2021-2022 学年

第 2 学期

《深度学习》

课程实验报告

姓名：林超超 学号：1120191287

实验 5：基于结构化数据的序列到序列生成任务

一、 实验目的

本次实验通过使用 PyTorch 完成基于结构化数据的 Seq2Seq 生成任务，掌握文本生成任务数据获取、预处理、语料对齐、分词等内容，理解并能够实现编解码器架构和注意力机制。

二、 实验内容

1. 学习编码器-解码器架构的基本原理，了解编解码器架构适用的任务类型；

2. 学习和理解注意力机制的原理，了解常用注意力机制的实现方式；

3. 了解序列到序列任务的处理流程，使用 PyTorch 编码完成规定的 Seq2Seq 任务——基于 E2E 数据集的表格到文本生成任务或基于 WikiSQL 数据集的自然语言到 SQL 语句生成任务。

三、 实验原理

1. Encoder-Decoder 架构

编解码器（Encoder-Decoder）架构适用于输入和输出均为序列的任务，其由编码器和解码器两部分构成。编码器对输入序列编码，得到一个或一组表示向量 C，该向量涵盖了输入序列的语义信息；解码器用表示向量 C 初始化解码出对应的目标序列。

2. Attention 机制

注意力机制（Attention）是一种广泛应用于 Encoder-Decoder 架构的方法。Attention 使得模型在解码的不同时刻可以有侧重地关注输入序列的不同部分，从而提升 Seq2Seq 任务的表现。

四、 实验过程

4.1 实验环境

笔记本使用的操作系统是 Windows10，CPU 是 i9-8950HK，显卡是 GTX 1080，内存 32GB。另外使用第三方库和内置模块有：深度学习库 torch、进度条库 tqdm、数据计算库 numpy、绘图库 matplotlib 和 seaborn、操作系统交互模块 os/sys、正则表达式模块 re、数据结构和数据分析工具 pandas、常用数据结构模块 collections。

4.2 代码结构

本实验涉及多个代码文件，文件目录结构及用途如下：
e2e_dataset 文件夹：保存训练集、验证集、测试集 csv 文件。
dataset.py：数据集读取及预处理文件，用于实验数据的读取和预处理。
config.py：配置文件，用于设置实验参数。
model.py：模型文件，用于设计模型结构。
metric.py：评估文件，用于模型的 BLEU 分数评估。
main.py：主函数文件，用于完成模型训练、验证、测试等主流程。
visualize.py：可视化文件，用于对模型注意力效果进行可视化。

4.3 数据集加载及预处理

该部分在 dataset.py 中完成。

（1）数据集结构分析

数据集 e2e_dataset 文件夹下包含三个 csv 文件 trainset.csv、devset.csv、testset.txt 分别存储训练集、验证集和测试集。 本次数据集中，训练集共 4862 条结构化文本 mr 对应共 42061 条参考文本 ref；验证集共有 547 条结构化文本 mr 对应共 4672 条参考文本 ref；测试集中共有 630 条结构化文本 MR。训练集、验证集、测试集的结构化文本之比约为 8：1：1。

表 1 数据集字段统计

数据集	mr/MR	ref
trainset.csv	4862	42061
devset.csv	547	4672
trainset.csv	630	------

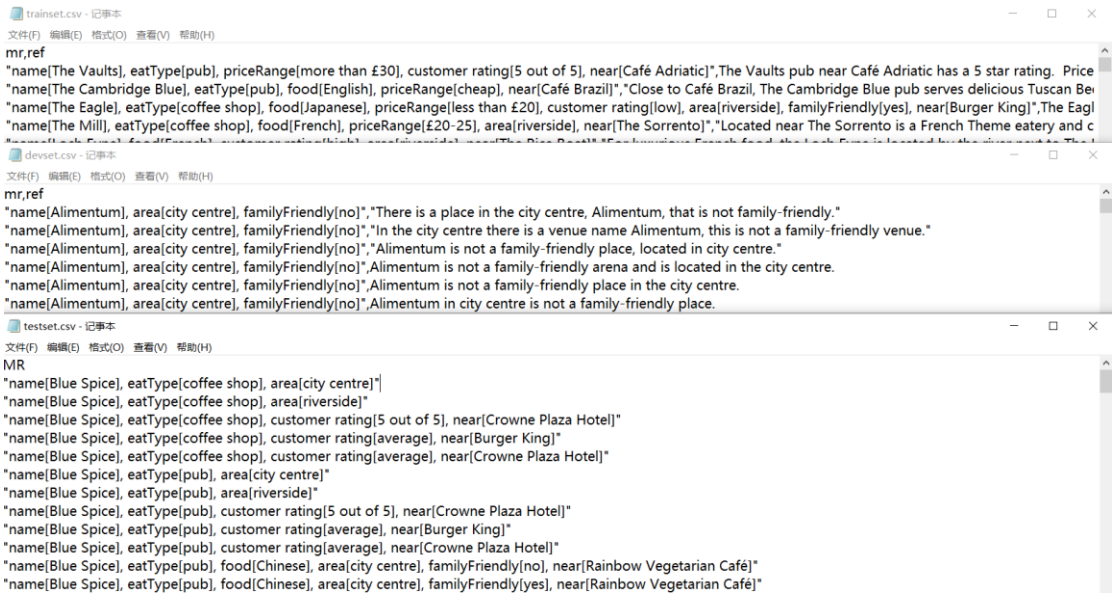

图 1 数据集结构

训练集和验证集中共有 2 个字段，分别为 mr 和 ref，mr 为结构化文本，描述了餐馆的信息，其中包括 2 个开放属性名字 name、地点 near 以及 6 个非开放的属性是否家庭友好 familyFriendly、餐馆类型 eatType、餐馆食物类型 food、priceRange 价格区间、区域 area、客流量 customer rating 总共 8 个属性；ref 是结构化文本对应的参考文本，即将结构化文本中的信息组合成具有实际含义的完整句子。测试集中仅有 1 个字段，即结构化文本 MR（测试集中该字段为大写），要求使用模型推理出相应的参考文本 ref。

（2）词到编号的映射

定义 Tokenizer 类用于构建完成词到编号的映射关系的编码器，输入是由词到编号映射的词典 token_dict，为方便计算，将字典的键值对相互调换，得到编号到词的映射字典 token_dict_rev 并统计字典大小 vocab_size。分词器类有四个功能函数实现字符串的编码和解码，分别是 id_to_token、token_to_id、encode、decode。

```
# 构建词到编号的映射关系
class Tokenizer:
    def __init__(self, token_dict):
        # 词->编号的映射
        self.token_dict = token_dict
        # 编号->词的映射
        self.token_dict_rev = {value: key for key, value in self.token_
dict.items()}
        # 词汇表大小
```

```
        self.vocab_size = len(self.token_dict)
```

1. 编码

编码即将给定字符串映射为数字编号，字符串由多个词组成，对于单个词token，利用构建的词到编号的映射字典，使用 get 方法在字典中查找得到单个词编号。如果词未出现在字典中，则返回低频标记[UNK]的编号。

```
    # 给定一个编号，查找词汇表中对应的词
    def id_to_token(self, token_id):
        return self.token_dict_rev[token_id]
```

对于字符串 tokens，遍历每个词，调用 token_to_id 函数得到对应的编号列表。另外，在编号序列头尾分别加上开始和结束的特殊词[BOS]和[EOS]。

```
    # 给定一个字符串 s，在头尾分别加上标记开始和结束的特殊字符
    # 并将它转成对应的编号序列
    def encode(self, tokens):
        # 加上开始标记
        token_ids = [self.token_to_id('[BOS]'), ]
        # 加入字符串编号序列
        for token in tokens:
            token_ids.append(self.token_to_id(token))
        # 加上结束标记
        token_ids.append(self.token_to_id('[EOS]'))
        return token_ids
```

2. 编码

解码即将给定数字编号序列映射为字符串，对于单个数字编号 token_id，利用构建的编号到词的映射字典，使用 get 方法在字典中查找得到单个编号对应的词。由于编号是人为给定的，因此不存在不在映射字典的情况。

```
    # 给定一个词，查找它在词汇表中的编号，未找到则返回低频词[UNK]的编号
    def token_to_id(self, token):
        return self.token_dict.get(token, self.token_dict['[UNK]'])
```

对于编号序列 token_ids，遍历每个编号，调用 id_to_token 函数得到对应的字符列表。另外，略过编号序列起始标记字符[BOS]和[EOS]]。最后，使用 join 方法拼接字符串得到解码的文本结果。

```
    # 给定一个编号序列，将它解码成字符串
    def decode(self, token_ids):
        # 起止标记字符特殊处理
        spec_tokens = {'[BOS]', '[EOS]', '[PAD]'}
        # 保存解码出的字符的 list
        tokens = []
        for token_id in token_ids:
            token = self.id_to_token(token_id)
            if token in spec_tokens:
                continue
            tokens.append(token)
        # 拼接字符串
```

```
        return ' '.join(tokens)
```

（3）数据集读取

自定义数据集加载类 class E2EDataSet(Dataset)，继承自 torch.utils.data 中的 Dataset 类。自定义数据集类需完成__init__、__getitem__、__len__三个函数，另外定义了四个函数用于减少上面几个函数的代码量，使结构更加清晰：分别是属性词典构造函数 create_field、文本预处理函数 preprocess、文本词典构造函数 create_voc、序列填充和截断函数 sequence_padding。在该类中，对结构化文本和参考文本进行了去词化（Delexicalization）等预处理，并构建了属性词典和文本词典用于对结构化文本和参考文本进行编码。

1. 初始化函数__init__

__init__在实例化自定义类时调用，作用是初始化类并进行参数赋值。这里设定函数有五个输入：数据集路径 path、数据集类型 mode、属性词典 field_tokenizer、文本词典 tokenizer、最大结构化文本长度 max_src_len、最大目标参考文本长度 max_tgt_len。

```
class E2EDataset(Dataset):
    def __init__(self, path, mode='train', field_tokenizer=None,
tokenizer=None, max_src_len=80, max_tgt_len=80):
        self.mode = mode  # 数据集类型
        self.max_src_len = max_src_len  # 最大结构化文本长度
        self.max_tgt_len = max_tgt_len  # 最大目标参考文本长度
```

首先，调用 pandas 库的 read_csv 函数读取给定路径的数据文件。如果 mode 参数为 train 或者 dev，训练集和验证集包含参考文本（ref 字段），因此直接读取并且转换为列表，这里定义了 str2dict 函数用于将字符串格式的结构化文本（mr 字段）处理为字典格式，主要是为了方便后续直接读取进行属性词典的构建和预处理等操作；如果 mode 参数为 test，测试集不包含参考文本（ref 字段），则只从 csv 文件中读取结构化文本字段 MR（测试集中该字段名为大写），参考文本 ref 则利用空字符串进行占位，可以直接进行返回无需改变其他函数定义；当然如果 mode 参数不在给定的 train、dev、test 三种模式下，则抛出错误。

```
        df = pd.read_csv(path)  # 读取数据
        # print(len(set(df['mr'].values.tolist())))
        if mode == 'train' or mode == 'dev':
            # mr 字段 处理为字典形式
            self.mr = str2dict(df['mr'].values.tolist())
            self.ref = df['ref'].values.tolist()  # ref 字段
        elif mode == 'test':
            # MR 字段 处理为字典形式
            self.mr = str2dict(df['MR'].values.tolist())
            self.ref = ['' for _ in range(len(self.mr))]
        else:
            raise ValueError("mode must in [train, dev, test]")
```

上面所述的 str2dict 函数定义如下，该函数的功能是将字符串格式的结构化文本处理为字典格式，输入是字符串列表 str_list，即包含所有结构化文本的列表，

首先，利用 map 函数和 lambda 匿名函数将结构化文本按逗号和空格进行分隔，得到多个键值对，格式为 key[value]，再遍历所有键值对利用[符号进行分隔得到键和值并构建结构化文本的字典，并添加到字典列表，最后将每一条结构化文本转换为字典的键值对映射格式。

```
# 将字符串转换为结构化的字典
def str2dict(str_list):
    dict_list = []
    # 利用分隔符逗号将属性和值对分开
    map_os = list(map(lambda x: x.split(', '), str_list))
    # [['A[a]', 'B[b]', ...], ['A[a]', 'B[b]', ...], ...]
    for map_o in map_os:  # ['A[a]', 'B[b]', ...]
        _dict = {}
        for item in map_o:  # 获取键值对  'A[a]'
            key = item.split('[')[0]  # 键 A
            value = item.split('[')[1].replace(']', '')  # 值 a
            dict[key] = value  # 构建字典
        dict_list.append(_dict)
    return dict_list
```

回到初始化函数__init__，根据数据集的类型，如果 mode 参数为 train，则利用训练集构建字典，调用 create_field 构建属性词典，再调用 preprocess 函数对文本数据进行分词、去词化等预处理操作，最后调用 create_voc 函数构建文本词典；如果 mode 参数为 dev 或 test，则先确认是否有给定词典，如果没有则会抛出错误，确认输入的词典存在后，赋值属性词典和文本词典，并得到属性词典长度，然后调用 preprocess 函数对文本数据进行预处理。

```
        if mode == 'train':  # 训练模式下构建词典
            self.create_field()  # 构建属性词典
            self.preprocess()  # 数据预处理、去词化
            self.create_voc()  # 构建文本词典
        else:  # 验证和测试模式下读取词典
            if field_tokenizer is None or tokenizer is None:
                raise ValueError("field tokenizer and tokenizer must
                                 not be None")
            self.field_tokenizer = field_tokenizer
            self.key_num = len(self.field_tokenizer)
            self.tokenizer = tokenizer
            self.preprocess()
```

2. 属性词典构造函数 create_field

create_field 函数用于构建属性词典，前面已经将 mr 字段进行了预处理，得到了字典形式的格式化文本，这里直接利用 map 函数和 lambda 匿名函数得到所有结构化文本的键，然后调用 collections 库中的 Counter 统计所有训练数据的键的出现次数，并根据统计的词频，对属性名表进行排序，保留属性名列表。利用 zip 函数得到属性名和编号的组合，构建属性名到编号的映射字典，从 0 开始为

每一个属性名编号，记录总共出现的属性名的数量 key_num。

```
def create_field(self):
    # mr 字段 值
    mr_key = list(map(lambda x: list(x.keys()), self.mr))
    # 统计词频
    counter = Counter()
    for line in mr_key:
        counter.update(line)
    # 按词频排序
    _tokens = [(token, count) for token, count in counter.items()]
    _tokens = sorted(_tokens, key=lambda x: -x[1])
    # 去掉词频，只保留词列表
    _tokens = [token for token, count in _tokens]
    # 创建词典 token->id 映射关系
    self.field_tokenizer = dict(zip(_tokens, range(len(_tokens))))
    self.key_num = len(self.field_tokenizer)
```

3. 文本预处理函数 preprocess

preprocess 函数是对文本的预处理，对文本进行了分词、去词化等操作。raw_data_x 用于存储结构化文本数据，也就是特征；raw_data_y 用于存储参考文本数据，也就是目标值；lexicalizations 用于存储去词化原词，用于编解码后的句子；multi_data_y 用于存储结构化文本对应的多个参考文本，其格式为字典，键为结构化文本，值为多个参考文本的列表。遍历所有的数据，这里将结构化文本转换为长度为属性数量（8 维）的属性列表，列表不同位置对应不同属性，使用填充词的编码 PAD_ID 进行初始化（0），lex 列表用于存储当前文本的去词化原词，长度为 2，分别对应餐馆名字 name 和地点 near，初始化为两个空字符串。

```
def preprocess(self):
    self.raw_data_x = []
    self.raw_data_y = []
    self.lexicalizations = []
    self.muti_data_y = {}
    for index in range(len(self.ref)):
        mr_data = [PAD_ID] * self.key_num
        lex = ['', '']
```

首先，先对结构化文本转换为属性列表并去词化处理，遍历结构化文本的字典，得到结构化文本的属性名和属性值，利用属性词典 field_tokenizer 得到属性名的编号，在属性列表中编号对应的位置记录属性值，这里对餐馆名 name 属性和地点 near 属性进行去词化（Delexicalization）处理，即记录其属性值为 NAME_TOKEN 或 NEAR_TOKEN，减少模型训练时各种各样的餐厅名字和地点名的干扰，其他属性则按具体的属性值进行记录。最终得到描述结构化文本的定长列表，对于数据中没有出现的属性，其相应位置为 0；对于数据中出现的属性，其相应位置为对应的属性值。

```
        # 将 mr 处理成列表并进行 Delexicalization
        for item in self.mr[index].items():
            key = item[0]
            value = item[1]
            key_idx = self.field_tokenizer[key]
            # Delexicalization
            if key == 'name':
                mr_data[key_idx] = NAME_TOKEN
                lex[0] = value
            elif key == 'near':
                mr_data[key_idx] = NEAR_TOKEN
                lex[1] = value
            else:
                mr_data[key_idx] = value
```

下一步将参考文本也处理为列表形式，如果句子为空说明是测试集，那么不做处理。否则，根据去词化的原词（为空则不处理），使用 replace 函数先对参考句子进行去词化，将餐馆名和地点名分别替换为 NAME_TOKEN 和 NEAR_TOKEN，利用 re 模块的 split 函数去除数据中的标点，并按空格进行词的划分，得到参考文本的词列表。

```
        # 将 ref 处理成列表
        ref_data = self.ref[index]
        if ref_data == '':  # 句子为空说明是测试集没有 ref
            ref_data = ['']
        else:
            # Delexicalize
            if lex[0]:
                ref_data = ref_data.replace(lex[0], NAME_TOKEN)
            if lex[1]:
                ref_data = ref_data.replace(lex[1], NEAR_TOKEN)
            ref_data = list(map(lambda x: re.split(r"([.,!?\"':;)(])
", x)[0], ref_data.split()))
```

将处理后的单条结构化文本数据 mr_data、参考文本数据 ref_data 以及去词化原词追加到相应的列表中，对于多个参考文本，则将其添加到字典中。

```
        # 追加列表
        self.raw_data_x.append(mr_data)
        self.raw_data_y.append(ref_data)
        self.lexicalizations.append(lex)
        # 多参考文本
        mr_data_str = ''.join(mr_data)
        if mr_data_str in self.muti_data_y.keys():
            self.muti_data_y[mr_data_str].append(self.ref[index])
        else:
            self.muti_data_y[mr_data_str] = [self.ref[index]]
```

4. 文本词典构造函数 create_voc

create_voc 函数用于构造文本词典，由于结构化文本中的属性值大多会出现在参考文本中，因此为了更好地反映结构化文本属性值和参考文本的关系，这里只构建一个共享的词典，也就是结构化文本属性值和参考文本使用同一本词典进行编解码。

首先使用 collections 库的 Counter 对属性值的文本和参考文本的词进行词频统计，对词表进行排序，保留词列表，将特殊词加入到词表，特殊词 [PAD]、[BOS]、 [EOS]、[UNK]，分别表示填充标记、句首标记、结束标记、未知标记。得到词表后，创建词到编号的映射字典，从 0 开始给每一个词编号。使用字典建立分词器类 Tokenizer。

```
    def create_voc(self):
        # 统计词频
        counter = Counter()
        for line in self.raw_data_x:
            counter.update(line)
        for line in self.raw_data_y:
            counter.update(line)
        # 按词频排序
        _tokens = [(token, count) for token, count in counter.items()]
        _tokens = sorted(_tokens, key=lambda x: -x[1])
        # 去掉词频，只保留词列表
        _tokens = ['[PAD]', '[BOS]', '[EOS]', '[UNK]'] + [token for toke
n, count in _tokens]
        # 创建词典 token->id 映射关系
        token_id_dict = dict(zip(_tokens, range(len(_tokens))))
        # 使用新词典重新建立分词器
        self.tokenizer = Tokenizer(token_id_dict)
```

5. 序列填充和截断函数 sequence_padding

接下来定义 sequence_padding 函数对不足长度的句子进行填充以及对超过长度的句子进行截断，输入是句子 data、最大长度 max_len、填充词 padding 默认为 None，若不输入则采用填充词[PAD]的 ID 进行填充，即 0。首先计算需要填充的长度，若其大于 0 则说明需要填充，在句子后面添加所需要填充长度的 padding；若小于 0 则说明超过长度，此时进行截断。

```
# 将给定数据填充到相同长度
    def sequence_padding(self, data, max_len, padding=None):
        # 计算填充数据
        if padding is None:
            padding = self.tokenizer.token_to_id('[PAD]')
        self.padding = padding
        # 开始填充
        padding_length = max_len - len(data)
        # 不足就进行填充
```

```
        if padding_length > 0:
            outputs = data + [padding] * padding_length
        # 超过就进行截断
        else:
            outputs = data[:max_len]
        return outputs
```

6. 数据获取函数__getitem__

_getitem__是重写继承的父类 Dataset 中的函数，在读取数据时调用，输入是序号 index，每次返回一组数据，通过序号 index 得到结构化文本和目标参考文本，使用词典进行编码并进行序列填充得到 x 和 y，如果是训练模式，则直接返回 x 和 y，如果是验证或测试模式，则从 lexicalizations 列表中读取当前文本中被去词化的词 lex 以及从 muti_data_y 字典中读取当前结构化文本的多参考文本列表 muti_y，返回四个参数。

```
    def __getitem__(self, index):
        x = np.array(self.sequence_padding(self.tokenizer.encode(self.
raw_data_x[index]), self.max_src_len))
        y = np.array(self.sequence_padding(self.tokenizer.encode(self.
raw_data_y[index]), self.max_tgt_len))
        if self.mode == 'train':
            return x, y
        else:
            lex = self.lexicalizations[index]
            muti_y = self.muti_data_y[''.join(self.raw_data_x[index])]
            return x, y, lex, muti_y
```

7. 数据长度获取函数__len__

__len__函数在对自定义类使用 len 函数时调用，这里返回数据集的大小。

```
    # 使用 len 函数时调用 返回数据集大小
    def __len__(self):
        return len(self.ref)
```

4.4 设置参数

该部分在 config.py 中完成。

为方便调用，定义 Config 类用于设置参数，具体参数名及其含义和默认值如下表。

表 2 参数名及含义

参数类型	参数名	含义	默认值
数据集路径	train_data	训练集路径	‘./e2e_dataset/trainset.csv’
	dev_data	验证集路径	‘./e2e_dataset/devset.csv’
	test_data	测试集路径	‘./e2e_dataset/testset.csv’
存储路径	model_save_path	模型存储路径	'./model.pkl'
	result_save_path	结果存储路径	'./result.txt'
最大长度	max_src_len	最大结构文本长度	80

	max_tgt_len	最大参考文本长度	80
词嵌入层	embedding_dim	词嵌入维度	256
	embedding_dropout	词嵌入 Dropout	0.1
编码器	encoder_input_size	编码器输入维度	256
	encoder_hidden_size	编码器隐藏单元数	512
解码器	decoder_input_size	解码器输入维度	512
	decoder_hidden_size	解码器隐藏单元数	512
训练	n_epochs	训练迭代次数	30
	val_num	进行验证的代数	1
	batch_size	批数据大小	16
	learning_rate	学习率	0.1
	device	设备	torch.device('cpu')

4.5 Seq2Seq 模型搭建

该部分在 model.py 中完成。

Seq2Seq 模型属于 Encoder-Decoder 的一种，即利用两个模块，一个模块作为编码器，另一个模块作为解码器。

编码器负责将输入序列压缩成指定长度的向量，可以看作序列的语义，可以直接将最后一个输入隐含状态作为语义向量，或者对其进行变换后再作为语义向量，也可以将所有隐含状态进行变换后再作为语义向量。

解码器则负责根据语义向量生成指定的序列，可以将编码器中得到的语义变量作为初始状态输入到解码器中得到输出序列。

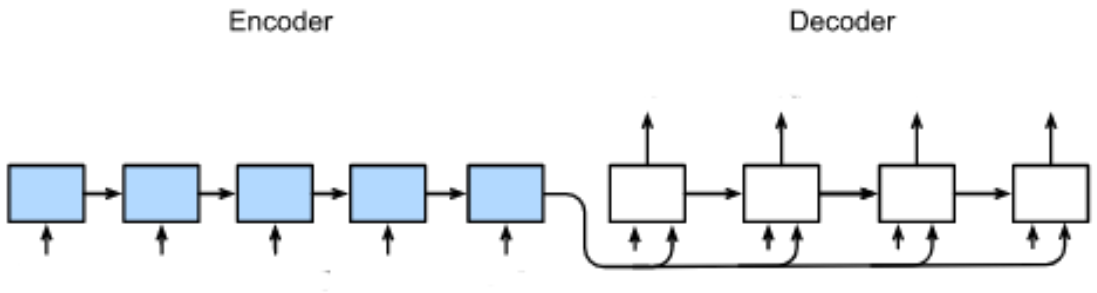

图 2 Seq2Seq 模型

（1）编码器模块 Encoder

首先自定义编码器模块类 class Encoder (nn.Module)，继承自 torch.nn 的 Module 类，自定义模型类需完成__init__、forward 两个函数。

在__init__函数中，定义网络的结构，本实验中采用的编码器是多层感知机 MLP，即使用全连接层 Linear 进行特征提取，然后经过 ReLU 激活，构建 Encoder 时的输入是全连接层的输入大小 input_size 和输出大小 hidden_size。

```
# 编码器 MLP
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size):
```

```
        super(Encoder, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.W = nn.Linear(self.input_size, self.hidden_size)
        self.relu = nn.ReLU()
```

forward 函数在实例化传递参数时自动调用，定义了编码器前馈的连接结构，用在__init__函数中定义的线性层和激活函数层传递输入 input_embedded 得到输出 outputs，然后将输出求和作为解码器的隐藏状态，这里要注意维度。

```
    def forward(self, input_embedded):
        seq_len, batch_size, emb_dim = input_embedded.size()
        outputs = self.relu(self.W(input_embedded.view(-1, emb_dim)))
        outputs = outputs.view(seq_len, batch_size, -1)
        dec_hidden = torch.sum(outputs, 0)
        return outputs, dec_hidden.unsqueeze(0)
```

（2）解码器模块 Decoder

然后自定义解码器模块类 class Decoder (nn.Module)，继承自 torch.nn 的 Module 类，同样需完成__init__、forward 两个函数。

在__init__函数中，定义网络的结构，本实验的采用了注意力解码器，它是由一个简单的解码器和一个注意力模块组成的，简单的解码器通常是 RNN，用于捕捉编码器和解码器之间传递的上下文信息，这里利用 nn.GRU 模块定义 GRU（门控循环神经网络），为了更好地捕捉数据中间隔较大的依赖关系。 Attention 是自定义的注意力模块（在下一节中讲解），通过该模块计算了一组注意力权重，用这些权重乘上编码器输出向量进行加权组合，从而使其包含有关输入序列特定部分的信息，帮助解码器选择正确的输出词。构建 Decoder 时的输入是 RNN 的输入大小 input_size、RNN 的隐藏大小 hidden_size、最终解码输出大小 output_size、词嵌入维度 embedding_dim 以及编码器的输出大小 encoder_ hidden_size。

```
class Decoder(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, embedding
_dim, encoder_hidden_size):
        super(Decoder, self).__init__()
        self.rnn = nn.GRU(input_size, hidden_size, bidirectional=False)
        # 注意力模块
        self.attn_module = Attention(encoder_hidden_size, hidden_size)
        self.W_combine = nn.Linear(embedding_dim + encoder_hidden_size,
 hidden_size)
        self.W_out = nn.Linear(hidden_size, output_size)
        self.log_softmax = nn.LogSoftmax()
```

forward 函数在实例化传递参数时自动调用，定义了解码器前馈的连接结构，输入是上一时刻的输出 prev_y_batch、上一时刻的隐藏状态 prev_h_batch、以及编码器的输出 encoder_outputs_batch。首先使用在__init__函数中定义的注意力模块 att_module 传递上一时刻的隐藏状态 prev_h_batch 和编码器的输出 encoder_outputs_batch（具体过程在下节），得到注意力权重，调用 bmm 函数对

编码器输出的结果进行加权，然后再经过 GRU 进行解码，得到当前时刻解码的输出和隐藏状态，再经过全连接层和 softmax 进行激活得到词的概率。

```
    def forward(self, prev_y_batch, prev_h_batch, encoder_outputs_batch):
        attn_weights = self.attn_module(prev_h_batch, encoder_outputs_batch)  # B x SL
        # 对编码器的输出进行加权
        context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs_batch.transpose(0, 1))  # B x 1 x MLP_H
        # 经过 RNN（GRU）解码
        # B x (prev_y_dim+(enc_dim * num_enc_directions))
        y_ctx = torch.cat((prev_y_batch, context.squeeze(1)), 1)
        rnn_input = self.W_combine(y_ctx)  # B x H
        dec_rnn_output, dec_hidden = self.rnn(rnn_input.unsqueeze(0), prev_h_batch)  # 1 x B x H, 1 x B x H
        # 计算输出概率
        unnormalized_logits = self.W_out(dec_rnn_output[0])  # B x TV
        dec_output = self.log_softmax(unnormalized_logits)  # B x TV
        # 返回最终输出、隐藏状态以及注意力权重
        return dec_output, dec_hidden, attn_weights
```

（3）注意力模块 Attention

接下来自定义注意力模块类 class Attention(nn.Module)，继承自 torch.nn 的 Module 类，同样需完成__init__、forward 两个函数。

在__init__函数中，定义网络的结构，本实验的采用了 Bahdanau 等人提出的注意力模型。注意力的本质实际上是一些参数运算，其代表的实际含义是有关输入序列特定部分的信息，利用这些信息能够充分利用原始句子信息，减少信息缺失。线性层表示的是加权求和，定义 Linear 层 U、W、v，一个 Tanh 层和一个 Softmax 层，构建 Attention 时的输入是编码维度 enc_dim 和解码器维度 dec_dim。

```
# 注意力模块
class Attention(nn.Module):
    def __init__(self, enc_dim, dec_dim, attn_dim=None):
        super(Attention, self).__init__()
        self.num_directions = 1
        self.h_dim = enc_dim
        self.s_dim = dec_dim
        self.a_dim = self.s_dim if attn_dim is None else attn_dim
        # 构建注意力
        self.U = nn.Linear(self.h_dim * self.num_directions, self.a_dim)
        self.W = nn.Linear(self.s_dim, self.a_dim)
        self.v = nn.Linear(self.a_dim, 1)
        self.tanh = nn.Tanh()
        self.softmax = nn.Softmax()
```

forward 函数在实例化传递参数时自动调用，定义了解码器前馈的连接结构，输入是上一时刻的隐藏状态 prev_h_batch 和编码器的输出 enc_outputs，根据 Bahdanau 等人提出的注意力模型的计算公式：

$$a(s_{i-1}, h_j) = v_a^T \tanh(W_a s_{i-1} + U_a h_j)$$

其中，s_{i-1}表示上一时刻的隐藏状态，h_j表示编码器的输出。上面定义了线性层 U、W、v，这里直接根据公式，先将编码器的输出 enc_outputs 传入 U 进行加权求和，再上一时刻的隐藏状态 prev_h_batch 传入 W 进行加权求和，然后经过 tanh 函数，再与 v 进行加权求和，过程中注意维度的一致性。

```
    def forward(self, prev_h_batch, enc_outputs):
        src_seq_len, batch_size, enc_dim = enc_outputs.size()
        uh = self.U(enc_outputs.view(-1, self.h_dim)).view(src_seq_len,
 batch_size, self.a_dim)  # SL x B x self.attn_dim
        wq = self.W(prev_h_batch.view(-1, self.s_dim)).unsqueeze(0)  #
1 x B x self.a_dim
        wq3d = wq.expand_as(uh)
        wquh = self.tanh(wq3d + uh)
        attn_unnorm_scores = self.v(wquh.view(-1, self.a_dim)).view(bat
ch_size, src_seq_len)
        attn_weights = self.softmax(attn_unnorm_scores)  # B x SL
        return attn_weights
```

（4）序列到序列模型 E2EModel

最后自定义序列到序列模型类 class E2EModel (nn.Module)，继承自 torch.nn 的 Module 类，同样需完成__init__、forward 两个函数，另外还定义了用于推理的 predict 函数。

在__init__函数中，定义网络的结构，利用上面定义的编码器模块 Encoder、带注意力的解码器模块 Decoder 构建网络。网络中除了编码器和解码器，还定义了词嵌入层 Embedding 和一个 Dropout 层，Dropout 层主要是为了放弃一些 Embedding 的无用特征，防止过拟合。构建 E2EModel 时的输入是配置类 cfg、结构化文本词表大小 src_vocab_size 和参考文本词表大小 tgt_vocab_size。

```
# Seq2Seq 模型
class E2EModel(nn.Module):
    def __init__(self, cfg, src_vocab_size, tgt_vocab_size):
        super(E2EModel, self).__init__()
        self.device = cfg.device  # 设备
        self.cfg = cfg
        self.src_vocab_size = src_vocab_size
        self.tgt_vocab_size = tgt_vocab_size
        # 构建词嵌入层
        self.embedding_mat = nn.Embedding(src_vocab_size, cfg.embedding
_dim, padding_idx=PAD_ID)
        self.embedding_dropout_layer = nn.Dropout(cfg.embedding_dropou
t)
        # 构建编码器和解码器
```

```
        self.encoder = Encoder(input_size=cfg.encoder_input_size,
                               hidden_size=cfg.encoder_hidden_size)
        self.decoder = Decoder(input_size=cfg.decoder_input_size,
                               hidden_size=cfg.decoder_hidden_size,
                               output_size=tgt_vocab_size,
                               embedding_dim=cfg.embedding_dim,
                               encoder_hidden_size=cfg.encoder_hidden_size)
```

forward 函数在实例化传递参数时自动调用，定义了解码器前馈的连接结构，输入是批数据 data。首先从批数据中解析得到 batch_x_var 和 batch_y_var，然后进行词嵌入和 Dropout，再使用 encoder 对嵌入向量进行编码，然后使用 decoder 对编码结果进行逐一解码，这里使用了 teaching forcing 机制进行学习，teaching forcing 机制是为解决在训练开始初期，由于前一个单元产生的结果对后续单元的不良影响，导致学习速度变慢，难以收敛。teaching forcing 机制是指在预测下一时刻的输出和隐藏状态时，总是使用标准答案作为上一时刻的输出，这样基于标准答案的推理可以加速模型收敛的速度，最后得到解码的每个位置的词概率。

```
    def forward(self, data):
        batch_x_var, batch_y_var = data  # SL x B, TL x B
        # 词嵌入 # SL x B x E
        encoder_input_embedded = self.embedding_mat(batch_x_var)
        encoder_input_embedded = self.embedding_dropout_layer(encoder_
input_embedded)
        # 编码 SL x B x H; 1 x B x H
        encoder_outputs, encoder_hidden = self.encoder(encoder_input_em
bedded)
        # 解码
        dec_len = batch_y_var.size()[0]
        batch_size = batch_y_var.size()[1]
        dec_hidden = encoder_hidden
        dec_input = Variable(torch.LongTensor([BOS_ID] * batch_size)).t
o(self.device)
        logits = Variable(torch.zeros(dec_len, batch_size, self.tgt_voc
ab_size)).to(self.device)
        # 采用 Teacher forcing 机制，输入总是标准答案
        for di in range(dec_len):
            # 上一输出的词嵌入，B x E
            prev_y = self.embedding_mat(dec_input)
            # 解码
            dec_output, dec_hidden, attn_weights = self.decoder(prev_y,
 dec_hidden, encoder_outputs)
            logits[di] = dec_output  # 记录输出词概率
            dec_input = batch_y_var[di]  # 下一输入是标准答案
        return logits
```

predict 函数是在推理时调用的，输入只需要结构化文本的数据，由于测试时

没有参考的文本，因此验证时也需要模拟这种环境，从而得到更具参考价值的验证评分。与 forward 函数类似，首先进行词嵌入和 Dropout，再使用 encoder 对嵌入向量进行编码，然后使用 decoder 对编码结果进行逐一解码，在解码时，需要记录已经产生的结果长度，直到达到最大长度或者出现结束标记 EOS，解码的过程也和 forward 阶段类似，先对上一时刻的输出进行词嵌入然后进行解码得到输出、隐藏状态和注意力矩阵，区别在于这里需要选择最大的词概率对应的下标，即最大可能性的词编号，对词编号和注意力矩阵进行记录并将预测的当前时刻的词编号作为下一时刻的输入。最后，返回解码的结果和注意力矩阵。

```
    def predict(self, input_var):
        # 词嵌入
        encoder_input_embedded = self.embedding_mat(input_var)
        # 编码
        encoder_outputs, encoder_hidden = self.encoder(
                                              encoder_input_embedded)
        # 解码
        dec_ids, attn_w = [], []
        curr_token_id = BOS_ID
        curr_dec_idx = 0
        dec_input_var = Variable(torch.LongTensor([curr_token_id]))
        dec_input_var = dec_input_var.to(self.device)
        dec_hidden = encoder_hidden[:1]  # 1 x B x enc_dim
        # 直到 EOS 或达到最大长度
        while curr_token_id != EOS_ID and curr_dec_idx <= self.cfg.max_
tgt_len:
            prev_y = self.embedding_mat(dec_input_var)  # 上一输出的词嵌
入，B x E
            # 解码
            decoder_output, dec_hidden, decoder_attention = self.decode
r(prev_y, dec_hidden, encoder_outputs)
            # 记录注意力
            attn_w.append(decoder_attention.data.cpu().numpy().tolist()
[0])
            topval, topidx = decoder_output.data.topk(1)   # 选择最大概率
            curr_token_id = topidx[0][0]
            # 记录解码结果
            dec_ids.append(int(curr_token_id.cpu().numpy()))
            # 下一输入
            dec_input_var = (Variable(torch.LongTensor([curr_token_i
d]))).to(self.device)
            curr_dec_idx += 1
        return dec_ids, attn_w
```

4.6 评价指标计算

该部分在 metric.py 中完成。

本实验中使用的评价指标为 BLEU-4。BLEU 是一种基于准确率的指标，计算时不考虑候选文本的召回率。BLEU-n 则是选取长度不同的 n-gram 对候选文本进行综合评估。该指标通过 unigram 评估候选文本是否正确使用了关键数据中的单词；通过高阶的 n-gram 评估句子是否流畅。此外，BLEU 还引入了参数 BP（Brevity Penalty）惩罚长度过短的候选文本。

$$Bleu = BP \cdot exp\left(\sum_{n=1}^{N} w_n \, log \, P_n\right)$$

$$BP = \begin{cases} 1, c > r \\ e^{1-r/c}, c \le r \end{cases}$$

其中，w_n为 n-gram 的权重(一般而言不同 n-gram 权重相同)，P_n为候选文本 n-gram 的得分，c为候选文本的长度，r为与候选文本长度最近接的参考文本的长度。

为保证评价指标在验证集上的结果对本任务中的评分标准有一定指示作用，评价指标代码参考 https://github.com/salesforce/WikiSQL 中 BLEU 代码，使其能够直接被主函数调用并在验证环节进行评价。

（1）初始化 BLEUScore 类

定义 BLEUScore 类用于计算 BLEU score，TINY 和 SMALL 为两个极小量，初始化__init__函数有两个输入，分别是最大 gram 数量 max_ngram 和样本的敏感性 case_sensitive（控制是否对大小写敏感），赋值这些属性，并调用 reset 函数对命中列表和候选长度列表进行初始化。

```
class BLEUScore:
    TINY = 1e-15
    SMALL = 1e-9
    def __init__(self, max_ngram=4, case_sensitive=False):
        self.max_ngram = max_ngram  # 最大 n-gram
        self.case_sensitive = case_sensitive
        self.reset()
```

reset 函数用于初始化参考文本长度为 0，以及两个用于记录不同 gram 的命中列表和候选长度列表，列表长度对应为 max_ngram。

```
    def reset(self):
        self.ref_len = 0
        self.cand_lens = [0] * self.max_ngram
        self.hits = [0] * self.max_ngram
```

（2）统计预测句子和参考句子的命中信息

append 函数用来统计不同 gram 的命中数量和候选长度，输入是预测的句子 pred_sent 和参考文本列表 ref_sents，首先调用 tokenize 函数对输入的句子进行分词，主要是去除一些标点符号并将词按空格分开，和前面定义的预处理函数相似，这里不再赘述，对于参考文本的列表则对每个参考句子分别进行分词，然后分别计算直到 max_ngram 为止的每个 gram，调用 compute_ hits 函数计算命中次数并累加，cand_lens 用于统计每个 gram 的预测长度，最后选择最相近的参考文本并

记录参考文本的长度。

```
    # 添加句子
    def append(self, pred_sent, ref_sents):
        pred_sent = pred_sent if isinstance(pred_sent, list) else self.
tokenize(pred_sent)
        ref_sents = [ref_sent if isinstance(ref_sent, list) else self.t
okenize(ref_sent)
                     for ref_sent in ref_sents]
        for i in range(self.max_ngram):
            # 计算每个 gram 的命中次数
            self.hits[i] += self.compute_hits(i + 1, pred_sent,
                                                    ref_sents)
            # 计算每个 gram 的预测长度
            self.cand_lens[i] += len(pred_sent) - i
        # 选择长度最相近的参考文本
        closest_ref = min(ref_sents, key=lambda ref_sent: (abs(len(ref_
sent) - len(pred_sent)), len(ref_sent)))
        # 记录参考文本长度
        self.ref_len += len(closest_ref)
```

compute_hits 函数用于统计命中次数，n 是 gram 数，调用 get_ngram_counts 得到 ngram 的句子列表，遍历句子列表统计预测句子和参考文本的命中次数，

```
    def compute_hits(self, n, pred_sent, ref_sents):
        merged_ref_ngrams = self.get_ngram_counts(n, ref_sents)
        pred_ngrams = self.get_ngram_counts(n, [pred_sent])
        hits = 0
        for ngram, cnt in pred_ngrams.items():
            hits += min(merged_ref_ngrams.get(ngram, 0), cnt)
        return hits
```

get_ngram_counts 函数作用是得到按 gram 数 n 进行聚合后的句子列表。

```
    def get_ngram_counts(self, n, sents):
        merged_ngrams = {}
        # 按 gram 数聚合句子
        for sent in sents:
            ngrams = defaultdict(int)
            if not self.case_sensitive:
                ngrams_list = list(zip(*[[tok.lower() for tok in sent
[i:]] for i in range(n)]))
            else:
                ngrams_list = list(zip(*[sent[i:] for i in range(n)]))
            for ngram in ngrams_list:
                ngrams[ngram] += 1
            for ngram, cnt in ngrams.items():
```

```
                merged_ngrams[ngram] = max((merged_ngrams.get(ngram, 0),
cnt))
        return merged_ngrams
```

（3）计算 BLEU score

score 函数用于计算 BLEU score，根据 BLEU 评分的计算方式，前半部分计算BP，分为 c>r 和 c<=r 两种情况，注意防止分母为 0，后半部分计算$\sum_{n=1}^{N} log\, P_n$，w_n为 1/max_ngram，最后将其相乘得到$Bleu = BP \cdot exp(\sum_{n=1}^{N} w_n\, log\, P_n)$。

```
    def score(self):
        bp = 1.0
        # c <= r : BP=e^(1-r/c)
        # c > r : BP=1.0
        if self.cand_lens[0] <= self.ref_len:
            bp = math.exp(1.0 - self.ref_len / (float(self.cand_lens[0])
if self.cand_lens[0] else 1e-5))
        prec_log_sum = 0.0

        for n_hits, n_len in zip(self.hits, self.cand_lens):
            n_hits = max(float(n_hits), self.TINY)
            n_len = max(float(n_len), self.SMALL)
            # 计算∑logPn=∑log(n_hits/n_len)
            prec_log_sum += math.log(n_hits / n_len)
        return bp * math.exp((1.0 / self.max_ngram) * prec_log_sum)
```

（4）调用示例

最后，利用 if __name__ == ‘__main__’在 metric.py 中定义使用 BLEUScore 类的示例，主程序运行时不会调用该部分。首先，实例化 BLEUScore 类，设置 max_ngram 为 2，根据 PPT 上给定的句子 the cat sat on the mat 以及参考句子 the cat is on the mat，调用 append 函数在 scorer 中加入当前句子计算 1-gram、2-gram 的命中数和词总数，最后调用 score 函数计算 BLEU-2 的评分，当对数 log 底数为 10 时运行结果为 0.860264827803306，与 PPT 给定的示例一致，但由于评估时使用的对数是以 e 为底的，因此在验证时也以 e 作为 log 的底数，作为验证集上的评估参考。

```
if __name__ == '__main__':
    scorer = BLEUScore(max_ngram=2)
    sentence = 'the cat sat on the mat '
    target = ['the cat is on the mat ']
    scorer.append(sentence, target)
    print(scorer.score())
```

4.7 主程序

该部分在 main.py 中完成。

首先，根据当前设备是否能够使用 GPU，设置 device 变量。

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("using", device, "...")
```

实例化 Config 类为 cfg 定义实验所需的参数，除 device 外，其余参数均为 4.4 中的默认值。

（1）实例化数据加载器

调用定义的数据集类 E2EDataSet，加载训练集、验证集、测试集，然后调用 torch.utils.data 中的 DataLoader 构建数据集加载器，是否混洗 shuffle 等参数。

```
# 读取数据
train_set = E2EDataset(cfg.train_data,
                       mode='train',
                       max_src_len=cfg.max_src_len,
                       max_tgt_len=cfg.max_tgt_len)
dev_set = E2EDataset(cfg.dev_data,
                     mode='dev',
                     max_src_len=cfg.max_src_len,
                     max_tgt_len=cfg.max_tgt_len,
                     field_tokenizer=train_set.field_tokenizer,
                     tokenizer=train_set.tokenizer)
test_set = E2EDataset(cfg.test_data,
                      mode='test',
                      max_src_len=cfg.max_src_len,
                      max_tgt_len=cfg.max_tgt_len,
                      field_tokenizer=train_set.field_tokenizer,
                      tokenizer=train_set.tokenizer)
train_loader = DataLoader(train_set, batch_size=cfg.batch_size,
shuffle=True)
```

（2）实例化模型

调用上面定义的 E2EModel 类实例化自定义网络类，并将其移至 device 上。

```
# 初始化模型
model = E2EModel(cfg,
            src_vocab_size=train_set.tokenizer.vocab_size,
            tgt_vocab_size=train_set.tokenizer.vocab_size).to(device)
```

（3）配置训练优化器、损失函数等

在模型训练前，需要设置损失函数和优化器以及训练轮数和经过多少轮数验证，后两者均用命令行参数直接赋值。对于多分类任务，常用的损失函数有交叉熵 CrossEntroy、负对数似然 NLLoss 等，这里选择负对数似然 NLLoss，这些损失函数都直接用 torch.nn 中已设置好的函数，设置填充词 PAD_ID 的权重为 0，即优化时不考虑填充词的损失；常用的优化器有 SGD、Adam 等，这里选择 SGD，设置参数为参与更新计算的模型参数 model.parameters，并设置学习率 lr，优化器同样用来自 torch.optim 模块中自带的优化器；分数评估器使用上面定义的 BLEUScore 类，并指定 max_ngram 为 4，即计算 BLEU-4 评分。学习率调整策略来自 torch.optim.lr_scheduler 模块，这里设置 CosineAnnealingLR 在训练过程中自

适应调整学习率。

```
# 初始化 batch 大小，epoch 数，损失函数，优化器等
batch_size = cfg.batch_size
weight = torch.ones(train_set.tokenizer.vocab_size)
weight[PAD_ID] = 0
# 定义分数评估
scorer = BLEUScore(max_ngram=4)
# 定义损失函数
criterion = nn.NLLLoss(weight, size_average=True).to(device)
# 定义优化器
optimizer = optim.SGD(params=model.parameters(), lr=cfg.learning_rat
e)
# 定义学习率下降
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_m
ax=200)
MAX_EPOCH = cfg.n_epochs
VAL_NUM = cfg.val_num
```

这里还定义了 best_bleu 参数记录最高 bleu 用于模型保存以及三个列表记录 loss、bleu 和 lr，用于后续绘制变化曲线。

```
best_bleu = 0.0  # 最高 bleu
loss_li = []
bleu_li = []
lr_li = []
```

（4）模型训练

循环 MAX_EPOCH 个迭代进行模型的训练。

```
# 训练集用于训练，验证集用于评估
for epoch in range(MAX_EPOCH):
    train(model, train_loader, epoch)
```

模型训练调用 train 函数，每个 epoch 用所有训练数据训练一次，训练需要设置 model.train 切换回训练模式，因为验证阶段切会换到验证模式。由于训练过程较长，因此用进度条库 tqdm 进行训练过程的可视化。tqdm 函数传入参数有迭代器 train_loader 的长度，另外还设置了可选参数：desc 是进度条前显示的文字，这里指示 epoch 数；file 是进度条输出源，由于在代码中用到 print 函数，会导致输出进度条和文字顺序混乱，因此将进度条和 print 设为同一输出源，即 sys.stdout，进度条名为 t，然后遍历迭代器产生的批数据。

```
def train(model, iterator, iter):
    print_loss = 0.0  # 输出 loss
    model.train()
    with tqdm(total=len(iterator), desc='epoch{} [train]'.format(ite
r), file=sys.stdout) as t:
        for i, batch in enumerate(iterator):
```

接着是训练的主要内容，这里需要计算损失进行梯度下降。从迭代器返回的

批数据中提取结构化文本 src 和目标参考文本 ref，同样，将其移至自动检测的 GPU 或 CPU 设备上。

```
src, tgt = batch
# 移至设备上
src = src.to(device).transpose(0, 1)
tgt = tgt.to(device).transpose(0, 1)
```

然后，首先利用优化器的 zero_grad 方法初始化梯度值，计算模型预测的损失。将结构化文本 src 和目标参考文本 ref 输入当前模型 model 中得到输出 logits，并根据损失函数计算输出和目标参考文本之间的 loss，这里需要注意维度，print_loss 参数用于打印，这里统计目前为止的损失和，用于显示损失均值。

```
optimizer.zero_grad()  # 初始化梯度值
# Forward
logits = model((src, tgt))
vocab_size = logits.size()[-1]
logits = logits.contiguous().view(-1, vocab_size)
targets = tgt.contiguous().view(-1, 1).squeeze(1)
loss = criterion(logits, targets.long())
print_loss += loss.data.item()
```

梯度下降时，首先利用优化器的 zero_grad 方法初始化梯度值，再根据损失的 backward 方法反向传播求梯度，利用优化器的 step 方法更新参数。

```
# Backward
loss.backward()  # 反向传播求梯度
# Update
optimizer.step()  # 更新参数
```

然后，在进度条更新平均损失和当前的学习率。

```
# Backward
loss.backward()  # 反向传播求梯度
# Update
optimizer.step()  # 更新参数
t.set_postfix(loss=print_loss / (i + 1), lr=scheduler.get_last_lr()[0])
t.update(1)
```

最后，将当前损失和学习率添加到列表中用于后续绘图，利用学习率调整器的 step 方法更新学习率。

```
loss_li.append(print_loss / len(iterator))
lr_li.append(scheduler.get_last_lr()[0])
scheduler.step()
```

（5）模型验证

在每 VAL_NUM 个迭代进行一次模型的评估。

```
# 指定轮数验证
if epoch % VAL_NUM == 0:
    evaluate(model, dev_set, epoch)
```

模型验证环节调用 evaluate 函数，每指定 epoch 数调用一次。total_num 统计验证集总数据数量，bleu 统计所有句子的 BLEU-4 分数和，最后取均值。调用 model.eval 切换为验证状态，用以关闭某些特定层的行为，如 Dropout、BatchNormal 等，在验证阶段固定这些层的参数，防止在 batch_size 小时对测试结果的影响。这里使用 torch.no_grad 函数，由于验证环节不需要梯度下降，这部分的代码不跟踪梯度，从而实现一定速度提升并且节省用于存储梯度的存储空间。

```
def evaluate(model, iterator, iter):
    global best_bleu
    model.eval()
    bleu = 0.0
    total_num = 0
    # 重置分数统计器
    scorer.reset()
    with torch.no_grad():
```

类似于训练阶段，这里同样使用 tqdm 库显示验证进度，使用验证集数据集的 dataset，验证时得到的数据为结构化文本 src、目标参考文本 tgt、去词化原词 lex 以及多参考文本 multi_tgt。在验证阶段，不需要进行损失计算和梯度下降，仅需要利用 model 的 predict 函数对参考文本进行预测并进行解码得到预测的句子 sentence，调用 append 函数和 score 函数对当前句子和多个参考文本进行评估，得到 BLEU-4 评分并累记。

```
        for data in tqdm(iterator, desc='{} [valid]'.format(" " * (5 + len(str(iter)))), file=sys.stdout):
            # 重置分数统计器
            src, tgt, lex, muti_tgt = data
            src = torch.as_tensor(src[:, np.newaxis]).to(device)
            sentence, attention = model.predict(src)
            # 解码句子
            sentence = train_set.tokenizer.decode(sentence).replace('[NAME]', lex[0]).replace('[NEAR]', lex[1])
            scorer.append(sentence, muti_tgt)
```

最后，计算 BLEU-4 的均值并对模型进行保存，当模型的 BLEU-4 大于记录的最优 BLEU-4 时，则将其保存在配置的模型保存路径。

```
    bleu = scorer.score()  # 计算 BLEU
    bleu_li.append(bleu)
    print("BLEU SCORE: {:.4f}".format(bleu))
    if bleu > best_bleu:
        best_bleu = bleu
        torch.save(model, cfg.model_save_path)  # 保存模型 参数 网络 路径
        print("保存模型成功！")
```

（6）训练损失和验证 BLEU 变化曲线

利用前面定义的 loss_li、acc_li、lr_li 列表中的历史训练损失、验证 BLEU 分数和学习率，使用 matplotlib.pyplot 模块绘制训练损失和验证评分的变化图，并

保存至本地。

```
        # 绘制验证 BLEU 曲线
        draw_carve("valid_bleu", './valid_bleu.png', epoch // VAL_NUM
    + 1, bleu_li)
    # 绘制训练 LOSS 曲线
    draw_carve("train_loss", './train_loss.png', epoch + 1, loss_li)
    # 绘制训练 LR 曲线
    draw_carve("train_lr", './train_lr.png', epoch + 1, lr_li)
```

这里定义了 draw_carve 函数用于绘图，输入是图像标题 title、保存路径 save_path、自变量 x 和因变量 y，调用 plt 库的 clf 函数清空之前的图像，title 函数设置图像标题，plot 函数绘制曲线，savefig 函数保存图像到本地。

```
def draw_carve(title, save_path, x, y):
    plt.clf()  # 清空图像
    plt.title(title)  # 图像标题
    plt.plot(range(x), y)  # 绘制图像
    plt.savefig(save_path)  # 保存图像
```

（7）模型测试

模型测试时需要对所有测试集进行预测并按行输出到 txt 文件中。同样，测试不需要计算梯度，类似于训练和验证阶段，读取批数据并利用训练的模型预测。得到预测结果列表，利用词典进行解码为句子，然后逐行写入文本。

```
model.eval()
with torch.no_grad():
    for data in tqdm(test_set, desc='[test]', file=sys.stdout):
        src, tgt, lex, _ = data
        src = torch.as_tensor(src[:, np.newaxis]).to(device)
        # 模型预测
        sentence, attention = model.predict(src)
        # 解码句子
        sentence = train_set.tokenizer.decode(sentence).replace('[NAM
E]', lex[0]).replace('[NEAR]', lex[1])
        # 写入文本
        with open(cfg.result_save_path, 'a+', encoding='utf-8') as f:
            f.write(sentence + '.\n')
print('Finished Testing!')
```

4.8 注意力可视化

该部分在 visualize.py 中完成。

该部分是独立于主函数的单独模块，通过加载训练好的模型，并对单句进行预测得到注意力矩阵，根据注意力矩阵的值进行热力图的绘制。首先调用 torch.load 函数加载模型，然后选择单句进行输入，得到预测的句子编码和注意力矩阵。

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("using", device, "...")
# 读取模型
model = torch.load('./model.pkl').to(device)
dataset = E2EDataset('./e2e_dataset/trainset.csv', mode='train')
dataset.mode = 'dev'
# 获得数据
src, tgt, lex, _ = dataset[0]
src = torch.as_tensor(src[np.newaxis, :]).to(device).transpose(0, 1)
sentence, attention = model.predict(src)
```

根据词典和去词化的原词，还原结构化文本和预测的输出作为坐标轴的标签。

```
# 还原文本
src_txt = list(map(lambda x: dataset.tokenizer.id_to_token(x), src.fla
tten().cpu().numpy().tolist()[:10]))
for i in range(len(src_txt)):
    if src_txt[i] == '[NAME]':
        src_txt[i] = lex[0]
    elif src_txt[i] == '[NEAR]':
        src_txt[i] = lex[1]
sentence_txt = list(map(lambda x: dataset.tokenizer.id_to_token(x), se
ntence))
for i in range(len(src_txt)):
    if sentence_txt[i] == '[NAME]':
        sentence_txt[i] = lex[0]
    elif sentence_txt[i] == '[NEAR]':
        sentence_txt[i] = lex[1]
```

最后调用 seaborn 库中的 heatmap 函数进行绘图，并利用 yticks 和 xticks 设置坐标轴的标签，并调用 plt.show 展示图像。

```
# 绘制热力图
ax = sns.heatmap(np.array(attention)[:, :10] * 100, cmap='YlGnBu')
# 设置坐标轴
plt.yticks([i + 0.5 for i in range(len(sentence_txt))], labels=sentenc
e_txt, rotation=360, fontsize=12)
plt.xticks([i + 0.5 for i in range(len(src_txt))], labels=src_txt, fon
tsize=12)
plt.show()
```

五、 实验结果与分析

运行输出如图 3 所示，参数均采用第四部分中表 2 给出的默认参数。训练集中有 42061 条数据，因此每个训练 epoch 需要 42061÷256=165 次迭代，每个训

练 epoch 花费平均时间为 4min，最后一次训练损失为 1.58；验证集中有 4672 条数据，因此每次验证需要 4672 次迭代，每 1 次训练验证一次，每个验证花费平均时间为 2min，30 个 epoch 中验证 BLEU-4 评分为 0.6738；测试集中有 630 条数据，因此测试需要 630 次迭代，花费时间为 18s。

```
epoch25 [train]: 100%|██████████| 165/165 [04:11<00:00,  1.52s/it, loss=1.61, lr=0.0962]
        [valid]: 100%|██████████| 4672/4672 [01:58<00:00, 39.33it/s]
BLEU SCORE: 0.6448
epoch26 [train]: 100%|██████████| 165/165 [04:15<00:00,  1.55s/it, loss=1.6, lr=0.0959]
        [valid]: 100%|██████████| 4672/4672 [02:08<00:00, 36.23it/s]
BLEU SCORE: 0.6656
epoch27 [train]: 100%|██████████| 165/165 [04:06<00:00,  1.49s/it, loss=1.59, lr=0.0956]
        [valid]: 100%|██████████| 4672/4672 [02:10<00:00, 35.78it/s]
BLEU SCORE: 0.6738
保存模型成功!
epoch28 [train]: 100%|██████████| 165/165 [04:14<00:00,  1.54s/it, loss=1.58, lr=0.0952]
        [valid]: 100%|██████████| 4672/4672 [02:14<00:00, 34.73it/s]
BLEU SCORE: 0.6647
epoch29 [train]: 100%|██████████| 165/165 [04:06<00:00,  1.49s/it, loss=1.58, lr=0.0949]
        [valid]: 100%|██████████| 4672/4672 [02:07<00:00, 36.78it/s]
BLEU SCORE: 0.6771
保存模型成功!
Finished Training!

[test]: 100%|██████████| 630/630 [00:18<00:00, 33.35it/s]
Finished Testing!
```

图 3 运行过程

输出的训练损失变化曲线、验证 BLEU 变化曲线以及学习率变化曲线如下图所示，前 5 个 epoch 模型收敛较快，此时学习率大，损失变化曲线较陡，之后收敛稍慢，学习率减小，损失和 BLEU 变化较缓，30 个 epoch 时型已逐渐收敛。

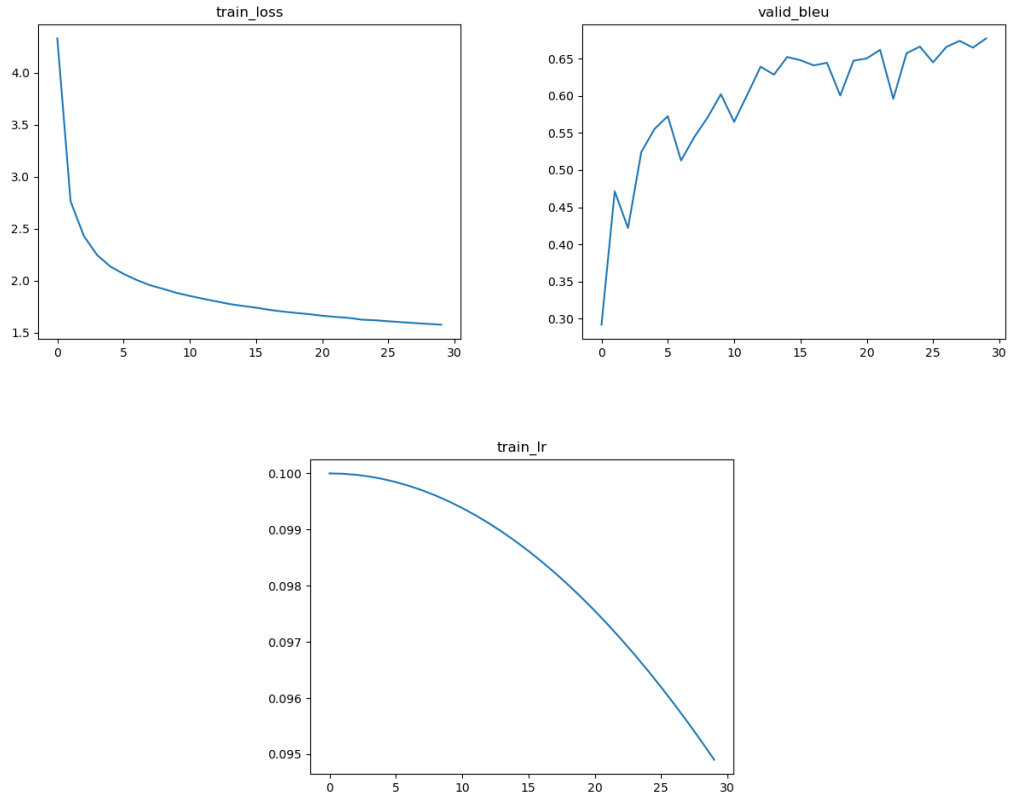

图 4 训练 LOSS、验证 BLEU、训练 LR 变化曲线

图 5 是在利用某个结构化文本生成文本时输出的注意力权值绘制的热力图，其权值被放大了 100 倍。横坐标为属性值分别对应名字 name、餐馆食物类型 food、priceRange 价格区间、客流量 customer rating、是否家庭友好 familyFriendly、区域 area、地点 near、餐馆类型 eatType，0 则是未给定属性值；纵坐标表示编码的结果句子结果，在图像中最明显的深蓝色区域对应的横坐标是输入属性 Café Adriatic，纵坐标是输出结果中的词 near，而 Café Adriatic 正好代表了当前餐馆的地点邻近信息，这一项权值大说明模型中 Café Adriatic 对于推理出 near 这个词的帮助很大，而这正是一项有用的信息，这也说明注意力机制能够帮助模型关注到与当前输出相关的有用的输入信息，从而提高输出的质量，更好地表示这些信息。

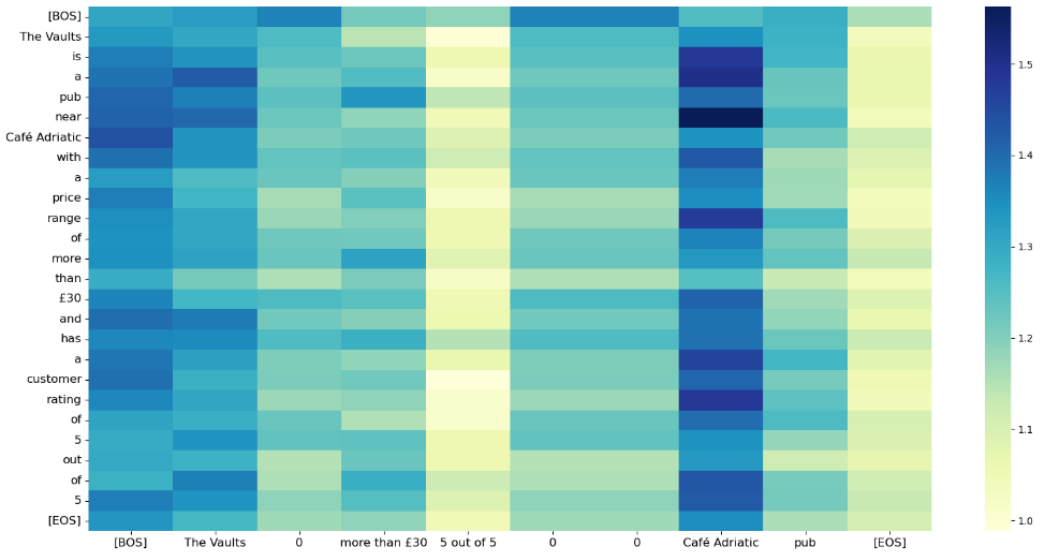

图 5 Attention 效果可视化

六、 心得体会

本实验基于结构化数据的序列到序列生成任务，本次实验中通过对结构化文本的预处理、定义了 Seq2Seq 模型，并参照 BLEU 评估源码添加了最终实现从结构化数据到参考文本的生成。实验比之前难度有所提升，对于结构化数据，如何编码是值得考虑的问题，实验中将所有的结构化数据都加入到相同维度的列表中，用不同位置表示不同属性，这样作为模型的输入向量。相比于传统的由两个 RNN 的 Seq2Seq 模型，本实验中采用了 MLP 作为编码器并使用了增加注意力机制组成的 GRU 解码器，最终模型在验证集上的 BLEU-4 能够达到在验证集上 0.67。

通过几次实验循序渐进，这次学习到了 Seq2Seq 模型的构建方式，并尝试了 Attention 机制，对其原理也有一定的了解，学习了结构化数据的序列到序列生成的过程，并且在验证集上取得了相对较好的得分，绘制了热力图将注意力矩阵进行可视化，熟悉了注意力机制的作用和效果。