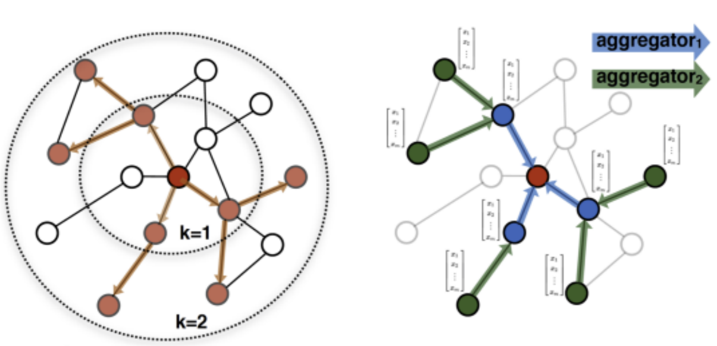
Graph sage

理论

graphsage是可以运用在无监督学习即节点聚类任务的一种方式，因为其sample和aggregate使得它能够具有捕捉图结构的特性。当然，监督任务，即节点分类，边连接预测也是可以做到的。



1. Sample neighborhood

2. Aggregate feature information from neighbors

输入参数

Input : Graph $\mathcal{G}(\mathcal{V},\mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1,...,K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1,...,K\}$; neighborhood function $\mathcal{N}: v \rightarrow 2^{\mathcal{V}}$

1. Graph（V,E):V—节点集合，E—边集合
2. input features {xv, ∀v ∈ V};输入的节点特征
3.depth K：是表示深度计算k阶邻居
4.Wk（weight matrices）-指得是对k层在其进行聚合的时候都赋予一个单独的权重矩阵
5. neighborhood function N : v → 2^V表示的是v节点的邻居节点

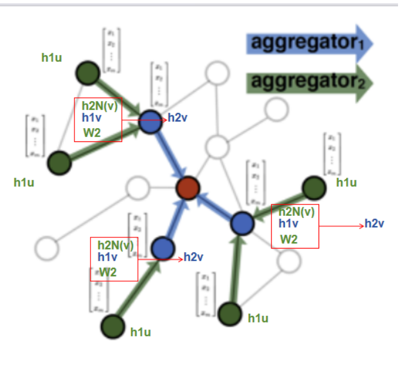
输出参数

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

节点v的向量表示

实现流程

```
1 $\mathbf{h}^0_v \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$ ;
2 for $k = 1...K$ do
3     for $v \in \mathcal{V}$ do
4         $\mathbf{h}^k_{\mathcal{N}(v)} \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}^{k-1}_u, \forall u \in \mathcal{N}(v)\})$;
5         $\mathbf{h}^k_v \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}^{k-1}_v, \mathbf{h}^k_{\mathcal{N}(v)})\right)$
6     end
7     $\mathbf{h}^k_v \leftarrow \mathbf{h}^k_v / \|\mathbf{h}^k_v\|_2, \forall v \in \mathcal{V}$
8 end
9 $\mathbf{z}_v \leftarrow \mathbf{h}^K_v, \forall v \in \mathcal{V}$
```



这里k的含义是进行几次聚合操作，而不是k阶邻居。

代码实现

1. 数据预处理

1-1. 节点特征矩阵构建、标签列表、节点索引、标签索引

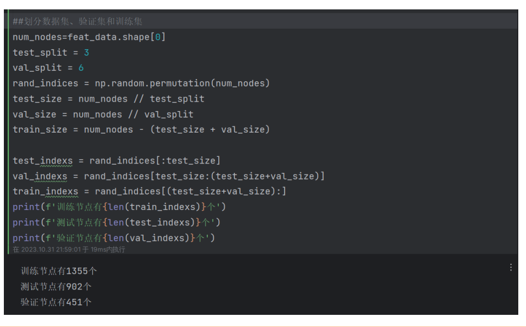
```
from collections import defaultdict
##设置读取路径
cora_content_file = config['file_path.cora_content']
cora_cite_file = config['file_path.cora_cite']

feat_data = []# 节点特征矩阵
labels = [] # 标签列表
node_map = {} # 节点索引
label_map = {} # 标签索引
with open(cora_content_file) as fp:
    for i,line in enumerate(fp):
        ##分割行
        info = line.strip().split()
        ##遍历出每个列中的第二个到倒数第二个，从而获得节点特征表示
        feat_data.append([float(x) for x in info[1:-1]])
        ##获取节点编号并且将其转化为节点索引
        node_map[info[0]] = i
        ##如果标签列表中的最后一个位置没有标签索引
        if not info[-1] in label_map:
        ##将他的标签索引设定为它所在位置索引
            label_map[info[-1]] = len(label_map)
        ##将其进行标签索引代入标签列表
        labels.append(label_map[info[-1]])
        ##将节点特征列表转化为数组形式
feat_data = np.asarray(feat_data)
        ##将标签列表转化为数组形式
labels = np.asarray(labels, dtype=np.int64)
```

1-2 构建邻接字典

```
##将元组转化为列表
adj_lists = defaultdict(set)
with open(cora_cite_file) as fp:
    for i,line in enumerate(fp):
        ## 将边转化为两个节点
        info = line.strip().split()
        ##判断是否转化后是两个节点
        assert len(info) == 2
        ##将起始节点赋值到paper1中将并将节点编号转化为索引
        paper1 = node_map[info[0]]
        ##将截止节点复制到paper2中将并将节点编号转化为索引
        paper2 = node_map[info[1]]
        ##形成节点和邻接节点连接的字典
        adj_lists[paper1].add(paper2)
        adj_lists[paper2].add(paper1)
        ##判断节点特征个数应该等于标签个数应该等于节点和邻接节点连接的字典
assert len(feat_data) == len(labels) == len(adj_lists)
print(adj_lists)
```

```
defaultdict(<class 'set'>, {163: {1536, 2563, 2564, 1030, 1544, 523, 530, 1530, 22,
1558, 2077, 546, 1571, 1572, 1573, 1060, 1575, 2600, 1065, 42, 2604, 1069, 1070, 563,
1590, 55, 1594, 2116, 1098, 1611, 1099, 2638, 1106, 1110, 1113, 602, 1114, 606,
2659, 1127, 1130, 2667, 1136, 2673, 624, 1139, 1650, 2173, 2175, 2177, 129, 1153,
1159, 1675, 141, 145, 658, 659, 2196, 1685, 2200, 1689, 2202, 1178, 2204, 1691, 1696,
174, 689, 1715, 1717, 1205, 1206, 1207, 2232, 188, 189, 191, 1728, 1729, 1218, 1219,
1734, 1224, 1225, 2248, 2251, 714, 717, 2259, 727, 2265, 219, 2274, 1253, 743, 744,
2280, 1257, 237, 2286, 1775, 757, 2295, 1784, 2296, 1274, 2298, 765, 1277, 1792, 769,
1286, 1799, 266, 2316, 781, 1807, 1303, 1305, 793, 800, 290, 1831, 813, 309, 1334,
1333, 2361, 1850, 2363, 1872, 1362, 856, 346, 2396, 1890, 1379, 1380, 1905, 380,
1404, 1410, 390, 395, 910, 402, 2451, 1943, 415, 422, 935, 940, 942, 1457, 1971,
1467, 448, 961, 966, 2518, 982, 2521, 1498, 1515, 2039, 2554, 2557}, 402: {163, 681,
```

2. 划分数据集



```
##划分数据集，验证集和训练集
num_nodes=feat_data.shape[0]
test_split = 3
val_split = 6
rand_indices = np.random.permutation(num_nodes)
test_size = num_nodes // test_split
val_size = num_nodes // val_split
train_size = num_nodes - (test_size + val_size)

test_indexs = rand_indices[:test_size]
val_indexs = rand_indices[test_size:(test_size+val_size)]
train_indexs = rand_indices[(test_size+val_size):]
print(f'训练节点有{len(train_indexs)}个')
print(f'测试节点有{len(test_indexs)}个')
print(f'验证节点有{len(val_indexs)}个')
```

```
训练节点有1355个
测试节点有902个
验证节点有451个
```

2. 模型架构

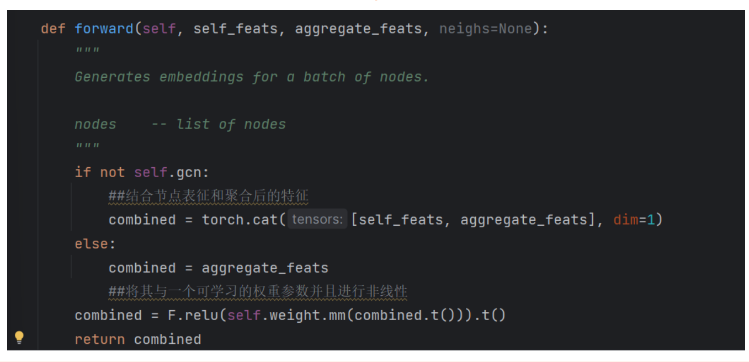
2-1 Sage layer

2-1-1 初始化设定

```
class SageLayer(nn.Module):
    """
    Encodes a node's using 'convolutional' GraphSage approach
    """
    def __init__(self, input_size, out_size, gcn=False):
        super(SageLayer, self).__init__()
    ##输入维度
        self.input_size = input_size
    #输出维度
        self.out_size = out_size
    ##gcn
        self.gcn = gcn
    ##可学习的权重参数，如果使用gcn就是输入输入维度，如果是不使用gcn那就是输入输入维度的两倍
        self.weight = nn.Parameter(torch.FloatTensor( *args: out_size, self.input_size if self.gcn else 2 * self.input_size))
    ##初始化参数
        self.init_params()

    1 个用法
    def init_params(self):
        for param in self.parameters():
            nn.init.xavier_uniform_(param)
```

对weight的条件设定中引入了是否使用gcn的原因是，gcn是本身是会建立一个自连接的边从而在更新节点特征的时候不单单只是聚合邻居节点特征而是会包含自身节点特征，但是如果不引用gcn的话那么是将聚合邻居节点的特征和自身节点特征进行拼接然后以2*input_feature的输入维度和output_feature 的shape作为节点特征数量（维度）变化的权重参数矩阵。从而保持输入和输出后的节点特征维度的一致性。

2-1-2 前向传播

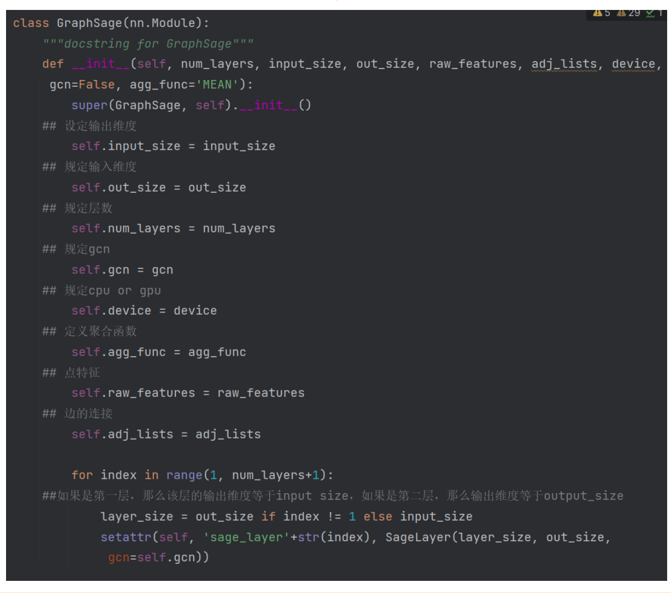


```
def forward(self, self_feats, aggregate_feats, neighs=None):
    """
    Generates embeddings for a batch of nodes.

    nodes     -- list of nodes
    """
    if not self.gcn:
        ##结合节点表征和聚合后的特征
        combined = torch.cat( tensors: [self_feats, aggregate_feats], dim=1)
    else:
        combined = aggregate_feats
        ##将其与一个可学习的权重参数并且进行非线性
    combined = F.relu(self.weight.mm(combined.t())).t()
    return combined
```

2-1-2-1 获取节点的邻居——_get_unique_neighs_list

2-2 graphsage

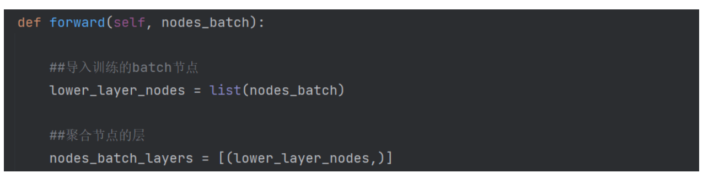
2-2-1 初始化设定



```
class GraphSage(nn.Module):
    """docstring for GraphSage"""
    def __init__(self, num_layers, input_size, out_size, raw_features, adj_lists, device,
     gcn=False, agg_func='MEAN'):
        super(GraphSage, self).__init__()
    ## 设定输出维度
        self.input_size = input_size
    ## 规定输入维度
        self.out_size = out_size
    ## 规定层数
        self.num_layers = num_layers
    ## 规定gcn
        self.gcn = gcn
    ## 规定cpu or gpu
        self.device = device
    ## 定义聚合函数
        self.agg_func = agg_func
    ## 点特征
        self.raw_features = raw_features
    ## 边的连接
        self.adj_lists = adj_lists

        for index in range(1, num_layers+1):
    ##如果是第一层，那么该层的输出维度等于input size，如果是第二层，那么输出维度等于output_size
            layer_size = out_size if index != 1 else input_size
            setattr(self, 'sage_layer'+str(index), SageLayer(layer_size, out_size,
             gcn=self.gcn))
```

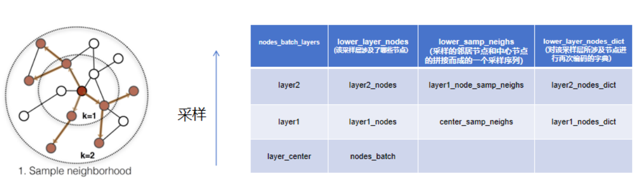
2-2-2 前向传播

2-2-2-1 训练批次的节点导入& 聚合层列表的建立

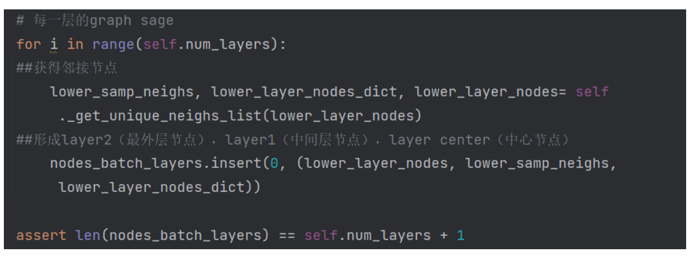


```
def forward(self, nodes_batch):

    ##导入训练的batch节点
    lower_layer_nodes = list(nodes_batch)

    ##聚合节点的层
    nodes_batch_layers = [(lower_layer_nodes,)]
```

2-2-2-2 构建采样层列表



nodes_batch_layers	lower_layer_nodes (该采样层涉及了哪些节点)	lower_samp_neighs (采样的邻居节点和中心节点的拼接而成的一个采样序列)	lower_layer_nodes_dict (对该采样层所涉及节点进行再次编码的字典)
layer2	layer2_nodes	layer1_node_samp_neighs	layer2_nodes_dict
layer1	layer1_nodes	center_samp_neighs	layer1_nodes_dict
layer_center	nodes_batch		



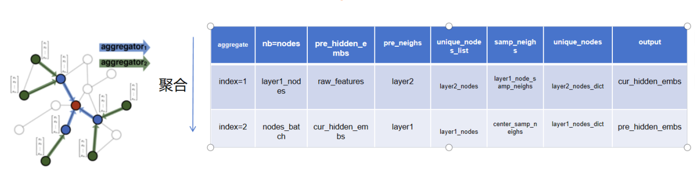
```
# 每一层的graph sage
for i in range(self.num_layers):
##获得邻接节点
    lower_samp_neighs, lower_layer_nodes_dict, lower_layer_nodes= self
     ._get_unique_neighs_list(lower_layer_nodes)
##形成layer2（最外层节点），layer1（中间层节点），layer center（中心节点）
    nodes_batch_layers.insert(0, (lower_layer_nodes, lower_samp_neighs,
     lower_layer_nodes_dict))

assert len(nodes_batch_layers) == self.num_layers + 1
```

1. 先对训练batch的节点（layer_center)使用采样邻居函数，获得layer_center的邻居节点(layer1)随后将其插入nodes_batch_layers列表的0号位置，即生成[layer1,layer_center]
2. 再对layer1节点使用采样邻居函数，获得layer1中节点的邻居节点（layer2）随后将其插入nodes_batch_layers 列表的0号位置即生成[layer2,layer1,layer_center]
3.总结：通过这样的方式其实是实现了从内到外的采样节点，并且为之后的聚合节点提供了一个聚合层

运用采样邻居函数

```
##获得节点的邻居
def _get_unique_neighs_list(self, nodes, num_sample=10):
    _set = set
    ##获取每个节点的邻居节点
    to_neighs = [self.adj_lists[int(node)] for node in nodes]

    ##进行采样
    if not num_sample is None:
        _sample = random.sample
    ##如果邻居节点个数大于等于十个，则随机取十个，如果邻居节点个数小于十个则全取
        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
    ##将采样的邻居节点和本身的中心节点结合得到节点所属邻居的集合
    samp_neighs = [samp_neigh | set([nodes[i]]) for i, samp_neigh in enumerate
     (samp_neighs)]

    ##进行去重操作
    _unique_nodes_list = list(set.union(*samp_neighs))

    ##进行重新排列
    i = list(range(len(_unique_nodes_list)))

    ##对字典进行重新编号
    unique_nodes = dict(list(zip(_unique_nodes_list, i)))

    ##返回所有节点的邻居，返回所有节点的编号，返回所有节点的列表
    return samp_neighs, unique_nodes, _unique_nodes_list
```

2-2-2-3 聚合邻居节点更新节点信息



aggregate	nb=nodes	pre_hidden_embs	pre_neighs	unique_nodes_list	samp_neighs	unique_nodes	output
index=1	layer1_nodes	raw_features	layer2	layer2_nodes	layer1_node_samp_neighs	layer2_nodes_dict	cur_hidden_embs
index=2	nodes_batch	cur_hidden_embs	layer1	layer1_nodes	center_samp_neighs	layer1_nodes_dict	pre_hidden_embs

```
##将节点特征赋予到变量pre_hidden_embs
pre_hidden_embs = self.raw_features
##从最外层即layer2开始遍历，遍历到layer_center
##nodes_batch_layers=[layer2,layer1,layer_center]
##index表示聚合的第几次
for index in range(1, self.num_layers+1):
##以下一层（聚合第一次是layer1，聚合第二次是layer_center)作为更新目标
    nb = nodes_batch_layers[index][0]
##以本层作为邻居节点作为提供聚合的信息（第一次是layer2，第二次是layer1）
    pre_neighs = nodes_batch_layers[index-1]

##聚合函数进行聚合得到中心节点特征
    aggregate_feats = self.aggregate(nb, pre_hidden_embs, pre_neighs)
    sage_layer = getattr(self, 'sage_layer'+str(index))

##如果聚合次数大于1
    if index > 1:
        nb = self._nodes_map(nb, pre_hidden_embs, pre_neighs)
##利用中心节点表征和聚合后的中心节点特征进行拼接然后连一个可学习的权重参数
    cur_hidden_embs = sage_layer(self_feats=pre_hidden_embs[nb],
                                 aggregate_feats=aggregate_feats)
    pre_hidden_embs = cur_hidden_embs
```

在论文中预设的是聚合两层的效果是最好的，而在该代码中也是进行两次的聚合，我们可见在进行完第二次聚合而更新节点特征后，我们会调用定义的sage_layer进行操作从而生成一个最终的embedding表示

聚合函数

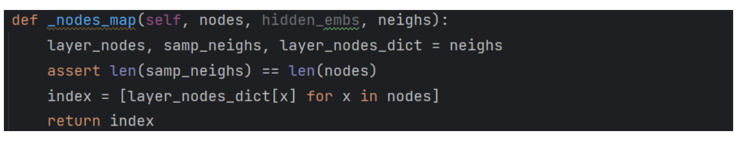
```
def aggregate(self, nodes, pre_hidden_embs, pre_neighs, num_sample=10):

    ##取出本层节点列表（第一次聚合是layer2，第二次聚合是layer1）
    unique_nodes_list, samp_neighs, unique_nodes = pre_neighs

    assert len(nodes) == len(samp_neighs)

    ##判断本层节点列表是否包含了聚合目标节点
    indicator = [(nodes[i] in samp_neighs[i]) for i in range(len(samp_neighs))]
    assert (False not in indicator)

    ##将本层节点列表包含聚合目标节点的部分去掉，因为最开始采样的时候加入了
    if not self.gcn:
        samp_neighs = [(samp_neighs[i]-set([nodes[i]])) for i in range(len(samp_neighs))]

    ##从全局节点特征矩阵中提取出本层节点列表涉及的节点的节点特征
    if len(pre_hidden_embs) == len(unique_nodes):
        embed_matrix = pre_hidden_embs
    else:
        embed_matrix = pre_hidden_embs[torch.LongTensor(unique_nodes_list)]

    ##构建shape为（聚合目标节点数量，聚合目标节点和本层节点数量）邻接矩阵
    mask = torch.zeros(len(samp_neighs), len(unique_nodes))
    column_indices = [unique_nodes[n] for samp_neigh in samp_neighs for n in samp_neigh]
    row_indices = [i for i in range(len(samp_neighs)) for j in range(len(samp_neighs[i]))]

    ##将其有链接的地方记为1
    mask[row_indices, column_indices] = 1
    print(mask.shape[1])
```

1. 通过mask得到了聚合目标节点和（本层节点+聚合目标节点）的连接关系即邻接矩阵。
2.利用embed_matrix来提取本层节点+聚合目标节点的节点特征矩阵
3.通过embed_matrix和mask可以聚合本层节点的信息从而更新聚合目标节点

选取聚合方式进行聚合

```
##将其有链接的地方记为1
mask[row_indices, column_indices] = 1
print(mask.shape[1])
##选择平均的方式进行聚合
if self.agg_func == 'MEAN':
##按行求和得到每个中心节点的连接的邻居节点的个数
    num_neigh = mask.sum( dim: 1, keepdim=True)
##按行进行归一化操作
    mask = mask.div(num_neigh).to(embed_matrix.device)
##矩阵相乘，相当于聚合周围临界信息并求和
    aggregate_feats = mask.mm(embed_matrix)

elif self.agg_func == 'MAX':
    # print(mask)
    indexs = [x.nonzero() for x in mask==1]
    aggregate_feats = []
    # self.dc.logger.info('5')
    for feat in [embed_matrix[x.squeeze()] for x in indexs]:
        if len(feat.size()) == 1:
            aggregate_feats.append(feat.view(1, -1))
        else:
            aggregate_feats.append(torch.max(feat,0)[0].view(1, -1))
    aggregate_feats = torch.cat(aggregate_feats,  dim: 0)

# self.dc.logger.info('6')

return aggregate_feats
```

_nodes_map



```
def _nodes_map(self, nodes, hidden_embs, neighs):
    layer_nodes, samp_neighs, layer_nodes_dict = neighs
    assert len(samp_neighs) == len(nodes)
    index = [layer_nodes_dict[x] for x in nodes]
    return index
```

在第二次聚合的时候会使用，主要是为了提取第一次聚合后的节点特征作为其送入sagelayer的初始特征

sage_layer

```
class SageLayer(nn.Module):
    def __init__(self, input_size, out_size, gcn=False):
        super(SageLayer, self).__init__()
    ##输入维度
        self.input_size = input_size
    #输出维度
        self.out_size = out_size
    ##gcn
        self.gcn = gcn
    ##可学习的权重参数，如果使用gcn就是输入输入维度，如果是不使用gcn那就是输入输入维度的两倍
        self.weight = nn.Parameter(torch.FloatTensor( *args: out_size, self.input_size if self.gcn else 2 * self.input_size))
    ##初始化参数
        self.init_params()

    1 个用法
    def init_params(self):
        for param in self.parameters():
            nn.init.xavier_uniform_(param)

    def forward(self, self_feats, aggregate_feats, neighs=None):
        """
        Generates embeddings for a batch of nodes.

        nodes     -- list of nodes
        """
        if not self.gcn:
            ##结合节点表征和聚合后的特征
            combined = torch.cat( tensors: [self_feats, aggregate_feats], dim=1)
        else:
            combined = aggregate_feats
            ##将其与一个可学习的权重参数并且进行非线性
        combined = F.relu(self.weight.mm(combined.t())).t()
        return combined
```

将初始特征和聚合后的特征进行拼接然后进行特征数量变化权重矩阵

2-3-1 分类器定义

```
class Classification(nn.Module):

    def __init__(self, emb_size, num_classes):
        super(Classification, self).__init__()

        #self.weight = nn.Parameter(torch.FloatTensor(emb_size, num_classes))
        self.layer = nn.Sequential(
                                nn.Linear(emb_size, num_classes)
                                #nn.ReLU()
                        )
        self.init_params()

    1 个用法
    def init_params(self):
        for param in self.parameters():
            if len(param.size()) == 2:
                nn.init.xavier_uniform_(param)

    def forward(self, embeds):
        logists = torch.log_softmax(self.layer(embeds),  dim: 1)
        return logists
```

设定一个全连接层，输出部分对应着分类的种类，随后再用一个log_softmax输出其属于每个类型的概率

2-3-2 损失计算

```
if learn_method == 'sup':
    # superivsed learning
    #用softmax的全连接进行最后的输出分类
    logists = classification(embs_batch)
    loss_sup = -torch.sum(logists[range(logists.size(0)), labels_batch], 0)
    loss_sup /= len(nodes_batch)
    loss = loss_sup
```

负对数似然损失，通常称为交叉熵损失，它通常用作具有 softmax 输出的监督分类任务的损失函数。 logits 是神经网络对每个类别的原始分数（未归一化概率）。
负对数似然损失使用 logits 和真实标签 labels_batch 来计算，它衡量了预测的概率与真实类别标签的匹配程度。
最后，将损失在批次中平均，得到每个样本的平均损失，并将这个值赋给变量 loss。