7.3 Programming APIs

(中文版)

This section covers the core python components commonly used in a training script. DGL provides three distributed data structures and various APIs for initialization, distributed sampling and workload split.

- `DistGraph` for accessing structure and feature of a distributedly stored graph.
- `DistTensor` for accessing node/edge feature tensor that is partitioned across machines.
- `DistEmbedding` for accessing learnable node/edge embedding tensor that is partitioned across machines.

Initialization of the DGL distributed module

`dgl.distributed.initialize()` initializes the distributed module. If invoked by a trainer, this API creates sampler processes and builds connections with graph servers; if invoked by graph server, this API starts a service loop to listen to trainer/sampler requests. The API *must* be called before `torch.distributed.init_process_group()` and any other `dgl.distributed` APIs as shown in the order below:

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

Note

If the training script contains user-defined functions (UDFs) that have to be invoked on the servers (see the section of DistTensor and DistEmbedding for more details), these UDFs have to be declared before `initialize()`.

Distributed graph

`DistGraph` is a Python class to access the graph structure and node/edge features in a cluster of machines. Each machine is responsible for one and only one partition. It loads the partition data (the graph structure and the node data and edge data in the partition) and makes it accessible to all trainers in the cluster. `DistGraph` provides a small subset of `DGLGraph` APIs for data access.

Distributed mode vs. standalone mode

`DistGraph` can run in two modes: *distributed mode* and *standalone mode*. When a user executes a training script in a Python command line or Jupyter Notebook, it runs in a standalone mode. That is, it runs all computation in a single process and does not communicate with any other processes. Thus, the standalone mode requires the input graph to have only one partition. This mode is mainly used for development and testing (e.g., develop and run the code in Jupyter Notebook). When a user executes a training script with a launch script (see the section of launch script), `DistGraph` runs in the distributed mode. The launch tool starts servers (node/edge feature access and graph sampling) behind the scene and loads the partition data in each machine automatically. `DistGraph` connects with the servers in the cluster of machines and access them through the network.

DistGraph creation

In the distributed mode, the creation of `DistGraph` requires the graph name given during graph partitioning. The graph name identifies the graph loaded in the cluster.

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

When running in the standalone mode, it loads the graph data in the local machine. Therefore, users need to provide the partition configuration file, which contains all information about the input graph.

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

Note

DGL only allows one single `DistGraph` object. The behavior of destroying a DistGraph and creating a new one is undefined.

Accessing graph structure

`DistGraph` provides a set of APIs to access the graph structure. Currently, most APIs provide graph information, such as the number of nodes and edges. The main use case of DistGraph is to run sampling APIs to support mini-batch training (see Distributed sampling).

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

Access node/edge data

Like `DGLGraph`, `DistGraph` provides `ndata` and `edata` to access data in nodes and edges. The difference is that `ndata` / `edata` in `DistGraph` returns `DistTensor`, instead of the tensor of the underlying framework. Users can also assign a new `DistTensor` to `DistGraph` as node data or edge data.

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

Distributed Tensor

As mentioned earlier, DGL shards node/edge features and stores them in a cluster of machines. DGL provides distributed tensors with a tensor-like interface to access the partitioned node/edge features in the cluster. In the distributed setting, DGL only supports dense node/edge features.

`DistTensor` manages the dense tensors partitioned and stored in multiple machines. Right now, a distributed tensor has to be associated with nodes or edges of a graph. In other words, the number of rows in a DistTensor has to be the same as the number of nodes or the number of edges in a graph. The following code creates a distributed tensor. In addition to the shape and dtype for the tensor, a user can also provide a unique tensor name. This name is useful if a user wants to reference a persistent distributed tensor (the one exists in the cluster even if the `DistTensor` object disappears).

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

Note

`DistTensor` creation is a synchronized operation. All trainers have to invoke the creation and the creation succeeds only when all trainers call it.

A user can add a `DistTensor` to a `DistGraph` object as one of the node data or edge data.

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

Note

The node data name and the tensor name do not have to be the same. The former identifies node data from `DistGraph` (in the trainer process) while the latter identifies a distributed tensor in DGL servers.

`DistTensor` has the same APIs as regular tensors to access its metadata, such as the shape and dtype. It also supports indexed reads and writes but does not support computation operators, such as sum and mean.

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

Note

Currently, DGL does not provide protection for concurrent writes from multiple trainers when a machine runs multiple servers. This may result in data corruption. One way to avoid concurrent writes to the same row of data is to run one server process on a machine.

Distributed DistEmbedding

DGL provides `DistEmbedding` to support transductive models that require node embeddings. Creating distributed embeddings is very similar to creating distributed tensors.

```
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)
```

Internally, distributed embeddings are built on top of distributed tensors, and, thus, has very similar behaviors to distributed tensors. For example, when embeddings are created, they are sharded and stored across all machines in the cluster. It can be uniquely identified by a name.

Note

The initializer function is invoked in the server process. Therefore, it has to be declared before `dgl.distributed.initialize`.

Because the embeddings are part of the model, a user has to attach them to an optimizer for mini-batch training. Currently, DGL provides a sparse Adagrad optimizer `SparseAdagrad` (DGL will add more optimizers for sparse embeddings later). Users need to collect all distributed embeddings from a model and pass them to the sparse optimizer. If a model has both node

embeddings and regular dense model parameters and users want to perform sparse updates on the embeddings, they need to create two optimizers, one for node embeddings and the other for dense model parameters, as shown in the code below:

```
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` does not inherit `torch.nn.Module`, so we recommend using it outside of your own NN module.

Distributed sampling

DGL provides two levels of APIs for sampling nodes and edges to generate mini-batches (see the section of mini-batch training). The low-level APIs require users to write code to explicitly define how a layer of nodes are sampled (e.g., using `dgl.sampling.sample_neighbors()`). The high-level sampling APIs implement a few popular sampling algorithms for node classification and link prediction tasks (e.g., `NodeDataLoader` and `EdgeDataLoader`).

The distributed sampling module follows the same design and provides two levels of sampling APIs. For the lower-level sampling API, it provides `sample_neighbors()` for distributed neighborhood sampling on `DistGraph`. In addition, DGL provides a distributed DataLoader (`DistDataLoader`) for distributed sampling. The distributed DataLoader has the same interface as Pytorch DataLoader except that users cannot specify the number of worker processes when creating a dataloader. The worker processes are created in `dgl.distributed.initialize()`.

Note

When running `dgl.distributed.sample_neighbors()` on `DistGraph`, the sampler cannot run in Pytorch DataLoader with multiple worker processes. The main reason is that Pytorch DataLoader creates new sampling worker processes in every epoch, which leads to creating and destroying `DistGraph` objects many times.

When using the low-level API, the sampling code is similar to single-process sampling. The only difference is that users need to use `dgl.distributed.sample_neighbors()` and `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:
        ...
```

The high-level sampling APIs (`NodeDataLoader` and `EdgeDataLoader`) has distributed counterparts (`DistNodeDataLoader` and `DistEdgeDataLoader`). The code is exactly the same as single-process sampling otherwise.

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

Split workloads

To train a model, users first need to split the dataset into training, validation and test sets. For distributed training, this step is usually done before we invoke `dgl.distributed.partition_graph()` to partition a graph. We recommend to store the data split in boolean arrays as node data or edge data. For node classification tasks, the length of these boolean arrays is the number of nodes in a graph and each of their elements indicates the existence of a node in a training/validation/test set. Similar boolean arrays should be used for link prediction tasks. `dgl.distributed.partition_graph()` splits these boolean arrays (because they are stored as the node data or edge data of the graph) based on the graph partitioning result and store them with graph partitions.

During distributed training, users need to assign training nodes/edges to each trainer. Similarly, we also need to split the validation and test set in the same way. DGL provides `node_split()` and `edge_split()` to split the training, validation and test set at runtime for distributed training. The two functions take the boolean arrays constructed before graph partitioning as input, split them and return a portion for the local trainer. By default, they ensure that all portions have the same number of nodes/edges. This is important for synchronous SGD, which assumes each trainer has the same number of mini-batches.

The example below splits the training set and returns a subset of nodes for the local process.

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