Pytorch Parallelism

> 💡 介绍Pytorch的并行计算机制

TODO:

1. 介绍Pipeline Parallelism
2. 实现3D并行示例代码

简介

Pytorch的distributed模块提供了底层的通讯机制，DeepSpeed和Megatron都是调用这套机制实现并行策略定制的。所以理论上来说，我们可以用pytorch自己实现DeepSpeed和Megatron。进程之间的通讯机制可参考**WRITING DISTRIBUTED APPLICATIONS WITH PYTORCH**

Pytorch并行机制的核心在于是对进程rank （id）的使用。

数据并行（Data Parallel）

每个进程都能获取所有的数据，但是每个进程使用数据的不同部分进行训练，然后集合这些部分数据上的训练信号更新模型。

实现上，DistributedDataParallel 根据进程的rank来指导进程获取其对应的数据切片（DistributedSampler），如：

第i个进程只索引整个数据集rank_i * num_per_rank 到 (rank_i+1)*num_per_rank位置的数据。

注意，每个进程存储了所有的模型参数，只是在参数更新的时候会利用其他进程的训练信号（all_reduce）。

张量并行（Tensor Parallel）

张量并行是在不同的进程中存储矩阵的不同部分来实现的。

实现上，一般是通过进程的rank（或者local_rank in group）来指导进程存储指定参数，比如一个想把一个矩阵前n行存储在rank=0的进程中，后m行元素存储在rank=1的进程中，此时我们可以在rank=0的进程中创建一个行数为n的矩阵，在rank=1的进程中创建一个行数为m的矩阵（condition on rank）：

```
if rank == 0:
  self.w = np.random.rand(n, d)
elif rank == 1:
  self.w = np.random.rand(m, d)
```

流水线并行（Pipeline Parallel）

流水线并行是将模型的不同层（如Transformer不同的encoder层）放到不同的进程（GPU) 中，从而克服一个GPU或一台机器无法存储所有模型参数的问题。一种典型的解决方案是把不同层放到同一台机器的不同GPU上：

```
# Need to initialize RPC framework first.
os.environ['MASTER_ADDR'] = 'localhost'
os.environ['MASTER_PORT'] = '29500'
torch.distributed.rpc.init_rpc('worker', rank=0, world_size=1)
# Build pipe.
```

```
fc1 = nn.Linear(16, 8).cuda(0)
fc2 = nn.Linear(8, 4).cuda(1)
model = nn.Sequential(fc1, fc2)
model = Pipe(model, chunks=8)
input = torch.rand(16, 16).cuda(0)
output_rref = model(input)
output = output_rref.local_value() # value is placed on CPU
output = output.cuda(0)
```

更复杂的流水线并行是将不同的模块放到不同的机器上。在一个流水线并行的group（所有参与流水线并行的进程组成的一个group）中，用一个主进程在其他进程中创建不同的模块，然后在主进程中按照模型顺序调用这些模块。具体实现可参考RPC (Remote Procedure Call) 文档。

常见术语定义

1. `Node` **- A physical instance or a container; maps to the unit that the job manager works with. 节点，也即机器**
2. `Worker` **- A worker in the context of distributed training. 进程**
3. `WorkerGroup` **- The set of workers that execute the same function (e.g. trainers). 分布式训练中所有的进程组成的group**
4. `LocalWorkerGroup` **- A subset of the workers in the worker group running on the same node. 在同一台机器上运行的分布式进程组成的group**
5. `RANK` **- The rank of the worker within a worker group. 一个group里面的进程id**
6. `WORLD_SIZE` **- The total number of workers in a worker group. 一个group中的所有进程**
7. `LOCAL_RANK` **- The rank of the worker within a local worker group.**
8. `LOCAL_WORLD_SIZE` **- The size of the local worker group. 当前进程所在的机器上运行的分布式进程的数目**
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.**
10. `rdzv_backend` **- The backend of the rendezvous (e.g.** `c10d`**). This is typically a strongly consistent key-value store.**
11. `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:

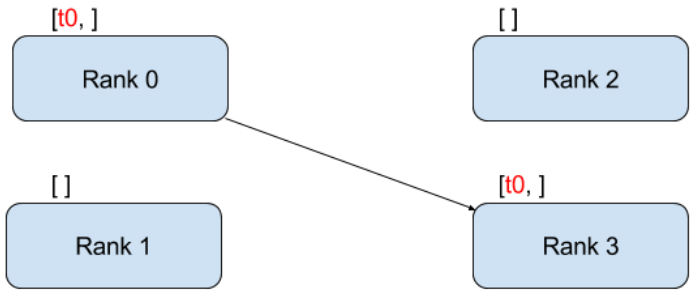
1. `LOCAL_RANK` **- The local rank.**
2. `RANK` **- The global rank.**
3. `GROUP_RANK` **- The rank of the worker group. A number between 0 and** `max_nnodes`**. When running a single worker group per node, this is the rank of the node.**
4. `ROLE_RANK` **- The rank of the worker across all the workers 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.**

例如，可以在进程中`用` `int(os.environ["LOCAL_RANK"])` **获取当前进程的local_rank**

Torch 基础通讯机制和API

点到点通讯



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`.

```
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])
```

这里需要注意的是rank0 send的tensor的size要和rank1 receive 的tensor的size一致; send出来的tensor要被recv接收，否则rank0会一直等待，如果rank1结束运行前没有receive信号那么rank0会会报错；如果rank0 多次send，则rank1需要相同次数的recv，同时第i次接收的信号对应rank0第i次send的信号。

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.**

Isend & Irecv

异步通讯，isend对象不必等到发出的信号被接收也能继续运行程序，同时可以通过返回对象的wait()函数来确保信号已经被接收；irecv对象也不必等到已经接收到信号才能继续运行程序，同时通过wait()函数来确保已经接收到了目标信号。

如果在信号被接收之前 isend对象修改了send的信号，那么可能会出错；如果在接收到信号之前irecv对象就使用了信号也会导致使用的信号不是正确的信号。

On the other hand immediates are non-blocking; the script continues its execution and the methods return a `Work` object upon which we can choose to `wait()`.

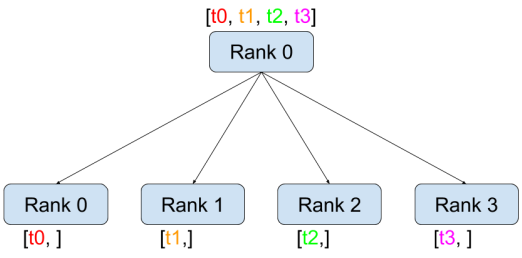
```
"""Non-blocking point-to-point communication."""

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')
    req.wait()
    print('Rank ', rank, ' has data ', tensor[0])
```

When using immediates we have to be careful about how we use the sent and received tensors. Since we do not know when the data will be communicated to the other process, we should not modify the sent tensor nor access the received tensor before `req.wait()` has completed. In other words,

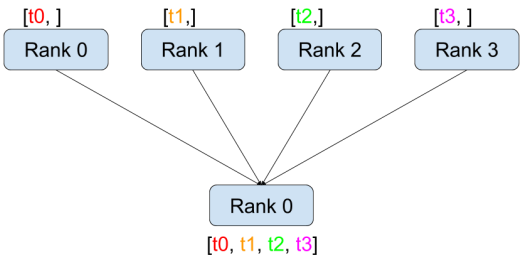
- writing to `tensor` after `dist.isend()` will result in undefined behaviour.
- reading from `tensor` after `dist.irecv()` will result in undefined behaviour.

However, after `req.wait()` has been executed we are guaranteed that the communication took place.

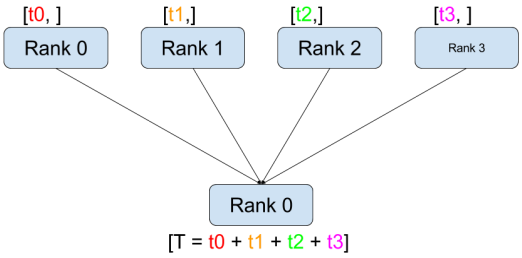
一对多、多对一通讯



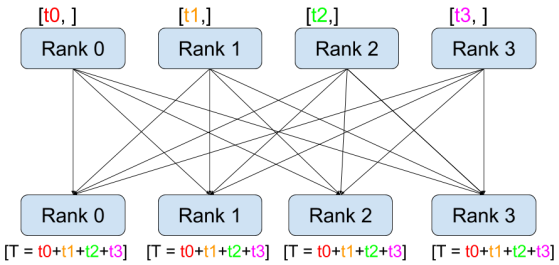
Scatter (在Rank 0上将t0, t1, t2, t3分发到进程Rank0-3上)



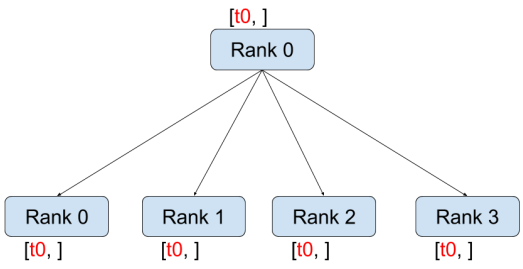
Gather (分别从Rank0-3进程中拷贝t0-3到rank0进程中)



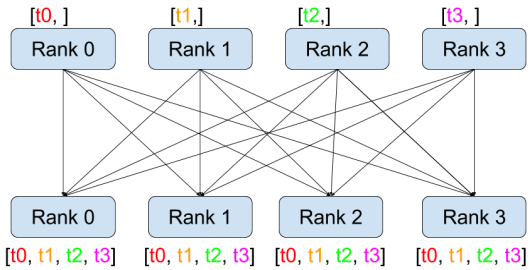
Reduce



All Reduce



Broadcast



All Broadcast

Reduce

```
def run(rank, size):
    # create a group with all processors
    group = dist.new_group(list(range(size)))
    tensor = torch.ones(1)
    # sending all tensors to rank 0 and sum them
    dist.reduce(tensor, dst=0, op=dist.ReduceOp.SUM, group=group)
    # can be dist.ReduceOp.PRODUCT, dist.ReduceOp.MAX, dist.ReduceOp.MIN
    # only rank 0 will have four
    print(f"[{rank}] data = {tensor[0]}")

Output:
[3] data = 1.0
[2] data = 2.0
[1] data = 3.0
[0] data = 4.0
```

这里的输出值得注意，他实际运行流程是每个进程把该进程的tensor交给任务调度器，然后任务调度器将该tensor和**中间变量累加**得到一个结果，并把这个结果返回给当前进程，当所有进程的通讯完成之后会把最终结果返回给dst=0进程。

All Reduce

```
def run(rank: int, size: int):
    # create a group with all processors
    group = dist.new_group(list(range(size)))
    tensor = torch.ones(1)
    dist.all_reduce(tensor, op=dist.ReduceOp.SUM, group=group)
    # can be dist.ReduceOp.PRODUCT, dist.ReduceOp.MAX, dist.ReduceOp.MIN
    # will output 4 for all ranks
    print(f"[{rank}] data = {tensor[0]}")

Output:
[3] data = 4.0
[0] data = 4.0
[1] data = 4.0
[2] data = 4.0
```

Scatter

```
def do_scatter(rank: int, size: int):
    # create a group with all processors
    group = dist.new_group(list(range(size)))
    tensor = torch.empty(1)
    # sending all tensors from rank 0 to the others
    if rank == 0:
        tensor_list = [torch.tensor([i + 1], dtype=torch.float32) for i in range(size)]
        # tensor_list = [tensor(1), tensor(2), tensor(3), tensor(4)]
        dist.scatter(tensor, scatter_list=tensor_list, src=0, group=group)
    else:
        dist.scatter(tensor, scatter_list=[], src=0, group=group)
```

```
    # each rank will have a tensor with their rank number
    print(f"[{rank}] data = {tensor[0]}")

Outputs:
[2] data = 3.0
[1] data = 2.0
[3] data = 4.0
[0] data = 1.0
```

Gather

对于gather, 首先需要在master 进程新建一个空的list来存储tensor，如果有4个节点则list长度为4，分别存储rank 0, 1, 2, 3 进程的这个变量的值。接下来，dist.gather()第一个参数指明了需要获取的每个进程的具体变量名。而slave node只需要将tensor传出即可，不需要新建list存储tensor。

```
# master node (从其他进程中获取数据，获取的数据会被存储到var_list中，
# 同时进程自身也会参与到gather中将var发送给调度系统)
var_list = [torch.zeros_like(var) for _ in range(4)]
dist.gather(var, var_list, dst=0, group=group, async_op=False)
# slave node （将进程的私有数据发送给调度系统，让master进程能获取该数据）
dist.gather(var, dst=0, group=group, async_op=False)
```

All Gather

```
def do_all_gather(rank: int, size: int):
    # create a group with all processors
    group = dist.new_group(list(range(size)))
    tensor = torch.tensor([rank], dtype=torch.float32)
    # create an empty list we will use to hold the gathered values
    tensor_list = [torch.empty(1) for i in range(size)]
    # sending all tensors to the others
    dist.all_gather(tensor_list, tensor, group=group)
    # all ranks will have [tensor([0.]), tensor([1.]), tensor([2.]), tensor([3.])]
    print(f"[{rank}] data = {tensor_list}")

Outputs:
[0] data = [tensor([0.]), tensor([1.]), tensor([2.]), tensor([3.])]
[2] data = [tensor([0.]), tensor([1.]), tensor([2.]), tensor([3.])]
[3] data = [tensor([0.]), tensor([1.]), tensor([2.]), tensor([3.])]
[1] data = [tensor([0.]), tensor([1.]), tensor([2.]), tensor([3.])]
```

3D并行机制实现

启动分布式训练

这节主要参考**Multi Node PyTorch Distributed Training Guide For People In A Hurry这一blog。**

Multiprocessing (Spawn)

首先，每个进程都需要进行分布式训练初始化，告诉系统『我是并行训练体系中的一员，我是哪个分布式任务中的进程，我在这个任务中的id是什么，我和其他进程之间的通讯协议是什么』，如下面给出的示例代码：

```
"""run.py:"""
#!/usr/bin/env python
import os
import torch
import torch.distributed as dist
import torch.multiprocessing as mp

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])

def init_process(rank, size, fn, backend='gloo'):
    """ 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 = []
    mp.set_start_method("spawn")
    for rank in range(size):
        p = mp.Process(target=init_process, args=(rank, size, run))
        p.start()
        processes.append(p)

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

这里调用torch提供的python多进程的封装来启动分布式训练（torch还提供了一些其他更便捷、功能更全面的封城来启动分布式训练），init_process先告诉系统『我是并行训练体系中的一员，我是哪个分布式任务中的进程，我在这个任务中的id是什么，我和其他进程之间的通讯协议是什么』，然后进入每个进程的入口程序run。在run程序中，根据进程的id运行不同代码模块。

这里的**MASTER_ADDR** 和**MASTER_PORT**唯一地确定一个分布式任务，拥有相同ADDR和PORT的进程属于同一个job。

上面这段代码只能在同一个节点（也即同一台机器）上运行分布式任务，如果要在不同的节点上运行分布式任务，我们需要：

1. 修改**MASTER_ADDR为某个节点其他节点可见的ip**
2. 全局化一个进程管理器，自动分配进程id

为了实现这一目的，pytorch引入了torch.distributed.launch 和torchrun

torch.distributed.launch

torch.distributed.launch提供了进程管理脚本，它负责协调不同进程，为每个进程设置id，master_addr 和 master_port，计算每个进程在所在节点的local rank，监视每个进程的运行状态；因此我们可以进一步简化代码：

```
"""run.py:"""
#!/usr/bin/env python
import os
import torch
import torch.distributed as dist

LOCAL_RANK = int(os.environ['LOCAL_RANK'])
WORLD_SIZE = int(os.environ['WORLD_SIZE'])
WORLD_RANK = int(os.environ['RANK'])

def run(backend):
    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])

def init_process(backend='gloo'):
    """ Initialize the distributed environment. """
    dist.init_process_group(backend, rank=WORLD_RANK, world_size=WORLD_SIZE)
    run(backend)


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    # 如果用torchrun的话这个是不需要的（加入这个会在当前进程中添加LOCAL_RANK环境）
    parser.add_argument("--local_rank", type=int, help="Local rank. Necessary for using the torch.distributed.launch utility.")
    parser.add_argument("--backend", type=str, default="nccl", choices=['nccl', 'gloo'])
    args = parser.parse_args()
    init_process(backen=args.backend)
```

然后我将将这份代码拷贝到**所有要参与分布式计算的机器上**（下面的示例考虑在两台机器上进行分布式计算），然后运行启动脚本。对于数据，一个比较推荐的做法是将数据存储在一个数据服务器上，每个进程都去这台数据服务器上获取数据，而不是将数据拷贝到每台参与计算的机器上。

```
'''启动脚本'''
# 在104.171.200.62 节点的terminal上运行下面这段命令(the master node)
python3 -m torch.distributed.launch \
--nproc_per_node=2 --nnodes=2 --node_rank=0 \
--master_addr=104.171.200.62 --master_port=1234 \
main.py \
--backend=gloo --use_syn

# 在104.171.200.182 节点的terminal上运行下面这段命令 (the worker node)
python3 -m torch.distributed.launch \
--nproc_per_node=2 --nnodes=2 --node_rank=1 \
--master_addr=104.171.200.62 --master_port=1234 \
main.py \
--backend=gloo --use_syn
```

注意，torch.distruted.launch根据nnodes来判断运行节点的个数，它会等到指定个数的节点（更具体的是进程）被运行起来之后才启动分布式计算（这里的use_syn就是指示调度中心要等所有的进程都起来之后才开始分布式计算的）。

launch会根据 world_size = nproc_per_node * nnodes 来设定world_size，根据node_rank 和nproc_per_node 来设定每个进程的rank，local_rank等环境变量；自动配MASTER_ADDR 和MASTER_PORT环境变量。

Torchrun

Created by the PyTorch team, `torchrun` works similarly to `torch.distributed.launch` but with some extra functionalities that gracefully handle failed workers and elasticity. In fact, `torchrun` can work with the exact same script as `torch.distributed.launch` does.

启动命令和 `torch.distributed.launch` 相同

数据并行

这节说明主要参考这篇pytorch官方说明文档

首先定义数据切分代码（所有进程的代码是同一份，只是在运行的过程中根据进程id运行不同的代码模块）

```
""" 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])
```

获取当前进程应该使用的数据

```
""" 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() # 获取整个分布式任务的进程数
    bsz = 128 / float(size) # 设置当前进程的batch size
    partition_sizes = [1.0 / size for _ in range(size)] # 每个进程均等切分数据
    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 # 获取当前进程的训练数据
```

定义进程入口程序

```
""" Distributed Synchronous SGD Example """
def run(rank, size):
    torch.manual_seed(1234)
    train_set, bsz = partition_dataset()
    model = Net() # 注意：DDP中，不同进程使用的模型是一样的
    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:
            optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            epoch_loss += loss.item()
            loss.backward()
            # 归并其他进程中的梯度以更新模型参数
```

```
            average_gradients(model)
            optimizer.step()
        print('Rank ', dist.get_rank(), ', epoch ',
              epoch, ': ', epoch_loss / num_batches)
```

定义梯度归并函数

```
""" 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.ReduceOp.SUM)
        param.grad.data /= size
```

这里使用的是all_reduce API来归并模型梯度。

以上代码可以看作是DistributedDataParallel+DistributedSampler的自我实现，当然，DistributedDataParallel经过了优化和反复测试，稳定性和效率会更高。

张量并行

这节主要参考Megatron的VocabParallelEmbedding class的实现

```
class VocabParallelEmbedding(torch.nn.Module):
    """Embedding parallelized in the vocabulary dimension.

    This is mainly adapted from torch.nn.Embedding and all the default
    values are kept.
    Arguments:
        num_embeddings: vocabulary size.
        embedding_dim: size of hidden state.
        init_method: method to initialize weights.
    """

    def __init__(self, num_embeddings, embedding_dim,
                 init_method=init.xavier_normal_):
        super(VocabParallelEmbedding, self).__init__()
        # Keep the input dimensions.
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim
        # Set the detauls for compatibility.
        self.padding_idx = None
        self.max_norm = None
        self.norm_type = 2.
        self.scale_grad_by_freq = False
        self.sparse = False
        self._weight = None
        self.model_parallel_size = get_model_parallel_world_size()
        # Divide the weight matrix along the vocaburaly dimension.
        self.vocab_start_index, self.vocab_end_index = \
            VocabUtility.vocab_range_from_global_vocab_size(
                self.num_embeddings, get_model_parallel_rank(),
                self.model_parallel_size)
        self.num_embeddings_per_partition = self.vocab_end_index - \
            self.vocab_start_index

        # Allocate weights and initialize.
        args = get_args()
        if args.use_cpu_initialization:
            self.weight = Parameter(torch.empty(
                self.num_embeddings_per_partition, self.embedding_dim,
                dtype=args.params_dtype))
            _initialize_affine_weight_cpu(
                self.weight, self.num_embeddings, self.embedding_dim,
                self.num_embeddings_per_partition, 0, init_method)
        else:
```

```
            self.weight = Parameter(torch.empty(
                self.num_embeddings_per_partition, self.embedding_dim,
                device=torch.cuda.current_device(), dtype=args.params_dtype))
            _initialize_affine_weight_gpu(self.weight, init_method,
                                          partition_dim=0, stride=1)

    def forward(self, input_):
        if self.model_parallel_size > 1:
            # Build the mask.
            input_mask = (input_ < self.vocab_start_index) | \
                         (input_ >= self.vocab_end_index)
            # Mask the input.
            masked_input = input_.clone() - self.vocab_start_index
            masked_input[input_mask] = 0
        else:
            masked_input = input_
            # Get the embeddings.
        output_parallel = F.embedding(masked_input, self.weight,
                                      self.padding_idx, self.max_norm,
                                      self.norm_type, self.scale_grad_by_freq,
                                      self.sparse)
        # Mask the output embedding.
        if self.model_parallel_size > 1:
            output_parallel[input_mask, :] = 0.0
        # Reduce across all the model parallel GPUs. all_reduce
        output = reduce_from_model_parallel_region(output_parallel)
        return output
```

在数据并行中，我们是通过对数据进行切片，然后让不同的进程使用不同的数据切片来实现数据并行；而在张量并行中，我们是通过对张量进行切片，不同的进程创建、存储不同的张量切片，并在该切片的基础上进行前向后向计算，并最终归并不同进程上的计算结果来实现张量并行。

例如上面的例子是对word embedding 矩阵进行切片，每个进程只创建一个切片对应的参数。在forward过程中，如果输入的input_id在当前切片中则用当前切片对该input_id进行赋值，否则将该input_id在当前进程中的表示置零。最后用all_reduce实现所有input_id的向量化。

流水线并行

这节主要参考torch的RPC （Remote Procedure Call) 说明文档