PyTorch

Table of Contents

TORCHRUN (ELASTIC LAUNCH)

`torchrun` provides a superset of the functionality as `torch.distributed.launch` with the following additional functionalities:

1. Worker failures are handled gracefully by restarting all workers.
2. Worker `RANK` and `WORLD_SIZE` are assigned automatically.
3. Number of nodes is allowed to change between minimum and maximum sizes (elasticity).

• NOTE

`torchrun` is a python console script to the main module torch.distributed.run declared in the `entry_points` configuration in setup.py. It is equivalent to invoking `python -m torch.distributed.run`.

Transitioning from torch.distributed.launch to torchrun

`torchrun` supports the same arguments as `torch.distributed.launch` **except** for `--use-env` which is now deprecated. To migrate from `torch.distributed.launch` to `torchrun` follow these steps:

1. If your training script is already reading `local_rank` from the `LOCAL_RANK` environment variable. Then you need simply omit the `--use-env` flag, e.g.:

`torch.distributed.launch`	`torchrun`
`$ python -m torch.distributed.launch --use-env train_script.py`	`$ torchrun train_script.py`

2. If your training script reads local rank from a `--local-rank` cmd argument. Change your training script to read from the `LOCAL_RANK` environment variable as demonstrated by the following code snippet:

`torch.distributed.launch`	`torchrun`

`import argparse`
`parser = argparse.ArgumentParser()`
`parser.add_argument("--local-rank", type=int)`
`args = parser.parse_args()`
 	
`local_rank = args.local_rank`
	`import os`
	`local_rank = int(os.environ["LOCAL_RANK"])`

The aformentioned changes suffice to migrate from `torch.distributed.launch` to `torchrun`. To take advantage of new features such as elasticity, fault-tolerance, and error reporting of `torchrun` please refer to:

- Train script for more information on authoring training scripts that are `torchrun` compliant.
- the rest of this page for more information on the features of `torchrun`.

Usage

Single-node multi-worker

```
torchrun
    --standalone
    --nnodes=1
    --nproc-per-node=$NUM_TRAINERS
    YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

Stacked single-node multi-worker

To run multiple instances (separate jobs) of single-node, multi-worker on the same host, we need to make sure that each instance (job) is setup on different ports to avoid port conflicts (or worse, two jobs being merged as a single job). To do this you have to run with `--rdzv-backend=c10d` and specify a different port by setting `--rdzv-endpoint=localhost:$PORT_k`. For `--nodes=1`, its often convenient to let `torchrun` pick a free random port automatically instead of manually assigning different ports for each run.

```
torchrun
    --rdzv-backend=c10d
    --rdzv-endpoint=localhost:0
    --nnodes=1
    --nproc-per-node=$NUM_TRAINERS
    YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

Fault tolerant (fixed sized number of workers, no elasticity, tolerates 3 failures)

```
torchrun
    --nnodes=$NUM_NODES
    --nproc-per-node=$NUM_TRAINERS
    --max-restarts=3
    --rdzv-id=$JOB_ID
    --rdzv-backend=c10d
    --rdzv-endpoint=$HOST_NODE_ADDR
    YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

`HOST_NODE_ADDR`, in form <host>[:<port>] (e.g. node1.example.com:29400), specifies the node and the port on which the C10d rendezvous backend should be instantiated and hosted. It can be any node in your training cluster, but ideally you should pick a node that has a high bandwidth.

• NOTE

If no port number is specified `HOST_NODE_ADDR` defaults to 29400.

Elastic (`min=1`, `max=4`, tolerates up to 3 membership changes or failures)

```
torchrun
    --nnodes=1:4
    --nproc-per-node=$NUM_TRAINERS
    --max-restarts=3
    --rdzv-id=$JOB_ID
    --rdzv-backend=c10d
    --rdzv-endpoint=$HOST_NODE_ADDR
    YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

`HOST_NODE_ADDR`, in form <host>[:<port>] (e.g. node1.example.com:29400), specifies the node and the port on which the C10d rendezvous backend should be instantiated and hosted. It can be any node in your training cluster, but ideally you should pick a node that has a high bandwidth.

• NOTE

If no port number is specified `HOST_NODE_ADDR` defaults to 29400.

Note on rendezvous backend

For multi-node training you need to specify:

1. `--rdzv-id`: A unique job id (shared by all nodes participating in the job)
2. `--rdzv-backend`: An implementation of `torch.distributed.elastic.rendezvous.RendezvousHandler`
3. `--rdzv-endpoint`: The endpoint where the rendezvous backend is running; usually in form `host:port`.

Currently `c10d` (recommended), `etcd-v2`, and `etcd` (legacy) rendezvous backends are supported out of the box. To use `etcd-v2` or `etcd`, setup an etcd server with the `v2` api enabled (e.g. `--enable-v2`).

• WARNING

`etcd-v2` and `etcd` rendezvous use etcd API v2. You MUST enable the v2 API on the etcd server. Our tests use etcd v3.4.3.

• WARNING

For etcd-based rendezvous we recommend using `etcd-v2` over `etcd` which is functionally equivalent, but uses a revised implementation. `etcd` is in maintenance mode and will be removed in a future version.

Definitions

1. `Node` - A physical instance or a container; maps to the unit that the job manager works with.
2. `Worker` - A worker in the context of distributed training.
3. `WorkerGroup` - The set of workers that execute the same function (e.g. trainers).
4. `LocalWorkerGroup` - A subset of the workers in the worker group running on the same node.
5. `RANK` - The rank of the worker within a worker group.
6. `WORLD_SIZE` - The total number of workers in a worker group.
7. `LOCAL_RANK` - The rank of the worker within a local worker group.
8. `LOCAL_WORLD_SIZE` - The size of the local worker group.
9. `rdzv_id` - A user-defined id that uniquely identifies the worker group for a job. This id is used by each node to join as a member of a particular worker group.
9. `rdzv_backend` - The backend of the rendezvous (e.g. `c10d`). This is typically a strongly consistent key-value store.
10. `rdzv_endpoint` - The rendezvous backend endpoint; usually in form `<host>:<port>`.

A `Node` runs `LOCAL_WORLD_SIZE` workers which comprise a `LocalWorkerGroup`. The union of all `LocalWorkerGroups` in the nodes in the job comprise the `WorkerGroup`.

Environment Variables

The following environment variables are made available to you in your script:

1. `LOCAL_RANK` - The local rank.
2. `RANK` - The global rank.
3. `GROUP_RANK` - The rank of the worker group. A number between 0 and `max_nnodes`. When running a single worker group per node, this is the rank of the node.
4. `ROLE_RANK` - The rank of the worker across all the workers that have the same role. The role of the worker is specified in the `WorkerSpec`.
5. `LOCAL_WORLD_SIZE` - The local world size (e.g. number of workers running locally); equals to `--nproc-per-node` specified on `torchrun`.
6. `WORLD_SIZE` - The world size (total number of workers in the job).
7. `ROLE_WORLD_SIZE` - The total number of workers that was launched with the same role specified in `WorkerSpec`.
8. `MASTER_ADDR` - The FQDN of the host that is running worker with rank 0; used to initialize the Torch Distributed backend.
9. `MASTER_PORT` - The port on the `MASTER_ADDR` that can be used to host the C10d TCP store.
10. `TORCHELASTIC_RESTART_COUNT` - The number of worker group restarts so far.
11. `TORCHELASTIC_MAX_RESTARTS` - The configured maximum number of restarts.
12. `TORCHELASTIC_RUN_ID` - Equal to the rendezvous `run_id` (e.g. unique job id).
13. `PYTHON_EXEC` - System executable override. If provided, the python user script will use the value of `PYTHON_EXEC` as executable. The *sys.executable* is used by default.

Deployment

1. (Not needed for the C10d backend) Start the rendezvous backend server and get the endpoint (to be passed as `--rdzv-endpoint` to the launcher script)
2. Single-node multi-worker: Start the launcher on the host to start the agent process which creates and monitors a local worker group.
3. Multi-node multi-worker: Start the launcher with the same arguments on all the nodes participating in training.

When using a job/cluster manager the entry point command to the multi-node job should be this launcher.

Failure Modes

1. Worker failure: For a training job with `n` workers, if `k<=n` workers fail all workers are stopped and restarted up to `max_restarts`.
2. Agent failure: An agent failure results in a local worker group failure. It is up to the job manager to fail the entire job (gang semantics) or attempt to replace the node. Both behaviors are supported by the agent.
3. Node failure: Same as agent failure.

Membership Changes

1. Node departure (scale-down): The agent is notified of the departure, all existing workers are stopped, a new `WorkerGroup` is formed, and all workers are started with a new `RANK` and `WORLD_SIZE`.
2. Node arrival (scale-up): The new node is admitted to the job, all existing workers are stopped, a new `WorkerGroup` is formed, and all workers are started with a new `RANK` and `WORLD_SIZE`.

Important Notices

1. This utility and multi-process distributed (single-node or multi-node) GPU training currently only achieves the best performance using the NCCL distributed backend. Thus NCCL backend is the recommended backend to use for GPU training.
2. The environment variables necessary to initialize a Torch process group are provided to you by this module, no need for you to pass `RANK` manually. To initialize a process group in your training script, simply run:

```
>>> import torch.distributed as dist
>>> dist.init_process_group(backend="gloo|nccl")
```

3. In your training program, you can either use regular distributed functions or use `torch.nn.parallel.DistributedDataParallel()` module. If your training program uses GPUs for training and you would like to use `torch.nn.parallel.DistributedDataParallel()` module, here is how to configure it.

```
local_rank = int(os.environ["LOCAL_RANK"])
model = torch.nn.parallel.DistributedDataParallel(model,
                                                  device_ids=[local_rank],
                                                  output_device=local_rank)
```

Please ensure that `device_ids` argument is set to be the only GPU device id that your code will be operating on. This is generally the local rank of the process. In other words, the `device_ids` needs to be `[int(os.environ("LOCAL_RANK"))]`, and `output_device` needs to be `int(os.environ("LOCAL_RANK"))` in order to use this utility

4. On failures or membership changes ALL surviving workers are killed immediately. Make sure to checkpoint your progress. The frequency of checkpoints should depend on your job's tolerance for lost work.
5. This module only supports homogeneous `LOCAL_WORLD_SIZE`. That is, it is assumed that all nodes run the same number of local workers (per role).
6. `RANK` is NOT stable. Between restarts, the local workers on a node can be assigned a different range of ranks than before. NEVER hard code any assumptions about the stable-ness of ranks or some correlation between `RANK` and `LOCAL_RANK`.
7. When using elasticity (`min_size!=max_size`) DO NOT hard code assumptions about `WORLD_SIZE` as the world size can change as nodes are allowed to leave and join.
8. It is recommended for your script to have the following structure:

```
def main():
  load_checkpoint(checkpoint_path)
  initialize()
  train()

def train():
  for batch in iter(dataset):
    train_step(batch)

    if should_checkpoint:
      save_checkpoint(checkpoint_path)
```

9. (Recommended) On worker errors, this tool will summarize the details of the error (e.g. time, rank, host, pid, traceback, etc). On each node, the first error (by timestamp) is heuristically reported as the "Root Cause" error. To get tracebacks as part of this error summary print out, you must decorate your main entrypoint function in your training script as shown in the example below. If not decorated, then the summary will not include the traceback of the exception and will only contain the exitcode. For details on torchelastic error handling see: https://pytorch.org/docs/stable/elastic/errors.html

```
from torch.distributed.elastic.multiprocessing.errors import record

@record
def main():
    # do train
    pass

if __name__ == "__main__":
    main()
```

‹ Previous Next ›

© Copyright 2023, PyTorch Contributors.
Built with Sphinx using a theme provided by Read the Docs.

Docs

Access comprehensive developer documentation for PyTorch

View Docs ›

Tutorials

Get in-depth tutorials for beginners and advanced developers

View Tutorials ›

Resources

Find development resources and get your questions answered

View Resources ›

PyTorch

Get Started

Features

Ecosystem

Blog

Contributing

Resources

Tutorials

Docs

Discuss

Github Issues

Brand Guidelines

Stay up to date

Facebook

Twitter

YouTube

LinkedIn

PyTorch Podcasts

Spotify

Apple

Google

Amazon

Terms | Privacy

© Copyright The Linux Foundation. The PyTorch Foundation is a project of The Linux Foundation. For web site terms of use, trademark policy and other policies applicable to The PyTorch Foundation please see www.linuxfoundation.org/policies/. The PyTorch Foundation supports the PyTorch open source project, which has been established as PyTorch Project a Series of LF Projects, LLC. For policies applicable to the PyTorch Project a Series of LF Projects, LLC, please see www.lfprojects.org/policies/.