ predictions, references =
true_ labels)
  return {
    " precision": results [" overall_ precision"],
    " recall": results [" overall_ recall"],
    " f1": results [" overall_ f1"],
    " accuracy": results [" overall_ accuracy"],
}
```

（18）创建 Trainer 对象，并对其进行相应的训练。

```
trainer = Trainer(
    model,
    args,
    train_dataset = tokenized_datasets["train"],
    eval_dataset = tokenized_datasets["validation"],
    data_collator = data_collator,
    tokenizer = tokenizer,
    compute_metrics = compute_metrics
)
trainer.train()
```

（19）运行 Trainer 对象的 train( )方法之后，结果如图 6 -9 所示。

[2634/2634 20:09, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Precision	Recall	F1	Accuracy	Runtime	Samples Per Second
1	0.035800	0.043440	0.937072	0.944800	0.940920	0.988454	17.061700	190.486000
2	0.019100	0.043591	0.939359	0.951531	0.945406	0.989311	16.797100	193.486000
3	0.014500	0.043591	0.939359	0.951531	0.945406	0.989311	16.790100	193.567000

图 6 -9　运行 train( )方法之后的 Trainer 结果

（20）训练之后需要保存模型和分词器。

```
model.save_pretrained("ner_model")
```

① 原著此处有误，def( )函数定义必须换行。——译者注

```
tokenizer.save_pretrained("tokenizer")
```

（21）如果希望将模型与管道一起使用，则必须读取配置文件，并根据在 label_list 对象中使用的标签正确分配 label2id 和 id2label。

```
id2label = {
     str(i): label for i,label in enumerate(label_list)
}
label2id = {
     label: str(i) for i,label in enumerate(label_list)
}
import json
config = json.load(open("ner_model/config.json"))
config["id2label"] = id2label
config["label2id"] = label2id
json.dump(config, open("ner_model/config.json","w"))
```

（22）使用模型，示例代码如下所示。

```
from transformers import pipeline
model = AutoModelForTokenClassification.from_pretrained("ner_model")①
nlp = pipeline("ner", model =mmodel, tokenizer =tokenizer)
example = "I live in Istanbul"
ner_results = nlp(example)
print(ner_results)
```

结果如下所示。

```
[{'entity':'B-LOC','score':0.9983942,'index':4,
'word':'istanbul','start':10,'end':18}]
```

到目前为止，本节介绍了如何使用 BERT 实现词性标注，阐述了如何使用 Transformer 训练自己的词性标记模型，并且测试了该模型。下一节将重点讨论问题回答系统。

6.4 基于标记分类的问题回答系统

问题回答系统问题通常被定义为自然语言处理问题，为人工智能提供给定的文本和问题，并得到与问题相关的回复。通常，这个答案可以在原文中找到，解决这个问题有不同的方法。在视觉问题回答（Visual Question Answering，VQA）系统中，问题是关于视觉实体或视觉概念，而不是文本，但问题本身是文本形式。

① 原著此处误用换行符，译者做了适当的调整。——译者注

视觉问题回答系统示例如图 6 – 10 所示。

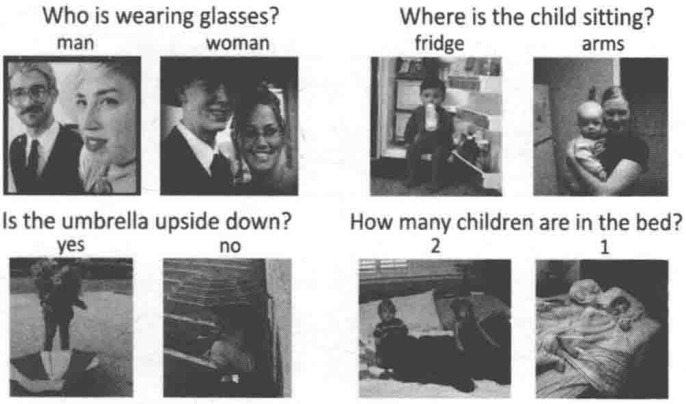


图 6 – 10　视觉问题回答系统示例

视觉问题回答系统中使用的大多数模型都是多模态模型，可以理解问题的视觉背景并正确生成答案。然而，单模态完全文本问题回答系统或问题回答系统仅基于文本上下文和文本问题，并有各自的文本答案。

（1）SQuAD 是问题回答领域最著名的数据集之一。为了查看 SQuAD 数据集的样例并检查这些样例，可以使用以下代码。

```
from pprint import pprint
from datasets import load_dataset
squad = load_dataset("squad")
for item in squad["train"][1].items():
    print(item[0])
    pprint(item[1])
    print("="*20)
```

结果如下所示。

```
answers
{'answer_start': [188], 'text': ['a copper statue of
Christ']}
====================
Context
('Architecturally, the school has a Catholic character.
Atop the Main' "Building's gold dome is a golden statue
of the Virgin Mary. Immediately in "'front of the Main
Building and facing it, is a copper statue of Christ with
''arms upraised with the legend "Venite Ad Me Omnes".
Next to the Main''Building is the Basilica of the
Sacred Heart. Immediately behind the''basilica is the
Grotto, a Marian place of prayer and reflection. It is
a''replica of the grotto at Lourdes, France where the
```

```
Virgin Mary reputedly''appeared to Saint Bernadette
Soubirous in 1858. At the end of the main drive''(and
in a direct line that connects through 3 statues and
the Gold Dome), is''a simple, modern stone statue of
Mary.')
 ====================
Id
'5733be284776f4190066117f'
 ====================
Question
'What is in front of the Notre Dame Main Building?'
 ====================
Title
'University_of_Notre_Dame'
 ====================
```

然而，存在第二版的 SQuAD 数据集，其中包含更多的训练样本，强烈建议使用第二版的 SQuAD 数据集。为了全面了解如何为问题回答系统问题训练模型，本节重点关注此问题的当前部分。

（2）导入第二版的 SQuAD 数据集，代码如下所示。

```
from datasets import load_dataset
squad = load_dataset("squad_v2")
```

（3）加载 SQuAD 数据集后，可以使用以下代码查看此数据集的详细信息。

```
>>> squad
```

结果如图 6－11 所示。

```
DatasetDict({
    train: Dataset({
        features: ['id', 'title', 'context', 'question', 'answers'],
        num_rows: 130319
    })
    validation: Dataset({
        features: ['id', 'title', 'context', 'question', 'answers'],
        num_rows: 11873
    })
})
```

图 6－11　SQuAD 数据集（第二版）的详细信息

在图 6－11 中，SQuAD 数据集包含 130 000 多个训练样本和 11 000 多个验证样本。

（4）与命名实体识别的训练过程相同，必须对数据进行预处理，以获得模型使用的正确形式。为此，必须首先加载分词器，这是一个预训练的分词器，并且使用的是预训练模型，同时希望针对问题回答系统问题对其进行微调。

```
from transformers import AutoTokenizer
model = "distilbert-base-uncased"
```

```
tokenizer = AutoTokenizer.from_pretrained(model)
```

如上面的代码所示，此处使用 DistilBERT 模型。

根据 SQuAD 样例，需要为模型提供多个文本，一个用于问题，一个用于上下文。因此，需要分词器将这两个标记并排放置，并使用特殊的［SEP］标记将这两个标记分开，因为 DistilBERT 是基于 BERT 的模型。

问题回答系统范围中还存在另一个问题，即上下文的大小。上下文的大小可以超过模型输入的大小，但不能将其减小到模型所能接收的大小。对于其他问题，用户可以这样做，但是在问题回答系统中，答案可能位于截断部分。本节将展示一个使用文档步幅（document stride）解决此问题的示例。

（5）下面是一个演示如何使用分词器的示例。

```
max_length = 384
doc_stride = 128
example = squad["train"][173]
tokenized_example = tokenizer(
    example["question"],
    example["context"],
    max_length = max_length,
    truncation = "only_second",
    return_overflowing_tokens = True,
    stride = doc_stride
)
```

（6）stride 是用于返回第二部分步幅的文档步幅，类似窗口，而 return_overflowing_tokens 指示模型是否应返回额外标记。tokenized_example 的结果不仅是一个标记化的输出，还有两个输入 ID。在以下内容中，可以看到如下结果。

```
>>> len(tokenized_example['input_ids'])
>>> 2
```

（7）运行以下 for 循环代码，可以看到完整的结果。

```
for input_ids in tokenized_example["input_ids"][:2]:
    print(tokenizer.decode(input_ids))
    print("-"*50)
```

结果如下所示。

```
[CLS] beyonce got married in 2008 to whom? [SEP] on april
4, 2008, beyonce married jay z. she publicly revealed
their marriage in a video montage at the listening party
for her third studio album, i am... sasha fierce, in
```

```
manhattan's sony club on october 22, 2008. i am... sasha
fierce was released on november 18, 2008 in the united
states. the album formally introduces beyonce's alter
ego sasha fierce, conceived during the making of her
2003 single " crazy in love ", selling 482, 000 copies
in its first week, debuting atop the billboard 200, and
giving beyonce her third consecutive number - one album
in the us. the album featured the number - one song "
single ladies ( put a ring on it ) " and the top - five
songs " if i were a boy " and " halo ". achieving the
accomplishment of becoming her longest - running hot
100 single in her career, " halo " 's success in the us
helped beyonce attain more top - ten singles on the list
than any other woman during the 2000s. it also included
the successful " sweet dreams ", and singles " diva ",
" ego ", " broken - hearted girl " and " video phone ".
the music video for " single ladies " has been parodied
and imitated around the world, spawning the " first
major dance craze " of the internet age according to the
toronto star. the video has won several awards, including
best video at the 2009 mtv europe music awards, the 2009
scottish mobo awards, and the 2009 bet awards. at the
2009 mtv video music awards, the video was nominated for
nine awards, ultimately winning three including video
of the year. its failure to win the best female video
category, which went to american country pop singer
taylor swift's " you belong with me ", led to kanye west
interrupting the ceremony and beyonce [SEP]
--------------------------------------------------------
[CLS] beyonce got married in 2008 to whom? [SEP] single
ladies " has been parodied and imitated around the world,
spawning the " first major dance craze " of the internet
age according to the toronto star. the video has won
several awards, including best video at the 2009 mtv
europe music awards, the 2009 scottish mobo awards, and
the 2009 bet awards. at the 2009 mtv video music awards,
the video was nominated for nine awards, ultimately
winning three including video of the year. its failure
to win the best female video category, which went to
american country pop singer taylor swift's " you belong
with me ", led to kanye west interrupting the ceremony
and beyonce improvising a re - presentation of swift's
award during her own acceptance speech. in march 2009,
beyonce embarked on the i am... world tour, her second
headlining worldwide concert tour, consisting of 108
shows, grossing $ 119. 5 million. [SEP]
--------------------------------------------------------
```

从上面的输出中可以看出，在 128 个标记窗口中，上下文的其余部分将在输入 ID 的第二个输出中再次被复制。

另一个问题是结束跨度的值，该值在数据集中不存在，但给出了答案的开始跨度或开始字符。很容易找到答案的长度并将其添加到开始跨度，这将自动生成结束跨度。

（8）现在已经了解了数据集的所有细节以及如何处理数据集，因此可以很容易地将以上内容组合在一起，形成一个预处理函数。

```
def prepare_train_features(examples):
    # 分词器示例
    tokenized_examples = tokenizer(
        examples["question" if pad_on_right else "context"],
        examples["context" if pad_on_right else "question"],
        truncation = "only_second" if pad_on_right else "only_first",
                max_length = max_length,
                stride = doc_stride,
                return_overflowing_tokens = True,
                return_offsets_mapping = True,
                padding = "max_length",
        )
    # 从特征到其样本的映射
    sample_mapping = \①
        tokenized_examples.pop("overflow_to_sample_mapping")
    offset_mapping = tokenized_examples.pop("offset_ mapping")②
    tokenized_examples["start_positions"] = []
    tokenized_examples["end_positions"] = []
    # 将不可能的答案标记为 CLS
    # 开始标记和结束标记之间的内容是每个问题的答案
    for i, offsets in enumerate(offset_mapping):
        input_ids = tokenized_examples["input_ids"][i]
        input_ids = tokenized_examples["input_ids"][i]
        cls_index = ③input_ids.index(tokenizer.cls_token_id)
        sequence_ids = ④tokenized_examples.sequence_ids(i)
        sample_index = sample_mapping[i]
        answers = examples["answers"][sample_index]
        if len(answers["answer_start"]) = = 0:
            tokenized_examples["start_positions"].append(cls_index)
            tokenized_examples["end_positions"].append(cls_index)
        else:
            start_char = answers["answer_start"][0]
            end_char = start_char + len(answers["text"][0])
```

① 原著此处误用换行符，译者做了适当的调整。——译者注
② 原著此处误用换行符，译者做了适当的调整。——译者注
③ 原著此处误用换行符，译者做了适当的调整。——译者注
④ 原著此处误用换行符，译者做了适当的调整。——译者注

```
            token_start_index = 0
            while sequence_ids[token_start_index]! = (①1 if pad_on_right
            else 0):
                token_start_index + = 1
            token_end_index = len(input_ids) - 1
            while sequence_ids[token_end_index]! = (1 if pad_on_right
            else 0):
                token_end_index - = 1
            if not (offsets[token_start_index][0] < =
                start_char and offsets[token_end_index][1] >= end_
                char):
                tokenized_examples["start_positions"].append(cls_index)
                tokenized_examples["end_positions"].append(cls_index)
            else:
                while token_start_index < len(offsets) and \②
                    offsets[token_start_index][0] <= start_char:
                    token_start_index + = 1
                tokenized_examples["start_positions"].append(token_
                start_index - 1)
                while offsets[token_end_index][1] >= end_char:
                    token_end_index - = 1
                tokenized_examples["end_positions"].append(token_end_
                index + 1)
        return tokenized_examples③
```

（9）将此函数映射到数据集，以应用所有必需的更改。

```
>>> tokenized_datasets = squad.map(prepare_train_features,
        batched = True, remove_columns = squad["train"].column_names)
```

（10）与其他示例一样，现在可以加载需要进行微调的预训练模型。

```
from transformers import AutoModelForQuestionAnswering, TrainingArguments,
Trainer
model = AutoModelForQuestionAnswering.from_pretrained(model)
```

（11）创建训练参数。

```
args = TrainingArguments(
    "test - squad",
    evaluation_strategy = "epoch",
```

① 原著此处代码有误，译者查阅了源代码后做了修正——译者注

② 此处需要增加换行符。建议读者参阅源代码，这样比较清楚明了。——译者注

③ 原著此处代码缩进有误，译者参阅源代码做了修正。这段代码比较长，因此存在较多的排版错误。对于比较长的代码，建议读者参阅源代码，这样比较清楚明了。——译者注