TORCH.UTILS.DATA

杨资璋（翻译）

2021 年 10 月 15 日

Pytorch 的数据加载工具核心是`torch.utils.data.DataLoader`类。它提供一个遍历数据集的 iterable，并支持

- 映射风格和迭代风格的数据集
- 自定义数据加载顺序
- 自动形成批
- 单进程多进程数据加载
- 自动内存固定

这些选项通过 DataLoader 的构建函数参数设置，它的签名是：

```
DataLoader(
    dataset,
    batch_size=1,
    shuffle=False,
    sampler=None,
    batch_sampler=None,
    num_worker=0,
    collate_fn=None,
    pin_memory=False,
    drop_last=False,
    timeout=0,
    worker_init_fn=None,
    *,
    prefetch_factor=2,
    persistent_workers=False
)
```

下面的章节详细描述这些选项的功能和用法。

1 数据集类型

DataLoader的构造函数中最重要的参数就是dataset，它表明了加载数据的来源。Pytorch 支持两种不同类型的数据集：

- 映射风格的数据集
- 迭代形式的数据集

1.1 映射形式的数据集

一个映射风格的数据集是实现了__getitem__()和__len__()协议的数据集，它提供一个从索引或者键到数据样本的映射（可能是不完整的）。

例如，这样的数据集，当使用dataset[idx]访问时，可以从磁盘上的文件夹读取第idx个图片和它对应的标签。

更多细节见Dataset。

1.2 迭代形式的数据集

一个迭代形式的数据集是实现了__iter__()协议的IterableDataset的自类的实例，并提供遍历数据样本的 iterable。这种类型的数据集尤其适合任意读取非常昂贵甚至不可行的情况，以及批大小取决于取得的数据的情况。

例如，这样的数据集，当调用iter(dataset)时，将会返回一个数据集的数据读取流，一个远程的服务器甚至是记录可以实时生成。

更多细节见IterableDataset。

NOTE：当配合使用IterableDataset和多进程数据加载时，每一个进程的数据集对象是重复的，所以必须设置备份来避免重复数据。如何实现见IterableDataset文档。

1.3 IterableDataset 的文档

一个可迭代数据集。

所有表示数据样本的 iterable 都应该继承它。当数据来自一个流时，这种形式的数据集尤其有用。

所有的子类应该重写__iter__()，它将返回一个这个数据集中样本的 iterator。

当通过DataLoader使用一个子类时,数据集中的每一个项都会通过DataLoader iterator 得到。当num_worker > 0时，每个工人进程将会有数据集对象的一个不同备份，所以人们通常想独立设置每个备份来避免各个工人返回相同的数据。当在工人进程中调用get_worker_info()时，它会返回关于这个工人的信息。可以在数据集的__iter__()或DataLoader的worker_init_info选项中使用它来修改每个备份的行为。

例子 1：在__iter__()中修改所有工人的负担：

```
>>> class MyIterableDataset(torch.utils.data.IterableDataset):
...     def __init__(self, start, end):
...         super(MyIterableDataset).__init__()
...         assert end > start, "this example code only works with end >= start"
...         self.start = start
...         self.end = end
...
...     def __iter__(self):
...         worker_info = torch.utils.data.get_worker_info()
...         if worker_info is None:  # single-process data loading, return the full iterator
...             iter_start = self.start
...             iter_end = self.end
...         else:  # in a worker process
...             # split workload
...             per_worker = int(math.ceil((self.end - self.start) / float(worker_info.num_workers)))
...             worker_id = worker_info.id
...             iter_start = self.start + worker_id * per_worker
...             iter_end = min(iter_start + per_worker, self.end)
...         return iter(range(iter_start, iter_end))
...
>>> # should give same set of data as range(3, 7), i.e., [3, 4, 5, 6].
>>> ds = MyIterableDataset(start=3, end=7)

>>> # Single-process loading
>>> print(list(torch.utils.data.DataLoader(ds, num_workers=0)))
[3, 4, 5, 6]

>>> # Mult-process loading with two worker processes
>>> # Worker 0 fetched [3, 4].  Worker 1 fetched [5, 6].
>>> print(list(torch.utils.data.DataLoader(ds, num_workers=2)))
[3, 5, 4, 6]

>>> # With even more workers
>>> print(list(torch.utils.data.DataLoader(ds, num_workers=20)))
```

```
[3, 4, 5, 6]
```

例子 2：使用worker_init_fn修改所有工人的负担：

```
>>> class MyIterableDataset(torch.utils.data.IterableDataset):
...     def __init__(self, start, end):
...         super(MyIterableDataset).__init__()
...         assert end > start, "this example code only works with end >= start"
...         self.start = start
...         self.end = end
...
...     def __iter__(self):
...         return iter(range(self.start, self.end))
...
>>> # should give same set of data as range(3, 7), i.e., [3, 4, 5, 6].
>>> ds = MyIterableDataset(start=3, end=7)

>>> # Single-process loading
>>> print(list(torch.utils.data.DataLoader(ds, num_workers=0)))
[3, 4, 5, 6]
>>>
>>> # Directly doing multi-process loading yields duplicate data
>>> print(list(torch.utils.data.DataLoader(ds, num_workers=2)))
[3, 3, 4, 4, 5, 5, 6, 6]

>>> # Define a `worker_init_fn` that configures each dataset copy differently
>>> def worker_init_fn(worker_id):
...     worker_info = torch.utils.data.get_worker_info()
...     dataset = worker_info.dataset  # the dataset copy in this worker process
...     overall_start = dataset.start
...     overall_end = dataset.end
...     # configure the dataset to only process the split workload
...     per_worker = int(math.ceil((overall_end - overall_start) / float(worker_info.num_workers)))
...     worker_id = worker_info.id
...     dataset.start = overall_start + worker_id * per_worker
...     dataset.end = min(dataset.start + per_worker, overall_end)
...

>>> # Mult-process loading with the custom `worker_init_fn`
>>> # Worker 0 fetched [3, 4].  Worker 1 fetched [5, 6].
>>> print(list(torch.utils.data.DataLoader(ds, num_workers=2, worker_init_fn=worker_init_fn)))
[3, 5, 4, 6]
```

```
39
40 >>> # With even more workers
41 >>> print(list(torch.utils.data.DataLoader(ds, num_workers=20,
       worker_init_fn=worker_init_fn)))
42 [3, 4, 5, 6]
```

2 数据加载顺序和 Sampler

对于迭代形式的数据集，数据加载顺序完全由用户定义的 iterable 控制。这使得块读取和动态批大小（也就是，每次通过得到一个批打包的样本）的实现更加容易。

这个章节的剩余部分考虑映射风格的数据集。`torch.utils.data.Sampler`类被用来指定数据加载中使用的索引/键序列。它们表示遍历数据集索引的 iterable 对象。例如，在随机梯度下降的通常情形中，`Sampler`可以随意排列索引列表并每次得到一个索引，或者为迷你批随机梯度下降得到一小部分索引。

根据传入`DataLoader`的`shuffle`参数，一个序列或者打乱的采样器会被自动构建。此外，用户也可以使用`sampler`参数来指定自定义的每次得到下一个将要抓取的索引/键的`Sampler`对象。

每次得到批索引列表的自定义`Sampler`可以通过`batch_sampler`传入。自动批形成也可以通过`batch_size`和`drop_last`参数使能。更多细节见下一个章节。

NOTE：由于迭代形式的数据集没有键或索引的表示，所以`sampler`和`batch_sampler`都与迭代形式的数据集不兼容。

3 加载批和非批数据

`DataLoader`支持通过`batch_size`，`drop_last`和`batch_sampler`参数自动将每一个抓取的数据样本整理成批。

3.1 自动形成批（默认）

这是最常见的情形，对应于抓取数据的迷你批并将他们整理为批样本，也就是，包含有一个批维度的张量（通常是第一个）。

当`batch_size`（默认为 1）不是`None`时，数据加载器得到的是样本的批而不是单独的样本。`batch_size`和`drop_last`参数指定了数据加载如何得到数据集键

的批。对于映射形式的数据集，用户也可以选择指定每次得到一个键列表的batch_sampler。

NOTE：batch_size和drop_last参数实际上被用来从sampler构建一个batch_sampelr。对于映射形式的数据集，sampler通过用户提供或者基于shuffle参数构建。对于迭代形式的数据集，sampler是一个虚假的无穷 iterable。更多关于采样器的细节见这个部分。

NOTE：当配合使用迭代形式的数据集和多线程抓取时，drop_last参数丢弃每一个工人的数据集备份的最后一个不满批。

在使用采样器的索引抓取了一个样本列表后，作为collate_fn参数传入的函数将将样本列表整理成批。

在这种情况下，从一个映射形式的数据集加载大致等价于：

```
for indices in batch_sampelr:
    yield collate_fn([dataset[i] for i in indices])
```

从一个迭代形式的数据集加载则大致等价于：

```
dataset_iter = iter(dataset)
for indices in batch_sampler:
    yield collate_fn([next(dataset_iter) for _ in indices])
```

自定义的collate_fn可以被用来自定义整理方式，例如，将序列化数据填充至批中的最大长度。关于collate_fn的更多信息见这个部分。

3.2 关闭自动批形成

在某些情况下，用户可能想在数据集代码中手动处理形成批的过程，或者简单地加载每个单独的样本。例如，直接加载成批的数据（例如，从一个数据库批读或者读取内存的连续块）可能更加便宜，或者批大小是数据相关的，或者程序被设计用来在单独的样本上工作。在这些情况下，似乎不适用自动成批（collate_fn被用来整理样本）更好，而是让数据加载器直接返回dataset对象的成员。

当batch_size和batch_sampler都是None时（batch_sampler的默认值便是None），自动批形成将会关闭。从dataset得到的每一个样本将会作为参数传入collate_fn处理。

当自动批形成关闭时，默认的collate_fn简单地将 Numpy 数组转化为 PyTorch 张量，并不做其他事情。

在这种情况下，从一个映射形式的数据集加载大致等价于：

```
for index in sampler:
    yield collate_fn(dataset[index])
```

从一个迭代形式的数据集加载则大致等价于：

```
for dat in iter(dataset):
    yield collate_fn(data)
```

关于collate_fn的更多信息见这个章节。

3.3 和 collate_fn 一起工作

collate_fn的用法在自动成批使能和关闭的情况下有所不同。

当自动批形成关闭时，collate_fn通过每一个单独的数据样本调用，输出通过数据加载器 iterator 得到。在这种情况下，默认的collate_fn简单地将 Numpy 数组转化为 PyTorch 张量。

当自动成批使能时，collate_fn每次通过数据样本的列表调用。它应该将输入样本整理为从数据加载器 iterator 得到的批。这个章节的剩余内容描述了这种情况下collate_fn的默认行为。

例如，如果每一个数据样本由一个三通道图片和一个整数类别标签组成，也就是，数据集的每一个成员返回一个(image, class_index)元组，默认的collate_fn将这样的元组列表整理为一个成批图片张量和成批类别标签张量的单独元组。特别的，默认的collaet_fn有如下性质：

- 它总是添加一个作为批维度的新维度
- 它自动将 NumPy 数组和 Python 数值转化为 PyTorch 张量
- 它保留数据的结构，例如，如果每一个样本是一个字典，它输出一个拥有相同键但是值换为了批张量（如果值无法转化为张量，则是列表）。对于list, tuple和namedtuple等处理方式相同

用户可以使用自定义的collate_fn来实现自定义成批过程，例如，沿着其他而不是第一个维度成批，填充不同长度的序列或者增添对自定义数据类型的支持。

4 单线程和多线程数据加载

DataLoader默认使用单线程数据加载。

在一个 Python 进程中，全局解释器锁阻止了多个线程间的真正完全并行。为了避免数据加载堵塞计算代码，PyTorch 提供了一种通过将num_workers设置为一个正整数的方式来进行多进程数据加载的简单转化。

4.1 单进程数据加载（默认）

在这种模式下，数据抓取会在与DataLoader初始化相同的进程中完成。因此，数据加载可能堵塞计算。然而，当用于进程间共享的资源（例如共享存储，文件描述符）有限或者数据集小到可以全部加载入内存时，这个模式更好。额外地，单进程加载通常会显示可读性更强的错误痕迹，因此对于 debug 更加有用。

4.2 多进程数据加载

将num_workers设置为正整数将会开启特定加载工人进程数的多进程数据加载。

在这种模式下，每当DataLoader被创建（例如，当你调用enumerate(datalaoder)时），num_workers个工人进程也会被创建。在这时，dataset，collate_fn和worker_init_fn会被传给每个工人来进行初始化以及抓取数据。这意味着通过数据集内部 IO 访问数据集以及传输（包括collate_fn）会在工人进程中运行。

torch.utils.data.get_worker_info()返回工人进程中的多个有用信息（包括工人 id，数据集备份和初始化种子等），同时在主进程返回None。用户可以在数据集代码和/或worker_init_fn中使用这个函数来分别设置每个数据集备份并决定代码是否在工人进程中运行。例如，这对于数据集碎片化尤其有用。

对于映射形式的数据集，主进程使用sampler生成索引并将它们发送给工人。所以任意的打乱随机操作应该在主进程中完成，主进程会通过分发加载的索引来指导加载。

对于迭代形式的数据集，由于每个工人进程会得到一个dataset对象的备份，简单的多进程加载通常会导致重复数据。通过使用torch.utils.data.get_worker_info()和/或worker_init_fn，用户可以独立设置每个备份。（如何实现见IterabhleDataset文档。）出于类似的原因，在多进程加载中，drop_last参数会丢弃每一个工人的迭代式数据集备份的最后一个不全批。

一旦到达迭代的结尾或者 iterator 变成了被收集了的垃圾，工人进程就会被关闭。

WARNING：一般来说，因为在多进程中使用 CUDA 和共享 CUDA 张量存在诸多微妙之处，所以在多进程加载中返回 CUDA 张量是不推荐的。然而，我们推荐使用自动内存锁定（也就是，设置`pin_memory=True`），这将会使得将数据快速传输到 CUDA 使能的 GPU。

4.2.1 平台特定的行为

由于工人依赖于 Python 的`multiprocessing`，所以工人的启动行为在 Windows 和 Unix 是不同的。

- 在 Unix，`fork()`是`multiprocessing`的默认开始方法。使用`fork()`，子工人通常可以通过克隆的地址空间直接访问`dataset`和 Python 参数函数
- 在 Windows 或 MacOS，`spawn()`是`multiprocessing`的默认开始方法。使用`spawn()`，另外一个运行你的主脚本的解释器会被启动，接着内部工人函数通过`pickle`序列化来接受`dataset`和`collate_fn`和其他参数。

分开序列化意味着，当你使用多进程数据加载时，你应该通过两个步骤来确保你和 Windows 是兼容的：

- 将你主脚本的大部分代码使用`if __name__ == '__main__'`包裹，来确保它在每个工人进程启动时不会再次运行（更有可能产生错误）。由于数据集和`DataLoader`实例创建逻辑不需要在工人进程中再次执行，所以你可以将它们放在这里。
- 确保任何自定义的`collate_fn`，`worker_init_fn`或`dataset`代码被定义为顶级定义，也就是在`__main__`检查之外。这确保了它们在工人进程中是可用的。（由于函数仅被保存为引用而不是`bytecode`，所以这是有必要的。）

4.2.2 多进程数据加载中的随机性

在默认情况下，每个工人的 PyTorch 种子会设置为`base_seed + worker_id`，其中`base_seed`是一个主进程使用它的 RNG（因此，会强制小消耗一个 RNG 状态）或者一个指定的`generator`产生的长整数。然而，其他库的种子可能会在初始化工人时重复，导致每个工人返回相同的随机数。

在`worker_init_fn`中，你可以通过`torch.utils.data.get_worker_info().seed`或者`torch.initial_seed()`来设置 PyTorch 种子，并在数据加载前使用它来作为其他库的种子。

5 内存锁定

当主机到 GPU 备份来自固定（页锁定）内存时，它们的速度要快得多。关于何时以及如何使用固定内存的更多细节，见使用固定内存缓存。

对于数据加载，将`DataLoader`设置为`pin_memory=True`会自动将抓取的数据张量放在固定内存，因此使得到 CUDA 使能的 GPU 数据传输更快。

默认的内存固定逻辑仅仅识别张量和包含张量的映射以及 iteralbe。在默认情况下，如果固定逻辑发现一个自定义类型的批（当你使用返回自定义批类型的`collate_fn`时），或者如果批的每一个元素都是自定义类型，固定逻辑不会识别它们，并会在不固定内存的情况下返回批（或者哪些元素）。为了使能对于自定义批的内存固定，在你的自定义类型上定义`pin_memory()`方法。

见下面的例子。

```
class SimpleCustomBatch:
    def __init__(self, data):
        transposed_data = list(zip(*data))
        self.inp = torch.stack(transposed_data[0], 0)
        self.tgt = torch.stack(transposed_data[1], 0)

    # 在自定义类型上的自定义固定内存方法
    def pin_memory(self):
        self.inp = self.inp.pin_memory()
        self.tgt = self.tgt.pin_memory()
        return self

def collate_wrapper(batch):
    return SimpleCustomBatch(batch)

inps = torch.arange(10 * 5, dtype=torch.float32).view(10, 5)
tgts = torch.arange(10 * 5, dtype=torch.float32).view(10, 5)
dataset = TensorDataset(inps, tgts)

loader = DataLoader(dataset,
                    batch_size=2,
                    collate_fn=collate_wrapper,
                    pin_memory=Treu
                    )

for batch_ndx, sample in enumerate(loader):
    print(sample.inp.is_pinned())
    print(sample.tgt.is_pinned())
```