第五章 图卷积网络

现有图神经网络皆基于邻居聚合的框架，即为每个目标节点通过聚合其邻居刻画结构信息，进而学习目标节点的表示。因此，在建模图神经网络时，研究人员的关注重点是如何在网络上构建聚合算子，聚合算子的目的是刻画节点的局部结构。现有的聚合算子构建分为两类：

- 谱域方法：利用图上卷积定理从谱域定义图卷积。
- 空间域方法：从节点域出发，通过在节点层面定义聚合函数来聚合每个中心节点和其邻近节点。主要包括空域图卷积神经网络、GraphSAGE和图注意力网络。

5.1 谱域图卷积神经网络

谱域图卷积神经网络主要包括谱卷积神经网络、切比雪夫网络和图卷积神经网络。我们会先从信号与系统的角度讲解谱图理论，从而得到最早的谱卷积神经网络，然后通过简化得到切比雪夫网络，最后进行再次简化得到图卷积神经网络。

5.1.1 谱图理论和图卷积

过去几年，卷积神经网络在图像领域大放异彩。卷积算子定义了加权和操作，其本身是聚合算子。借助于卷积神经网络对局部结构的建模能力以及网络数据上普遍存在的节点依赖关系，通过定义网络数据上的卷积算子进而设计图神经网络已经成为其中最活跃最重要的一支。但网络数据上平移不变性的缺失，给在节点域定义卷积算子带来困难。谱方法利用卷积定理从谱域定义卷积算子。 我们首先给出卷积定理的背景知识。

卷积的傅里叶变换

卷积定理：信号卷积的傅立叶变换等价于信号傅立叶变换的乘积

$$F(f*g)=F(f)\cdot F(g)$$

其中的 f,g 表示两个原始信号，$F(f)$ 表示 f 的傅立叶变换，$\cdot$ 表示乘积算子，$*$ 表示卷积算子。

对上面的公式做傅立叶逆变换，可以得到

$$f * g = F^{-1}(F(f) \cdot F(g))$$

其中 $F^{-1}(f)$ 表示信号 f 的傅立叶逆变换。

利用卷积定理，我们可以对谱空间的信号做乘法，再利用傅里叶逆变换将信号转换到原空间来实现图卷积，从而避免了图数据不满足平移不变性而造成的卷积定义困难问题。

图傅里叶变换

图傅立叶变换依赖于图上的拉普拉斯矩阵 L。对 L 做谱分解，我们可以得到

$$L = U \Lambda U^T$$

其中 $\Lambda = diag(\lambda_0, \ldots, \lambda_{N-1}) \in \mathbb{R}^{N \times N}$ 是特征值矩阵, $U = [u_0, \ldots, u_{N-1}] \in \mathbb{R}^{N \times N}$ 是对应的特征向量矩阵。如下图所示

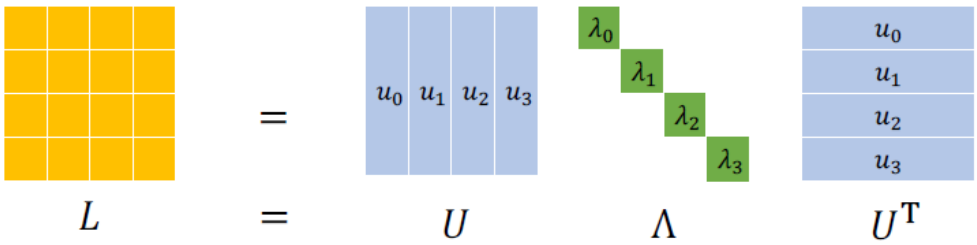

图5-1. 谱分解

图上傅立叶变换的定义依赖于拉普拉斯矩阵的特征向量。以特征向量作为谱空间下的一组基底，图上信号 x 的傅立叶变换为：

$$\hat{x} = U^T x$$

其中 x 指信号在节点域的原始表示。$\hat{x}$ 指信号 x 变换到谱域后的表示，U^T 表示特征向量矩阵的转置，用于做傅立叶变换。信号 x 的傅立叶逆变换为

$$x = U \hat{x}$$

图卷积

为了完成图上的卷积操作，我们要做的事情就是，**先将图进行傅里叶变化，在谱域完成卷积操作，然后再将频域信号转换回原域**。我们将卷积核定义为 $g_\theta(\Lambda)$，那么卷积的频域表示可以写为

先对输入 x （其输出为 y） 进行傅里叶变换得到 $U^T x$ （以及 $U^T y$），然后对其应用卷积核得到

$$\hat{y} = U^T y = g_\theta(\Lambda) U^T x$$

最后，再利用傅里叶逆变换得到

$$y = UU^{T}y = Ug_{\theta}(\Lambda)U^{T}x$$

以上就是最重要的图卷积操作。它一般可以被简化为

$$y = g_{\theta}(U\Lambda U^{T})x = g_{\theta}(L)x$$

以上过程可以被表示为下图

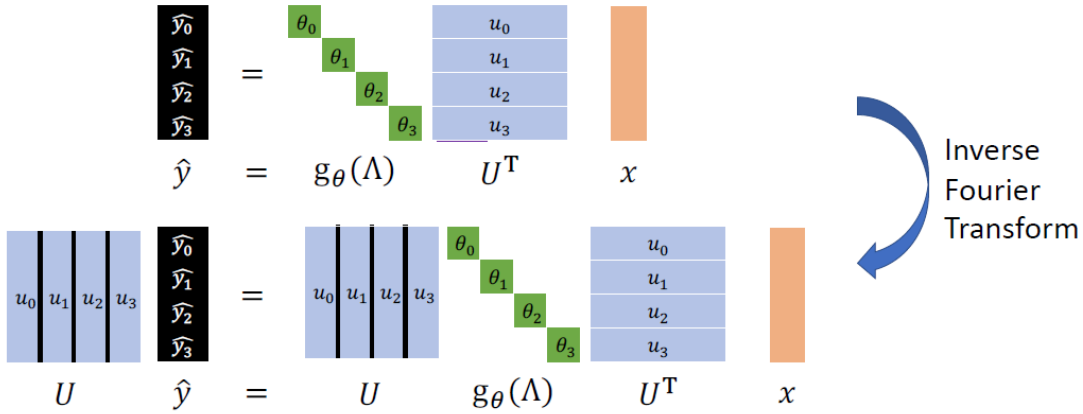

图5-2. 图卷积

基于此卷积算子的定义，国内外陆续涌现出一些基于卷积聚合的图神经网络。

5.1.2 谱卷积神经网络

谱卷积神经网络（Spectral Convolutional Neural Network）是最早提出在网络数据上构建图神经网络的方法，该方法完全按照上述得到的卷积操作，堆叠多层得到。我们可以将上面公式的 x 和 y 替换为图上的节点特征 $H^{(l)}$ 和 $H^{(l+1)}$，那么第 l 层的结构如下

$$H^{(l+1)} = \sigma(g_{\theta}(U\Lambda U^{T})H^{(l)})$$

σ 表示非线性激活函数。

5.1.3 切比雪夫网络

谱卷积神经网络基于全图的傅里叶卷积来实现图的卷积，其缺点非常明显，难以从卷积形式中保证节点的信息更新由其邻居节点贡献，因此无法保证局部性。另外，谱卷积神经网络的计算复杂度比较大，难以扩展到大型图网络结构中。切比雪夫网络 (ChebyNet)，采用切比雪夫多项式替代了谱卷积神经网络的卷积核，有效的解决了上述的问题。

g_θ 是需要学的卷积核，在谱卷积神经网络中，g_θ 对角阵的形式，且有 n 个需要学的参数。切比雪夫网络 (ChebyNet) 对卷积核 g_θ 进行参数化

$$g_\theta = \sum_{i=0}^{K-1} \theta_k T_k(\hat{\Lambda})$$

其中 θ_k 是需要学的系数，并定义 $\hat{\Lambda} = \frac{2\Lambda}{\lambda_{\max}} - I_n$。切比雪夫多项式是可以通过递归求解，递归表达式为

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$$

其中初始值 $T_0(x) = 1, T_1(x) = x$。

令 $\hat{L} = \frac{2L}{\lambda_{\max}} - I_n$，切比雪夫网络第 l 层的结构定义如下：

$$\begin{aligned} H^{(l+1)} &= \sigma\left(U\left(\sum_{k=0}^{K-1}\theta_k T_k(\hat{\Lambda})\right)U^\top H^{(l)}\right) \\ &= \sigma\left(\sum_{k=0}^{K-1}\theta_k T_k(\hat{L})H^{(l)}\right) \end{aligned}$$

切比雪夫网络利用特征值矩阵的多项式参数化卷积核，实现谱卷积神经网络，且巧妙的利用 $L = U\Lambda U^T$ 引入拉普拉斯矩阵，从而避免了拉普拉斯矩阵的特征分解，同时参数复杂度从 $O(n \times p \times q)$ 下降到 $O(K \times p \times q)$。此外，在拉普拉斯矩阵中，当且仅当两个节点满足 K 跳可达时，其拉普拉斯矩阵中这一项不为 0，这一性质使得当 K 较小时，切比雪夫网络具有局部性。

5.1.3 图卷积神经网络

图卷积神经网络（Graph Convolutional Network, GCN）对切比雪夫网络进行了简化，只取 0 阶和 1 阶，形式如下，

$$y = g_\theta(L)x = \sum_{i=0}^{1}\theta_k T_k(\hat{\Lambda}) = \theta_0 x + \theta_1 \hat{L}x$$

代入定义 $\hat{L} = \frac{2L}{\lambda_{max}} - I$，且 $\lambda_{max} = 2$ 我们可以得到

$$y = \theta_0 x + \theta_1(L - I)x$$

此时的拉普拉斯矩阵 L 为标准化后的拉普拉斯矩阵满足 $L = I - D^{-1/2}AD^{-1/2}$。然后我们令 $\theta = \theta_0 = -\theta_1$可以得到，

$$y = \theta x - \theta(D^{-1/2}AD^{-1/2})x = \theta(I + D^{-1/2}AD^{-1/2})x$$

然后应用一个重新标准化的trick: $I + D^{-1/2}AD^{-1/2} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$，我们可以得到

$$y = \theta(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2})x$$

在应用于图神经网络的第 l 层时，我们令 θ 为 W，就可以得到和GCN 原论文中类似的公式

$$H^{(l+1)} = \sigma(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(l)}W^{(l)})$$

5.1.4 图卷积神经网络代码

我们将使用 PyTorch Geometric (PyG) 框架学习 GCN，应用于图分类任务。 图分类是指在给定图数据集的情况下，根据某些结构图属性对整个图进行分类的问题。

图分类最常见的任务是分子属性预测，其中分子被表示为图，任务可能是推断分子是否抑制HIV病毒复制。

多特蒙德工业大学收集了各种不同的图形分类数据集，称为 TUDatasets，可以通过 PyTorch Geometric 中的 torch_geometric.datasets.TUDataset 访问。 让我们加载并检查较小的数据集之一，即 MUTAG 数据集：

```python
import torch
from torch_geometric.datasets import TUDataset

dataset = TUDataset(root='data/TUDataset', name='MUTAG') # 加载数

print()
print(f'Dataset: {dataset}:')
print('====================')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')

data = dataset[0]  # 得到数据中的第一个图

print()
print(data)
print('=============================================================')

# 获得图的一些统计特征
print(f'Number of nodes: {data.num_nodes}')
```

```
print(f'Number of edges: {data.num_edges}')
print(f'Average node degree: {data.num_edges / data.num_nodes:.2
print(f'Has isolated nodes: {data.has_isolated_nodes()}')
print(f'Has self-loops: {data.has_self_loops()}')
print(f'Is undirected: {data.is_undirected()}')
```

```
Downloading https://www.chrsmrrs.com/graphkerneldatasets/MUTAG.z
Extracting data/TUDataset/MUTAG/MUTAG.zip
Processing...

Dataset: MUTAG(188):
====================
Number of graphs: 188
Number of features: 7
Number of classes: 2

Data(edge_index=[2, 38], x=[17, 7], edge_attr=[38, 4], y=[1])
=============================================================
Number of nodes: 17
Number of edges: 38
Average node degree: 2.24
Has isolated nodes: False
Has self-loops: False
Is undirected: True
Done!
```

该数据集提供了 `188` 个不同的图，任务是将每个图分类为两类中的一类。通过检查数据集的第一个图对象，我们可以看到它有 `17` 个节点（具有 `7 维特征向量）和 38 条边（平均节点度为 2.24`）`，它还有一个标签 (y=[1])。除了之前的数据集之外，还提供了额外的 4 维边缘特征 (edge_attr=[38, 4])。 然而，为了简单起见，我们这次不会使用它们。

PyTorch Geometric 提供了一些有用的实用程序来处理图数据集，例如，我们可以打乱数据集并使用前 150 个图作为训练图，同时使用剩余的图形进行测试：

Python

```
torch.manual_seed(12345)
dataset = dataset.shuffle()

train_dataset = dataset[:150]
```

```
test_dataset = dataset[150:]

print(f'Number of training graphs: {len(train_dataset)}')
print(f'Number of test graphs: {len(test_dataset)}')
```

```
Number of training graphs: 150
Number of test graphs: 38
```

批处理对于图数据比较复杂和麻烦。PyTorch Geometric 选择了一种和常见图像数据集不同的方法来实现多个示例的并行化。 在这里，邻接矩阵以对角方式堆叠（创建一个包含多个孤立子图的巨型图），并且节点和目标特征在节点维度中简单地连接。与其他批处理程序相比，该程序具有一些关键优势：（1）依赖于消息传递方案的 GNN 算子不需要修改，因为属于不同图的两个节点之间不会交换消息；（2）由于邻接矩阵以稀疏方式保存，仅保存非零条目（即边），因此不存在计算或内存开销。

PyTorch Geometric 在 torch_geometric.data.DataLoader 类的帮助下自动将多个图批处理为单个巨型图，我们并不需要手动进行上述的复杂步骤。

```python
from torch_geometric.loader import DataLoader

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=Fa
```

在这里，我们选择 batch_size 为 64，从而产生 3 个（随机洗牌）小批量，包含所有 2·64+22=150 个图。

训练 GNN 进行图分类通常遵循一个简单的方法：

- 通过执行多轮消息传递来嵌入每个节点。
- 将节点嵌入聚合为统一的图嵌入（读出层）。
- 在图嵌入上训练最终分类器。

对于整图分类，我们需要一个读出层（readout layer），但最常见的一种是简单地取节点嵌入的平均值：

$$x_{out} = \frac{1}{|V|} \sum_{v \in V} x_v^{(L)}$$

PyTorch Geometric 通过 `torch_geometric.nn.global_mean_pool` 提供此功能，它接受小批量中所有节点的节点嵌入和分配向量批量，以计算批量中每个图的大小为 `[batch_size, hide_channels]` 的图嵌入。也就是说，我们在这里不需要考虑批大小。

将 GNN 应用到图分类任务的最终架构如下所示，并允许**完整的端到端训练**：

python

```python
from torch.nn import Linear
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.nn import global_mean_pool


class GCN(torch.nn.Module):
    def __init__(self, hidden_channels):
        super(GCN, self).__init__()
        torch.manual_seed(12345)
        # 使用 GCNConv
        # 为了让模型更稳定我们也可以使用带有跳跃链接的 GraphConv
        # from torch_geometric.nn import GraphConv
        self.conv1 = GCNConv(dataset.num_node_features, hidden_c
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.conv3 = GCNConv(hidden_channels, hidden_channels)
        self.lin = Linear(hidden_channels, dataset.num_classes)

    def forward(self, x, edge_index, batch):
        # 1. 获得节点的嵌入
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = self.conv2(x, edge_index)
        x = x.relu()
        x = self.conv3(x, edge_index)

        # 2. 读出层
        x = global_mean_pool(x, batch)  # [batch_size, hidden_ch

        # 3. 应用最后的分类器
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.lin(x)

        return x
```

```
model = GCN(hidden_channels=64)
print(model)
```

```
GCN(
  (conv1): GCNConv(7, 64)
  (conv2): GCNConv(64, 64)
  (conv3): GCNConv(64, 64)
  (lin): Linear(in_features=64, out_features=2, bias=True)
)
```

在这里，我们再次使用 `GCNConv` 和 `ReLU(x)=max(x,0)` 激活来获得局部节点嵌入，然后再将最终分类器应用到图读出层之上。

让我们训练我们的网络几个周期，看看它在训练和测试集上的表现如何：

python

```
from IPython.display import Javascript
display(Javascript('''google.colab.output.setIframeHeight(0, tru

model = GCN(hidden_channels=64)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()

def train():
    model.train()

    for data in train_loader:  # 迭代获得各个批数据
         out = model(data.x, data.edge_index, data.batch)  # 前向
         loss = criterion(out, data.y)  # 计算损失
         loss.backward()  # 反向传播
         optimizer.step()  # 参数更新
         optimizer.zero_grad()  # 梯度清零

def test(loader):
     model.eval()

     correct = 0
     for data in loader:  # 迭代获得各个批数据
         out = model(data.x, data.edge_index, data.batch)
         pred = out.argmax(dim=1)  # 取最大概率的类作为预测
```

```
            correct += int((pred == data.y).sum())  # 与真实标签做比较
        return correct / len(loader.dataset)  # 计算准确率


for epoch in range(1, 171):
    train()
    train_acc = test(train_loader)
    test_acc = test(test_loader)
    print(f'Epoch: {epoch:03d}, Train Acc: {train_acc:.4f}, Test
```

```
Epoch: 001, Train Acc: 0.6467, Test Acc: 0.7368
Epoch: 002, Train Acc: 0.6467, Test Acc: 0.7368
...
Epoch: 169, Train Acc: 0.8000, Test Acc: 0.7632
Epoch: 170, Train Acc: 0.8000, Test Acc: 0.7632
```

可以看到，我们的模型达到了 `76%` 左右的测试准确率。 准确率波动的原因可以用相当小的数据集（只有 `38` 个测试图）来解释，并且一旦将 GNN 应用到更大的数据集，通常就会消失。

5.2 空间域图卷积神经网络

频域方法中，采用傅里叶卷积定理。空域卷积，则是从邻居节点信息聚合的角度出发，更加注重节点的局域环境。

5.2.1 图卷积神经网络的空域理解

如果我们从空域的角度出发，即不考虑严格的卷积定义，而是从邻居节点信息聚合的角度出发。图卷积神经网络，应该做的如下两件事情

- 对节点的信息进行转换（Message Transformation）
- 对节点信息进行聚合 （Message Aggregation）

根据上面这两个点，那么 `GCN` 要做的事情可以表示为如下公式

$$h_v^{(l+1)} = \sigma(W^l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + h_v^{(l)} B^{(l)})$$

其中第一项 $W^l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|}$ 表示邻居节点信息的转换和聚合，第二项 $h_v^{(l)} B^{(l)}$ 表示自身节点信息的变换。

现在我们将上面的公式进行矩阵化。我们知道度矩阵 $D_{v,v} = Deg(v) = |N(v)|$，因此 $D_{v,v}^{-1} = 1/|N(v)|$，所以我们可以将 $\sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|}$ 转化为 $D^{-1}AH^l$。更近一步，上述的公式的矩阵表达可以写为

$$H^{(l+1)} = \sigma(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(l)}W^{(l)} + H^{(l)}B^{(l)})$$

这个时候，如果我们认为 W 也包含自身节点的操作的话，我们就可以得到和谱域 `GCN` 完全一样的公式了。这里，我们分开表达他们是为了强调，一定要对邻居和自身节点都做信息变换和聚合。

5.2.3 空域图卷积的统一范式和 GraphSAGE

图卷积的统一范式

从空域图卷积神经网络必须做的两件事情出发，我们可以得到一个统一范式的图卷积网络。

- 对节点的信息进行转换（Message Transformation）
- 对节点信息进行聚合 （Message Aggregation）

下面的公式就是空域图卷积的统一范式：

$$h_v^{(l+1)} = AGG_2^l(AGG_1^l(\{TRANS_u^{(l)}(h_u^{(l)}), u \in N(v)\}), TRANS_v^{(l)}(h_u^{(l)}))$$

其中 $TRANS_u^{(l)}$ 表示对邻居节点信息的转换，$TRANS_v^{(l)}$ 表示对自身节点信息的转换，AGG_1^l 表示对邻居节点信息的聚合，AGG_2^l 表示对自身节点信息的聚合。

在这个统一范式下，在介绍 `GraphSAGE` 之前，我们简单介绍一下我们将会在后面章节学习的图注意力网络（Graph Attention Network, GAT）。我们可以看出来 `GCN` 在进行聚合的时候是没有考虑边的权重的而当作 `1` 进行简单的加和。`GAT` 的目的就是通过网络来学习边的权重，然后把学到的权重用于聚合。具体地，我们将边两端得节点送入一个网络，学习输出得到这条边得权重。

GraphSAGE

GraphSAGE 是 **SA**mple aggre**G**at**E** for **Graph**。其实上面的空域图卷积的统一视角也是 GraphSAGE 的作者提出来的。此外，它从两个方面对传统的 GCN 做了改进：

1. 在训练时，采样方式将 GCN 的全图采样优化到部分以节点为中心的邻居抽样，这使得大规模图数据的分布式训练成为可能，并且使得网络可以学习没有见过的节点，这也使得 GraphSAGE 可以做 Inductive Learning。
2. GraphSAGE 研究了若干种邻居聚合的方式，及其 AGG 聚合函数可以使用
   - 平均
   - Max Pooling
   - LSTM

在 GraphSAGE 之前的 GCN 模型中，都是采用的全图的训练方式，也就是说每一轮的迭代都要对全图的节点进行更新，当图的规模很大时，这种训练方式无疑是很耗时甚至无法更新的。mini-batch 的训练时深度学习一个非常重要的特点，那么能否将 mini-batch 的思想用到 GraphSAGE 中呢，GraphSAGE 提出了一个解决方案。它的流程大致分为3步：

1. 对邻居进行随机采样，每一跳抽样的邻居数不多于 S_k 个；
2. 生成目标节点的 embedding：先聚合二跳邻居的特征，生成一跳邻居的 embedding，再聚合一跳的 embedding，生成目标节点的 embedding；
3. 将目标节点的 embedding 输入全连接网络得到目标节点的预测值。

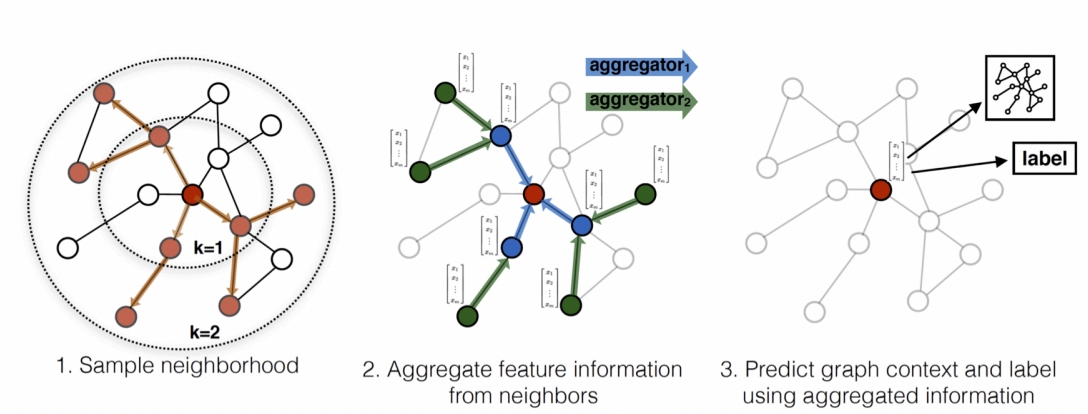

图5-3. GraphSAGE 算法流程（包含采样和聚合）

下面是原论文的算法流程：

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 $v \in \mathcal{V}$

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

图5-4. GraphSAGE 前向传播的算法流程

5.2.4 GraphSAGE 代码

下面我们介绍一下 GraphSAGE 的代码实现，使用的是 DGL 框架（这里我们又引入 DGL 这个框架是因为 PyG 和 DGL 现在都被广泛使用，读者应当对这两个框架都有所了解）。我们用 link prediction 作为模型的任务来举例。我们先简单的介绍一下链接预测这个任务。许多应用，如社交推荐、项目推荐、知识图谱补全等，都可以表述为链接预测，即预测两个特定节点之间是否存在边。

```python
# 导入相关的库
import dgl
import torch
import torch.nn as nn
import torch.nn.functional as F
import itertools
import numpy as np
import scipy.sparse as sp
```

```python
# 导入Cora数据集
import dgl.data

dataset = dgl.data.CoraGraphDataset()
g = dataset[0]
```

```python
# 准备training set 和 testing set
u, v = g.edges()

eids = np.arange(g.number_of_edges())
eids = np.random.permutation(eids)
test_size = int(len(eids) * 0.1)
train_size = g.number_of_edges() - test_size
test_pos_u, test_pos_v = u[eids[:test_size]], v[eids[:test_size]]
train_pos_u, train_pos_v = u[eids[test_size:]], v[eids[test_size

# 分离负样本
adj = sp.coo_matrix((np.ones(len(u)), (u.numpy(), v.numpy())))
adj_neg = 1 - adj.todense() - np.eye(g.number_of_nodes())
neg_u, neg_v = np.where(adj_neg != 0)
```

```
neg_eids = np.random.choice(len(neg_u), g.number_of_edges())
test_neg_u, test_neg_v = neg_u[neg_eids[:test_size]], neg_v[neg_
train_neg_u, train_neg_v = neg_u[neg_eids[test_size:]], neg_v[ne
```

训练时，您需要从原始图中删除测试集中的边。您可以通过 `dgl.remove_edges` 来完成此操作。 `dgl.remove_edges` 的工作原理是从原始图创建子图，从而生成副本，因此对于大型图来说可能会很慢。如果是这样，您可以将训练和测试图保存到磁盘，就像预处理一样。

python

```
train_g = dgl.remove_edges(g, eids[:test_size])
```

下面我们正式定义一个GraphSAGE模型：

python

```
from dgl.nn import SAGEConv

# 构建一个两层的 GraphSAGE 模型
class GraphSAGE(nn.Module):
    def __init__(self, in_feats, h_feats):
        super(GraphSAGE, self).__init__()
        self.conv1 = SAGEConv(in_feats, h_feats, 'mean')
        self.conv2 = SAGEConv(h_feats, h_feats, 'mean')

    def forward(self, g, in_feat):
        h = self.conv1(g, in_feat)
        h = F.relu(h)
        h = self.conv2(g, h)
        return h
```

然后，该模型通过计算两个节点的表示之间的得分来预测边缘存在的概率，通常通过一层MLP或者直接计算点积。

python

```
# 构建正样本和负样本的图
train_pos_g = dgl.graph((train_pos_u, train_pos_v), num_nodes=g.
train_neg_g = dgl.graph((train_neg_u, train_neg_v), num_nodes=g.
```

```
test_pos_g = dgl.graph((test_pos_u, test_pos_v), num_nodes=g.num
test_neg_g = dgl.graph((test_neg_u, test_neg_v), num_nodes=g.num
```

构建上面提到的预测函数，如点积和MLP，即 DotPredictor 和 MLPPredictor:

python

```python
import dgl.function as fn

class DotPredictor(nn.Module):
    def forward(self, g, h):
        with g.local_scope():
            g.ndata['h'] = h
            # 通过点积计算一个新的边的分数
            g.apply_edges(fn.u_dot_v('h', 'h', 'score'))
            # u_dot_v 返回了一个 1-element 的向量，所以需要压平它
            return g.edata['score'][:, 0]

class MLPPredictor(nn.Module):
    def __init__(self, h_feats):
        super().__init__()
        self.W1 = nn.Linear(h_feats * 2, h_feats)
        self.W2 = nn.Linear(h_feats, 1)

    def apply_edges(self, edges):
        """
        Computes a scalar score for each edge of the given graph

        Parameters
        ----------
        edges :
            Has three members ``src``, ``dst`` and ``data``, eac
            which is a dictionary representing the features of t
            source nodes, the destination nodes, and the edges
            themselves.

        Returns
        -------
        dict
            A dictionary of new edge features.
        """
        h = torch.cat([edges.src['h'], edges.dst['h']], 1)
        return {'score': self.W2(F.relu(self.W1(h))).squeeze(1)}
```

```
    def forward(self, g, h):
        with g.local_scope():
            g.ndata['h'] = h
            g.apply_edges(self.apply_edges)
            return g.edata['score']
```

下面展示整个任务的训练过程

python

```
model = GraphSAGE(train_g.ndata['feat'].shape[1], 16)
# You can replace DotPredictor with MLPPredictor.
# pred = MLPPredictor(16)
pred = DotPredictor()

def compute_loss(pos_score, neg_score):
    scores = torch.cat([pos_score, neg_score])
    labels = torch.cat([torch.ones(pos_score.shape[0]), torch.ze
    return F.binary_cross_entropy_with_logits(scores, labels)

from sklearn.metrics import roc_auc_score
def compute_auc(pos_score, neg_score):
    scores = torch.cat([pos_score, neg_score]).numpy()
    labels = torch.cat(
        [torch.ones(pos_score.shape[0]), torch.zeros(neg_score.s
    return roc_auc_score(labels, scores)
```

python

```
optimizer = torch.optim.Adam(itertools.chain(model.parameters(),

# 训练
all_logits = []
for e in range(100):
    # 前向传播
    h = model(train_g, train_g.ndata['feat'])
    pos_score = pred(train_pos_g, h)
    neg_score = pred(train_neg_g, h)
    loss = compute_loss(pos_score, neg_score)

    # 更新参数
```

```
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if e % 5 == 0:
            print('In epoch {}, loss: {}'.format(e, loss))

    # 计算AUC
    with torch.no_grad():
        pos_score = pred(test_pos_g, h)
        neg_score = pred(test_neg_g, h)
        print('AUC', compute_auc(pos_score, neg_score))
```

5.3 参考文献

[1] Cheng-han Jiang, Hung-yi Lee (李宏毅), 机器学习：补充资料 Graph Neural Network

[2] Jure Leskovec, Stanford University CS224W: Machine Learning with Graphs

[3] PyG 官方教程

[4] DGL 官方教程 - Link Prediction using Graph Neural Networks (GraphSAGE)

上一篇 第四章 图表示学习

下一篇 第六章 关系图卷积神经网络