Chapter 7: Distributed Training

(中文版)

Note

Distributed training is only available for PyTorch backend.

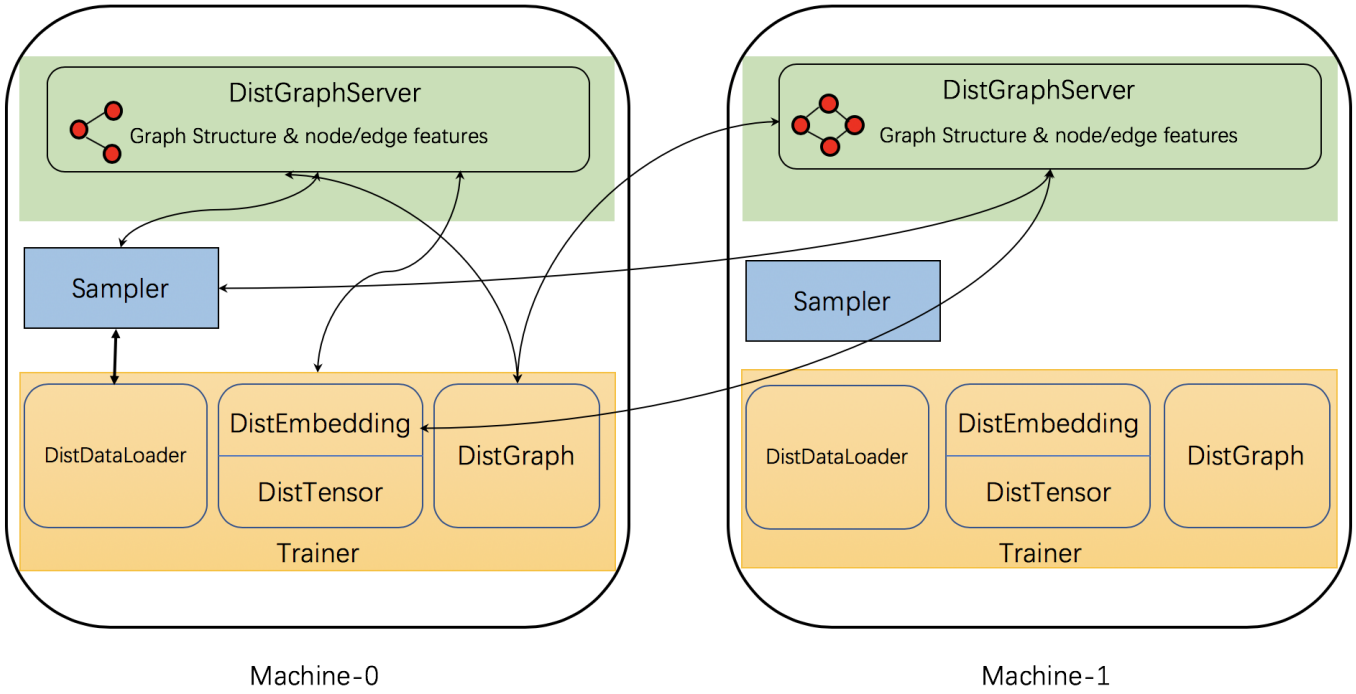
DGL adopts a fully distributed approach that distributes both data and computation across a collection of computation resources. In the context of this section, we will assume a cluster setting (i.e., a group of machines). DGL partitions a graph into subgraphs and each machine in a cluster is responsible for one subgraph (partition). DGL runs an identical training script on all machines in the cluster to parallelize the computation and runs servers on the same machines to serve partitioned data to the trainers.

For the training script, DGL provides distributed APIs that are similar to the ones for mini-batch training. This makes distributed training require only small code modifications from mini-batch training on a single machine. Below shows an example of training GraphSage in a distributed fashion. The notable code modifications are: 1) initialization of DGL's distributed module, 2) create a distributed graph object, and 3) split the training set and calculate the nodes for the local process. The rest of the code, including sampler creation, model definition, training loops are the same as mini-batch training.

```
import dgl
from dgl.dataloading import NeighborSampler
from dgl.distributed import DistGraph, DistDataLoader, node_split
import torch as th

# initialize distributed contexts
dgl.distributed.initialize('ip_config.txt')
th.distributed.init_process_group(backend='gloo')
# load distributed graph
g = DistGraph('graph_name', 'part_config.json')
pb = g.get_partition_book()
# get training workload, i.e., training node IDs
train_nid = node_split(g.ndata['train_mask'], pb, force_even=True)


# Create sampler
sampler = NeighborSampler(g, [10,25],
                          dgl.distributed.sample_neighbors,
                          device)

dataloader = DistDataLoader(
    dataset=train_nid.numpy(),
    batch_size=batch_size,
    collate_fn=sampler.sample_blocks,
    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):
    with model.join():
        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()
```

DGL implements a few distributed components to support distributed training. The figure below shows the components and their interactions.



Specifically, DGL's distributed training has three types of interacting processes: *server*, *sampler* and *trainer*.

- **Servers** store graph partitions which includes both structure data and node/edge features. They provide services such as sampling, getting or updating node/edge features. Note that each machine may run multiple server processes simultaneously to increase service throughput. One of them is *main server* in charge of data loading and sharing data via shared memory with *backup servers* that provide services.

- **Sampler processes** interact with the servers and sample nodes and edges to generate mini-batches for training.
- **Trainers** are in charge of training networks on mini-batches. They utilize APIs such as `DistGraph` to access partitioned graph data, `DistEmbedding` and `DistTensor` to access node/edge features/embeddings and `DistDataLoader` to interact with samplers to get mini-batches. Trainers communicate gradients among each other using PyTorch's native `DistributedDataParallel` paradigm.

Besides Python APIs, DGL also provides tools for provisioning graph data and processes to the entire cluster.

Having the distributed components in mind, the rest of the section will cover the following distributed components:

- 7.1 Data Preprocessing
- 7.2 Tools for launching distributed training/inference
- 7.3 Programming APIs

For more advanced users who are interested in more details:

- 7.4 Advanced Graph Partitioning
- 7.5 Heterogeneous Graph Under The Hood