7.2 Tools for launching distributed training/inference

DGL provides a launching script `launch.py` under dgl/tools to launch a distributed training job in a cluster. This script makes the following assumptions:

- The partitioned data and the training script have been provisioned to the cluster or a shared storage (e.g., NFS) accessible to all the worker machines.
- The machine that invokes `launch.py` has passwordless ssh access to all other machines. The launching machine must be one of the worker machines.

Below shows an example of launching a distributed training job in a cluster.

```
python3 tools/launch.py              \
  --workspace /my/workspace/         \
  --num_trainers 2                   \
  --num_samplers 4                   \
  --num_servers 1                    \
  --part_config data/mygraph.json    \
  --ip_config ip_config.txt          \
  "python3 my_train_script.py"
```

The argument specifies the workspace path, where to find the partition metadata JSON and machine IP configurations, how many trainer, sampler, and server processes to be launched on each machine. The last argument is the command to launch which is usually the model training/evaluation script.

Each line of `ip_config.txt` is the IP address of a machine in the cluster. Optionally, the IP address can be followed by a network port (default is `30050`). A typical example is as follows:

```
172.31.19.1
172.31.23.205
172.31.29.175
172.31.16.98
```

The workspace specified in the launch script is the working directory in the machines, which contains the training script, the IP configuration file, the partition configuration file as well as the graph partitions. All paths of the files should be specified as relative paths to the

workspace.

The launch script creates a specified number of training jobs (`--num_trainers`) on each machine. In addition, users need to specify the number of sampler processes for each trainer (`--num_samplers`).

Launching a Persistent Graph Server

> **Warning**
>
> Persistent graph server is an experimental feature. It is only available when the `net_etype` argument of `dgl.distributed.initialize()` is `"tensorpipe"` .

Normally, all the server and trainer processes will be killed after the training is done. However, sometimes users may wish to try out different models or training configurations against the *same* graph data. Repetitively loading the same graph data could be costly. To avoid that, DGL allows users to launch a persistent graph server to be shared across multiple training jobs. A persistent graph server will stay alive even all training workers have finished and exited. Below shows an example of launching a persistent graph server:

We first launch the graph server together with the first group of training workers.

```
python3 tools/launch.py              \
  --workspace /my/workspace/         \
  --num_trainers 2                   \
  --num_samplers 4                   \
  --num_servers 1                    \
  --part_config data/mygraph.json    \
  --ip_config ip_config.txt          \
  --keep_alive                       \
  --server_name long_live            \
  "python3 my_train_script.py"
```

Pay attention to the `--keep_alive` option, which indicates the server should stay alive after workers have finished. `--server_name` is the given name of the server which will be referred when launching new training jobs.

Then launch trainers as normal which will automatically connect to the existing persistent server.

```
python3 tools/launch.py                \
  --workspace /my/workspace/           \
  --num_trainers 2                     \
  --num_samplers 4                     \
  --num_servers 1                      \
  --part_config data/mygraph.json      \
  --ip_config ip_config.txt            \
  "python3 my_train_script.py"
```

There are several restrictions when using persistent graph servers:

- All the arguments for `launch.py` should be kept same as previous launch. And below arguments for specific training script should be kept same as well: `--graph-name`, `--ip_config`.
- There is no data consistency control on the server side so data update must be carefully handled. For example, it is recommended to avoid having multiple groups of trainers update node/edge embeddings at the same time.