基于GPT2搭建医疗问诊机器人

学习目标

- 理解医疗问诊机器人的开发背景.
- 了解企业中聊天机器人的应用场景
- 掌握基于GPT2模型搭建医疗问诊机器人的实现过程

项目背景

- 聊天机器人是一种基于自然语言处理技术的智能对话系统，能够模拟人类的自然语言交流，与用户进行对话和互动。聊天机器人能够理解用户的问题或指令，并给出相应的回答或建议。其目标是提供友好、智能、自然的对话体验.
- 当前，聊天机器人在多个领域得到广泛应用。首先，它们常用于在线客服系统，能够快速、准确地回答用户的常见问题，解决疑问。其次，聊天机器人可以作为个人助手，提供个性化的推荐、建议和日程安排等服务，提升用户体验。此外，聊天机器人还被应用于社交娱乐、语言学习、旅游指南等领域，为用户提供有趣、便捷的对话体验.
- 常见的相关聊天机器人产品：

小冰

小密

小度

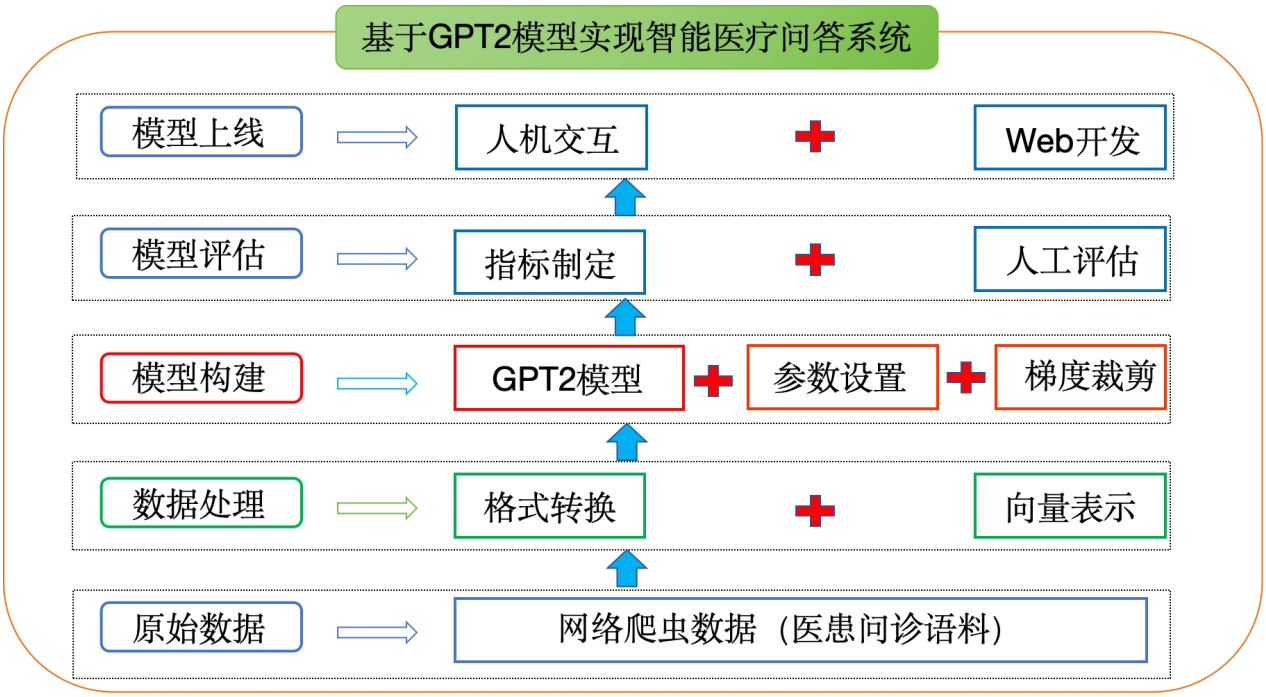
> 微软小冰：微软公司开发。它具备自然语言处理、情感分析和对话生成等功能，能够与用户进行智能对话，提供情感支持和娱乐等服务。
>
> 阿里云小蜜：阿里云公司推出，提供了丰富的智能对话服务。它具备自然语言处理和对话管理能力，支持多领域的应用场景，如在线客服、智能助手和虚拟导购等。
>
> 百度智能云小度：百度智能云开发，提供了多领域的智能对话能力。小度机器人可应用于家庭助理、智能音箱和移动应用等场景，通过语音和文本交互与用户进行智能对话，提供信息查询、音乐播放和日程安排等功能。

- 本项目基于医疗领域数据构建了智能医疗问答系统,目的是为为用户提供准确、高效、优质的医疗问答服务。

环境准备

- python3.6、
- transformers==4.2.0 、
- pytorch==1.7.0

项目整体结构



1. 数据介绍

- 数据存放位置：/Users/**/PycharmProjects/llm/ptune_chatglm/data
- data文件夹里面包含两个文件：medical_train.txt, medical_valid.txt

1.1 数据展示

- medical_train.txt, medical_valid.txt文件的内容均为对话文本

帕金森叠加综合征的辅助治疗有些什么?

综合治疗；康复训练；生活护理指导；低频重复经颅磁刺激治疗

卵巢癌肉瘤的影像学检查有些什么?

超声漏诊；声像图；MR检查；肿物超声；术前超声；CT检查

低T3综合征的并发症是什么?

心力衰竭；甲状腺结节；糖尿病；感染性休克

重复胚胎停止发育的高危因素有些什么?

黄体功能不足；高龄孕妇

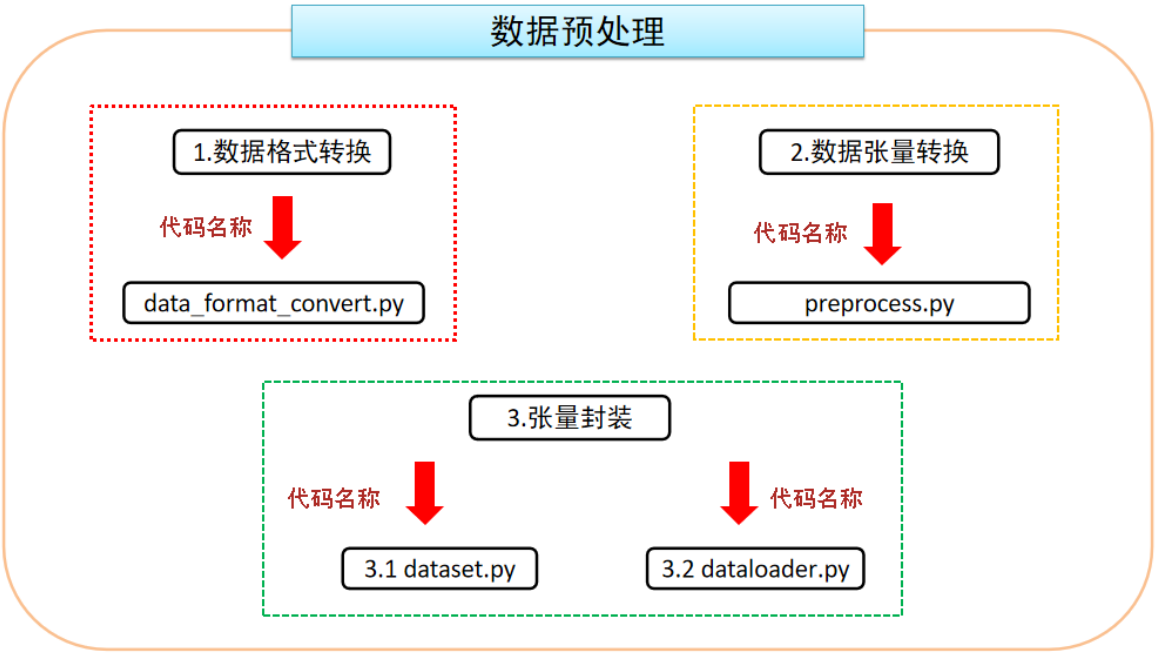
> 原始train文档中中一共包含91487条数据，valid文档中包含1244条数据
>
> 每两行数据为一段对话内容，注意：如果想要实现多轮对话，那么不同对话间实现多条对话语句对即可

2.数据处理

- 目的：将中文文本数据处理成模型能够识别的张量形式。
- 实现过程：
  - 运行preprocess.py，对data/medical_train.txt对话语料进行tokenize，然后进行序列化保存到data/medical_train.txt.pkl。medical_train.pkl中序列化的对象的类型为List[List],记录对话列表中,每个对话包含的token。

```
python preprocess.py
train_path data/medical_train.txt --》save_path data/medical_train.pkl
```

- 数据处理基本流程：



2.1 数据张量转换

- 代码路径：/home/user/ProjectStudy/Gpt2_Chatbot/data_preprocess/preprocess.py

```python
# 导入分词器
from transformers import BertTokenizerFast
# 将数据保存为pkl文件，方便下次读取
import pickle
# 读取数据的进度条展示
from tqdm import tqdm


def preprocess(train_txt_path, train_pkl_path):
    """
    对原始语料进行tokenize，将每段对话处理成如下形式："[CLS]utterance1[SEP]utterance2[SEP]utterance3[SEP]"
    """

    '''初始化tokenizer，使用BertTokenizerFast.创建一个tokenizer对象'''
    tokenizer = BertTokenizerFast('./vocab/vocab.txt',
                                  sep_token="[SEP]",
                                  pad_token="[PAD]",
                                  cls_token="[CLS]")

    sep_id = tokenizer.sep_token_id  # 获取分隔符[SEP]的token ID
    cls_id = tokenizer.cls_token_id  # 获取起始符[CLS]的token ID

    # 读取训练数据集
    with open(train_txt_path, 'rb') as f:
        data = f.read().decode("utf-8")  # 以UTF-8编码读取文件内容

```

```
    # 根据换行符区分不同的对话段落，需要区分Windows和Linux环境下的换行符
    if "\r\n" in data:
        train_data = data.split("\r\n\r\n")
    else:
        train_data = data.split("\n\n")

    print(len(train_data))  # 打印对话段落数量
    # 开始进行tokenize
    # 保存所有的对话数据,每条数据的格式为："[CLS]seq1[SEP]seq2[SEP]seq3[SEP]"
    dialogue_len = []  # 记录所有对话tokenize之后的长度，用于统计中位数与均值
    dialogue_list = []  # 记录所有对话

    for index, dialogue in enumerate(tqdm(train_data)):
        if "\r\n" in data:
            sequences = dialogue.split("\r\n")
        else:
            sequences = dialogue.split("\n")

        input_ids = [cls_id]  # 每个dialogue以[CLS]开头
        for sequence in sequences:
            # 将每个对话句子进行tokenize，并将结果拼接到input_ids列表中
            input_ids += tokenizer.encode(sequence,
add_special_tokens=False)
            input_ids.append(sep_id)  # 每个seq之后添加[SEP]，表示seqs会话结束

        dialogue_len.append(len(input_ids))  # 将对话的tokenize后的长度添加到
对话长度列表中
        dialogue_list.append(input_ids)  # 将tokenize后的对话添加到对话列表中

    print(f'dialogue_len--->{dialogue_len}')  # 打印对话长度列表
    print(f'dialogue_list--->{dialogue_list}')  # 打印

    # 保存pkl文件数据
    with open(train_pkl_path, "wb") as f:
        pickle.dump(dialogue_list, f)
```

2.2 数据张量再次封装

2.2.1 封装DataSet对象

- 代码路径：/home/user/ProjectStudy/Gpt2_Chatbot/data_preprocess/dataset.py

```
from torch.utils.data import Dataset  # 导入Dataset模块，用于定义自定义数据集
import torch  # 导入torch模块，用于处理张量和构建神经网络


class MyDataset(Dataset):
    """
```

```
    自定义数据集类，继承自Dataset类
    """

    def __init__(self, input_list, max_len):
        """
        初始化函数，用于设置数据集的属性
        :param input_list: 输入列表，包含所有对话的tokenize后的输入序列
        :param max_len: 最大序列长度，用于对输入进行截断或填充
        """
        self.input_list = input_list  # 将输入列表赋值给数据集的input_list属性
        self.max_len = max_len  # 将最大序列长度赋值给数据集的max_len属性

    def __len__(self):
        """
        获取数据集的长度
        :return: 数据集的长度
        """
        return len(self.input_list)  # 返回数据集的长度

    def __getitem__(self, index):
        """
        根据给定索引获取数据集中的一个样本
        :param index: 样本的索引
        :return: 样本的输入序列张量
        """
        input_ids = self.input_list[index]  # 获取给定索引处的输入序列
        input_ids = input_ids[:self.max_len]  # 根据最大序列长度对输入进行截断或填充
        input_ids = torch.tensor(input_ids, dtype=torch.long)  # 将输入序列转换为long类型
        return input_ids  # 返回样本的输入序列张量

```

2.2.2 封装DataLoader对象

- 代码路径：/home/user/ProjectStudy/Gpt2_Chatbot/data_preprocess/dataloader.py

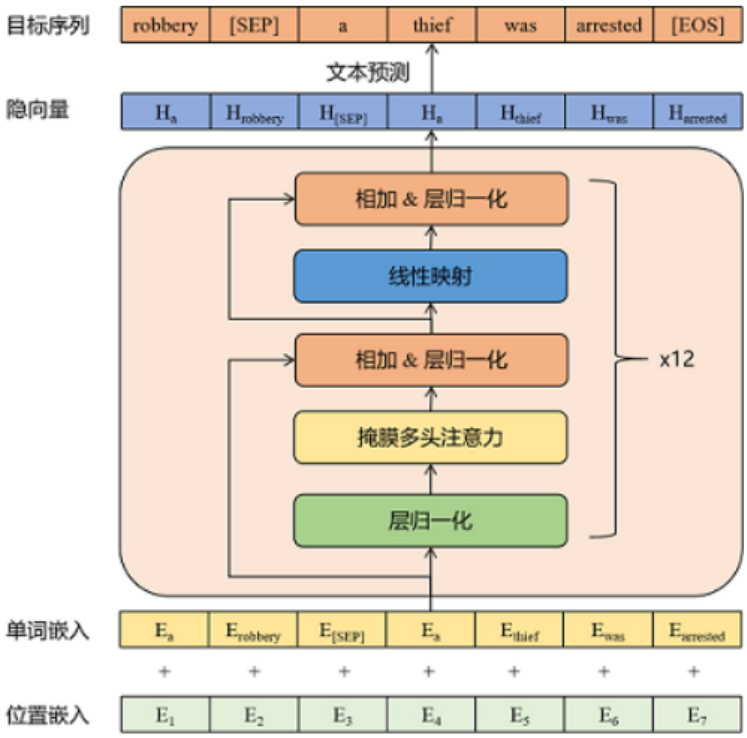
```
# 导入rnn_utils模块，用于处理可变长度序列的填充和排序
import torch.nn.utils.rnn as rnn_utils

# 导入Dataset和DataLoader模块，用于加载和处理数据集
from torch.utils.data import Dataset, DataLoader

import torch  # 导入torch模块，用于处理张量和构建神经网络
import pickle  # 导入pickle模块，用于序列化和反序列化Python对象
from dataset import *  # 导入自定义的数据集类

def load_dataset(train_path, valid_path):
```

```
    # print('进入函数')
    """
    加载训练集和验证集
    :param train_path: 训练数据集路径
    :return: 训练数据集和验证数据集
    """
    with open(train_path, "rb") as f:
        train_input_list = pickle.load(f)  # 从文件中加载输入列表

    with open(valid_path, "rb") as f:
        valid_input_list = pickle.load(f)  # 从文件中加载输入列表
    # 划分训练集与验证集
    # print(len(train_input_list))  # 打印输入列表的长度
    # print(train_input_list[0])
    #
    train_dataset = MyDataset(train_input_list, 300)  # 创建训练数据集对象
    val_dataset = MyDataset(valid_input_list, 300)  # 创建验证数据集对象
    return train_dataset, val_dataset  # 返回训练数据集和验证数据集

def collate_fn(batch):
    """
    自定义的collate_fn函数，用于将数据集中的样本进行批处理
    :param batch: 样本列表
    :return: 经过填充的输入序列张量和标签序列张量
    """
     # 对输入序列进行填充，使其长度一致
    input_ids = rnn_utils.pad_sequence(batch, batch_first=True,
padding_value=0)

    # 对标签序列进行填充，使其长度一致
    labels = rnn_utils.pad_sequence(batch, batch_first=True,
padding_value=-100)

    return input_ids, labels  # 返回经过填充的输入序列张量和标签序列张量

def get_dataloader(train_path, valid_path):
    """
    获取训练数据集和验证数据集的DataLoader对象
    :param train_path: 训练数据集路径
    :return: 训练数据集的DataLoader对象和验证数据集的DataLoader对象
    """
    # 加载训练数据集和验证数据集
    train_dataset, val_dataset = load_dataset(train_path, valid_path)
    print(f'train_dataset-->{len(train_dataset)}')
    print(f'val_dataset-->{len(val_dataset)}')
     # 创建训练数据集的DataLoader对象
    train_dataloader = DataLoader(train_dataset,
                                  batch_size=4,
                                  shuffle=True,
```

```
                                    collate_fn=collate_fn,
                                    drop_last=True)
    # 创建验证数据集的DataLoader对象
    validate_dataloader = DataLoader(val_dataset,
                                     batch_size=4,
                                     shuffle=True,
                                     collate_fn=collate_fn,
                                     drop_last=True)
    # 返回训练数据集的DataLoader对象和验证数据集的DataLoader对象
    return train_dataloader, validate_dataloader


```

3. 模型搭建

3.1 模型架构介绍



- 模型架构解析：
  - 输入层：词嵌入层：WordEmbedding +位置嵌入层：PositionEmbedding
  - 中间层：Transformer的Decoder模块---12层
  - 输出层：LayerNorm层+线性全连接层
- 模型主要参数简介(详见模型的config.json文件):
  - n_embd: 768
  - n_head: 12
  - n_layer: 12
  - n_positions: 1024

- vocab_size: 21128

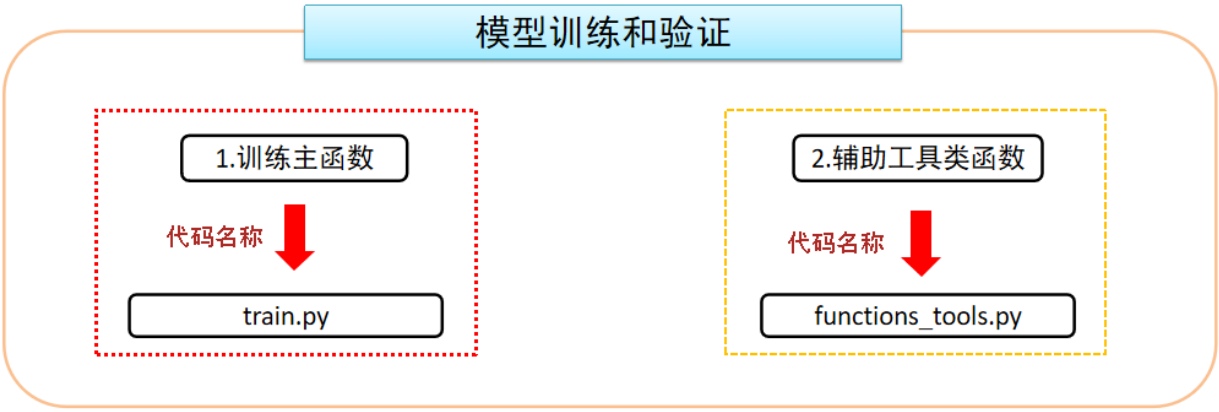
3.2 GPT2模型准备

- 本次项目使用GPT2的预训练模型，因此不需要额外搭建Model类，下面代码是如何直接加载使用GPT2预训练模型
- 代码示例:

```
from transformers import GPT2LMHeadModel, GPT2Config
# 创建模型
if params.pretrained_model:
    # 加载预训练模型
  model = GPT2LMHeadModel.from_pretrained(params.pretrained_model)
else:
    # 初始化模型
  model_config = GPT2Config.from_json_file(params.config_json)
  model = GPT2LMHeadModel(config=model_config)
```

4. 模型训练和验证

- 主要代码



- 代码路径
  - 训练主函数：/home/user/ProjectStudy/Gpt2_Chatbot/data_preprocess/train.py
  - 辅助工具类：/home/user/ProjectStudy/Gpt2_Chatbot/data_preprocess/functions_tools.py

- trian.py代码解析

```
import torch
import os
from datetime import datetime
import transformers
from transformers import GPT2LMHeadModel, GPT2Config
```

```python
from transformers import BertTokenizerFast
from functions_tools import *
from parameter_config import *
from data_preprocess.dataloader import *
from pytorch_tools import EarlyStopping


def train_epoch(model,
                train_dataloader,
                optimizer, scheduler,
                epoch, args):
    model.train()
    device = args.device
    # 对于ignore_index的label token不计算梯度
    ignore_index = args.ignore_index
    epoch_start_time = datetime.now()
    total_loss = 0  # 记录下整个epoch的loss的总和

    # epoch_correct_num:每个epoch中,output预测正确的word的数量
    # epoch_total_num: 每个epoch中,output预测的word的总数量
    epoch_correct_num, epoch_total_num = 0, 0

    for batch_idx, (input_ids, labels) in enumerate(train_dataloader):
        input_ids = input_ids.to(device)
        labels = labels.to(device)
        outputs = model.forward(input_ids, labels=labels)

        logits = outputs.logits
        loss = outputs.loss
        loss = loss.mean()



        # 统计该batch的预测token的正确数与总数
        batch_correct_num, batch_total_num = calculate_acc(logits,
labels, ignore_index=ignore_index)

        # 统计该epoch的预测token的正确数与总数
        epoch_correct_num += batch_correct_num
        epoch_total_num += batch_total_num
        # 计算该batch的accuracy
        batch_acc = batch_correct_num / batch_total_num

        total_loss += loss.item()
        if args.gradient_accumulation_steps > 1:
            loss = loss / args.gradient_accumulation_steps

        loss.backward()
        # 梯度裁剪 # 避免梯度爆炸的方式。
```

```
        torch.nn.utils.clip_grad_norm_(model.parameters(), args.max_grad_norm)

        # 进行一定step的梯度累计之后，更新参数
        if (batch_idx + 1) % args.gradient_accumulation_steps == 0:
            # 更新参数
            optimizer.step()
            # 更新学习率
            scheduler.step()
            # 清空梯度信息
            optimizer.zero_grad()

        if (batch_idx + 1) % args.loss_step == 0:
            print(
                "batch {} of epoch {}, loss {}, batch_acc {}, lr {}".format(
                    batch_idx + 1, epoch + 1, loss.item() * args.gradient_accumulation_steps, batch_acc, scheduler.get_lr()))

        del input_ids, outputs


    # 记录当前epoch的平均loss与accuracy
    epoch_mean_loss = total_loss / len(train_dataloader)
    epoch_mean_acc = epoch_correct_num / epoch_total_num
    print(
        "epoch {}: loss {}, predict_acc {}".format(epoch + 1, epoch_mean_loss, epoch_mean_acc))

    # save model
    if epoch % 10 == 0 or epoch == args.epochs:
        print('saving model for epoch {}'.format(epoch + 1))
        model_path = os.path.join(args.save_model_path, 'bj_epoch{}'.format(epoch + 1))
        if not os.path.exists(model_path):
            os.mkdir(model_path)
        model.save_pretrained(model_path)
        print('epoch {} finished'.format(epoch + 1))
        epoch_finish_time = datetime.now()
        print('time for one epoch: {}'.format(epoch_finish_time - epoch_start_time))

    return epoch_mean_loss


def validate_epoch(model, validate_dataloader, epoch, args):
    print("start validating")
    model.eval()
    device = args.device
```

```
97      ignore_index = args.ignore_index
98      epoch_start_time = datetime.now()
99      total_loss = 0
100     # 捕获cuda out of memory exception
101     with torch.no_grad():
102         for batch_idx, (input_ids, labels) in
enumerate(validate_dataloader):
103             input_ids = input_ids.to(device)
104             labels = labels.to(device)
105             outputs = model.forward(input_ids, labels=labels)
106             logits = outputs.logits
107             loss = outputs.loss
108             loss = loss.mean()
109
110             total_loss += loss.item()
111             del input_ids, outputs
112
113         # 记录当前epoch的平均loss
114         epoch_mean_loss = total_loss / len(validate_dataloader)
115         print(
116             "validate epoch {}: loss {}".format(epoch+1,
epoch_mean_loss))
117         epoch_finish_time = datetime.now()
118         print('time for validating one epoch:
{}'.format(epoch_finish_time - epoch_start_time))
119         return epoch_mean_loss
120
121
122 def train(model,  train_dataloader, validate_dataloader, args):
123
124     # early_stopping = EarlyStopping(patience=0, verbose=True)
125     t_total = len(train_dataloader) // args.gradient_accumulation_steps *
args.epochs
126     optimizer = transformers.AdamW(model.parameters(), lr=args.lr,
eps=args.eps)
127     '''
128     这里对于模型的参数，分别就行权重参数的衰减优化：防止过拟合，以及学习率预热处理优
化：
129     在初始阶段将学习率从较小的值逐步增加到设定的初始值，然后按照设定的学习率调整策略进
行训练。
130     学习率预热的目的是让模型在初始阶段更快地适应数据，避免训练过程中因为学习率过大导致
的梯度爆炸等问题，
131     从而提高模型的训练效果和泛化性能。
132     optimizer：  优化器
133     num_warmup_steps：初始预热步数
134     num_training_steps：整个训练过程的总步数
135     '''
136     scheduler = transformers.get_linear_schedule_with_warmup
137     (
```

```
        optimizer,
        num_warmup_steps=args.warmup_steps,
        num_training_steps=t_total
    )

    print('starting training')

    # 用于记录每个epoch训练和验证的loss
    train_losses, validate_losses = [], []
    # 记录验证集的最小loss
    best_val_loss = 10000
    # 开始训练
    for epoch in range(args.epochs):
        # ========== train ========== #
        train_loss = train_epoch(
            model=model, train_dataloader=train_dataloader,
            optimizer=optimizer, scheduler=scheduler,
            epoch=epoch, args=args)
        train_losses.append(train_loss)

        # ========== validate ========== #
        validate_loss = validate_epoch(
            model=model, validate_dataloader=validate_dataloader,
            epoch=epoch, args=args)
        validate_losses.append(validate_loss)

        # 保存当前困惑度最低的模型，困惑度低，模型的生成效果不一定会越好
        if validate_loss < best_val_loss:
            best_val_loss = validate_loss
            print('saving current best model for epoch {}'.format(epoch +
1))
            model_path = os.path.join(args.save_model_path,
                                      'min_ppl_model_bj'.format(epoch +
1))
            if not os.path.exists(model_path):
                os.mkdir(model_path)
            model.save_pretrained(model_path)


def main():
    # 初始化配置参数
    params = ParameterConfig()

    # 设置使用哪些显卡进行训练:默认为0
    os.environ["CUDA_VISIBLE_DEVICES"] = '0'


    # 初始化tokenizer
    tokenizer = BertTokenizerFast(params.vocab_path,
```

```
                                    sep_token="[SEP]",
                                    pad_token="[PAD]",
                                    cls_token="[CLS]")


    # 创建模型的输出目录
    if not os.path.exists(params.save_model_path):
        os.mkdir(params.save_model_path)

    # 创建模型
    if params.pretrained_model:
        # 加载预训练模型
        model = GPT2LMHeadModel.from_pretrained(params.pretrained_model)
    else:
        # 初始化模型
        model_config = GPT2Config.from_json_file(params.config_json)
        model = GPT2LMHeadModel(config=model_config)
    model = model.to(params.device)
    assert model.config.vocab_size == tokenizer.vocab_size


    # 计算模型参数数量
    num_parameters = 0
    parameters = model.parameters()
    for parameter in parameters:
        num_parameters += parameter.numel()
    print(f'模型参数总量---》{num_parameters}')

    # 加载训练集和验证集
    # ========= Loading Dataset ========= #
    train_dataloader, validate_dataloader =
get_dataloader(params.train_path)
    train(model, train_dataloader, validate_dataloader, params)

if __name__ == '__main__':
    main()

```

- functions_tools.py代码解析

```
import torch
import torch.nn.functional as F

def calculate_acc(logit, labels, ignore_index=-100):
    logit = logit[:, :-1, :].contiguous().view(-1, logit.size(-1))
    labels = labels[:, 1:].contiguous().view(-1)
    _, logit = logit.max(dim=-1)  # 对于每条数据，返回最大的index
    '''
```

```
    在 PyTorch 中，labels.ne(ignore_index) 表示将标签张量 labels 中的值不等于 ignore_index 的位置标记为 True，等于 ignore_index 的位置标记为 False。
    这个操作通常被用于计算交叉熵损失，以过滤掉 ignore_index 对损失的贡献
    '''
    # 进行非运算，返回一个tensor，若labels的第i个位置为pad_id，则置为0，否则为1
    non_pad_mask = labels.ne(ignore_index)
    '''
    在 PyTorch 中，
    logit.eq(labels) 表示将模型的预测输出值 logit 中等于标签张量 labels 的位置标记为 True，不等于标签张量 labels 的位置标记为 False。这个操作通常被用于计算交叉熵损失，以标记出预测输出值和标签值相等的位置。
    masked_select(non_pad_mask) 表示将张量中非填充标记的位置选出来。这个操作通常被用于计算损失时，过滤掉填充标记对损失的影响。
    '''
    n_correct = logit.eq(labels).masked_select(non_pad_mask).sum().item()
    n_word = non_pad_mask.sum().item()
    return n_correct, n_word
```

5. 模型预测（人机交互）

- 运行interact.py，使用训练好的模型，进行人机交互，输入Ctrl+Z结束对话之后，聊天记录将保存到sample目录下的sample.txt文件中。

```
import os
from datetime import datetime
from transformers import GPT2LMHeadModel
from transformers import BertTokenizerFast
import torch.nn.functional as F
from parameter_config import *

PAD = '[PAD]'
pad_id = 0


def top_k_top_p_filtering(logits, top_k=0, top_p=0.0, filter_value=-float('Inf')):
    """使用top-k和/或nucleus（top-p）筛选来过滤logits的分布
        参数：
            logits: logits的分布，形状为（词汇大小）
            top_k > 0: 保留概率最高的top k个标记（top-k筛选）。
            top_p > 0.0: 保留累积概率大于等于top_p的top标记（nucleus筛选）。
    """
    assert logits.dim() == 1  # batch size 1 for now - could be updated for more but the code would be less clear
    top_k = min(top_k, logits.size(-1))  # Safety check
    print(f'top_k---->{top_k}')
    if top_k > 0:
```

```
        # Remove all tokens with a probability less than the last token of the top-k
        # torch.topk()返回最后一维最大的top_k个元素，返回值为二维(values,indices)
        # ...表示其他维度由计算机自行推断
        print(f'torch.topk(logits, top_k)-->{torch.topk(logits, top_k)}')
        indices_to_remove = logits < torch.topk(logits, top_k)[0][..., -1, None]
        logits[indices_to_remove] = filter_value  # 对于topk之外的其他元素的logits值设为负无穷

    if top_p > 0.0:
        sorted_logits, sorted_indices = torch.sort(logits, descending=True)  # 对logits进行递减排序
        print(f'sorted_logits-->{sorted_logits}')
        print(f'sorted_indices-->{sorted_indices}')
        cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-1), dim=-1)

        # Remove tokens with cumulative probability above the threshold
        sorted_indices_to_remove = cumulative_probs > top_p
        # Shift the indices to the right to keep also the first token above the threshold
        sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[..., :-1].clone()
        sorted_indices_to_remove[..., 0] = 0

        indices_to_remove = sorted_indices[sorted_indices_to_remove]
        logits[indices_to_remove] = filter_value
    return logits


def main():
    pconf = ParameterConfig()
    # 当用户使用GPU,并且GPU可用时
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    print('using device:{}'.format(device))
    os.environ["CUDA_VISIBLE_DEVICES"] = '0'
    tokenizer = BertTokenizerFast(vocab_file=pconf.vocab_path,
                                  sep_token="[SEP]",
                                  pad_token="[PAD]",
                                  cls_token="[CLS]")
    model = GPT2LMHeadModel.from_pretrained('./save_model/epoch25')
    model = model.to(device)
    model.eval()
    # 保存聊天记录的文件路径
    if pconf.save_samples_path:
        if not os.path.exists(pconf.save_samples_path):
            os.makedirs(pconf.save_samples_path)
```

```
        samples_file = open(pconf.save_samples_path + '/samples.txt', 'a', encoding='utf8')
        samples_file.write("聊天记录{}:\n".format(datetime.now()))
    # 存储聊天记录，每个utterance以token的id的形式进行存储
    history = []
    print('开始和chatbot聊天，输入CTRL + Z以退出')

    while True:
        try:
            text = input("user:")
            # text = "你好"
            if pconf.save_samples_path:
                samples_file.write("user:{}\n".format(text))
            text_ids = tokenizer.encode(text, add_special_tokens=False)
            print(f'text_ids-->{text_ids}')
            print('*' * 80)
            history.append(text_ids)
            input_ids = [tokenizer.cls_token_id]  # 每个input以[CLS]为开头
            print(f'history---.{history}')
            print(f'input_ids---.{input_ids}')
            print('*' * 80)
            print(f'history[-pconf.max_history_len:]-->{history[-pconf.max_history_len:]}')
            for history_id, history_utr in enumerate(history[-pconf.max_history_len:]):
                input_ids.extend(history_utr)
                print(input_ids)
                input_ids.append(tokenizer.sep_token_id)
                print(input_ids)
            print('*'*80)
            print(f'new_inut--->{input_ids}')
            input_ids = torch.tensor(input_ids).long().to(device)
            input_ids = input_ids.unsqueeze(0)
            print(f'las--inputs_ids{input_ids}')
            response = []  # 根据context，生成的response
            # 最多生成max_len个token
            for _ in range(pconf.max_len):
                outputs = model(input_ids=input_ids)
                logits = outputs.logits
                print(f'logits--->{logits}')
                print(f'logits--->{logits.shape}')
                print('*'*80)
                next_token_logits = logits[0, -1, :]
                # 对于已生成的结果generated中的每个token添加一个重复惩罚项，降低其生成概率
                print(f'next_token_logits-->{next_token_logits}')
                for id in set(response):
                    print(f'id--->{id}')
                    next_token_logits[id] /= pconf.repetition_penalty
```

```
                # 对于[UNK]的概率设为无穷小，也就是说模型的预测结果不可能是[UNK]这个token
 next_token_logits[tokenizer.convert_tokens_to_ids('[UNK]')] = -float('Inf')
                filtered_logits = top_k_top_p_filtering(next_token_logits, top_k=pconf.topk, top_p=pconf.topp)
                print(f'filtered_logits-->{filtered_logits}')
                # torch.multinomial表示从候选集合中无放回地进行抽取num_samples个元素，权重越高，抽到的几率越高，返回元素的下标
                next_token = torch.multinomial(F.softmax(filtered_logits, dim=-1), num_samples=1)
                print(f'next_token-->{next_token}')
                if next_token == tokenizer.sep_token_id:  # 遇到[SEP]则表明response生成结束
                    break
                response.append(next_token.item())
                input_ids = torch.cat((input_ids, next_token.unsqueeze(0)), dim=1)
                # his_text = tokenizer.convert_ids_to_tokens(curr_input_tensor.tolist())
                # print("his_text:{}".format(his_text))
            print(f'response-->{response}')
            history.append(response)
            text = tokenizer.convert_ids_to_tokens(response)
            print("chatbot:" + "".join(text))
            if pconf.save_samples_path:
                samples_file.write("chatbot:{}\n".format("".join(text)))
        except KeyboardInterrupt:
            if pconf.save_samples_path:
                samples_file.close()
            break


if __name__ == '__main__':
    main()
```