GraphSAGE/GCN/GAT

GraphSAGE

http://snap.stanford.edu/graphsage/

https://github.com/williamleif/GraphSAGE

https://github.com/williamleif/graphsage-simple/

inductive

原理解读

Graph Sample And Aggregate：graphsage

分为聚合，参数学习 两部分（主要侧重这两个方面）

聚合

整个核心算法如下所示（encoder部分）：

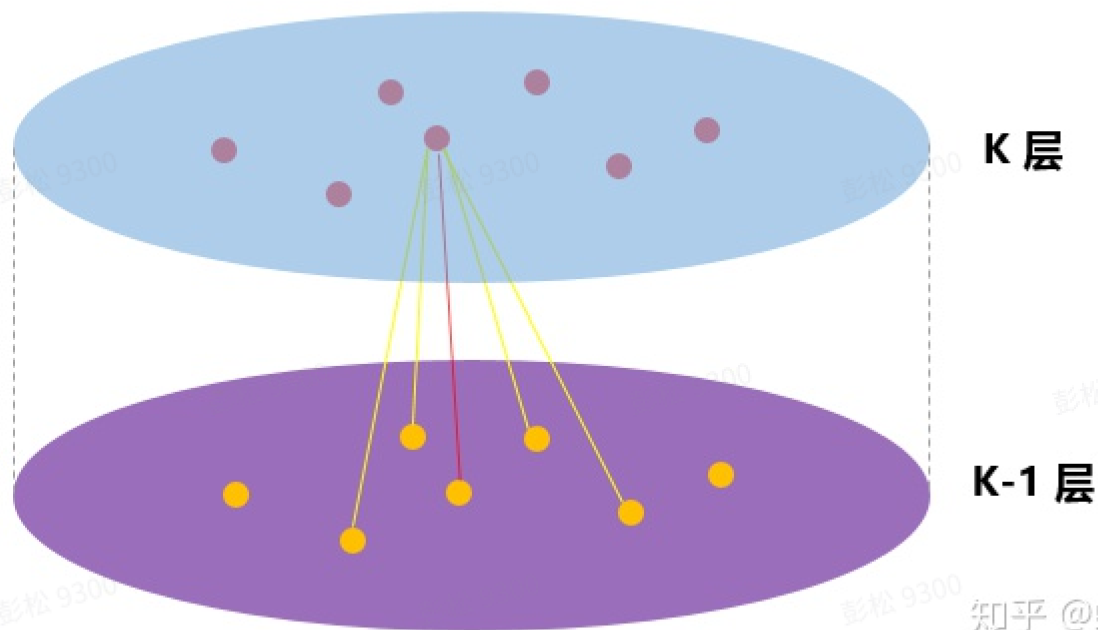
Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

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 $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

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

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

直观上可见下图（原论文中的聚合图不够直观，我在知乎上找了一个图）：



其本意就是第k层node的embedding由其周围的node embedding聚合得到，k层之后，原node 的embedding就会学到。

聚合的方法在论文中提到了Mean aggregator，LSTM aggregator，Pooling aggregator。

Mean aggregator：邻边节点取均值

LSTM aggregator：邻边节点的随机序列过lstm 【lstm有更强大的表达能力】

Pooling aggregator：邻边节点embedding每一维对应的最大值作为最终embedding的对应维度值

实验结果看似乎Pooling的方法更好：

	Citation		Reddit		PPI	
Name	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	**0.908**	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	**0.907**	**0.954**	0.482	**0.612**
GraphSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%

参数学习

此处我主要想讲graphSAGE和w2v不同的地方，节点的初始化是用节点的特征来初始化的，最终经过k层聚合后得到Zv，而无监督学习的时候并没有使用context embedding，而是直接用encoder的结果，如下（即line中order等于1的情况）：

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right), \quad (1)$$

where v is a node that co-occurs near u on fixed-length random walk, σ is the sigmoid function, P_n is a negative sampling distribution, and Q defines the number of negative samples. Importantly, unlike previous embedding approaches, the representations $\mathbf{z}_u$ that we feed into this loss function are generated from the features contained within a node's local neighborhood, rather than training a unique embedding for each node (via an embedding look-up).

关于超参设置，K=2，S1=25，S2=10，自行调整。

关于样本，使用randwalk生成序列。

【我看tf_euler中使用了context embedding】

源码解读

阅读的是pytorch版本。

pytorch版本实现了GraphSAGE-mean和GraphSAGE-GCN有监督的版本。

使用的数据是cora数据集，该数据集包含两个文件：

content文件为如下格式：

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

第二个参数表明在词表中的每一个词是否在该文章中，第三个参数是这个文章的类别。

cites文件如下所示：

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

每一行包含两个paper id，第一个id是被引用的paper id，第二个id是引用了第一个paper id的paper id。

核心代码就这么几行：

```
68      num_nodes = 2708
69      feat_data, labels, adj_lists = load_cora()
70      features = nn.Embedding(2708, 1433)
71      features.weight = nn.Parameter(torch.FloatTensor(feat_data), requires_grad=False)
72     # features.cuda()
73
74      agg1 = MeanAggregator(features, cuda=True)
75      enc1 = Encoder(features, 1433, 128, adj_lists, agg1, gcn=True, cuda=False)
76      agg2 = MeanAggregator(lambda nodes : enc1(nodes).t(), cuda=False)
77      enc2 = Encoder(lambda nodes : enc1(nodes).t(), enc1.embed_dim, 128, adj_lists, agg2,
78              base_model=enc1, gcn=True, cuda=False)
79      enc1.num_samples = 5
80      enc2.num_samples = 5
81
82      graphsage = SupervisedGraphSage(7, enc2)
```

可以看到最开始的embedding是使用node的特征进行初始化的，使用了两层聚合。

MeanAggregator过程如下：

```
30      def forward(self, nodes, to_neighs, num_sample=10):
31          """
32          nodes --- list of nodes in a batch
33          to_neighs --- list of sets, each set is the set of neighbors for node in batch
34          num_sample --- number of neighbors to sample. No sampling if None.
35          """
36          # Local pointers to functions (speed hack)
37          _set = set
38          if not num_sample is None:
39              _sample = random.sample
40              samp_neighs = [_set(_sample(to_neigh,
41                              num_sample,
42                              )) if len(to_neigh) >= num_sample else to_neigh for to_neigh in to_neighs]
43          else:
44              samp_neighs = to_neighs
45
46          if self.gcn:
47              samp_neighs = [samp_neigh + set([nodes[i]]) for i, samp_neigh in enumerate(samp_neighs)]
48          unique_nodes_list = list(set.union(*samp_neighs))
49          unique_nodes = {n:i for i,n in enumerate(unique_nodes_list)}
50          mask = Variable(torch.zeros(len(samp_neighs), len(unique_nodes)))
51          column_indices = [unique_nodes[n] for samp_neigh in samp_neighs for n in samp_neigh]
52          row_indices = [i for i in range(len(samp_neighs)) for j in range(len(samp_neighs[i]))]
53          mask[row_indices, column_indices] = 1
54          if self.cuda:
55              mask = mask.cuda()
56          num_neigh = mask.sum(1, keepdim=True)
57          mask = mask.div(num_neigh)
58          if self.cuda:
59              embed_matrix = self.features(torch.LongTensor(unique_nodes_list).cuda())
60          else:
61              embed_matrix = self.features(torch.LongTensor(unique_nodes_list))
62          to_feats = mask.mm(embed_matrix)
63          return to_feats
```

Encoder部分如下：

```
33        def forward(self, nodes):
34            """
35            Generates embeddings for a batch of nodes.
36
37            nodes     -- list of nodes
38            """
39            neigh_feats = self.aggregator.forward(nodes, [self.adj_lists[int(node)] for node in nodes],
40                    self.num_sample)
41            if not self.gcn:
42                if self.cuda:
43                    self_feats = self.features(torch.LongTensor(nodes).cuda())
44                else:
45                    self_feats = self.features(torch.LongTensor(nodes))
46                combined = torch.cat([self_feats, neigh_feats], dim=1)
47            else:
48                combined = neigh_feats
49            combined = F.relu(self.weight.mm(combined.t()))
50            return combined
```

训练的过程中是按batch训练的，两阶段聚合过程：先找到batch node (记为A)的邻边节点，将这些邻边节点去重组成一个batch node(记为B)后再喂到encoder中，encoder的aggregator再找到这个batch node的邻边节点，及对应的embedding（从features中），mean聚合后得到B中每个node的embedding，mean聚合进而得到A中node 的embedding。

也就是说整个过程始终只会涉及到部分节点，同时在训练过程中train也是在不断的shuffle，以保证batch node的随机性。

GraphSAGE-GCN：将node和其邻边node同等对待

GraphSAGE-mean：将node embedding和其邻边聚合embedding同等对待

扩展

细想一下整个过程，迭代次数越多，越看重邻边节点，本质上和cnn类似。

我理解用z来表示最后的node embedding，而输入x是node的feature，那即特征压缩，默认node feature和neighbor node feature能够表示node的全部信息，有点类似dssm中user和item的交互。

GCN

https://arxiv.org/abs/1609.02907

http://tkipf.github.io/graph-convolutional-networks/#gcns-part-iii-embedding-the-karate-club-network

transductive

原理

原论文过于复杂，可以照着第二个链接看。

N个节点，会对应N*N的邻接矩阵A（边度数），N*D的特征矩阵X，那么GCN直观的理解如下：

$$H^{(l+1)} = f(H^{(l)}, A),$$

with $H^{(0)} = X$ and $H^{(L)} = Z$ (or z for graph-level outputs), L being the number of layers.

当前node的feature由上一层node的邻边node feature决定，feature即为隐藏层。

再进一步即为：

$$f(H^{(l)}, A) = \sigma\left(AH^{(l)}W^{(l)}\right),$$

where $W^{(l)}$ is a weight matrix for the l-th neural network layer and $\sigma(\cdot)$ is a non-linear activation function

H表示l层node的隐藏层，A表示node的邻边node度数(权重)，即l+1层node的隐藏层由l层node邻边node隐藏层决定，最后W是维度变换，再过激活函数得到l+1层node表示。

因为A是没有进行归一化的，而且对角线上的元素都为0，即没有考虑node本身的特征，因此做如下改动：

$$f(H^{(l)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right),$$

with $\hat{A} = A + I$, where I is the identity matrix and $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$.

D为度数矩阵，是一个对角矩阵，只包含顶点的度数，即顶点出(入)边的度数，将A中的对角线设置为1，上式DAD本质上就是对A归一化，即A中node的邻边度数除以node的度数。来看一个更直观的理解：

$$h_{v_i}^{(l+1)} = \sigma\left(\sum_j \frac{1}{c_{ij}} h_{v_j}^{(l)} W^{(l)}\right),$$

where j indexes the neighboring nodes of v_i. c_{ij} is a normalization constant for the edge (v_i, v_j) which originates from using the symmetrically normalized adjacency matrix $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ in our GCN model. We now see that this

本质就是用上一层node feature和邻边node feature来表示该层node的feature，以及该如何聚合的问题。

代码

https://github.com/tkipf/pygcn

代码比较简单，可以参考https://zhuanlan.zhihu.com/p/78191258

改代码是处理全部的节点。

GAT

https://arxiv.org/abs/1710.10903

inductive

原理

GCN中node的邻边节点聚合使用的是权重的归一化，GAT只是将权重换成attention中的aij（我理解是这样的）【一层GAT】

The input to our layer is a set of node features, $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}, \vec{h}_i \in \mathbb{R}^F$, where N is the number of nodes, and F is the number of features in each node. The layer produces a new set of node features (of potentially different cardinality F'), $\mathbf{h}' = \{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\}, \vec{h}'_i \in \mathbb{R}^{F'}$, as its output.

In order to obtain sufficient expressive power to transform the input features into higher-level features, at least one learnable linear transformation is required. To that end, as an initial step, a shared linear transformation, parametrized by a *weight matrix*, $\mathbf{W} \in \mathbb{R}^{F' \times F}$, is applied to every node. We then perform *self-attention* on the nodes—a shared attentional mechanism $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \rightarrow \mathbb{R}$ computes *attention coefficients*

$$e_{ij} = a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j) \quad (1)$$

that indicate the *importance* of node j's features to node i. In its most general formulation, the model allows every node to attend on every other node, *dropping all structural information*. We inject the graph structure into the mechanism by performing *masked attention*—we only compute e_{ij} for nodes $j \in \mathcal{N}_i$, where $\mathcal{N}_i$ is some *neighborhood* of node i in the graph. In all our experiments, these will be exactly the first-order neighbors of i (including i). To make coefficients easily comparable across different nodes, we normalize them across all choices of j using the softmax function:

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}. \quad (2)$$

In our experiments, the attention mechanism a is a single-layer feedforward neural network, parametrized by a weight vector $\vec{\mathbf{a}} \in \mathbb{R}^{2F'}$, and applying the LeakyReLU nonlinearity (with negative input slope $\alpha = 0.2$). Fully expanded out, the coefficients computed by the attention mechanism (illustrated by Figure 1 (left)) may then be expressed as:

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)} \quad (3)$$

$$\vec{h}'_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}\vec{h}_j\right).$$

考虑多头的情况：

$$\vec{h}'_i = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right)$$

代码

https://github.com/Diego999/pyGAT

扩展

inductive和transductive

transductive（直推式学习）：在固定的图上直接学习每个节点的embedding，每次只考虑当前数据

inductive（归纳学习）：即学习在图上生成节点embedding的方法，而不是直接学习节点的embedding，GraphSAGE是以学习聚合节点邻居生成节点embedding的函数方式，将GCN扩展成归纳学习任务。