PyTorch 分布式(1) --- 数据加载之 DistributedSample

0x01 数据加载

1.1 加速途径

当分布式训练时候，为了加速训练，有三个层面的工作需要处理。

- 数据加载层面
- 多机通讯层面
- 代码层面

在**数据层面**，可以使用多进程并行加载来加速数据预处理过程，也有利用GPU特点来加速，比如Nvidia DALI 通过将数据预处理放到 GPU 处理来解决 CPU 瓶颈问题。

在**多机通讯层面**，有各种集合通信库可以利用，比如NCCL，OpenMPI, Gloo 等。

在**代码层面**，可以使用框架提供的分布式API，或者利用 Horovod 来改造单机版代码，使其支持分布式任务。

接下来我们就看看数据层面如何加速。

1.2 并行处理

AI框架的数据处理主要如下并行处理：

- 数据加载/处理使用CPU。
- 训练使用GPU。

在理想状态下，应该是每轮迭代训练之前，CPU就完成加载，准备好训练数据，这样训练就可以持续无缝迭代。

然而，GPU算力每年会提升一倍，CPU的提升速度远远落后于GPU，所以CPU会是拖后腿的那个角色。这里不仅仅是CPU算力不足的问题，也包括村存储中读取数据速度不足的问题。

因此，机器学习对于数据加载和前期预处理的要求越来越高，必须在GPU计算时间内，完成下一迭代数据的准备工作，不能让GPU因为等待训练数据而空闲。

1.3 流水线

对于机器学习训练，加载数据可以分为三个步骤：

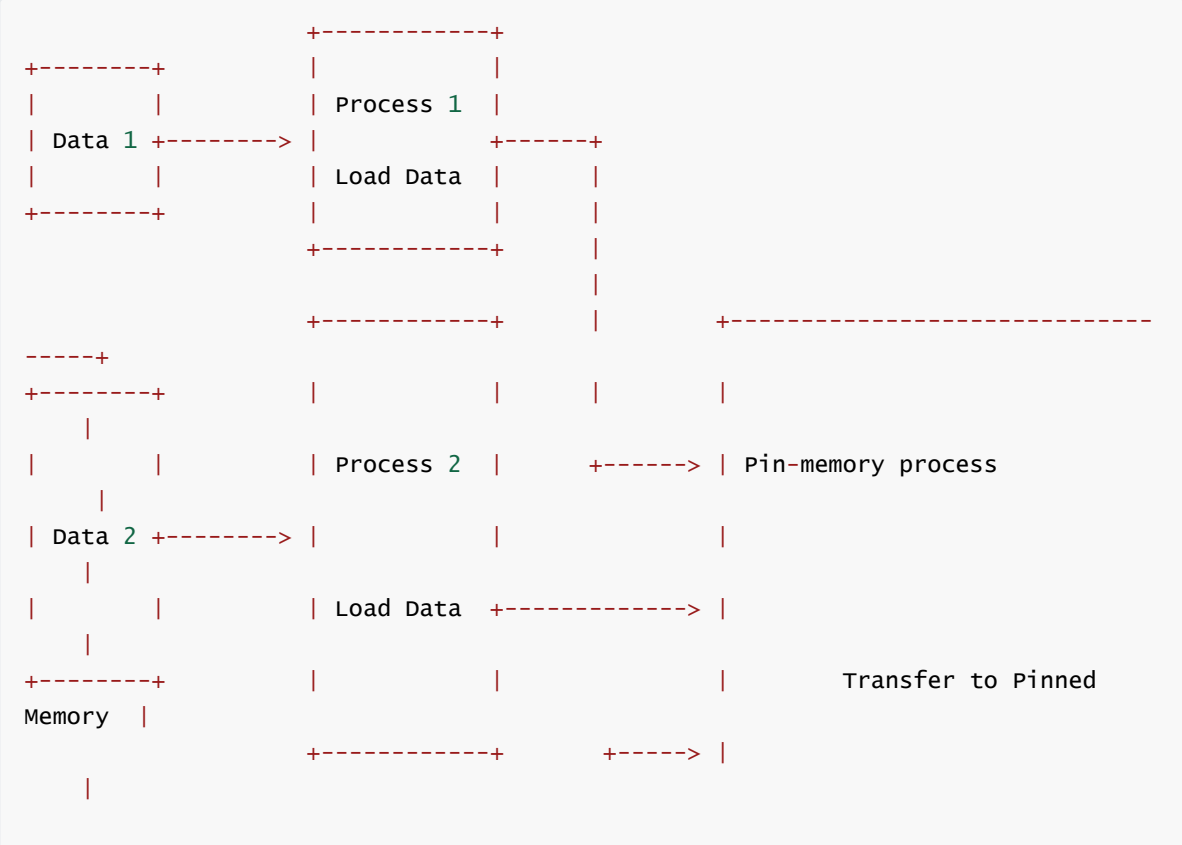
- 将数据从磁盘或者分布式存储加载到主机（CPU）。
- 将数据从主机可分页内存传输到主机固定内存。
- 将数据从主机固定内存转移到主机GPU。

因此，流行的深度学习框架会依据加载步骤的特点和异构硬件的特点来进行流水线处理，从而提高数据处理过程的吞吐量。

流水线一般包括多个算子，每个算子内部由数据队列组成一个缓冲区，上游算子完成处理之后会传给给下游算子进行处理。这样每个算子任务会彼此独立，算子内部可以使用细粒度的多线程/多进程来并行加速，每个算子可以独立控制处理速度和内存以适配不同网络对于处理速度的需求。

如果算子内部数据队列不为空，模型就会一直源源不断获得数据，就不会因为等待训练数据而产生瓶颈。

下面是并行流水线逻辑：



```
                                          |       +-----------------------------
-----+
+--------+          +------------+       |
|        |          |            |       |
| Data 3 +--------> | Process 3  +-------+
|        |          |            |
+--------+          | Load Data  |
                    +------------+
```

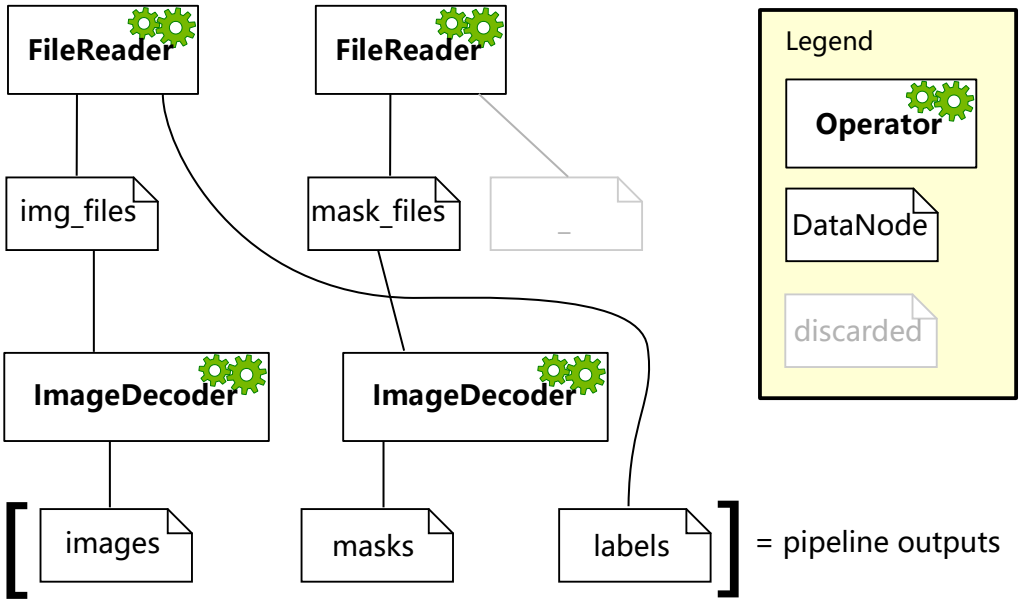
1.4 GPU

本文到现在是解决CPU侧的数据传输问题，即：从磁盘加载数据，从可分页到固定内存。

但是，从固定内存到GPU的数据传输（`tensor.cuda()`）也可以使用CUDA流进行流水线处理。

另外，深度学习应用程序需要复杂的多阶段数据处理管道，包括加载、解码、裁剪、调整大小和许多其他增强功能。这些目前在 CPU 上执行的数据处理管道已经成为瓶颈，限制了训练和推理的性能和可扩展性。

Nvidia DALI 通过将数据预处理放到 GPU 处理来解决 CPU 瓶颈问题，用户可以依据自己模型的特点，构建基于 GPU 的 pipeline，或者基于CPU的pipeline。



接下来我们就介绍PyTorch的数据加载，而且主要是从分布式的角度进行切入。

0x02 PyTorch分布式加载

2.1 DDP

pytorch为数据分布式训练提供了多种选择。随着应用从简单到复杂，从原型到产品，常见的开发轨迹可以是：

- 如果数据和模型能放入单个GPU，使用单设备训练，此时不用担心训练速度；
- 如果服务器上有多个GPU，并且你在代码修改量最小的情况下加速训练，使用单个机器多GPU DataParallel；
- 如果你想进一步加速训练并且愿意写一点代码来启动，使用单个机器多个GPU DistributedDataParallel；
- 如果应用程序跨机器边界扩展，使用多机器DistributedDataParallel和启动脚本；

- 如果预期有错误（比如OOM）或者资源在训练过程中可以动态连接和分离，使用torchelastic来启动分布式训练。

与本文最相关的部分就是DDP，Distributed Data-Parallel Training（DDP）是一个广泛采用的单程序多数据训练方法。使用DDP，模型会被复制到每个进程，然后每个模型副本会被输入数据样本的不同子集。DDP负责梯度通信以保持模型副本的同步，并将其与梯度计算重叠以加快训练速度。

2.2 分布式加载

我们首先要看看分布式加载的总体结构。

给出示例代码，可以看到主要使用了 DataSet, DistributedSampler，DataLoader 这三个实体。

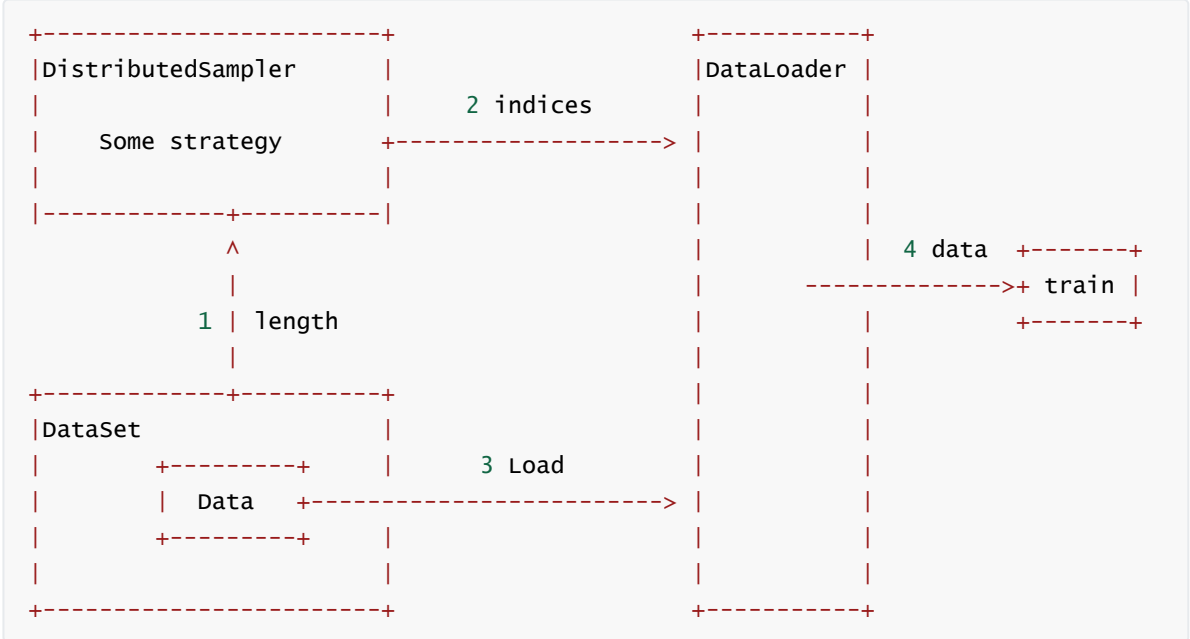
```
sampler = DistributedSampler(dataset) if is_distributed else None
loader = DataLoader(dataset, shuffle=(sampler is None), sampler=sampler)
for epoch in range(start_epoch, n_epochs):
    if is_distributed:
        sampler.set_epoch(epoch)
        train(loader)
```

这三个概念的逻辑关系如下：

- **Dataset** : 从名字可以知道，是数据集的意思。负责对原始训练数据的封装，将其封装成 Python 可识别的数据结构，Dataset的派生类必须提供接口一边获取单个数据。
- **Sampler** : 从名字可知，是采样器，负责采样方式或者说是采样策略，实现某种提取/采样策略从Dataset之中拿到数据索引，供DataLoade使用。可以认为，Sampler 是指挥者，负责决定战斗在哪里开展。
- **DataLoader** : 负责依据索引来从数据集中加载数据。支持 Map-style 和 Iterable-style 两种 Dataset，支持单进程/多进程加载。Loader 就是具体作战的斗士，负责按照 Sampler的命令进行战斗。

具体如下图，简要说就是：

1. DataSet 把数据集数目发给DistributedSampler。
2. Sampler 按照某种规则发送数据indices给Loader。
3. Loader 依据indices加载数据。
4. Loader 把数据发给模型，进行训练。



因为数据集不是分布式训练重点，所以本文接下来主要分析 Sampler。

Sampler 的重点就是：如何让每个worker在数据集中只加载自己所属的部分，并且worker之间实现对数据集的正交分配。

0x03 DistributedSampler

对于数据并行和分布式训练，DistributedSampler 负责其数据采样的任务。

DistributedSampler 是 Sampler 的派生类。当 DistributedDataParallel 使用DistributedSampler 时，每个并行的进程都会得到一个DistributedSampler 实例，这个DistributedSampler 实例会给DataLoader发送指示，从而 DataLoader 加载具体数据。

DistributedSampler 加载策略负责只提供加载数据集中的一个子集，这些DistributedSampler 提供的子集之间不重叠，不交叉。

3.1 初始化

`__init__` 初始化代码主要是设置了本worker节点的各种信息，比如数据集dataset，rank（全局GPU序号），num_replicas 副本数目。并且计算出来所有样本数目total_size。

几个参数如下：

- dataset： 就是采样的数据集。
- num_replicas： 参与分布式训练的进程数目，如果没有设置，则从group之中得到world_size作为进程数目。
- rank : 当前进程的序号，如果没有设置，则从group之中得到。
- shuffle： 采样是否需要打乱indices。
- seed： 如果需要打乱，则设定一个random seed。
- drop_last： 如果不能均匀分割数据，是否需要把无法分配的尾部数据丢掉。
- epoch： 每次epoch都会shuffle数据集，如何保持shuffle之后数据集一致性？就是通过epoch完成。

具体代码如下，省略了异常处理。

```
class DistributedSampler(Sampler[T_co]):

    def __init__(self, dataset: Dataset, num_replicas: Optional[int] = None,
                 rank: Optional[int] = None, shuffle: bool = True,
                 seed: int = 0, drop_last: bool = False) -> None:

        self.dataset = dataset
        self.num_replicas = num_replicas
        self.rank = rank
        self.epoch = 0
        self.drop_last = drop_last
        # If the dataset length is evenly divisible by # of replicas, then there
        # is no need to drop any data, since the dataset will be split equally.
        if self.drop_last and len(self.dataset) % self.num_replicas != 0:  #
type: ignore[arg-type]
            # Split to nearest available length that is evenly divisible.
            # This is to ensure each rank receives the same amount of data when
            # using this Sampler.
            self.num_samples = math.ceil(
                # `type:ignore` is required because Dataset cannot provide a
default __len__
                # see NOTE in pytorch/torch/utils/data/sampler.py
```

```
                (len(self.dataset) - self.num_replicas) / self.num_replicas  #
type: ignore[arg-type]
            )
        else:
            self.num_samples = math.ceil(len(self.dataset) / self.num_replicas)
 # type: ignore[arg-type]
        self.total_size = self.num_samples * self.num_replicas
        self.shuffle = shuffle
        self.seed = seed
```

3.2 迭代方法

DistributedSampler 被实现成一个迭代器（类似于循环），因此会用到 python 抽象类的魔法方法：

- `__len__(self)`：当被 `len()` 函数调用时的行为，一般返回迭代器中元素的个数。
- `__iter__(self)`：当迭代容器中元素时的行为，实际上是返回是一个迭代器（通常是迭代器本身），每一次迭代得到的结果会被用来作为下一次迭代的初始值。

`__iter__` 代码的一个技术细节是：

```
indices = indices[self.rank:self.total_size:self.num_replicas]
```

当一个list之中有双引号，比如 `list[start:end:step]`，其意义是：

- start: 起始位置
- end: 结束位置
- step: 步长

我们用一个例子来看看，比如：

```
a = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
print(a[0:15:3])
print(a[1:15:3])
print(a[2:15:3])
```

得到：

```
[1, 4, 7, 10, 13]
[2, 5, 8, 11, 14]
[3, 6, 9, 12, 15]
```

因为 `indices[self.rank:self.total_size:self.num_replicas]` 之中，num_replicas 实际就是rank的总数，所以这里每个worker就会严格返回自己rank对应的那部分数据序号。

总结一下DistributedSampler的分配方法是：每段连续的 `num_replicas` 个数据被拆成一个一个，分给 `num_replicas` 个进程，而且是通过每个worker 的 rank 来获取数据，这样就达到了不重叠不交叉的目的，但也要注意的是：这样每个进程拿到的数据是不连续的。

具体代码如下：

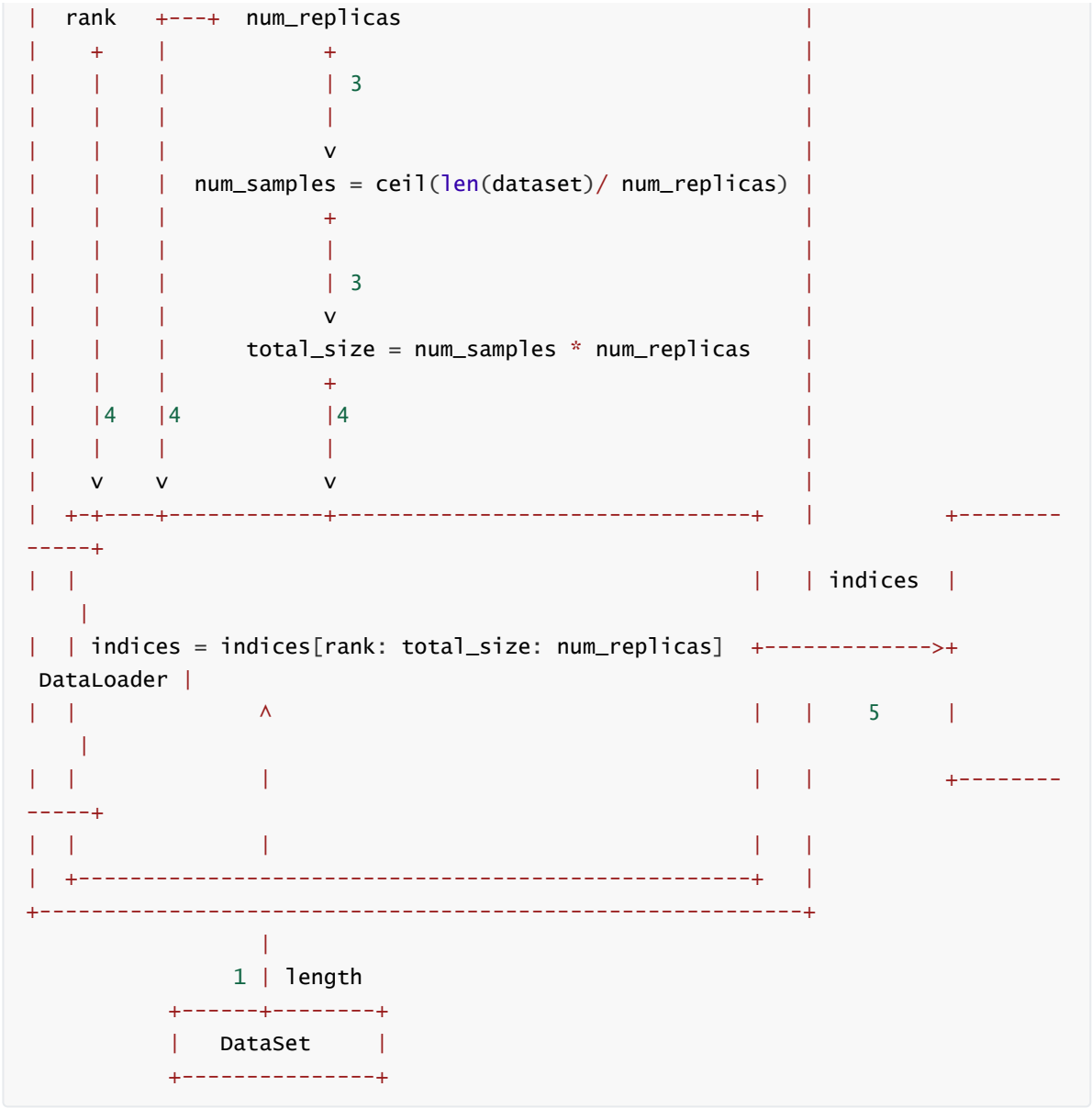
```
class DistributedSampler(Sampler[T_co]):

    def __iter__(self) -> Iterator[T_co]:

        if self.shuffle: # 如果需要shuffle，则会基于epoch和seed进行处理
```

```
            # deterministically shuffle based on epoch and seed
            g = torch.Generator()
            g.manual_seed(self.seed + self.epoch)
            indices = torch.randperm(len(self.dataset), generator=g).tolist()  #
type: ignore[arg-type]
        else: # 否则直接返回数据集长度序列
            indices = list(range(len(self.dataset)))  # type: ignore[arg-type]

        # 是否需要补齐数据
        if not self.drop_last:
            # add extra samples to make it evenly divisible
            padding_size = self.total_size - len(indices)
            if padding_size <= len(indices):
                indices += indices[:padding_size]
            else:
                indices += (indices * math.ceil(padding_size / len(indices)))
[:padding_size]
        else:
            # remove tail of data to make it evenly divisible.
            indices = indices[:self.total_size]
        assert len(indices) == self.total_size

        # subsample
        # 依据自己的rank，依次返回自己的数据序号
        indices = indices[self.rank:self.total_size:self.num_replicas]
        assert len(indices) == self.num_samples

        return iter(indices)

    def __len__(self) -> int:
        return self.num_samples

    def set_epoch(self, epoch: int) -> None:
        r"""
        Sets the epoch for this sampler. When :attr:`shuffle=True`, this ensures
all replicas
        use a different random ordering for each epoch. Otherwise, the next
iteration of this
        sampler will yield the same ordering.

        Args:
            epoch (int): Epoch number.
        """
        self.epoch = epoch
```

内部变量之间逻辑如下：

1. 从数据集获取长度length；
2. 从配置得到 num_replicas（有几个rank），本身rank；
3. 依据 数据集长度 和 num_replicas得到 num_samples 和 total_size；
4. 最终给出 indices = indices[rank: total_size: num_replicas]；
5. 返回 indices 给DataLoader

```
+-----------------------------------------------------------+
| DistributedSampler                                        |
|                                                           |
|    2                2                                     |
```



3.3 shuffle数据集

每次epoch都会shuffle数据集，但是**不同进程如何保持shuffle之后数据集一致性**?

DistributedSampler 使用当前的epoch作为随机数种子，在计算index之前就进行配置，从而保证不同进程都使用同样的随机数种子，这样shuffle出来的数据就能确保一致。

3.3.1 使用

从下面代码可以看出来，如果需要分布式训练，就对 sampler 设置 epoch。

```
sampler = DistributedSampler(dataset) if is_distributed else None
loader = DataLoader(dataset, shuffle=(sampler is None), ...,
                            sampler=sampler)
    for epoch in range(start_epoch, n_epochs):
        if is_distributed:
            sampler.set_epoch(epoch) # 这设置epoch
        train(loader)
```

3.3.2 python

具体对应 DistributedSampler 的实现。

设置 epoch 很简单，就是配置下。

```
    def set_epoch(self, epoch: int) -> None:
        r"""
        Sets the epoch for this sampler. When :attr:`shuffle=True`, this ensures
all replicas
        use a different random ordering for each epoch. Otherwise, the next
iteration of this
        sampler will yield the same ordering.

        Args:
            epoch (int): Epoch number.
        """
        self.epoch = epoch
```

设置 random 种子的具体使用是在迭代函数之中：

```
    def __iter__(self) -> Iterator[T_co]:
        if self.shuffle:
            # deterministically shuffle based on epoch and seed
            g = torch.Generator()
            g.manual_seed(self.seed + self.epoch) # 这里设置随机种子
            indices = torch.randperm(len(self.dataset), generator=g).tolist()  #
type: ignore[arg-type]
        else:
            indices = list(range(len(self.dataset)))  # type: ignore[arg-type]

        # 省略其他代码
```

3.3.3 C++

我们也可以提前看看在C++ 代码的DistributedRandomSampler，这是C++ API，也起到python同样作用。

我们可以看到设置种子和shuffle如下：

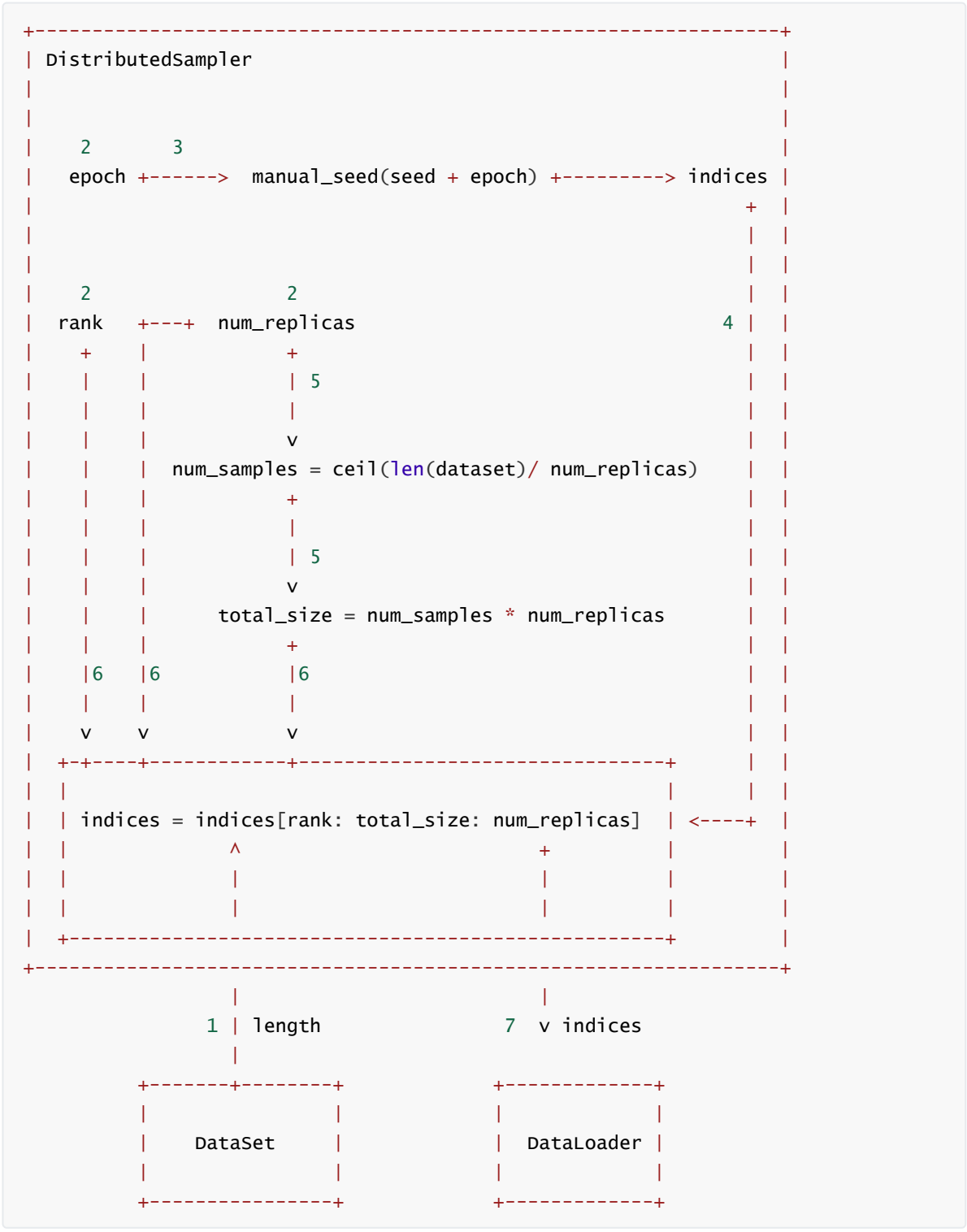
```
void DistributedRandomSampler::reset(optional<size_t> new_size) {
  size_ = new_size.value_or(size_);
  populate_indices();

  std::mt19937 rand(epoch_);
  std::shuffle(all_indices_.begin(), all_indices_.end(), rand);
  sample_index_ = begin_index_;
}
```

3.3.4 小结

我们扩展目前逻辑如下：

1. 从数据集获取长度length；
2. 从配置得到 num_replicas（有几个rank），本身rank，epoch；
3. 用 epoch 来设置random seed；

4. 利用random seed 对数据集 indices 进行打乱，后续就会一直使用 这个打乱的indices；
5. 依据 数据集长度 和 num_replicas得到 num_samples 和 total_size；
6. 结合之上各种数据条件，最终给出 indices = indices[rank: total_size: num_replicas]；
7. 返回 indices 给DataLoader



3.4 Sampler in C++

因为某些公司是C++开发，他们也有迫切的使用pytorch的需求，所以pytorch也提供了C++ API，我们接下来就看看如何实现。

3.4.1 定义

其类定义在：torch\csrc\api\include\torch\data\samplers\distributed.h

我们可以看到，DistributedSampler 是基类，主要成员变量是：

- size_t size_ 文件大小
- size_t num_replicas_ 副本数目
- size_t rank_ 本sampler 对应哪个进程或者GPU
- size_t epoch 本次训练的epoch
- bool allow_duplicates_ 是否允许备份

接下来是两个派生类： DistributedRandomSampler 和 DistributedSequentialSampler。

```
/// A `Sampler` that selects a subset of indices to sample from and defines a
/// sampling behavior. In a distributed setting, this selects a subset of the
/// indices depending on the provided num_replicas and rank parameters. The
/// `Sampler` performs a rounding operation based on the `allow_duplicates`
/// parameter to decide the local sample count.
template <typename BatchRequest = std::vector<size_t>>
class DistributedSampler : public Sampler<BatchRequest> {
 public:
  DistributedSampler(
      size_t size,
      size_t num_replicas = 1,
      size_t rank = 0,
      bool allow_duplicates = true)
      : size_(size),
        num_replicas_(num_replicas),
        rank_(rank),
        epoch_(0),
        allow_duplicates_(allow_duplicates) {}

  /// Set the epoch for the current enumeration. This can be used to alter the
  /// sample selection and shuffling behavior.
  void set_epoch(size_t epoch) {
    epoch_ = epoch;
  }

  size_t epoch() const {
    return epoch_;
  }

 protected:
  size_t local_sample_count() {
    if (allow_duplicates_) {
      return (size_ + num_replicas_ - 1) / num_replicas_;
    } else {
      return size_ / num_replicas_;
    }
  }

  size_t size_;
  size_t num_replicas_;
  size_t rank_;
  size_t epoch_;
  bool allow_duplicates_;
};
```

```
/// Select samples randomly. The sampling order is shuffled at each `reset()`
/// call.
class TORCH_API DistributedRandomSampler : public DistributedSampler<> {
 public:
  DistributedRandomSampler(
      size_t size,
      size_t num_replicas = 1,
      size_t rank = 0,
      bool allow_duplicates = true);

  /// Resets the `DistributedRandomSampler` to a new set of indices.
  void reset(optional<size_t> new_size = nullopt) override;

  /// Returns the next batch of indices.
  optional<std::vector<size_t>> next(size_t batch_size) override;

  /// Serializes the `DistributedRandomSampler` to the `archive`.
  void save(serialize::OutputArchive& archive) const override;

  /// Deserializes the `DistributedRandomSampler` from the `archive`.
  void load(serialize::InputArchive& archive) override;

  /// Returns the current index of the `DistributedRandomSampler`.
  size_t index() const noexcept;

 private:
  void populate_indices();

  size_t begin_index_;
  size_t end_index_;
  size_t sample_index_;
  std::vector<size_t> all_indices_;
};

/// Select samples sequentially.
class TORCH_API DistributedSequentialSampler : public DistributedSampler<> {
 public:
  DistributedSequentialSampler(
      size_t size,
      size_t num_replicas = 1,
      size_t rank = 0,
      bool allow_duplicates = true);

  /// Resets the `DistributedSequentialSampler` to a new set of indices.
  void reset(optional<size_t> new_size = nullopt) override;

  /// Returns the next batch of indices.
  optional<std::vector<size_t>> next(size_t batch_size) override;

  /// Serializes the `DistributedSequentialSampler` to the `archive`.
  void save(serialize::OutputArchive& archive) const override;

  /// Deserializes the `DistributedSequentialSampler` from the `archive`.
  void load(serialize::InputArchive& archive) override;

  /// Returns the current index of the `DistributedSequentialSampler`.
```

```
  size_t index() const noexcept;

 private:
  void populate_indices();

  size_t begin_index_;
  size_t end_index_;
  size_t sample_index_;
  std::vector<size_t> all_indices_;
};
```

3.4.2 实现

类的具体实现位于：torch\csrc\api\src\data\samplers\distributed.cpp

3.4.2.1 DistributedRandomSampler

我们首先看看DistributedRandomSampler。

其作用就是依据本worker 的 rank_获取打乱的index。我们按照逻辑顺序讲解各个函数。

- 初始化时候会调用 reset(size_) 进行 shuffle。
- reset 函数作用是让sampler指向一组新的indices：
  - 首先调用 populate_indices() 设置对应本rank的起始index，终止index。
  - populate_indices 函数之中，会对数据index 进行设置，即配置了 all_indices_，也同时配置了本rank的起始index，终止index。
  - 然后对 all_indices_ 进行shuffle。
- next 函数就相对简单了，因为主要工作被reset做了，所以此时数据已经被随机打乱了，所以找到起始位置，返回数据中对行数即可。

因为下面用到了 iota 函数，可能有的同学不熟悉，这里说明下iota的作用：

```
std::vector<int> test;
test.resize(10);
std::iota(test.begin(), test.end(), 5);// 将从 5 开始的 10 次递增值赋值给 test

//test结果:5 6 7 8 9 10 11 12 13 14
```

具体代码如下：

```
DistributedRandomSampler::DistributedRandomSampler(
    size_t size,
    size_t num_replicas,
    size_t rank,
    bool allow_duplicates)
    : DistributedSampler(size, num_replicas, rank, allow_duplicates),
      begin_index_(0),
      end_index_(0),
      sample_index_(0) {
  // shuffle first time.
  reset(size_);
}

// 每次加载新epoch时候，都要调用reset
void DistributedRandomSampler::reset(optional<size_t> new_size) {
  size_ = new_size.value_or(size_);
```

```
  populate_indices();

  std::mt19937 rand(epoch_);
  // 对于数据进行shuffle
  std::shuffle(all_indices_.begin(), all_indices_.end(), rand);
  sample_index_ = begin_index_;
}

void DistributedRandomSampler::populate_indices() {
  size_t num_local_samples = local_sample_count();
  // 得到样本数量
  size_t sample_count =
      num_replicas_ == 1 ? size_ : num_local_samples * num_replicas_;
  all_indices_.resize(sample_count);

  // std::iota 的作用是用顺序递增的值赋值指定范围内的元素
  // 这里是给all_indices_设置从0开始到sample_count这些数值
  std::iota(std::begin(all_indices_), std::end(all_indices_), 0);
  // 如果sample count大于size_，则需要给多出来的那些index再赋一些数值
  for (size_t i = size_; i < sample_count; ++i) {
    // we may have added duplicate samples to make all
    // replicas to have the same number of samples.
    all_indices_[i] = i - size_;
  }
  begin_index_ = rank_ * num_local_samples; // 对应本rank的起始index
  end_index_ = begin_index_ + num_local_samples; // 对应本rank的终止index
  sample_index_ = begin_index_;
}

size_t DistributedRandomSampler::index() const noexcept {
  return sample_index_;
}

// 注意，每次加载新epoch时候，都要调用reset，因此对于next函数来说，工作已经很小
optional<std::vector<size_t>> DistributedRandomSampler::next(
    size_t batch_size) {
  if (sample_index_ == end_index_) { // 已经提取完数据
    return nullopt;
  }

  size_t end = sample_index_ + batch_size; // 本次迭代的终止位置
  if (end > end_index_) {
    end = end_index_;
  }

  auto iter = all_indices_.begin(); // 因为此时数据已经被随机打乱了，找到起始位置即可
  std::vector<size_t> res(iter + sample_index_, iter + end); // 从所有数据中提取前面
若干行
  sample_index_ = end;
  return res;
}
```

3.4.2.2 DistributedSequentialSampler

然后看看 DistributedSequentialSampler。

其作用就是依据本worker 的 rank_获取顺序的index。我们按照逻辑顺序讲解各个函数。

- reset 函数就简单多了，使用populate_indices按照顺序设置index即可。
- next 函数就相对复杂，不但要顺序返回index，还需要设置下次的起始位置。

```
DistributedSequentialSampler::DistributedSequentialSampler(
    size_t size,
    size_t num_replicas,
    size_t rank,
    bool allow_duplicates)
    : DistributedSampler(size, num_replicas, rank, allow_duplicates),
      begin_index_(0),
      end_index_(0),
      sample_index_(0) {
  populate_indices(); // 这里会设定本rank对应的起始位置
}

void DistributedSequentialSampler::reset(optional<size_t> new_size) {
  size_t size = new_size.value_or(size_);
  if (size != size_) {
    size_ = size;
    populate_indices();
  } else {
    sample_index_ = begin_index_;
  }
}

void DistributedSequentialSampler::populate_indices() {
  begin_index_ = rank_ * local_sample_count(); // 本rank对应的起始位置
  end_index_ = begin_index_ + local_sample_count();
  sample_index_ = begin_index_;
}

size_t DistributedSequentialSampler::index() const noexcept {
  return sample_index_;
}

optional<std::vector<size_t>> DistributedSequentialSampler::next(
    size_t batch_size) {
  if (sample_index_ == end_index_) { // 已经循环结束
    return nullopt;
  }

  size_t end = sample_index_ + batch_size; // 本次的终止行
  if (end > end_index_) {
    end = end_index_;
  }

  std::vector<size_t> res(end - sample_index_); // 返回的vector大小
  // 给res设置从sample_index_开始递增(end - sample_index_)这么大的这些数值，这就是顺序返回了index
  std::iota(std::begin(res), std::end(res), sample_index_);
  if (end >= size_) {
    for (size_t& index : res) { //遍历 vector，得到本次的index
```

```
      index = index % size_;
    }
  }
  sample_index_ = end; // 设置下次开始行
  return res;
}
```