7.2 分布式计算的API

(English Version)

本节介绍了在训练脚本中使用的分布式计算API。DGL提供了三种分布式数据结构和多种API，用于初始化、分布式采样和数据分割。 对于分布式训练/推断，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` 。

DGL分布式模块的初始化

`initialize()` 可以用于初始化分布式模块。当训练脚本在训练器模式下运行时， 这个API会与DGL服务器建立连接并创建采样器进程。当脚本在服务器模式下运行时，这个API将运行服务器代码， 直到训练任务结束。必须在DGL的任何其他分布式API之前，调用此API。在使用PyTorch时，必须在 `torch.distributed.init_process_group` 之前调用 `initialize()` 。 通常，初始化API应按以下顺序调用：

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

Note: 如果训练脚本里包含需要在服务器(细节内容可以在下面的DistTensor和DistEmbedding章节里查看)上调用的用户自定义函数(UDF)， 这些UDF必须在 `initialize()` 之前被声明。

分布式图

`DistGraph` 是一个Python类，用于访问计算机集群中的图结构和节点/边特征。每台计算机负责一个且只负责一个分区。 它加载分区数据(包括分区中的图结构、节点数据和边数据)，并使集群中的所有训练器均可访问它们。 `DistGraph` 提供了一小部分 `DGLGraph` 的API以方便数据访问。

Note: `DistGraph` 当前仅支持一种节点类型和一种边类型的图。

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 Notebook中执行训练脚本时，它将以独立模式运行。也就是说，它在单个进程中运行所有计算， 并且不与任何其他进程通信。因此，独立模式要求输入图仅具有一个分区。此模式主要用于开发和测试 (例如，在Jupyter Notebook中开发和运行代码)。当用户使用启动脚本

执行训练脚本时(请参见启动脚本部分)， `DistGraph` 将以分布式模式运行。启动脚本在后台启动服务器(包括访问节点/边特征和图采样)， 并将分区数据自动加载到每台计算机中。 `DistGraph` 与集群中的服务器连接并通过网络访问它们。

创建DistGraph

在分布式模式下， `DistGraph` 的创建需要(定义)在图划分期间的图名称。 图名称标识了集群中所需加载的图。

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

在独立模式下运行时，DistGraph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')
```

Note: 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的行为没有被定义。

访问图结构

`DistGraph` 提供了几个API来访问图结构。当前，它们主要被用来提供图信息，例如节点和边的数量。 主要应用场景是运行采样API以支持小批量训练(请参阅下文里分布式图采样部分)。

```
print(g.num_nodes())
```

访问节点/边数据

与 `DGLGraph` 一样， `DistGraph` 也提供了 `ndata` 和 `edata` 来访问节点和边中的数据。它们的区别在于 `DistGraph` 中的 `ndata` / `edata` 返回的是 `DistTensor` ， 而不是底层框架里的张量。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)
```

分布式张量

如前所述，在分布式模式下，DGL会划分节点和边特征，并将它们存储在计算机集群中。DGL为分布式张量提供了类似于单机普通张量的接口，以访问群集中的分区节点和边特征。在分布式设置中，DGL仅支持密集节点和边特征，暂不支持稀疏节点和边特征。

`DistTensor` 管理在多个计算机中被划分和存储的密集张量。 目前，分布式张量必须与图的节点或边相关联。换句话说，DistTensor中的行数必须与图中的节点数或边数相同。 以下代码创建一个分布式张量。 除了张量的形状和数据类型之外，用户还可以提供唯一的张量名称。 如果用户要引用一个固定的分布式张量(即使 `DistTensor` 对象消失，该名称仍存在于群集中)，则(使用这样的)名称就很有用。

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

Note: `DistTensor` 的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，此创建过程才能成功。

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
```

Note: 节点数据名称和张量名称不必相同。前者在 `DistGraph` 中标识节点数据(在训练器进程中)， 而后者则标识DGL服务器中的分布式张量。

`DistTensor` 提供了一些功能。它具有与常规张量相同的API，用于访问其元数据， 例如形状和数据类型。 `DistTensor` 支持索引读取和写入， 但不支持一些计算运算符，例如求和以及求均值。

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

Note: 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。

分布式嵌入

DGL提供 `DistEmbedding` 以支持需要节点嵌入的直推(transductive)模型。 分布式嵌入的创建与分布式张量的创建非常相似。

```
def initializer(shape, dtype):
    arr = th.zeros(shape, dtype=dtype)
    arr.uniform_(-1, 1)
    return arr
emb = dgl.distributed.DistEmbedding(g.num_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，DGL会将它们分片并存储在集群中的所有计算机上。(分布式嵌入)可以通过名称唯一标识。

Note: 服务器进程负责调用初始化函数。因此，必须在初始化(`initialize`)之前声明分布式嵌入。

因为嵌入是模型的一部分，所以用户必须将其附加到优化器上以进行小批量训练。当前，DGL提供了一个稀疏的Adagrad优化器 `SparseAdagrad` (DGL以后将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。如果模型同时具有节点嵌入和规则的密集模型参数， 并且用户希望对嵌入执行稀疏更新，则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，如以下代码所示：

```
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()
```

Note: `DistEmbedding` 不是PyTorch的nn模块，因此用户无法从nn模块的参数访问它。

分布式采样

DGL提供了两个级别的API，用于对节点和边进行采样以生成小批次训练数据(请参阅小批次训练的章节)。 底层API要求用户编写代码以明确定义如何对节点层进行采样(例如，使用 `dgl.sampling.sample_neighbors()`)。 高层采样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，也提供两个级别的采样API。对于底层的采样API，它为 `DistGraph` 上的分布式邻居采样提供了 `sample_neighbors()` 。另外，DGL提供了用于分布式采样的分布式数据加载器(`DistDataLoader`)。除了用户在创建数据加载器时无法指定工作进程的数量， 分布式数据加载器具有与PyTorch DataLoader相同的接口。其中的工作进程(worker)在 `dgl.distributed.initialize()` 中创建。

Note: 在 `DistGraph` 上运行 `dgl.distributed.sample_neighbors()` 时， 采样器无法在具有多个工作进程的PyTorch DataLoader中运行。主要原因是PyTorch DataLoader在每个训练周期都会创建新的采样工作进程， 从而导致多次创建和删除 `DistGraph` 对象。

使用底层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.distributed.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:
        ...
```

`NodeDataLoader` 和 `EdgeDataLoader` 有分布式的版本 `DistNodeDataLoader` 和 `DistEdgeDataLoader` 。使用 时分布式采样代码与单进程采样几乎完全相同。

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

分割数据集

用户需要分割训练集，以便每个训练器都可以使用自己的训练集子集。同样，用户还需要以相同的方式分割验证和测试集。

对于分布式训练和评估，推荐的方法是使用布尔数组表示训练、验证和测试集。对于节点分类任务， 这些布尔数组的长度是图中节点的数量，并且它们的每个元素都表示训练/验证/测试集中是否存在对应节点。 链接预测任务也应使用类似的布尔数组。

DGL提供了 `node_split()` 和 `edge_split()` 函数来在运行时拆分训练、验证和测试集，以进行分布式训练。这两个函数将布尔数组作为输入，对其进行拆分，并向本地训练器返回一部分。 默认情况下，它们确保所有部分都具有相同数量的节点和边。这对于同步SGD非常重要， 因为同步SGD会假定每个训练器具有相同数量的小批次。

下面的示例演示了训练集拆分，并向本地进程返回节点的子集。

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