第5章

Transformer与人机畅聊

当读完本章时，应该能够：

- 理解机器是如何学习说话和学会说话的。
- 理解机器问答的技术路线。
- 理解机器问答语言模型及评价方法。
- 理解 Transformer 模型的原理与方法。
- 理解 Transformer 独特的注意力机制。
- 理解 Transformer 模型的定义与训练。
- 实战基于腾讯聊天数据集＋Transformer 的聊天机器人设计。
- 实战基于 Web API 的聊天服务器设计。
- 实战 Android 版人机畅聊客户机设计。

5.1 项目动力

机器视觉与自然语言处理是人工智能的两大热点领域。对于机器视觉学习，读者较容易入门，但对于自然语言的处理，特别是对于机器翻译、机器问答、人机聊天等应用的理解与学习，难度较高。究其原因，读者往往很难独立去完成一个上述自然语言处理项目，一旦缺少了亲力亲为的实战体验，理论不能与实践紧密结合，不能相互印证，那么对理论的掌握往往就是空中楼阁。

本章案例将带领读者从零起步，以 Transformer 为建模基础，在腾讯发布的中文聊天数据集上，训练出一款会聊天的机器人程序。聊天机器人除了不知疲倦、有求必应、一对多并发服务的优点外，聊天机器人的未来将兼具多种风格，例如滔滔不绝的机器人、激情演讲的机器人、能做灵魂触碰的机器人、能心有灵犀的机器人、能聊出思想火花的机器人、

能与你一起头脑风暴的机器人、能自我学习提高的机器人……

总之,具备强大语言能力的机器人、具备思想能力的机器人,正沿着人类追求的航向,奋力前行。

5.2 机器问答技术路线

机器问答与机器聊天(本书特指人机聊天)都是自然语言处理领域的经典问题,二者并不完全相同。

机器问答一般常见于阅读理解的场景,给定参考语料,要求机器人根据语料回答问题,问题的答案往往是在语料之中做了显式陈述,或者根据语料可以推断的。

对于机器聊天而言,聊天的随机性决定了应答难度更大一些,不但需要及时切换话题,还需要通过简短的交谈揣摩对方的真实意图,否则会答非所问。

机器问答与机器聊天的界限有时也是模糊的,例如,电商平台部署的客服机器人,确实可以解决客户的若干问题,节省了大规模的人力资源。这些客服机器人基于其收集的大量客服会话数据训练而成,兼具机器问答与机器聊天的特点。

经常听到新入门学生询问如何开发一个机器问答或者机器聊天的项目,图 5.1 给出了项目实施路径的四部曲。

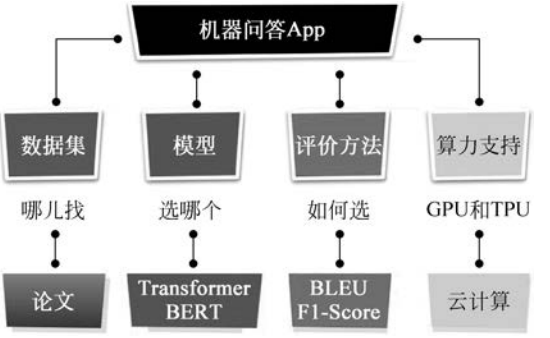

图 5.1 机器问答技术路线

第一步,解决数据集的问题。俗话说,巧妇难为无米之炊,强大的语料数据集对语言建模至关重要。基本原则是根据问题目标,选择适配的语料库。例如斯坦福大学发布的 CoQA、SQuAD 2.0 语料库,华盛顿大学牵头发布的 QuAC 语料库,百度发布的 DuConv 语料库,京东的 JDDC 客服语料库,腾讯的 NaturalConv 聊天语料库,清华大学的 KdConv 语料库等。这些经典的语料库都有论文解析,通过阅读论文可以得知语料库的规模、覆盖的领域、适合解决的问题。如果条件允许,可以考虑自建语料库。或者自建小规模语料库,用迁移学习的方法解决数据量不足的问题。

第二步,选择模型(算法)。工欲善其事,必先利其器。近年来,能够代表自然语言处理领域前沿热点的模型非 Transformer 和 BERT 莫属。研读前沿文献不难发现,活跃在各大语言建模挑战赛排行榜前列的模型,无不是以 Transformer 和 BERT 结构为核心的衍生品。图 5.2 给出了适合自然语言建模的经典技术,自底向上,反映了语言建模技术的

变迁与演进。

第三步，理解语言模型的评价方法。之所以强调这一点，是因为学生往往比较熟悉机器视觉领域的分类或回归问题的评价方法，这些方法在自然语言处理领域并不好用。以机器聊天或机器推理为例，可接受的正确答案往往不止一个。如何去评价这些语句长度不一、文字不一的句子，需要新方法。例如 BLEU 即是一种经典方法。

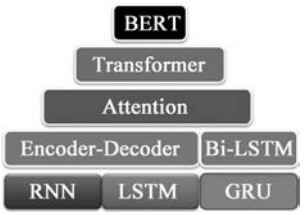

图 5.2 语言建模技术的演进

第四步，解决算力问题。普通的个人计算机，即使对于小规模数据集，也难以胜任模型训练。GPU 性能稍好的工作站，往往也难以满足中等建模问题对算力的需求。对于学生而言，需要寻找免费算力资源，才能不让自己的想法卡在最后一公里。幸运的是，百度、Google 等都提供了免费算力平台。

以本章的机器人聊天项目为例，在 3.7 节给出的计算机配置上，完成 38 个 epoch 的训练，需要 4 小时左右。在 Kaggle 平台提供的免费 TPUv3-8 上，只需要 12 分钟。速度提升 20 倍。

5.3 腾讯聊天数据集

2021 年，腾讯人工智能实验室发布了主题驱动的多轮中文聊天数据集，数据集名称为 NaturalConv，详情参见论文 *Naturalconv: A chinese dialogue dataset towards multi-turn topic-driven conversation*（WANG，X Y，LI C，ZHAO J，et al. 2021）。

NaturalConv 包含体育、娱乐、科技、游戏、教育和健康六个领域的 19.9K 场双人对话，合计 40 万条对话语句，每场对话平均包含 20.1 个句子，即对话双方进行了 10 次左右的表达。为了获取上述数据，腾讯支付了语料库整理人员 5 万美元的信息采集和加工整理费用。现在这个数据集可以在腾讯人工智能实验室官方网站免费下载。

NaturalConv 数据集包含的对话是以 6500 篇网络新闻为背景展开的。对话具备自然性，即对话双方围绕一个共同主题展开话语交流，属于自然情况下的聊天行为。

口语化是聊天的显著特点，这也是 NaturalConv 数据集设计时遵循的原则。表 5.1 给出了 NaturalConv 中的一场对话文本，A、B 两名学生关于当天的一场荷甲球赛展开的话题聊天。

表 5.1 A、B 两人对话样本抽样观察

轮　次	A、B 两人的对话内容
A-1	嗨，你来得挺早啊。
B-1	是啊，你怎么来得这么晚？
A-2	昨晚我看了球赛，所以今早起晚了，也没吃饭。
B-2	现在这个点食堂应该有饭，你看什么球赛啊？篮球吗？
A-3	不是，足球。
B-3	怪不得，足球时间长。
A-4	你知道吗？每次都是普罗梅斯进球。
B-4	这个我刚才也看了新闻了，他好有实力啊。

续表

轮　次	A、B 两人的对话内容
A-5	是啊，尤其是他那个帽子戏法，让我看得太惊心动魄了。
B-5	我一个同学在群里说了，每次聊天都离不开他，可见他的实力有多强大。
A-6	是啊，看来你那个同学和我是一样的想法。
B-6	我好不容易摆脱他的话题，你又来说出他的名字。
A-7	哈哈，你不懂我们对足球有多热爱。
B-7	我知道你热爱，我还记得你参加初中比赛还拿到冠军呢。你功不可没啊。
A-8	哈哈，还是你能记得我当时的辉煌。
B-8	没办法，咱俩从小一起长大的，彼此太了解了。
A-9	嗯，老师来了。
B-9	快打开课本，老师要检查。
A-10	嗯嗯，下课再聊。
B-10	嗯。

话题参照的新闻稿件如下。

新闻稿：北京时间今天凌晨，荷甲第 4 轮进行了两场补赛。阿贾克斯和埃因霍温均在主场取得了胜利，两队 7 轮后同积 17 分，阿贾克斯以 6 个净胜球的优势领跑积分榜。0 点 30 分，埃因霍温与格罗宁根的比赛开战，埃因霍温最终 3∶1 在主场获胜。2 点 45 分，阿贾克斯主场与福图纳锡塔德之战开始。由于埃因霍温已经先获胜了，阿贾克斯必须获胜才能在积分榜上咬住对方。在整个上半场，阿贾克斯得势不得分，双方 0∶0 互交白卷。在下半场中，阿贾克斯突然迎来了大爆发。在短短 33 分钟内，阿贾克斯疯狂打进 5 球，平均每 6 分钟就能取得 1 个进球。在第 50 分钟时，新援普罗梅斯为阿贾克斯打破僵局。塔迪奇左侧送出横传，普罗梅斯后点推射破门。第 53 分钟时亨特拉尔头球补射，内雷斯在门线前头球接力破门。第 68 分钟时，普罗梅斯近距离补射梅开二度。这名 27 岁的前场多面手，跑到场边来了一番尬舞。第 77 分钟时阿贾克斯收获第 4 球，客队后卫哈里斯在防传中时伸腿将球一捅，结果皮球恰好越过门将滚入网窝。在第 83 分钟时，普罗梅斯上演了帽子戏法，比分也最终被定格为 5∶0。在接到塔迪奇直传后，普罗梅斯禁区左侧反越位成功，他的单刀低射从门将裆下入网。普罗梅斯这次的庆祝动作是秀出三根手指，不过他手指从上到下抹过面部时的动作，很有点像是在擦鼻涕。

NaturalConv 数据集与 CMU DoG、Wizard of Wiki、DuConv、KdConv 的对比如表 5.2 所示。NaturalConv 涉及的主题更为广泛，双方对话的回合数更多，语料库规模更大。

表 5.2　与其他数据集对比

Dataset	Language	Document Type	Annotation Level	Topic	Avg. # turns	# uttrs
CMU DoG	English	Text	Sentence	Film	22.6	130k
Wizard of Wiki	English	Text	Sentence	Multiple	9.0	202k
DuConv	Chinese	Text&KG	Dialogue	Film	5.8	91k
KdConv	Chinese	Text&KG	Sentence	Film, Music, Travel	19.0	86k
NaturalConv	**Chinese**	**Text**	**Dialogue**	**Sports, Ent, Tech, Games, Edu, Health**	**20.1**	**400k**

NaturalConv语料库中各主题的分布情况如表5.3所示。可以看到，体育类主题的文档数占比接近一半，健康类主题最少，只有52篇背景文档。

表5.3 主题的分布情况

	Sports	Ent	Tech	Games	Edu	Health	Total
# document	3124	1331	1476	103	414	52	6500
# dialogues	9740	4403	4061	308	1265	142	19 919
# dialogues per document	3.1	3.3	2.8	3.0	3.1	2.7	3.0
# utterances	195 643	88 457	81 587	6180	25 376	2852	400 095
Avg. # utterances per dialogue	20.1	20.1	20.1	20.1	20.1	20.1	20.1
Avg. # tokens per utterance	12.0	12.4	12.3	12.1	12.6	12.5	12.2
Avg. # characters per utterance	17.8	18.1	18.6	17.8	18.1	18.3	18.1
Avg. # tokens per dialogue	241.1	248.2	247.5	242.9	248.3	251.1	244.8
Avg. # characters per dialogue	357.5	363.2	372.8	356.5	356.5	368.0	363.1

每篇文档相关的对话场次平均为3场。每场对话包含的语句平均为20.1个。平均每场对话包含的字数为360个左右。

可见，NaturalConv语料库的主题分布并不均衡，这也可以解释，为什么后面训练的聊天机器人，偏爱于体育类的话题，或者对体育类的话题更“健谈”一些。语料库相当于语言模型的先天基因，决定了语言模型表现的倾向性。

5.4 Transformer模型解析

Transformer模型参见论文 *Attention is all you need*（VASWANI A，SHAZEER N，PARMAR N，et al. 2017）。Transformer这个词的原意是“变形金刚”，作者用Transformer寓意其可伸缩性好，可以胜任的应用领域非常广泛。

Transformer模型结构如图5.3所示。左侧代表编码器，右侧代表解码器。编码器与解码器的结构非常相似。

编码器由 N 个结构重复的单元连接而成。每个单元包含两个残差块：第一个残差块以多头注意力模块为核心；第二个残差块以全连接网络为核心。

解码器的主体是由 N 个结构重复的单元连接而成，最后跟上一个全连接层和Softmax分类层。每个单元包含三个残差块：第一个残差块以带掩码的多头注意力模块为核心；第二个残差块是交叉多头注意力模块；第三个模块为全连接网络模块。

注意，Transformer中用Linear表示的全连接网络，不包含激活函数，是一个线性结构。

编码器的输入层需要将序列的嵌入向量与序列的位置编码向量做叠加运算，然后并行输入到三个Linear网络，得到 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 三个输入向量。

编码器输出层的输出将直接给到解码器各个单元的第二个残差块。

解码器除了接收来自编码器的输入，还有一个被称作Output Embedding的输入。在模型训练期间，Output Embedding用样本的标签向量表示。在模型推理时，解码器是

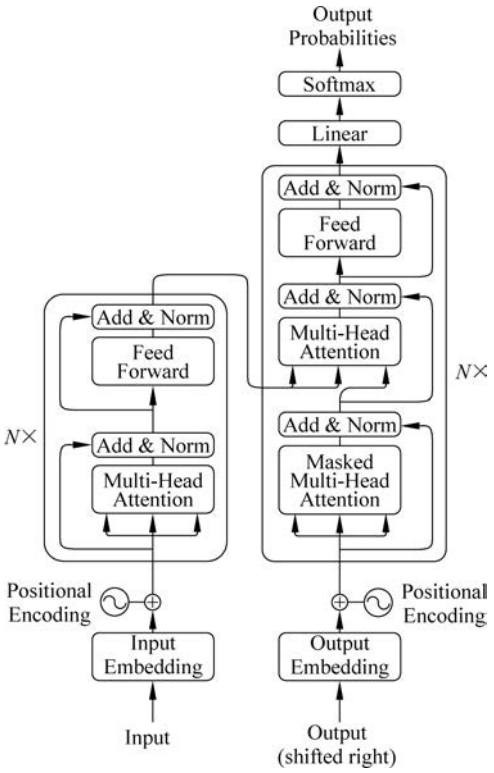

图 5.3 Transformer 模型结构(见彩插)

一个自回归结构,单步推理生成的输出,将回馈给 Output Embedding 作为解码器的输入,参入整个推理过程。

编码器与解码器之间的连接方式有很多,图 5.4 给出了 Transformer 的经典推荐方式。即编码器最后一个单元的输出,给到解码器各个单元的交叉多头注意力模块。

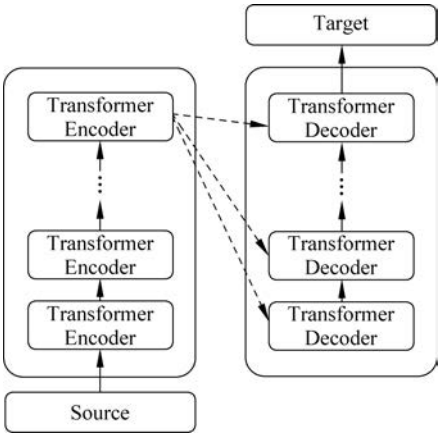

图 5.4 Transformer 的经典推荐方式

事实上,也可以让编码器各个单元的输出,只给到解码器的同层次单元,或者给到解

码器的所有单元。

Transformer 的核心运算体现在注意力机制上，图 5.5 给出了 Transformer 的单头注意力机制与多头注意力机制的计算逻辑。

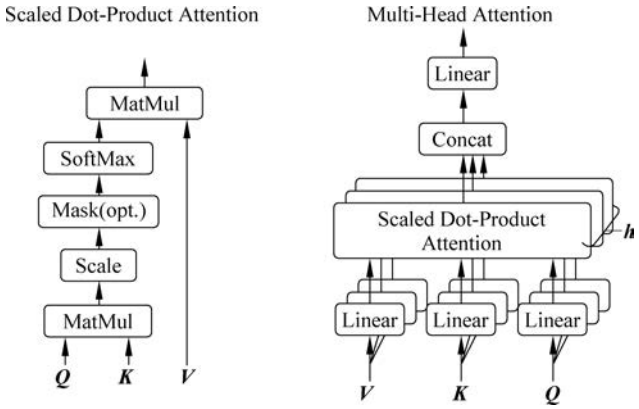

图 5.5 Transformer 的单头注意力机制与多头注意力机制的计算逻辑

Transformer 将编码的输入序列，通过全连接网络学习为 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 三个嵌入向量。左图为基于点积的单头注意力逻辑。右图为 h 个多头注意力并行堆叠、合并计算的逻辑。

$\boldsymbol{Q}$、$\boldsymbol{K}$ 之间完成注意力计算，形成注意力权重，用注意力权重乘以 $\boldsymbol{V}$，得到单个注意力模块的输出。

直观上看，$\boldsymbol{Q}$、$\boldsymbol{K}$ 之间的注意力强度反映了序列中上下文之间的关系强弱，将这种强弱关系映射到向量 $\boldsymbol{V}$ 上，即可实现对输入序列的编码和特征提取。$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 之间的计算逻辑与互动关系如图 5.6 所示。

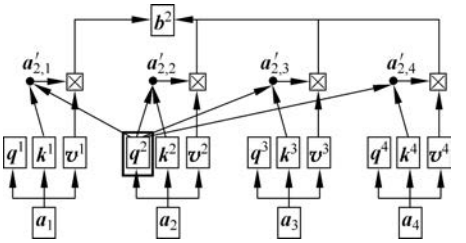

图 5.6 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 之间的计算逻辑与互动关系

假设输入序列为 $\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_3\boldsymbol{a}_4$，现在只考虑 $\boldsymbol{a}_2$ 对应的输出 $\boldsymbol{b}^2$，通过注意力计算得到 $\boldsymbol{b}^2$ 的步骤解析如下。

(1) $\boldsymbol{a}_2$ 通过三个独立的全连接网络学习，得到编码 q^2、k^2、v^2，q^2 与 k^2 按照图 5.5 所示的点积注意力计算逻辑，经 Softmax 输出 q^2 与 k^2 之间的归一化关系权重向量 $\boldsymbol{a}'_{2,2}$。

(2) q^2 与 k^1 经注意力计算，Softmax 层输出 q^2 与 k^1 的归一化关系权重向量 $\boldsymbol{a}'_{2,1}$。

(3) 重复步骤(2)，得到 $\boldsymbol{a}'_{2,3}$ 和 $\boldsymbol{a}'_{2,4}$。

至此，$\boldsymbol{a}_2$ 与序列 $\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_3\boldsymbol{a}_4$ 中其他单词的关系(包括与自身的关系)，已经通过 q^2 与 k^1、k^2、k^3、k^4 之间的注意力计算得到，表示为四个权重向量 $\boldsymbol{a}'_{2,1}$、$\boldsymbol{a}'_{2,2}$、$\boldsymbol{a}'_{2,3}$ 和 $\boldsymbol{a}'_{2,4}$。

(4) 得到 $\boldsymbol{b}^2=\boldsymbol{a}'_{2,1}\times v^1+\boldsymbol{a}'_{2,2}\times v^2+\boldsymbol{a}'_{2,3}\times v^3+\boldsymbol{a}'_{2,4}\times v^4$。现在,可以认为向量 $\boldsymbol{b}^2$ 是对向量 $\boldsymbol{a}_2$ 施加上下文全局注意力后的新表示。

同样的方法可以得到 $\boldsymbol{b}^1$、$\boldsymbol{b}^3$、$\boldsymbol{b}^4$。

从 $\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_3\boldsymbol{a}_4$ 到 $\boldsymbol{b}^1\boldsymbol{b}^2\boldsymbol{b}^3\boldsymbol{b}^4$,依靠注意力机制,完成了一次特征提取与变换。

下面通过一个例子演示注意力的计算过程。如图 5.7 所示(图片源自 Ketan Doshi 博客),假设输入的序列为 You are welcome,规定序列长度为 4,所以后面需要补一个空位,不妨用 PAD 表示。假定每个单词向量的编码长度为 3。

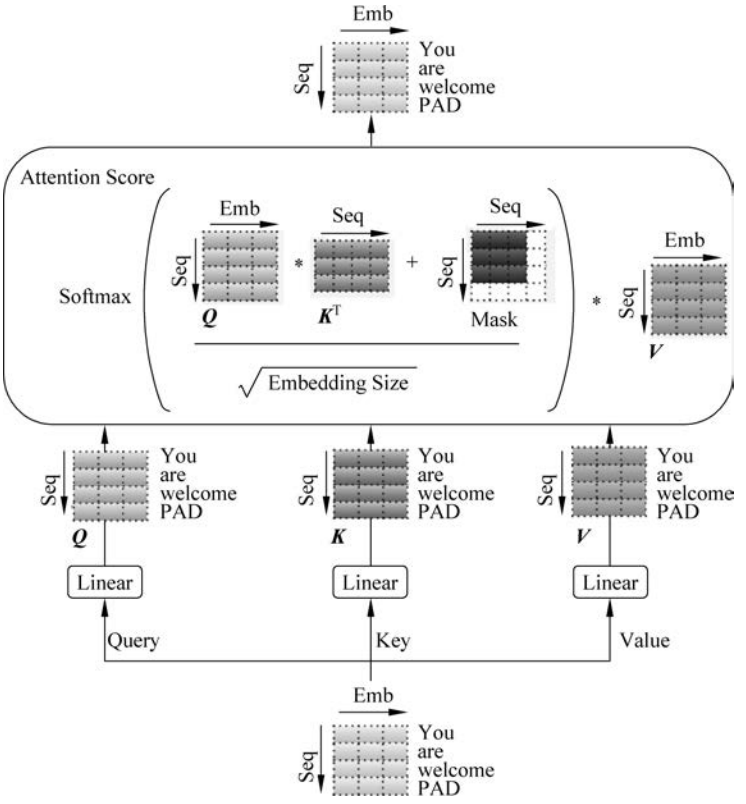

图 5.7 注意力计算举例

You are welcome PAD 构成了一个维度为(4,3)的特征矩阵,分别送入三个 Linear 网络,得到 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 三个特征编码矩阵,维度均为(4,3)。

注意力的计算可以归纳为式(5.1)。

$$\text{Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})=\text{Softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{5.1}$$

其中,d_k 表示 $\boldsymbol{Q}$、$\boldsymbol{K}$ 序列中单词向量的长度。考虑到 $\boldsymbol{Q}$ 与 $\boldsymbol{K}$ 的点积运算,有可能放大了特征的输出值,所以分母除以 $\sqrt{d_k}$,这也是 Scaled Dot-Product Attention 名称的由来。

Transformer 的相关参数设置如表 5.4 所示。

表 5.4 Transformer 的相关参数设置

参 数 名 称	参 数 值
编码器单元数 N	6
解码器单元数 N	6
输入输出向量的长度 d_{model}	512
注意力头数 h	8
$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 的向量长度	$d_k = d_v = d_{\text{model}}/h = 64$

5.5 机器人项目初始化

用 PyCharm 在 TensorFlow_to_Android 项目下新建目录 MyRobot。在 MyRobot 目录下新建 models 和 transformer 两个子目录。

models 目录用于存放训练好的聊天机器人模型。transformer 目录用于存放数据集预处理程序和模型定义程序。

在 transformer 目录下新建子目录 dataset。将下载的腾讯自然语言聊天数据集 NaturalConv 解压后存放到 dataset 目录下。

数据集可在腾讯人工智能实验室官方网站免费下载。下载地址为 https://ai.tencent.com/ailab/nlp/dialogue/#datasets。

在 MyRobot 目录下新建主程序 main.py，负责模型的训练、保存、评估和测试。

初始化后的项目结构如图 5.8 所示。

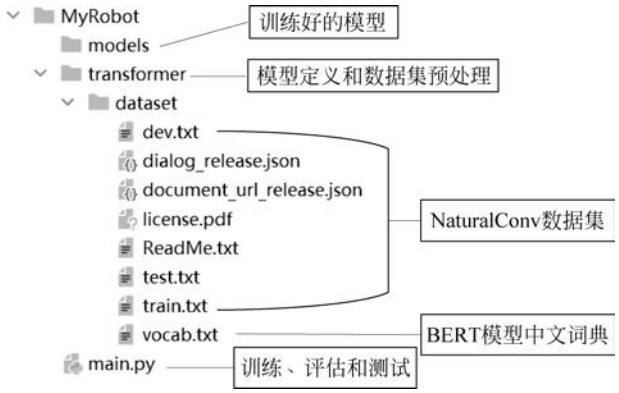

图 5.8 MyRobot 项目初始结构

dataset 目录下的 vocab.txt 是 BERT 中文模型的词典文件。这个文件不属于腾讯自然语言聊天数据集，需要从 BERT 官方网站下载中文模型，从中抽取中文词典文件。

BERT 中文预训练模型可从官方网站下载。下载地址为 https://github.com/google-research/bert。

本章案例采用 BERT 的分词模型对中文语句分词。

5.6 数据集预处理与划分

模型训练之前,需要首先准备好数据,让数据能够直接“喂入”模型进行训练,为了提高模型“喂入”的效率,往往还要设计数据加载模式。

在 transformer 目录下新建 dataset.py 程序,如程序源码 P5.1 所示。

程序源码 P5.1 dataset.py 数据集预处理与划分

```
import codecs
import json
import re
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_sequences
# 析取数据集(训练集、验证集、测试集),将所有的"问"与"答"分开
def extract_conversations(hparams, data_list, dialog_list):
  inputs, outputs = [], []                          # 问答列表
  for dialog in dialog_list:
    if dialog['dialog_id'] in data_list:
      if len(dialog['content']) % 2 == 0:
        i = 0
        for line in dialog['content']:
          if (i % 2 == 0):
            inputs.append(line)                     # "问"列表
          else:
            outputs.append(line)                    # "答"列表
          i += 1
          # 限定样本总数
          # if len(inputs) >= hparams.total_samples:
              # return inputs, outputs
  return inputs, outputs
# 分词,过滤掉超过长度的句子,短句补齐
def tokenize_and_filter(hparams, inputs, outputs, tokenizer):
    tokenized_inputs, tokenized_outputs = [], []
    for (sentence1, sentence2) in zip(inputs, outputs):
        sentence1 = tokenizer.tokenize(sentence1)                    # 分词
        sentence1 = tokenizer.convert_tokens_to_ids(sentence1)       # ids
        sentence2 = tokenizer.tokenize(sentence2)
        sentence2 = tokenizer.convert_tokens_to_ids(sentence2)
        sentence1 = hparams.start_token + sentence1 + hparams.end_token
        sentence2 = hparams.start_token + sentence2 + hparams.end_token
        if len(sentence1) <= hparams.max_length and len(sentence2) <= hparams.max_
        length:
            tokenized_inputs.append(sentence1)
            tokenized_outputs.append(sentence2)
    # 补齐
    tokenized_inputs = pad_sequences(tokenized_inputs, \
                        maxlen=hparams.max_length, padding='post')
```

```
    tokenized_outputs = pad_sequences(tokenized_outputs, \
                        maxlen = hparams.max_length, padding = 'post')
    return tokenized_inputs, tokenized_outputs
# 读文件
def get_data(datafile):
    with open(f'{datafile}', 'r') as f:
        data_list = f.readlines()
        for i in range(len(data_list)):
            data_list[i] = re.sub(r'\n', '', data_list[i])
        return data_list
# 返回训练集和验证集
def get_dataset(hparams, tokenizer, dialog_file, train_file, valid_file):
    dialog_list = json.loads(codecs.open(f"{dialog_file}", "r", "utf-8").read())
    print(dialog_list[0])
    train_list = get_data(f'{train_file}')
    train_questions, train_answers = extract_conversations(hparams, train_list, dialog_list)
    train_questions, train_answers = tokenize_and_filter(hparams, \
                    list(train_questions), list(train_answers), tokenizer)
    # 构建训练集
    train_dataset = tf.data.Dataset.from_tensor_slices((
      {
          'inputs': train_questions,
          # 解码器使用正确的标签作为输入
          'dec_inputs': train_answers[:, :-1]      # 去掉最后一个元素或 END_TOKEN
      },
      {
          'outputs': train_answers[:, 1:]          # 去掉 START_TOKEN
      },
    ))
    train_dataset = train_dataset.cache()
    train_dataset = train_dataset.shuffle(len(train_questions))
    train_dataset = train_dataset.batch(hparams.batchSize)
    train_dataset = train_dataset.prefetch(tf.data.AUTOTUNE)
    # 构建验证集
    valid_list = get_data(f'{valid_file}')
    valid_questions, valid_answers = extract_conversations( \
                        hparams,valid_list, dialog_list)
    valid_questions, valid_answers = tokenize_and_filter(hparams, \
            list(valid_questions), list(valid_answers), tokenizer)
    valid_dataset = tf.data.Dataset.from_tensor_slices((
        {
            'inputs': valid_questions,
            # 解码器使用正确的标签作为输入
            'dec_inputs': valid_answers[:, :-1] # 去掉最后一个元素或 END_TOKEN
        },
        {
            'outputs': valid_answers[:, 1:] # 去掉 START_TOKEN
        },
    ))
```

```
    valid_dataset = valid_dataset.cache()
    valid_dataset = valid_dataset.shuffle(len(valid_questions))
    valid_dataset = valid_dataset.batch(hparams.batchSize)
    valid_dataset = valid_dataset.prefetch(tf.data.AUTOTUNE)
    return train_dataset, valid_dataset
```

程序源码解析参见本节视频教程。

5.7 定义 Transformer 输入层编码

模型定义的完整流程如图 5.9 所示。本节首先完成输入层的编码定义。后续各节分步完成注意力机制、编码器、解码器的模块定义,最后合成为模型的整体定义。

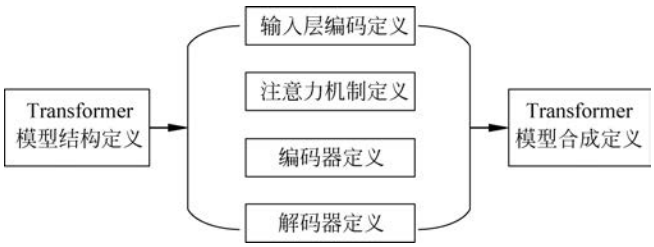

图 5.9 Transformer 模型定义的完整流程

在 transformer 目录下新建模型定义程序 model.py。关于编码器输入层的定义逻辑如程序源码 P5.2 所示。

程序源码 P5.2 model.py 之输入层定义

```
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import Input, Model
from tensorflow.keras.layers import Dense, Lambda, Embedding, Dropout, \
                add, LayerNormalization
from tensorflow.keras.utils import plot_model
# 定义掩码矩阵
def create_padding_mask(x):
    # 找出序列中的 padding,设置掩码值为 1
    mask = tf.cast(tf.math.equal(x, 0), tf.float32)
    # (batch_size, 1, 1, sequence length)
    return mask[:, tf.newaxis, tf.newaxis, :]
# 测试语句
print(create_padding_mask(tf.constant([[1, 2, 0, 3, 0], [0, 0, 0, 4, 5]])))
# 解码器的前向掩码
def create_look_ahead_mask(x):
    seq_len = tf.shape(x)[1]
    look_ahead_mask = 1 - tf.linalg.band_part(tf.ones((seq_len, seq_len)), -1, 0)
    padding_mask = create_padding_mask(x)
    return tf.maximum(look_ahead_mask, padding_mask)
```

```
# 测试
print(create_look_ahead_mask(tf.constant([[1, 2, 0, 4, 5]])))
# 位置编码类
class PositionalEncoding(tf.keras.layers.Layer):
    def __init__(self, position, d_model):
        super(PositionalEncoding, self).__init__()
        self.pos_encoding = self.positional_encoding(position, d_model)
    def get_config(self):
        config = super(PositionalEncoding, self).get_config()
        config.update({
            'position': self.position,
            'd_model': self.d_model,
        })
        return config
    def get_angles(self, position, i, d_model):
        angles = 1 / tf.pow(10000, (2 * (i //2)) / tf.cast(d_model, tf.float32))
        return position * angles
    def positional_encoding(self, position, d_model):
        angle_rads = self.get_angles( \
            position = tf.range(position, dtype = tf.float32)[:, tf.newaxis], \
            i = tf.range(d_model, dtype = tf.float32)[tf.newaxis, :], \
            d_model = d_model)
        # 奇数位置用正弦函数
        sines = tf.math.sin(angle_rads[:, 0::2])
        # 偶数位置用余弦函数
        cosines = tf.math.cos(angle_rads[:, 1::2])
        pos_encoding = tf.concat([sines, cosines], axis = -1)
        pos_encoding = pos_encoding[tf.newaxis, ...]
        return tf.cast(pos_encoding, tf.float32)
    def call(self, inputs):
        return inputs + self.pos_encoding[:, :tf.shape(inputs)[1], :]
# 测试
sample_pos_encoding = PositionalEncoding(50, 512)
plt.pcolormesh(sample_pos_encoding.pos_encoding.numpy()[0], cmap = 'RdBu')
plt.xlabel('Depth')
plt.xlim((0, 512))
plt.ylabel('Position')
plt.colorbar()
plt.show()
```

假定输入层向量的维度为(50,512),即序列长度为50(也即有50个单词),每个单词的嵌入向量长度为512,则整个输入层向量的位置编码及其几何分布如图5.10所示。对于中文而言,分词以单个中文字符为单位,所以位置编码是针对单个中文字符而言的。对于英文,分词以英文单词为单位。函数曲线的变化趋势体现了位置编码函数对不同位置编码的差异性。

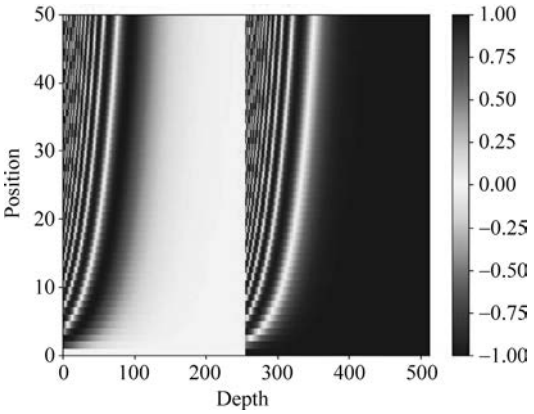

图 5.10　位置编码及其几何分布

5.8　定义 Transformer 注意力机制

先定义单头注意力函数,实现单头注意力机制的计算,然后合成多头注意力计算模块。编码逻辑如程序源码 P5.3 所示。

程序源码 P5.3　model.py 之注意力机制

```
# 计算注意力
def scaled_dot_product_attention(query, key, value, mask):
    matmul_qk = tf.matmul(query, key, transpose_b=True)
    # 计算 qk
    depth = tf.cast(tf.shape(key)[-1], tf.float32)
    logits = matmul_qk / tf.math.sqrt(depth)
    # 添加掩码以将填充标记归零
    if mask is not None:
        logits += (mask * -1e9)
    # 在最后一个轴上实施 softmax
    attention_weights = tf.nn.softmax(logits, axis=-1)
    output = tf.matmul(attention_weights, value)
    return output
# 定义多头注意力类,继承了 Layer 类
class MultiHeadAttention(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, name="multi_head_attention"):
        super(MultiHeadAttention, self).__init__(name=name)
        self.num_heads = num_heads
        self.d_model = d_model
        assert d_model % self.num_heads == 0
        self.depth = d_model //self.num_heads
        self.query_dense = Dense(units=d_model)
        self.key_dense = Dense(units=d_model)
        self.value_dense = Dense(units=d_model)
        self.dense = Dense(units=d_model)
```

```
    def get_config(self):
        config = super(MultiHeadAttention, self).get_config()
        config.update({
            'num_heads':self.num_heads,
            'd_model':self.d_model,
        })
        return config
    def split_heads(self, inputs, batch_size):
        inputs = Lambda(lambda inputs:tf.reshape(inputs, \
                shape=(batch_size, -1, self.num_heads, self.depth)))(inputs)
        return Lambda(lambda inputs: tf.transpose(inputs, perm=[0, 2, 1, 3]))(inputs)
    def call(self, inputs):
        query, key, value, mask = inputs['query'], inputs['key'], inputs[
            'value'], inputs['mask']
        batch_size = tf.shape(query)[0]
        # 线性层变换
        query = self.query_dense(query)
        key = self.key_dense(key)
        value = self.value_dense(value)
        # 分头
        query = self.split_heads(query, batch_size)
        key = self.split_heads(key, batch_size)
        value = self.split_heads(value, batch_size)
        # 定义缩放的点积注意力
        scaled_attention = scaled_dot_product_attention(query, key, value, mask)
        scaled_attention = Lambda(lambda scaled_attention: tf.transpose(
            scaled_attention, perm=[0, 2, 1, 3]))(scaled_attention)
        # 堆叠注意力头
        concat_attention = Lambda(lambda scaled_attention: tf.reshape( \
            scaled_attention,(batch_size, -1, self.d_model)))(scaled_attention)
        # 多头注意力最后一层
        outputs = self.dense(concat_attention)
        return outputs
```

5.9 定义 Transformer 编码器

先完成编码器一个单元的定义,然后由多个单元合成整个编码器。编码逻辑如程序源码 P5.4 所示。

程序源码 P5.4 model.py 之编码器定义

```
# 编码器中的一层,即编码器的一个单元定义
def encoder_layer(units, d_model, num_heads, dropout, name="encoder_layer"):
    inputs = tf.keras.Input(shape=(None, d_model), name="inputs")
    padding_mask = tf.keras.Input(shape=(1, 1, None), name="padding_mask")
    attention = MultiHeadAttention( \
      d_model, num_heads, name="attention")({ \
```

```
            'query': inputs,
            'key': inputs,
            'value': inputs,
            'mask': padding_mask
        })
    attention = Dropout(rate = dropout)(attention)
    add_attention = add([inputs,attention])
    attention = LayerNormalization(epsilon = 1e - 6)(add_attention)
    outputs = Dense(units = units, activation = 'relu')(attention)
    outputs = Dense(units = d_model)(outputs)
    outputs = Dropout(rate = dropout)(outputs)
    add_attention = add([attention,outputs])
    outputs = LayerNormalization(epsilon = 1e - 6)(add_attention)
    return Model(inputs = [inputs, padding_mask], outputs = outputs, name = name)
# 测试
sample_encoder_layer = encoder_layer(
    units = 512,
    d_model = 128,
    num_heads = 4,
    dropout = 0.3,
    name = "sample_encoder_layer")
plot_model(sample_encoder_layer, to_file = 'encoder_layer.png', show_shapes = True)
# 定义编码器,由多个单元合成编码器
def encoder(vocab_size,
            num_layers,
            units,
            d_model,
            num_heads,
            dropout,
            name = "encoder"):
    inputs = Input(shape = (None,), name = "inputs")
    padding_mask = Input(shape = (1, 1, None), name = "padding_mask")
    embeddings = Embedding(vocab_size, d_model)(inputs)
    embeddings * = Lambda(lambda d_model: tf.math.sqrt(tf.cast(d_model, tf.float32)))(d_model)
    embeddings = PositionalEncoding(vocab_size,d_model)(embeddings)
    outputs = Dropout(rate = dropout)(embeddings)
    for i in range(num_layers):
        outputs = encoder_layer(
            units = units,
            d_model = d_model,
            num_heads = num_heads,
            dropout = dropout,
            name = "encoder_layer_{}".format(i),
        )([outputs, padding_mask])
    return Model(inputs = [inputs, padding_mask], outputs = outputs, name = name)
# 编码器测试
sample_encoder = encoder(
    vocab_size = 21128,
    num_layers = 2,
```

```
    units = 512,
    d_model = 128,
    num_heads = 4,
    dropout = 0.3,
    name = "sample_encoder")
plot_model(sample_encoder, to_file = 'encoder.png', show_shapes = True)
```

测试编码器程序，观察生成的编码器逻辑结构，与 Transformer 论文解析的结构做对照，在实践中灵活配置编码器的相关参数，可得到适配问题需求的编码器。

5.10 定义 Transformer 解码器

解码器的设计思路与编码器类似。先完成解码器一个单元的定义，然后由多个单元合成整个解码器。编码逻辑如程序源码 P5.5 所示。

程序源码 P5.5 model.py 之解码器定义

```
# 定义解码器中的一层，一个解码单元
def decoder_layer(units, d_model, num_heads, dropout, name = "decoder_layer"):
    inputs = Input(shape = (None, d_model), name = "inputs")
    enc_outputs = Input(shape = (None, d_model), name = "encoder_outputs")
    look_ahead_mask = Input(shape = (1, None, None), name = "look_ahead_mask")
    padding_mask = Input(shape = (1, 1, None), name = 'padding_mask')
    attention1 = MultiHeadAttention(
      d_model, num_heads, name = "attention_1")(inputs = {
          'query': inputs,
          'key': inputs,
          'value': inputs,
          'mask': look_ahead_mask
      })
    add_attention = tf.keras.layers.add([attention1,inputs])
    attention1 = tf.keras.layers.LayerNormalization(epsilon = 1e - 6)(add_attention)
    attention2 = MultiHeadAttention(
      d_model, num_heads, name = "attention_2")(inputs = {
          'query': attention1,
          'key': enc_outputs,
          'value': enc_outputs,
          'mask': padding_mask
      })
    attention2 = Dropout(rate = dropout)(attention2)
    add_attention = add([attention2,attention1])
    attention2 = LayerNormalization(epsilon = 1e - 6)(add_attention)
    outputs = Dense(units = units, activation = 'relu')(attention2)
    outputs = Dense(units = d_model)(outputs)
    outputs = Dropout(rate = dropout)(outputs)
    add_attention = add([outputs,attention2])
    outputs = LayerNormalization(epsilon = 1e - 6)(add_attention)
```

```
    return Model(
          inputs = [inputs, enc_outputs, look_ahead_mask, padding_mask],
          outputs = outputs,
          name = name)
# 测试
sample_decoder_layer = decoder_layer(
    units = 512,
    d_model = 128,
    num_heads = 4,
    dropout = 0.3,
    name = "sample_decoder_layer")
plot_model(sample_decoder_layer, to_file = 'decoder_layer.png', show_shapes = True)
# 定义解码器,合成多个解码单元
def decoder(vocab_size,
            num_layers,
            units,
            d_model,
            num_heads,
            dropout,
            name = 'decoder'):
    inputs = Input(shape = (None,), name = 'inputs')
    enc_outputs = Input(shape = (None, d_model), name = 'encoder_outputs')
    look_ahead_mask = Input(shape = (1, None, None), name = 'look_ahead_mask')
    padding_mask = Input(shape = (1, 1, None), name = 'padding_mask')
    embeddings = Embedding(vocab_size, d_model)(inputs)
    embeddings * = Lambda(lambda d_model: tf.math.sqrt( \
                          tf.cast(d_model, tf.float32)))(d_model)
    embeddings = PositionalEncoding(vocab_size, d_model)(embeddings)
    outputs = Dropout(rate = dropout)(embeddings)
    for i in range(num_layers):
        outputs = decoder_layer(
            units = units,
            d_model = d_model,
            num_heads = num_heads,
            dropout = dropout,
            name = 'decoder_layer_{}'.format(i),
        )(inputs = [outputs, enc_outputs, look_ahead_mask, padding_mask])
    return Model(
          inputs = [inputs, enc_outputs, look_ahead_mask, padding_mask],
          outputs = outputs,
          name = name)
# 解码器测试
sample_decoder = decoder(
    vocab_size = 21128,
    num_layers = 2,
    units = 512,
    d_model = 128,
    num_heads = 4,
    dropout = 0.3,
```

```
80      name = "sample_decoder")
81  plot_model(sample_decoder, to_file = 'decoder.png', show_shapes = True)
```

测试解码器程序，观察生成的解码器逻辑结构，与 Transformer 论文解析的结构做对照，在实践中灵活配置解码器的相关参数，可得到适配问题需求的解码器。

5.11 Transformer 模型合成

在前面分步完成的各个模块的基础上，定义 Transformer 的完整模型。编程逻辑如程序源码 P5.6 所示。

程序源码 P5.6 model.py 之 Transformer 定义

```
1   # 定义 Transformer 模型
2   def transformer(vocab_size,
3                   num_layers,
4                   units,
5                   d_model,
6                   num_heads,
7                   dropout,
8                   name = "transformer"):
9       inputs = Input(shape = (None,), name = "inputs")
10      dec_inputs = Input(shape = (None,), name = "dec_inputs")
11      enc_padding_mask = Lambda(
12        create_padding_mask, output_shape = (1, 1, None),
13        name = 'enc_padding_mask')(inputs)
14      # 解码器第一个注意力块的前向掩码
15      look_ahead_mask = Lambda(
16        create_look_ahead_mask,
17        output_shape = (1, None, None),
18        name = 'look_ahead_mask')(dec_inputs)
19      # 对编码器输出到解码器第二个注意力块的内容掩码
20      dec_padding_mask = Lambda(
21        create_padding_mask, output_shape = (1, 1, None),
22        name = 'dec_padding_mask')(inputs)
23      enc_outputs = encoder(
24            vocab_size = vocab_size,
25            num_layers = num_layers,
26            units = units,
27            d_model = d_model,
28            num_heads = num_heads,
29            dropout = dropout,
30          )(inputs = [inputs, enc_padding_mask])
31      dec_outputs = decoder(
32            vocab_size = vocab_size,
33            num_layers = num_layers,
34            units = units,
```

```
            d_model = d_model,
            num_heads = num_heads,
            dropout = dropout,
        )(inputs = [dec_inputs, enc_outputs, look_ahead_mask, dec_padding_mask])
    outputs = Dense(units = vocab_size, name = "outputs")(dec_outputs)
    return Model(inputs = [inputs, dec_inputs], outputs = outputs, name = name)
# 测试
sample_transformer = transformer(
    vocab_size = 21128,
    num_layers = 4,
    units = 512,
    d_model = 128,
    num_heads = 4,
    dropout = 0.3,
    name = "sample_transformer")
plot_model(sample_transformer, to_file = 'transformer.png', show_shapes = True)
```

查看生成的 Transformer 结构图,理解 Transformer 的逻辑运算过程。实践中可灵活调整相关参数配置,以与目标问题相适配。

Transformer 模型定义期间,为了测试各模块程序的逻辑,在每一个模块后面都编写了测试语句。模型程序 model.py 可以迭代运行,观察测试语句的输出结果,加强对模型的理解。其中的 plot_model 函数可以绘制模型结构图,但是需要安装图形支持包 graphviz,根据程序运行时的相关提示,完成环境配置。

Transformer 各模块的逻辑解析及其测试,参见视频教程。

5.12 模型结构与参数配置

从本节到 5.16 节,分步完成聊天机器人模型的训练与评估。打开 5.5 节创建的主程序 main.py,首先完成库的导入和 Transformer 聊天机器人的参数配置和结构定义。编程逻辑如程序源码 P5.7 所示。

程序源码 P5.7 main.py 之模型结构与参数配置

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.losses import SparseCategoricalCrossentropy
from tensorflow import multiply, minimum
from tensorflow.keras.optimizers.schedules import LearningRateSchedule
from tensorflow.keras.metrics import sparse_categorical_accuracy
from tensorflow.python.ops.math_ops import rsqrt
from tensorflow.keras.optimizers import Adam
from bert.tokenization.bert_tokenization import FullTokenizer
import numpy as np
# 用 BLEU 方法评估模型
from nltk.translate.bleu_score import sentence_bleu
from transformer.model import *
```

```
from transformer.dataset import *
if __name__ == "__main__" :
    dialog_list = json.loads( \
        codecs.open("transformer/dataset/dialog_release.json", \
        "r", "utf-8").read())
    print(dialog_list[0])                    # 第一条数据
    # 以下参数可根据需要调整,为了演示,可以将相关参数调低一些
    # 最大句子长度
    MAX_LENGTH = 40
    # 最大样本数量
    MAX_SAMPLES = 120000                     # 可根据需要调节
    BATCH_SIZE = 64                          # 批处理大小
    # Transformer 参数定义
    NUM_LAYERS = 2                           # 编码器解码器 block 重复数,论文中是 6
    D_MODEL = 128                            # 编码器解码器宽度,论文中是 512
    NUM_HEADS = 4                            # 注意力头数,论文中是 8
    UNITS = 512                              # 全连接网络宽度,论文中输入输出为 512
    DROPOUT = 0.1                            # 与论文一致
    VOCAB_SIZE = 21128                       # BERT 词典长度
    START_TOKEN = [VOCAB_SIZE]               # 序列起始标志
    END_TOKEN = [VOCAB_SIZE + 1]             # 序列结束标志
    VOCAB_SIZE = VOCAB_SIZE + 2              # 加上开始与结束标志后的词典长度
    EPOCHS = 50                              # 训练代数
    bert_vocab_file = 'transformer/dataset/vocab.txt'
    tokenizer = FullTokenizer(bert_vocab_file)
    # 聊天模型参数配置与结构定义
    model = transformer(
        vocab_size = VOCAB_SIZE,
        num_layers = NUM_LAYERS,
        units = UNITS,
        d_model = D_MODEL,
        num_heads = NUM_HEADS,
        dropout = DROPOUT)
    model.summary()
```

为了满足教学演示需要,程序源码 P5.7 中将 Transformer 编码器与解码器的单元数均缩减为 2,其他参数也有相应缩减,模型可训练参数总量为 9 060 746 个。

模型采用了 BERT 分词方法,故需要安装 BERT 模型库。安装命令为:

```
pip install bert-for-tf2
```

模型采用了两种评价方法:一是计算模型的回归损失;二是计算 BLEU 得分。需要安装 BLEU 函数库。安装命令为:

```
pip install nltk
```

5.13 学习率动态调整

为了优化模型训练过程,加快模型收敛速度,指定了学习率动态调整策略,编码逻辑如程序源码 P5.8 所示。

程序源码 P5.8 main.py 之学习率动态调整

```
# 学习率动态调整
class CustomSchedule(LearningRateSchedule):
    def __init__(self, d_model, warmup_steps=4000):
        super(CustomSchedule, self).__init__()
        self.d_model = tf.constant(d_model, dtype=tf.float32)
        self.warmup_steps = warmup_steps
    def get_config(self):
        return {"d_model": self.d_model, "warmup_steps": self.warmup_steps}
    def __call__(self, step):
        arg1 = rsqrt(step)
        arg2 = step * (self.warmup_steps ** -1.5)
        return multiply(rsqrt(self.d_model), minimum(arg1, arg2))
# 测试
sample_learning_rate = CustomSchedule(d_model=256)
plt.plot(sample_learning_rate(tf.range(200000, dtype=tf.float32)))
plt.ylabel("Learning Rate")
plt.xlabel("Train Step")
plt.show()
learning_rate = CustomSchedule(D_MODEL)          # 学习率
```

学习率变化曲线如图 5.11 所示,训练初期学习率采取上升策略以加快训练速度,训练中期保持学习率为一个稳定值,训练后期对学习率采取衰减策略,以期寻找最优解。

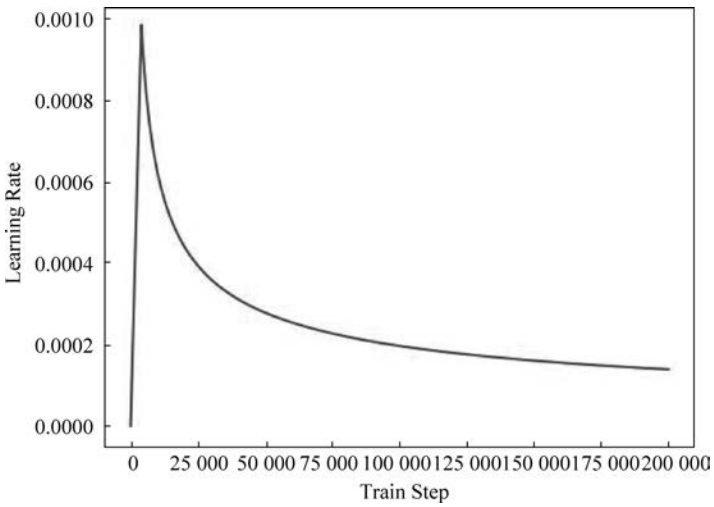

图 5.11 学习率变化曲线

当然，图 5.11 显示本案例跳过了学习率恒定的阶段，在达到最高值后直接开始衰减。

5.14 模型训练过程

模型训练之前，指定模型采用的优化算法为 Adam，定义分类交叉熵损失函数，并定义准确率评价标准，完成模型编译。训练过程中，保存可能取得的最优模型的权重，用提前终止回调函数控制模型训练进程。编码逻辑如程序源码 P5.9 所示。

程序源码 P5.9 main.py 之模型训练过程

```
# 定义损失函数
def loss_function(y_true, y_pred):
    y_true = tf.reshape(y_true, shape=(-1, MAX_LENGTH - 1))
    loss = SparseCategoricalCrossentropy(
        from_logits=True, reduction='none')(y_true, y_pred)
    mask = tf.cast(tf.not_equal(y_true, 0), tf.float32)
    loss = tf.multiply(loss, mask)
    return tf.reduce_mean(loss)
# 自定义准确率函数
def accuracy(y_true, y_pred):
    # 调整标签的维度为:(batch_size, MAX_LENGTH - 1)
    y_true = tf.reshape(y_true, shape=(-1, MAX_LENGTH - 1))
    return sparse_categorical_accuracy(y_true, y_pred)
# 优化算法
optimizer = Adam(learning_rate, beta_1=0.9, beta_2=0.98, epsilon=1e-9)
model.compile(optimizer=optimizer, loss=loss_function, metrics=[accuracy])
# 定义回调函数:保存最优模型
checkpoint = ModelCheckpoint("robot_weights.h5",
                             monitor="val_loss",
                             mode="min",
                             save_best_only=True,
                             save_weights_only=True,
                             verbose=1)
# 定义回调函数:提前终止训练
earlystop = EarlyStopping(monitor='val_loss',
                          min_delta=0,
                          patience=10,
                          verbose=1,
                          restore_best_weights=True)
# 将回调函数组织为回调列表
callbacks = [earlystop, checkpoint]
dialog_file = 'transformer/dataset/dialog_release.json'
train_file = 'transformer/dataset/train.txt'
valid_file = 'transformer/dataset/dev.txt'
class Hparams() :
    def __init__(self,start_token,end_token,batchSize,total_samples,max_length):
```

```
        self.start_token = start_token
        self.end_token = end_token
        self.batchSize = batchSize
        self.total_samples = total_samples
        self.max_length = max_length
hparams = Hparams
hparams.start_token = START_TOKEN
hparams.end_token = END_TOKEN
hparams.total_samples = MAX_SAMPLES
hparams.batchSize = BATCH_SIZE
hparams.max_length = MAX_LENGTH
# 加载并划分数据集
train_dataset, valid_dataset = get_dataset(hparams, tokenizer, dialog_file,
train_file, valid_file)
# 模型训练
history = model.fit(train_dataset, epochs = EPOCHS, validation_data = valid_dataset,
callbacks = callbacks)
```

执行程序源码 P5.9,开始模型训练。训练结束后,在当前目录下会保存最佳模型的权重文件 robot_weights.h5。

如果计算机配置不够,可以先不要考虑模型可用性,适当降低模型参数配置,先运行并通过项目逻辑。

本项目在 Kaggle 服务器上的训练结果可以参见链接 https://www.kaggle.com/code/upsunny/naturalconv-chatbot/notebook。

如果训练模型的计算机配置过低,无法完成模型训练时,可以先将本书素材包中的 robot_weights_l2.h5 模型复制到 MyRobot 的 models 目录下。按照视频教程演示的方法,调用预训练模型完成后续测试任务。

5.15 损失函数与准确率曲线

绘制模型损失函数曲线与准确率曲线,有助于观察模型的过拟合情况,判断模型的泛化能力。编码逻辑如程序源码 P5.10 所示。

程序源码 P5.10 main.py 之损失函数与准确率曲线

```
# 损失函数曲线
plt.figure(figsize = (12, 6))
x = range(1, len(history.history['loss']) + 1)
plt.plot(x, history.history['loss'])
plt.plot(x, history.history['val_loss'])
plt.xticks(x)
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['train', 'test'])
```

```
plt.title('Loss over training epochs')
plt.savefig('loss.png')
plt.show()
# 准确率曲线
plt.figure(figsize = (12, 6))
plt.plot(x, history.history['accuracy'])
plt.plot(x, history.history['val_accuracy'])
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.xticks(x)
plt.legend(['train', 'test'])
plt.title('Accuracy over training epochs')
plt.savefig('acc.png')
plt.show()
```

损失函数曲线如图 5.12 所示。模型在第 14 代之前，损失保持了较快的下降速度。从第 20 代开始，模型优化的幅度不明显，逐渐呈现过拟合趋势。

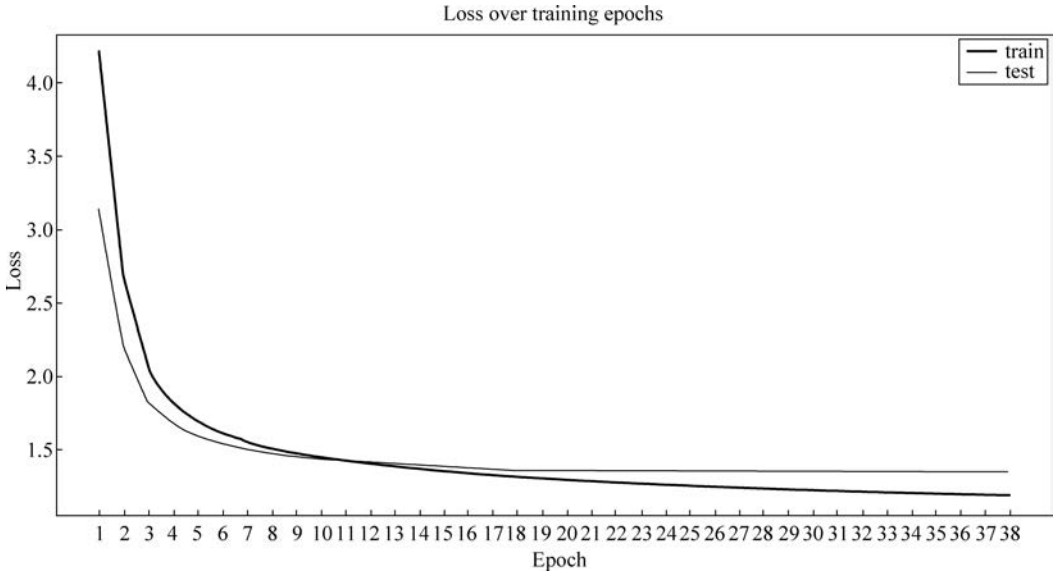

图 5.12 损失函数曲线

准确率曲线如图 5.13 所示，与损失函数曲线基本保持了一致的判断。在第 14 代之前，模型准确率保持较快的增长，第 20 代之后，训练集上的准确率仍保持缓慢增长，验证集上的准确率则几乎保持不变，模型逐渐呈现过拟合趋势。

模型设置了提前结束的条件，如果连续 10 代的损失函数不下降，则提前终止模型训练。这就是为什么设置了 50 代的训练，却在第 38 代终止训练的原因。

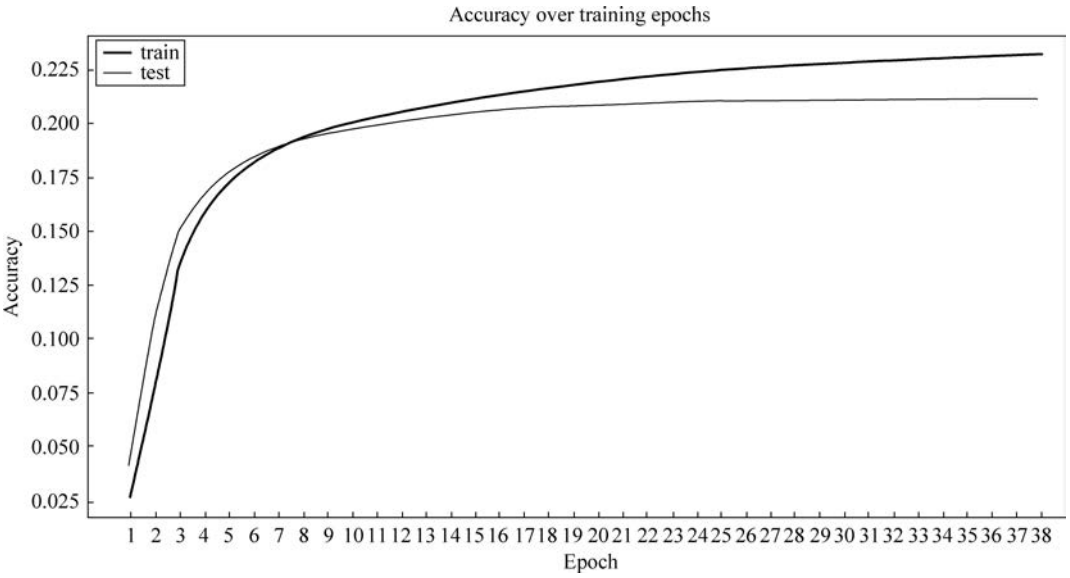

图 5.13　准确率曲线

5.16　聊天模型评估与测试

用 5.15 节训练好的模型做随机对话测试,并用 BLEU 评分观察预测结果。编码逻辑如程序源码 P5.11 所示。

程序源码 P5.11　main.py 之聊天模型评估与测试

```
# 加载训练好的模型
model.load_weights('models/robot_weights_l2.h5')
# 用模型做聊天推理,A、B 两人聊天,输入 A 的句子,得到 B 的回应
def evaluate(sentence):
    sentence = tokenizer.tokenize(sentence)
    sentence = START_TOKEN + tokenizer.convert_tokens_to_ids(sentence) + END_TOKEN
    sentence = tf.expand_dims(sentence, axis=0)
    output = tf.expand_dims(START_TOKEN, 0)
    for i in range(MAX_LENGTH):
        predictions = model(inputs=[sentence, output], training=False)
        # 选择最后一个输出
        predictions = predictions[:, -1:, :]
        predicted_id = tf.cast(tf.argmax(predictions, axis=-1), tf.int32)
        # 如果是 END_TOKEN 则结束预测
        if tf.equal(predicted_id, END_TOKEN[0]):
            break
        # 把已经得到的预测值串联起来,作为解码器的新输入
        output = tf.concat([output, predicted_id], axis=-1)
    return tf.squeeze(output, axis=0)
# 模拟聊天间的问答,输入问话,输出回答
```

```
def predict(question):
    prediction = evaluate(question)                    # 调用模型推理
    predicted_answer = tokenizer.convert_ids_to_tokens(
        np.array([i for i in prediction if i < VOCAB_SIZE - 2]))
    print(f'问话者: {question}')
    print(f'答话者: {"".join(predicted_answer)}')
    return predicted_answer
# 几组随机测试
output1 = predict('嗨,你好呀.')                        # 训练集中的样本
print("")
output2 = predict('昨晚的比赛你看了吗?')               # 随机问话 1
print("")
output3 = predict('你最喜欢的人是谁?')                 # 随机问话 2
print("")
output4 = predict('真热,下点儿雨就好了')               # 随机问话 3
print("")
output5 = predict('这个老师讲课怎么样?')               # 随机问话 4
print("")
output6 = predict('今天收获大吗?')                     # 随机问话 5
print("")
# 多轮对话测试,自问自答
sentence = '你最近听说过«中国女排»这部电影吗?'
for _ in range(5):
    sentence = "".join(predict(sentence))
    print("")
reference = '是呀,我觉得这部«中国女排»应该能有很高的收视率.'
pred_sentence = predict(reference)
reference = tokenizer.tokenize(reference)
# 1-gram BLEU 计算
BLEU_1 = sentence_bleu([reference], pred_sentence, weights = (1, 0, 0, 0))
print(f"\n BLEU-1 评分: {BLEU_1}")
# 2-gram BLEU 计算
BLEU_2 = sentence_bleu([reference], pred_sentence, weights = (0.5, 0.5, 0, 0))
print(f"\n BLEU-2 评分: {BLEU_2}")
# 3-gram BLEU 计算
BLEU_3 = sentence_bleu([reference], pred_sentence, weights = (0.33, 0.33, 0.33, 0))
print(f"\n BLEU-3 评分:: {BLEU_3}")
# 4-gram BLEU 计算
BLEU_4 = sentence_bleu([reference], pred_sentence, weights = (0.25, 0.25, 0.25, 0.25))
print(f"\n BLEU-4 评分:: {BLEU_4}")
# 5-gram BLEU 计算
BLEU_5 = sentence_bleu([reference], pred_sentence, weights = (0.2, 0.2, 0.2, 0.2, 0.2))
print(f"\n BLEU-5 评分:: {BLEU_5}")
```

程序源码 P5.11 测试结果如表 5.5 所示。假定问话者由人扮演,答话者由机器扮演。自问自答部分由机器同时扮演两个角色。

表 5.5 聊天模型推理抽样观察

简单问答：(一问一答模式)
问话者：嗨，你好呀。
答话者：你好。
问话者：昨晚的比赛你看了吗？
答话者：看了，昨天我看了，昨天晚上的比赛，我看了。
问话者：你最喜欢的人是谁？
答话者：我喜欢的就是那个梅西，他的球技也很好。
问话者：真热，下点儿雨就好了。
答话者：是啊，我也是刚刚看到这个新闻，你看了吗？
问话者：这个老师讲课怎么样？
答话者：我们学校的教师，学校也是很重要的。
问话者：今天收获大吗？
答话者：今天休班。
自问自答：(连续推理测试)
问话者：你最近听说过《中国女排》这部电影吗？
答话者：没有哎，我最近没怎么关注电影。
问话者：没有哎，我最近没怎么关注电影。
答话者：你看了吗？
问话者：你看了吗？
答话者：看了，这部电影的预告片很好看。
问话者：看了，这部电影的预告片很好看。
答话者：是啊，这部电影的主演是谁啊？
问话者：是啊，这部电影的主演是谁啊？
答话者：这部电影是《中国机长》，叫《我的祖国》。

注意：表中的对话解析参见视频教程。

模型还对下面这组问答给出了 BLEU 评分。

问话者：是呀，我觉得这部《中国女排》应该能拿下很高的收视率。

答话者：是呀，这次的世界杯的表现也是非常不错的。

BLEU-1 评分：0.22517932221294598

BLEU-2 评分：0.1332178835716084

BLEU-3 评分：0.09227103858589292

BLEU-4 评分：1.8955151000606497e－78

BLEU-5 评分：1.8662507507148366e－124

事实上，受限于答话句子的长度，BLEU-4 与 BLEU-5 评分没有实际意义。BLEU-1 的得分值表明模型具备一定的可用性与参考性。

直观看，机器的回答是有些跑题，甚至答非所问。但是似乎前后又有一定联系。因为前者说收视率很高，是一个正面评价。后者给出的是对世界杯的正面评价。或许，机器人并不知道如何理解和接续问话者的表达，只是根据自己建模得到的经验做了一个力所能及的回答。

至于为什么这个回答与世界杯有关，而不是与电影有关，一是问话者的语言中包含“中国女排”，这可以理解为体育相关的话题；二是在5.3节已经指明，给定的数据集偏重体育语料，会使得训练的模型偏爱体育表达。

事实上，对人类之间的交流而言，这完全不是问题，因为其中的“这部”“收视率”等字眼表明谈论的《中国女排》是一部电影。

5.17 聊天模型部署到服务器

将训练好的Transformer聊天模型部署到Web服务器上，可以实现一对多的聊天服务。在第1章已经搭建了一个基于Flask的Web API服务框架。在此基础上，迭代追加支持机器人聊天的Web API设计。

打开Server目录下的app.py程序，追加聊天机器人的服务逻辑，如程序源码P5.12所示。

程序源码 P5.12 app.py之聊天模型部署到服务器

```
# Transformer参数定义
# 最大句子长度
MAX_LENGTH = 40
NUM_LAYERS = 2                          # 编码器解码器block重复数,论文中是6
D_MODEL = 128                           # 编码器解码器宽度,论文中是512
NUM_HEADS = 4                           # 注意力头数,论文中是8
UNITS = 512                             # 全连接网络宽度,论文中输入输出为512
DROPOUT = 0.1                           # 与论文一致
VOCAB_SIZE = 21128                      # 词典长度
START_TOKEN = [VOCAB_SIZE]              # 序列起始标志
END_TOKEN = [VOCAB_SIZE + 1]            # 序列结束标志
VOCAB_SIZE = VOCAB_SIZE + 2             # 加上开始与结束标志后的词典长度
# 分词器
bert_vocab_file = '../MyRobot/transformer/dataset/vocab.txt'
tokenizer = FullTokenizer(bert_vocab_file)
# 模型
robot_model = transformer(
    vocab_size = VOCAB_SIZE,
    num_layers = NUM_LAYERS,
    units = UNITS,
    d_model = D_MODEL,
    num_heads = NUM_HEADS,
    dropout = DROPOUT)
# 加载权重文件.注意,上面的模型参数必须与权重文件对应的结构一致
robot_model.load_weights('../MyRobot/models/robot_weights_l2.h5')
# 用模型做聊天推理,A、B两人聊天,输入A的句子,得到B的回应
def robot_evaluate(sentence):
    sentence = tokenizer.tokenize(sentence)
    sentence = START_TOKEN + tokenizer.convert_tokens_to_ids(sentence) + END_TOKEN
    sentence = tf.expand_dims(sentence, axis = 0)
```

```
    output = tf.expand_dims(START_TOKEN, 0)
    for i in range(MAX_LENGTH):
        predictions = robot_model(inputs=[sentence, output], training=False)
        # 选择最后一个输出
        predictions = predictions[:, -1:, :]
        predicted_id = tf.cast(tf.argmax(predictions, axis=-1), tf.int32)
        # 如果是 END_TOKEN 则结束预测
        if tf.equal(predicted_id, END_TOKEN[0]):
            break
        # 把已经得到的预测值串联起来,作为解码器的新输入
        output = tf.concat([output, predicted_id], axis=-1)
    return tf.squeeze(output, axis=0)
# 输入问话,输出回答
def robot_predict(question):
    prediction = robot_evaluate(question)
    predicted_answer = tokenizer.convert_ids_to_tokens(
                        np.array([i for i in prediction if i < VOCAB_SIZE - 2]))
    return "".join(predicted_answer)
# 机器人聊天 API
@app.route('/robot', methods=['post'])
def robot():
    message = request.get_json(force=True)
    question = message['question']
    answer = robot_predict(question)
    response = {
        'answer': answer
    }
    print(response)
    return jsonify(response), 200
```

待完成 Android 客户机后,再与服务器做联合测试。

5.18 Android 项目初始化

新建 Android 项目,模板选择 Empty Activity,项目参数设定如图 5.14 所示。项目

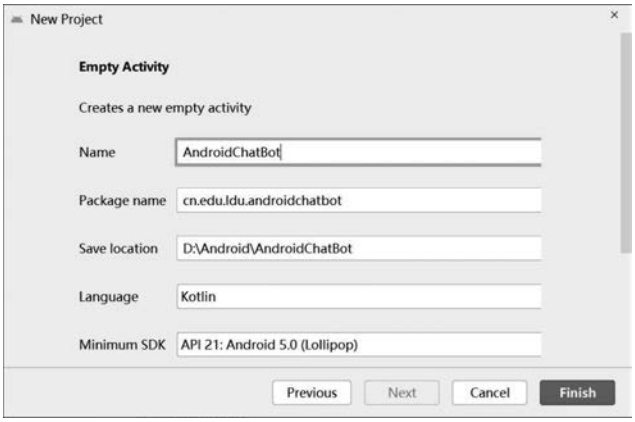

图 5.14 项目参数设定

名称为 AndroidChatBot，包名称为 cn. edu. ldu. androidchatbot，编程语言为 Kotlin，SDK 最小版本号设置为 API 21：Android 5.0(Lollipop)，单击 Finish，完成项目初始化。

选择项目根目录，右击，借助快捷菜单命令 New→Package 分别新建 ui、utils、network、data 四个子目录。各子目录的功能及其包含的程序如表 5.6 所示。

表 5.6 项目各子目录的功能及其包含的程序

子目录名	程序名	功能
ui	MainActivity	主控界面逻辑控制
	MessagingAdapter	主控界面数据适配器
utils	Constant	定义全局性常量对象
	Time	定义时间戳对象
network	ApiService	定义网络访问服务接口
data	Answer	匹配服务器应答消息结构的实体类
	Message	消息实体类

依照表 5.6 的提示，依次完成各个程序模块的创建。MainActivity 在项目初始化时已自动生成，将其移动到 ui 目录下即可。项目初始结构如图 5.15 所示。

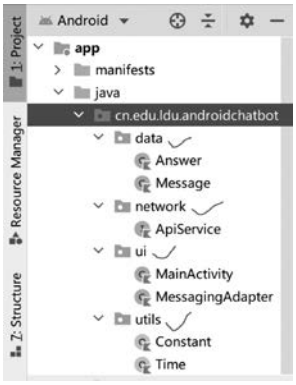

图 5.15 项目初始结构

其他一些简单的初始化工作包括：

(1) 定义实体类 Answer。

```
data class Answer(val answer:String)
```

(2) 定义实体类 Message。

```
data class Message(val message:String, val id:String, val
time:String)
```

(3) 在清单文件中声明 Internet 访问权限。

```
<uses-permission android:name="android.permission.INTERNET" />
```

(4) 支持 HTTP 通信。

考虑到本案例采用 HTTP 通信，还需要在 appliacation 节点中添加支持 HTTP 的属性。

```
android:usesCleartextTraffic="true"
```

(5) 添加模块依赖。

在模块配置文件开头的 plugins{}节点后面添加 Kotlin 扩展语句。

```
apply plugin: 'kotlin-android-extensions'
```

在末尾的 dependencies{}节点中追加以下依赖库：

```
// RecyclerView
implementation("androidx.recyclerview:recyclerview:1.2.1")
```

```
// For control over item selection of both touch and mouse driven selection
implementation("androidx.recyclerview:recyclerview-selection:1.1.0")
// Coroutines
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.0'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.5.0'
// Retrofit2
implementation "com.squareup.retrofit2:retrofit:2.9.0"
implementation "com.squareup.retrofit2:converter-scalars:2.9.0"
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
// Glide
implementation 'com.github.bumptech.glide:glide:4.11.0'
annotationProcessor 'com.github.bumptech.glide:compiler:4.11.0'
```

注意,修改 build.gradle 文件之后,需要同步更新项目配置。系统设计过程中,可根据需要,随时为模块引入相关依赖库。

(6) 定义工具性对象类。

Constant 定义两个常量标识符,用于区别消息发送者和接收者。

```
package cn.edu.ldu.androidchatbot.utils
object Constant {
    const val SEND_ID = "SEND_ID"
    const val RECEIVE_ID = "RECEIVE_ID"
}
```

Time 定义时间戳,表示消息收发时间。

```
package cn.edu.ldu.androidchatbot.utils
import java.sql.Date
import java.sql.Timestamp
import java.text.SimpleDateFormat
object Time {
    fun timeStamp(): String {
        val timeStamp = Timestamp(System.currentTimeMillis())
        val sdf = SimpleDateFormat("HH:mm")
        val time = sdf.format(Date(timeStamp.time))
        return time.toString()
    }
}
```

(7) 定义网络通信模块。

```
package cn.edu.ldu.androidchatbot.network
import okhttp3.RequestBody
import okhttp3.ResponseBody
import retrofit2.Response
import retrofit2.http.Body
```

```
import retrofit2.http.POST
interface ApiService {
    @POST("/robot")
    suspend fun getAnswer(@Body body: RequestBody): Response<ResponseBody>
}
```

修改程序名称为“我的聊天机器人”。接下来，转入界面设计和主控逻辑设计。

5.19 Android 聊天界面设计

聊天界面是程序的主控界面，核心控件是构建聊天列表的 RecyclerView。主控界面由布局文件 activity_main.xml 定义。RecyclerView 中的单行布局由 message_item.xml 定义。

为了美化界面风格，定义两个形状控件，用于渲染聊天消息的背景。send_box.xml 定义的形状用于衬托和突出用户发送的消息。round_box.xml 定义的形状作为发送按钮的背景轮廓。receive_box.xml 定义的形状用于衬托和突出用户收到的消息。

相关资源列表如图 5.16 所示。

选择 drawable 目录，右击，在弹出的快捷菜单中执行 New→Drawable Recourse File 命令，设置文件名称为 receive_box，根元素为 shape，如图 5.17 所示，单击 OK 按钮，创建形状资源文件。

图 5.16 与界面布局相关的资源列表

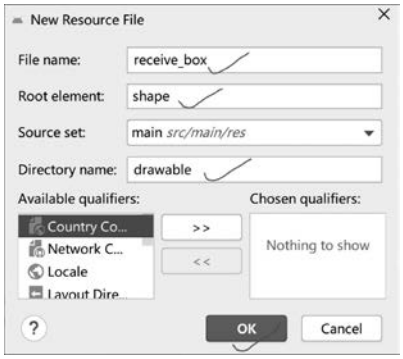

图 5.17 创建 receive_box.xml 文件

替换 receive_box.xml 文件内容。

```
<?xml version = "1.0" encoding = "utf - 8"?>
<shape xmlns:android = "http://schemas.android.com/apk/res/android">
    <corners android:topLeftRadius = "0dp"
        android:topRightRadius = "20dp"
        android:bottomLeftRadius = "20dp"
        android:bottomRightRadius = "20dp"/>
</shape>
```

这是一个左上角为直角、其他三个角为圆角的矩形框。

用类似的方法定义 send_box.xml。

```
<?xml version = "1.0" encoding = "utf - 8"?>
< shape xmlns:android = "http://schemas.android.com/apk/res/android">
    < corners android:topLeftRadius = "20dp"
        android:topRightRadius = "20dp"
        android:bottomLeftRadius = "20dp"
        android:bottomRightRadius = "0dp"/>
</shape >
```

这是一个右下角为直角、其他三个角为圆角的矩形框。

再定义 round_box.xml,这是一个四个角都是圆角的矩形框,用于修饰主界面下方文本框和按钮所在的区域。

```
<?xml version = "1.0" encoding = "utf - 8"?>
< shape xmlns:android = "http://schemas.android.com/apk/res/android">
    < corners android:radius = "20dp"></corners >
</shape >
```

用程序源码 P5.13 所示的脚本,完成主控界面布局。

程序源码 P5.13　activity_main.xml 主控界面布局脚本

```
1  <?xml version = "1.0" encoding = "utf - 8"?>
2  < RelativeLayout xmlns:android = "http://schemas.android.com/apk/res/android"
3      xmlns:tools = "http://schemas.android.com/tools"
4      android:layout_width = "match_parent"
5      android:layout_height = "match_parent"
6      tools:context = ".ui.MainActivity">
7      < LinearLayout
8          android:id = "@ + id/ll_layout_bar"
9          android:layout_width = "match_parent"
10         android:layout_height = "wrap_content"
11         android:layout_alignParentBottom = "true"
12         android:background = "#E4E4E4"
13         android:orientation = "horizontal">
14         < EditText
15             android:id = "@ + id/input_box"
16             android:inputType = "textShortMessage"
17             android:layout_width = "match_parent"
18             android:layout_height = "wrap_content"
19             android:layout_margin = "10dp"
20             android:layout_weight = ".5"
21             android:background = "@drawable/round_box"
22             android:backgroundTint = "@android:color/white"
23             android:hint = "输入消息..."
24             android:padding = "10dp"
```

```
            android:singleLine = "true" />
        < Button
            android:id = "@ + id/btn_send"
            android:layout_width = "match_parent"
            android:layout_height = "match_parent"
            android:layout_margin = "10dp"
            android:layout_weight = "1"
            android:background = "@drawable/round_box"
            android:backgroundTint = "#26A69A"
            android:text = "发 送"
            android:textColor = "@android:color/white" />
    </LinearLayout >
    < androidx.recyclerview.widget.RecyclerView
        android:id = "@ + id/chat_view"
        android:layout_width = "match_parent"
        android:layout_height = "match_parent"
        android:layout_above = "@id/ll_layout_bar"
        android:layout_below = "@ + id/dark_divider"
        tools:itemCount = "20"
        tools:listitem = "@layout/message_item" />
    < View
        android:layout_width = "match_parent"
        android:layout_height = "10dp"
        android:background = "#42A5F5"
        android:id = "@ + id/dark_divider" />
</RelativeLayout >
```

还有最后一项工作,定义 RecyclerView 中每一行的结构。

选择 layout 目录,右击,在弹出的快捷菜单中执行 New→Layout Resource File 命令,如图 5.18 所示,设定文件名称为 message_item,单击 OK 按钮。

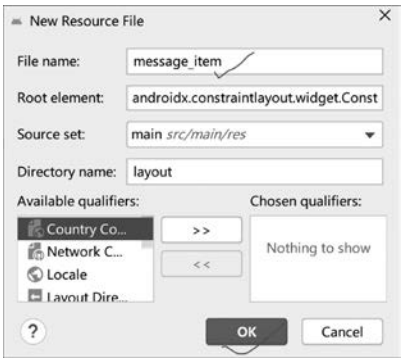

图 5.18 创建单行布局文件

用程序源码 P5.14 所示的脚本程序,定义单行布局。

程序源码 P5.14 message_item.xml 单行布局文件

```
1  <?xml version = "1.0" encoding = "utf - 8"?>
2  < RelativeLayout xmlns:android = "http://schemas.android.com/apk/res/android"
```

```
3       android:layout_width = "match_parent"
4       android:layout_height = "wrap_content">
5       < TextView
6           android:id = "@ + id/tv_message"
7           android:layout_width = "200dp"
8           android:layout_height = "wrap_content"
9           android:layout_margin = "4dp"
10          android:background = "@drawable/send_box"
11          android:backgroundTint = "#26A69A"
12          android:padding = "14dp"
13          android:text = "发出的消息"
14          android:textColor = "@android:color/white"
15          android:textSize = "18sp"
16          android:layout_alignParentEnd = "true" />
17      < TextView
18          android:visibility = "visible"
19          android:id = "@ + id/tv_bot_message"
20          android:layout_width = "200dp"
21          android:layout_height = "wrap_content"
22          android:layout_margin = "4dp"
23          android:background = "@drawable/receive_box"
24          android:backgroundTint = "#FF7043"
25          android:padding = "14dp"
26          android:text = "收到的消息"
27          android:textColor = "@android:color/white"
28          android:textSize = "18sp"
29          android:layout_alignParentStart = "true" />
30  </RelativeLayout >
```

运行主程序,观察主控界面效果。

5.20 Android 聊天逻辑设计

主控逻辑包括两部分：一部分放在 ActivityMain 中；另一部分放在 MessagingAdapter 中。在 ui 包目录下新建消息适配器程序 MessagingAdapter,编码逻辑如程序源码 P5.15 所示。

程序源码 P5.15 MessagingAdapter 消息适配器

```
1   package cn.edu.ldu.androidchatbot.ui
2   import android.view.LayoutInflater
3   import android.view.View
4   import android.view.ViewGroup
5   import androidx.recyclerview.widget.RecyclerView
6   import cn.edu.ldu.androidchatbot.R
7   import cn.edu.ldu.androidchatbot.data.Message
8   import cn.edu.ldu.androidchatbot.utils.Constant.RECEIVE_ID
```

```
import cn.edu.ldu.androidchatbot.utils.Constant.SEND_ID
import kotlinx.android.synthetic.main.message_item.view.*
class MessagingAdapter: RecyclerView.Adapter<MessagingAdapter.MessageViewHolder>() {
    var messageList = mutableListOf<Message>()
    inner class MessageViewHolder(itemView: View): RecyclerView.ViewHolder(itemView) {
        init {
            itemView.setOnClickListener {
                messageList.removeAt(adapterPosition)
                notifyItemRemoved(adapterPosition)
            }
        }
    }
    override fun onCreateViewHolder(parent: ViewGroup,
                                    viewType: Int): MessageViewHolder {
        return MessageViewHolder(
            LayoutInflater.from(parent.context).inflate(
                R.layout.message_item, parent, false))
    }
    override fun onBindViewHolder(holder: MessageViewHolder, position: Int) {
        val currentMessage = messageList[position]
        when (currentMessage.id) {
            SEND_ID -> {
                holder.itemView.tv_message.apply {
                    text = currentMessage.message
                    visibility = View.VISIBLE
                }
                holder.itemView.tv_bot_message.visibility = View.GONE
            }
            RECEIVE_ID -> {
                holder.itemView.tv_bot_message.apply {
                    text = currentMessage.message
                    visibility = View.VISIBLE
                }
                holder.itemView.tv_message.visibility = View.GONE
            }
        }
    }
    override fun getItemCount(): Int {
        return messageList.size
    }
    fun insertMessage(message:Message) {
        this.messageList.add(message)
        notifyItemInserted(messageList.size)
        notifyDataSetChanged()
    }
}
```

ActivityMain 编码逻辑如程序源码 P5.16 所示，负责与用户的交互，包括收发消息及界面滚动。收发消息均通过后台协程实现。

程序源码 P5.16　ActivityMain 主控逻辑

```
package cn.edu.ldu.androidchatbot.ui
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import androidx.recyclerview.widget.LinearLayoutManager
import cn.edu.ldu.androidchatbot.R
import cn.edu.ldu.androidchatbot.data.Answer
import cn.edu.ldu.androidchatbot.data.Message
import cn.edu.ldu.androidchatbot.network.ApiService
import cn.edu.ldu.androidchatbot.utils.Constant.RECEIVE_ID
import cn.edu.ldu.androidchatbot.utils.Constant.SEND_ID
import cn.edu.ldu.androidchatbot.utils.Time
import com.google.gson.Gson
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import okhttp3.MediaType
import okhttp3.RequestBody
import org.json.JSONObject
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
class MainActivity : AppCompatActivity() {
    // 腾讯服务器教学演示地址
    private val BASE_URL = "http://120.53.107.28"
    // private val BASE_URL = "http://192.168.0.103:5000"   // 本地服务器地址
    private val retrofit = Retrofit.Builder()                // 初始化 Retrofit 框架
        .addConverterFactory(GsonConverterFactory.create())
        .baseUrl(BASE_URL)
        .build()
        .create(ApiService::class.java)
    private lateinit var adapter:MessagingAdapter
    var messageList = mutableListOf<Message>()               // 消息列表
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        recyclerView()                                       // 列表视图
        clickEvents()                                        // 单击事件
        customMessage("你好,很高兴见到你!")                    // 欢迎语
    }
    private fun clickEvents() {
        btn_send.setOnClickListener {                        // 发送消息单击事件
            sendMessage()
        }
        input_box.setOnClickListener {                       // 输入消息事件
            GlobalScope.launch {
                withContext(Dispatchers.Main){
```

```
                    chat_view.scrollToPosition(adapter.itemCount - 1)
                }
            }
        }
    }
    override fun onStart() {
        super.onStart()
        GlobalScope.launch {
            withContext(Dispatchers.Main) {
                chat_view.scrollToPosition(adapter.itemCount - 1)
            }
        }
    }
    private fun recyclerView() {                                    // 视图绑定到消息适配器
        adapter = MessagingAdapter()
        chat_view.adapter = adapter
        chat_view.layoutManager = LinearLayoutManager(applicationContext)
    }
    private fun sendMessage() {                                     // 发送消息
        val message = input_box.text.toString()
        val timeStamp = Time.timeStamp()
        if (message.isNotEmpty()) {
            messageList.add(Message(message,SEND_ID,timeStamp))
            input_box.setText("")
            adapter.insertMessage(Message(message,SEND_ID,timeStamp))
            chat_view.scrollToPosition(adapter.itemCount - 1)
            botResponse(message)
        }
    }
    private fun botResponse(message: String) {                      // 接收消息
        val timeStamp = Time.timeStamp()
        GlobalScope.launch(Dispatchers.Main) {
            val request = JSONObject()
            request.put("question", message)
            val body: RequestBody =
                RequestBody.create(
                    MediaType.parse("application/json"),
                    request.toString()
                )
            val response = retrofit.getAnswer(body)
            if (response.isSuccessful) {
                val json: String = response.body()!!.string()
                var gson = Gson()
                var reply = gson.fromJson(
                    json,
                    Answer::class.java
                )
                messageList.add(Message(reply.answer,RECEIVE_ID,timeStamp))
                adapter.insertMessage(Message(reply.answer,RECEIVE_ID,timeStamp))
                chat_view.scrollToPosition(adapter.itemCount - 1)
            }else{ }
        }
```

```
100      }
101      private fun customMessage(message: String) {              // 自定义欢迎消息
102          GlobalScope.launch {
103              val timeStamp = Time.timeStamp()
104              withContext(Dispatchers.Main){
105                  messageList.add(Message(message,RECEIVE_ID,timeStamp))
106                  adapter.insertMessage(Message(message,RECEIVE_ID,timeStamp))
107                  chat_view.scrollToPosition(adapter.itemCount - 1)
108              }
109          }
110      }
111  }
```

程序解析参见本节视频教程。

5.21 客户机与服务器联合测试

现在可以开始项目联合测试了。先运行 Web 服务器,再运行客户机。可以用模拟器测试,也可以用真机测试。既可以本地测试,也可以远程测试。

如果做本地测试,首先运行 Server 目录下的服务器程序 app.py,观察控制台上输出的服务器地址,将程序源码 P5.16 中第 26 行语句中的地址修改为 Web API 运行的地址,注释掉第 25 行中的远程服务器地址,然后做客户机与服务器的本地联合测试。

如果做远程测试,可以采用书中的腾讯服务器地址,直接运行 Android 客户机程序,在模拟器与真机上与远程服务器做人机畅聊测试。图 5.19 给出了一组随机对话,显示了

(a) 与本地服务器测试　　(b) 与远程服务器测试

图 5.19　真机与本地服务器和远程服务器通信的测试效果

用真机分别与本地服务器和远程服务器通信的测试效果。

至此，本章实现的聊天机器人已经初露锋芒。无论其聊天水平怎么样，对于一个仅有900万训练参数的小模型，依靠一个超小规模语料库，实现的语言对答还是令人惊讶的。今天的一小步，孕育着未来的一大步。

事实上，图5.19展示的人机聊天，是故意增加了难度的。首先，语料库中关于健康和航天类的语料是偏少的。其次，对于第一组聊天，涉及奥密克戎这个新生词汇，超过了模型的“知识范畴”；对于第二组聊天，航天的话题无疑也超过了模型的“知识范畴”。

可以试试体育类的话题，机器人的回答则会生动得多，有趣得多。做完这个项目，逗着自己缔造的机器人玩玩，成就感与幸福感如期而至。原来，是如此容易让机器说话的呀。更多测试细节，参见本节视频教程。

5.22 小结

本章以人机畅聊的境界追求为动力，遵循机器问答的技术设计路线，完成了Transformer聊天模型＋Web API＋Android聊天客户机的项目设计。本章项目对于提升读者在自然语言智能领域的理论水平和实践能力，具备非常好的教学示范效果。

沿着语料库分析，Transformer模型解析，聊天模型建模、训练、评估，Web服务器模型部署以及Android客户机设计这些环环相扣的步骤，读者可以体验到学习过程中的循序渐进，体验到量变到质变，体验到从混沌到顿悟、从顿悟到彻悟的华丽蜕变。

机器是如何学习说话的？机器是如何学会说话的？机器说话的本质是什么？现阶段的局限是什么？所有这些不再是秘密。

5.23 习题

1. 简要描述机器问答与机器聊天的不同。
2. 简要描述实施机器问答项目的方法与步骤。
3. 腾讯自然语言聊天数据集NaturalConv有哪些特点？
4. Transformer的创新点有哪些？
5. 简要解析Transformer单头注意力的计算逻辑。
6. Transformer模型适合哪些场景的应用？
7. 结合项目实战，谈谈为什么本章实现的聊天模型偏爱体育方面的表达。
8. 根据Transformer多头注意力机制的编程设计，绘制其计算逻辑流程图。
9. 绘制Transformer单层编码器的工作流程图。
10. 绘制Transformer单层解码器的工作流程图。
11. 模型训练期间，采用动态学习率调整策略有何优势？
12. Transformer聊天机器人中的损失函数是如何定义的？

13. 描述 Transformer 聊天机器人在 Web 服务器上的部署流程。

14. 简述 Android 聊天机器人的客户机界面布局设计。

15. 绘图说明 Android 聊天机器人的客户机运行逻辑。

16. 为什么 Android 客户机的收发消息流程需要定义到协程中?

17. 结合本章项目实战,谈谈机器是如何学习说话的、机器是如何学会说话的、机器说话的本质是什么。