Writing Distributed Applications with PyTorch

Sebastien Arnold

June 14, 2017

Abstract

In this short tutorial, we will be going over the distributed package of PyTorch. We'll see how to set up the distributed setting, use the different communication strategies, and go over part of the internals of the package.

1 Setup

The distributed package included in PyTorch (i.e., `torch.distributed`) enables researchers and practitioners to easily distribute their computations across processes and clusters of machines. To do so, it leverages the messaging passing semantics allowing each process to communicate data to any of the other processes. As opposed to the multiprocessing (`torch.multiprocessing`) package, processes can use different communication backends and are not restricted to being executed on the same machine.

In order to get started we should thus be able to run multiple processes simultaneously. If you have access to compute cluster you should check with your local sysadmin or use your favorite coordination tool. (e.g., pdsh, clustershell, or others) For the purpose of this tutorial, we will use a single machine and can fork multiple processes using the following template.

```
"""run.py:"""

#!/usr/bin/env python
import os
import torch
```

```
import torch.distributed as dist
from torch.multiprocessing import Process

def run(rank, size):
    """ Distributed function to be implemented later. """
    pass

def init_processes(rank, size, fn, backend='tcp'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=size)
    fn(rank, size)


if __name__ == "__main__":
    size = 2
    processes = []
    for rank in range(size):
        p = Process(target=init_processes, args=(rank, size, run))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()
```

In the above, the script spawns two processes who will each setup the distributed environment, initialize the process group (`dist.init_process_group`), and finally execute the given function.

The `init_processes` function is what interests us for now. It ensures that every process will be able to coordinate through a master, using the same ip address and port. Note that we used the TCP backend, but we could have used MPI or Gloo instead, provided they are installed. We will go over the magic hapening in `dist.init_process_group` at the end of this tutorial, but it essentially allows processes to communicate with each other by sharing their locations.

2 Point-to-Point Communication

A transfer of data from one process to another is called a point-to-point communication. These are achieved through the `send` and `recv` functions or their *immediate* counter-parts, `isend` and `irecv`.

```
"""Blocking point-to-point communication."""

def run(rank, size):
    tensor = torch.zeros(1)
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
        dist.send(tensor=tensor, dst=1)
    else:
        # Receive tensor from process 0
        dist.recv(tensor=tensor, src=0)
    print('Rank ', rank, ' has data ', tensor[0])
```

In the above example, both processes start with a zero tensor, then process 0 increments the tensor and sends it to process 1 so that they both end up with 1.0. Notice that process 1 needs to allocate memory in order to store the data it will receive.

Also notice that `send`/`recv` are **blocking**: both processes stop until the communication is completed. Immediates on the other hand are **non-blocking**, the script continues its execution and the methods return a `DistributedRequest` object upon which we can choose to `wait()`.

```
"""Non-blocking point-to-point communication."""

def run(rank, size):
    tensor = torch.zeros(1)
    req = None
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
```

```
        req = dist.isend(tensor=tensor, dst=1)
        print('Rank 0 started sending')
    else:
        # Receive tensor from process 0
        req = dist.irecv(tensor=tensor, src=0)
        print('Rank 1 started receiving')
        print('Rank 1 has data ', tensor[0])
    req.wait()
    print('Rank ', rank, ' has data ', tensor[0])
```

Running the above function a couple of times will sometimes result in process 1 still having 0.0 while having already started receiving. However, after `req.wait()` has been executed we are guaranteed that the communication took place.

Point-to-point communication is useful when we want a fine-grained control over the communication of our processes. They can be used to implement fancy algorithms, such as the one used in Baidu's DeepSpeech or Facebook's large-scale experiments.

3 Collective Communication

As opposed to point-to-point communcation, collectives allow for communication patterns across all processes in a **group**. A group is a subset of all your processes. To create a group, we can pass a list of ranks to `dist.new_group(group)`. By default, collectives are executed on the all processes, also known as the **world**. Then, in order to obtain the sum of all tensors at all processes, we can use the `dist.all_reduce(tensor, op, group)` collective.

```
""" All-Reduce example."""
def run(rank, size):
    """ Simple point-to-point communication. """
    group = dist.new_group([0, 1])
    tensor = torch.ones(1)
    dist.all_reduce(tensor, op=dist.reduce_op.SUM, group=group)
    print('Rank ', rank, ' has data ', tensor[0])
```

Since we wanted the sum of all tensors in the group, we used `dist.reduce_op.SUM` as the reduce operator. Generally speaking, any commutative mathematical operation can be used as an operator. PyTorch comes with 4 out-of-the-box, all working at the element-wise level:

- `dist.reduce_op.SUM`,
- `dist.reduce_op.PRODUCT`,
- `dist.reduce_op.MAX`,
- `dist.reduce_op.MIN`.

In addition to `dist.all_reduce(tensor, op, group)`, there are a total of 4 collectives that are currently implemented in PyTorch.

- `dist.broadcast(tensor, src, group)`: Copies tensor from src to all other processes.
- `dist.reduce(tensor, dst, op, group)`: Applies op to all tensor and stores the result at dst.
- `dist.all_reduce(tensor, op, group)`: Same as reduce, but the result is stored at all processes.
- `dist.all_gather(tensor_list, tensor, group)`: Copies tensor from all processes to tensor_list, on all processes.

3.0.1 What about scatter and gather ?

Those familiar with MPI will have noticed that the gather and scatter methods are absent from the current API. However, PyTorch exposes

- `dist.scatter_send(tensor_list, tensor, group)`,
- `dist.scatter_recv(tensor, dst, group)`,
- `dist.gather_send(tensor_list, tensor, group)`, and
- `dist.gather_recv(tensor, dst, group)`

which can be used to implement the standard scatter and gather behaviours.

```
""" Custom scatter and gather implementation. """

def scatter(tensor, rank, tensor_list=None, root=0, group=None):
    """
        Sends the ith tensor in tensor_list on root to the ith process.
```

```
    """
    if group is None:
        group = dist.group.WORLD
    if rank == root:
        assert(tensor_list is not None)
        dist.scatter_send(tensor_list, tensor, group)
    else:
        dist.scatter_recv(tensor, root, group)


def gather(tensor, rank, tensor_list=None, root=0, group=None):
    """
        Sends tensor to root process, which store it in tensor_list.
    """
    if group is None:
        group = dist.group.WORLD
    if rank == root:
        assert(tensor_list is not None)
        dist.gather_recv(tensor_list, tensor, group)
    else:
        dist.gather_send(tensor, root, group)
```

4 Distributed Training

- Gloo Backend
- Simple all_reduce on the gradients
- Point to optimized DistributedDataParallel

5 Internals

- The magic behind init_process_group

5.0.1 Acknowledgements

- PyTorch docs + well written tests.

5.0.2 Questions

- Why scatter_send/recv and gather_send/recv ? And why no gather() / scatter() ?
- How to install gloo ?