DGL 中文手册

目录

begin 安装.....4
使用 conda 安装.....4
使用 pip 安装.....4
使用源安装.....4
设置默认的 Backend.....4
1. 图.....4
1.1 关于图的一些基本定义.....4
1.2 图、节点和边.....5
1.3 节点和边特征.....6
1.4 从外部来源创建图.....7
1.4.1 从外部库创建.....7
1.4.2 从磁盘创建.....8
1.5 异构图.....9
1.5.1 创建一个异构图.....9
1.5.2 处理多种类型.....11
1.5.3 从磁盘建立异构图.....12
1.5.4 边类型子图.....12
1.5.5 将异构图转换为同构图.....13
1.6 在 GPU 上使用 DGLGraph.....14
2. 消息传递.....15
2.1 消息传递案例.....15
2.2 内置功函数和消息传递 APIs.....15
2.3 编写有效的消息传递代码.....17
2.4 在图的一部分上应用消息传递.....18
2.5 在消息传递中使用边权重.....18
2.6 在异构图中的消息传递.....18
3. 建立 GNN 模块.....19
3.1 DGL 中 NN Module 的构造函数.....19
3.2 GDL 中 NN Module 前向函数.....20
3.2.1 图检查和图类型规范化.....21
3.2.2 消息传递和归约.....22
3.2.3 归约得到输出后更新特征.....22
3.3 异构 GraphConv 模块.....23
3.3.1 HeteroGraphConv 实现逻.....23
4. 图数据管道.....24
4.1 DGLDataset 类.....24
4.2 下载原始数据（可选）.....27
4.3 处理数据.....28
4.3.1 处理图分类数据集.....28
4.3.2 处理节点分类数据集.....30

4.3.3 处理链接预测数据集.....31
4.4 保存和加载数据.....33
4.5 使用 ogb 包加载 OGB 数据集.....34
5. 训练图神经网络.....35
总览.....35
异构图.....35
5.1 节点分类/回归.....36
5.1.1 总览.....37
5.1.2 编写神经网络模型.....37
5.1.3 训练循环.....37
5.1.4 异构图.....38
5.2 边分类/回归.....40
5.2.1 总览.....40
5.2.2 模型实现和节点分类的区别.....40
5.2.3 训练循环.....41
5.2.4 异构图.....42
5.2.5 预测异构图上现有边的边类型.....43
5.3 链接预测.....45
5.3.1 总览.....45
5.3.2 与边分类模型实现的区别.....45
5.3.3 训练循环.....45
5.4 图分类.....46
5.4.1 总览.....46
5.4.2 图批量.....47
5.4.3 图读出.....48
5.4.4 编写神经网络模型.....48
5.4.5 训练循环.....49
5.4.6 异构图.....50
6. 大图的随机训练.....51
6.1 通过邻域采样训练 GNN 以进行节点分类.....52
6.1.1 定义邻域采样器和数据加载器.....52
6.1.2 为 minibatch 训练调整你的模型.....53
6.1.3 训练循环.....53
6.1.4 对于异构图.....54
6.2 训练利用邻域采样进行边分类的 GNN.....55
6.2.1 定义邻域采样器和数据加载器.....55
6.2.2 从原始图中移除 minibatch 中的边以进行邻域采样.....56
6.2.3 调整模型以进行 minibatch 训练.....56
6.2.4 训练循环.....57
6.2.5 对于异构图.....58
6.3 通过邻域采样训练 GNN 进行链路预测.....60
6.3.1 使用负采样定义邻域采样器和数据加载器.....60
6.3.2 调整模型以进行 minibatch 训练.....61
6.3.3 训练循环.....62

6.3.4 对于异构图.....62
6.4 自定义领域采样器.....64
6.4.1 用 pencil and paper 进行邻域采样.....65
6.4.2 查找消息传递依赖项.....65
6.4.3 多层 minibatch 消息传递的双向结构.....67
6.4.4 块在异构图上工作.....68
6.4.5 实现自定义邻域采样器.....69
6.4.6 为异构图生成边界.....71
6.5 实现自定义 GNN 模块进行 minibatch 训练.....71
6.5.1 异构图.....72
6.5.2 编写可用于同构图，二部图和块的模块.....74
6.6 对大型图进行精确离线推断.....74
6.6.1 实现离线推理.....75
7. 分布式训练.....76
7.1 分布式训练的预处理.....78
7.1.1 负载均衡.....79
7.2 分布式 APIs.....79
7.2.1 DGL 分布式模块的初始化.....79
7.2.2 分布式图.....80
7.2.3 分布式张量.....81
7.2.4 分布式嵌入.....81
7.2.5 分布式采样.....82
7.2.6 分割工作量.....83
7.3 用于启动分布式训练/推理的工具.....83

begin 安装

安装的方式：pip、conda
安装的平台：Ubuntu 16.04、macOS X、Windows 10
安装的 Python 版本；>=3.5
当前版本：0.3

使用 conda 安装

```
conda install -c dglteam dgl             # For CPU Build
conda install -c dglteam dgl-cuda9.0     # For CUDA 9.0 Build
conda install -c dglteam dgl-cuda10.0    # For CUDA 10.0 Build
conda install -c dglteam dgl-cuda10.1    # For CUDA 10.1 Build
conda install -c dglteam dgl-cuda10.2    # For CUDA 10.2 Build
```

使用 pip 安装

```
pip install dgl          # For CPU Build
pip install dgl-cu90     # For CUDA 9.0 Build
pip install dgl-cu92     # For CUDA 9.2 Build
pip install dgl-cu100    # For CUDA 10.0 Build
pip install dgl-cu101    # For CUDA 10.1 Build
pip install dgl-cu102    # For CUDA 10.2 Build
```

使用源安装

略

设置默认的 Backend

```
python -m dgl.backend.set_default_backend [BACKEND]
```

backend 可选：pytorch、tensorflow、mxnet

1. 图

图是由节点和边构成的，节点和边都可以有不同的类型（比如用户节点和商品节点是两种不同的节点类型）。DGL 提供了一个核心数据结构-- graph-centric（以图为中心的）编程抽象。DGLGraph 提供了处理图结构、节点和边特征、使用组件进行结果计算的接口。

1.1 关于图的一些基本定义

图的定义：G=(V,E)，即图是由节点和节点之间的关系-边构成。V 是节点的集合，E 是边的集合。

图可以是有向的，也可以是无向的。

图的边是可以带权重的。

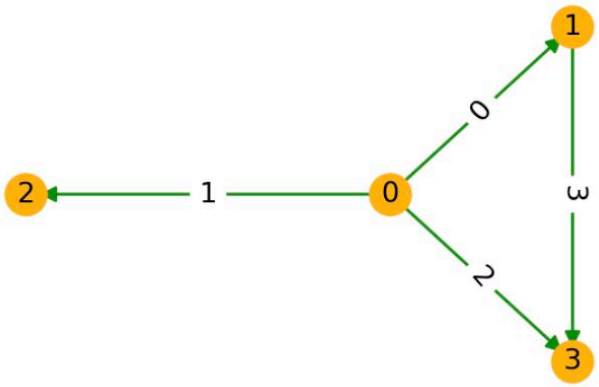
图可以是同构的，也可以是异构的。同构指的是所有图节点是同一种类型的，所有的边表示的关系是同种类型的，比如一个社交网络，节点是人，边表示关联关系。与此对应的异构图指的是节点和边可以是不同的类型。比如，一个市场会

有买家、卖家和产品节点，有想要买、已经买了、正在买、正在卖等边关系。

多图是在同一对节点（包括自环）之间可以具有多个（有向）边的图。例如，两位作者可以在不同的年份共同撰写论文，从而导致边具有不同的特征。

1.2 图、节点和边

DGL 使用唯一的整型（节点 ID）来表示每一个节点，使用唯一的整型（边 ID）表示每一条边。(U[i], V[i])表示从节点 U 到节点 V 的边。可以从 dgl.graph()构建一个 DGLGraph，或者通过其它数据结构来构建。



以下代码展示的是构建上图中的图结构：

```
>>> import dgl
>>> import torch as th

>>> # edges 0->1, 0->2, 0->3, 1->3
>>> u, v = th.tensor([0, 0, 0, 1]), th.tensor([1, 2, 3, 3])
>>> g = dgl.graph((u, v))
>>> print(g) # number of nodes are inferred from the max node IDs in the given edges
Graph(num_nodes=4, num_edges=4,
      ndata_schemes={}
      edata_schemes={})

>>> # Node IDs
>>> print(g.nodes())
tensor([0, 1, 2, 3])
>>> # Edge end nodes
>>> print(g.edges())
(tensor([0, 0, 0, 1]), tensor([1, 2, 3, 3]))
>>> # Edge end nodes and edge IDs
>>> print(g.edges(form='all'))
(tensor([0, 0, 0, 1]), tensor([1, 2, 3, 3]), tensor([0, 1, 2, 3]))

>>> # If the node with the largest ID is isolated (meaning no edges),
>>> # then one needs to explicitly set the number of nodes
>>> g = dgl.graph((u, v), num_nodes=8)
```

对于无向图而言，需要为两个方向都创建边，可以使用 dgl.to_bidirected()：

```
>>> bg = dgl.to_bidirected(g)
>>> bg.edges()
(tensor([0, 0, 0, 1, 1, 2, 3, 3]), tensor([1, 2, 3, 0, 3, 0, 0, 1]))
```

注意：

在 DGL API 中，通常优选使用 Tensor 类型，因为它们在 C 中的有效内部存储以及显式数据类型和设备上下文信息。但是，大多数 DGL API 确实支持将 python 可迭代（例如 list）或 numpy.ndarray 作为快速原型的参数。

DGL 可以使用 32 位或 64 位整数存储节点和边 ID。节点和边 ID 的数据类型应该相同。通过使用 64 位，DGL 可以处理最多 $2^{63}-1$ 节点或边的图。 但是，如果一个图包含少于 $2^{31}-1$ 个节点或边，则应使用 32 位整数，因为它可以提高速度，并需要较少的内存。 DGL 提供了进行此类转换的方法。 请参阅下面的示例。

```
>>> edges = th.tensor([2, 5, 3]), th.tensor([3, 5, 0])  # edges 2->3, 5->5, 3->0
>>> g64 = dgl.graph(edges)  # DGL uses int64 by default
>>> print(g64.idtype)
torch.int64
>>> g32 = dgl.graph(edges, idtype=th.int32)  # create a int32 graph
>>> g32.idtype
torch.int32
>>> g64_2 = g32.long()  # convert to int64
>>> g64_2.idtype
torch.int64
>>> g32_2 = g64.int()  # convert to int32
>>> g32_2.idtype
torch.int32
```

1.3 节点和边特征

DGLGraph 的节点和边可以具有多个用户定义的命名特征，用于存储节点和边的特定属性。 可以通过 ndata 和 edata 接口访问这些功能。 例如，以下代码创建两个节点特征（在第 5 和 9 行中分别命名为“ x”和“ y”）和一个边特征（在第 6 行中命名为“ x”）。

```
01. >>> import dgl
02. >>> import torch as th
03. >>> g = dgl.graph(([0, 0, 1, 5], [1, 2, 2, 0])) # 6 nodes, 4 edges
04. >>> g
    Graph(num_nodes=6, num_edges=4,
          ndata_schemes={}
          edata_schemes={})
05. >>> g.ndata['x'] = th.ones(g.num_nodes(), 3)              # node feature of length 3
06. >>> g.edata['x'] = th.ones(g.num_edges(), dtype=th.int32)  # scalar integer feature
```

```
07. >>> g
    Graph(num_nodes=6, num_edges=4,
          ndata_schemes={'x' : Scheme(shape=(3,), dtype=torch.float32)}
          edata_schemes={'x' : Scheme(shape=(,), dtype=torch.int32)})
08. >>> # different names can have different shapes
09. >>> g.ndata['y'] = th.randn(g.num_nodes(), 5)
10. >>> g.ndata['x'][1]                # get node 1's feature
    tensor([1., 1., 1.])
11. >>> g.edata['x'][th.tensor([0, 3])]  # get features of edge 0 and 3
    tensor([1, 1], dtype=torch.int32)
```

有关 ndata / edata 接口的重要事实：
（1）仅允许使用数字类型的特征（例如 float，double 和 int）。它们可以是标量，向量或多维张量。
（2）每个节点特征都有一个唯一的名称，每个边特征都有一个唯一的名称。节点和边的特征可以具有相同的名称。（例如，上例中的“ x”）。
（3）通过张量分配创建特征，该张量将特征分配给图中的每个节点/边。该张量的 leading dimension 必须等于图中的节点/边的数量。您不能将特征分配给图中的节点/边的子集。
（4）具有相同名称的特征必须具有相同的维度和数据类型。
（5）特征张量以行为主的布局-每个行切片都存储一个节点或边的特征（例如，参见上面示例中的第 10-11 行）。
对于加权图，可以将权重存储为边特征，如下所示。

```
>>> # edges 0->1, 0->2, 0->3, 1->3
>>> edges = th.tensor([0, 0, 0, 1]), th.tensor([1, 2, 3, 3])
>>> weights = th.tensor([0.1, 0.6, 0.9, 0.7])  # weight of each edge
>>> g = dgl.graph(edges)
>>> g.edata['w'] = weights  # give it a name 'w'
>>> g
Graph(num_nodes=4, num_edges=4,
      ndata_schemes={}
      edata_schemes={'w' : Scheme(shape=(,), dtype=torch.float32)})
```

1.4 从外部来源创建图

从外部来源创建图包括：

- 从外部库创建图和系数矩阵（NetworkX 和 SciPy）
- 从磁盘加载

1.4.1 从外部库创建

以下代码段是根据 SciPy 稀疏矩阵和 NetworkX 图创建图的示例。

```
>>> import dgl
>>> import torch as th
>>> import scipy.sparse as sp
```

```
>>> spmat = sp.rand(100, 100, density=0.05) # 5% nonzero entries
>>> dgl.from_scipy(spmat)                  # from SciPy
Graph(num_nodes=100, num_edges=500,
      ndata_schemes={}
      edata_schemes={})

>>> import networkx as nx
>>> nx_g = nx.path_graph(5) # a chain 0-1-2-3-4
>>> dgl.from_networkx(nx_g) # from networkx
Graph(num_nodes=5, num_edges=8,
      ndata_schemes={}
      edata_schemes={})
```

请注意，从 nx.path_graph(5)构造时，生成的 DGLGraph 具有 8 个边而不是 4 个边。这是因为 nx.path_graph(5）构造了无向 NetworkX 图 networkx.Graph，而 DGLGraph 始终是有向的。 在将无向 NetworkX 图转换为 DGLGraph 时， DGL 在内部将无向边转换为两个有向边。 使用有向 NetworkX 图 networkx.DiGraph 可以避免这种行为。

```
>>> nxg = nx.DiGraph([(2, 1), (1, 2), (2, 3), (0, 0)])
>>> dgl.from_networkx(nxg)
Graph(num_nodes=4, num_edges=4,
      ndata_schemes={}
      edata_schemes={})
```

注意： DGL 在内部将 SciPy 矩阵和 NetworkX 图转换为张量以构造图。 因此， 这些构造方法并不适用于性能关键部件。

1.4.2 从磁盘创建

用于存储图的数据格式有很多， 无法列举所有选项。 因此， 本节仅提供一些通用的方式。

（1） CSV

参考地址：

https://github.com/dglai/WWW20-Hands-on-Tutorial/blob/master/basic_tasks/1_load_data.ipynb

（2） JSON/GML Format

尽管不是特别快， 但是 NetworkX 提供了许多实用程序来解析各种数据格式， 这间接允许 DGL 从这些源创建图。

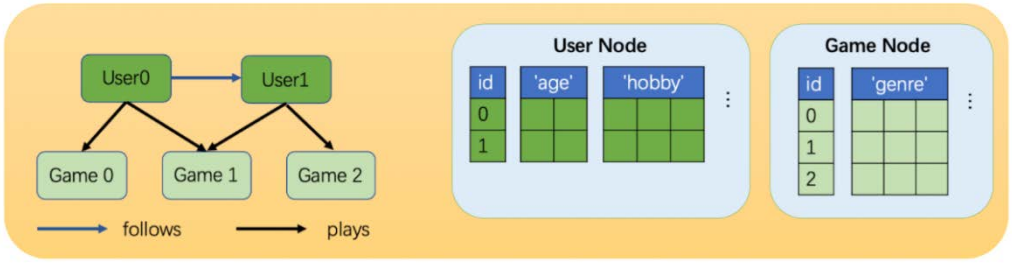
（3）DGL Binary Format

DGL 提供了 API，可以从以二进制格式存储的磁盘上保存和加载图。 除了图结构之外，API 还处理特征数据和图级别的标签数据。DGL 还支持直接指向 S3 或 HDFS 的检查点图。 参考手册提供了有关用法的更多详细信息。

See APIs: dgl.save_graphs(), dgl.load_graphs().

1.5 异构图

异构图可以具有不同类型的节点和边。 不同类型的节点/边具有独立的 ID 空间和功能存储。 例如，在下图中，用户和游戏节点 ID 均从零开始，并且它们具有不同的功能。



具有两种类型的节点（用户和游戏）和两种类型的边（跟随和游戏）的示例异构图。

1.5.1 创建一个异构图

在 DGL，一个异构图是指一系列以下的图：一种关系，每一个关系的类型是字符串三元组(source node type, edge type, destination node type)。由于关系消除了边类别的歧义， DGL 将其称为规范边类型。

```
{relation1 : node_tensor_tuple1,
 relation2 : node_tensor_tuple2,
 ...}
```

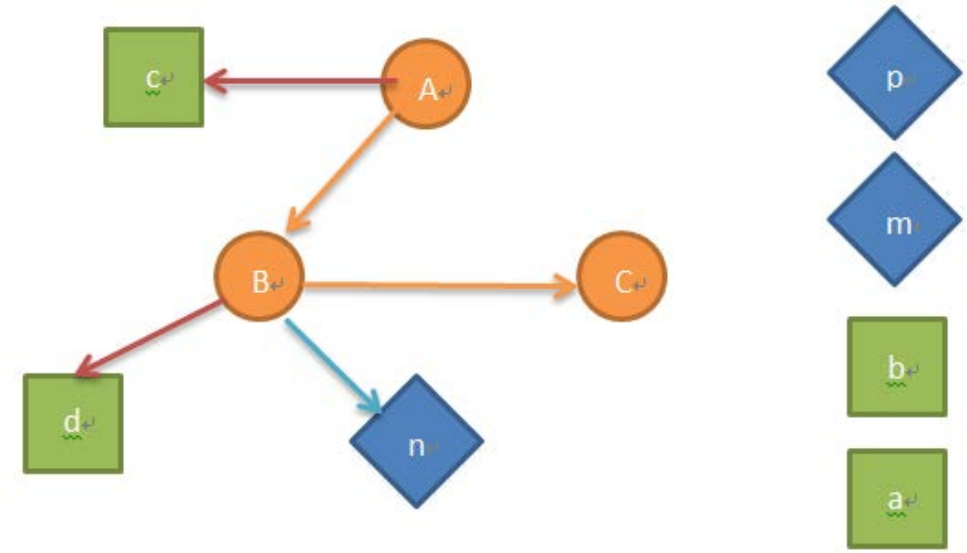
以下代码段是在 DGL 中创建异构图的示例。

```
>>> import dgl
>>> import torch as th

>>> # Create a heterograph with 3 node types and 3 edges types.
>>> graph_data = {
...    ('drug', 'interacts', 'drug'): (th.tensor([0, 1]), th.tensor([1, 2])),
...    ('drug', 'interacts', 'gene'): (th.tensor([0, 1]), th.tensor([2, 3])),
...    ('drug', 'treats', 'disease'): (th.tensor([1]), th.tensor([2]))
... }
>>> g = dgl.heterograph(graph_data)
>>> g.ntypes
['disease', 'drug', 'gene']
>>> g.etypes
['interacts', 'interacts', 'treats']
>>> g.canonical_etypes
[('drug', 'interacts', 'drug'),
 ('drug', 'interacts', 'gene'),
 ('drug', 'treats', 'disease')]
```

具体说明：

节点类型	A	B	C	
drug	0	1	2	
节点类型	a	b	c	d
gene	0	1	2	3
节点类型	p	m	n	
disease	0	1	2	

构造的异构图
含义就是(第一个元组是源节点，第二个数组是目标节点)，对应的位置就构成一条边，如下所示：



请注意，齐次图和二部图只是具有一种关系的特殊异构图。

```
>>> # A homogeneous graph
>>> dgl.heterograph({('node_type', 'edge_type', 'node_type'): (u, v)})
>>> # A bipartite graph
>>> dgl.heterograph({('source_type', 'edge_type', 'destination_type'): (u, v)})
```

与异构图相关联的元图是图的架构。 它在节点集和节点之间的边上指定类型约束。 元图中的节点 u 对应于关联的异构图中的节点类型。 元图中的边（u，v）表示关联的异构图中存在从类型 u 的节点到类型 v 的节点的边。

```
>>> g
Graph(num_nodes={'disease': 3, 'drug': 3, 'gene': 4},
      num_edges={('drug', 'interacts', 'drug'): 2,
                 ('drug', 'interacts', 'gene'): 2,
                 ('drug', 'treats', 'disease'): 1},
      metagraph=[('drug', 'drug', 'interacts'),
                 ('drug', 'gene', 'interacts'),
                 ('drug', 'disease', 'treats')])
>>> g.metagraph().edges()
OutMultiEdgeDataView([('drug', 'drug'), ('drug', 'gene'), ('drug', 'disease')])
```

See APIs: dgl.heterograph(), ntypes, etypes, canonical_etypes, metagraph.

1.5.2 处理多种类型

当引入多种节点/边类型时，用户在调用 DGLGraph API 以获取特定于类型的信息时需要指定特定的节点/边类型。 此外，不同类型的节点/边 具有单独的 IDs。

```
>>> # Get the number of all nodes in the graph
>>> g.num_nodes()
10
>>> # Get the number of drug nodes
>>> g.num_nodes('drug')
3
>>> # Nodes of different types have separate IDs,
>>> # hence not well-defined without a type specified
>>> g.nodes()
DGLError: Node type name must be specified if there are more than one node types.
>>> g.nodes('drug')
tensor([0, 1, 2])
```

为了设置/获取特定节点/边类型的功能，DGL 提供了两种新的语法类型 - g.nodes ['node_type'],data ['feat_name']和 g.edges ['edge_type'],data ['feat_name ']。

```
>>> # Set/get feature 'hv' for nodes of type 'drug'
>>> g.nodes['drug'].data['hv'] = th.ones(3, 1)
>>> g.nodes['drug'].data['hv']
tensor([[1.],
        [1.],
        [1.]])
>>> # Set/get feature 'he' for edge of type 'treats'
>>> g.edges['treats'].data['he'] = th.zeros(1, 1)
>>> g.edges['treats'].data['he']
tensor([[0.]])
```

如果图仅具有一种节点/边类型，则无需指定节点/边类型。

```
>>> g = dgl.heterograph({
...    ('drug', 'interacts', 'drug'): (th.tensor([0, 1]), th.tensor([1, 2])),
...    ('drug', 'is similar', 'drug'): (th.tensor([0, 1]), th.tensor([2, 3]))
... })
>>> g.nodes()
tensor([0, 1, 2, 3])
>>> # To set/get feature with a single type, no need to use the new syntax
>>> g.ndata['hv'] = th.ones(4, 1)
```

注意：当边类型唯一确定源节点和目标节点的类型时，可以仅使用一个字符串而不是字符串三元组来指定边类型。 例如，对于具有两个关系（“用户”，“玩”，“游戏”）和（“用户”，“喜欢”，“游戏”）的异构图，仅使用“游戏”或“喜欢”代指两个关系是安全的。

1.5.3 从磁盘建立异构图

（1）从 CSV 中建立

存储异异构图的一种常见方法是将不同类型的节点和边存储在不同的 CSV 文件中。 一个例子如下。

```
# data folder
data/
|-- drug.csv        # drug nodes
|-- gene.csv        # gene nodes
|-- disease.csv     # disease nodes
|-- drug-interact-drug.csv  # drug-drug interaction edges
|-- drug-interact-gene.csv  # drug-gene interaction edges
|-- drug-treat-disease.csv  # drug-treat-disease edges
```

与同构图类似，可以使用 Pandas 之类的程序包将 CSV 文件解析为 numpy 数组或框架张量，然后建立关系字典并从中构造出一个异构图。 该方法也适用于其他流行格式，例如 GML / JSON。

（2）从 DGL Binary Format 中家建立

DGL 分别提供了 dgl.save_graphs()和 dgl.load_graphs()，用于以二进制格式保存异构图并从二进制格式加载它们。

1.5.4 边类型子图

可以通过指定要保留的关系来创建异构图的子图（如果有的话会复制特征）。

```
>>> g = dgl.heterograph({
...   ('drug', 'interacts', 'drug'): (th.tensor([0, 1]), th.tensor([1, 2])),
...   ('drug', 'interacts', 'gene'): (th.tensor([0, 1]), th.tensor([2, 3])),
...   ('drug', 'treats', 'disease'): (th.tensor([1]), th.tensor([2]))
... })
>>> g.nodes['drug'].data['hv'] = th.ones(3, 1)

>>> # Retain relations ('drug', 'interacts', 'drug') and ('drug', 'treats',
'disease')
>>> # All nodes for 'drug' and 'disease' will be retained
>>> eg = dgl.edge_type_subgraph(g, [('drug', 'interacts', 'drug'),
...                                 ('drug', 'treats', 'disease')])
>>> eg
Graph(num_nodes={'disease': 3, 'drug': 3},
      num_edges={('drug', 'interacts', 'drug'): 2, ('drug', 'treats', 'disease'):
1},
      metagraph=[('drug', 'drug', 'interacts'), ('drug', 'disease', 'treats')])
>>> # The associated features will be copied as well
>>> eg.nodes['drug'].data['hv']
tensor([[1.],
        [1.],
        [1.]])
```

1.5.5 将异构图转换为同构图

异构图提供了一个简洁的界接口，用于管理不同类型的节点/边及其关联特征。在以下情况下特别有用：
（1）不同类型的节点/边的特征具有不同的数据类型或大小。
（2）我们想对不同类型的节点/边应用不同的操作。
如果上述条件不成立，并且不想在建模中区分节点/边类型，则 DGL 允许使用 dgl.DGLGraph.to_homogeneous()API 将异构图转换为同构图。 其过程如下：

```
>>> g = dgl.heterograph({
...    ('drug', 'interacts', 'drug'): (th.tensor([0, 1]), th.tensor([1, 2])),
...    ('drug', 'treats', 'disease'): (th.tensor([1]), th.tensor([2]))})
>>> g.nodes['drug'].data['hv'] = th.zeros(3, 1)
>>> g.nodes['disease'].data['hv'] = th.ones(3, 1)
>>> g.edges['interacts'].data['he'] = th.zeros(2, 1)
>>> g.edges['treats'].data['he'] = th.zeros(1, 2)

>>> # By default, it does not merge any features
>>> hg = dgl.to_homogeneous(g)
>>> 'hv' in hg.ndata
False

>>> # Copy edge features
>>> # For feature copy, it expects features to have
>>> # the same size and dtype across node/edge types
>>> hg = dgl.to_homogeneous(g, edata=['he'])
DGLError: Cannot concatenate column ‘he’ with shape Scheme(shape=(2,),
dtype=torch.float32) and shape Scheme(shape=(1,), dtype=torch.float32)

>>> # Copy node features
>>> hg = dgl.to_homogeneous(g, ndata=['hv'])
>>> hg.ndata['hv']
tensor([[1.],
        [1.],
        [1.],
        [0.],
        [0.],
        [0.]])

The original node/edge types and type-specific IDs are stored
in :py:attr:`~dgl.DGLGraph.ndata` and :py:attr:`~dgl.DGLGraph.edata`.
>>> # Order of node types in the heterograph
>>> g.ntypes
['disease', 'drug']
```

```
>>> # Original node types
>>> hg.ndata[dgl.NTYPE]
tensor([0, 0, 0, 1, 1, 1])
>>> # Original type-specific node IDs
>>> hg.ndata[dgl.NID]
>>> tensor([0, 1, 2, 0, 1, 2])

>>> # Order of edge types in the heterograph
>>> g.etypes
['interacts', 'treats']
>>> # Original edge types
>>> hg.edata[dgl.ETYPE]
tensor([0, 0, 1])
>>> # Original type-specific edge IDs
>>> hg.edata[dgl.EID]
tensor([0, 1, 0])
```

出于建模目的，可能需要将一些关系分组在一起，并对它们应用相同的操作。为了满足这一需求，可以先获取异构图的边型子图，然后将该子图转换为齐次图。

```
>>> g = dgl.heterograph({
...    ('drug', 'interacts', 'drug'): (th.tensor([0, 1]), th.tensor([1, 2])),
...    ('drug', 'interacts', 'gene'): (th.tensor([0, 1]), th.tensor([2, 3])),
...    ('drug', 'treats', 'disease'): (th.tensor([1]), th.tensor([2]))
... })
>>> sub_g = dgl.edge_type_subgraph(g, [('drug', 'interacts', 'drug'),
...                                    ('drug', 'interacts', 'gene')])
>>> h_sub_g = dgl.to_homogeneous(sub_g)
>>> h_sub_g
Graph(num_nodes=7, num_edges=4,
```

1.6 在 GPU 上使用 DGLGraph

通过在构建过程中传递两个 GPU 张量，可以在 GPU 上创建 DGLGraph。 另一种方法是使用 to()API 将 DGLGraph 复制到 GPU，GPU 将图结构以及特征数据复制到给定设备。

```
>>> import dgl
>>> import torch as th
>>> u, v = th.tensor([0, 1, 2]), th.tensor([2, 3, 4])
>>> g = dgl.graph((u, v))
>>> g.ndata['x'] = th.randn(5, 3)  # original feature is on CPU
>>> g.device
device(type='cpu')
>>> cuda_g = g.to('cuda:0')  # accepts any device objects from backend framework
>>> cuda_g.device
device(type='cuda', index=0)
```

```
>>> cuda_g.ndata['x'].device      # feature data is copied to GPU too
device(type='cuda', index=0)

>>> # A graph constructed from GPU tensors is also on GPU
>>> u, v = u.to('cuda:0'), v.to('cuda:0')
>>> g = dgl.graph((u, v))
>>> g.device
device(type='cuda', index=0)
```

涉及 GPU 图的任何操作都在 GPU 上执行。 因此，它们要求所有张量参数都已经放置在 GPU 上，并且结果（图或张量）也将放置在 GPU 上。 此外，GPU 图仅接受 GPU 上的特征数据。

```
>>> cuda_g.in_degrees()
tensor([0, 0, 1, 1, 1], device='cuda:0')
>>> cuda_g.in_edges([2, 3, 4])   # ok for non-tensor type arguments
(tensor([0, 1, 2], device='cuda:0'), tensor([2, 3, 4], device='cuda:0'))
>>> cuda_g.in_edges(th.tensor([2, 3, 4]).to('cuda:0'))  # tensor type must be on
GPU
(tensor([0, 1, 2], device='cuda:0'), tensor([2, 3, 4], device='cuda:0'))
>>> cuda_g.ndata['h'] = th.randn(5, 4)  # ERROR! feature must be on GPU too!
DGLError: Cannot assign node feature "h" on device cpu to a graph on device
cuda:0. Call DGLGraph.to() to copy the graph to the same device.
```

2. 消息传递

2.1 消息传递案例

设 $x_v \in R^{d1}$ 是节点 v 的特征，而 $w_e \in R^{d2}$ 是边(u, v)的特征。 消息传递范例在步骤 t + 1 定义了以下按节点和按边的计算：

$$\text{Edge-wise: } m_e^{(t+1)} = \phi\left(x_v^{(t)}, x_u^{(t)}, w_e^{(t)}\right), (u, v, e) \in \mathcal{E}.$$

$$\text{Node-wise: } x_v^{(t+1)} = \psi\left(x_v^{(t)}, \rho\left(\left\{m_e^{(t+1)} : (u, v, e) \in \mathcal{E}\right\}\right)\right).$$

在上述等式中，φ 是定义在每个边上的消息函数，通过将边特征与其附带的节点的特征相结合来生成消息；ψ 是在每个节点上定义的更新函数，通过使用 reduce 函数 ρ 聚合传入的消息来更新节点特征。

2.2 内置功函数和消息传递 APIs

在 DGL 中，

message 函数采用单个参数 edges（具有三个成员 src，dst 和 data）分别访问源节点，目标节点和边的特征。

reduce 函数采用单个参数节点。 节点可以访问其邮箱来收集其邻域通过边发送给它的消息。 一些最常见的归约运算包括 sum，max，min 等。

update 函数采用单个参数节点。 此函数操作通过 reduce 函数巨聚合后的结

果，典型是结合最后一个 step 节点的特征，并将保存为节点特征。

DGL 已实现了常用的作为 dgl.function 命名空间的 message 函数、reduce 函数。通常，我们建议尽可能使用内置函数，因为它们经过了充分优化并可以自动处理维度广播。

如果您的消息传递功能无法使用内置函数来实现，则可以实现用户定义的 mesage/reduce 函数（又名 UDF）。

内置 message 函数可以是一元或二进制。 我们目前支持一元副本。 对于二进制函数，我们现在支持 add，sub，mul，div，dot。 消息内置函数的命名约定是：u 表示 src 节点，v 表示 dst 节点，e 表示边。 这些函数的参数是字符串，指示相应节点和边的输入和输出字段名称。 这是受支持的内置函数的 dgl.function。 例如，要从 src 节点添加 hu 功特征，从 dst 节点添加 hv 特征，然后将结果保存在 he 字段的边中，我们可以使用内置函数 dgl.function.u_add_v('hu', 'hv'，'he')，这等效于消息 UDF：

```
def message_func(edges):
    return {'he': edges.src['hu'] + edges.dst['hv']}
```

内置的 reduce 函数支持 sum， max， min， prod 和 mean。 Reduce 函数通常具有两个参数，一个用于邮箱中的字段名称，一个用于目标中的字段名称，两者都是字符串。例如，dgl.function.sum('m','h')等同于对消息 m 求和的 Reduce UDF：

```
import torch
def reduce_func(nodes):
    return {'h': torch.sum(nodes.mailbox['m'], dim=1)}
```

在 DGL 中，调用 edge-wise 计算的接口是 apply_edges()。 apply_edges 的参数是消息函数和有效边类型，如 API Doc 中所述（默认情况下，所有边都会更新）。例如：

```
import dgl.function as fn
graph.apply_edges(fn.u_add_v('el', 'er', 'e'))
```

调用节点计算的接口是 update_all()。 update_all 的参数是消息函数，reduce 函数和更新函数。 通过将第三个参数保留为空，可以在 update_all 外部调用 update 函数。 建议这样做，因为通常可以将 update 函数编写为纯张量操作以使代码简洁。 例如：

```
def updata_all_example(graph):
    # store the result in graph.ndata['ft']
    graph.update_all(fn.u_mul_e('ft', 'a', 'm'),
                     fn.sum('m', 'ft'))
    # Call update function outside of update_all
    final_ft = graph.ndata['ft'] * 2
    return final_ft
```

此调用将通过将 src 节点特征 ft 和边特征 a 相乘来生成消息 m，将消息 m 求和以更新节点特征 ft，最后将 ft 乘以 2 得到结果 final_ft。 通话后，中间消息 m 将被清除。 上述函数的数学公式为：

$$final\_ft_i = 2 * \sum_{j \in \mathcal{N}(i)} (ft_j * a_{ij})$$

update_all 是一个高级 API，可在单个调用中 message generation, message

reduction and node update，这为优化留下了空间，如下所述。

2.3 编写有效的消息传递代码

DGL 优化了内存消耗和消息传递的计算速度。 优化包括：

- 将多个内核合并到一个内核中：通过使用 update_all 一次调用多个内置函数来实现。（速度优化）
- 节点和边上的并行性：DGL 将 edge-wise 式计算 apply_edges 抽象为广义采样的密集矩阵乘法（gSDDMM）操作，并跨边并行化计算。同样，DGL 将节点式计算 update_all 抽象为广义的稀疏-密度矩阵乘法（gSPMM）运算，并跨节点并行化计算。（速度优化）
- 避免将不必要的内存复制到边：要生成一条消息，要求源节点和目标节点具有该特征，一种选择是将源节点和目标节点特征复制到边。对于某些图，边的数量远大于节点的数量。这种复制可能很昂贵。 DGL 内置消息函数通过使用条目索引对节点特征进行采样来避免此内存复制。（内存和速度优化）
- 避免实例化边特征向量：完整的消息传递过程包括消息生成，消息缩减和节点更新。在 update_all 调用中，如果 message 函数和 reduce 函数是内置的，则它们会合并到一个内核中。边上没有消息实现。（内存优化）

根据上面所述，利用这些优化的一个常见做法是构造您自己的消息传递函数，将 update_all 调用与作为参数的内置函数结合起来。。

对于像 GATConv 这样需要在边保存消息的情况，我们需要使用内置函数调用 apply_edges。有时边上的消息可能是高维的，这会消耗内存。我们建议将 edata 维度保持在尽可能低的水平。

下面的示例说明了如何通过将边上的操作分割到节点来实现这一点。该选项做如下操作：拼接 src 特征和 dst 特征，然后应用一个线性层，即 W 乘以(u||v)。src 和 dst 特征维数高，线性层输出维数低。一个直接的实现是这样的:

```
linear = nn.Parameter(th.FloatTensor(size=(1, node_feat_dim*2)))
def concat_message_function(edges):
    {'cat_feat': torch.cat([edges.src.ndata['feat'],
edges.dst.ndata['feat']])}
g.apply_edges(concat_message_function)
g.edata['out'] = g.edata['cat_feat'] * linear
```

建议的实现将线性操作一分为二，一个应用于 src 特征，另一个应用于 dst 特征。在最终阶段，将线性操作的输出添加到边上，即 $W_l \times u + W_r \times v$，因为 $W \times (u||v) = W_l \times u + W_r \times v$，其中 W_l 和 W_r 分别为矩阵 W 的左右半部分:

```
linear_src = nn.Parameter(th.FloatTensor(size=(1, node_feat_dim)))
linear_dst = nn.Parameter(th.FloatTensor(size=(1, node_feat_dim)))
out_src = g.ndata['feat'] * linear_src
out_dst = g.ndata['feat'] * linear_dst
g.srcdata.update({'out_src': out_src})
g.dstdata.update({'out_dst': out_dst})
g.apply_edges(fn.u_add_v('out_src', 'out_dst', 'out'))
```

上述两种实现在数学上是等价的。后面的方法效率更高，因为我们不需要将 feat_src 和 feat_dst 保存在边上，保存在边上对内存效率不高。另外，可以使用

DGL的内置函数u_add_v对加法进行优化，进一步加快计算速度并节省内存占用。

2.4 在图的一部分上应用消息传递

如果我们只想更新图中的一部分节点，那么实践是通过为我们想要在更新中包含的节点提供 id 来创建子图，然后在子图上调用 update_all。例如:

```
nid = [0, 2, 3, 6, 7, 9]
sg = g.subgraph(nid)
sg.update_all(message_func, reduce_func, apply_node_func)
```

这是在 minibatch 训练中常见的用法。查看第 6 章:大型图的随机训练用户指南，了解更多详细的用法。

2.5 在消息传递中使用边权重

在 GNN 建模中，一个常见的做法是在消息聚合之前对消息应用边权重，例如在 GAT 和一些 GCN 变体中。在 DGL 中，处理这个的方法是:

- 将权重保存为边特征。
- 在消息函数中将边特征与 src 节点特征相乘。

例如：

```
graph.edata['a'] = affinity
graph.update_all(fn.u_mul_e('ft', 'a', 'm'),
                 fn.sum('m', 'ft'))
```

在上面，我们使用亲和度作为边权值。边的权值通常是一个标量。

2.6 在异构图中的消息传递

异构（1.5 异构图的用户指南）或简称为异形图是包含不同类型的节点和边的图。 不同类型的节点和边倾向于具有不同类型的属性，这些属性旨在捕获每个节点和边类型的特征。 在图神经网络的上下文中，根据其复杂性，可能需要使用具有不同维数的表示来对某些节点和边类型进行建模。

传递到异位图上的消息可以分为两部分：

- 每个关系 r 中的消息计算和聚合。
- 归约合并来自多个关系的相同节点类型上的结果。

DGL 调用在异形图上传递消息的接口是 multi_update_all()。 multi_update_all 接受一个字典，其中包含每个关系中 update_all 的参数，并使用关系作为键，一个字符串表示交叉类型 reducer。这个 reducer 可以是 sum, min, max, mean, stack 其中的一个，这是一个例子：

```
for c_etype in G.canonical_etypes:
    srctype, etype, dsttype = c_etype
    Wh = self.weight[etype](feat_dict[srctype])
    # Save it in graph for message passing
    G.nodes[srctype].data['Wh_%s' % etype] = Wh
    # Specify per-relation message passing functions: (message_func,
reduce_func).
    # Note that the results are saved to the same destination feature 'h', which
    # hints the type wise reducer for aggregation.
    funcs[etype] = (fn.copy_u('Wh_%s' % etype, 'm'), fn.mean('m', 'h'))
```

```
    # Trigger message passing of multiple types.
    G.multi_update_all(funcs, 'sum')
    # return the updated node feature dictionary
    return {ntype : G.nodes[ntype].data['h'] for ntype in G.ntypes}
```

3. 建立 GNN 模块

DGL NN 模块是 GNN 模型的构建块。它继承了 Pytorch 的神经网络模块，MXNet Gluon 的神经网络块和 TensorFlow 的 Keras 层，依赖于使用中的 DNN 框架后端。在 DGL NN 模块中，构造函数的参数配准和前向函数的张量运算与后端框架相同。通过这种方式，DGL 代码可以无缝地集成到后端框架代码中。主要区别在于消息传递操作在 DGL 中是唯一的。

DGL 集成了许多常用的 Conv Layers, Dense Conv Layers, Global Pooling Layers, and Utility Modules。我们欢迎您的贡献。

在本节中，我们将以带有 Pytorch 后端的 SAGEConv 为例，介绍如何构建自己的 DGL NN 模块。

3.1 DGL 中 NN Module 的构造函数

构造函数将完成以下工作：

- 设置选项；
- 注册可学习的参数或子模块；
- 重新设置参数；

```
import torch as th
from torch import nn
from torch.nn import init

from .... import function as fn
from ....base import DGLError
from ....utils import expand_as_pair, check_eq_shape

class SAGEConv(nn.Module):
    def __init__(self,
                 in_feats,
                 out_feats,
                 aggregator_type,
                 bias=True,
                 norm=None,
                 activation=None):
        super(SAGEConv, self).__init__()

        self._in_src_feats, self._in_dst_feats = expand_as_pair(in_feats)
        self._out_feats = out_feats
        self._aggre_type = aggregator_type
        self.norm = norm
        self.activation = activation
```

在构造函数中，我们首先需要设置数据维度。 对于一般的 Pytorch 模块，尺寸通常是输入尺寸，输出尺寸和隐藏尺寸。 对于图神经，输入维度可以分为源节点维和目标节点维度。

除了数据维度，图神经网络的典型选择是聚合类型（self._aggre_type）。 聚合类型决定了如何为特定目标节点聚合不同边上的消息。 常用的聚合类型包括 mean, sum, max, min。一些模块可能会应用更复杂的聚合，例如 lstm。

这里的 norm 是用于特征标准化的可调用函数。 在 SAGEConv 论文上，这样的归一化可以是 l2 范数：$h_v = h_v \ / \ ||h_v||_2$。

```
    # aggregator type: mean, max_pool, lstm, gcn
    if aggregator_type not in ['mean', 'max_pool', 'lstm', 'gcn']:
        raise KeyError('Aggregator type {} not supported.'.format(aggregator_type))
    if aggregator_type == 'max_pool':
        self.fc_pool = nn.Linear(self._in_src_feats, self._in_src_feats)
    if aggregator_type == 'lstm':
        self.lstm = nn.LSTM(self._in_src_feats, self._in_src_feats,
batch_first=True)
    if aggregator_type in ['mean', 'max_pool', 'lstm']:
        self.fc_self = nn.Linear(self._in_dst_feats, out_feats, bias=bias)
    self.fc_neigh = nn.Linear(self._in_src_feats, out_feats, bias=bias)
    self.reset_parameters()
```

注册参数和子模块。 在 SAGEConv 中，子模块根据聚合类型而有所不同。 这些模块是纯 Pytorch nn 模块，例如 nn.Linear， nn.LSTM 等。在构造函数的最后，通过调用 reset_parameters()来应用权重初始化。

```
    def reset_parameters(self):
        """Reinitialize learnable parameters."""
        gain = nn.init.calculate_gain('relu')
        if self._aggre_type == 'max_pool':
            nn.init.xavier_uniform_(self.fc_pool.weight, gain=gain)
        if self._aggre_type == 'lstm':
            self.lstm.reset_parameters()
        if self._aggre_type != 'gcn':
            nn.init.xavier_uniform_(self.fc_self.weight, gain=gain)
        nn.init.xavier_uniform_(self.fc_neigh.weight, gain=gain)
```

3.2 GDL 中 NN Module 前向函数

在 NN 模块中，forward()函数执行实际的消息传递和计算。 与通常以张量为参数的 Pytorch NN 模块相比，DGL NN 模块采用了附加参数 dgl.DGLGraph。forward()函数的工作量可以分为三部分：

- 图检查和图类型规范化；
- 消息传递和归约；
- 归约得到输出后更新特征；

让我们深入研究 SAGEConv 示例中的 forward()函数。

3.2.1 图检查和图类型规范化

```
def forward(self, graph, feat):
    with graph.local_scope():
        # Specify graph type then expand input feature according to graph type
        feat_src, feat_dst = expand_as_pair(feat, graph)
```

forward()需要处理输入中的许多极端情况，这些情况可能导致计算和消息传递中的值无效。 诸如 GraphConv 之类的 conv 模块的典型检查方法是验证输入图中是否有 0 入度节点。 当节点的入度为 0 时，邮箱将为空，而 reduce 函数将产生全零值。 这可能会导致模型性能 silent regression。 但是，在 SAGEConv 模块中，聚合的表示将与原始节点特征拼接在一起，forward()的输出将不会为全零。在这种情况下，无需进行此类检查。

DGL NN 模块应可在不同类型的图输入中重用，包括：齐次图，异构图（1.5 异构图），子图块（第 6 章：大图的随机训练）。

SAGEConv 的数学公式为:

$$h_{\mathcal{N}(dst)}^{(l+1)} = \text{aggregate}\left(\{h_{src}^{l}, \forall src \in \mathcal{N}(dst)\}\right)$$

$$h_{dst}^{(l+1)} = \sigma\left(W \cdot \text{concat}(h_{dst}^{l}, h_{\mathcal{N}(dst)}^{l+1} + b)\right)$$

$$h_{dst}^{(l+1)} = \text{norm}(h_{dst}^{l})$$

我们需要根据图类型指定源节点特征 feat_src 和目标节点特征 feat_dst。用于指定图类型并将 feat 扩展为 feat_src 和 feat_dst 的函数为 expand_as_pair()。 该功能的详细信息如下所示。

```
def expand_as_pair(input_, g=None):
    if isinstance(input_, tuple):
        # Bipartite graph case
        return input_
    elif g is not None and g.is_block:
        # Subgraph block case
        if isinstance(input_, Mapping):
            input_dst = {
                k: F.narrow_row(v, 0, g.number_of_dst_nodes(k))
                for k, v in input_.items()}
        else:
            input_dst = F.narrow_row(input_, 0, g.number_of_dst_nodes())
        return input_, input_dst
    else:
        # Homograph case
        return input_, input_
```

对于同构全图训练，源节点和目标节点是相同的。它们是图中的所有节点。

对于异质情况，图可以分解成几个二部图，每个二部图对应一个关系。关系表示为(src_type、edge_type、dst_dtype)。当我们确定输入特征是一个元组时，

我们将把图当作二部图。元组中的第一个元素将是源节点特征，第二个元素将是目标节点特征。

在 minibatch 训练中，将计算应用于给定的一堆目标节点所采样的子图。 子图在 DGL 中称为块。在消息传递之后，只有那些目标节点将被更新，因为它们与它们在原始完整图中拥有的邻域相同 在块创建阶段，dst 节点位于节点列表的最前面。 我们可以通过索引[0：g.number_of_dst_nodes()]找到 feat_dst。

在确定了 feat_src 和 feat_dst 之后，上述三种图类型的计算是相同的。

3.2.2 消息传递和归约

```
    if self._aggre_type == 'mean':
        graph.srcdata['h'] = feat_src
        graph.update_all(fn.copy_u('h', 'm'), fn.mean('m', 'neigh'))
        h_neigh = graph.dstdata['neigh']
    elif self._aggre_type == 'gcn':
        check_eq_shape(feat)
        graph.srcdata['h'] = feat_src
        graph.dstdata['h'] = feat_dst     # same as above if homogeneous
        graph.update_all(fn.copy_u('h', 'm'), fn.sum('m', 'neigh'))
        # divide in_degrees
        degs = graph.in_degrees().to(feat_dst)
        h_neigh = (graph.dstdata['neigh'] + graph.dstdata['h']) /
(degs.unsqueeze(-1) + 1)
    elif self._aggre_type == 'max_pool':
        graph.srcdata['h'] = F.relu(self.fc_pool(feat_src))
        graph.update_all(fn.copy_u('h', 'm'), fn.max('m', 'neigh'))
        h_neigh = graph.dstdata['neigh']
    else:
        raise KeyError('Aggregator type {} not
recognized.'.format(self._aggre_type))

    # GraphSAGE GCN does not require fc_self.
    if self._aggre_type == 'gcn':
        rst = self.fc_neigh(h_neigh)
    else:
        rst = self.fc_self(h_self) + self.fc_neigh(h_neigh)
```

该代码实际上执行消息传递并归约计算。 这部分代码因模块而异。 请注意，以上代码中的所有消息传递都是使用 update_all() API 和内置的消息/归约函数实现的，以充分利用 DGL 的性能优化，如第二章：消息传递中所述。

3.2.3 归约得到输出后更新特征

```
    # activation
    if self.activation is not None:
        rst = self.activation(rst)
    # normalization
```

```
if self.norm is not None:
    rst = self.norm(rst)
return rst
```

forward()函数的最后一部分是在 reduce 函数之后更新特征。 常见的更新操作是根据对象构造阶段中设置的选项应用激活功能和规范化。

3.3 异构 GraphConv 模块

dgl.nn.pytorch.HeteroGraphConv 是模块级封装，用于在异构图上运行 DGL NN 模块。 实现逻辑与消息传递级别 API multi_update_all()相同：

- 每个关系 r 中的 DGL NN 模块。
- 归约合并来自多个关系的相同节点类型上的结果。

这可以表述为：

$$h_{dst}^{(l+1)} = \underset{r \in \mathcal{R}, r_{dst}=dst}{AGG} \left(f_r(g_r, h_{r_{src}}^l, h_{r_{dst}}^l) \right)$$

其中 f_r 为各关系 r 的神经网络模块，AGG 为聚合函数。

3.3.1 HeteroGraphConv 实现逻

```
class HeteroGraphConv(nn.Module):
    def __init__(self, mods, aggregate='sum'):
        super(HeteroGraphConv, self).__init__()
        self.mods = nn.ModuleDict(mods)
        if isinstance(aggregate, str):
            self.agg_fn = get_aggregate_fn(aggregate)
        else:
            self.agg_fn = aggregate
```

异义图卷积采用字典 mods，该字典 mods 将每个关系映射到 nn 模块。 并设置将来自多个关系的结果聚合到相同节点类型上的功能。

```
def forward(self, g, inputs, mod_args=None, mod_kwargs=None):
    if mod_args is None:
        mod_args = {}
    if mod_kwargs is None:
        mod_kwargs = {}
    outputs = {nty : [] for nty in g.dsttypes}
```

除了输入图和输入张量之外，forwar()函数还使用两个附加的字典参数 mod_args 和 mod_kwargs。这两个词典与 self.mods 具有相同的键。当在 self.mods 中为不同类型的关系调用其对应的 NN 模块时，它们将用作自定义参数。

创建一个输出字典来保存每种目标类型的输出张量。 请注意，每个 nty 的值是一个列表，指示如果多个关系具有 nty 作为目标类型，则单个节点类型可能会获得多个输出。 我们会将它们保留在列表中以进行进一步汇总。

```
if g.is_block:
    src_inputs = inputs
    dst_inputs = {k: v[:g.number_of_dst_nodes(k)] for k, v in inputs.items()}
```

```
    else:
        src_inputs = dst_inputs = inputs

    for stype, etype, dtype in g.canonical_etypes:
        rel_graph = g[stype, etype, dtype]
        if rel_graph.number_of_edges() == 0:
            continue
        if stype not in src_inputs or dtype not in dst_inputs:
            continue
        dstdata = self.mods[etype](
            rel_graph,
            (src_inputs[stype], dst_inputs[dtype]),
            *mod_args.get(etype, ()),
            **mod_kwargs.get(etype, {}))
        outputs[dtype].append(dstdata)
```

输入 g 可以是异构图，也可以是异构图的子图块。与普通的 NN 模块一样，forward()函数需要分别处理不同的输入图类型。

每个关系都表示为 canonical_etype，即（stype，etype，dtype）。 使用 canonical_etype 作为键，我们可以提取二部图 rel_graph。 对于二部图，输入特征将被组织为元组（src_inputs [stype]，dst_inputs [dtype]）。 调用每个关系的 NN 模块，并保存输出。 为避免不必要的调用，将跳过没有边或没有其 src 类型的节点的关系。

```
    rsts = {}
    for nty, alist in outputs.items():
        if len(alist) != 0:
            rsts[nty] = self.agg_fn(alist, nty)
```

最后，使用 self.agg_fn 函数汇总来自多个关系的相同目标节点类型上的结果。可以在 dgl.nn.pytorch.HeteroGraphConv 的 API 文档中找到示例。

4 图数据管道

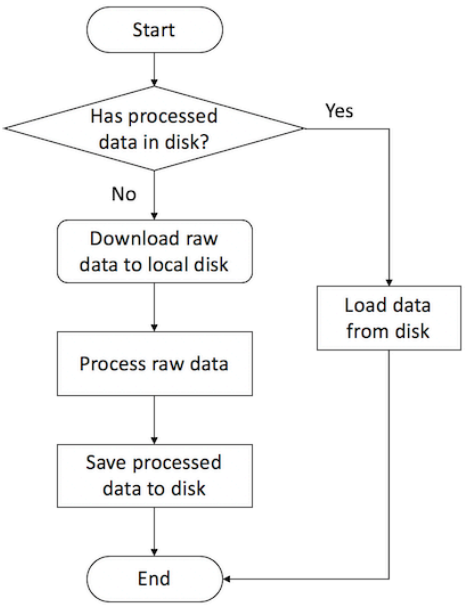
DGL 在 dgl.data 中实现了许多常用的图数据集。

它们遵循在 dgl.data.DGLDataset 类中定义的标准管道。 我们强烈建议将图数据处理到 dgl.data.DGLDataset 子类中，因为管道为加载，处理和保存图数据提供了简单而干净的解决方案。

本章介绍如何为我们自己的图数据创建 DGL 数据集。 以下内容说明了管道的工作方式，并展示了如何实现管道的每个组件。

4.1 DGLDataset 类

dgl.data.DGLDataset 是处理，加载和保存 dgl.data 中定义的图数据集的基类。它实现了用于处理图数据的基本管道。 以下流程图显示了管道的工作方式。




```
from dgl.data import DGLDataset

class MyDataset(DGLDataset):
    """ Template for customizing graph datasets in DGL.

    Parameters
    ----------
    url : str
        URL to download the raw dataset
    raw_dir : str
        Specifying the directory that will store the
        downloaded data or the directory that
        already stores the input data.
        Default: ~/.dgl/
    save_dir : str
        Directory to save the processed dataset.
        Default: the value of `raw_dir`
    force_reload : bool
        Whether to reload the dataset. Default: False
    verbose : bool
        Whether to print out progress information
    """
    def __init__(self,
                 url=None,
                 raw_dir=None,
                 save_dir=None,
                 force_reload=False,
                 verbose=False):
        super(MyDataset, self).__init__(name='dataset_name',
```

```
                                    url=url,
                                    raw_dir=raw_dir,
                                    save_dir=save_dir,
                                    force_reload=force_reload,
                                    verbose=verbose)

    def download(self):
        # download raw data to local disk
        pass

    def process(self):
        # process raw data to graphs, labels, splitting masks
        pass

    def __getitem__(self, idx):
        # get one example by index
        pass

    def __len__(self):
        # number of data examples
        pass

    def save(self):
        # save processed data to directory `self.save_path`
        pass

    def load(self):
        # load processed data from directory `self.save_path`
        pass

    def has_cache(self):
        # check whether there are processed data in `self.save_path`
        pass
```

dgl.data.DGLDataset 类具有抽象函数 process()，__ getitem __(idx)和__len __()，必须在子类中实现。 但是我们建议也实现保存和加载，因为它们可以节省大量时间来处理大型数据集，并且有多个 API 可以简化此操作（请参阅保存和加载数据）。

请注意，dgl.data.DGLDataset 的目的是提供一种标准且方便的方式来加载图数据。 我们可以存储有关数据集的图，特征，标签，蒙版和基本信息，例如类别数，标签数等。诸如 dgl.data.DGLDataset 子类之外的操作如采样，分区或特征归一化。

本章的其余部分显示了在管道中实现功能的最佳实践。

4.2 下载原始数据（可选）

如果我们的数据集已经在本地磁盘中，请确保它在目录 raw_dir 中。 如果我们想在任何地方运行我们的代码而又不想麻烦下载数据并将其移动到正确的目录，则可以通过实现函数 download()来自动进行操作。

如果数据集是一个 zip 文件，则使 MyDataset 继承自 dgl.data.DGLBuiltinDataset 类，该类将为我们处理 zip 文件的提取。 否则，请像 dgl.data.QM7bDataset 中那样实现 download()：

```
import os
from dgl.data.utils import download


def download(self):
    # path to store the file
    file_path = os.path.join(self.raw_dir, self.name + '.mat')
    # download file
    download(self.url, path=file_path)
```

上面的代码将一个.mat 文件下载到目录 self.raw_dir 中。 如果文件是.gz，.tar，.tar.gz 或.tgz 文件，请使用 dgl.data.utils.extract_archive()函数进行提取。 以下代码显示了如何在 dgl.data.BitcoinOTCDataset 中下载.gz 文件：

```
from dgl.data.utils import download, extract_archive


def download(self):
    # path to store the file
    # make sure to use the same suffix as the original file name's
    gz_file_path = os.path.join(self.raw_dir, self.name + '.csv.gz')
    # download file
    download(self.url, path=gz_file_path)
    # check SHA-1
    if not check_sha1(gz_file_path, self._sha1_str):
        raise UserWarning('File {} is downloaded but the content hash does not
match.'
                          'The repo may be outdated or download may be incomplete.
'
                          'Otherwise you can create an issue for
it.'.format(self.name + '.csv.gz'))
    # extract file to directory `self.name` under `self.raw_dir`
    self._extract_gz(gz_file_path, self.raw_path)
```

上面的代码会将文件提取到 self.raw_dir 下的目录 self.name 中。 如果该类继承自 dgl.data.DGLBuiltinDataset 来处理 zip 文件，则它将也将文件提取到目录 self.name 中。

（可选）我们可以按照上面的示例检查下载文件的 SHA-1 字符串，以防作者有一天在远程服务器上更改了文件。

4.3 处理数据

我们在 process()函数中实现数据处理代码，并假定原始数据已经位于 self.raw_dir 中。 在图上进行机器学习时，通常有三种类型的任务：图分类，节点分类和链接预测。 我们将展示如何处理与这些任务相关的数据集。

在这里，我们重点介绍处理图，特征和蒙版的标准方法。 我们将以内置数据集为例，并跳过从文件构建图的实现，但添加了指向详细实现的链接。 请参考 1.4 从外部源创建图以查看有关如何从外部源构建图的完整指南。

4.3.1 处理图分类数据集

图分类数据集与使用微型批处理训练的典型机器学习任务中的大多数数据集几乎相同。 因此，我们将原始数据处理为 dgl.DGLGraph 对象的列表和标签张量的列表。 此外，如果原始数据已拆分为多个文件，则可以添加参数拆分以加载数据的特定部分。

以 dgl.data.QM7bDataset 为例：

```
class QM7bDataset(DGLDataset):
    _url = 'http://deepchem.io.s3-website-us-west-1.amazonaws.com/' \
           'datasets/qm7b.mat'
    _sha1_str = '4102c744bb9d6fd7b40ac67a300e49cd87e28392'
    def __init__(self, raw_dir=None, force_reload=False, verbose=False):
        super(QM7bDataset, self).__init__(name='qm7b',
                                          url=self._url,
                                          raw_dir=raw_dir,
                                          force_reload=force_reload,
                                          verbose=verbose)

    def process(self):
        mat_path = self.raw_path + '.mat'
        # process data to a list of graphs and a list of labels
        self.graphs, self.label = self._load_graph(mat_path)

    def __getitem__(self, idx):
        """ Get graph and label by index

        Parameters
        ----------
        idx : int
            Item index

        Returns
        -------
        (dgl.DGLGraph, Tensor)
        """
        return self.graphs[idx], self.label[idx]
```

```
    def __len__(self):
        """Number of graphs in the dataset"""
        return len(self.graphs)
```

在 process()中，原始数据被处理为图列表和标签列表。 我们必须实现__getitem __(idx)和__len __()进行迭代。 我们建议使用__getitem __(idx)返回上面的元组（图，标签）。 请检查 QM7bDataset 源代码以获取 self._load_graph()和__getitem__的详细信息。

我们还可以向类添加属性以指示数据集的一些有用信息。

在 dgl.data.QM7bDataset 中，我们可以添加属性 num_labels 来指示此多任务数据集中的预测任务总数：

```
@property
def num_labels(self):
    """Number of labels for each graph, i.e. number of prediction tasks."""
    return 14
```

完成所有这些编码之后，我们最终可以如下使用 dgl.data.QM7bDataset：

```
from torch.utils.data import DataLoader

# load data
dataset = QM7bDataset()
num_labels = dataset.num_labels

# create collate_fn
def _collate_fn(batch):
    graphs, labels = batch
    g = dgl.batch(graphs)
    labels = torch.tensor(labels, dtype=torch.long)
    return g, labels

# create dataloaders
dataloader = DataLoader(dataset, batch_size=1, shuffle=True,
collate_fn=_collate_fn)

# training
for epoch in range(100):
    for g, labels in dataloader:
        # your training code here
        pass
```

训练图分类模型的完整指南可以在 5.4 图分类中找到。

有关图分类数据集的更多示例，请参考我们内置的图分类数据集：

Graph isomorphism network dataset

Mini graph classification dataset

QM7b dataset

TU dataset

4.3.2 处理节点分类数据集

与图分类不同，节点分类通常在单个图上。 这样，数据集的拆分位于图的节点上。 我们建议使用节点掩码来指定拆分。 我们以内置数据集 CitationGraphDataset 为例：

```
import dgl
from dgl.data import DGLBuiltinDataset

class CitationGraphDataset(DGLBuiltinDataset):
    _urls = {
        'cora_v2' : 'dataset/cora_v2.zip',
        'citeseer' : 'dataset/citeseer.zip',
        'pubmed' : 'dataset/pubmed.zip',
    }

    def __init__(self, name, raw_dir=None, force_reload=False, verbose=True):
        assert name.lower() in ['cora', 'citeseer', 'pubmed']
        if name.lower() == 'cora':
            name = 'cora_v2'
        url = _get_dgl_url(self._urls[name])
        super(CitationGraphDataset, self).__init__(name,
                                            url=url,
                                            raw_dir=raw_dir,
                                            force_reload=force_reload,
                                            verbose=verbose)

    def process(self):
        # Skip some processing code
        # === data processing skipped ===

        # build graph
        g = dgl.graph(graph)
        # splitting masks
        g.ndata['train_mask'] = generate_mask_tensor(train_mask)
        g.ndata['val_mask'] = generate_mask_tensor(val_mask)
        g.ndata['test_mask'] = generate_mask_tensor(test_mask)
        # node labels
        g.ndata['label'] = F.tensor(labels)
        # node features
        g.ndata['feat'] = F.tensor(_preprocess_features(features),
                                dtype=F.data_type_dict['float32'])
        self._num_labels = onehot_labels.shape[1]
        self._labels = labels
        self._g = g
```

```
    def __getitem__(self, idx):
        assert idx == 0, "This dataset has only one graph"
        return self._g

    def __len__(self):
        return 1
```

为简便起见，我们跳过了 process()中的一些代码，以突出显示用于处理节点分类数据集的关键部分：存储在 g.ndata 中的分割掩码，节点特征和节点标签。有关详细的实现，请参考 CitationGraphDataset 源代码。

注意，__ getitem __(idx)和__len __()的实现也被更改，因为通常只有一个图用于节点分类任务。 蒙版是 PyTorch 和 TensorFlow 中的布尔张量，而 MXNet 是 float 张量。

我们使用 CitationGraphDataset 的子类 dgl.data.CiteseerGraphDataset 来显示其用法：

```
# load data
dataset = CiteseerGraphDataset(raw_dir='')
graph = dataset[0]

# get split masks
train_mask = graph.ndata['train_mask']
val_mask = graph.ndata['val_mask']
test_mask = graph.ndata['test_mask']

# get node features
feats = graph.ndata['feat']

# get labels
labels = graph.ndata['label']
```

有关训练节点分类模型的完整指南，请参见 5.1 节点分类/回归。

有关节点分类数据集的更多示例，请参阅我们的内置数据集：

Citation network dataset

CoraFull dataset

Amazon Co-Purchase dataset

Coauthor dataset

Karate club dataset

Protein-Protein Interaction dataset

Reddit dataset

Symmetric Stochastic Block Model Mixture dataset

Stanford sentiment treebank dataset

RDF datasets

4.3.3 处理链接预测数据集

链接预测数据集的处理与节点分类的处理相似，数据集中通常只有一个图。

我们以内置数据集 KnowledgeGraphDataset 为例，仍然跳过详细的数据处理代码，以突出显示用于处理链接预测数据集的关键部分：

```
# Example for creating Link Prediction datasets
class KnowledgeGraphDataset(DGLBuiltinDataset):
    def __init__(self, name, reverse=True, raw_dir=None, force_reload=False,
verbose=True):
        self._name = name
        self.reverse = reverse
        url = _get_dgl_url('dataset/') + '{}.tgz'.format(name)
        super(KnowledgeGraphDataset, self).__init__(name,
                                                    url=url,
                                                    raw_dir=raw_dir,
                                                    force_reload=force_reload,
                                                    verbose=verbose)

    def process(self):
        # Skip some processing code
        # === data processing skipped ===

        # splitting mask
        g.edata['train_mask'] = train_mask
        g.edata['val_mask'] = val_mask
        g.edata['test_mask'] = test_mask
        # edge type
        g.edata['etype'] = etype
        # node type
        g.ndata['ntype'] = ntype
        self._g = g

    def __getitem__(self, idx):
        assert idx == 0, "This dataset has only one graph"
        return self._g

    def __len__(self):
        return 1
```

如代码所示，我们将分割蒙版添加到图的 edata 字段中。

检查 KnowledgeGraphDataset 源代码以查看完整的代码。 我们使用 KnowledgeGraphDataset 的子类 dgl.data.FB15k237Dataset 来显示其用法：

```
import torch

# load data
dataset = FB15k237Dataset()
graph = dataset[0]
```

```
# get training mask
train_mask = graph.edata['train_mask']
train_idx = torch.nonzero(train_mask).squeeze()
src, dst = graph.edges(train_idx)
# get edge types in training set
rel = graph.edata['etype'][train_idx]
```

有关训练链接预测模型的完整指南，请参见 5.3 链接预测。

有关链接预测数据集的更多示例，请参考我们的内置数据集：

Knowlege graph dataset

BitcoinOTC dataset

4.4 保存和加载数据

我们建议实现保存和加载功能，以将处理后的数据缓存在本地磁盘中。 在大多数情况下，这样可以节省大量数据处理时间。 我们提供了四个使事情变得简单的功能：

- dgl.save_graphs()和 dgl.load_graphs()：将 DGLGraph 对象和标签保存到本地磁盘/从本地磁盘加载。
- dgl.data.utils.save_info()和 dgl.data.utils.load_info()：将数据集（python dict 对象）的有用信息保存到本地磁盘/从本地磁盘加载。

以下示例显示了如何保存和加载图和数据集信息列表。

```
import os
from dgl import save_graphs, load_graphs
from dgl.data.utils import makedirs, save_info, load_info

def save(self):
    # save graphs and labels
    graph_path = os.path.join(self.save_path, self.mode + '_dgl_graph.bin')
    save_graphs(graph_path, self.graphs, {'labels': self.labels})
    # save other information in python dict
    info_path = os.path.join(self.save_path, self.mode + '_info.pkl')
    save_info(info_path, {'num_classes': self.num_classes})

def load(self):
    # load processed data from directory `self.save_path`
    graph_path = os.path.join(self.save_path, self.mode + '_dgl_graph.bin')
    self.graphs, label_dict = load_graphs(graph_path)
    self.labels = label_dict['labels']
    info_path = os.path.join(self.save_path, self.mode + '_info.pkl')
    self.num_classes = load_info(info_path)['num_classes']

def has_cache(self):
    # check whether there are processed data in `self.save_path`
    graph_path = os.path.join(self.save_path, self.mode + '_dgl_graph.bin')
    info_path = os.path.join(self.save_path, self.mode + '_info.pkl')
```

```
        return os.path.exists(graph_path) and os.path.exists(info_path)
```

请注意，在某些情况下不适合保存已处理的数据。 例如，在内置数据集dgl.data.GDELTDataset 中，处理后的数据非常大，因此在__getitem __(idx)中处理每个数据样本更为有效。

4.5 使用 ogb 包加载 OGB 数据集

开放图基准（OGB）是基准数据集的集合。OGB 官方软件包 ogb 提供了 API，用于将 OGB 数据集下载和处理到 dgl.data.DGLGraph 对象中。我们在这里介绍它们的基本用法。

首先使用 pip 安装 ogb 软件包：

```
pip install ogb
```

以下代码显示了如何为“图属性预测”任务加载数据集。

```
# Load Graph Property Prediction datasets in OGB
import dgl
import torch
from ogb.graphproppred import DglGraphPropPredDataset
from torch.utils.data import DataLoader


def _collate_fn(batch):
    # batch is a list of tuple (graph, label)
    graphs = [e[0] for e in batch]
    g = dgl.batch(graphs)
    labels = [e[1] for e in batch]
    labels = torch.stack(labels, 0)
    return g, labels

# load dataset
dataset = DglGraphPropPredDataset(name='ogbg-molhiv')
split_idx = dataset.get_idx_split()
# dataLoader
train_loader = DataLoader(dataset[split_idx["train"]], batch_size=32,
shuffle=True, collate_fn=_collate_fn)
valid_loader = DataLoader(dataset[split_idx["valid"]], batch_size=32,
shuffle=False, collate_fn=_collate_fn)
test_loader = DataLoader(dataset[split_idx["test"]], batch_size=32,
shuffle=False, collate_fn=_collate_fn)
```

加载节点属性预测数据集是相似的，但是请注意，这种数据集中只有一个图对象。

```
# Load Node Property Prediction datasets in OGB
from ogb.nodeproppred import DglNodePropPredDataset

dataset = DglNodePropPredDataset(name='ogbn-proteins')
split_idx = dataset.get_idx_split()
```

```
# there is only one graph in Node Property Prediction datasets
g, labels = dataset[0]
# get split labels
train_label = dataset.labels[split_idx['train']]
valid_label = dataset.labels[split_idx['valid']]
test_label = dataset.labels[split_idx['test']]
```

链接属性预测数据集中的每个数据集还包含一个图：

```
# Load Link Property Prediction datasets in OGB
from ogb.linkproppred import DglLinkPropPredDataset

dataset = DglLinkPropPredDataset(name='ogbl-ppa')
split_edge = dataset.get_edge_split()

graph = dataset[0]
print(split_edge['train'].keys())
print(split_edge['valid'].keys())
print(split_edge['test'].keys())
```

5 训练图神经网络

总览

本章通过第 2 章介绍的消息传递方法和第 3 章介绍的神经网络模块构建 GNN 模块以进行节点分类，边分类，链接预测和小图的图分类。

本章假设您的图及其所有节点和边特征都可以放入 GPU。 如果不能，请参见第 6 章：大图上的随机训练。

以下文本假定已经准备好图和节点/边特征。 如果您打算使用 DGL 提供的数据集或其他兼容的 DGLDataset（如第 4 章：图数据管道中所述），则可以使用以下内容获取单图数据集的图：

```
import dgl

dataset = dgl.data.CiteseerGraphDataset()
graph = dataset[0]
```

注意：在本章中，我们将使用 PyTorch 作为后端

异构图

有时您想处理异构图。 在这里，我们以合成异构图为例，演示节点分类，边分类和链接预测任务。

合成异构图 hetero_graph 具有以下边类型：

- ('user', 'follow', 'user')
- ('user', 'followed-by', 'user')
- ('user', 'click', 'item')
- ('item', 'clicked-by', 'user')
- ('user', 'dislike', 'item')

- ('item', 'disliked-by', 'user')

```
import numpy as np
import torch

n_users = 1000
n_items = 500
n_follows = 3000
n_clicks = 5000
n_dislikes = 500
n_hetero_features = 10
n_user_classes = 5
n_max_clicks = 10

follow_src = np.random.randint(0, n_users, n_follows)
follow_dst = np.random.randint(0, n_users, n_follows)
click_src = np.random.randint(0, n_users, n_clicks)
click_dst = np.random.randint(0, n_items, n_clicks)
dislike_src = np.random.randint(0, n_users, n_dislikes)
dislike_dst = np.random.randint(0, n_items, n_dislikes)

hetero_graph = dgl.heterograph({
    ('user', 'follow', 'user'): (follow_src, follow_dst),
    ('user', 'followed-by', 'user'): (follow_dst, follow_src),
    ('user', 'click', 'item'): (click_src, click_dst),
    ('item', 'clicked-by', 'user'): (click_dst, click_src),
    ('user', 'dislike', 'item'): (dislike_src, dislike_dst),
    ('item', 'disliked-by', 'user'): (dislike_dst, dislike_src)})

hetero_graph.nodes['user'].data['feature'] = torch.randn(n_users,
n_hetero_features)
hetero_graph.nodes['item'].data['feature'] = torch.randn(n_items,
n_hetero_features)
hetero_graph.nodes['user'].data['label'] = torch.randint(0, n_user_classes,
(n_users,))
hetero_graph.edges['click'].data['label'] = torch.randint(1, n_max_clicks,
(n_clicks,)).float()
# randomly generate training masks on user nodes and click edges
hetero_graph.nodes['user'].data['train_mask'] = torch.zeros(n_users,
dtype=torch.bool).bernoulli(0.6)
hetero_graph.edges['click'].data['train_mask'] = torch.zeros(n_clicks,
dtype=torch.bool).bernoulli(0.6)
```

5.3 节点分类/回归

图神经网络最流行和广泛采用的任务之一是节点分类，其中训练/验证/测试

集中的每个节点都从一组预定义的类别中分配了一个 ground-truth 类别。 节点回归是相似的，其中训练/验证/测试集中的每个节点都分配有 ground-truth 数字。

5.1.1 总览

为了对节点进行分类，图神经网络执行了第 2 章：消息传递中讨论的消息传递，以利用节点自身的特征以及相邻节点和边特征。 消息传递可以重复多次，以合并来自更大范围邻域的信息。

5.1.2 编写神经网络模型

DGL 提供了一些内置的图卷积模块，可以执行一轮消息传递。 在本指南中，我们选择 dgl.nn.pytorch.SAGEConv（在 MXNet 和 Tensorflow 中也提供），这是 GraphSAGE 的图卷积模块。

通常对于图上的深度学习模型，我们需要一个多层图神经网络，在其中进行多轮消息传递。 这可以通过如下堆叠图卷积模块来实现。

```
# Contruct a two-layer GNN model
import dgl.nn as dglnn
import torch.nn as nn
import torch.nn.functional as F
class SAGE(nn.Module):
    def __init__(self, in_feats, hid_feats, out_feats):
        super().__init__()
        self.conv1 = dglnn.SAGEConv(
            in_feats=in_feats, out_feats=hid_feats, aggregator_type='mean')
        self.conv2 = dglnn.SAGEConv(
            in_feats=hid_feats, out_feats=out_feats, aggregator_type='mean')

    def forward(self, graph, inputs):
        # inputs are features of nodes
        h = self.conv1(graph, inputs)
        h = F.relu(h)
        h = self.conv2(graph, h)
        return h
```

请注意，您不仅可以将上述模型用于节点分类，还可以为其他下游任务（例如 5.2 边分类/回归，5.3 链接预测或 5.4 图分类）获得隐藏的节点表示。

有关内置图卷积模块的完整列表，请参考 dgl.nn。

有关 DGL 神经网络模块如何工作以及如何通过消息传递编写自定义神经网络模块的更多详细信息，请参阅第 3 章：构建 GNN 模块中的示例。

5.1.3 训练循环

利用整张图进行训练只涉及上文所定义的模型的前向传播，并且在被训练的节点上比较预测值和 ground-truth 来计算损失。

本节使用一个 DGL 内置的数据集 dgl.data.CiteseerGraphDataset 展现训练循环。节点特征和标签被存储在其图实例中，训练/验证/测试切分也通过布尔掩码的形式存储在图上。 这类似于您在第 4 章：图数据管道中看到的内容。

```
node_features = graph.ndata['feat']
node_labels = graph.ndata['label']
train_mask = graph.ndata['train_mask']
valid_mask = graph.ndata['val_mask']
test_mask = graph.ndata['test_mask']
n_features = node_features.shape[1]
n_labels = int(node_labels.max().item() + 1)
```

以下是通过准确性评估模型的示例。

```
def evaluate(model, graph, features, labels, mask):
    model.eval()
    with torch.no_grad():
        logits = model(graph, features)
        logits = logits[mask]
        labels = labels[mask]
        _, indices = torch.max(logits, dim=1)
        correct = torch.sum(indices == labels)
        return correct.item() * 1.0 / len(labels)
```

然后，您可以编写我们的训练循环，如下所示。

```
model = SAGE(in_feats=n_features, hid_feats=100, out_feats=n_labels)
opt = torch.optim.Adam(model.parameters())

for epoch in range(10):
    model.train()
    # forward propagation by using all nodes
    logits = model(graph, node_features)
    # compute loss
    loss = F.cross_entropy(logits[train_mask], node_labels[train_mask])
    # compute validation accuracy
    acc = evaluate(model, graph, node_features, node_labels, valid_mask)
    # backward propagation
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())

    # Save model if necessary.  Omitted in this example.
```

GraphSAGE 提供了一个端到端的同构图节点分类示例。 您可以在示例中的 GraphSAGE 类中看到相应的模型实现，其中具有可调整的层数，丢失概率以及可自定义的聚合函数和非线性。

5.1.4 异构图

如果您的图是异构的，则您可能希望从所有边类型的邻域那里收集消息。 您可以使用模块 dgl.nn.pytorch.HeteroGraphConv（在 MXNet 和 Tensorflow 中也可用）在所有边类型上执行消息传递，然后为每种边类型组合不同的图卷积模块。

以下代码将定义一个异构图卷积模块，该模块首先对每种边类型执行单独的图卷积，然后将每种边类型上的消息聚合求和，作为所有节点类型的最终结果。

```
# Define a Heterograph Conv model
import dgl.nn as dglnn

class RGCN(nn.Module):
    def __init__(self, in_feats, hid_feats, out_feats, rel_names):
        super().__init__()

        self.conv1 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(in_feats, hid_feats)
            for rel in rel_names}, aggregate='sum')
        self.conv2 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(hid_feats, out_feats)
            for rel in rel_names}, aggregate='sum')

    def forward(self, graph, inputs):
        # inputs are features of nodes
        h = self.conv1(graph, inputs)
        h = {k: F.relu(v) for k, v in h.items()}
        h = self.conv2(graph, h)
        return h
```

dgl.nn.HeteroGraphConv 接收节点类型和节点特征张量的字典作为输入，并返回另一个由节点类型和节点特征的组成的字典。

因此，假设我们在上面的示例中具有 user 和 item 特征。

```
model = RGCN(n_hetero_features, 20, n_user_classes, hetero_graph.etypes)
user_feats = hetero_graph.nodes['user'].data['feature']
item_feats = hetero_graph.nodes['item'].data['feature']
labels = hetero_graph.nodes['user'].data['label']
train_mask = hetero_graph.nodes['user'].data['train_mask']
```

只需执行以下正向传播即可：

```
node_features = {'user': user_feats, 'item': item_feats}
h_dict = model(hetero_graph, {'user': user_feats, 'item': item_feats})
h_user = h_dict['user']
h_item = h_dict['item']
```

训练循环与同构图训练循环相同，不同之处在于，现在您有了一个节点表示的字典，并且可以从该字典计算预测结果。 例如，如果仅预测 user 节点，则可以从返回的字典中提取 user 节点嵌入：

```
opt = torch.optim.Adam(model.parameters())

for epoch in range(5):
    model.train()
    # forward propagation by using all nodes and extracting the user embeddings
    logits = model(hetero_graph, node_features)['user']
```

```
        # compute loss
        loss = F.cross_entropy(logits[train_mask], labels[train_mask])
        # Compute validation accuracy.  Omitted in this example.
        # backward propagation
        opt.zero_grad()
        loss.backward()
        opt.step()
        print(loss.item())

        # Save model if necessary.  Omitted in the example.
```

DGL 提供了用于节点分类的 RGCN 的端到端示例。 您可以在模型实现文件的 RelGraphConvLayer 中看到异构图卷积的定义。

5.4 边分类/回归

有时您希望预测图边上的属性，或者甚至预测两个给定节点之间是否存在边。在这种情况下，您想要一个边分类/回归模型。

在这里，我们生成一个用于边预测的随机图作为演示。

```
    src = np.random.randint(0, 100, 500)
    dst = np.random.randint(0, 100, 500)
    # make it symmetric
    edge_pred_graph = dgl.graph((np.concatenate([src, dst]), np.concatenate([dst,
src])))
    # synthetic node and edge features, as well as edge labels
    edge_pred_graph.ndata['feature'] = torch.randn(100, 10)
    edge_pred_graph.edata['feature'] = torch.randn(1000, 10)
    edge_pred_graph.edata['label'] = torch.randn(1000)
    # synthetic train-validation-test splits
    edge_pred_graph.edata['train_mask'] = torch.zeros(1000,
dtype=torch.bool).bernoulli(0.6)
```

5.4.1 总览

从上一节中，您学习了如何使用多层 GNN 进行节点分类。 可以将相同技术应用于计算任何节点的隐藏表示。 然后可以从其入射节点的表示中得出边的预测。

对边进行预测最常见的情况是将其表示为-其入射节点表示的-参数化函数，以及可选的边本身的特征。

5.4.2 模型实现和节点分类的区别

假设您使用上一部分中的模型计算节点表示，则只需要编写另一个使用 apply_edges()方法计算边预测的组件。

例如，如果您想为每个边计算分数以进行边回归，则以下代码将计算每个边上入射节点表示的点积。

```
    import dgl.function as fn
    class DotProductPredictor(nn.Module):
```

```
    def forward(self, graph, h):
        # h contains the node representations computed from the GNN above.
        with graph.local_scope():
            graph.ndata['h'] = h
            graph.apply_edges(fn.u_dot_v('h', 'h', 'score'))
            return graph.edata['score']
```

还可以编写一种预测函数，该函数使用 MLP 预测每个边的向量。 这个向量可用于进一步的下游任务，例如 logits of a categorical distribution。

```
class MLPPredictor(nn.Module):
    def __init__(self, in_features, out_classes):
        super().__init__()
        self.W = nn.Linear(in_features * 2, out_classes)

    def apply_edges(self, edges):
        h_u = edges.src['h']
        h_v = edges.dst['h']
        score = self.W(torch.cat([h_u, h_v], 1))
        return {'score': score}

    def forward(self, graph, h):
        # h contains the node representations computed from the GNN above.
        with graph.local_scope():
            graph.ndata['h'] = h
            graph.apply_edges(self.apply_edges)
            return graph.edata['score']
```

5.4.3 训练循环

给定节点表示计算模型和边预测器模型，我们可以轻松编写一个全图训练循环，在其中计算所有边的预测。

以下示例将上一节中的 SAGE 作为节点表示计算模型，并将 DotPredictor 作为边预测器模型。

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.sage = SAGE(in_features, hidden_features, out_features)
        self.pred = DotProductPredictor()
    def forward(self, g, x):
        h = self.sage(g, x)
        return self.pred(g, h)
```

在此示例中，我们还假定训练/验证/测试边集由边上的布尔掩码标识。 此示例也不包括提前停止和模型保存。

```
node_features = edge_pred_graph.ndata['feature']
edge_label = edge_pred_graph.edata['label']
train_mask = edge_pred_graph.edata['train_mask']
```

```
model = Model(10, 20, 5)
opt = torch.optim.Adam(model.parameters())
for epoch in range(10):
    pred = model(edge_pred_graph, node_features)
    loss = ((pred[train_mask] - edge_label[train_mask]) ** 2).mean()
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())
```

5.4.4 异构图

异构图上的边分类与同类图上的边分类没有太大区别。 如果希望对一种边类型执行边分类，则只需要计算所有节点类型的节点表示形式，并使用apply_edges()方法对该边类型进行预测。

例如，要使 DotProductPredictor 在异构图的一种边类型上工作，只需在 apply_edges 方法中指定边类型。

```
class HeteroDotProductPredictor(nn.Module):
    def forward(self, graph, h, etype):
        # h contains the node representations for each edge type computed from
        # the GNN above.
        with graph.local_scope():
            graph.ndata['h'] = h   # assigns 'h' of all node types in one shot
            graph.apply_edges(fn.u_dot_v('h', 'h', 'score'), etype=etype)
            return graph.edges[etype].data['score']
```

您可以类似地编写一个 HeteroMLPPredictor。

```
class MLPPredictor(nn.Module):
    def __init__(self, in_features, out_classes):
        super().__init__()
        self.W = nn.Linear(in_features * 2, out_classes)

    def apply_edges(self, edges):
        h_u = edges.src['h']
        h_v = edges.dst['h']
        score = self.W(torch.cat([h_u, h_v], 1))
        return {'score': score}

    def forward(self, graph, h, etype):
        # h contains the node representations computed from the GNN above.
        with graph.local_scope():
            graph.ndata['h'] = h   # assigns 'h' of all node types in one shot
            graph.apply_edges(self.apply_edges, etype=etype)
            return graph.edges[etype].data['score']
```

预测单个边类型上每个边得分的端到端模型将如下所示：

```
class Model(nn.Module):
```

```
    def __init__(self, in_features, hidden_features, out_features, rel_names):
        super().__init__()
        self.sage = RGCN(in_features, hidden_features, out_features, rel_names)
        self.pred = HeteroDotProductPredictor()
    def forward(self, g, x, etype):
        h = self.sage(g, x)
        return self.pred(g, h, etype)
```

使用模型仅涉及向模型提供节点类型和特征的字典。

```
model = Model(10, 20, 5, hetero_graph.etypes)
user_feats = hetero_graph.nodes['user'].data['feature']
item_feats = hetero_graph.nodes['item'].data['feature']
label = hetero_graph.edges['click'].data['label']
train_mask = hetero_graph.edges['click'].data['train_mask']
node_features = {'user': user_feats, 'item': item_feats}
```

然后，训练循环看起来与同构图中的几乎相同。 例如，如果您希望预测边类型 click 上的边标签，则只需:

```
opt = torch.optim.Adam(model.parameters())
for epoch in range(10):
    pred = model(hetero_graph, node_features, 'click')
    loss = ((pred[train_mask] - label[train_mask]) ** 2).mean()
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())
```

5.4.5 预测异构图上现有边的边类型

有时您可能想预测现有边线属于哪种类型。

例如，给定上面的异构图，您的任务将获得一条连接 user 和 item 的边，预测 user 是 click 还是 dislike 该项目。

这是评级预测的简化版本，在推荐文献中很常见。

您可以使用异构图卷积网络来获取节点表示。 例如，您仍然可以将上面的 RGCN 用于此目的。

要预测边的类型，您可以简单地重新利用上面的 HeteroDotProductPredictor，以便它使用另一种仅包含一个边类型的图来“合并”所有要预测的边类型，并为每个边发出每种类型的得分。

在此处的示例中，您将需要一个图，该图具有两种节点类型：user 和 item，以及一个单一的边类型，用于“合并”user 和 item 中的所有边类型，即“like”和“dislike”。 这可以使用关系切片方便地创建。

```
dec_graph = hetero_graph['user', :, 'item']
```

由于上面的语句还返回了原始边类型，这种边类型是名为 dgl.ETYPE 的特征，因此我们可以将其用作标签。

```
edge_label = dec_graph.edata[dgl.ETYPE]
```

给定上图作为边类型预测器模块的输入，您可以如下编写预测器模块。

```
class HeteroMLPPredictor(nn.Module):
```

```
    def __init__(self, in_dims, n_classes):
        super().__init__()
        self.W = nn.Linear(in_dims * 2, n_classes)

    def apply_edges(self, edges):
        x = torch.cat([edges.src['h'], edges.dst['h']], 1)
        y = self.W(x)
        return {'score': y}

    def forward(self, graph, h):
        # h contains the node representations for each edge type computed from
        # the GNN above.
        with graph.local_scope():
            graph.ndata['h'] = h   # assigns 'h' of all node types in one shot
            graph.apply_edges(self.apply_edges)
            return graph.edata['score']
```

结合了节点表示模块和边类型预测器模块的模型如下：

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, rel_names):
        super().__init__()
        self.sage = RGCN(in_features, hidden_features, out_features, rel_names)
        self.pred = HeteroMLPPredictor(out_features, len(rel_names))
    def forward(self, g, x, dec_graph):
        h = self.sage(g, x)
        return self.pred(dec_graph, h)
```

然后，训练循环如下：

```
model = Model(10, 20, 5, hetero_graph.etypes)
user_feats = hetero_graph.nodes['user'].data['feature']
item_feats = hetero_graph.nodes['item'].data['feature']
node_features = {'user': user_feats, 'item': item_feats}

opt = torch.optim.Adam(model.parameters())
for epoch in range(10):
    logits = model(hetero_graph, node_features, dec_graph)
    loss = F.cross_entropy(logits, edge_label)
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())
```

DGL 提供了 Graph Convolutional Matrix Completion 作为等级预测的一个示例，该等级是通过预测异构图上现有边的类型来制定的。 model implementation file 中的节点表示模块称为 GCMCLayer。 边类型预测器模块称为 BiDecoder。 两者都比此处描述的设置复杂。

5.3 链接预测

在其他一些设置中，您可能希望预测两个给定节点之间是否存在边。 这种模型称为链接预测模型。

5.3.1 总览

基于 GNN 的链接预测模型将两个节点 u 和 v 之间的连通性的可能性表示为$h_u^{(L)}$和$h_v^{(L)}$的函数，它们的节点表示是从多层 GNN 计算得出的 。

$$y_{u,v} = \phi(\boldsymbol{h}_u^{(L)}, \boldsymbol{h}_v^{(L)})$$

在本节中，我们将 $y_{u,v}$ 称为节点 u 和节点 v 之间的分数。

训练链接预测模型涉及将通过边连接的节点之间的分数与任意一对节点之间的分数进行比较。 例如，给定一条连接 u 和 v 的边，我们鼓励节点 u 和 v 之间的得分高于从任意噪声分布 $v' \sim P_n(v)$中采样的节点 u 和采样节点 v'之间的得分。这种方法称为负采样。

有许多损失函数可以实现上述最小化行为。非详尽的清单包括：

Cross-entropy loss: $\mathcal{L} = -\log \sigma(y_{u,v}) - \sum_{v_i \sim P_n(v), i=1,\ldots,k} \log[1 - \sigma(y_{u,v_i})]$

BPR loss: $\mathcal{L} = \sum_{v_i \sim P_n(v), i=1,\ldots,k} -\log \sigma(y_{u,v} - y_{u,v_i})$

Margin loss: $\mathcal{L} = \sum_{v_i \sim P_n(v), i=1,\ldots,k} \max(0, M - y_{u,v} + y_{u,v_i})$, where M is a constant hyperparameter.

如果您知道隐式反馈或噪声对比估计是什么，您可能会觉得这个想法很熟悉。

5.3.2 与边分类模型实现的区别

用于计算 u 和 v 之间得分的神经网络模型与上述边回归模型相同。

这是一个使用点积计算边得分的示例。

```
class DotProductPredictor(nn.Module):
    def forward(self, graph, h):
        # h contains the node representations computed from the GNN above.
        with graph.local_scope():
            graph.ndata['h'] = h
            graph.apply_edges(fn.u_dot_v('h', 'h', 'score'))
            return graph.edata['score']
```

5.3.3 训练循环

因为我们的分数预测模型是在图上运行的，所以我们需要将负样本表示为另一个图。 该图将包含作为边的所有负节点对。

```
def construct_negative_graph(graph, k):
    src, dst = graph.edges()

    neg_src = src.repeat_interleave(k)
    neg_dst = torch.randint(0, graph.number_of_nodes(), (len(src) * k,))
```

```
        return dgl.graph((neg_src, neg_dst), num_nodes=graph.number_of_nodes())
```

预测边得分的模型与边分类/回归的模型相同。

```
    class Model(nn.Module):
        def __init__(self, in_features, hidden_features, out_features):
            super().__init__()
            self.sage = SAGE(in_features, hidden_features, out_features)
            self.pred = DotProductPredictor()
        def forward(self, g, neg_g, x):
            h = self.sage(g, x)
            return self.pred(g, h), self.pred(neg_g, h)
```

训练循环类似于同构图。

```
    def compute_loss(pos_score, neg_score):
        # Margin loss
        n_edges = pos_score.shape[0]
        return (1 - neg_score.view(n_edges, -1) +
pos_score.unsqueeze(1)).clamp(min=0).mean()


    k = 5
    model = Model(10, 20, 5, hetero_graph.etypes)
    user_feats = hetero_graph.nodes['user'].data['feature']
    item_feats = hetero_graph.nodes['item'].data['feature']
    node_features = {'user': user_feats, 'item': item_feats}
    opt = torch.optim.Adam(model.parameters())
    for epoch in range(10):
        negative_graph = construct_negative_graph(hetero_graph, k, ('user', 'click',
'item'))
        pos_score, neg_score = model(hetero_graph, negative_graph, node_features,
('user', 'click', 'item'))
        loss = compute_loss(pos_score, neg_score)
        opt.zero_grad()
        loss.backward()
        opt.step()
        print(loss.item())
```

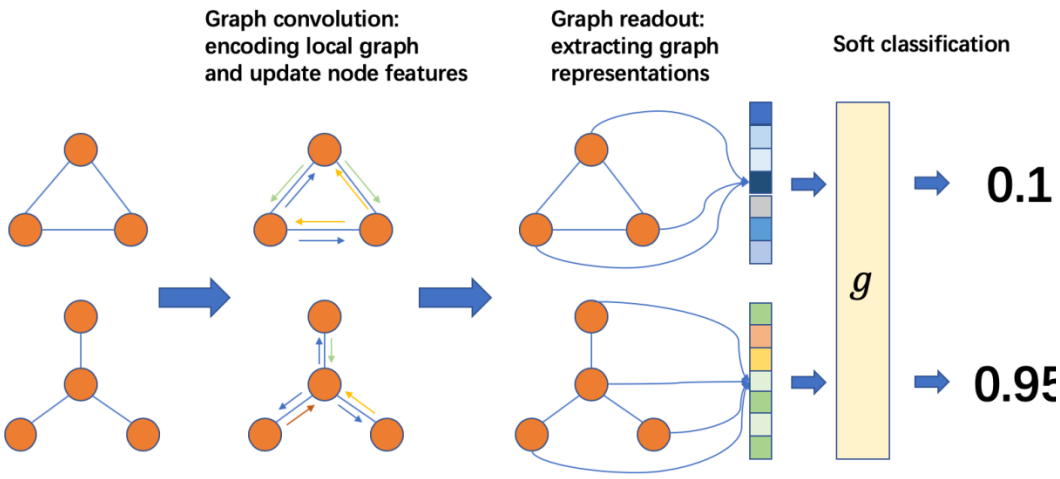
5.4 图分类

有时候，我们可能会以多个图的形式获得数据，而不是一个大图，例如，一系列不同类型的人的社区。 通过用图表描述同一社区中人们之间的友谊，我们得到了一系列图表进行分类。 在这种情况下，图分类模型可以帮助识别社区的类型，即根据结构和整体信息对每个图进行分类。

5.4.1 总览

图分类与节点分类或链接预测之间的主要区别在于，预测结果表征了整个输入图的特征。 我们像以前的任务一样，通过节点/边执行消息传递，但是还尝试检索图级表示。

图分类如下：



图分类过程。

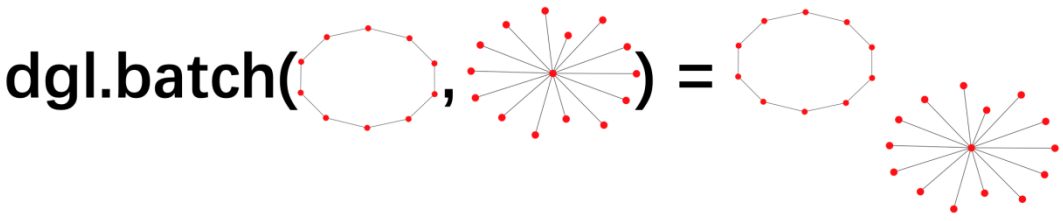
从左到右，通常的做法是：

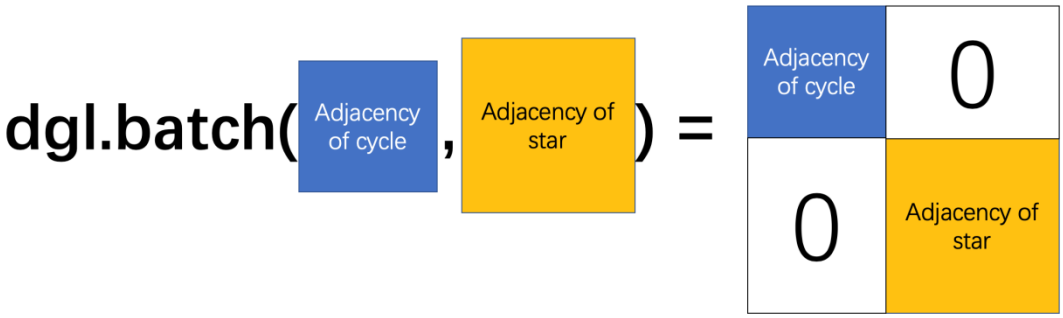
- 将图准备为一批图
- 消息在批处理图上传递以更新节点/边特征
- 将节点/边特征聚合为图级表示
- 任务分类负责人

5.4.2 图批量

通常，一个图分类任务会训练很多图，如果在训练模型时一次只使用一个图，则效率将非常低下。 从常见的深度学习实践中借用 minibatch 训练的想法，我们可以构建一批多个图并将它们一起发送以进行一次训练迭代。

在 DGL 中，我们可以构建一系列图的单个批处理图。 该批处理图可以简单地用作单个大图，其中分离的组件代表相应的原始小图。





5.4.3 图读出

数据中的每个图都可能有其独特的结构，以及节点和边的特征。为了做出一个单一的预测，我们通常对可能丰富的信息进行聚合和总结。这种类型的操作称为读出。常用的聚合包括所有节点或边特征的 summation, average, maximum or minimum。

$$h_g = \frac{1}{|\mathcal{V}|}\sum_{v\in\mathcal{V}} h_v$$

在 DGL 中，相应的函数调用是 dgl.readout_nodes()。

一旦 h_g 可用，我们就可以将其传递到 MLP 层以进行分类输出。

5.4.4 编写神经网络模型

模型的输入是具有节点和边特征的批处理图。 要注意的一件事是，批处理图中的节点和边特征没有批处理维。 模型中应特别注意以下几点：

- 批处理图上的计算

接下来，我们讨论批处理图的计算属性。

首先，一批中的不同图完全分开，即没有边连接两个图。 有了这个不错的属性，所有消息传递函数仍然具有相同的结果。

其次，将对每个图分别执行批处理图上的读取功能。 假设批次大小为 B，要聚合的特征尺寸为 D，则读取结果的形状将为（B，D）。

```
g1 = dgl.graph(([0, 1], [1, 0]))
g1.ndata['h'] = torch.tensor([1., 2.])
g2 = dgl.graph(([0, 1], [1, 2]))
g2.ndata['h'] = torch.tensor([1., 2., 3.])

dgl.readout_nodes(g1, 'h')
# tensor([3.])  # 1 + 2

bg = dgl.batch([g1, g2])
dgl.readout_nodes(bg, 'h')
# tensor([3., 6.])  # [1 + 2, 1 + 2 + 3]
```

最后，批处理图上的每个节点/边特征张量采用将所有图上对应的特征张量串联起来的形式。

```
bg.ndata['h']
# tensor([1., 2., 1., 2., 3.])
```

- 模型定义

了解了上述计算规则后，我们可以定义一个非常简单的模型。

```
class Classifier(nn.Module):
    def __init__(self, in_dim, hidden_dim, n_classes):
        super(Classifier, self).__init__()
        self.conv1 = dglnn.GraphConv(in_dim, hidden_dim)
```

```
        self.conv2 = dglnn.GraphConv(hidden_dim, hidden_dim)
        self.classify = nn.Linear(hidden_dim, n_classes)

    def forward(self, g, feat):
        # Apply graph convolution and activation.
        h = F.relu(self.conv1(g, h))
        h = F.relu(self.conv2(g, h))
        with g.local_scope():
            g.ndata['h'] = h
            # Calculate graph representation by average readout.
            hg = dgl.mean_nodes(g, 'h')
            return self.classify(hg)
```

5.4.5 训练循环

（1） 数据加载

定义模型后，我们就可以开始训练了。 由于图分类处理的是很多相对较小的图，而不是一个大图，因此我们通常可以在随机的 mini-batches 图上进行有效的训练，而无需设计复杂的图采样算法。

假设我们有第 4 章：图数据管道中介绍的图分类数据集。

```
import dgl.data
dataset = dgl.data.GINDataset('MUTAG', False)
```

图分类数据集中的每一项都是一对图及其标签。 我们可以通过利用 DataLoader，自定义 collate 函数来批量处理图来加快数据加载过程：

```
def collate(samples):
    graphs, labels = map(list, zip(*samples))
    batched_graph = dgl.batch(graphs)
    batched_labels = torch.tensor(labels)
    return batched_graph, batched_labels
```

然后可以创建一个 DataLoader，用来在小批处理中迭代图的数据集。

```
from torch.utils.data import DataLoader
dataloader = DataLoader(
    dataset,
    batch_size=1024,
    collate_fn=collate,
    drop_last=False,
    shuffle=True)
```

（2） 循环

然后，训练循环仅涉及遍历数据加载器并更新模型。

```
model = Classifier(10, 20, 5)
opt = torch.optim.Adam(model.parameters())
for epoch in range(20):
    for batched_graph, labels in dataloader:
        feats = batched_graph.ndata['feats']
```

```
        logits = model(batched_graph, feats)
        loss = F.cross_entropy(logits, labels)
        opt.zero_grad()
        loss.backward()
        opt.step()
```

DGL 实现了 GIN 作为图分类的示例。 训练循环位于 `main.py<https://github.com/dmlc/dgl/blob/master/examples/pytorch/gin/main.py>`__ 中的函数训练中。 该模型实现位于 gin.py<https://github.com/dmlc/dgl/blob/master/examples/pytorch/gin/gin.py>`__ 内部，其中包含更多组件，例如使用 dgl.nn.pytorch。 GINConv（也可在 MXNet 和 Tensorflow 中使用）作为图卷积层，批量归一化等。

5.4.6 异构图

异构图的图分类与同构图的图分类略有不同。 除了需要异构图卷积模块外，你还需要在 eadout function.中的不同类型的节点上进行聚合。

下面显示了对每种节点类型的节点表示的平均值进行求和的示例。

```
class RGCN(nn.Module):
    def __init__(self, in_feats, hid_feats, out_feats, rel_names):
        super().__init__()

        self.conv1 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(in_feats, hid_feats)
            for rel in rel_names}, aggregate='sum')
        self.conv2 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(hid_feats, out_feats)
            for rel in rel_names}, aggregate='sum')

    def forward(self, graph, inputs):
        # inputs are features of nodes
        h = self.conv1(graph, inputs)
        h = {k: F.relu(v) for k, v in h.items()}
        h = self.conv2(graph, h)
        return h

class HeteroClassifier(nn.Module):
    def __init__(self, in_dim, hidden_dim, n_classes, rel_names):
        super().__init__()

        self.conv1 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(in_feats, hid_feats)
            for rel in rel_names}, aggregate='sum')
        self.conv2 = dglnn.HeteroGraphConv({
```

```
            rel: dglnn.GraphConv(hid_feats, out_feats)
            for rel in rel_names}, aggregate='sum')
        self.classify = nn.Linear(hidden_dim, n_classes)

    def forward(self, g):
        h = g.ndata['feat']
        # Apply graph convolution and activation.
        h = F.relu(self.conv1(g, h))
        h = F.relu(self.conv2(g, h))

        with g.local_scope():
            g.ndata['h'] = h
            # Calculate graph representation by average readout.
            hg = 0
            for ntype in g.ntypes:
                hg = hg + dgl.mean_nodes(g, 'h', ntype=ntype)
            return self.classify(hg)
```

其余代码与同构图相同。

```
model = HeteroClassifier(10, 20, 5)
opt = torch.optim.Adam(model.parameters())
for epoch in range(20):
    for batched_graph, labels in dataloader:
        logits = model(batched_graph)
        loss = F.cross_entropy(logits, labels)
        opt.zero_grad()
        loss.backward()
        opt.step()
```

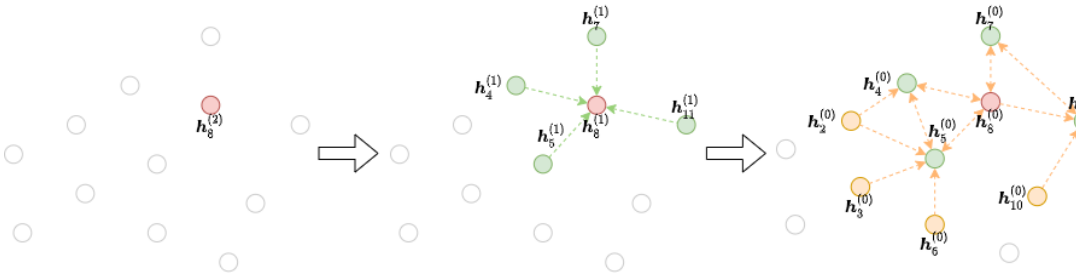
6 大图的随机训练

如果我们有一个包含数百万甚至数十亿个节点或边的大规模图，那么通常无法按照第 5 章：训练图神经网络中所述进行全图训练。 考虑一个运行在 N 个节点图上的，具有隐藏状态大小 H 的 L 层图卷积网络。 存储中间隐藏状态需要 O(NLH)内存，很容易超过一个具有大小为 N 的 GPU 的容量。

本节提供了一种执行随机 minibatch 训练的方法，其中我们不必将所有节点的特征都适合 GPU。

邻域抽样方法概述

邻域抽样方法通常如下所示。对于每一个梯度下降步骤，我们选择一小批节点，这些节点在第 L 层的最终表示需要计算。然后我们取它们在 L - 1L 层的所有或一些相邻的节点。这个过程一直持续到我们到达输入点为止。这个迭代过程从输出开始建立依赖关系图，并向后工作到输入，如下图所示:



这样一来，可以节省用于在大型图上训练 GNN 的工作量和计算资源。

DGL 提供了一些邻域采样器和用于通过邻域采样训练 GNN 的管道，以及自定义采样策略的方法。

6.3 通过邻域采样训练 GNN 以进行节点分类

为了使您的模型进行随机训练，您需要执行以下操作：

- 定义邻域采样器。
- 调整模型以进行 minibatch 训练。
- 修改您的训练循环。

以下小节逐一介绍了这些步骤。

6.3.1 定义邻域采样器和数据加载器

DGL 提供了几个邻域采样器类，这些类在给定我们希望在其上进行计算的节点的情况下生成每一层所需的计算依赖项。

最简单的邻域采样器是 MultiLayerFullNeighborSampler，它使节点从其所有邻域收集消息。

要使用 DGL 提供的采样器，还需要将其与 NodeDataLoader 结合使用，后者可以在 minibatches 中的一组节点上进行迭代。

例如，以下代码创建了一个 PyTorch DataLoader，它分批迭代训练节点 ID 数组 train_nids，并将生成的块列表放到 GPU 上。

```
import dgl
import dgl.nn as dglnn
import torch
import torch.nn as nn
import torch.nn.functional as F

sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
dataloader = dgl.dataloading.NodeDataLoader(
    g, train_nids, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

在 DataLoader 上迭代将生成一列专门创建的图，表示每个层上的计算依赖关系。在 DGL 中称为 block。

```
input_nodes, output_nodes, blocks = next(iter(dataloader))
```

```
print(blocks)
```

迭代器一次生成三个项。input_nodes 描述了计算 output_nodes 表示所需的节点。块描述了每个 GNN 层将计算哪些节点表示作为输出，哪些节点表示作为输入，以及输入节点的表示如何传播到输出节点。

有关支持的内建采样器的完整列表，请参考邻域采样器 API 参考。

如果您希望开发您自己的邻域采样器，或您想更详细地解释块的概念，请参阅 6.4 定制邻域采样器。

6.3.2 为 minibatch 训练调整你的模型

如果您的消息传递模块都是由 DGL 提供的，那么使您的模型适应 minibatch 训练所需的更改是最小的。以多层 GCN 为例。如果你的全图模型实现如下:

```
class TwoLayerGCN(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.conv1 = dglnn.GraphConv(in_features, hidden_features)
        self.conv2 = dglnn.GraphConv(hidden_features, out_features)

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

然后，您所需要的就是用上面生成的块替换 g。

```
class StochasticTwoLayerGCN(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.conv1 = dgl.nn.GraphConv(in_features, hidden_features)
        self.conv2 = dgl.nn.GraphConv(hidden_features, out_features)

    def forward(self, blocks, x):
        x = F.relu(self.conv1(blocks[0], x))
        x = F.relu(self.conv2(blocks[1], x))
        return x
```

上面的 DGL GraphConv 模块接受数据加载器生成的块中的一个元素作为参数。

每个 NN 模块的 API 引用将告诉您它是否支持接受块作为参数。

如果您希望使用自己的消息传递模块，请参阅 6.5 实现自定义 GNN 模块进行 minibatch 训练。

6.3.3 训练循环

训练循环只包含使用定制的批处理迭代器对数据集进行迭代。在产生一组块的每个迭代中，

- 我们:将输入节点对应的节点特征加载到 GPU 上。节点特征可以存储在内存或外部存储中。请注意，我们只需要加载输入节点的特征，而不是像在完整图训练中那样加载所有节点的特征。如果特征存储在 g.ndata 中，那么可以通过访问 blocks[0].srcdata 中的特征来加载特征。第一个块的输

入节点的特征，它与计算最终表示所需的所有节点相同。

- 将块列表和输入节点特征提供给多层 GNN 并获得输出。
- 将与输出节点对应的节点标签加载到 GPU 上。类似地，节点标签可以存储在内存或外部存储中。同样，请注意，我们只需要加载输出节点的标签，而不需要像在完整图训练中那样加载所有节点的标签。如果特征存储在 g.ndata 中，那么可以通过利用 blocks[-1]srcdata 特征来加载标签。最后一个块的输出节点的特征，它与我们希望计算最终表示的节点相同。
- 计算损失和反向传播。

```
model = StochasticTwoLayerGCN(in_features, hidden_features, out_features)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())

for input_nodes, output_nodes, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    input_features = blocks[0].srcdata['features']
    output_labels = blocks[-1].dstdata['label']
    output_predictions = model(blocks, input_features)
    loss = compute_loss(output_labels, output_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

DGL 提供了一个端到端的随机训练示例 GraphSAGE 实现。
https://github.com/dmlc/dgl/blob/master/examples/pytorch/graphsage/train_sampling.py

6.3.4 对于异构图

对异构图进行节点分类训练的图神经网络是相似的。

例如，我们之前已经了解了如何在全图上训练一个 2 层 RGCN。在 minibatch 训练中实现 RGCN 的代码与此非常相似(为了简单起见，去掉了自循环、非线性和基分解):

```
class StochasticTwoLayerRGCN(nn.Module):
    def __init__(self, in_feat, hidden_feat, out_feat):
        super().__init__()
        self.conv1 = dglnn.HeteroGraphConv({
                rel : dglnn.GraphConv(in_feat, hidden_feat, norm='right')
                for rel in rel_names
            })
        self.conv2 = dglnn.HeteroGraphConv({
                rel : dglnn.GraphConv(hidden_feat, out_feat, norm='right')
                for rel in rel_names
            })

    def forward(self, blocks, x):
        x = self.conv1(blocks[0], x)
```

```
        x = self.conv2(blocks[1], x)
        return x
```

DGL 提供的一些采样器也支持异构图。例如，仍然可以使用提供的 MultiLayerFullNeighborSampler 类和 NodeDataLoader 类进行随机训练。对于全邻域采样，唯一的区别是您将为训练集指定节点类型和节点 id 的字典。

```
sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
dataloader = dgl.dataloading.NodeDataLoader(
    g, train_nid_dict, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

训练循环与同构图的训练循环几乎相同，除了 compute_loss 的实现将接受两个节点类型和预测字典之外。

```
model = StochasticTwoLayerRGCN(in_features, hidden_features, out_features)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())

for input_nodes, output_nodes, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    input_features = blocks[0].srcdata     # returns a dict
    output_labels = blocks[-1].dstdata     # returns a dict
    output_predictions = model(blocks, input_features)
    loss = compute_loss(output_labels, output_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

DGL 提供了一个端到端的随机训练实例 RGCN 实现。

https://github.com/dmlc/dgl/blob/master/examples/pytorch/rgcn-hetero/entity_classify_mb.py

6.4 训练利用邻域采样进行边分类的 GNN

边分类/回归的训练与节点分类/回归的训练有些相似，但有几个显著的区别。

6.4.1 定义邻域采样器和数据加载器

您可以使用与节点分类相同的邻域采样器。

```
sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
```

要使用 DGL 提供的邻域采样器进行边分类，需要将其与 EdgeDataLoader 结合使用，后者在 minibatch 中迭代一组边，生成由 edge minibatch 生成的子图和上面模块消耗的块。

例如，下面的代码创建了一个 PyTorch DataLoader，它分批迭代训练边 ID 数组 train_eids，并将生成的块列表放到 GPU 上。

```
dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_eid_dict, sampler,
```

```
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

有关受支持的内置采样器的完整列表，请参阅邻域采样器 API 参考。

如果您希望开发自己的邻域采样器，或者想要对块的概念进行更详细的说明，请参阅 6.4 自定义邻域采样器。

6.4.2 从原始图中移除 minibatch 中的边以进行邻域采样

在训练边分类模型时，有时您希望从计算依赖性中删除出现在训练数据中的边，就好像它们根本不存在一样。 否则，模型将“知道”两个节点之间存在边的事实，并有可能利用它来获得优势。

因此，在边分类中，您有时会希望从原始图中排除在 minibatches 中采样的边，以及在无向图中的采样边的反向边，以进行邻域采样。 您可以在 EdgeDataLoader 的实例化中指定 exclude ='reverse'，同时将边 ID 映射到其反向边 ID。通常这样做会导致采样过程变慢，因为要定位涉及到小批的反向边并去除它们。

```
n_edges = g.number_of_edges()
dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_eid_dict, sampler,

    # The following two arguments are specifically for excluding the minibatch
    # edges and their reverse edges from the original graph for neighborhood
    # sampling.
    exclude='reverse',
    reverse_eids=torch.cat([
        torch.arange(n_edges // 2, n_edges), torch.arange(0, n_edges // 2)]),

    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

6.4.3 调整模型以进行 minibatch 训练

边分类模型通常由两部分组成:

- 一部分是获取入射节点表示。
- 另一部分是根据入射节点表示计算边分数。

前一部分与节点分类中的部分完全相同，我们可以简单地重用它。输入仍然是由 DGL 提供的数据加载器生成的块列表以及输入特征。

```
class StochasticTwoLayerGCN(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.conv1 = dglnn.GraphConv(in_features, hidden_features)
        self.conv2 = dglnn.GraphConv(hidden_features, out_features)
```

```
    def forward(self, blocks, x):
        x = F.relu(self.conv1(blocks[0], x))
        x = F.relu(self.conv2(blocks[1], x))
        return x
```

后一部分的输入通常是前一部分的输出，as well as the subgraph of the original graph induced by the edges in the minibatch。子图是由相同的数据加载器生成的。可以调用 dgl.DGLHeteroGraph.apply_edges()来计算边子图上的边的分数。

下面的代码显示了通过拼接入射节点特征并将其投射到稠密层来预测边上的得分的示例。

```
class ScorePredictor(nn.Module):
    def __init__(self, num_classes, in_features):
        super().__init__()
        self.W = nn.Linear(2 * in_features, num_classes)

    def apply_edges(self, edges):
        data = torch.cat([edges.src['x'], edges.dst['x']])
        return {'score': self.W(data)}

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

整个模型将采用数据加载程序生成的块列表和边子图以及输入节点特征，如下所示：

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, num_classes):
        super().__init__()
        self.gcn = StochasticTwoLayerGCN(
            in_features, hidden_features, out_features)
        self.predictor = ScorePredictor(num_classes, out_features)

    def forward(self, edge_subgraph, blocks, x):
        x = self.gcn(blocks, x)
        return self.predictor(edge_subgraph, x)
```

DGL 确保边子图中的节点与生成的块列表中最后一个块的输出节点相同。

6.4.4 训练循环

训练循环非常类似于节点分类。您可以遍历 dataloader 并获得由 minibatch 中的边生成的子图，以及计算其入射节点表示所需的块列表。

```
model = Model(in_features, hidden_features, out_features, num_classes)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())
```

```
for input_nodes, edge_subgraph, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    edge_subgraph = edge_subgraph.to(torch.device('cuda'))
    input_features = blocks[0].srcdata['features']
    edge_labels = edge_subgraph.edata['labels']
    edge_predictions = model(edge_subgraph, blocks, input_features)
    loss = compute_loss(edge_labels, edge_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

6.4.5 对于异构图

计算异构图上的节点表示的模型也可以用于计算边分类/回归的入射节点表示。

```
class StochasticTwoLayerRGCN(nn.Module):
    def __init__(self, in_feat, hidden_feat, out_feat):
        super().__init__()
        self.conv1 = dglnn.HeteroGraphConv({
                rel : dglnn.GraphConv(in_feat, hidden_feat, norm='right')
                for rel in rel_names
            })
        self.conv2 = dglnn.HeteroGraphConv({
                rel : dglnn.GraphConv(hidden_feat, out_feat, norm='right')
                for rel in rel_names
            })

    def forward(self, blocks, x):
        x = self.conv1(blocks[0], x)
        x = self.conv2(blocks[1], x)
        return x
```

对于分数预测，同构图和异构图之间唯一的实现差异是我们遍历了apply_edges()的边类型。

```
class ScorePredictor(nn.Module):
    def __init__(self, num_classes, in_features):
        super().__init__()
        self.W = nn.Linear(2 * in_features, num_classes)

    def apply_edges(self, edges):
        data = torch.cat([edges.src['x'], edges.dst['x']])
        return {'score': self.W(data)}

    def forward(self, edge_subgraph, x):
```

```
        with edge_subgraph.local_scope():
            edge_subgraph.ndata['x'] = x
            for etype in edge_subgraph.canonical_etypes:
                edge_subgraph.apply_edges(self.apply_edges, etype=etype)
            return edge_subgraph.edata['score']
```

数据加载器的定义也与节点分类非常相似。 唯一的区别是您需要EdgeDataLoader而不是NodeDataLoader，并且将提供边类型和边ID张量的字典，而不是节点类型和节点ID张量的字典。

```
sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_eid_dict, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

如果您希望排除异构图的反向边，情况会有所不同。 在异构图上，反向边通常具有与边本身不同的边类型，以便区分“正向”和“反向”关系（例如，e.g. follow and followed by are reverse relations of each other, purchase and purchased by are reverse relations of each other, etc.）。

如果类型中的每个边具有与另一个类型中相同的 ID 的反向边，则可以指定边类型及其反向类型之间的映射。排除minibatch中的边及其反向边的方法如下。

```
dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_eid_dict, sampler,

    # The following two arguments are specifically for excluding the minibatch
    # edges and their reverse edges from the original graph for neighborhood
    # sampling.
    exclude='reverse_types',
    reverse_etypes={'follow': 'followed by', 'followed by': 'follow',
                    'purchase': 'purchased by', 'purchased by': 'purchase'}

    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

训练循环同样与同构图中的训练循环几乎相同，不同之处在于，compute_loss的实现将采用两个节点类型和预测字典。

```
model = Model(in_features, hidden_features, out_features, num_classes)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())

for input_nodes, edge_subgraph, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
```

```
    edge_subgraph = edge_subgraph.to(torch.device('cuda'))
    input_features = blocks[0].srcdata['features']
    edge_labels = edge_subgraph.edata['labels']
    edge_predictions = model(edge_subgraph, blocks, input_features)
    loss = compute_loss(edge_labels, edge_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

GCMC 是二部图上边分类的示例。

https://github.com/dmlc/dgl/tree/master/examples/pytorch/gcmc

6.5 通过邻域采样训练 GNN 进行链路预测

6.5.1 使用负采样定义邻域采样器和数据加载器

您仍然可以使用与节点/边分类中相同的邻域采样器。

```
sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
```

DGL 中的 EdgeDataLoader 也支持为链路预测生成阴性样本。为此，需要提供负采样函数。Uniform 是一个进行均匀采样的函数。对于每条边的每一个源节点，它抽取 k 个 negative destination nodes。

下面的数据加载器将为每条边的每个源节点均匀地选择 5 个 negative destination nodes。

```
dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_seeds, sampler,
    negative_sampler=dgl.dataloading.negative_sampler.Uniform(5),
    batch_size=args.batch_size,
    shuffle=True,
    drop_last=False,
    pin_memory=True,
    num_workers=args.num_workers)
```

有关内置的负采样器，请参见链接预测的负采样器。

https://docs.dgl.ai/api/python/dgl.dataloading.html#api-dataloading-negative-sampling

您也可以给定自己的负采样函数，只要它接受原始图 g 和 minibatch 边 ID 数组 eid，并返回一对源 ID 数组和目标 ID 数组即可。

下面给出了一个自定义负采样器的示例，该采样器根据与度数成正比的概率分布对负目标节点进行采样。

```
class NegativeSampler(object):
    def __init__(self, g, k):
        # caches the probability distribution
        self.weights = g.in_degrees().float() ** 0.75
        self.k = k

    def __call__(self, g, eids):
        src, _ = g.find_edges(eids)
```

```
        src = src.repeat_interleave(self.k)
        dst = self.weights.multinomial(len(src), replacement=True)
        return src, dst

dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_seeds, sampler,
    negative_sampler=NegativeSampler(g, 5),
    batch_size=args.batch_size,
    shuffle=True,
    drop_last=False,
    pin_memory=True,
    num_workers=args.num_workers)
```

6.5.2 调整模型以进行 minibatch 训练

如 5.3 链接预测中所述，通过比较边（正样本）与不存在边（负样本）的得分来训练链接预测。 要计算边的分数，您可以重用在边分类/回归中看到的节点表示计算模型。

```
class StochasticTwoLayerGCN(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.conv1 = dgl.nn.GraphConv(in_features, hidden_features)
        self.conv2 = dgl.nn.GraphConv(hidden_features, out_features)

    def forward(self, blocks, x):
        x = F.relu(self.conv1(blocks[0], x))
        x = F.relu(self.conv2(blocks[1], x))
        return x
```

对于分数预测，由于只需要预测每个边的标量分数而不是概率分布，因此本示例说明了如何使用入射节点表示的点积来计算分数。

```
class ScorePredictor(nn.Module):
    def forward(self, edge_subgraph, x):
        with edge_subgraph.local_scope():
            edge_subgraph.ndata['x'] = x
            edge_subgraph.apply_edges(dgl.function.u_dot_v('x',
'x', 'score'))
            return edge_subgraph.edata['score']
```

提供负采样器后，DGL 的数据加载器将为每个 minibatch 生成三项：

- 一个 positive 图，其中包含 minibatch 中采样的所有边。
- 一个 negative 图，其中包含由负采样器生成的所有不存在的边。
- 邻域采样器生成的块的列表。

因此，可以如下定义链接预测模型，该模型包含三个项以及输入特征。

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
```

```
        self.gcn = StochasticTwoLayerGCN(
            in_features, hidden_features, out_features)

    def forward(self, positive_graph, negative_graph, blocks, x):
        x = self.gcn(blocks, x)
        pos_score = self.predictor(positive_graph, x)
        neg_score = self.predictor(negative_graph, x)
        return pos_score, neg_score
```

6.5.3 训练循环

训练循环只涉及遍历数据加载器，并将图和输入特征输入到上面定义的模型中。

```
model = Model(in_features, hidden_features, out_features)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())

for input_nodes, positive_graph, negative_graph, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    positive_graph = positive_graph.to(torch.device('cuda'))
    negative_graph = negative_graph.to(torch.device('cuda'))
    input_features = blocks[0].srcdata['features']
    pos_score, neg_score = model(positive_graph, blocks, input_features)
    loss = compute_loss(pos_score, neg_score)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

DGL 提供了无监督学习 GraphSAGE，它显示了同类图上的链接预测示例。

https://github.com/dmlc/dgl/blob/master/examples/pytorch/graphsage/train_sampling_unsupervised.py

6.5.4 对于异构图

计算异构图上的节点表示的模型也可以用于计算边分类/回归的入射节点表示。

```
class StochasticTwoLayerRGCN(nn.Module):
    def __init__(self, in_feat, hidden_feat, out_feat):
        super().__init__()
        self.conv1 = dglnn.HeteroGraphConv({
                rel : dglnn.GraphConv(in_feat, hidden_feat, norm='right')
                for rel in rel_names
            })
        self.conv2 = dglnn.HeteroGraphConv({
                rel : dglnn.GraphConv(hidden_feat, out_feat, norm='right')
                for rel in rel_names
            })
```

```
    def forward(self, blocks, x):
        x = self.conv1(blocks[0], x)
        x = self.conv2(blocks[1], x)
        return x
```

对于分数预测，同构图和异构图之间唯一的实现差异是我们遍历dgl.DGLHeteroGraph.apply_edges()的边类型。

```
class ScorePredictor(nn.Module):
    def forward(self, edge_subgraph, x):
        with edge_subgraph.local_scope():
            edge_subgraph.ndata['x'] = x
            for etype in edge_subgraph.canonical_etypes:
                edge_subgraph.apply_edges(
                    dgl.function.u_dot_v('x', 'x', 'score'), etype=etype)
            return edge_subgraph.edata['score']
```

数据加载器的定义也与边分类/回归的定义非常相似。 唯一的区别是您需要提供负采样器，并且将提供边类型和边 ID 张量的字典，而不是节点类型和节点 ID 张量的字典。

```
sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_eid_dict, sampler,
    negative_sampler=dgl.dataloading.negative_sampler.Uniform(5),
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

如果要提供自己的负采样函数，则该函数应采用原始图以及边线类型和边线 ID 张量的字典。 它应该返回边类型和源-目标数组对的字典。 给出一个示例，如下所示：

```
class NegativeSampler(object):
    def __init__(self, g, k):
        # caches the probability distribution
        self.weights = {
            etype: g.in_degrees(etype=etype).float() ** 0.75
            for etype in g.canonical_etypes}
        self.k = k

    def __call__(self, g, eids_dict):
        result_dict = {}
        for etype, eids in eids_dict.items():
            src, _ = g.find_edges(eids, etype=etype)
            src = src.repeat_interleave(self.k)
            dst = self.weights.multinomial(len(src), replacement=True)
            result_dict[etype] = (src, dst)
```

```
        return result_dict

dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_eid_dict, sampler,
    negative_sampler=negative_sampler=NegativeSampler(g, 5),
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

训练循环同样与同构图中的训练循环几乎相同，不同之处在于，compute_loss 的实现将采用两个节点类型和预测字典。

```
model = Model(in_features, hidden_features, out_features, num_classes)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())

for input_nodes, positive_graph, negative_graph, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    positive_graph = positive_graph.to(torch.device('cuda'))
    negative_graph = negative_graph.to(torch.device('cuda'))
    input_features = blocks[0].srcdata['features']
    edge_labels = edge_subgraph.edata['labels']
    edge_predictions = model(edge_subgraph, blocks, input_features)
    loss = compute_loss(edge_labels, edge_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

6.6 自定义领域采样器

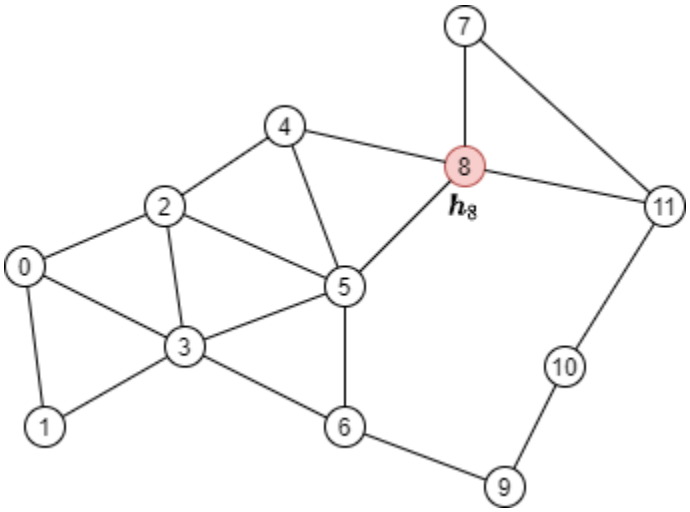
尽管 DGL 提供了一些邻域采样策略，但有时用户还是希望编写自己的采样策略。 本节说明如何编写自己的策略并将其插入到随机 GNN 训练框架中。

回想一下图神经网络的强大功能，消息传递的定义是：

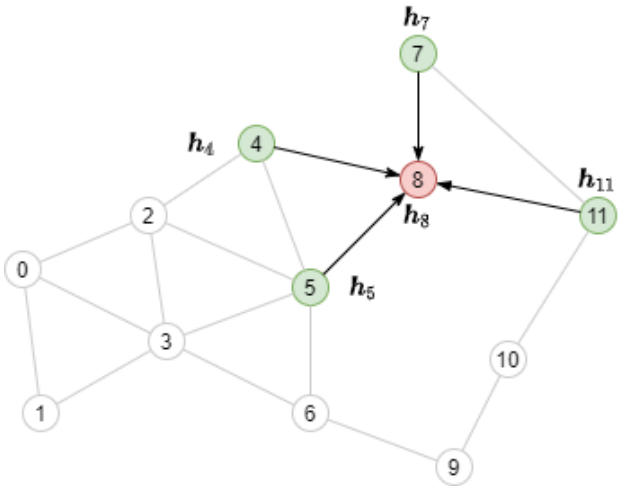
$$\boldsymbol{a}_v^{(l)} = \rho^{(l)}\left(\left\{\boldsymbol{h}_u^{(l-1)} : u \in \mathcal{N}(v)\right\}\right)$$
$$\boldsymbol{h}_v^{(l)} = \phi^{(l)}\left(\boldsymbol{h}_v^{(l-1)}, \boldsymbol{a}_v^{(l)}\right)$$

其中 $\rho^{(l)}$和 $\varphi^{(l)}$是参数化函数，并且 N(v)定义为图 G 上 v 的前辈（如果图是无向的，则是邻域）。

例如，要执行消息传递以更新下图中的红色节点：



需要聚合其邻域的节点特征，如绿色节点:



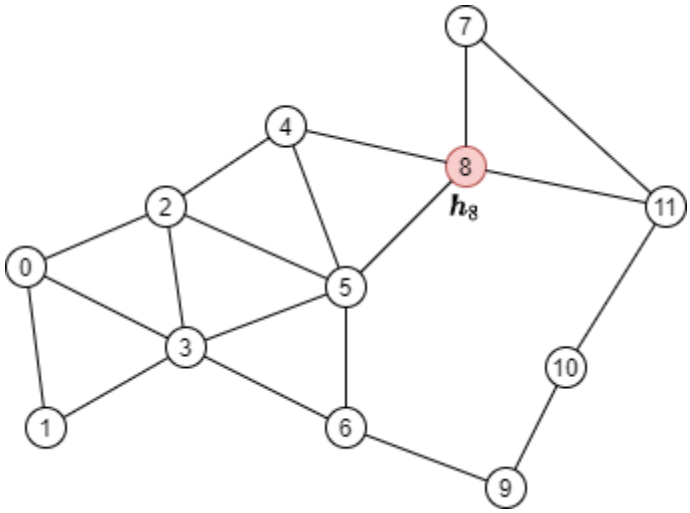
6.6.1 用 pencil and paper 进行邻域采样

然后，我们考虑多层消息传递如何工作以计算单个节点的输出。 在下文中，我们将要计算其 GNN 输出的节点称为种子节点。

```
import torch
import dgl

src = torch.LongTensor(
    [0, 0, 0, 1, 2, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 8, 9, 10,
     1, 2, 3, 3, 3, 4, 5, 5, 6, 5, 8, 6, 8, 9, 8, 11, 11, 10, 11])
dst = torch.LongTensor(
    [1, 2, 3, 3, 3, 4, 5, 5, 6, 5, 8, 6, 8, 9, 8, 11, 11, 10, 11,
     0, 0, 0, 1, 2, 2, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 8, 9, 10])
g = dgl.graph((src, dst))
g.ndata['x'] = torch.randn(12, 5)
g.ndata['y'] = torch.randn(12, 1)
```

6.6.2 查找消息传递依赖项

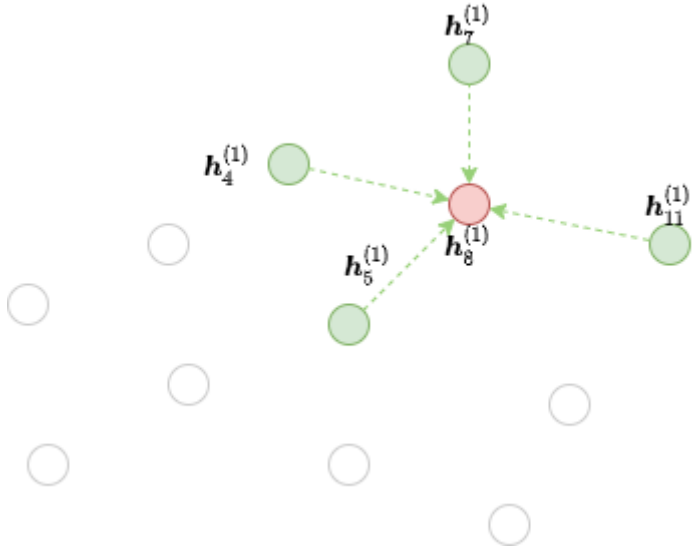
在下图中，考虑使用 2 层 GNN 计算种子节点 8（红色）的输出：



通过公式：

$$a_8^{(2)} = \rho^{(2)}\left(\left\{h_u^{(1)} : u \in \mathcal{N}(8)\right\}\right) = \rho^{(2)}\left(\left\{h_4^{(1)}, h_5^{(1)}, h_7^{(1)}, h_{11}^{(1)}\right\}\right)$$
$$h_8^{(2)} = \phi^{(2)}\left(h_8^{(1)}, a_8^{(2)}\right)$$

从公式可以看出，要计算$h_8^{(2)}$，我们需要来自节点 4、5、7 和 11（绿色）的消息，沿着下面可视化的边。



该图包含原始图中的所有节点，但仅包含消息传递到给定输出节点所需的边。我们称红色节点 8 为第二个 GNN 层的边界。

几个函数可用于生成边界。 例如，dgl.in_subgraph()通过包含原始图中的所有节点，但仅包括给定节点的所有传入边来生成子图的函数。 您可以将其用作沿所有传入边传递的消息的边界。

```
frontier = dgl.in_subgraph(g, [8])
print(frontier.all_edges())
```

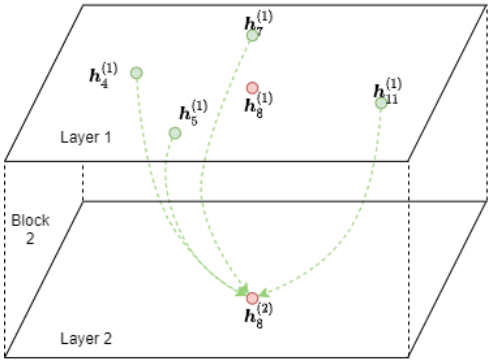
有关具体列表，请参阅子图提取操作和 dgl.sampling。

从技术上讲，任何具有与原始图相同的节点集的图都可以用作边界。 这是

实现自定义邻域采样器的基础。

6.6.3 多层 minibatch 消息传递的双向结构

但是，要从 $h^{(1)}$计算$h_8^{(2)}$，我们不能简单地直接在边界上执行消息传递，因为它仍然包含原始图中的所有节点。 也就是说，我们只需要节点 4、5、7、8 和 11（绿色和红色节点）作为输入，以及节点 8（红色节点）作为输出。 由于用于输入和输出的节点数不同，因此我们需要在一个小的二部图上执行消息传递。我们称这种仅包含必要的输入节点和输出节点的二元结构图为一个块。 下图显示了节点 8 的第二个 GNN 层的块。



请注意，输出节点也出现在输入节点中。 原因是消息传递后的特征组合需要前一层的输出节点表示（即 $\varphi^{(2)}$）。

DGL 提供 dgl.to_block()以将任何边界转换为块，其中第一个参数指定边界，第二个参数指定输出节点。 例如，可以使用以下代码将上述边界转换为带有输出节点 8 的块。

```
output_nodes = torch.LongTensor([8])
block = dgl.to_block(frontier, output_nodes)
```

要查找给定节点类型的输入节点和输出节点的数量，
可以使用 dgl.DGLHeteroGraph.number_of_src_nodes()
和 dgl.DGLHeteroGraph.number_of_dst_nodes()方法。

```
num_input_nodes, num_output_nodes = block.number_of_src_nodes(),
block.number_of_dst_nodes()
print(num_input_nodes, num_output_nodes)
```

可以通过成员 dgl.DGLHeteroGraph.srcdata 和 dgl.DGLHeteroGraph.srcnodes 访问该块的输入节点特征，并且可以通过成员 dgl.DGLHeteroGraph.dstdata 和 dgl.DGLHeteroGraph.dstnodes 访问其输出节点特征。

srcdata / dstdata 和 srcnodes / dstnodes 的语法与普通图中的 dgl.DGLHeteroGraph.ndata 和 dgl.DGLHeteroGraph.nodes 相同。

```
block.srcdata['h'] = torch.randn(num_input_nodes, 5)
block.dstdata['h'] = torch.randn(num_output_nodes, 5)
```

如果从边界转换图块，再从图转换图块，则可以通过以下方式直接读取块的输入和输出节点的特征。

```
print(block.srcdata['x'])
print(block.dstdata['y'])
```

可以找到块中输入节点和输出节点的原始节点 id 作为特征 dgl.NID。可以通

过特征 dgl.EID 找到从块的边 id 到输入边界的边 id 的映射。

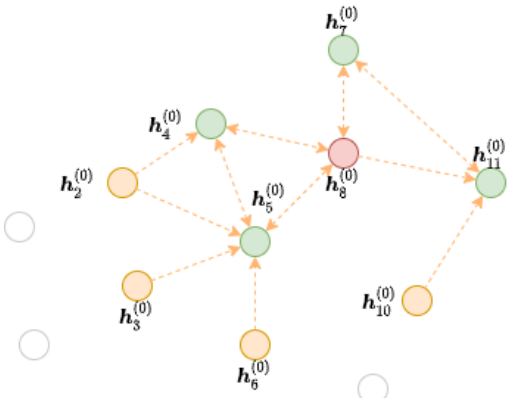
输出节点：

DGL 确保块的输出节点将始终出现在输入节点中。 输出节点将始终首先在输入节点中建立索引。

```
input_nodes = block.srcdata[dgl.NID]
output_nodes = block.dstdata[dgl.NID]
assert torch.equal(input_nodes[:len(output_nodes)], output_nodes)
```

因此，输出节点必须覆盖边界上某个边的目的地的所有节点。

例如，考虑以下边界：



其中红色和绿色节点（即节点 4、5、7、8 和 11）都是作为边目标的所有节点。 然后，以下代码将引发错误，因为输出节点未覆盖所有这些节点。

```
dgl.to_block(frontier2, torch.LongTensor([4, 5]))   # ERROR
```

但是，输出节点可以具有比以上更多的节点。 在这种情况下，我们将有隔离的节点，这些节点没有任何边与其连接。 隔离的节点将同时包含在输入节点和输出节点中。

```
# Node 3 is an isolated node that do not have any edge pointing to it.
block3 = dgl.to_block(frontier2, torch.LongTensor([4, 5, 7, 8, 11, 3]))
print(block3.srcdata[dgl.NID])
print(block3.dstdata[dgl.NID])
```

6.6.4 块在异构图上工作

块也可用于异构图。 假设我们有以下边界：

```
hetero_frontier = dgl.heterograph({
    ('user', 'follow', 'user'): ([1, 3, 7], [3, 6, 8]),
    ('user', 'play', 'game'): ([5, 5, 4], [6, 6, 2]),
    ('game', 'played-by', 'user'): ([2], [6])
}, num_nodes_dict={'user': 10, 'game': 10})
```

也可以使用输出节点 User＃3， ＃6 和＃8 以及 Game＃2 和＃6 创建一个块。

```
hetero_block = dgl.to_block(hetero_frontier, {'user': [3, 6, 8], 'block': [2, 6]})
```

还可以按类型获取输入节点和输出节点：

```
# input users and games
print(hetero_block.srcnodes['user'].data[dgl.NID],
hetero_block.srcnodes['game'].data[dgl.NID])
# output users and games
```

```
    print(hetero_block.dstnodes['user'].data[dgl.NID],
hetero_block.dstnodes['game'].data[dgl.NID])
```

6.6.5 实现自定义邻域采样器

回想一下，以下代码对节点分类执行邻域采样。

```
sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
```

为了实现自己的邻域采样策略，基本上可以将采样器对象替换为自己的采样器对象。 为此，我们首先来看一下 MultiLayerFullNeighborSampler 的父类 BlockSampler。

BlockSampler 负责使用方法 sample_block()从最后一层开始生成块列表。sample_blocks 的默认实现是向后迭代，生成边界并将其转换为块。

因此，对于邻域采样，您仅需要实现 sample_frontier()方法。 给定采样器正在为其生成边界的层以及原始图和要计算表示形式的节点，此方法负责为其生成边界。

同时，您还需要将必须拥有的 GNN 层数传递给父类。

例如，MultiLayerFullNeighborSampler 的实现可以如下进行。

```
class MultiLayerFullNeighborSampler(dgl.dataloading.BlockSampler):
    def __init__(self, n_layers):
        super().__init__(n_layers)

    def sample_frontier(self, block_id, g, seed_nodes):
        frontier = dgl.in_subgraph(g, seed_nodes)
        return frontier
```

dgl.dataloading.neighbor.MultiLayerNeighborSampler 是一个更复杂的邻域采样器类，它允许您对少量邻域进行采样以收集每个节点的消息，如下所示。

```
class MultiLayerNeighborSampler(dgl.dataloading.BlockSampler):
    def __init__(self, fanouts):
        super().__init__(len(fanouts))

        self.fanouts = fanouts

    def sample_frontier(self, block_id, g, seed_nodes):
        fanout = self.fanouts[block_id]
        if fanout is None:
            frontier = dgl.in_subgraph(g, seed_nodes)
        else:
            frontier = dgl.sampling.sample_neighbors(g, seed_nodes, fanout)
        return frontier
```

虽然上述函数可以生成边界，但任何与原始图具有相同节点的图都可以作为边界。

例如，如果要随机地将入站边按一定概率放到种子节点上，可以简单地定义采样器如下:

```
class MultiLayerDropoutSampler(dgl.dataloading.BlockSampler):
    def __init__(self, p, n_layers):
```

```
        super().__init__()

        self.n_layers = n_layers
        self.p = p

    def sample_frontier(self, block_id, g, seed_nodes, *args, **kwargs):
        # Get all inbound edges to `seed_nodes`
        src, dst = dgl.in_subgraph(g, seed_nodes).all_edges()
        # Randomly select edges with a probability of p
        mask = torch.zeros_like(src).bernoulli_(self.p)
        src = src[mask]
        dst = dst[mask]
        # Return a new graph with the same nodes as the original graph as a
        # frontier
        frontier = dgl.graph((src, dst), num_nodes=g.number_of_nodes())
        return frontier

    def __len__(self):
        return self.n_layers
```

在实现采样器之后，您可以创建一个数据加载器来接收采样器，它将像往常一样在遍历种子节点时继续生成块列表。

```
sampler = MultiLayerDropoutSampler(0.5, 2)
dataloader = dgl.dataloading.NodeDataLoader(
    g, train_nids, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)

model = StochasticTwoLayerRGCN(in_features, hidden_features, out_features)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())

for input_nodes, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    input_features = blocks[0].srcdata     # returns a dict
    output_labels = blocks[-1].dstdata     # returns a dict
    output_predictions = model(blocks, input_features)
    loss = compute_loss(output_labels, output_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

6.6.6 为异构图生成边界

为异构图生成边界与为同类图生成边界没有什么不同。 只要使返回的图具有与原始图相同的节点，就可以正常工作。 例如，我们可以重写上面的 MultiLayerDropoutSampler 以遍历所有边类型，以便它也可以在异构图上使用。

```
class MultiLayerDropoutSampler(dgl.dataloading.BlockSampler):
    def __init__(self, p, n_layers):
        super().__init__()

        self.n_layers = n_layers
        self.p = p

    def sample_frontier(self, block_id, g, seed_nodes, *args, **kwargs):
        # Get all inbound edges to `seed_nodes`
        sg = dgl.in_subgraph(g, seed_nodes)

        new_edges_masks = {}
        # Iterate over all edge types
        for etype in sg.canonical_etypes:
            edge_mask = torch.zeros(sg.number_of_edges(etype))
            edge_mask.bernoulli_(self.p)
            new_edges_masks[etype] = edge_mask.bool()

        # Return a new graph with the same nodes as the original graph as a
        # frontier
        frontier = dgl.edge_subgraph(new_edge_masks, preserve_nodes=True)
        return frontier

    def __len__(self):
        return self.n_layers
```

6.7 实现自定义 GNN 模块进行 minibatch 训练

如果您熟悉如何编写自定义 GNN 模块以更新同构图或异构图的整个图（请参见第 3 章：构建 GNN 模块），则用于块计算的代码是相似的，除了将节点分为输入节点和输出节点。

例如，考虑以下自定义图卷积模块代码。 请注意，它不一定是最有效的实现之一，它们仅用作自定义 GNN 模块显示的示例。

```
class CustomGraphConv(nn.Module):
    def __init__(self, in_feats, out_feats):
        super().__init__()
        self.W = nn.Linear(in_feats * 2, out_feats)

    def forward(self, g, h):
        with g.local_scope():
```

```
            g.ndata['h'] = h
            g.update_all(fn.copy_u('h', 'm'), fn.mean('m', 'h_neigh'))
            return self.W(torch.cat([g.ndata['h'], g.ndata['h_neigh']], 1))
```

如果您有一个用于整个图的自定义消息传递 NN 模块，并且希望使其适用于块，则只需重写正向函数，如下所示。 请注意，对全图实现中的相应语句进行了注释； 您可以将原始语句与新语句进行比较。

```
class CustomGraphConv(nn.Module):
    def __init__(self, in_feats, out_feats):
        super().__init__()
        self.W = nn.Linear(in_feats * 2, out_feats)

    # h is now a pair of feature tensors for input and output nodes, instead of
    # a single feature tensor.
    # def forward(self, g, h):
    def forward(self, block, h):
        # with g.local_scope():
        with block.local_scope():
            # g.ndata['h'] = h
            h_src = h
            h_dst = h[:block.number_of_dst_nodes()]
            block.srcdata['h'] = h_src
            block.dstdata['h'] = h_dst

            # g.update_all(fn.copy_u('h', 'm'), fn.mean('m', 'h_neigh'))
            block.update_all(fn.copy_u('h', 'm'), fn.mean('m', 'h_neigh'))

            # return self.W(torch.cat([g.ndata['h'], g.ndata['h_neigh']], 1))
            return self.W(torch.cat(
                [block.dstdata['h'], block.dstdata['h_neigh']], 1))
```

通常，您需要执行以下操作以使 NN 模块适用于块。

- 通过对前几行进行切片，从输入特征中获取输出节点的特征。 行数可以通过 block.number_of_dst_nodes 获得。
- 如果原始图只有一种节点类型，则将 g.ndata 替换为 block.srcdata 用于输入节点上的特征，或者替换为 block.dstdata 用于输出节点上的特征。
- 如果原始图具有多种节点类型，则将 g.nodes 替换为 block.srcnodes 用于输入节点上的特征，或者替换为 block.dstnodes 用于输出节点上的特征。
- 将 g.number_of_nodes 分别替换为 block.number_of_src_nodes 或 block.number_of_dst_nodes 作为输入节点或输出节点的数量。

6.5.1 异构图

对于异构图，编写自定义 GNN 模块的方式是相似的。 例如，考虑以下适用于全图的模块。

```
class CustomHeteroGraphConv(nn.Module):
    def __init__(self, g, in_feats, out_feats):
```

```
            super().__init__()
            self.Ws = nn.ModuleDict()
            for etype in g.canonical_etypes:
                utype, _, vtype = etype
                self.Ws[etype] = nn.Linear(in_feats[utype], out_feats[vtype])
            for ntype in g.ntypes:
                self.Vs[ntype] = nn.Linear(in_feats[ntype], out_feats[ntype])

        def forward(self, g, h):
            with g.local_scope():
                for ntype in g.ntypes:
                    g.nodes[ntype].data['h_dst'] = self.Vs[ntype](h[ntype])
                    g.nodes[ntype].data['h_src'] = h[ntype]
                for etype in g.canonical_etypes:
                    utype, _, vtype = etype
                    g.update_all(
                        fn.copy_u('h_src', 'm'), fn.mean('m', 'h_neigh'),
                        etype=etype)
                    g.nodes[vtype].data['h_dst'] = g.nodes[vtype].data['h_dst'] + \
                        self.Ws[etype](g.nodes[vtype].data['h_neigh'])
                return {ntype: g.nodes[ntype].data['h_dst'] for ntype in g.ntypes}
```

对于 CustomHeteroGraphConv，原理是将 g.nodes 替换为 g.srcnodes 或 g.dstnodes，具体取决于特征是用于输入还是输出。

```
class CustomHeteroGraphConv(nn.Module):
    def __init__(self, g, in_feats, out_feats):
        super().__init__()
        self.Ws = nn.ModuleDict()
        for etype in g.canonical_etypes:
            utype, _, vtype = etype
            self.Ws[etype] = nn.Linear(in_feats[utype], out_feats[vtype])
        for ntype in g.ntypes:
            self.Vs[ntype] = nn.Linear(in_feats[ntype], out_feats[ntype])

    def forward(self, g, h):
        with g.local_scope():
            for ntype in g.ntypes:
                h_src, h_dst = h[ntype]
                g.dstnodes[ntype].data['h_dst'] = self.Vs[ntype](h[ntype])
                g.srcnodes[ntype].data['h_src'] = h[ntype]
            for etype in g.canonical_etypes:
                utype, _, vtype = etype
                g.update_all(
                    fn.copy_u('h_src', 'm'), fn.mean('m', 'h_neigh'),
                    etype=etype)
```

```
            g.dstnodes[vtype].data['h_dst'] = \
                g.dstnodes[vtype].data['h_dst'] + \
                self.Ws[etype](g.dstnodes[vtype].data['h_neigh'])
        return {ntype: g.dstnodes[ntype].data['h_dst']
                for ntype in g.ntypes}
```

6.5.2 编写可用于同构图，二部图和块的模块

DGL 中的所有消息传递模块都可用于同构图，单向二部图（具有两种节点类型和一种边类型）以及具有一种边类型的块中。 本质上，内置 DGL 神经网络模块的输入图和特征必须满足以下任一情况。

- 如果输入特征是一对张量，则输入图必须是单向二分的。
- 如果输入特征是单个张量，而输入图是一个块，则 DGL 将自动在输出节点上将特征设置为输入节点特征的前几行。
- 如果输入特征必须是单个张量，并且输入图不是块，则输入图必须是齐次的。

例如，以下内容是对 dgl.nn.PyTorch.SAGEConv 的 PyTorch 实现的简化。 (也可以在 MXNet 和 Tensorflow 中使用)(消除规范化，只处理平均聚合等)。

```
import dgl.function as fn
class SAGEConv(nn.Module):
    def __init__(self, in_feats, out_feats):
        super().__init__()
        self.W = nn.Linear(in_feats * 2, out_feats)

    def forward(self, g, h):
        if isinstance(h, tuple):
            h_src, h_dst = h
        elif g.is_block:
            h_src = h
            h_dst = h[:g.number_of_dst_nodes()]
        else:
            h_src = h_dst = h

        g.srcdata['h'] = h_src
        g.dstdata['h'] = h_dst
        g.update_all(fn.copy_u('h', 'm'), fn.sum('m', 'h_neigh'))
        return F.relu(
            self.W(torch.cat([g.dstdata['h'], g.dstdata['h_neigh']], 1)))
```

第 3 章：构建 GNN 模块还在 dgl.nn.pytorch.SAGEConv 上提供了一个演示，该教程适用于单向二部图，同构图和块。

6.6 对大型图进行精确离线推断

子图采样和邻域采样都可以减少使用 GPU 训练 GNN 的内存和时间消耗。执行推断时，通常最好在所有邻居上进行真实汇总，而不要摆脱采样带来的随机性。但是，由于内存有限，全图正向传播通常在 GPU 上是不可行的，而在 CPU 上由

于计算速度较慢则很慢。本节介绍了通过 minibatch 和邻域采样在有限的 GPU 内存下进行全图正向传播的方法。

推理算法不同于训练算法，从第一层开始，逐层计算所有节点的表示。具体来说，对于一个特定的层，我们需要以 minibatch 的方式计算来自这个 GNN 层的所有节点的输出表示。其结果是，推理算法将在各个层上进行外部循环迭代，而在节点的小批上进行内部循环迭代。相比之下，训练算法有一个外循环在小批节点上进行迭代，以及一个内循环在各层上进行邻域采样和消息传递。

以下动画显示了计算的样子（请注意，对于每一层，只绘制了前三个 minibatch）。（由于是动图就不贴了）

6.6.1 实现离线推理

考虑我们在 6.5.1 节中提到的两层 GCN。 实现离线推理的方法仍然涉及使用 MultiLayerFullNeighborSampler，但一次仅采样一层。 请注意，离线推理是 GNN 模块的一种方法，因为在一层上的计算还取决于消息的聚合和组合方式。

```
class StochasticTwoLayerGCN(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.hidden_features = hidden_features
        self.out_features = out_features
        self.conv1 = dgl.nn.GraphConv(in_features, hidden_features)
        self.conv2 = dgl.nn.GraphConv(hidden_features, out_features)
        self.n_layers = 2

    def forward(self, blocks, x):
        x_dst = x[:blocks[0].number_of_dst_nodes()]
        x = F.relu(self.conv1(blocks[0], (x, x_dst)))
        x_dst = x[:blocks[1].number_of_dst_nodes()]
        x = F.relu(self.conv2(blocks[1], (x, x_dst)))
        return x

    def inference(self, g, x, batch_size, device):
        """
        Offline inference with this module
        """
        # Compute representations layer by layer
        for l, layer in enumerate([self.conv1, self.conv2]):
            y = torch.zeros(g.number_of_nodes(),
                            self.hidden_features
                            if l != self.n_layers - 1
                            else self.out_features)
            sampler = dgl.dataloading.MultiLayerFullNeighborSampler(1)
            dataloader = dgl.dataloading.NodeDataLoader(
                g, torch.arange(g.number_of_nodes()), sampler,
                batch_size=batch_size,
```

```
            shuffle=True,
            drop_last=False)

        # Within a layer, iterate over nodes in batches
        for input_nodes, output_nodes, blocks in dataloader:
            block = blocks[0]

            # Copy the features of necessary input nodes to GPU
            h = x[input_nodes].to(device)
            # Compute output.  Note that this computation is the same
            # but only for a single layer.
            h_dst = h[:block.number_of_dst_nodes()]
            h = F.relu(layer(block, (h, h_dst)))
            # Copy to output back to CPU.
            y[output_nodes] = h.cpu()

    return y
```

注意，为了在模型选择的验证集上计算评估指标，我们通常不需要计算精确的离线推断。原因是我们需要计算每一层上每一个节点的表示，这通常是非常昂贵的，特别是在有大量未标记数据的半监督系统中。邻域抽样可以很好地用于模型选择和验证。

我们可以看到 GraphSAGE 和 RGCN 作为离线推理的例子。

https://github.com/dmlc/dgl/blob/master/examples/pytorch/graphsage/train_sampling.py

https://github.com/dmlc/dgl/blob/master/examples/pytorch/rgcn-hetero/entity_classify_mb.py

7 分布式训练

DGL 采用了一种完全分布式的方法，通过一组计算资源来分布数据和计算。在本节的上下文中，我们将假设一个集群设置(即一组机器)。DGL 将一个图划分为子图，集群中的每台机器负责一个子图(分区)。DGL 在集群中的所有机器上运行相同的训练脚本，以并行化计算，并在相同的机器上运行服务器，将分区的数据提供给训练器。

对于训练脚本，DGL 提供了类似于用于 minibatch 训练的分布式 api。这使得分布式训练只需要在一台机器上进行小批量的代码修改。下面展示了一个以分布式方式训练 GraphSage 的示例。修改的代码仅位于第 4-7 行:

1)初始化 DGL 的分布式模块；

2)创建分布式图 objec t；

3)分割训练集并计算本地流程的节点。剩下的代码，包括创建采样器、模型定义和训练循环，与小批量训练相同。

```
import dgl
import torch as th
```

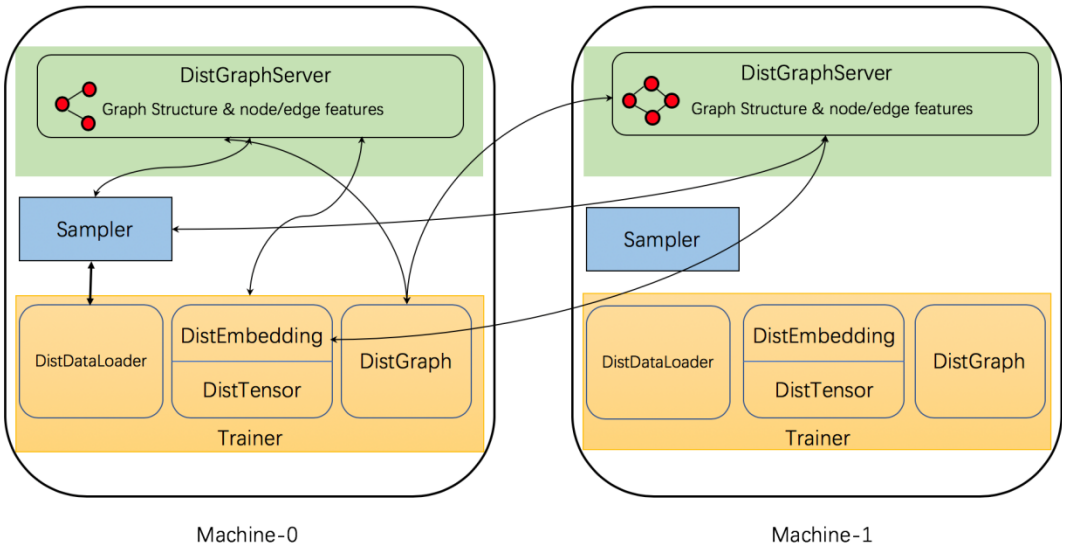
```
    dgl.distributed.initialize(ip_config, num_workers=num_workers)
    th.distributed.init_process_group(backend='gloo')
    g = dgl.distributed.DistGraph('ip_config.txt', 'graph_name')
    train_nid = dgl.distributed.node_split(g.ndata['train_mask'])


    # Create sampler
    sampler = dgl.dataloading.MultiLayerNeighborSampler([10, 25])
    train_dataloader = dgl.dataloading.NodeDataLoader(g, train_nid,
                                                      sampler, batch_size=1024,
                                                      shuffle=True,drop_last=False)

    # Define model and optimizer
    model = SAGE(in_feats, num_hidden, n_classes, num_layers, F.relu, dropout)
    model = th.nn.parallel.DistributedDataParallel(model)
    loss_fcn = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=args.lr)

    # training loop
    for epoch in range(args.num_epochs):
        for step, blocks in enumerate(dataloader):
            batch_inputs, batch_labels = load_subtensor(g,
blocks[0].srcdata[dgl.NID],
                                                        blocks[-1].dstdata[dgl.NID])
            batch_pred = model(blocks, batch_inputs)
            loss = loss_fcn(batch_pred, batch_labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

在一个机器集群中运行训练脚本时，DGL 提供了将数据复制到集群的机器并在所有机器上启动训练作业的工具。

注意:当前的分布式训练 API 只支持 Pytorch 后端。

注意:当前的实现只支持一种节点类型和一种边类型的图。

DGL 实现了几个分布式组件来支持分布式训练。下图显示了组件及其交互。



具体来说，DGL 的分布式训练有三种类型的交互过程：服务器、采样器和训练器。

服务器进程运行在存储图分区(包括图结构和节点/边特征)的每台机器上。这些服务器一起工作，为训练人员提供图数据。请注意，一台机器可以同时运行多个服务器进程来并行计算和网络通信。

采样器进程与服务器、样本节点和边交互，生成用于训练的小批量。

训练器包含多个类来与服务器交互。It has DistGraph to get access to partitioned graph data and has DistEmbedding and DistTensor to access the node/edge features/embeddings. It has DistDataLoader to interact with samplers to get mini-batches.

7.3 分布式训练的预处理

DGL 需要对分布训练的图数据进行预处理，包括两个步骤:1)将图划分为子图，2)分配新 id 给节点/边。DGL 提供了执行这两个步骤的分区 API。API 既支持随机分区，也支持基于 metis 的分区。Metis 分区的好处是，它可以生成具有最小切边的分区，减少用于分布式训练和推理的网络通信。DGL 使用最新版本的 Metis，并针对具有幂律分布的实际图优化了选项。分区之后，API 以一种易于在训练期间加载的格式构造分区结果。

注意：图分区 API 当前在一台计算机上运行。 因此，如果图很大，则用户将需要一台大型计算机来对图进行分区。 将来，DGL 将支持分布式图分区。

默认情况下，分区 API 将新 ID 分配给输入图中的节点和边，以在分布式训练/推理期间帮助定位节点/边。 分配 ID 后，分区 API 会相应地对所有节点数据和边数据进行混洗。 在训练期间，用户仅使用新的节点/边 ID。 但是，仍然可以通过 g.ndata ['orig_id']和 g.edata ['orig_id']访问原始 ID，其中 g 是 DistGraph 对象（请参见 DistGraph 部分）。

分区结果存储在输出目录中的多个文件中。它始终包含一个名为 xxx.json 的 JSON 文件，其中 xxx 是提供给分区 API 的图名称。JSON 文件包含所有分区配置。如果分区 API 没有为节点和边分配新的 ID，它将生成两个附加的 Numpy 文件：node_map.npy 和 edge_map.npy，它们存储节点/边 ID 与分区 ID 之间的映射。 对于具有数十亿个节点和边的图，两个文件中的 Numpy 数组很大，因为它们在图

中的每个节点和边都有一个条目。 在每个分区的文件夹内，有三个文件以 DGL 格式存储分区数据。graph.dgl 存储分区的图结构以及节点和边上的一些元数据。node_feats.dgl 和 edge_feats.dgl 存储属于该分区的节点和边的所有特征。

```
data_root_dir/
    |-- xxx.json                # partition configuration file in JSON
    |-- node_map.npy       # partition id of each node stored in a numpy array (optional)
    |-- edge_map.npy       # partition id of each edge stored in a numpy array (optional)
    |-- part0/                  # data for partition 0
        |-- node_feats.dgl   # node features stored in binary format
        |-- edge_feats.dgl   # edge features stored in binary format
        |-- graph.dgl           # graph structure of this partition stored in binary
format
    |-- part1/                    # data for partition 1
        |-- node_feats.dgl
        |-- edge_feats.dgl
        |-- graph.dgl
```

7.3.1 负载均衡

默认情况下，在对图进行分区时，Metis 只平衡每个分区中的节点数。这可能导致次优配置，具体取决于手头的任务。例如，在半监督节点分类的情况下，一个训练器在局部分区的标记节点子集上执行计算。仅平衡图中的节点(标记和未标记)的分区可能会导致计算负载不平衡。为了在每个分区中获得平衡的工作负载，分区 API 允许根据每个节点类型中的节点数量在分区之间进行平衡，方法是在 dgl. distribu. partition_graph()中指定 balance_ntypes。用户可以利用这一点，并考虑训练集、验证集和测试集中的节点属于不同的节点类型。

下面的例子认为训练集内的节点和训练集外的节点是两种类型的节点:

```
dgl.distributed.partition_graph(g, 'graph_name', 4, '/tmp/test',
balance_ntypes=g.ndata['train_mask'])
```

除了平衡节点类型之外， dgl.distribu.partition_graph()还允许通过指定 balance_edges 在不同节点类型的 In -degree 节点之间进行平衡。这平衡了关联到不同类型节点的边的数量。

注意:传递给 dgl. distribu. partition_graph()的图名称是一个重要的参数。图名称将由 dgl.distributed 使用。扩展图标识一个分布式图。一个合法的图名称应该只包含字母和下划线。

7.4 分布式 APIs

本节介绍了训练脚本中使用的分布式 API。 DGL 提供了三种分布式数据结构和各种 API，用于初始化，分布式采样和工作负载拆分。 对于分布式训练/推理，DGL 提供了三种分布式数据结构：用于分布式图的 DistGraph，用于分布式张量的 DistTensor 和用于分布式可学习嵌入的 DistEmbedding。

7.4.1 DGL 分布式模块的初始化

initialize()初始化分布式模块。当训练脚本在 trainer 模式下运行时，这个 API

构建与 DGL 服务器的连接并创建采样进程;当脚本在服务器模式下运行时，该 API 将运行服务器代码，并且永远不会返回。这个 API 必须在 DGL 的任何分布式 API 之前调用。在使用 Pytorch 时，必须在 torch. distribu. init_process_group 之前调用 initialize()。通常，初始化 api 应该按照以下顺序调用:

```
dgl.distributed.initialize('ip_config.txt', num_workers=4)
th.distributed.init_process_group(backend='gloo')
```

注意:如果训练脚本包含必须在服务器上调用的用户定义函数(udf)(更多细节请参阅 DistTensor 和 DistEmbedding 一节)，那么必须在 initialize()之前声明这些 udf。

7.4.2 分布式图

DistGraph 是一个 Python 类，用于访问机器集群中的图结构和节点/边特征。每台机器只负责一个分区。它加载分区数据(图结构、分区中的节点数据和边数据)，并使集群中的所有训练器都可以访问它。DistGraph 提供了用于数据访问的 DGLGraph api 的一个小子集。

注意:DistGraph 目前只支持一种节点类型和一种边类型的图。

（1） 分布式模式与独立模式

DistGraph 可以在两种模式下运行:分布式模式和独立模式。当用户在 Python 命令行或 Jupyter 笔记本中执行训练脚本时，它将以独立模式运行。也就是说，它在一个进程中运行所有的计算，并且不与任何其他进程通信。因此，独立模式要求输入图只有一个分区。该模式主要用于开发和测试(如在 Jupyter 笔记本中开发和运行代码)。当用户使用启动脚本执行训练脚本时(参见启动脚本一节)，DistGraph 将以分布式模式运行。启动工具启动场景背后的服务器(节点/边特征访问和图采样)，并自动加载每台机器中的分区数据。DistGraph 连接到机器集群中的服务器，并通过网络访问它们。

（2） DistGraph 创建

在分布式模式下，DistGraph 的创建需要在图分区期间使用的图名称。 图名称标识集群中加载的图。

```
import dgl
g = dgl.distributed.DistGraph('graph_name')
```

在独立模式下运行时，它将图数据加载到本地计算机中。 因此，用户需要提供分区配置文件，其中包含有关输入图的所有信息。

```
import dgl
g = dgl.distributed.DistGraph('graph_name',
part_config='data/graph_name.json')
```

注意：在当前实现中，DGL 仅允许创建单个 DistGraph 对象。 销毁 DistGraph 并创建一个新 DistGraph 的行为是不确定的。

（3） 访问图结构

DistGraph 提供了很少的 API 来访问图结构。 当前，大多数 API 提供图信息，例如节点和边的数量。 DistGraph 的主要用例是运行采样 API 以支持小批量训练（请参阅分布式图采样部分）。

```
print(g.number_of_nodes())
```

（4） 接入节点和边数据

与 DGLGraph 一样，DistGraph 提供 ndata 和 edata 来访问节点和边中的数据。

区别在于 DistGraph 中的 ndata / edata 返回 DistTensor，而不是基础框架的张量。用户还可以将新的 DistTensor 分配给 DistGraph 作为节点数据或边数据。

```
g.ndata['train_mask']
<dgl.distributed.dist_graph.DistTensor at 0x7fec820937b8>
g.ndata['train_mask'][0]
tensor([1], dtype=torch.uint8)
```

7.4.3 分布式张量

如前所述，DGL 对节点/边特征进行切分并将它们存储在一个机器集群中。DGL 为分布式张量提供了一个类似张量的接口，用于访问集群中的分区节点/边特征。在分布式设置中，DGL 只支持密集节点/边特征。

dist 张量管理分区并存储在多台机器中的密集张量。现在，一个分布式张量必须与图的节点或边相关。换句话说，dist 张量的行数必须与图中的节点数或边数相同。下面的代码创建了一个分布式张量。除了张量的形状和 dtype 外，用户还可以提供唯一的张量名。如果用户希望引用一个持久的分布式张量(即使 dist 张量对象消失了，该张量也存在于集群中)，则此名称非常有用。

```
tensor = dgl.distributed.DistTensor((g.number_of_nodes(), 10), th.float32,
name=’test’)
```

注意：DistTensor 创建是一个同步操作。 所有训练器都必须调用创建，并且只有当所有训练器都调用它时，创建才能成功。

用户可以将 DistTensor 作为节点数据或边数据之一添加到 DistGraph 对象。

```
g.ndata['feat'] = tensor
```

注意：节点数据名称和张量名称不必相同。 前者从 DistGraph 标识节点数据（在训练过程中），而后者则标识 DGL 服务器中的分布式张量。

DistTensor 提供了一小组功能。 它具有与常规张量相同的 API，用于访问其元数据，例如 shape 和 dtype。 DistTensor 支持索引读取和写入，但不支持计算运算符，例如总和和均值。

```
data = g.ndata['feat'][[1, 2, 3]]
print(data)
g.ndata['feat'][[3, 4, 5]] = data
```

注意：当前，当一台机器运行多个服务器时，DGL 不提供对来自多个训练器的并发写入的保护。 这可能会导致数据损坏。 避免同时写入同一行数据的一种方法是在计算机上运行一个服务器进程。

7.4.4 分布式嵌入

DGL 提供 DistEmbedding 以支持需要节点嵌入的转换模型。 创建分布式嵌入与创建分布式张量非常相似。

```
def initializer(shape, dtype):
    arr = th.zeros(shape, dtype=dtype)
    arr.uniform_(-1, 1)
    return arr
emb = dgl.distributed.DistEmbedding(g.number_of_nodes(), 10,
init_func=initializer)
```

在内部，分布式嵌入建立在分布式张量之上，因此，其行为与分布式张量非

常相似。例如，创建嵌入时，会将它们分片并存储在群集中的所有计算机上。可以通过名称唯一标识。

注意：初始化函数是在服务器进程中调用 因此，必须在初始化之前声明它。

因为嵌入是模型的一部分，所以用户必须将它们附加到优化器上，以便进行小批处理训练。目前，DGL 提供了一个稀疏的 Adagrad 优化器 SparseAdagrad (DGL 稍后会为稀疏嵌入添加更多的优化器)。用户需要从模型中收集所有分布式嵌入，并将它们传递给稀疏优化器。如果一个模型同时具有节点嵌入和规则密集模型参数，而用户想要对嵌入执行稀疏更新，他们需要创建两个优化器，一个用于节点嵌入，另一个用于密集模型参数，如下代码所示:

```
sparse_optimizer = dgl.distributed.SparseAdagrad([emb], lr=lr1)
optimizer = th.optim.Adam(model.parameters(), lr=lr2)
feats = emb(nids)
loss = model(feats)
loss.backward()
optimizer.step()
sparse_optimizer.step()
```

注意：DistEmbedding 不是 Pytorch nn 模块，因此我们无法从 Pytorch nn 模块的参数访问它。

7.2.5 分布式采样

DGL 提供了两层 api，用于对节点和边进行采样，以生成 minibatch(请参阅 minibatch 训练一节)。低级 api 要求用户编写代码来显式定义如何对一层节点进行采样(例如，使用 dgl. samples.sample_neighbors())。高级采样 api 为节点分类和链接预测任务实现了一些流行的采样算法(例如， NodeDataloader 和 EdgeDataloader)。

分布式采样模块遵循相同的设计，并提供了两层采样 api。对于底层采样 API，它提供了 sample_neighbors()用于在 DistGraph 上进行分布式邻近采样。此外，DGL 还提供了分布式采样的分布式数据 DataLoader (DistDataLoader)。分布式 Dataloader 与 Pytorch Dataloader 具有相同的接口，只是用户在创建 Dataloader 时不能指定工作进程的数量。工作进程是在 dgl.distribu.initialize()中创建的。

注意:当在 DistGraph 上运行 dgl.distribu.sample_neighbors()时，采样器不能在 Pytorch Dataloader 中与多个 worker 进程一起运行。主要原因是 Pytorch Dataloader 在每个 epoch 中都创建新的采样工作进程，这导致了对 DistGraph 对象的多次创建和销毁。

同样的高级采样 api (NodeDataloader 和 EdgeDataloader)也适用于 DGLGraph 和 DistGraph。在使用 NodeDataloader 和 EdgeDataloader 时，分布式采样代码与单进程采样完全相同。

在使用低级 API 时，采样代码类似于单进程采样。唯一的区别是用户需要使用 dgl.distribu.sample_neighbors()和 DistDataLoader。

```
def sample_blocks(seeds):
    seeds = th.LongTensor(np.asarray(seeds))
    blocks = []
    for fanout in [10, 25]:
        frontier = dgl.distributed.sample_neighbors(g, seeds, fanout,
```

```
replace=True)
            block = dgl.to_block(frontier, seeds)
            seeds = block.srcdata[dgl.NID]
            blocks.insert(0, block)
            return blocks
        dataloader = dgl.distributed.DistDataLoader(dataset=train_nid,
                                                    batch_size=batch_size,
                                                    collate_fn=sample_blocks,
                                                    shuffle=True)
        for batch in dataloader:
            ...
```

使用高级 API 时，分布式采样代码与单机采样相同：

```
sampler = dgl.sampling.MultiLayerNeighborSampler([10, 25])
dataloader = dgl.sampling.NodeDataLoader(g, train_nid, sampler,
                                         batch_size=batch_size, shuffle=True)
for batch in dataloader:
    ...
```

7.2.6 分割工作量

用户需要对训练集进行分割，以便每个训练器在自己的子集上工作。类似地，我们还需要以同样的方式分割验证和测试集。

分布式训练和评估,推荐的方法是使用布尔数组来表示节点的训练/验证/测试集。分类任务,这些布尔数组的长度是图中的节点的数量和每个元素表示一个节点的存在的训练/验证/测试集。类似的布尔数组应该用于链接预测任务。

DGL 提供 node_split()和 edge_split()来在运行时分割训练、验证和测试集，用于分布式训练。这两个函数将布尔数组作为输入，分割它们并返回一部分给本地训练器。默认情况下，它们确保所有部分具有相同数量的节点/边。这对于同步 SGD 很重要，因为它假定每个训练器都有相同数量的小批量。

下面的示例分割训练集并返回本地流程的节点子集。

```
train_nids = dgl.distributed.node_split(g.ndata['train_mask'])
```

7.5 用于启动分布式训练/推理的工具

DGL 提供了两个脚本来协助进行分布式培训：

- tools / copy_files.py 用于将图分区复制到图，
- tools / launch.py，用于在机器集群中启动分布式培训工作。

copy_files.py 将一台机器(图是分区的)中的分区数据和相关文件(例如，训练脚本)复制到一个机器集群(在这里进行分布式训练)。该脚本将一个分区复制到分布式培训作业需要该分区的机器上。该脚本包含四个参数:

- ——part_config 指定分区配置文件，该文件包含本地计算机中分区数据的信息。
- ——ip_config 指定集群的 IP 配置文件。
- ——workspace 指定训练机器中的目录,所有与分布式训练相关的数据都存储在其中。

- ——rel_data_path 指定存储分区数据的工作区目录下的相对路径。
- ——script_folder 指定存储用户训练脚本的工作区目录下的相对路径。

注意:copy_files.py 根据 IP 配置文件找到存储分区的正确机器。因此，copy_files.py 和 launchy.py 应该使用相同的 IP 配置文件。

DGL 提供了在集群中启动分布式培训作业的工具/launch.py。这个脚本做了以下假设:

- 分区数据和训练脚本已被复制到集群或集群中所有机器都可以访问的全局存储(例如 NFS)。
- 主机器(执行启动脚本的地方)具有对所有其他机器的无密码 ssh 访问。

注意:必须在集群中的一台机器上调用启动脚本。

下面是在集群中启动分布式培训作业的示例。

（1）配置文件 ip_config.txt 包含集群中机器的 IP 地址。ip_config.txt 的一个典型例子如下:

每一行是一台机器的 IP 地址。可选地，IP 地址后面可以跟着一个端口，该端口指定训练器之间的网络通信所使用的端口。当没有提供端口时，默认端口是 30050。

（2）在启动脚本中指定的工作区是机器中的工作目录，其中包含训练脚本、IP 配置文件、分区配置文件以及图分区。所有文件的路径都应该指定为工作区的相对路径。

（3）启动脚本在每台机器上创建指定数量的培训作业(——num_trainer)。此外，用户需要为每个培训器指定采样器进程的数量(——num_samplers)。采样进程的数量必须与 initialize()中指定的工作进程的数量匹配。