Getting Started

Installation

- Installing is as simple as `pip install deepspeed`, see more details.
- To get started with DeepSpeed on AzureML, please see the AzureML Examples GitHub (https://github.com/Azure/azureml-examples/tree/main/cli/jobs/deepspeed)
- DeepSpeed has direct integrations with HuggingFace Transformers (https://github.com/huggingface/transformers) and PyTorch Lightning (https://github.com/PyTorchLightning/pytorch-lightning). HuggingFace Transformers users can now easily accelerate their models with DeepSpeed through a simple `--deepspeed` flag + config file See more details (https://huggingface.co/docs/transformers/deepspeed). PyTorch Lightning provides easy access to DeepSpeed through the Lightning Trainer See more details (https://pytorch-lightning.readthedocs.io/en/stable/advanced/multi_gpu.html?highlight=deepspeed#deepspeed).
- DeepSpeed on AMD can be used via our ROCm images (https://hub.docker.com/r/deepspeed/rocm501/tags), e.g., `docker pull deepspeed/rocm501:ds060_pytorch110`.
- DeepSpeed also supports Intel Xeon CPU, Intel Data Center Max Series XPU, Intel Gaudi HPU, Huawei Ascend NPU etc, please refer to the accelerator setup guide

Writing DeepSpeed Models

DeepSpeed model training is accomplished using the DeepSpeed engine. The engine can wrap any arbitrary model of type `torch.nn.module` and has a minimal set of APIs for training and checkpointing the model. Please see the tutorials for detailed examples.

To initialize the DeepSpeed engine:

```
model_engine, optimizer, _, _ = deepspeed.initialize(args=cmd_args,
                                                     model=model,
                                                     model_parameters=params)
```

`deepspeed.initialize` ensures that all of the necessary setup required for distributed data parallel or mixed precision training are done appropriately under the hood. In addition to wrapping the model, DeepSpeed can construct and manage the training optimizer, data loader, and the learning rate scheduler based on the parameters passed to `deepspeed.initialize` and the DeepSpeed configuration file. Note that DeepSpeed automatically executes the learning rate schedule at every training step.

If you already have a distributed environment setup, you'd need to replace:

```
torch.distributed.init_process_group(...)
```

with:

```
deepspeed.init_distributed()
```

The default is to use the NCCL backend, which DeepSpeed has been thoroughly tested with, but you can also override the default (https://deepspeed.readthedocs.io/en/latest/initialize.html#distributed-initialization).

But if you don't need the distributed environment setup until after `deepspeed.initialize()` you don't have to use this function, as DeepSpeed will automatically initialize the distributed environment during its `initialize` . Regardless, you will need to remove `torch.distributed.init_process_group` if you already had it in place.

Training

Once the DeepSpeed engine has been initialized, it can be used to train the model using three simple APIs for forward propagation (callable object), backward propagation (`backward`), and weight updates (`step`).

```
for step, batch in enumerate(data_loader):
    #forward() method
    loss = model_engine(batch)

    #runs backpropagation
    model_engine.backward(loss)

    #weight update
    model_engine.step()
```

Under the hood, DeepSpeed automatically performs the necessary operations required for distributed data parallel training, in mixed precision, with a pre-defined learning rate scheduler:

- **Gradient Averaging**: in distributed data parallel training, `backward` ensures that gradients are averaged across data parallel processes after training on an `train_batch_size` .
- **Loss Scaling**: in FP16/mixed precision training, the DeepSpeed engine automatically handles scaling the loss to avoid precision loss in the gradients.
- **Learning Rate Scheduler**: when using a DeepSpeed's learning rate scheduler (specified in the `ds_config.json` file), DeepSpeed calls the `step()` method of the scheduler at every training step (when `model_engine.step()` is executed). When not using DeepSpeed's learning rate scheduler:
    - if the schedule is supposed to execute at every training step, then the user can pass the scheduler to `deepspeed.initialize` when initializing the DeepSpeed engine and let DeepSpeed manage it for update or save/restore.
    - if the schedule is supposed to execute at any other interval (e.g., training epochs), then the user should NOT pass the scheduler to DeepSpeed during initialization and must manage it explicitly.

Model Checkpointing

Saving and loading the training state is handled via the `save_checkpoint` and `load_checkpoint` API in DeepSpeed which takes two arguments to uniquely identify a checkpoint:

- `ckpt_dir` : the directory where checkpoints will be saved.
- `ckpt_id` : an identifier that uniquely identifies a checkpoint in the directory. In the following code snippet, we use the loss value as the checkpoint identifier.

```
#load checkpoint
_, client_sd = model_engine.load_checkpoint(args.load_dir, args.ckpt_id)
step = client_sd['step']

#advance data loader to ckpt step
dataloader_to_step(data_loader, step + 1)

for step, batch in enumerate(data_loader):

    #forward() method
    loss = model_engine(batch)

    #runs backpropagation
    model_engine.backward(loss)

    #weight update
    model_engine.step()

    #save checkpoint
    if step % args.save_interval:
        client_sd['step'] = step
        ckpt_id = loss.item()
        model_engine.save_checkpoint(args.save_dir, ckpt_id, client_sd =
client_sd)
```

DeepSpeed can automatically save and restore the model, optimizer, and the learning rate scheduler states while hiding away these details from the user. However, the user may want to save additional data that are unique to a given model training. To support these items, `save_checkpoint` accepts a client state dictionary `client_sd` for saving. These items can be retrieved from `load_checkpoint` as a return argument. In the example above, the `step` value is stored as part of the `client_sd` .

Important: all processes must call this method and not just the process with rank 0. It is because each process needs to save its master weights and scheduler+optimizer states. This method will hang waiting to synchronize with other processes if it's called just for the process with rank 0.

DeepSpeed Configuration

DeepSpeed features can be enabled, disabled, or configured using a config JSON file that should be specified as `args.deepspeed_config`. A sample config file is shown below. For a full set of features see API doc.

```
{
  "train_batch_size": 8,
  "gradient_accumulation_steps": 1,
  "optimizer": {
    "type": "Adam",
    "params": {
      "lr": 0.00015
    }
  },
  "fp16": {
    "enabled": true
  },
  "zero_optimization": true
}
```

Launching DeepSpeed Training

DeepSpeed installs the entry point `deepspeed` to launch distributed training. We illustrate an example usage of DeepSpeed with the following assumptions:

1. You have already integrated DeepSpeed into your model
2. `client_entry.py` is the entry script for your model
3. `client args` is the `argparse` command line arguments
4. `ds_config.json` is the configuration file for DeepSpeed

Resource Configuration (multi-node)

DeepSpeed configures multi-node compute resources with hostfiles that are compatible with OpenMPI (https://www.open-mpi.org/) and Horovod (https://github.com/horovod/horovod). A hostfile is a list of *hostnames* (or SSH aliases), which are machines accessible via passwordless SSH, and *slot counts*, which specify the number of GPUs available on the system. For example,

```
worker-1 slots=4
worker-2 slots=4
```

specifies that two machines named *worker-1* and *worker-2* each have four GPUs to use for training.

Hostfiles are specified with the `--hostfile` command line option. If no hostfile is specified, DeepSpeed searches for `/job/hostfile`. If no hostfile is specified or found, DeepSpeed queries the number of GPUs on the local machine to discover the number of local slots available.

The following command launches a PyTorch training job across all available nodes and GPUs specified in `myhostfile`:

```
deepspeed --hostfile=myhostfile <client_entry.py> <client args> \
  --deepspeed --deepspeed_config ds_config.json
```

Alternatively, DeepSpeed allows you to restrict distributed training of your model to a subset of the available nodes and GPUs. This feature is enabled through two command line arguments: `--num_nodes` and `--num_gpus`. For example, distributed training can be restricted to use only two nodes with the following command:

```
deepspeed --num_nodes=2 \
        <client_entry.py> <client args> \
        --deepspeed --deepspeed_config ds_config.json
```

You can instead include or exclude specific resources using the `--include` and `--exclude` flags. For example, to use all available resources **except** GPU 0 on node *worker-2* and GPUs 0 and 1 on *worker-3*:

```
deepspeed --exclude="worker-2:0@worker-3:0,1" \
        <client_entry.py> <client args> \
        --deepspeed --deepspeed_config ds_config.json
```

Similarly, you can use **only** GPUs 0 and 1 on *worker-2*:

```
deepspeed --include="worker-2:0,1" \
        <client_entry.py> <client args> \
        --deepspeed --deepspeed_config ds_config.json
```

Launching without passwordless SSH

DeepSpeed now supports launching training jobs without the need for passwordless SSH. This mode is particularly useful in cloud environments such as Kubernetes, where flexible container orchestration is possible, and setting up a leader-worker architecture with passwordless SSH adds unnecessary complexity.

To use this mode, you need to run the DeepSpeed command separately on all nodes. The command should be structured as follows:

```
deepspeed --hostfile=myhostfile --no_ssh --node_rank=<n> \
    --master_addr=<addr> --master_port=<port> \
    <client_entry.py> <client args> \
    --deepspeed --deepspeed_config ds_config.json
```

- `--hostfile=myhostfile` : Specifies the hostfile that contains information about the nodes and GPUs.
- `--no_ssh` : Enables the no-SSH mode.
- `--node_rank=<n>` : Specifies the rank of the node. This should be a unique integer from 0 to n - 1.
- `--master_addr=<addr>` : The address of the leader node (rank 0).
- `--master_port=<port>` : The port of the leader node.

In this setup, the hostnames in the hostfile do not need to be reachable via passwordless SSH. However, the hostfile is still required for the launcher to collect information about the environment, such as the number of nodes and the number of GPUs per node.

Each node must be launched with a unique `node_rank`, and all nodes must be provided with the address and port of the leader node (rank 0). This mode causes the launcher to act similarly to the `torchrun` launcher, as described in the PyTorch documentation (https://pytorch.org/docs/stable/elastic/run.html).

Multi-Node Environment Variables

When training across multiple nodes we have found it useful to support propagating user-defined environment variables. By default DeepSpeed will propagate all NCCL and PYTHON related environment variables that are set. If you would like to propagate additional variables you can specify them in a dot-file named `.deepspeed_env` that contains a new-line separated list of `VAR=VAL` entries. The DeepSpeed launcher will look in the local path you are executing from and also in your home directory (`~/`). If you would like to override the default name of this file or path and name with your own, you can specify this with the environment variable, `DS_ENV_FILE` . This is mostly useful if you are launching multiple jobs that all require different variables.

As a concrete example, some clusters require special NCCL variables to set prior to training. The user can simply add these variables to a `.deepspeed_env` file in their home directory that looks like this:

```
NCCL_IB_DISABLE=1
NCCL_SOCKET_IFNAME=eth0
```

DeepSpeed will then make sure that these environment variables are set when launching each process on every node across their training job.

MPI and AzureML Compatibility

As described above, DeepSpeed provides its own parallel launcher to help launch multi-node/multi-gpu training jobs. If you prefer to launch your training job using MPI (e.g., mpirun), we provide support for this. It should be noted that DeepSpeed will still use the torch distributed NCCL backend and *not* the MPI backend.

To launch your training job with mpirun + DeepSpeed or with AzureML (which uses mpirun as a launcher backend) you simply need to install the mpi4py (https://pypi.org/project/mpi4py/) python package. DeepSpeed will use this to discover the MPI environment and pass the necessary state (e.g., world size, rank) to the torch distributed backend.

If you are using model parallelism, pipeline parallelism, or otherwise require torch.distributed calls before calling `deepspeed.initialize(..)` we provide the same MPI support with an additional DeepSpeed API call. Replace your initial `torch.distributed.init_process_group(..)` call with:

```
deepspeed.init_distributed()
```

Resource Configuration (single-node)

In the case that we are only running on a single node (with one or more GPUs) DeepSpeed *does not* require a hostfile as described above. If a hostfile is not detected or passed in then DeepSpeed will query the number of GPUs on the local machine to discover the number of slots available. The `--include` and `--exclude` arguments work as normal, but the user should specify 'localhost' as the hostname.

Also note that `CUDA_VISIBLE_DEVICES` can be used with `deepspeed` to control which devices should be used on a single node. So either of these would work to launch just on devices 0 and 1 of the current node:

```
deepspeed --include localhost:0,1 ...
```

```
CUDA_VISIBLE_DEVICES=0,1 deepspeed ...
```

Updated: March 4, 2025