Web Information Processing @USTC2021Autumn

LAB@2 : Prediction of Tail Entity 实验报告

成员：PB18051081李钊佚、PB18051061黄育庆

简介：本次实验，我们先后共计尝试了三种实现思路（当然，最后的结果并不是都尽如人意）：

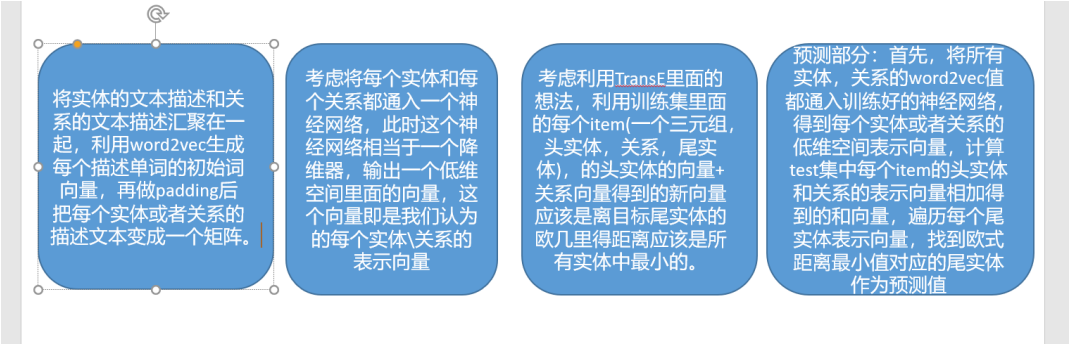
@第一种是基于对实体和关系的文本描述，做词向量嵌入后，将训练集中头实体，关系，尾实体都通过一个神经网络映射到一个新的低维向量空间中，在这个向量空间中，去计算vector(head)+vector(relation)-vector(tail)的二范数，取最小的值去预测尾实体。（我们独立设计的，但是预测指标很差，不work（Hit@5约为1%））

@第二种是基于对实体和关系的文本描述，做词向量嵌入后，利用自己手作的神经网络预测尾实体在词向量空间中的词向量表示（本质上将问题转化成Regression任务）。（我们独立设计的，预测结果还比较满意，最终提交的结果是这种方法预测的结果（Hit@5约为22.5%））

@第三种是基于课程介绍的TransE方法，基于训练集提供的在某个向量空间中的不同实体，关系之间的结构信息来不断缩短vec(head)+vec(relation)与vec(tail)之间的差距。（参考github上面代码（Hit@5约为5%））

下面，我们将更为详细地介绍一下这三种具体实现细节，我们的设计和预测效果。

- **@1基于文本信息的神经网络降维学习方法：**
  - **思维导图**



- **自己认为设计中比较好，或者是需要结合代码说明的地方**

send2.py > ...

```
194    count2 = 0
195    entdict = {}
196    for key in entity_dict.keys():
197        count1 += 1
198        print(count1)
199        key_dscrip = entity_dict[key]
200        key_vecs = sen2vecs(key_dscrip, entity_dict)
201        key_vecs = pad2Dvecs(key_vecs).tolist()
202        key_ten = torch.tensor(key_vecs).unsqueeze(0).transpose(1,2)
203        key_vec = mynet(key_ten)
204        entdict[key] = key_vec
205        del key_dscrip,key_vecs,key_ten,key_vec
206    reldict = {}
207    for key in relat_dict.keys():
208        count2 += 1
209        print(count2)
210        key_dscrip = relat_dict[key]
211        key_vecs = sen2vecs(key_dscrip, relat_dict)
212        key_vecs = pad2Dvecs(key_vecs).tolist()
213        key_ten = torch.tensor(key_vecs).unsqueeze(0).transpose(1,2)
214        key_vec = mynet(key_ten)
215        reldict[key] = key_vec
216        del key_dscrip,key_vecs,key_ten
```

我觉得这个地方设计的不错，先遍历一遍所有的实体和关系，把它们都送到train好的网络里面得到表示向量，这样我们在预测的时候就可以快速预测，而且这个表示向量的表可以存起来，以便下次使用。

- **预测结果分析**

很遗憾，这个方法的Hit@5只有1%左右，这表明它几乎不work，我们后来摒弃了这种方法，这部分的代码在提交代码的上方注释掉了。

- **@2基于文本信息和神经网络的尾实体预测Regression方法：**
  - **思维导图**

网络结构定义：

```
class TextDNN(nn.Module):
    def __init__(self, embedding_dim, hidden_dim0, hidden_dim1, hidden_dim2, hidden_dim3):
        super().__init__()
        self.fc1 = nn.Linear(2*embedding_dim, hidden_dim0)
        self.cnn = nn.Conv1d(1,1,3)
        self.avg = nn.AdaptiveAvgPool1d(hidden_dim1)
        self.fc2 = nn.Linear(hidden_dim1, hidden_dim2)
        self.fc3 = nn.Linear(hidden_dim2, hidden_dim3)
        self.fc4 = nn.Linear(hidden_dim3, embedding_dim)

    def forward(self,x):
        #x = F.dropout(self.fc1(x),0.5)
        x = self.fc1(x)
        x = x.unsqueeze(1)
        x = self.avg(self.cnn(x))
        x = x.squeeze(1)
        x = F.relu(x)
        x = self.fc2(x)
        #x = F.dropout(self.fc2(x),0.5)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.fc4(x)
        return x

model2 = TextDNN(100,300,200,300,400)
optimizer2 = torch.optim.SGD(model2.parameters(),lr=0.001)
```

- 自己认为设计中比较好，或者是有所迷惑，或者是需要结合代码说明的地方

  我觉得，这个方案里面儿比较好的地方是

  @1：由于朴素的Regression任务，是一个输入对应一个输出，但是我们这里是Triple（三元组）的关系，所以我们的输入应该把head entity和relationship结合起来作为一个输入，输出是tail entity。这里结合的方法有很多种，经过我们的分析和实验，觉得还是将head entity的表示向量和relationship的表示向量结合（cat）起来效果比较好，也更能反映这两者对尾实体的影响作用。

```
def train(model,optimizer):
    for head_batch,rela_batch,tail_batch in train_loader:
        '''
        head_batch:size = (batch_size, embedding_dim)
        '''
        head_rela_batch = torch.cat((head_batch,rela_batch),1)

        optimizer.zero_grad()

        pred_vecs = model(head_rela_batch)
        loss = torch.norm(pred_vecs - tail_batch)
        loss.backward()
        optimizer.step()
        print("loss = ",loss, "\n")
```

@2：我觉得我们的网络设计的虽然有点炼丹嫌疑，但是还是比较合情合理，为了增强网络模型的泛化性，我们设计了4层全连接层（深度），每层有上百个perceptrons，为了引入非线性因素，我们在其中三层中增加了relu激活函数，和在一开始引入了一个单channel卷积层，其实还尝试过引入drop_out，但是由于训练太花时间，故放弃drop_out层。

```python
class TextDNN(nn.Module):
    def __init__(self, embedding_dim, hidden_dim0, hidden_dim1, hidden_dim2, hidden_dim3):
        super().__init__()
        self.fc1 = nn.Linear(2*embedding_dim, hidden_dim0)
        self.cnn = nn.Conv1d(1,1,3)
        self.avg = nn.AdaptiveAvgPool1d(hidden_dim1)
        self.fc2 = nn.Linear(hidden_dim1, hidden_dim2)
        self.fc3 = nn.Linear(hidden_dim2, hidden_dim3)
        self.fc4 = nn.Linear(hidden_dim3, embedding_dim)

    def forward(self,x):
        #x = F.dropout(self.fc1(x),0.5)
        x = self.fc1(x)
        x = x.unsqueeze(1)
        x = self.avg(self.cnn(x))
        x = x.squeeze(1)
        x = F.relu(x)
        x = self.fc2(x)
        #x = F.dropout(self.fc2(x),0.5)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.fc4(x)
        return x

model2 = TextDNN(100,300,200,300,400)
optimizer2 = torch.optim.SGD(model2.parameters(),lr=0.001)
```

@3：在网络训练，或者利用训练好的网络进行forward计算时，我们都可以利用cuda来加速torch的矩阵计算，但是由于我们的算法，必须计算回归后的向量同所有尾实体词向量的norm范数，这就使得预测时间巨长无比，我们两个人用四台电脑预测了几乎一个白天才完成text集合中统共20000多条item的预测。目前还没有想到这部分预测应该如何加速orz

```python
def predict(model,head,relation):
    cur_min5 = [pos_inf, pos_inf, pos_inf, pos_inf, pos_inf]
    sel_key5 = [0, 0, 0, 0, 0]

    head_vec = entity_dict2[head]
    rela_vec = relat_dict2[relation]

    for tail in entity_dict2.keys():
        if (head, relation) in train_filter.keys() and tail in train_filter[(head, relation)]:
            continue
        key = tail
        tail_vec = torch.tensor(entity_dict2[key])
        head_rela_vec = torch.cat((torch.tensor(head_vec),torch.tensor(rela_vec)),0).unsqueeze(0)
        dis = torch.norm(model2(head_rela_vec).squeeze(0) - tail_vec)
        if dis < cur_min5[4]:
            if dis >= cur_min5[3]:
                cur_min5[4] = dis
                sel_key5[4] = key
            else:
                cur_min5[4] = cur_min5[3]
                sel_key5[4] = sel_key5[3]
                if dis >= cur_min5[2]:
                    cur_min5[3] = dis
                    sel_key5[3] = key
                else:
                    cur_min5[3] = cur_min5[2]
                    sel_key5[3] = sel_key5[2]
                    if dis >= cur_min5[1]:
                        cur_min5[2] = dis
                        sel_key5[2] = key
                    else:
                        cur_min5[2] = cur_min5[1]
                        sel_key5[2] = sel_key5[1]
                        if dis >= cur_min5[0]:
                            cur_min5[1] = dis
                            sel_key5[1] = key
```

- **预测结果截图分析**

在test集上的预测结果：

提交时间	文件名称	Hit@1	Hit@5
2021-12-19 21:08:44	59_1639919324_result.txt	0.153523	0.224568

可见，这种基于文本信息和神经网络的回归预测方法可以使Hit@1达到15.4%左右，Hit@5达到22.5%左右，这说明这种方法还是比较有效的。

- **@3基于结构信息的TransE方法：**
  - **算法**

Algorithm 1 Learning TransE

input Training set $S = \{(h, \ell, t)\}$, entities and rel. sets E and L, margin γ, embeddings dim. k.

initialize $\boldsymbol{\ell} \leftarrow \text{uniform}(-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each $\ell \in L$

$\boldsymbol{\ell} \leftarrow \boldsymbol{\ell} / \|\boldsymbol{\ell}\|$ for each $\ell \in L$

$\mathbf{e} \leftarrow \text{uniform}(-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each entity $e \in E$

loop

  $\mathbf{e} \leftarrow \mathbf{e} / \|\mathbf{e}\|$ for each entity $e \in E$

  $S_{batch} \leftarrow$ sample(S, b) // sample a minibatch of size b

  $T_{batch} \leftarrow \emptyset$ // initialize the set of pairs of triplets

  for $(h, \ell, t) \in S_{batch}$ **do**

    $(h', \ell, t') \leftarrow$ sample$(S'_{(h,\ell,t)})$ // sample a corrupted triplet

    $T_{batch} \leftarrow T_{batch} \cup \{((h, \ell, t), (h', \ell, t'))\}$

  end for

  Update embeddings w.r.t. $\sum_{((h,\ell,t),(h',\ell,t')) \in T_{batch}} \nabla \left[\gamma + d(\boldsymbol{h} + \boldsymbol{\ell}, \boldsymbol{t}) - d(\boldsymbol{h'} + \boldsymbol{\ell}, \boldsymbol{t'})\right]_+$

end loop

这部分的代码我们修改自助教给出的lab2_instruction:

3.1初始化

根据维度，为每个实体和关系初始化向量，并归一化

```
    def emb_initialize(self):
        relation_dict = {}
        entity_dict = {}

        for relation in self.relation:
            r_emb_temp = np.random.uniform(-6 / math.sqrt(self.embedding_dim),
                                           6 / math.sqrt(self.embedding_dim),
                                           self.embedding_dim)
            relation_dict[relation] = r_emb_temp / np.linalg.norm(r_emb_temp,
ord=2)

        for entity in self.entity:
            e_emb_temp = np.random.uniform(-6 / math.sqrt(self.embedding_dim),
                                           6 / math.sqrt(self.embedding_dim),
                                           self.embedding_dim)
            entity_dict[entity] = e_emb_temp / np.linalg.norm(e_emb_temp, ord=2)
```

3.2 选取batch

设置`nbatches`为batch数目，`batch_size = len(self.triple_list) // nbatches`

从训练集中随机选择`batch_size`个三元组，并随机构成一个错误的三元组S'，进行更新

```
    def train(self, epochs):
        nbatches = 400
        batch_size = len(self.triple_list) // nbatches
        print("batch size: ", batch_size)
        for epoch in range(epochs):
            start = time.time()
            self.loss = 0

            # Sbatch:list
            Sbatch = random.sample(self.triple_list, batch_size)
            Tbatch = []

            for triple in Sbatch:
                corrupted_triple = self.Corrupt(triple)
                if (triple, corrupted_triple) not in Tbatch:
                    Tbatch.append((triple, corrupted_triple))
            self.update_embeddings(Tbatch)
```

3.3梯度下降：

```
139     def update_embeddings(self, Tbatch):
140         copy_entity = copy.deepcopy(self.entity)
141         copy_relation = copy.deepcopy(self.relation)
142 
143         for triple, corrupted_triple in Tbatch:
144             # 取copy里的vector累积更新
145             h_correct_update = copy_entity[triple[0]]
146             t_correct_update = copy_entity[triple[1]]
147             relation_update = copy_relation[triple[2]]
148 
149             h_corrupt_update = copy_entity[corrupted_triple[0]]
150             t_corrupt_update = copy_entity[corrupted_triple[1]]
151 
152             # 取原始的vector计算梯度
153             h_correct = self.entity[triple[0]]
154             t_correct = self.entity[triple[1]]
155             relation = self.relation[triple[2]]
156 
157             h_corrupt = self.entity[corrupted_triple[0]]
158             t_corrupt = self.entity[corrupted_triple[1]]
159 
160             if self.L1:
161                 dist_correct = distanceL1(h_correct, relation, t_correct)
162                 dist_corrupt = distanceL1(h_corrupt, relation, t_corrupt)
163             else:
164                 dist_correct = distanceL2(h_correct, relation, t_correct)
165                 dist_corrupt = distanceL2(h_corrupt, relation, t_corrupt)
166 
167             err = self.hinge_loss(dist_correct, dist_corrupt)
```

```
            if err > 0:
                self.loss += err

                grad_pos = 2 * (h_correct + relation - t_correct)
                grad_neg = 2 * (h_corrupt + relation - t_corrupt)

                #  梯度计算
                if self.L1:
                    for i in range(len(grad_pos)):
                        if (grad_pos[i] > 0):
                            grad_pos[i] = 1
                        else:
                            grad_pos[i] = -1

                    for i in range(len(grad_neg)):
                        if (grad_neg[i] > 0):
                            grad_neg[i] = 1
                        else:
                            grad_neg[i] = -1

                #  梯度下降
                # head系数为正，减梯度；tail系数为负，加梯度
                h_correct_update -= self.learning_rate * grad_pos
                t_correct_update -= (-1) * self.learning_rate * grad_pos

                # corrupt项整体为负，因此符号与correct相反
                if triple[0] == corrupted_triple[0]:  # 若替换的是尾实体，则头实体更新两次
                    h_correct_update -= (-1) * self.learning_rate * grad_neg
                    t_corrupt_update -= self.learning_rate * grad_neg

                elif triple[1] == corrupted_triple[1]:  # 若替换的是头实体，则尾实体更新两次
                    h_corrupt_update -= (-1) * self.learning_rate * grad_neg
                    t_correct_update -= self.learning_rate * grad_neg

                # relation更新两次
                relation_update -= self.learning_rate * grad_pos
                relation_update -= (-1) * self.learning_rate * grad_neg

        # batch norm
        for i in copy_entity.keys():
            copy_entity[i] /= np.linalg.norm(copy_entity[i])
        for i in copy_relation.keys():
            copy_relation[i] /= np.linalg.norm(copy_relation[i])

        # 达到批量更新的目的
        self.entity = copy_entity
        self.relation = copy_relation

    def hinge_loss(self, dist_correct, dist_corrupt):
        return max(0, dist_correct - dist_corrupt + self.margin)
```

- **预测结果截图分析**

2021-12-19 09:47:51	59_1639878471_result.txt	0.011629	0.053748

可见，这种基于结构信息，不顾文本信息的TransE方法的Hit@5能达到5.4%左右，说明有一定作用，但是效果不如我们前面提出的基于文本信息的回归模型好