

# GraphSAGE 源码分析报告

——袁宇箭 2018K8009908017 2021.01.31

## 一、Graphsage 简介

### 0. 什么是深度学习？

深度学习是一类机器学习算法，它使用多层渐进地从原始输入中提取高级特征。例如，在图像处理中，较低的层可能识别边缘，而较高的层可能识别与人类相关的概念，如数字、字母或面孔。（来自 [https://en.wikipedia.org/wiki/Deep\\_learning](https://en.wikipedia.org/wiki/Deep_learning)）

总的来说，深度学习主要涉及三类方法：

- (1) 基于卷积运算的神经网络系统，即卷积神经网络(CNN)。
- (2) 基于多层神经元的自编码神经网络，包括自编码( Auto encoder)以及近年来受到广泛关注的稀疏编码两类( Sparse Coding)。
- (3) 以多层自编码神经网络的方式进行预训练，进而结合鉴别信息进一步优化神经网络权值的深度置信网络(DBN)。

我们目前所接触和学习到的大多是基于卷积运算的神经网络。近年来围绕卷积神经网络而开展的研究取得了相当可观的成果，python 也有 TensorFlow、pytorch、sklearn 等库支持神经网络的计算，如此一来代码的撰写则会方便许多，同时也可以保证模型运行的高效性。

### 1. 什么是 Graphsage？

GraphSAGE 即 Graph **S**ample and **aggreG**at**E**，类似于传统的图卷积神经网络 GCN，它也是一种图的深度学习算法，它的特点在于引入了 Inductive 和 sample 这两个特点。GraphSage 的出现完成了机器学习从 Transductive（直推式学习）到 inductive（归纳式学习）的转变，同时还提出了 Mean aggregator、Pooling aggregator、LSTM aggregator 这三种聚合函数，拥有更强的表达能力。

### 2. Grpaphsage 有什么特点？

(1) 以往 GCN 算法是典型的直推式学习方法，它所学习到的参数很大程度上与图的结构有关，一旦图发生了变化则需要重新学习参数；而 GraphSAGE 便是采用归纳式学习方法，它学习节点之间的聚合模式，利用结点领域的聚合模型直接学习处新节点的嵌入特征，只要图不发生太大的变化则无需重新学习参数，大大提高了算法的鲁棒性。

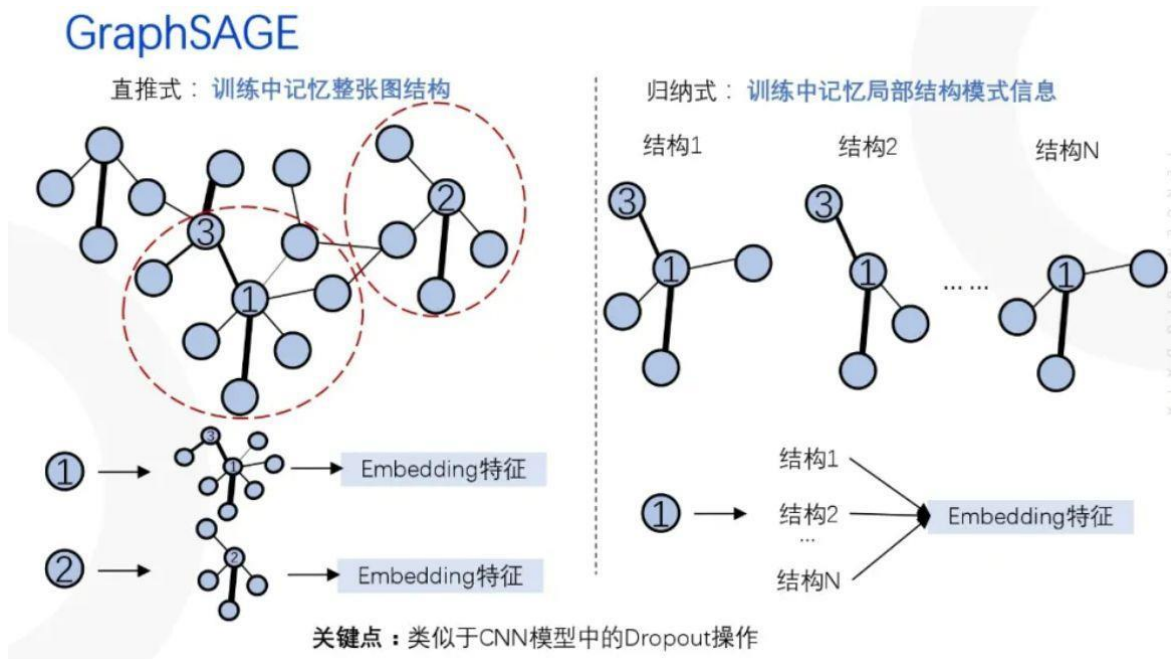


图 1 Graphsage 归纳式特点图

(2) 引入邻居采样，将直推式节点只表示一种局部结构转变为对应多种局部结构的节点归纳表示，可有效防止训练过拟合，增强泛化能力。

(3) 通过邻居采样的方式解决了 GCN 内存爆炸的问题，适用于大规模图。

### 3. 怎么使用 Grpaphsage 算法程序？

Graphsage 源码可以从 GitHub 上获取，链接 <https://github.com/williamleif/GraphSAGE>。

Graphsage 整体分为有监督学习和无监督学习部分，分别对应 supervised\_train.py 和 unsupervised\_train.py 文件，每个文件中均有自己单独的 main 函数，在运行的时候只需在对应的文件里运行即可。

具体的 API 也分为有监督学习和无监督学习的，此处以有监督学习为例。在 supervised\_train.py 文件中，存在 main 函数

```
def main(argv=None):
    print("Loading training data..")
    train_data = load_data(FLAGS.train_prefix)
    print("Done loading training data..")
    train(train_data)
```

首先利用 load\_data 函数（utils.py 文件中）从数据集中获取数据并将其设置为算法所需要的格式得到 train\_data，其函数头以及返回值如下：

```
def load_data(prefix, normalize=True, load_walks=False):
    ...
    return G, feats, id_map, walks, class_map
```

根据函数的返回值我们大致可以推断出我们需要准备的数据集的内容了，源码网站为我们提供了一个数据集格式的标准以及数据集示例，如下：






 toy-ppi-G.json	Rename.
 toy-ppi-class_map.json	Rename.
 toy-ppi-feats.npy	Rename.
 toy-ppi-id_map.json	Rename.
 toy-ppi-walks.txt	Rename.

图 2 数据集格式标准图

此处以 toy-ppi 数据集为例，其图（G）、class\_map 以及 id\_map 需要是 json 格式的文件，feats 需要 npy 型文件（可以调用 python 的 numpy 库来生成），walks 需要 txt 型文件。

之后调用 train 函数并代入 load\_data 函数处理得到的以上五个数据即可完成训练，具体内部的实现逻辑因为不是本课程的重点就不详细叙述了，在接下来的分析中将会重点针对面向对象的思想进行分析而非算法本身。

为了方便用户使用 graphsage 程序，开发者专门撰写了一个 example\_supervised.sh 文件，里面包含了运行程序需要输入的命令行，在命令行内可以指定聚合的方式等具体实现中的算法。如果需要更改内部算法进行训练则只需更改 example\_supervised.sh 文件中命令行的参数即可，以下是命令行的一个示例：

```
1 python -m graphsage.supervised_train --train_prefix ./example_data/ppi --model graphsage_mean --sigmoid
```

图 3 命令行示意图

该例子中 ./example\_data/ppi 表示数据集的路径，supervised\_train 表示采用的是有监督学习，graphsage\_mean 表示采用的聚合方法是 Mean aggregator。在 linux 系统中，设置好了数据集以及命令行参数后，只需输入“sudo ./example\_supervised.sh”即可开始训练。

自己设计数据集时，需要保证和样例中一样的格式，否则可能需要修改 load\_data 函数来完成训练（不过这是一个不明智的选择，因为代码在封装之后需要保证一定的封闭性，即用户不可以随意修改程序内部代码，以免造成程序出错从而产生错误的结果）。或者可

以根据 adapter 适配器模式的思想设计一个“转接口”以保证接口的统一（将其他格式的数据转换成 graphsage 要求的数据格式）。

## 二、主要功能分析与建模

### 0. 功能选取

此处主要针对有监督学习部分的功能进行分析。有监督学习是从标签化训练数据集中推断出函数的机器学习任务，比较像我们平时做题的过程：从已知答案的练习题中推断出做题规律，从而很好地运用在考试上。

### 1. 需求建模

对于机器学习，我们需要的是利用训练集对模型进行训练，并利用测试集测试模型的准确度。运用需求模型，我们可以得到以下的分析：

#### （1）WHAT

程序需要根据用户提供的数据集，利用图（G.json 文件）中邻居节点的特征，为先前未见过的数据有效地生成节点的 Embedding。

#### （2）WHY

大图学习容易造成内存溢出以及时间过长等问题，在暂时无法提升硬件水平的情况下，设计出更高效的算法相对来说是更好的选择。

#### （3）需求分析

采用用例法，包含正常处理、异常处理以及替代处理的流程

<p><b>【用例名称】</b></p> <p>Graphsage 有监督学习算法</p> <p><b>【场景】</b></p> <p>Who：用户、程序、数据集</p> <p>Where：内存</p> <p>When：运行时（训练时）</p> <p><b>【用例描述】</b></p> <ol style="list-style-type: none"><li>1. 用户将准备好的数据集放到指定的文件目录下</li><li>2. 用户在命令行参数处设置好数据集路径和聚合方式，并运行该 sh 文件。</li></ol>
---

2.1 输入路径不存在，或者该路径下无有效数据集，打印报错信息 “path error”

2.2 命令行输入格式出错，打印报错信息 “unknown command”

3. 程序解析命令行，并开始运行相关的 main 函数

4. 运行 load\_data 函数，从数据集中读取数据

4.1 遇到无效的数据，或者不符合格式的数据，打印报错信息 “invalid data”

4.2 数据量过大导致电脑内存无法容纳，打印报错信息 “unable to allocate memory”

5. 运行 train 函数，开始训练

5.1 输入的模型不存在，打印报错信息 “Error: model name unrecognized.”

5-A: 程序采用 graphsage\_mean 模型进行训练

5-B: 程序采用 gcn 模型进行训练

5-C: 程序采用 graphsage\_seq 模型进行训练

5-D: 程序采用 graphsage\_maxpool 模型进行训练

5-E: 程序采用 graphsage\_meanpool 模型进行训练

6. 进行 validation，验证训练结果

7. 打印训练结果（包括 loss、mic、mac、time 等信息），用户得到该信息

8. 利用训练结果对新节点进行预测

#### 【用例价值】

程序在训练完后用户可以得到训练的 Graphsage 模型，从而为先前未见过的数据有效地生成节点的 Embedding。

#### 【约束和限制】

1. 输入的数据集的格式应规范。

2. 用户必须下载程序所需使用的拓展包，并且拓展包的版本也有要求（见 requirement.txt 文件）

3. 程序的运行需要一定的环境，环境的安装方法在 Dockerfile 文件中有介绍。

我们从中提取出关键的名词和动词：

**【名词】**：用户、程序、模型、validation、数据集、命令行

**【动词】**：训练、预测、打印、解析

我们可以发现，用户只是业务的参与者，没有明确的用处。同时数据集以及命令行都是用户准备好的一个输入，与程序本身并无太大的关系。Validation 只是 train 函数中的一种方法，用于验证训练结果的，起到的只是一个辅助的作用。因此真正有用的名词就是“程序”。

对于动词来说，打印只需用 print 函数实现即可，没必要专门算作某一类的功能。解析命令行也是每个程序通用的功能。因此“训练”和“预测”便是关键动词。

因此我们大致抽取出来的类及其方法和属性为：

**【类】** supervised\_models（程序）

**【属性】** loss、model\_size 等

**【方法】** build（训练）、predict（预测）

## 2. 执行流程

简单来分析，graphsage 有监督学习的训练逻辑大致如下图：

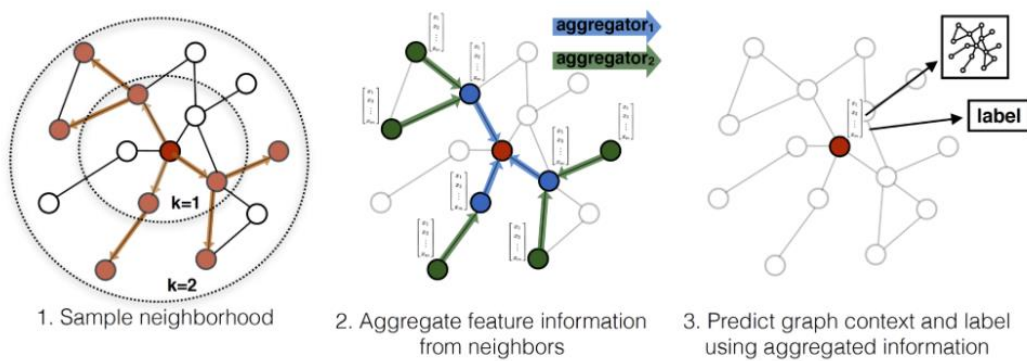


图 4 Graphsage 训练逻辑图

可以观察出图中每个结点都有一个或者多个邻居，图 2 展示了聚合的过程，蓝色结点将它的邻居（绿色结点）的特征集合在自己身上，红色结点便将其蓝色结点邻居的特征集中在自己身上，通过这样一层一层的聚合便可以计算得出结点的特征，从而用于新节点的预测。更直观的工作过程图如下：

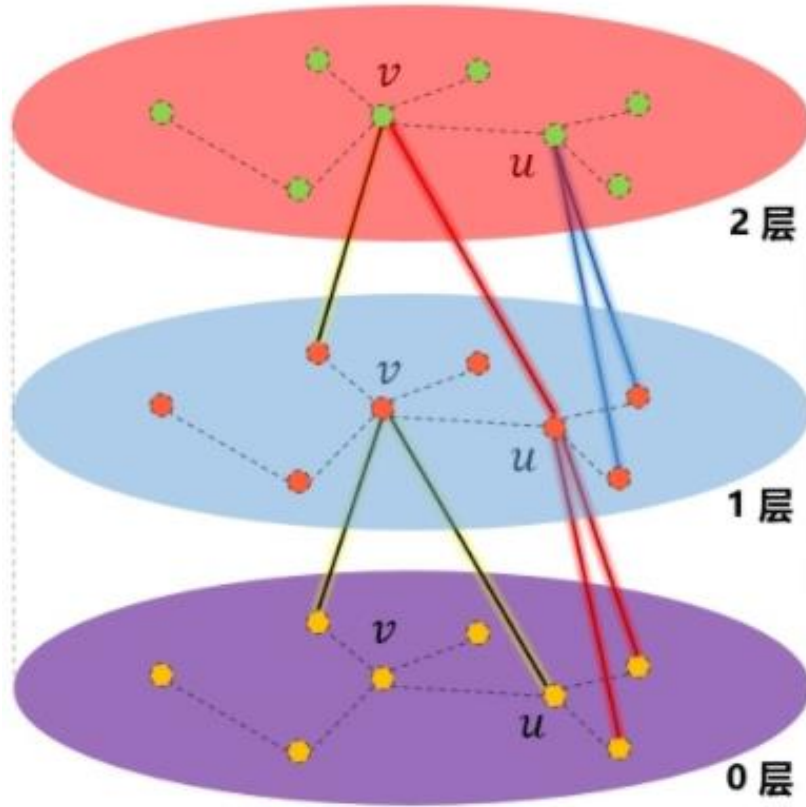


图 5 Graphsage 工作过程图

它的核心算法是对每一层将每一个点的邻居特征都聚合起来，所以需要使用两层循环来完成这个功能，具体伪代码如下：（如果对具体算法实现不感兴趣可以不用深究）

**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  $\mathbf{W}^k, \forall k \in \{1, \dots, K\}$ ; non-linearity  $\sigma$ ; differentiable aggregator functions  $\text{AGGREGATE}_k, \forall k \in \{1, \dots, K\}$ ; 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  $k = 1 \dots K$  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      $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k))$ 
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  $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$ 

```

图 6 Graphsage 核心算法伪代码图

在直观地了解了工作流程后，我们看看它是如何体现在代码上的。首先看看代码框架：

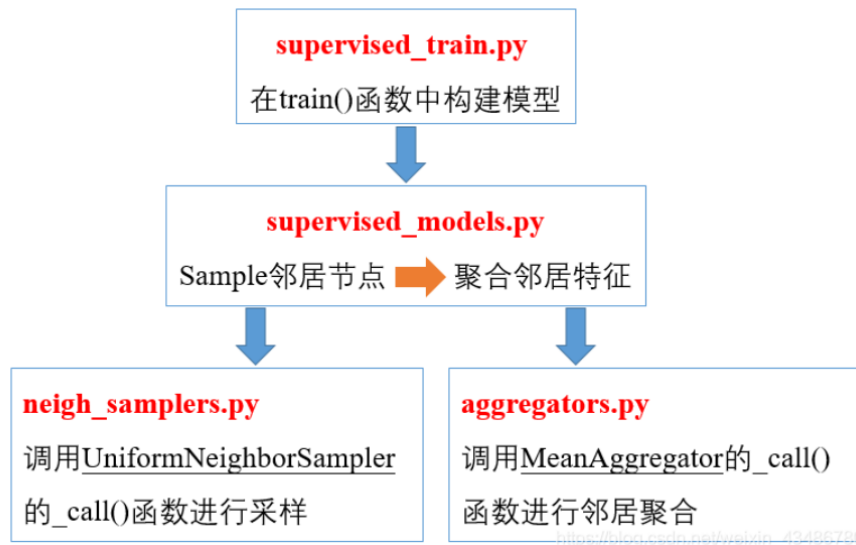


图 7 Graphsage 有监督学习部分代码框架图

程序主要调用的是 supervised\_train.py 文件中的 train 函数，在 train 函数中，首先会获得数据集的信息：

```

G = train_data[0]
features = train_data[1]
id_map = train_data[2]
class_map = train_data[4]
  
```

然后根据命令行选中的模型，进行不同的处理，此处以 graphsage\_mean 模型为例：

```

if FLAGS.model == 'graphsage_mean':
    # Create model
    sampler = UniformNeighborSampler(adj_info)
    .....
    model = SupervisedGraphsage(num_classes, placeholders,
                                features,
                                adj_info,
                                minibatch.deg,
                                layer_infos,
                                model_size=FLAGS.model_size,
                                sigmoid_loss = FLAGS.sigmoid,
                                identity_dim = FLAGS.identity_dim,
                                logging=True)
  
```

首先利用 adj\_info 参数构造了 UniformNeighborSampler 这一个类，并赋给 sample（成为一个对象），之后将会采用该类里面的 \_call 方法对邻居进行采样。

```

class UniformNeighborSampler(Layer):
    def __init__(self, adj_info, **kwargs):
        super(UniformNeighborSampler, self).__init__(**kwargs)
        self.adj_info = adj_info
  
```



```

def _call(self, inputs):
    ids, num_samples = inputs
    adj_lists = tf.nn.embedding_lookup(self.adj_info, ids)
    adj_lists = tf.transpose(tf.random_shuffle(tf.transpose(adj_lists)))
    adj_lists = tf.slice(adj_lists, [0,0], [-1, num_samples])
    return adj_lists

```

接着创建模型，利用 SupervisedGraphsage 这一个类创建一个对象 model，这个类包含三种方法：build、loss 和 predict。它正是有监督学习算法的核心类。

```

class SupervisedGraphsage(models.SampleAndAggregate):
    def __init__(self, num_classes,
                 placeholders, features, adj, degrees,
                 layer_infos, concat=True, aggregator_type="mean",
                 model_size="small", sigmoid_loss=False, identity_dim=0,**kwargs):
        .....
    def build(self):
        samples1, support_sizes1 = self.sample(self.inputs1, self.layer_infos)
        .....
        self.preds = self.predict()
    def _loss(self):
        # Weight decay loss
        for aggregator in self.aggregators:
            for var in aggregator.vars.values():
                .....
    def predict(self):
        if self.sigmoid_loss:
            return tf.nn.sigmoid(self.node_preds)
        else:
            return tf.nn.softmax(self.node_preds)

```

在得到了训练模型和 sample 之后，便需要初始化相关变量和任务然后开始训练了：

```

# Initialize session
sess = tf.Session(config=config)
merged = tf.summary.merge_all()
summary_writer = tf.summary.FileWriter(log_dir(), sess.graph)
# Init variables
sess.run(tf.global_variables_initializer(), feed_dict={adj_info_ph: minibatch.adj})

```

之后的过程便是涉及到具体的算法了，大致就是建立相应的字典，然后调用训练模型得到训练结果，再进行 validation，最后打印出学习到的模型中相应的参数，代码框架如下：

```

# Train model
for epoch in range(FLAGS.epochs):
    .....
    while not minibatch.end():

```

```

# Construct feed dictionary
.....
# Training step
.....
if iter % FLAGS.validate_iter == 0:
    # Validation
    .....
# Print results
if total_steps % FLAGS.print_every == 0:
    train_f1_mic, train_f1_mac = calc_f1(labels, outs[-1])
    print("Iter:", '%04d' % iter,.....

```

分析到这里我们发现实际上就有监督学习来说，类的数量还是比较少的，机器学习的重心还是在算法本身上，一般一种学习方法只需要单独设计一个类即可，不需要太多类之间的频繁交互。不过无论使用哪种学习方法，都需要采样，也都需要聚合，所以将必须会使用到 `sample` 以及 `aggregate` 对应的类。将这两个类封装好可以为程序的设计带来很大的便利。同时以上的几点也体现了模块设计的“高内聚、低耦合”原则，模块之间功能相互独立，不需要互相依赖，一个模块的错误不至于引发严重的连锁反应。

### 三、类的设计以及关联分析

上节主要针对算法流程进行分析，这一节就针对类的设计以及其间的关联进行分析。整体上整个项目文件的 UML 图为：（清晰的图见附件）

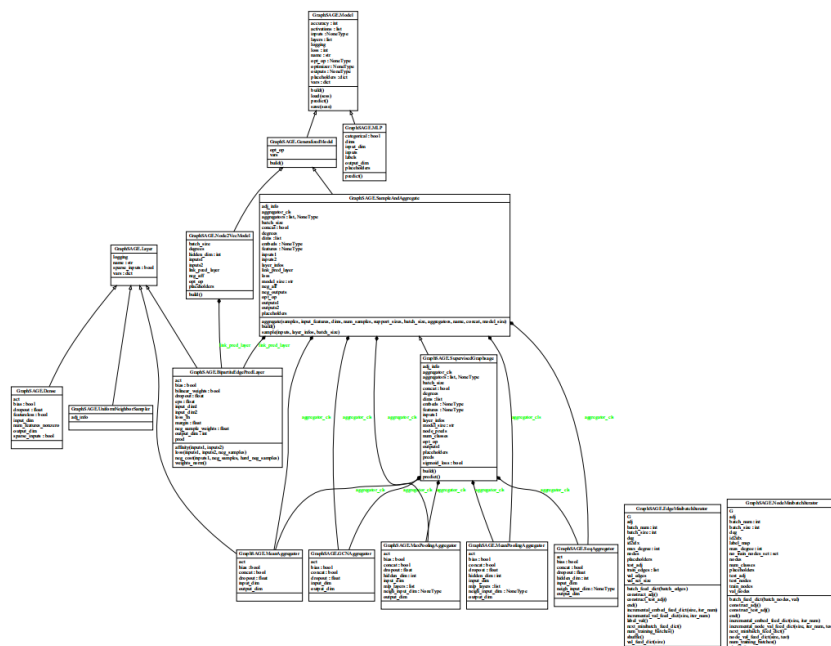


图 8 Graphsage 全项目 UML 图

从图中我们可以发现，整个项目的类虽然不少，但是联系却十分紧密，个别类之间是继承关系，而更多类之间是组合的关系。只有三个类保持与其它类相对独立的关系。组合关系占领首要地位说明设计遵循了组合/聚合复用原则，尽量使用组合/聚合达到复用而非继承，因此代码中更多的是直接复用现有的类来创建对象，而非再用继承关系形成很多子类，使框架更加复杂。

整体上，类之间的关系也体现出了算法的逻辑，比如 `supervisedGraphsage` 由 `MeanAggregator`、`GCNAggregator`、`MaxPoolingAggregator`、`MeanpoolingAggregator` 以及 `SeqAggregator` 组合而成，可以得到实际上有监督学习算法里面包含了五种聚合模式。其次，`supervisedGraphsage` 是继承 `SampleandAggregate` 类的，说明有监督学习也是一种先采样然后聚合得到模型的一种算法。

上图中大致存在两大块继承关系：

- 1.以 `Layer` 为父类，它包含了 `Dense`、`UniformNeighborSampler`、`MeanAggregator` 以及 `BipartiteEdgePredLayer` 为子类，整体上来说，`Layer` 是一个基础层类，它为所有的层的对象定义了基本的 API，相当于是其它所有类的一种基恩框架。
- 2.以 `Model` 为父类，包含四种方法 `build`、`load`、`predict` 和 `save`。它存在两个子类 `MLP` 和 `GeneralizedModel`，分别只拥有方法 `predict` 和 `build`，其中 `GeneralizedModel` 是 base class，是基础模型；而 `MLP` 是一个标准的多层感知器。`GeneralizedModel` 又存在两个子类，分别是 `Node2VecModel` 和 `SampleAndaggregate`，`SampleAndaggregate` 是 `graphsage` 无监督训练模型的基本实现，而 `Node2VecModel` 是一个转换器，是一个将节点转换成向量的模型。最后 `SampleAndaggregate` 存在一个子类 `supervisedGraphsage`，为有监督学习的训练和预测模型。

通过对每一个类的功能仔细分析我们可以发现，虽然类之间的关联性比较多，但是每一个类的功能都是相互独立的。因为就机器学习来说，每个类基本上都是为了一种功能而设计的，是按照算法流程来的，比如采样就是采样，采样后再将样本点进行聚合。设计的时候会避免出现边采样边聚合的情况，因为类的设计需要保证单一职责原则，采样与聚合同时进行将会导致模块间的耦合程度过大，违反了“高内聚，低耦合的原则”。

对于有监督训练模型，它的 UML 图如下：

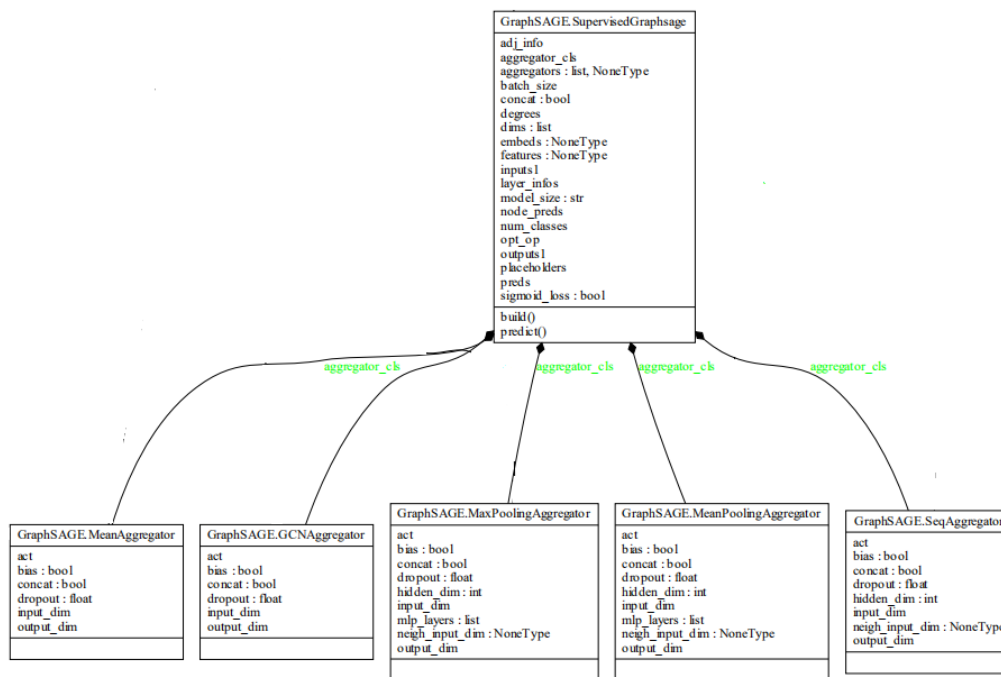


图9 Graphsage 有监督学习部分的 UML 图

它是 models.py 文件中 SampleAndaggregate 类的继承，可以看出它包含 build 以及 predict 两种方法，分别对应训练和预测。其内部包含了很多属性，分别是在算法过程中需要使用到的信息，此处就不再详细解释每个变量的作用了。从这里可以看出整个项目的设计还是很有条理性的，因为项目中会使用到很多模型，所以它就在最开始设计了一个总的模型 Model（相当是定义了一个模型的规范），然后在构造具体的模型的时候可以选择继承 Model 来表示是比它更加细化、功能更明确的一个模型。

## 四、高级意图设计分析

### 1. 工厂方法模式

**意图：** 定义一个用于创建对象的接口，让子类决定实例化哪一个类。

**主要作用：** 将类的实例化（具体产品的创建）延迟到工厂类的子类（具体工厂）中完成，即由子类来决定应该实例化（创建）哪一个类。

**应用场景：** 工厂需要频繁生产新产品的时候

**如何解决：** 采用工厂模式，利用 if-else 结构，添加新产品或者新算法的时候只需增加一层 else 的逻辑即可。

工厂方法模式在本项目中有两点典型的体现：

(1) 第一个是在有监督学习的 `train` 函数中，有一系列的 `if-else` 结构来针对命令行输入的聚合模型选择生成 `sample` 以及 `model`，代码框架如下：

```
if FLAGS.model == 'graphsage_mean':
    .....
elif FLAGS.model == 'gcn':
    .....
elif FLAGS.model == 'graphsage_seq':
    .....
elif FLAGS.model == 'graphsage_maxpool':
    .....
elif FLAGS.model == 'graphsage_meanpool':
    .....
else:
    raise Exception('Error: model name unrecognized.')
```

虽说这不是像 `java` 里典型的工厂方法模式，但是存在工厂方法模式的思想，针对不同的聚合方法，没有必要为每一种方法都专门设计一个类，这样就会使得类的数量过于多了，使结构更加复杂。

为了方便用户比对不同算法的运行结果并选取最好的算法建立模型，一般在设计的时候会设计多种聚合方法，尽管这是与 `GCN` 不同的 `graphsage` 算法，但还是在代码里保留了 `GCN` 的算法，为的就是将两种算法得到的结果进行对比。或者从另一个角度可以理解为，产品更新换代的时候并没有完全抛弃旧的版本，而是在旧版本的基础上再添加新版本。

这种设计模式非常方便算法的更新换代，一旦研究出某一种新的聚合方法，只需要添加一个新的 `else` 分支，然后再在这个分支下撰写此方法对应的代码即可（因为这些方法的初始化以及训练过程的流程都是一样的）。

(2) 第二个是在 `SupervisedGraphsage` 类里面：

```
if aggregator_type == "mean":
    self.aggregator_cls = MeanAggregator
elif aggregator_type == "seq":
    self.aggregator_cls = SeqAggregator
elif aggregator_type == "meanpool":
    self.aggregator_cls = MeanPoolingAggregator
elif aggregator_type == "maxpool":
    self.aggregator_cls = MaxPoolingAggregator
elif aggregator_type == "gcn":
    self.aggregator_cls = GCNAggregator
else:
```

```
raise Exception("Unknown aggregator: ", self.aggregator_cls)
```

此处相当于是根据创建对象时传入的聚合类型，为该类的 `aggregator_cls` 属性设置相应的值。之后再根据这个属性来确定具体需要创建哪一个用于聚合的类（`MeanAggregator`、`GCNAggregator`、`MeanPoolAggregator`、`MaxPoolAggregator` 和 `SeqAggregator` 中任选一个）。

## 2. 单例模式

**意图：** 保证一个类仅有一个实例，并提供一个访问它的全局访问点。

**主要解决：** 一个全局使用的类被频繁地创建与销毁。

**应用场景：** 需要控制实例数目、节省系统资源的时候。

**如何解决：** 只创建一个实例，且不销毁。

通过观察整个项目的类图我们可以发现，除了联系紧密的各个类之外，还有两个类是独立于他们之外的，即它们没有继承任何类，也没有组合关系。我们首先来观察他们的功能，`EdgeMinibatchIterator` 是一个小批迭代器，它在一批抽样边或随机对共现边上迭代；`NodeMinibatchIterator` 也是一个迭代器，它在节点上迭代以进行有监督学习。（UML 图如下）

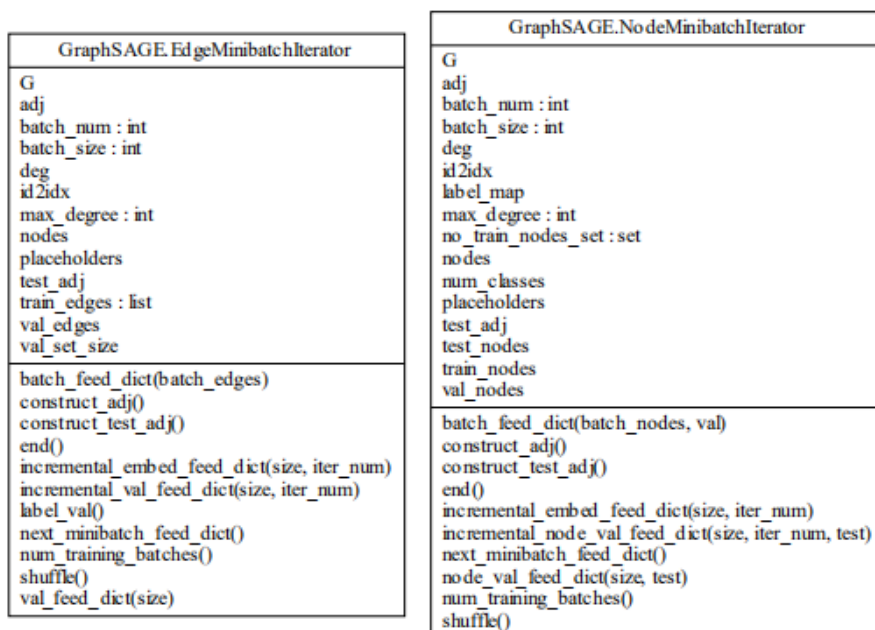


图9 Graphsage 两个迭代器的 UML 图

这两个迭代器的作用是在采样的过程中，分别用于处理图的边和节点信息的类，比如 `construct_adj` 方法是建立邻接表的方法，同时也会返回每个顶点的度。在采样的过程中如果顶点邻居不足那就抽样补满 128 个。

因为针对每一个输入的数据，都需要进行如上的操作，所以不如直接为它提供一个全局访问点，这种方法有点类似 java 里面的单例模式。因为毕竟 python 不是 java 语言，所以表现形式上难免会有不同，但是思想还是一样的。

```
class EdgeMinibatchIterator(object):  
    .....  
  
class NodeMinibatchIterator(object):  
    .....
```

这样的模式使得这些类不会被频繁地创建与销毁，当每次处理数据的时候直接使用这两个迭代器即可。

## 五、总结

整体上我感觉，其实深度学习一类的代码更侧重于算法实现逻辑，类与类之间的关系并没有一些大项目那么明显，而且类的方法基本脱离不开训练与预测这两种。不过话说回来，面向对象思想的应用可以使得模型的设计思路更加清晰，不仅有利于代码撰写，还有利于算法设计。对类进行封装还有利于维护代码的安全性，不至于用户修改到关键代码导致程序崩溃。因此，将一个本可以全部使用函数而实现的深度学习算法划分为一个一个类，并使用相互继承的关系实现整个算法流程是一种成熟的做法。往后对代码进行维护和优化的时候也会更加方便（可以对每一个类单独进行调试）。

其实我对 `graphsage` 的算法本身也挺感兴趣的，所以除了前文提到的一些优点之外，我还发现了本算法的一些缺点：

1. 首先它无法处理加权图，而只能从邻居处等权聚合特征，这算是一个比较大的局限性。
2. 其次该算法的采样引入了随机过程，推理阶段同一节点 `embedding` 特征不稳定，且邻居采样会导致反向传播时会带来较大梯度方差
3. 最后，采样数目的限制会导致部分节点的重要局部信息丢失

针对以上三个缺点，我想到一些改进的措施：

1. 为了处理加权图，在聚合之前需要将邻居特征归一化，也就是根据这条边的权重相应地修改邻居的特征向量，最后再进行特征融合
2. 为了处理 `embedding` 特征不稳定的问题，可以对搜索进行剪枝，思路类似于 KNN 算法。直接对原始网络进行剪枝操作，只保留每个节点权重最大的 `K` 条边。
3. 提前对每一个节点的特征与其所有邻居特征的均值进行合并，这样就可以使每一个节点初始状态下就拥有周围邻居节点的一些信息，通过该种方式在采样相同节点的前提下可获得更多的局部信息

因为本课程的侧重点不在代码的具体实现上，所以我也只能在总结出多发表一下自己对代码逻辑的看法。不过面向对象程序设计这门课使我意识到，有时候一个完整而合理的框架甚至比代码逻辑本身更加重要，尤其是进行大项目设计的时候。本学期进行完计算机体系结构实验课后这一点我深有体会，当代码量大的时候，代码直接的逻辑的清楚性显得尤为重要，一旦代码框架没有组织好，调 `bug` 的时候将会无从下手，十分痛苦。如果说设计一个项目是建一所楼房的话，那么面向对象思想架构就是地基，它有多牢固、多可靠决定了楼房能建多高。