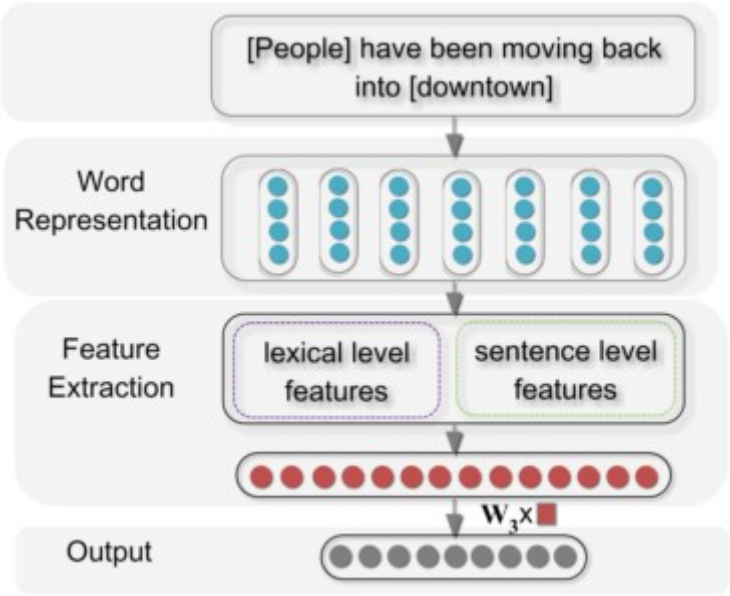
华东师范大学计算机科学技术系上机实践报告

课程名称：人工智能	**年级：2018级**	**上机实践成绩：**
指导教师：周爱民	**姓名： 汪子凡**	**创新实践成绩：**
上机实践名称：知识图谱实验	**学号：10185102153**	**上机实践日期：2020/7/2**
上机实践编号：No.5	**组号：**	**上机实践时间：**

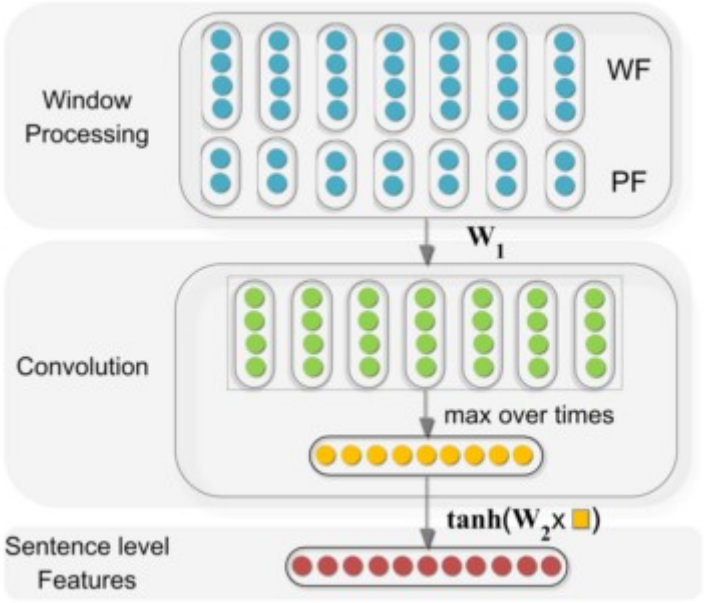
一、 问题介绍

本次实验的内容为关系抽取，所用模型为论文 Relation Classification via Convolutional Deep Neural Network.(ACL 2014)所提出的模型，学习框架为 Pytorch，所用数据集为 SemEval 2010 Task8。数据集内部共有 19 种关系，8000 条训练数据与 2717 条测试数据。

关系抽取是自然语言处理的一个经典任务，目标为给定标注了两个实体的句子，返回这条句子中所能显示出来的两个实体之间的语义关系。



模型的整体框架如上图所示，将带有实体标注的句子进行词向量化后进行特征抽取，得到词汇级别的特征和句子级别的特征，再将两个级别的特征向量拼接后经过全连接层得到维度与关系标签数相同的向量，再对得到的向量进行 soft-max 后输出。



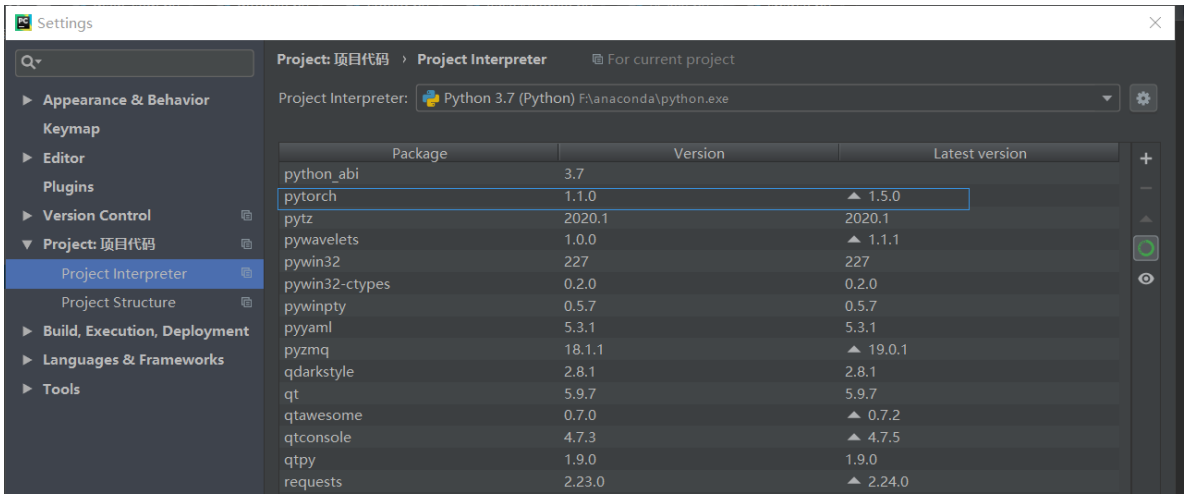
其中词汇级别的特征结构如上图所示，由实体 1，实体 2，实体 1 左右的单词，实体 2左右的单词，以及两个实体在语义网中的上位词组成，本次代码实验提供的数据中没有 L5即两个实体在语义网中的上位词；语句级别的特征提取过程如图 2 所示，首先拼接单词与位置信息的向量化表示，然后通过卷积层与使用最大池化的池化层，再经过使用 tanh 激活函数的全连接层得到。

模型的损失函数为交叉熵损失函数，使用 Adam 方法进行参数优化。(Zeng, . (2014). Relation Classification via Convolutional Deep Neural Network. Coling, 2335–2344)

二、 程序设计与算法分析

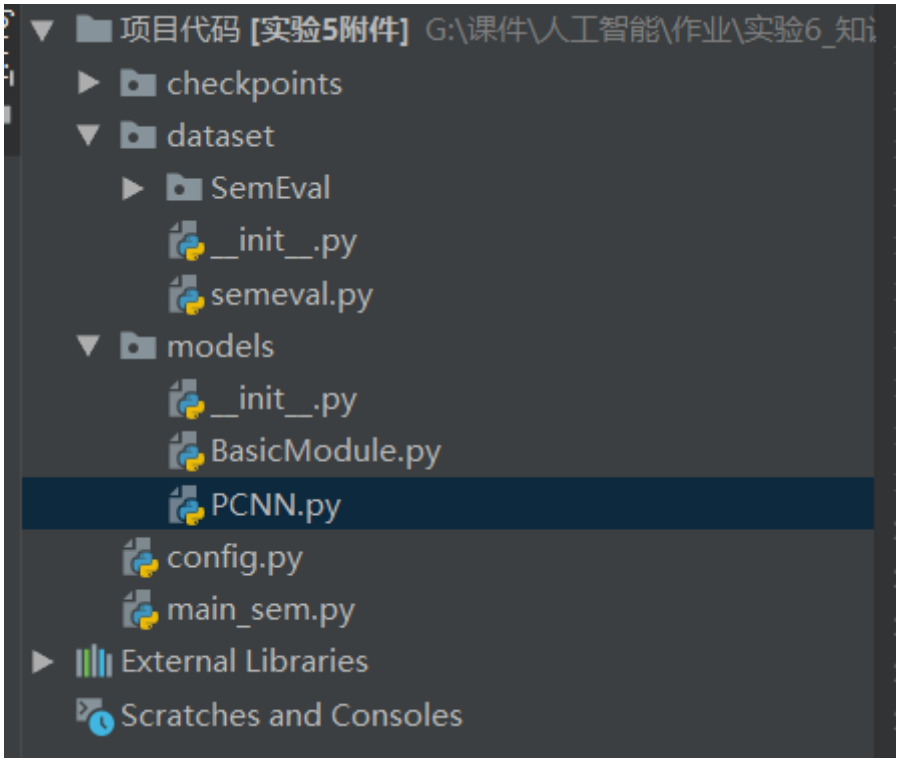
1.环境配置

前几次实验都在 jupyter notebook 上进行，用一个 ipynb 文件即可，而这次知识图谱的实验较为复杂，有多个文件，用 pycharm 比较方便。



之前在 anaconda 中已经下载好了 pytorch， 可以通过改变 pycharm 解释器的方法使用 anaconda 配好的环境，这样就不用重复下载 pytorch包了。

2.任务分析



其中，dataset 是实验的数据，已经预处理好，生成了 npy文件。models 是实验的模型，其中我们所用到的 PCNN 模型是 BasicModule模型的子类，在BasicModule中提供了save和load两个方法，而在 PCNN 中需要**补充模型/models/PCNN.py 中的网络结构与前向函数。**其中算法的损失函数和优化方式在中附件中给出，不需要实现。

config.py 是一些参数的配置，main_sem.py 是我们要进行训练的主函数，其中只有 train()，没有定义test(), 实验过程中会自动保存最优模型，我们需要**搭建并训练模型，并得到模型在测试集上的性能。**

3.知识了解

① Lexical-Feature

e.g. The [haft] of the [axe] is made of yew wood.

特征	含义	例子说明
L1	entity1	haft
L2	entity2	axe
L3	entity1的左右两个tokens	the, of
L4	entity2的左右两个tokens	the, is
L5	WordNet中两个entity的上位词	在wordNet中找到两个entity的上位词。

这5个特征都是lexical-level的，之后在使用它们的word embedding串联起来 作为最后的lexical-feature。

②word embedding

Embedding在数学上表示一个maping, f: X -> Y， 也就是一个function，其中该函数是injective（就是我们所说的单射函数，每个Y只有唯一的X对应，反之亦然）和structure-preserving (结构保存，比如在X所属的空间上X1 < X2,那么映射后在Y所属空间上同理 Y1 < Y2)。那么对于word embedding，就是将单词word映射到另外一个空间，其中这个映射具有injective和structure-preserving的特点。

Word Embedding，就是找到一个映射或者函数，生成在一个新的空间上的表达，该表达就是word representation。

③CNN中的 Sentence-level feature

如问题介绍中的整体框架图所示，在输入层的句子中每个word 有两种特征：

1) WF(word feature): 很简单就是词向量 word embedding

2) PF(position feature): 位置特征， 每个word与两个entity的相对距离。

这样每个word的feature维度为: $d_w + d_p * 2$，其中 d_w 为 word embedding 的维度，d_p为 position embedding的维度。

4.补全代码

在PCNN.py中，我们需要对模型的构建进行补全。

```
self.word_embs = nn.Embedding(self.opt.vocab_size, self.opt.word_dim)     #单词的向量化表示，从已有的 word_embedding 中 load 出来
self.pos1_embs = nn.Embedding(self.opt.pos_size + 1, self.opt.pos_dim)    #与句子中第一个实体的相对位置所对应的向量化表示
self.pos2_embs = nn.Embedding(self.opt.pos_size + 1, self.opt.pos_dim)    #与句子中第二个实体的相对位置所对应的向量化表示

feature_dim = self.opt.word_dim + self.opt.pos_dim * 2              #设置每个单词的feature维度
```

首先根据结构图，我们需要获得 word feature 和 position feature, 并且定义每个单词的 feature 维度为 $d_w + d_p * 2$。

```
self.convs = nn.ModuleList(                                         #设置卷积层设置sentence-level feature
    [nn.Conv2d(1, self.opt.filters_num, (k, feature_dim), padding=(int(k / 2), 0)) for k in self.opt.filters])
all_filter_num = self.opt.filters_num * len(self.opt.filters)
self.cnn_linear = nn.Linear(all_filter_num, self.opt.sen_feature_dim)

self.init_word_emb()
```

处理完输入之后，接下来是卷积层操作，卷积之后使用了Max Pooling操作，这样可以提取每一个卷积核的最有用的特征，在之后又接了一个全连接层最终得到了sentence-level feature。

```
#将lexical-feature 与 sentence-level feature 直接串起来
self.out_linear = nn.Linear(all_filter_num + self.opt.word_dim * 6, self.opt.rel_num)
self.dropout = nn.Dropout(self.opt.drop_out)
self.init_word_emb()
self.init_model_weight()
```

最终将上面的lexical-feature 与 sentence-level feature 串起来，作为整个句子的特征去做分类，然后加一个全连接层和softmax ，这样就完成了模型的构建。

而同时对于**向前传播函数**也需要进行补充，其中输入为长度为 4 的列表，其中 lexical feature 为(实体 1 的单词,实体 1 左边的单词,实体 1 右边的单词,实体 2,实体 2 左边的单词,实体 2 右边的单词)六元组。

一开始进行补全时，发生如下报错：

```
F:\anaconda\python.exe G:/课件/人工智能/作业/实验6_知识图谱/实验5附件/main_sem.py
**************************************************
user config:
**************************************************
loading train data
loading finish
loading test data
loading finish
train data: 8000; test data: 2717
Traceback (most recent call last):
  File "G:/课件/人工智能/作业/实验6_知识图谱/实验5附件/main_sem.py", line 131, in <module>
    train()
  File "G:/课件/人工智能/作业/实验6_知识图谱/实验5附件/main_sem.py", line 71, in train
    out = model(data[:-1])
  File "F:\anaconda\lib\site-packages\torch\nn\modules\module.py", line 493, in __call__
    result = self.forward(*input, **kwargs)
  File "G:\课件\人工智能\作业\实验6_知识图谱\实验5附件\models\PCNN.py", line 70, in forward
    lexical_level_emb = self.word_embs(lexical_feature)  # (batch_size, 6, word_dim
  File "F:\anaconda\lib\site-packages\torch\nn\modules\module.py", line 493, in __call__
    result = self.forward(*input, **kwargs)
  File "F:\anaconda\lib\site-packages\torch\nn\modules\sparse.py", line 117, in forward
    self.norm_type, self.scale_grad_by_freq, self.sparse)
  File "F:\anaconda\lib\site-packages\torch\nn\functional.py", line 1506, in embedding
    return torch.embedding(weight, input, padding_idx, scale_grad_by_freq, sparse)
RuntimeError: Expected tensor for argument #1 'indices' to have scalar type Long; but got CPUType instead (while checking arguments for embedding)

Process finished with exit code 1
```

所以去学习了 pytorch 的 tensor 类型，Tensor有不同的数据类型，每种类型分别有对应CPU和GPU版本（HalfTensor除外）。默认的Tensor是FloatTensor，可通过torch.set_default_tensor_type修改默认tensor类型（如果默认类型为GPU tensor，则所有操作都将在GPU上进行）。

数据类型	CPU Tensor	GPU Tensor
32 bit 浮点	**torch.FloatTensor**	**torch.cuda.FloatTensor**
64 bit 浮点	**torch.DoubleTensor**	**torch.cuda.DoubleTensor**
16 bit 半精度浮点	N/A	torch.cuda.HalfTensor
8 bit 无符号整形(0~255)	torch.ByteTensor	torch.cuda.ByteTensor
8 bit 有符号整形(-128~127)	torch.CharTensor	torch.cuda.CharTensor
16 bit 有符号整形	torch.ShortTensor	torch.cuda.ShortTensor
32 bit 有符号整形	**torch.IntTensor**	**torch.cuda.IntTensor**
64 bit 有符号整形	**torch.LongTensor**	**torch.cuda LongTensor**

这个报错是数据类型的报错，我传入的类型是 CPU 类型，但是希望得到张量long类型，所以在这里需要对类型进行一次转换，可以如下使用torch的转换方法：torch.Tensor,long()

```
    def forward(self, x):
        lexical_feature, word_feautre, left_pf, right_pf = x

        #处理输入，得到lexical_level_feature
        batch_size = lexical_feature.size(0)
        lexical_level_emb = self.word_embs(torch.Tensor.long(lexical_feature))  # (batch_size, 6, word_dim
        lexical_level_emb = lexical_level_emb.view(batch_size, -1)
        # lexical_level_emb = lexical_level_emb.sum(1)

        # 得到sentence level feature
        word_emb = self.word_embs(torch.Tensor.long(word_feautre))
        left_emb = self.pos1_embs(torch.Tensor.long(left_pf))
        right_emb = self.pos2_embs(torch.Tensor.long(right_pf))

        sentence_feature = torch.cat([word_emb, left_emb, right_emb], 2)
```

根据pdf的详细说明，需要将输入的信息向量化后得到 Lexical Level 与 Sentence Level 的向量，并对 Sentence Level 的特征进行二维卷积操作，得到的结果再经过一个 max pooling 层和一个全连接层。

```
# 后将处理过后的 Sentence Level Feature 与 Lexical Level Feature 拼接
x = sentence_feature.unsqueeze(1)
x = self.dropout(x)
x = [F.relu(conv(x)).squeeze(3) for conv in self.convs]
x = [F.max_pool1d(i, i.size(2)).squeeze(2) for i in x]
x = torch.cat(x, 1)


sen_level_emb = x

x = torch.cat([lexical_level_emb, sen_level_emb], 1)
x = self.dropout(x)
x = self.out_linear(x)
```

最后需要将处理过后的 Sentence Level Feature 与 Lexical Level Feature 拼接，经过全连接层得到维度为标签数的向量后输出。

这样模型就补全就大致完毕了，由于实验数据已经预处理好，只需要在 main_sem.py 运行主函数就可以得到这个模型的预测结果。

三、 实验结果

运行主函数后，过了一段时间，可以得到实验结果：

```
**************************************************
loading train data
loading finish
loading test data
loading finish
train data: 8000; test data: 2717
Epoch 0/25: train loss: 0.017565082162618637; test accuracy: 0.6128082443871917,  test loss 0.010183754281432566
Epoch 1/25: train loss: 0.011080084770917892; test accuracy: 0.6676481413323518,  test loss 0.008765329123859595
Epoch 2/25: train loss: 0.009606491260230542; test accuracy: 0.6834744203165256,  test loss 0.00817973819075589
Epoch 3/25: train loss: 0.008575116448104382; test accuracy: 0.6985645933014354,  test loss 0.007786699349624864
Epoch 4/25: train loss: 0.007620965868234635; test accuracy: 0.7070298122929702,  test loss 0.007562826431802322
Epoch 5/25: train loss: 0.007199554949998856; test accuracy: 0.715126978284873,  test loss 0.007476955290265411
Epoch 6/25: train loss: 0.006678555339574814; test accuracy: 0.7254324622745676,  test loss 0.007139380512581532
Epoch 7/25: train loss: 0.006180961139500141; test accuracy: 0.7302171512697828,  test loss 0.0070343117833269134
Epoch 8/25: train loss: 0.005731755018234253; test accuracy: 0.7261685682738315,  test loss 0.007244545350760978
215.9666497707367s
Epoch 9/25: train loss: 0.005197936721146107; test accuracy: 0.7331615752668385,  test loss 0.007066462072415259
Epoch 10/25: train loss: 0.004882389650453185; test accuracy: 0.7320574162679426,  test loss 0.007180674597319321
Epoch 11/25: train loss: 0.004378278890633581; test accuracy: 0.7379462642620538,  test loss 0.00704371828255427
Epoch 12/25: train loss: 0.004230103529989719 5; test accuracy: 0.7471475892528524,  test loss 0.006945777211419177
Epoch 13/25: train loss: 0.003827673900872469; test accuracy: 0.7486198012513802,  test loss 0.006846900542606358
Epoch 14/25: train loss: 0.0035226139687001704; test accuracy: 0.7548767022451233,  test loss 0.00691322471712019
Epoch 15/25: train loss: 0.0034034106284379966; test accuracy: 0.7519322782480677,  test loss 0.006924415911476885
Epoch 16/25: train loss: 0.0031703849285840 99; test accuracy: 0.7563489142436511,  test loss 0.007017214047281365
Epoch 17/25: train loss: 0.0027593248318880798; test accuracy: 0.7523003312476997,  test loss 0.007001842564416 8
Epoch 18/25: train loss: 0.0025828842986375094; test accuracy: 0.7515642252484358,  test loss 0.007102240062084627
421.56155920028687s
Epoch 19/25: train loss: 0.002566213324666023; test accuracy: 0.7534044902465955,  test loss 0.007125678047521 3504
Epoch 20/25: train loss: 0.002267579447 8056; test accuracy: 0.7556128082443871,  test loss 0.00724194799664 7076
Epoch 21/25: train loss: 0.002102863138541579; test accuracy: 0.7589252852410747,  test loss 0.007273118483656068
Epoch 22/25: train loss: 0.0020151988230645655; test accuracy: 0.7559808612440191,  test loss 0.007458449221727907
Epoch 23/25: train loss: 0.0019282964803278447; test accuracy: 0.7537725432462274,  test loss 0.00762450550249231
Epoch 24/25: train loss: 0.0017417015470564366; test accuracy: 0.7478836952521163,  test loss 0.00778182341765515
**************************************
the best acc: 0.7589252852410747;
```

随着不断地训练，准确率从60%左右上升到75%左右，其中最好情况为75.9%，这个最好模型保存在同一文件夹的 checkpoints 中。实验结果尚可，可能是实验的结构还是比较简单，但是使用了 Position Feature, 可能位置信息能增加一定的正确率。