姓名：王想
学号：2100013146

Homework 4

课程名称：自然语言处理基础
2023 年 6 月 28 日

1 parser_utils.py 每个函数所完成的任务

1.1 Class Config

这个类记录 Parser 的一些超参数 (比如是否带 label、构建词库的时候是否转化为小写、是否带有标点符号等) 和文件路径。

1.2 Class Parser

init 的作用是构建词库，构建从 token 或 action 到 id 的映射、从 id 到 token 或 action 的映射。

vectorize 的作用是把 read_conll 读取出来的句子字典中的字符串 token 转化成对应的词库的 id，从而将句子与标注向量化表示。

extract_features 的作用是抽取一个句子 parse 的特定的状态所对应的特征，具体实现将在下一节中介绍。

get_oracle 的作用是根据当前句子 parse 的状态和句子的标注确定下一步的动作 action，具体实现将在下一节中介绍。

create_instances 的作用生成训练所需的 groundtruth action 序列, 具体实现将在下一节中介绍。

legal_labels 的作用是判断哪些 label 是合法的，在返回的列表中所有合法的 label 对应的位置上为 1 否则为 0.

parse 的作用是调用 parser_transitions.py 中的 minibatch_parse 对句子进行 parse，并且计算出相应的 UAS，后来这部分我改成了 LAS，不过在这里无论是计算 UAS 还是 LAS 我感觉都没什么必要，因为在验证集或者测试集上测试的时候最后还是要调用 utils 中的 evaluate 函数统一计算 UAS 与 LAS。所以我看到这部分代码的时候还是有点疑惑的，不明白为什么这里要计算 UAS，这里完全可以不用计算 UAS，返回值也不需要包含 UAS，在 main 函数里面调用这个函数的时候也没有保留返回的 UAS。

1.3 Class ModelWrapper

init 初始化此类，无其他作用。

predict 功能是根据当前的 partial_parses 状态，预测接下来的 action。

1.4 其他函数

read_conll 功能是把.coll 数据文件中的数据按照固定的字典格式读取出来。每个句子的数据读取成一个字典，由四个键值对组成，'word' 的值为这个句子的单词列表，'pos' 的值为对应的单词的词性，'head' 的值为对应的单词的父单词，'label' 的值为对应的单词与其父单词的修饰关系。

build_dict 对输入的 keys 进行统计，并返回一个映射词到 id 的字典。

load_and_preprocess data 功能是读取训练集、验证集、测试集的数据并进行预处理。先调用 read_conll 函数把数据从.conll 文件读取成一批字典对象，再调用 parser.vectorize 函数把字典中的字符转成对应的 parser 中的 token 编号从而把数据向量化，最后单独对训练数据调用 parser.create_instances 函数生成训练所需的 groundtruth action 序列，并生成每个状态对应的 feature。另外，如果是 debug 模式就会限制数据量方便 debug。

2 自己实现代码的解释

由于数据集中有一部分数据是 non-projective 的，不能直接用课堂上讲到的对 projective 的句子的 action。比如下面这个例子，用 (x,y) 表示序号为 x 的单词的 head 是 y，0 表示 root 那么对于句子序列 (1,3), (2,3), (3,6), (4,3), (5,4), (6,0), ... 如果按照 projective 的 action，那么当 buffer 顶端是单词 4 时，stack 顶端是单词 3，于是采取 Right-Arc，而后续的单词 5 的 head 是单词 4，但这时候单词 4 已经不在 stack 与 buffer 里面了，从而导致无法生成正确的动作序列。

所以我对 action 的定义进行了更改，使其能够在 non-projective 的句子上也能得出正确的动作序列。同课程 PPT 一致，记 σ 为 stack，β 为 buffer，A 为 dependencies ，以下是新的定义：

1. $Shift$:
$$(\sigma, i|\beta, A) \Rightarrow (\sigma|i, \beta, A)$$
2. $Left-Arc_k$:
$$(\sigma|i_2|i_1, j|\beta, A) \Rightarrow (\sigma|i_1, j|\beta, A \bigcup \{(i_1, i_2, k)\})$$

3. $Right - Arc_k$:

$$(\sigma|i_2|i_1, j|\beta, A) \Rightarrow (\sigma|i_2, j|\beta, A \bigcup \{(i_2, i_1, k)\})$$

下面几个部分提到的 action 全部是按照这个定义，不再采用课程 PPT 上的标准定义。

2.1 parser_transitions.py

PartialParse 初始化时，把 stack 初始化为只含有 <ROOT> 的 list，buff 初始化为句子每个单词依次组成的列表，dependencies 初始化为空列表即可。

在进行 parse_step 时按照上面定义的操作进行即可，不过这里为了便于后续处理，先把 dependencies 的每一个元素设置为 (head, id, label) 的格式，如果直接设置成 (head, label) 在我的实现中后续难以进行排序。在我的实现中 PartialParse 的 dependencies 列表的顺序并不是与单词的顺序一一对应的，我需要记录下单词的顺序 id 才能在 parse 结束的时候重新排序成对应单词的顺序。

minibatch_parse 函数中，先分别把句子初始化成 PartialParse，再分批次一步一步用 parsing_model 预测 action 并进行相应的 stack, arcs, buffer 维护。注意到在 Note 中提到有浅拷贝的问题，我在实际实现的时候用另一个变量存切片的指针，每次维护的时候如果句子 parse 完毕就移除这个指针，但并不会把原本的空间释放，从而避开了这个问题。

2.2 parser_utils.py

extract_feature 函数抽取的特征我采用论文 A Fast and Accurate Dependency Parser using Neural Networks 中设计的特征，feature 一共 48 维，由 Sw, St, Sl 三部分组成。Sw 有 18 个特征维度，包括 stack 和 buffer 的前三个元素，共 6 个；stack 顶端两个元素的最左侧的子元素，第二最左侧的子元素，最右侧的子元素，第二最右侧的子元素，共 8 个；stack 顶端两个元素的最左侧元素的的子元素的最左侧子元素，最右侧元素的子元素的子元素，共 4 个。St 是与 Sw 对应的单词在句子中的位置特征，也是 18 维。Sl 是与 Sw 对应的单词的依存关系，由于 Sw 前六维度是单个元素，没有对应依存关系，故只有 12 个特征维度。如果 stack 或者 buffer 中不存在那么多元素就直接设置成 <NULL> 对应的 id 即可。实现这个函数的时候我实现了两个辅助函数，分别计算某个元素的左、右子元素列表，具体实现仅仅是遍历当前的依存关系查找这个元素的子元素并判断是左还是右子元素，然后根据位置进行排序。

get_oracle 函数是反向模拟，根据标注的单词序列的 head 与 label 得到 action。同样，这里反向模拟的过程与得出的 action 与前面给的的定义一致，与课程 PPT 不同。具体而言，如果当前 stack 中元素数量小于 2 并且 buffer 为空说明只有 <ROOT> 直接返回 None, 非空则 $Shift$ 的 id；如果 stack 顶端第二个元素的 head 是 stack 顶端的第一个元素就返回 $LeftArc_k$ 的 id；如果 stack 顶端第一个元素的 head 是 stack 顶端的第二个元素并且后面没有任何元素的 head 是 stack 顶端的第一个元素，那么就返回 $RightArc_k$ 的 id；如果以上情况都不成立，buffer 里面还有元素就返回 $Shift$ 的 id，没有元素时表明此时 parse 结束，返回 None。

create_instances 函数需要填充的部分其实和 parser_transitions.py 中的 parse_step 类似，都是用给定 action 更新 stack, arcs, buffer, 具体操作类似 parse_step。

2.3 parsing_model.py

这部分主要是根据提取的状态 feature 预测下一步要进行的 action 是什么，我把这个问题当做一个多分类问题进行处理。模型的结构如下，输入提取的 feature，先经过一层 word_embedding，再随机 dropout 一半，然后进入线性的 hidden layer，最后通过一层全连接层输出预测的每个类别的分数。在采用 UAS 的时候就只有 3 个类别，改为 LAS 之后类别的数目由 parser 的 label 数决定。

2.4 trainer.py, main.py, predict.py

训练时我保存验证集上 LAS 得分最高的模型, 且我修改了模型保存的代码, 直接保存一整个 parser, 这样在后续 predict.py 中就可以不用再加载训练数据去构建 parser。而 predict.py 主要参考了框架代码，读取测试文件并向量化，用模型预测并 parse，调用函数计算 UAS 和 LAS，最后以固定格式保存预测结果到 predict.txt 中。

在一个 epoch 中，先把数据随机打乱，再每次取一个 batch 用模型预测，将得到的预测结果 y_predict 和 groundtruth action 序列 y 输入损失函数并反向传播。介绍模型时提到过我把这个当做一个多分类任务，所以我采用的 loss 是 CrossEntropyLoss。与论文保持一致，我也加上了 l2 正则项。激活函数我最初也采用论文中推荐的 Cube activation function x^3，但后续尝试了一些其他的激活函数，包括 sigmoid, tanh, relu 等，我发现 relu 的效果比论文推荐的 cube 效果更好，所以最后还是采用了 relu 而不是原论文中的 cube。

在介绍 parser_transitions.py 中我提到过我把 PartialParse.dependencies 的结构修改

了，所以在验证集合测试集上 parse 之后对 dependencies 我进行了额外的操作以适应 evaluate 函数的要求，对于 gold_dependencies 我则写了一个辅助函数 get_gold_dependencies 获取，具体而言是遍历所有句子字典，把每一个词的 head 与 label 结合成元组后作为句子的 dependencies。

另外，论文 A Fast and Accurate Dependency Parser using Neural Networks 中采用的优化器是 Adagrad, 但我实验中发现 Adam 的效果更好，我尝试了原论文推荐的超参数设置，仍然比不过 Adam，所以最后我采用了 Adam 作为 optimizer。

3 实验结果与 parsing 展示

我尝试了一些不同的超参数组合，发现如下的超参数获得的效果最好 (综合考虑了训练时间和模型表现)：

- batch_size = 1024
- lr = 0.0001
- embedding_size = 128
- hidden_size = 512
- dropout = 0.5

最后模型在验证集上 UAS = 89.37, LAS = 87.13; 在测试集上 UAS = 89.33, LAS = 87.25。下面是在测试集上的 parsing 结果展示 (测试集上所有句子的 parsing 结果会保存到 prediction.txt 中)。大部分句子都能对绝大部分单词的 head 与 label 预测正确，也有的句子可以全部预测正确，但是也存在少数句子会预测错误。为了便于对比预测结果与标注，我在 predict.py 中写了一个辅助函数将预测结果与标注一起写入 prediction.txt，每一个单词独立一行形成 (word id, head, label) 的格式，句子与句子之间有空行分隔，head 与 label 的前者表示预测值，后者为标注值，比如 head：h_pred // h_gold 表示预测值为 h_pred、标注为 h_gold。另外，为了对其数据，我统一规定保留两位整数，不足则补 0 占位。在上述规定下，句子标注结果展示如下图（选取一正一反两个示例）。

```
118
119   word_id: 01     head: 03 // 03  label:25 // 25
120   word_id: 02     head: 03 // 03  label:15 // 15
121   word_id: 03     head: 04 // 04  label:36 // 36
122   word_id: 04     head: 25 // 25  label:38 // 38
123   word_id: 05     head: 06 // 06  label:20 // 20
124   word_id: 06     head: 04 // 04  label:17 // 17
125   word_id: 07     head: 06 // 06  label:33 // 33
126   word_id: 08     head: 10 // 10  label:28 // 28
127   word_id: 09     head: 10 // 10  label:16 // 16
128   word_id: 10     head: 07 // 07  label:26 // 26
129   word_id: 11     head: 12 // 12  label:20 // 20
130   word_id: 12     head: 06 // 06  label:17 // 17
131   word_id: 13     head: 16 // 16  label:16 // 16
132   word_id: 14     head: 16 // 16  label:25 // 25
133   word_id: 15     head: 16 // 16  label:15 // 15
134   word_id: 16     head: 12 // 12  label:35 // 35
135   word_id: 17     head: 18 // 18  label:20 // 20
136   word_id: 18     head: 12 // 12  label:30 // 30
137   word_id: 19     head: 20 // 20  label:25 // 25
138   word_id: 20     head: 18 // 18  label:35 // 35
139   word_id: 21     head: 22 // 22  label:28 // 28
140   word_id: 22     head: 20 // 20  label:26 // 26
141   word_id: 23     head: 25 // 25  label:08 // 08
142   word_id: 24     head: 25 // 25  label:36 // 36
143   word_id: 25     head: 00 // 00  label:00 // 00
144   word_id: 26     head: 25 // 25  label:08 // 08
145
```

图 1: **succeed**

prediction.txt

```
1   word_id: 01     head: 06 // 07  label:19 // 19
2   word_id: 02     head: 06 // 07  label:08 // 08
3   word_id: 03     head: 06 // 07  label:36 // 36
4   word_id: 04     head: 06 // 07  label:31 // 31
5   word_id: 05     head: 06 // 07  label:01 // 01
6   word_id: 06     head: 00 // 07  label:00 // 15
7   word_id: 07     head: 06 // 00  label:06 // 00
8   word_id: 08     head: 06 // 07  label:08 // 08
9
```

图 2: **faliure case**