尚硅谷大模型项目实战之智选新闻

（作者：尚硅谷研究院）

版本：V1.0

第 1 章 项目架构

1.1 项目简介

本项目基于 BART 预训练模型根据文本分类和摘要任务微调，用于实现新闻分类和新闻摘要。通过 Transformers 与 PyTorch 进行模型的搭建与训练，通过 FastAPI 提供 Web 服务。

1.2 应用场景

新闻推荐系统：基于用户行为推荐符合用户喜好的新闻类型。

新闻聚合网站：将不同新闻内容按照类别整理好展示给读者。

媒体编辑：对大量新闻稿做分类处理，满足媒体机构内部的需求。

订阅服务：用户通过个人喜好，订阅自己感兴趣的新闻主题。

搜索引擎优化：搜索引擎根据分类结果预先设定规则以达到快速定位相关内容的效果。

1.3 技术栈

深度学习框架：PyTorch

数据处理：pandas，datasets，Numpy

预训练模型使用：transformers

可视化：TensorBoard

Web 框架：FastAPI

1.4 项目准备

1.4.1 环境准备

1）创建 conda 虚拟环境

终端输入如下命令，创建项目的虚拟环境，并指定 Python 版本：

```
conda create -n news-classify-summarize python=3.12
```

激活虚拟环境：

```
conda activate news-classify-summarize
```

更换国内源：

```
pip config set global.index-url
https://pypi.tuna.tsinghua.edu.cn/simple
```

根据 CUDA 版本选择 PyTorch 版本并安装：

```
pip3 install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu126
```

安装其他所需的库：

```
pip install pandas matplotlib scikit-learn tqdm transformers datasets
tensorboard evaluate rouge-score nltk fastapi uvicorn
```

1.4.2 模型准备

BART 预训练模型。

1.4.3 项目目录结构

```
./
├── data/                    /原始数据集
│   └── news.csv
├── pretrained/              /预训练模型
│   └── bart-base-chinese/
├── rouge/                   /摘要评估指标
│   ├── rouge.json
│   └── rouge.py
├── templates/               /网页模板
│   └── index.html
├── finetuned/               /模型参数存储路径
├── common.py                /公共模块
├── models_def.py            /模型定义
├── preprocess.py            /数据预处理
├── train.py                 /模型训练
├── main.py                  /主程序
└── app.py                   /Web 模块
```

第 2 章 公共模块

common.py

存放项目的配置信息。

定义函数，用于固定所有的随机数种子，确保结果可复现。

```
import torch
import random
import numpy as np
```

```
class Config:
    DATA_PATH = "data/news.csv"
    CATEGORY_LIST = ["财经", "社会", "教育", "科技", "时政", "体育", "游戏
"]
    BART_PATH = "pretrained/bart-base-chinese"
    DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")


def set_seed(seed):
    """设置随机数种子"""
    # 设置 python 随机数种子
    random.seed(seed)
    # 设置 numpy 随机数种子，pandas 也会使用这个种子
    np.random.seed(seed)
    # 设置 torch 随机数种子
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
```

第 3 章 数据预处理模块

preprocess.py

```
import torch
import pandas as pd
from datasets import Dataset
from transformers import AutoTokenizer
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence


def process(
    task,
    data_path,
    max_examples,
    batch_size,
    tokenizer,
    train_ratio=0.8,
    test_ratio=0.1,
    category_list=None,
):
    """
    数据预处理
```

```
    参数:
    - task: 任务类型，["classify","summarize"]
    - data_path: 数据路径
    - train_ratio: 训练集比例
    - test_ratio: 测试集比例
    - max_examples: 最大样本数
    - batch_size: 批次大小
    - tokenizer: 分词器
    - category_list: 分类类别列表

    返回值:
    - dataloader: 数据加载器
    """

    def _map_fn(batch):
        fn = {
            "input_ids": tokenizer(batch["text"], max_length=1024,
truncation=True)[
                "input_ids"
            ]
        }
        if task == "summarize":
            fn["summary"] = tokenizer(
                batch["summary"], max_length=128, truncation=True
            )["input_ids"]
        elif task == "classify":
            fn["category"] = [category_map[cat] for cat in
batch["category"]]
        return fn

    def _collate_fn(batch):
        input_ids = [torch.tensor(x["input_ids"]) for x in batch]
        input_ids = pad_sequence(input_ids, True, tokenizer.pad_token_id)
        attention_mask = (input_ids != tokenizer.pad_token_id).int()
        if task == "summarize":
            labels = [torch.tensor(x["summary"]) for x in batch]
            labels = pad_sequence(labels, True, -100)
        elif task == "classify":
            labels = torch.tensor([x["category"] for x in batch])
        return {
            "input_ids": input_ids,
            "attention_mask": attention_mask,
            "labels": labels,
        }
```

```
    if task == "classify":
        # 如果是分类任务，检查是否提供分类类别列表
        assert category_list is not None, "缺少分类类别列表参数"
        # 从数据中读取文本和分类类别
        columns = ["text", "category"]
        # 定义分类类别到数字的映射
        category_map = {cat: i for i, cat in enumerate(category_list)}
    elif task == "summarize":
        # 从数据中读取文本和摘要
        columns = ["text", "summary"]
    else:
        raise ValueError("任务类型须为 classify 或 summarize")

    # 读取数据，随机采样
    df = pd.read_csv(data_path)[columns].dropna()
    df = df.sample(min(len(df), max_examples))
    dataset = Dataset.from_pandas(df)

    # tokenize 处理，分出训练集、验证集、测试集
    dataset = dataset.map(_map_fn, batched=True)
    train_size = int(dataset.num_rows * train_ratio)
    dataset = dataset.train_test_split(test_size=test_ratio)
    dataset["train"], dataset["valid"] = (
        dataset["train"].train_test_split(train_size=train_size).values(
)
    )
    # 转换为DataLoader
    return {
        phase: DataLoader(
            dataset=dataset[phase],
            batch_size=batch_size,
            collate_fn=_collate_fn,
            shuffle=(phase == "train"),
        )
        for phase in ["train", "test", "valid"]
    }


if __name__ == "__main__":
    from common import Config
    from transformers import AutoTokenizer

    dataloader = process(
```

```
        task="summarize",
        data_path=Config.DATA_PATH,
        train_ratio=Config.TRAIN_RATIO,
        test_ratio=Config.TEST_RATIO,
        max_examples=200,
        batch_size=Config.BATCH_SIZE,
        tokenizer=AutoTokenizer.from_pretrained(Config.BART_PATH),
        category_list=Config.CATEGORY_LIST,
    )
    print(next(iter(dataloader["train"])))
```

第 4 章 模型搭建模块

4.1 束搜索介绍

束搜索（Beam Search）是一种启发式搜索算法，广泛应用于自然语言处理（NLP）任务中，特别是在序列生成任务中。束搜索是一种改进的贪心搜索算法，它在一定程度上平衡了搜索效率和结果质量。

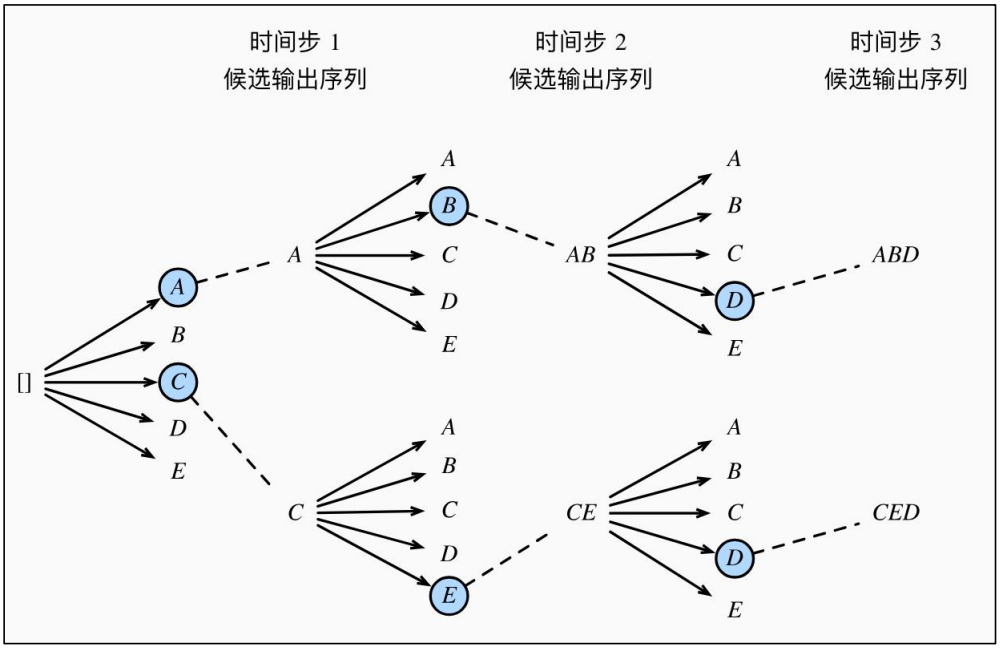
在序列生成任务中，模型需要从一个可能的词汇表中逐步生成输出序列。理想情况下，我们希望找到全局最优解（即概率最高的完整序列），但这需要穷举所有可能的序列组合，计算复杂度极高。

常见的搜索方式有如下几种：

- 贪心搜索（Greedy Search）：每次只选择当前时间步概率最高的词，虽然计算简单，但容易陷入局部最优。
- 穷举搜索（Exhaustive Search）：遍历所有可能的序列组合，理论上能找到最优解，但计算成本过高，无法实际应用。
- 束搜索（Beam Search）：在贪心搜索和穷举搜索之间折中，通过限制候选序列的数量（即“束宽”），在保证一定搜索效率的同时提升生成质量。

4.1.1 束搜索的基本原理

束搜索的核心思想是：在每一步生成时，保留若干个最高概率的部分序列（称为“候选束”），并在下一步继续扩展这些候选序列，直到生成完整的序列。



1）初始化

在序列生成的第一步，模型会计算每个可能词的概率，并选择概率最高的 k 个词作为初始候选序列（k 是束宽）。

2）扩展候选序列

对于每个候选序列，模型预测下一个时间步的所有可能词，并计算扩展后的序列概率。

扩展后的序列概率通常是基于累积概率计算的，例如：

$$P(sequence) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1, w_2) \cdots$$

或者取对数概率以避免数值下溢：

$$\log P(sequence) = \log P(w_1) + \log P(w_2|w_1) + \log P(w_3|w_1, w_2) + \cdots$$

3）剪枝

在每一步扩展后，保留概率最高的 k 个候选序列，丢弃其他低概率序列。

4）终止条件

当某个候选序列生成了结束标记（如<EOS>），该序列被视为完成。

搜索过程持续进行，直到所有候选序列都生成了结束标记或达到最大长度限制。

5）选择最终结果

从所有完成的候选序列中，选择概率最高的序列作为最终输出。

4.1.2 束搜索的优缺点

1）优点：

高效：相比穷举搜索，束搜索显著降低了计算复杂度。

质量提升：相比贪心搜索，束搜索能够探索更多的候选路径，从而提高生成序列的质量。

灵活：束宽可以根据任务需求调整，以平衡效率和质量。

2）缺点

次优解问题：束搜索仍然是一个启发式算法，不能保证找到全局最优解。

束宽限制：较小的束宽可能导致错过高质量的候选序列；较大的束宽则会显著增加计算开销。

缺乏多样性：束搜索倾向于生成高概率但较为保守的序列，可能导致输出缺乏新颖性或多样性。

4.2 models_def.py

分类模型：在 BART 预训练模型编码器基础上添加全连接层，用于分类任务。

摘要模型：在 BART 预训练模型基础上添加全连接层作为语言模型头，推理过程中使用束搜索进行生成。

```
import torch
import torch.nn as nn
from transformers import AutoConfig, AutoTokenizer, BartModel


def compute_parameters(self):
    """统计模型参数量"""
    print(
        f"{self.__class__.__name__}参数量:{sum(p.numel() for p in
self.parameters() if p.requires_grad):,}"
    )


def load_params(self, model_params_path):
    """
    加载模型参数

    参数:
    - model_params_path: 模型参数文件路径
    """
    try:
        self.load_state_dict(torch.load(model_params_path))
    except FileNotFoundError:
        print("模型参数文件不存在，使用默认参数")
    except RuntimeError:
```

```
        self.load_state_dict(torch.load(model_params_path,
map_location="cpu"))


class ClassifyModel(nn.Module):
    """分类模型"""

    compute_parameters = compute_parameters
    load_params = load_params

    def __init__(self, model_name: str, category_list: list):
        super().__init__()
        self.category_list = category_list
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)

        # 只加载 BART encoder 部分
        self.encoder = BartModel.from_pretrained(model_name).encoder
        self.classifier = nn.Linear(self.encoder.config.hidden_size,
len(category_list))

        # 使用与预训练模型相同的标准差参数初始化分类器，如果没有则默认使用 0.02
        init_std = getattr(self.encoder.config, "init_std", 0.02)
        self.classifier.weight.data.normal_(mean=0.0, std=init_std)
        if self.classifier.bias is not None:
            self.classifier.bias.data.zero_()

        self.loss_fn = nn.CrossEntropyLoss()

    def forward(self, input_ids, attention_mask=None, labels=None):
        output = self.encoder(input_ids=input_ids,
attention_mask=attention_mask)
        cls_hidden = output.last_hidden_state[:, 0, :]
        logits = self.classifier(cls_hidden)
        loss = self.loss_fn(logits, labels) if labels is not None else None
        return {"loss": loss, "logits": logits}

    @torch.inference_mode()
    def predict(self, text, device=torch.device("cpu"), batch_size=8):
        self.eval()
        self.to(device)

        res: list[str] = []
        # 统一转换为列表
        input_texts = text if isinstance(text, list) else [text]
```

```
        # 逐批次处理
        for i in range(0, len(input_texts), batch_size):
            batch_texts = input_texts[i : i + batch_size]
            inputs = self.tokenizer(
                batch_texts,
                max_length=1024,
                truncation=True,
                padding=True,
                return_tensors="pt",
            ).to(device)
            outputs = self.forward(inputs["input_ids"],
inputs["attention_mask"])
            logits = outputs["logits"]
            batch_res = [
                self.category_list[int(cat.item())]
                for cat in torch.argmax(logits, dim=1)
            ]
            res.extend(batch_res)

        return res if isinstance(text, list) else res[0]


class CustomSummarizeModel(nn.Module):
    """摘要模型"""

    compute_parameters = compute_parameters
    load_params = load_params

    def __init__(self, model_name: str):
        super().__init__()
        self.config = AutoConfig.from_pretrained(model_name)
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = BartModel.from_pretrained(model_name)
        # 语言模型头
        self.lm_head = nn.Linear(self.config.d_model,
self.config.vocab_size)
        # 将语言模型头权重设置为共享权重，共享嵌入层权重
        self.lm_head.weight = self.model.shared.weight
        self.loss_fn = nn.CrossEntropyLoss(ignore_index=-100)

    def forward(self, input_ids, attention_mask=None, labels=None):
        decoder_input_ids = None
        decoder_attention_mask = None
        # 如果提供了标签，将标签右移一位拼接在起始符之后，作为解码器输入
```

```
        if labels is not None:
            decoder_input_ids = labels.new_zeros(labels.shape)
            decoder_input_ids[:, 1:] = labels[:, :-1].clone()
            decoder_input_ids[:, 0] = self.config.decoder_start_token_id
            # 修改输入中的 -100 为 pad_token_id，防止 embedding 时报错
            decoder_input_ids.masked_fill_(
                decoder_input_ids == -100, self.config.pad_token_id
            )
            decoder_attention_mask = (
                decoder_input_ids != self.config.pad_token_id
            ).long()

        # 编码器前向传播
        encoder_outputs = self.model.encoder(input_ids, attention_mask)
        # 解码器前向传播
        outputs = self.model.decoder(
            input_ids=decoder_input_ids,
            attention_mask=decoder_attention_mask,
            encoder_hidden_states=encoder_outputs.last_hidden_state,
            encoder_attention_mask=attention_mask,
            use_cache=False,
        )

        logits = self.lm_head(outputs.last_hidden_state)
        loss = None
        if labels is not None:
            loss = self.loss_fn(
                logits.view(-1, self.config.vocab_size), labels.view(-1)
            )
        return {"loss": loss, "logits": logits}

    @torch.inference_mode()
    def generate(
        self,
        input_ids,
        attention_mask=None,
        max_length=128,
        num_beams=2,
    ):
        self.eval()
        # 贪婪搜索
        # return self.greedy_search(input_ids, attention_mask, max_length)
        # 束搜索
```

```
        return self.beam_search(input_ids, attention_mask, max_length,
num_beams)

    def beam_search(self, input_ids, attention_mask, max_length,
num_beams):
        """束搜索"""
        device = input_ids.device
        batch_size = input_ids.size(0)
        vocab_size = self.config.vocab_size

        # 编码器前向传播
        encoder_outputs = self.model.encoder(input_ids, attention_mask)
        # 复制编码器输出、编码器注意力掩码以匹配束数量
        encoder_hidden_states =
encoder_outputs.last_hidden_state.repeat_interleave(
            num_beams, dim=0
        )
        encoder_attention_mask =
attention_mask.repeat_interleave(num_beams, dim=0)

        # 初始化解码输入为起始符
        # decoder_input_ids:[batch_size * beam, 1]
        decoder_input_ids = torch.full(
            (batch_size * num_beams, 1),
            self.config.decoder_start_token_id,
            dtype=torch.long,
            device=device,
        )

        # 定义束索引偏移量
        beam_offset = torch.arange(batch_size, device=device) * num_beams
        # 初始化束分数，起初只有每个样本的第一个束有效
        beam_scores = torch.full((batch_size * num_beams,), -1e9,
device=device)
        beam_scores[beam_offset] = 0
        # 初始化完成状态
        done = torch.zeros(batch_size * num_beams, dtype=torch.bool,
device=device)

        for step in range(max_length):
            # 解码当前输入
            decoder_outputs = self.model.decoder(
                input_ids=decoder_input_ids,
                encoder_hidden_states=encoder_hidden_states,
```

```
                encoder_attention_mask=encoder_attention_mask,
                use_cache=False,
            )
            # 取最后一个时间步的 logits
            logits = self.lm_head(decoder_outputs.last_hidden_state[:,
-1, :])
            # 计算每个候选的得分
            log_probs = nn.functional.log_softmax(logits, dim=-1)

            # 禁止已经结束的束继续生成
            log_probs[done] = -float("inf")
            # 令已完成的束的 eos 对应的 log_prob 为 0，使其只能生成 eos，同时
也保证已完成的束得分不变
            log_probs[done, self.config.eos_token_id] = 0

            # 累积每个束的得分
            log_probs += beam_scores.view(-1).unsqueeze(1)
            # 重塑得分矩阵
            log_probs = log_probs.view(batch_size, num_beams * vocab_size)
            # 每个样本选择 top num_beams 个候选
            beam_scores, indices = torch.topk(log_probs, num_beams, dim=1)
            indices = indices.view(-1)

            # 获取候选对应的束索引和 token id
            beam_indices = indices // vocab_size
            beam_indices += beam_offset.repeat_interleave(num_beams,
dim=0)
            token_ids = indices % vocab_size

            # 更新输入序列
            decoder_input_ids = torch.cat(
                [decoder_input_ids[beam_indices], token_ids.view(-1, 1)],
dim=1
            )

            # 更新完成状态，若所有序列完成则提前终止
            done = token_ids.eq(self.config.eos_token_id) |
done[beam_indices]
            if done.all():
                break

        # 选择每个样本 beam_score 最大的束
        best_indices = beam_scores.argmax(dim=-1) + beam_offset
        return decoder_input_ids[best_indices]
```

```
def greedy_search(self, input_ids, attention_mask, max_length):
    """贪婪搜索"""
    batch_size = input_ids.size(0)
    device = input_ids.device

    # 编码器前向传播
    encoder_outputs = self.model.encoder(input_ids, attention_mask)

    # 初始化解码器输入
    decoder_input_ids = torch.full(
        (batch_size, 1),
        self.config.decoder_start_token_id,
        dtype=torch.long,
        device=device,
    )

    # 初始化完成状态
    done = torch.zeros(batch_size, dtype=torch.bool, device=device)

    # 逐步解码
    for step in range(max_length):
        # 构建 attention mask
        decoder_attention_mask = (
            decoder_input_ids != self.config.pad_token_id
        ).long()

        # 解码器前向传播
        decoder_outputs = self.model.decoder(
            input_ids=decoder_input_ids,
            attention_mask=decoder_attention_mask,
            encoder_hidden_states=encoder_outputs.last_hidden_state,
            encoder_attention_mask=attention_mask,
            use_cache=False,
        )
        logits = self.lm_head(decoder_outputs.last_hidden_state)

        # 取最后一步输出
        next_token_logits = logits[:, -1, :]
        next_tokens = torch.argmax(next_token_logits, dim=-1)

        # 更新输出序列
        decoder_input_ids = torch.cat(
            [decoder_input_ids, next_tokens.unsqueeze(1)], dim=1
```

```
            )

            # 检查是否全部生成结束
            done |= next_tokens == self.config.eos_token_id
            if done.all():
                break

        return decoder_input_ids

    def predict(self, text, device=torch.device("cpu"), batch_size=8):
        self.eval()
        self.to(device)

        res: list[str] = []
        # 统一转换为列表
        input_texts = text if isinstance(text, list) else [text]
        # 逐批次处理
        for i in range(0, len(input_texts), batch_size):
            batch_texts = input_texts[i : i + batch_size]
            inputs = self.tokenizer(
                batch_texts,
                max_length=1024,
                truncation=True,
                padding=True,
                return_tensors="pt",
            ).to(device)
            outputs = self.generate(
                input_ids=inputs["input_ids"],
                attention_mask=inputs["attention_mask"],
            )
            batch_res = self.tokenizer.batch_decode(
                outputs, skip_special_tokens=True,
clean_up_tokenization_spaces=True
            )
            batch_res = [a_res.replace(" ", "") for a_res in batch_res]
            res.extend(batch_res)
        return res if isinstance(text, list) else res[0]
```

第 5 章 训练模块

train.py

定义训练器类，用于训练、验证与测试。

```
import os
import tqdm
```

```
import torch
from evaluate import load
import torch.optim as optim
from torch.amp.autocast_mode import autocast
from torch.amp.grad_scaler import GradScaler
from sklearn.metrics import classification_report, roc_auc_score


class Trainer:
    """训练验证与测试"""

    def __init__(self, model, device, epochs, learning_rate,
checkpoint_steps=400):
        """
        参数:
        - model: 模型
        - device: 设备
        - epochs: 训练轮数
        - learning_rate: 学习率
        - checkpoint_steps: 多少步之后保存检查点
        """
        self.model = model
        self.device = device
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.checkpoint_steps = checkpoint_steps

        self.optimizer = optim.AdamW(self.model.parameters(),
lr=self.learning_rate)

    def __call__(
        self,
        dataloader,
        model_params_path=None,
        writer=None,
        is_test=False,
    ):
        """
        训练验证与测试

        参数:
        - dataloader: 数据加载器
        - model_params_path: 模型参数保存路径
        - writer: 记录器
```

```
        - is_test: 是否执行测试
        """
        self.dataloader = dataloader
        self.model_params_path = model_params_path
        self.writer = writer
        self.is_test = is_test

        self.model.to(self.device)
        self.global_step = 0

        # 测试
        if is_test:
            for k, v in self.run_epoch("test").items():
                print(f"Test {k}:", v)
            return

        # 训练并验证
        assert self.model_params_path is not None
        # 初始化梯度缩放器
        use_amp = self.device.type == "cuda"
        scaler = GradScaler() if use_amp else None
        best_valid_loss = float("inf")
        for epoch in range(self.epochs):
            print(f"Epoch: {epoch}")

            train_metrics = self.run_epoch("train", epoch, use_amp, scaler)
            for k, v in train_metrics.items():
                print(f"Train {k}:", v)

            valid_metrics = self.run_epoch("valid", epoch)
            for k, v in valid_metrics.items():
                print(f"Valid {k}:", v)

            # 保存最佳模型
            if valid_metrics["loss"] <= best_valid_loss:
                best_valid_loss = valid_metrics["loss"]
                parent_dir = os.path.dirname(self.model_params_path)
                if not os.path.exists(parent_dir):
                    os.makedirs(parent_dir)
                torch.save(self.model.state_dict(),
self.model_params_path)

    def run_epoch(self, phase, epoch=0, use_amp=False, scaler=None):
        self.model.train() if phase == "train" else self.model.eval()
```

```
        # 初始化总损失值和总样本数
        total_loss = 0.0
        total_examples = 0
        # 初始化记录
        records = {}

        with torch.set_grad_enabled(phase == "train"):
            for inputs in tqdm.tqdm(self.dataloader[phase], desc=phase):
                # 数据转移到设备
                inputs = {k: v.to(self.device) for k, v in inputs.items()}

                # 前向传播（如果可以，使用混合精度）
                with autocast(device_type=self.device.type,
enabled=use_amp):
                    outputs, loss = self.forward(inputs, phase)

                # 反向传播和优化（仅训练阶段）
                if phase == "train":
                    self.optimizer.zero_grad()
                    if scaler:
                        scaler.scale(loss).backward()
                        scaler.step(self.optimizer)
                        scaler.update()
                    else:
                        loss.backward()
                        self.optimizer.step()

                    # 向 TensorBoard 写入损失
                    if self.writer:
                        self.writer.add_scalar(
                            f"Loss/{phase}", loss.item(), self.global_step
                        )

                    self.global_step += 1

                    # 保存模型参数
                    if (
                        self.checkpoint_steps
                        and self.global_step % self.checkpoint_steps == 0
                    ):
                        checkpoint_path = str(self.model_params_path) +
".checkpoint"
                        torch.save(self.model.state_dict(),
checkpoint_path)
```

```
                # 记录损失
                current_batch_size = inputs["input_ids"].size(0)
                total_loss += loss.item() * current_batch_size
                total_examples += current_batch_size

                # 更新记录，用于评估
                if phase != "train":
                    self.update_records(inputs, outputs, records)

        # 计算平均损失
        avg_loss = total_loss / total_examples
        metrics = {"loss": avg_loss}

        # 计算评估指标
        if phase != "train":
            self.compute_metrics(metrics, records)
            if self.writer:
                for metric_name, value in metrics.items():
                    self.writer.add_scalar(f"{phase}/{metric_name}",
value, epoch)
        return metrics

    def forward(self, inputs, phase):
        """前向传播"""
        raise NotImplementedError

    def update_records(self, inputs, outputs, records):
        """更新记录"""
        raise NotImplementedError

    def compute_metrics(self, metrics, records):
        """计算评估指标"""
        raise NotImplementedError


class ClassifyTrainer(Trainer):
    def forward(self, inputs, phase):
        """前向传播"""
        outputs = self.model(
            input_ids=inputs["input_ids"],
            attention_mask=inputs["attention_mask"],
            labels=inputs["labels"],
        )
```

```
        return outputs, outputs["loss"]

    def update_records(self, inputs, outputs, records):
        """更新记录"""
        logits = outputs["logits"]
        probs = torch.softmax(logits, dim=1).detach().cpu()
        preds = logits.argmax(dim=1).detach().cpu()
        labels = inputs["labels"].detach().cpu()
        records.setdefault("probs", []).append(probs)
        records.setdefault("preds", []).append(preds)
        records.setdefault("labels", []).append(labels)

    def compute_metrics(self, metrics, records):
        """计算评估指标"""
        all_probs = torch.cat(records["probs"])
        all_preds = torch.cat(records["preds"])
        all_labels = torch.cat(records["labels"])
        report = classification_report(
            all_labels, all_preds, output_dict=True, zero_division=0
        )
        metrics["accuracy"] = report["accuracy"]
        metrics.update(report["weighted avg"])
        auc = roc_auc_score(
            all_labels, all_probs, multi_class="ovr",
labels=list(range(7))
        )
        metrics["auc"] = auc


class SummarizeTrainer(Trainer):
    def forward(self, inputs, phase):
        """前向传播"""
        outputs = {"loss": torch.tensor(0.0)}
        if phase != "test":
            outputs = self.model(
                input_ids=inputs["input_ids"],
                attention_mask=inputs["attention_mask"],
                labels=inputs["labels"],
            )
        if phase != "train":
            outputs["generated_ids"] = self.model.generate(
                inputs["input_ids"], inputs["attention_mask"]
            )
        return outputs, outputs["loss"]
```

```
    def update_records(self, inputs, outputs, records):
        """更新记录"""
        preds = self.model.tokenizer.batch_decode(
            outputs["generated_ids"], skip_special_tokens=True
        )
        labels = self.model.tokenizer.batch_decode(
            torch.where(
                inputs["labels"] == -100,
                self.model.tokenizer.pad_token_id,
                inputs["labels"],
            ),
            skip_special_tokens=True,
        )
        records.setdefault("preds", []).extend(preds)
        records.setdefault("labels", []).extend(labels)

    def compute_metrics(self, metrics, records):
        """计算评估指标"""
        # ROUGE(Recall-Oriented Understudy for Gisting Evaluation，面向召回率的摘要评估)
        # 通过比较生成文本与参考文本之间的 n-gram、词序列、最长公共子序列等重叠程度来进行评估
        # 评估内容是否覆盖参考文本的关键信息
        # 常用于摘要任务的评估
        rouge_scores = load("rouge").compute(
            predictions=records["preds"],
            references=records["labels"],
            tokenizer=self.model.tokenizer.tokenize,
        )
        metrics.update(rouge_scores)
```

第 6 章 主程序

main.py

执行数据预处理、模型搭建、模型训练验证测试、模型推理。

```
from datetime import datetime
from preprocess import process
from common import Config, set_seed
from train import ClassifyTrainer, SummarizeTrainer
from torch.utils.tensorboard.writer import SummaryWriter
from models_def import ClassifyModel, CustomSummarizeModel

set_seed(42)
```

```
max_examples = 100000
batch_size = 8
epochs = 2
learning_rate = 5e-5
device = Config.DEVICE
model_name = Config.BART_PATH
category_list = Config.CATEGORY_LIST
data_path = Config.DATA_PATH


def model_go(task, train=None, test=None, inference=None,
model_params_path=None):
    assert task in ["classify", "summarize"], "任务类型必须为 classify 或
summarize"
    match task:
        case "classify":
            model = ClassifyModel(model_name, category_list)
            trainer = ClassifyTrainer(model, device, epochs, learning_rate)
        case "summarize":
            model = CustomSummarizeModel(model_name)
            trainer = SummarizeTrainer(model, device, epochs,
learning_rate)
    if train or test:
        dataloader = process(
            task,
            data_path,
            max_examples,
            batch_size,
            model.tokenizer,
            category_list=category_list,
        )

    writer = None
    this_id = datetime.now().strftime("%Y%m%d%H%M%S")
    model.load_params(model_params_path)

    if train:
        writer = SummaryWriter(f"logs/{task}-{this_id}")
        model_params_path = f"finetuned/{task}-{this_id}.pt"
        trainer(dataloader, model_params_path, writer)

    if test:
        trainer(dataloader, writer=writer, is_test=True)
```

```
    if writer:
        writer.close()

    if inference:
        return model.predict(text, device)


text = [
    "4 月 13 日，全球首个人形机器人半程马拉松赛将在北京亦庄举行。赛事由北京市体育局、北京市经济和信息化局、中央广播电视总台北京总站、北京经济技术开发区管理委员会等单位联合主办。随着比赛临近，参赛机器人在跑道上进行了首次路测。路测期间，机器人们表现如何？",
    "据日本鹿儿岛地方气象台消息，当地时间 3 日 13 时 49 分左右，位于鹿儿岛县和宫崎县交界地区雾岛山的新燃岳火山喷发，火山灰柱最大高度达 5000 米。",
    "自 7 月 7 日开始，在中国人民抗日战争纪念馆举办“为了民族解放与世界和平—纪念中国人民抗日战争暨世界反法西斯战争胜利 80 周年主题展览”，展出照片 1525 张、文物 3237 件。主题展览将作为基本陈列长期展出。",
    "近日，印度中央邦博帕尔市一座铁路立交桥引发全球关注。这座桥的致命缺陷并非偷工减料或结构坍塌，而是惊现 90 度直角转弯，司机必须紧急刹车才能通过，被网友调侃为“现实版神庙逃亡赛道”。更讽刺的是，这座桥至今尚未通车，却已因设计失误导致 7 名工程师停职、两家建筑公司被列入黑名单。",
]

category = model_go("classify", 1, 1, 1, "finetuned/classify.pt")
summary = model_go("summarize", 1, 1, 1, "finetuned/summarize.pt")

for t, c, s in zip(text, category, summary):
    print(f"文本：{t}\n 类别：{c}\n 摘要：{s}\n")
```

第 7 章 Web 模块

app.py

```
import uvicorn
from common import Config
from fastapi import FastAPI
from pydantic import BaseModel, Field
from fastapi.responses import FileResponse
from fastapi.staticfiles import StaticFiles
from models_def import ClassifyModel, CustomSummarizeModel

classify_model = ClassifyModel(Config.BART_PATH, Config.CATEGORY_LIST)
classify_model.load_params("finetuned/classify.pt")

summarize_model = CustomSummarizeModel(Config.BART_PATH)
```

```
summarize_model.load_params("finetuned/summarize.pt")

app = FastAPI(debug=True)
app.mount("/static", StaticFiles(directory="templates"), name="static")


# 请求体模型
class NewsClassifySummarizeRequest(BaseModel):
    content: str = Field(..., example="新闻内容")


# 响应体模型
class NewsClassifySummarizeResponse(BaseModel):
    category: str = Field(..., example="分类")
    summary: str = Field(..., example="摘要")


@app.get("/")
async def homepage():
    return FileResponse("templates/index.html")


@app.post("/news_classify_summarize")
async def submit(
    request: NewsClassifySummarizeRequest,
) -> NewsClassifySummarizeResponse:
    content = request.content
    category = classify_model.predict(content)
    summary = summarize_model.predict(content)
    return NewsClassifySummarizeResponse(category=category,
summary=summary)


if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8089)
```