【源码阅读系列】一：GraphSAGE代码阅读（1）

原创 TwT 小韬学算法 2020-04-01

0.前言

昨天发了一篇关于GraphSAGE论文的大致讲解，今天对源码进行部分解析，源码链接。作者最原始的训练代码是Tensorflow版本的，这是一个PyTorch版本的，恰好最近学习PyTorch，同时也有一段时间不用Tensorflow了，所以就对PyTorch版本的进行解析（其实主要是PyTorch的源码简单还少）。代码可能一次性看不完，毕竟能力有限~~，本文只放置部分关键代码。分析链接为：

> https://github.com/TwT520Ly/Code-Reading

1.数据集分析

「Cora数据集」代码只用了Cora数据集的一部分，Cora数据集中样本是机器学习论文，论文被分为7类：

- Case_Based
- Genetic_Algorithms
- Neural_Networks
- Probabilistic_Methods
- Reinforcement_Learning
- Rule_Learning
- Theory

数据集共有2708篇论文，分为两个文件：

- .content

- .cites

第一个文件形式为：

```
<paper_id> <word_attributes>+ <class_label>
```

分别表示论文的唯一ID，文档词的0-1编码向量，类别标签；文档词中0表示不存在，1表示存在。第二个文件形式为：

```
<ID of cited paper> <ID of citing paper>
```

分别表示被引用论文和引用论文，即后者引用前者，paper2->paper1。

2.代码分析

2.1 aggregators.py

实现聚合类，对邻居信息进行AGGREGATE。

```
# 如果num_sample设置了具体数字
ifnot num_sample isNone:
     _sample = random.sample
     # 首先对每一个节点的邻居集合neigh进行遍历，判断一下已有邻居数和采样数大小，多于采样数进行抽样
     samp_neighs = [_set(_sample(to_neigh,
                   num_sample,
                   )) if len(to_neigh) >= num_sample else to_neigh for to_neigh in to_neighs]
 else:
     samp_neighs = to_neighs
```

这里是对一个batch中的每一个节点的邻接点set进行sample，主要计算量在 `random.sample` ，简单分析一下 `random.sample` ，该函数如果指定采样数为K，内部会进行K次循环，分别获取K个元素。

```
if n <= setsize:
     # An n-length list is smaller than a k-length set
     pool = list(population)
     for i in range(k):         # invariant:  non-selected at [0,n-i)
         j = randbelow(n-i)
         result[i] = pool[j]
         pool[j] = pool[n-i-1]   # move non-selected item into vacancy
```

```
    else:
        selected = set()
        selected_add = selected.add
        for i in range(k):
            j = randbelow(n)
            while j in selected:
                j = randbelow(n)
            selected_add(j)
            result[i] = population[j]
    return result
```

此处通过调用 `randbelow` 函数实现，简单的考虑，如果我要抽取K个元素，那么是不是只要从原序列中生成K次随机下标就可以了？时间复杂度为 `O(K)`？事实上没有这么简单，如果sample出来的序列需要维持原有的次序，就需要每次 `randbelow` 的下标有序插入到已经sample的序列中，搜索代价大致为 `O(logN)`，那么时间复杂度就是 `O(NlogN)`，如果是这样子的话，那SAGE的sample时间复杂度就会提升到 `O(MNlogN)`。不过上面的代码中有明显的一个 `if-else` 结构，所以实现方式应该没有这么简单。首先看到判断条件为 `setsize`，此变量来源如下：

```
setsize = 21# size of a small set minus size of an empty list
if k > 5:
    setsize += 4 ** _ceil(_log(k * 3, 4)) # table size for big sets
```

这一堆看着就奇怪，莫名其妙的公式（暂时不管，其实和set的内存设定有关系，此处不做详细说明）~~。反正就是利用K值计算出一个setsize，然后判断和输入序列大小n的大小关系，如果n相对较小，就好像是10个中抽样9个，采用无放回抽样算法，那么每次抽样后原始序列缩小一个单位，为了不改变原始输入序列在内存中数值，将其拷贝至 `pool` 列表，并通过尾元素填充被选元素+缩小随机范围的方式从逻辑上压缩pool列表：

```
pool[j] = pool[n-i-1]
```

那么如果n较大，就会执行else部分代码，比如1千万数组中抽取3个元素，采用上述策略效率太低，所以采用放回抽样+多次重试的策略，如果随机到的下标已经在之前select到了，就通过while循环进行多次尝试：

```
while j in selected:
    j = randbelow(n)
```

综上所述，采用混合实现的方式，random.sample的时间复杂度会稳定在 `O(K)` 上。说了这么多，继续回到SAGE的代码，那么如果当前节点设置的抽样数为 `num_sample`，则时间复杂

度为 O(num_sample * batch_size) 。

```
# *拆解列表后，转为为多个独立的元素作为参数给union，union函数进行去重合并
unique_nodes_list = list(set.union(*samp_neighs))
# 节点标号不一定都是从0开始的，创建一个字典，key为节点ID，value为节点序号
unique_nodes = {n:i for i,n in enumerate(unique_nodes_list)}
# print(len(nodes), len(unique_nodes), len(samp_neighs))
# nodes表示batch内的节点，unique_nodes表示batch内的节点用到的所有邻居节点，unique_nodes > nodes
# 创建一个nodes * unique_nodes大小的矩阵
mask = Variable(torch.zeros(len(samp_neighs), len(unique_nodes)))
# 遍历每一个邻居集合的每一个元素，并且通过ID(key)获取到节点对应的序号--列切片
column_indices = [unique_nodes[n] for samp_neigh in samp_neighs for n in samp_neigh]
# 行切片，比如samp_neighs = [{3,5,9}, {2,8}, {2}]，行切片为[0,0,0,1,1,2]
row_indices = [i for i in range(len(samp_neighs)) for j in range(len(samp_neighs[i]))]
# 利用切片创建邻接矩阵
mask[row_indices, column_indices] = 1
```

这一堆代码是为了构造邻接矩阵。

```
# 统计每一个节点的邻居数量
num_neigh = mask.sum(1, keepdim=True)
# 分比例
mask = mask.div(num_neigh)
# embed_matrix: [n, m]
# n: unique_nodes
# m: dim
if self.cuda:
    embed_matrix = self.features(torch.LongTensor(unique_nodes_list).cuda())
else:
    embed_matrix = self.features(torch.LongTensor(unique_nodes_list))
# mean操作
to_feats = mask.mm(embed_matrix)
```

这里就实现了mean方式的AGGREGATE。

[注]代码网络层部分还没有解析，这一块的内容比较简单，代码只给出了mean方式下的实现和监督式学习过程，后面会对实验效果进行分析，并对其他AGGREGATE方式和无监督方式进行实现。