Writing Distributed Applications with PyTorch

SÃľb 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.

```python
"""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 happening 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

Send and Recv

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`.

```python
"""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()`.

```python
"""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

Broadcast
AllGather
Reduce
AllReduce
Scatter
Gather

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 6 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.scatter(tensor, src, scatter_list, group)`: Copies `scatter_list[i]` to the i^{th} process.
- `dist.gather(tensor, dst, gather_list, group)`: Copies tensor from all processes to dst.
- `dist.all_gather(tensor_list, tensor, group)`: Copies tensor from all processes to tensor_list, on all processes.

4 Distributed Training

Note: You can find the full script of this example here.

Now that we understand how the distributed module works, let us write something useful with it. Our goal will be to replicate the functionality of DistributedDataParallel. Of course, this will be a didactic example and in a real-world situtation you should use the official, well-tested and well-optimized version linked above.

Quite simply we want to implement a distributed version of stochastic gradient descent. Our script will let all processes compute the gradients of their model on their batch of data and then average their gradients. In order to ensure replicability across runs, we will first have to partition our dataset. (You could also use tnt.dataset.SplitDataset])

```
""" Dataset partitioning helper """
class Partition(object):

    def __init__(self, data, index):
        self.data = data
        self.index = index

    def __len__(self):
        return len(self.index)

    def __getitem__(self, index):
        data_idx = self.index[index]
        return self.data[data_idx]


class DataPartitioner(object):

    def __init__(self, data, sizes=[0.7, 0.2, 0.1], seed=1234):
        self.data = data
        self.partitions = []
        rng = Random()
        rng.seed(seed)
        data_len = len(data)
        indexes = [x for x in range(0, data_len)]
        rng.shuffle(indexes)

        for frac in sizes:
            part_len = int(frac * data_len)
            self.partitions.append(indexes[0:part_len])
            indexes = indexes[part_len:]

    def use(self, partition):
        return Partition(self.data, self.partitions[partition])
```

With the above snippet, we can now simply partition any dataset using the following few lines:

```
""" Partitioning MNIST """
def partition_dataset():
    dataset = datasets.MNIST('./data', train=True, download=True,
                             transform=transforms.Compose([
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.1307,), (0.3081,))
                             ]))
    size = dist.get_world_size()
    bsz = 128 / float(size)
    partition_sizes = [1.0 / size for _ in range(size)]
    partition = DataPartitioner(dataset, partition_sizes)
    partition = partition.use(dist.get_rank())
    train_set = torch.utils.data.DataLoader(partition,
                                         batch_size=bsz,
                                         shuffle=True)
    return train_set, bsz
```

Assuming that we have 2 replicas, then each process will have a `train_set` of 60000 / 2 = 30000 samples. We also divide the batch size by the number of replicas in order to maintain the overall batch size of 128.

We can now write our usual forward-backward-optimize training code, and include a method to average the gradients of our models. (The following is largely inspired from the official PyTorch MNIST example.)

```
""" Distributed Synchronous SGD Example """
def run(rank, size):
        torch.manual_seed(1234)
        train_set, bsz = partition_dataset()
        model = Net()
        optimizer = optim.SGD(model.parameters(),
                              lr=0.01, momentum=0.5)
        num_batches = ceil(len(train_set.dataset) / float(bsz))
        for epoch in range(10):
            epoch_loss = 0.0
            for data, target in train_set:
                data, target = Variable(data), Variable(target)
                optimizer.zero_grad()
                output = model(data)
                loss = F.nll_loss(output, target)
                epoch_loss += loss.data[0]
                loss.backward()
                average_gradients(model)
                optimizer.step()
            print('Rank ', dist.get_rank(), ', epoch ',
                  epoch, ': ', epoch_loss / num_batches)
```

It remains to implement the `average_gradients(model)` function, which simply takes in a model and averages its gradients across the group.

```
""" Gradient averaging. """
def average_gradients(model):
    size = float(dist.get_world_size())
    for param in model.parameters():
        dist.all_reduce(param.grad.data, op=dist.reduce_op.SUM, group=0)
        param.grad.data /= size
```

Et voilà ! We successfully implemented distributed synchronous SGD and could train any model on a large computer cluster.

Note: While the last sentence is *technically* true, there are a lot more tricks required to implement a production-level implementation of synchronous SGD. Again, use what has been tested.

5 Advanced Topics

We are now ready to discover some of the more advanced functionalities of `torch.distributed`. Since there is a lot to cover, this section is divided into three subsections:

1. Communication Backends: Where we learn how to use MPI and Gloo for GPU-GPU communication.
2. Initialization Methods: Where we understand how to best setup the initial coordination phase in `dist.init_process_group()`.
3. Internals: Where we take a look at what is happening under the hood.

5.1 Communication Backends

One of the most elegant aspects of `torch.distributed` is its ability to use different backends. As mentioned before, there are currently three backends implemented in PyTorch: TCP, MPI, and Gloo. They all support different functions, depending on whether you use CPUs or GPUs. A comparative table can be found here.

5.1.1 TCP Backend

5.1.2 Gloo Backend

5.1.3 MPI Backend

5.2 Initialization Methods

5.2.1 Environment Variable

5.2.2 TCP Init & Multicast

5.2.3 Shared File System

5.3 Internals

- The magic behind init_process_group:

1. validate and parse the arguments
2. resolve the backend: name2channel.at()
3. Drop GIL & THDProcessGroupInit: instantiate the channel and add address of master from config
4. rank 0 inits master, others workers
5. master: create sockets for all workers -> wait for all workers to connect -> send them each the info about location of other processes
6. worker: create socket to master, send own info, receive info about each worker, and then handshake with each of them
7. By this time everyone has handshake with everyone.

5.3.1 Acknowledgements

- PyTorch docs + well written tests.

5.3.2 Questions

- Why scatter_send/recv and gather_send/recv ? And why no gather() / scatter() ?
- How to get started with gloo ? Does it support ptp ?