DISTRIBUTED TRAINING WITH UNEVEN INPUTS USING THE JOIN CONTEXT MANAGER

NOTE

`Join` is introduced in PyTorch 1.10 as a prototype feature. This API is subject to change.

In this recipe, you will see:

- An overview of the `Join` context manager.
- An example of how to use the context manager with `DistributedDataParallel`.
- An example of how to use the context manager with both `DistributedDataParallel` and `ZeroRedundancyOptimizer`.
- An example of passing in keyword arguments to the context manager.
- A dive into how the `Join` context manager works.
- An example showing how to make a toy class compatible with the context manager.

Requirements

- PyTorch 1.10+
- Getting Started with Distributed Data Parallel
- Shard Optimizer States with ZeroRedundancyOptimizer

What is `Join`?

In Getting Started with Distributed Data Parallel - Basic Use Case, you saw the general skeleton for using `DistributedDataParallel` to perform data parallel training. This implicitly schedules all-reduces in each backward pass to synchronize gradients across ranks. Such collective communications require participation from all ranks in the process group, so if a rank has fewer inputs, then the other ranks will hang or error (depending on the backend). More generally, this problem persists for any class that performs per-iteration synchronous collective communications.

`Join` is a context manager to be used around your per-rank training loop to facilitate training with uneven inputs. The context manager allows the ranks that exhaust their inputs early (i.e. *join early*) to shadow the collective communications performed by those that have not yet joined. The ways in which the communications are shadowed are specified by hooks.

Using `Join` with `DistributedDataParallel`

PyTorch's `DistributedDataParallel` works out-of-the-box with the `Join` context manager. Here is an example usage:

```
import os
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
from torch.distributed.algorithms.join import Join
from torch.nn.parallel import DistributedDataParallel as DDP

BACKEND = "nccl"
WORLD_SIZE = 2
NUM_INPUTS = 5

def worker(rank):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(BACKEND, rank=rank, world_size=WORLD_SIZE)

    model = DDP(torch.nn.Linear(1, 1).to(rank), device_ids=[rank])
    # Rank 1 gets one more input than rank 0
    inputs = [torch.tensor([1]).float() for _ in range(NUM_INPUTS + rank)]

    num_inputs = 0
    with Join([model]):
        for input in inputs:
            num_inputs += 1
            loss = model(input).sum()
            loss.backward()

    print(f"Rank {rank} has exhausted all {num_inputs} of its inputs!")

def main():
    mp.spawn(worker, nprocs=WORLD_SIZE, join=True)

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

This produces the following output (where the `print()` s from rank 0 and rank 1 may be arbitrarily ordered):

```
Rank 0 has exhausted all 5 of its inputs!
Rank 1 has exhausted all 6 of its inputs!
```

NOTE

`DistributedDataParallel` provided its own `join()` context manager prior to the introduction of this generic `Join` context manager. In the above example, using `with Join([model]):` is equivalent to using `with model.join():`. One limitation of the existing `DistributedDataParallel.join()` is that it does not allow multiple participating classes, e.g. `DistributedDataParallel` and `ZeroRedundancyOptimizer` together.

Using `Join` with `DistributedDataParallel` and `ZeroRedundancyOptimizer`

The `Join` context manager works not only with a single class but also with multiple classes together. PyTorch's `ZeroRedundancyOptimizer` is also compatible with the context manager, so here, we examine how to modify the previous example to use both `DistributedDataParallel` and `ZeroRedundancyOptimizer`:

```
from torch.distributed.optim import ZeroRedundancyOptimizer as ZeRO
from torch.optim import Adam

def worker(rank):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(BACKEND, rank=rank, world_size=WORLD_SIZE)

    model = DDP(torch.nn.Linear(1, 1).to(rank), device_ids=[rank])
    optim = ZeRO(model.parameters(), Adam, lr=0.01)
    # Rank 1 gets one more input than rank 0
    inputs = [torch.tensor([1]).float() for _ in range(NUM_INPUTS + rank)]

    num_inputs = 0
    # Pass both 'model' and 'optim' into 'Join()'
    with Join([model, optim]):
        for input in inputs:
            num_inputs += 1
            loss = model(input).sum()
            loss.backward()
            optim.step()

    print(f"Rank {rank} has exhausted all {num_inputs} of its inputs!")
```

This will yield the same output as before. The notable change was additionally passing in the `ZeroRedundancyOptimizer` instance into `Join()`.

Passing Keyword Arguments

Classes may provide keyword arguments that modify their behavior in the context manager at run time. For example, `DistributedDataParallel` provides an argument `divide_by_initial_world_size`, which determines if gradients are divided by the initial world size or by the effective world size (i.e. number of non-joined ranks). Such keyword arguments can be passed directly into the context manager.

```
with Join([model, optim], divide_by_initial_world_size=False):
    for input in inputs:
        ...
```


ZeroRedundancyOptimizer
Passing Keyword Arguments
+ How Does Join Work?
Making a Toy Class Work with Join

ZeroRedundancyOptimizer
Passing Keyword Arguments
+ How Does Join Work?
Making a Toy Class Work with Join

ZeroRedundancyOptimizer
Passing Keyword Arguments
+ How Does Join Work?
Making a Toy Class Work with Join

• WARNING

The keyword arguments passed into the context manager are shared across all participating classes. This should not be a limitation since we do not expect cases where multiple `Joinable`s need differing settings of the same argument. Nonetheless, this is something to keep in mind.

How Does `Join` Work?

Now that we have seen some preliminary examples of how to use the `Join` context manager, let us delve deeper into how it works. This will provide a greater insight into the full capability that it offers and prepare you to make your own custom classes compatible. Here, we will go over the `Join` class as well as the supporting classes `Joinable` and `JoinHook`.

`Joinable`

To begin, classes compatible with the `Join` context manager must inherit from the abstract base class `Joinable`. In particular, a `Joinable` must implement:

- `join_hook(self, **kwargs) -> JoinHook`

This returns the `JoinHook` instance for the `Joinable`, determining how joined processes should shadow the per-iteration collective communications performed by the `Joinable`.

- `join_device(self) -> torch.device`

This returns a device to be used by the `Join` context manager to perform collective communications, e.g. `torch.device("cuda:0")` or `torch.device("cpu")`.

- `join_process_group(self) -> ProcessGroup`

This returns the process group to be used by the `Join` context manager to perform collective communications.

`DistributedDataParallel` and `ZeroRedundancyOptimizer` already inherit from `Joinable` and implement the above methods, which is why we could directly use them in the previous examples.

`Joinable` classes should make sure to call the `Joinable` constructor since it initializes a `JoinConfig` instance, which is used internally by the context manager to ensure correctness. This will be saved in each `Joinable` as a field `_join_config`.

`JoinHook`

Next, let us break down the `JoinHook` class. A `JoinHook` provides two entry points into a context manager:

- `main_hook(self) -> None`

This hook is called repeatedly by each joined rank while there exists a rank that has not yet joined. It is meant to shadow the collective communications performed by the `Joinable` in each training iteration (e.g. in one forward pass, backward pass, and optimizer step).

- `post_hook(self, is_last_joiner: bool) -> None`

This hook is called once all ranks have joined. It is passed an additional `bool` argument `is_last_joiner`, which indicates if the rank was one of the last to join. The argument may be useful for synchronization.

To provide concrete examples of what these hooks may look like, the `ZeroRedundancyOptimizer` main hook performs an optimizer step per normal since the joined rank is still responsible for updating and synchronizing its shard of the parameters, and the `DistributedDataParallel` post-hook broadcasts the final updated model from one of the last joining ranks to ensure that it is the same across all ranks.

`Join`

Finally, let us examine how these fit into the `Join` class itself.

- `__init__(self, joinables: List[Joinable], enable: bool = True, throw_on_early_termination: bool = False)`

As we saw in the previous examples, the constructor takes in a list of the `Joinable`s that participate in the training loop. These should be the classes that perform collective communciations in each iteration.

`enable` is a `bool` that can be set to `False` if you know that there will not be uneven inputs, in which case the context manager becomes vacuous similar to `contextlib.nullcontext()`. This also may disable join-related computation in the participating `Joinable`s.

`throw_on_early_termination` is a `bool` that can be set to `True` to have each rank raise an exception the moment that uneven inputs are detected. This is useful for cases that do not conform to the context manager's requirements, which is most typically when there are collective communications from different classes that may be arbitrarily interleaved, such as when using `DistributedDataParallel` with a model that has `SyncBatchNorm` layers. In such cases, this argument should be set to `True` so that the application logic can catch the exception and determine how to proceed.

- The core logic occurs in the `__exit__()` method, which loops while there exists a non-joined rank, calling each `Joinable`'s main hook, and then once all ranks have joined, calls their post hooks. Both the main hooks and post-hooks are iterated over in the order that the `Joinable`s are passed in.
- The context manager requires a heartbeat from non-joined processes. As such, each `Joinable` class should make a call to `Join.notify_join_context()` before its per-iteration collective communications. The context manager will ensure that only the first `Joinable` passed in actually sends the heartbeat.

• WARNING

As mentioned above regarding `throw_on_early_termination`, the `Join` context manager is not compatible with certain compositions of classes. The `Joinable`'s `JoinHook`s must be serializable since each hook is fully executed before proceeding to the next. In other words, two hooks cannot overlap. Moreover, currently, both the main hooks and post-hooks are iterated over in the same deterministic order. If this appears to be a major limitation, we may modify the API to permit a customizable ordering.

Making a Toy Class Work with `Join`

Since the previous section introduced several concepts, let us see them in practice with a toy example. Here, we will implement a class that counts the number of inputs that are seen across all ranks before its rank joins. This should provide a basic idea of how you may make your own class compatible with the `Join` context manager.

Specifically, the following code has each rank print out (1) the number of inputs across all ranks that seen before it joins and (2) the total number of inputs across all ranks.

```
import os
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
from torch.distributed.algorithms.join import Join, Joinable, JoinHook

BACKEND = "nccl"
WORLD_SIZE = 2
NUM_INPUTS = 5

class CounterJoinHook(JoinHook):
    r"""
    Join hook for :class:`Counter`.

    Arguments:
        counter (Counter): the :class:`Counter` object using this hook.
        sync_max_count (bool): whether to sync the max count once all ranks
            join.
    """
    def __init__(
        self,
        counter,
        sync_max_count
    ):
        self.counter = counter
        self.sync_max_count = sync_max_count

    def main_hook(self):
        r"""
        Shadows the counter's all-reduce by all-reducing a dim-1 zero tensor.
        """
        t = torch.zeros(1, device=self.counter.device)
        dist.all_reduce(t)

    def post_hook(self, is_last_joiner: bool):
        r"""
        Synchronizes the max count across all :class:`Counter` s if
        ``sync_max_count=True``.
        """
        if not self.sync_max_count:
            return
        rank = dist.get_rank(self.counter.process_group)
        common_rank = self.counter.find_common_rank(rank, is_last_joiner)
        if rank == common_rank:
            self.counter.max_count = self.counter.count.detach().clone()
```

```
        dist.broadcast(self.counter.max_count, src=common_rank)

class Counter(Joinable):
    r"""
    Example :class:`Joinable` that counts the number of training iterations
    that it participates in.
    """
    def __init__(self, device, process_group):
        super(Counter, self).__init__()
        self.device = device
        self.process_group = process_group
        self.count = torch.tensor([0], device=device).float()
        self.max_count = torch.tensor([0], device=device).float()

    def __call__(self):
        r"""
        Counts the number of inputs processed on this iteration by all ranks
        by all-reducing a dim-1 one tensor; increments its own internal count.
        """
        Join.notify_join_context(self)
        t = torch.ones(1, device=self.device).float()
        dist.all_reduce(t)
        self.count += t

    def _join_hook(self, **kwargs) -> JoinHook:
        r"""
        Return a join hook that shadows the all-reduce in :meth:`__call__`.

        This join hook supports the following keyword arguments:
            sync_max_count (bool, optional): whether to synchronize the maximum
                count across all ranks once all ranks join; default is ``False``.
        """
        sync_max_count = kwargs.get("sync_max_count", False)
        return CounterJoinHook(self, sync_max_count)

    @property
    def _join_device(self) -> torch.device:
        return self.device

    @property
    def _join_process_group(self):
        return self.process_group

    def find_common_rank(self, rank, to_consider):
        r"""
        Returns the max rank of the ones to consider over the process group.
        """
        common_rank = torch.tensor([rank if to_consider else -1], device=self.device)
        dist.all_reduce(common_rank, op=dist.ReduceOp.MAX, group=self.process_group)
        common_rank = common_rank.item()
        return common_rank

def worker(rank):
    assert torch.cuda.device_count() >= WORLD_SIZE
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(BACKEND, rank=rank, world_size=WORLD_SIZE)

    counter = Counter(torch.device(f"cuda:{rank}"), dist.group.WORLD)
    inputs = [torch.tensor([1]).float() for _ in range(NUM_INPUTS + rank)]

    with Join([counter], sync_max_count=True):
        for _ in inputs:
            counter()

    print(f"{int(counter.count.item())} inputs processed before rank {rank} joined!")
    print(f"{int(counter.max_count.item())} inputs processed across all ranks!")

def main():
    mp.spawn(worker, nprocs=WORLD_SIZE, join=True)

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

Since rank 0 sees 5 inputs and rank 1 sees 6, this yields the output:

```
10 inputs processed before rank 0 joined!
11 inputs processed across all ranks!
11 inputs processed before rank 1 joined!
11 inputs processed across all ranks!
```

Some key points to highlight:

- A `Counter` instance performs a single all-reduce per iteration, so the main hook performs a single all-reduce as well to shadow it.
- The `Counter` class makes a call to `Join.notify_join_context()` at the beginning of its `__call__()` method since that is a place before its per- iteration collective communications (i.e. its all-reduce).
- The `is_last_joiner` argument is used to determine the broadcast source in the post-hooks.
- We pass in the `sync_max_count` keyword argument to the context manager, which is then forwarded to `Counter` 's join hook.

`ZeroRedundancyOptimizer`
Passing Keyword Arguments
+ How Does `Join` Work?
Making a Toy Class Work with `Join`

`ZeroRedundancyOptimizer`
Passing Keyword Arguments
+ How Does `Join` Work?
Making a Toy Class Work with `Join`

Rate this Tutorial ☆☆☆☆☆