Open in app ↗ Sign up Sign In

Published in Towards Data Science

You have **2** free member-only stories left this month. Sign up for Medium and get an extra one

Yousef Nami Follow

Jan 27, 2022 · 5 min read · ✦ · Listen

Save

Reading .h5 Files Faster with PyTorch Datasets

Using the method of weak shuffling

36 1

Photo courtesy of Amol taken from Unsplash.

Recently, I worked on a Multi-task Deep Learning project using the Oxford Pet Dataset (citation under references). While I was able to overcome the excruciatingly slow training speeds by utilising Colab's GPU, I soon ran into a bottleneck when loading the data. Despite using a hefty Multi-task model, for a batch size of 32, the

training speeds were around 0.5 seconds while the data loading process was around 12...

After searching deep within the internet, I found that this is a problem faced by many (see PT Forums, S/O #1 and S/O #2) and something that has not been addressed adequately. I came up with a PyTorch solution that I believe is valid for most cases. This decreased my loading times on Colab from 12 seconds to around 0.4. The article will first explain the bottleneck, then introduce the solution. This is followed by empirical results and a comment on theoretical guarantees. I assume knowledge of PyTorch Datasets, however good working knowledge of classes in Python is sufficient.

Where does the bottleneck come from?

The way standard Datasets are created in PyTorch is based on the `torch.utils.data.Dataset` class. These custom datasets have a `__getitem__` method which, upon calling, loads the data for a given set of indices. The Dataset that I wrote looked like this:

```
    def __getitem__(self, index):
    """Method to query data from pre-loaded .h5 file generators based on a given index
    :returns: (image tensor, target tensors dict)
    """
        inputs = self.inputs['data'][index]  # inputs is a dict containing generators of

        inputs = self.transform(inputs)

        targets = {
            task: transform(self.targets[task]['data'][index]) for task, transform in se
        }
        return (inputs, targets)


    def _load_h5_file_with_data(self, file_name):
      """Method for loading .h5 files

      :returns: dict that contains name of the .h5 file as stored in the .h5 file, as we
      """
        path = os.path.join(self.dir_path, file_name)
        file = h5py.File(path)
        key = list(file.keys())[0]
        data = file[key]
        return dict(file=file, data=data)
```

dataset.py hosted with ♥ by GitHub view raw

dataset.py hosted with ❤ by GitHub view raw

when the Dataset is initialised to pre-load the .h5 files as generator objects, so as to prevent them from being called, saved and deleted each time __getitem__ is called.

Then, using the `torch.utils.data.Dataloader` class, I was defining a `trainloader` to batch my data for training purposes. Herein lay the problem.

Every time the `trainloader` is iterated over, the `__getitem__` method from the Dataset is called `batch_size` number of times in order to generate a batch of the data. This is problematic with .h5 datasets, because the time complexity for querying data from them scales linearly with the number of calls made to them. So, if the overhead from querying a single index is **x**, then in our case we'd expect a total time per batch of **32*x**.

```
from torch.utils.data import Dataloader

trainset = OxpetDataset()  # defines the dataset, is subclass of torch.utils.data.Dataset
trainloader = Dataloader(trainset, batch_size=32, shuffle=True)

for (inputs, targets) in trainloader:  # iterates over trainloader
  #  do model stuff
```

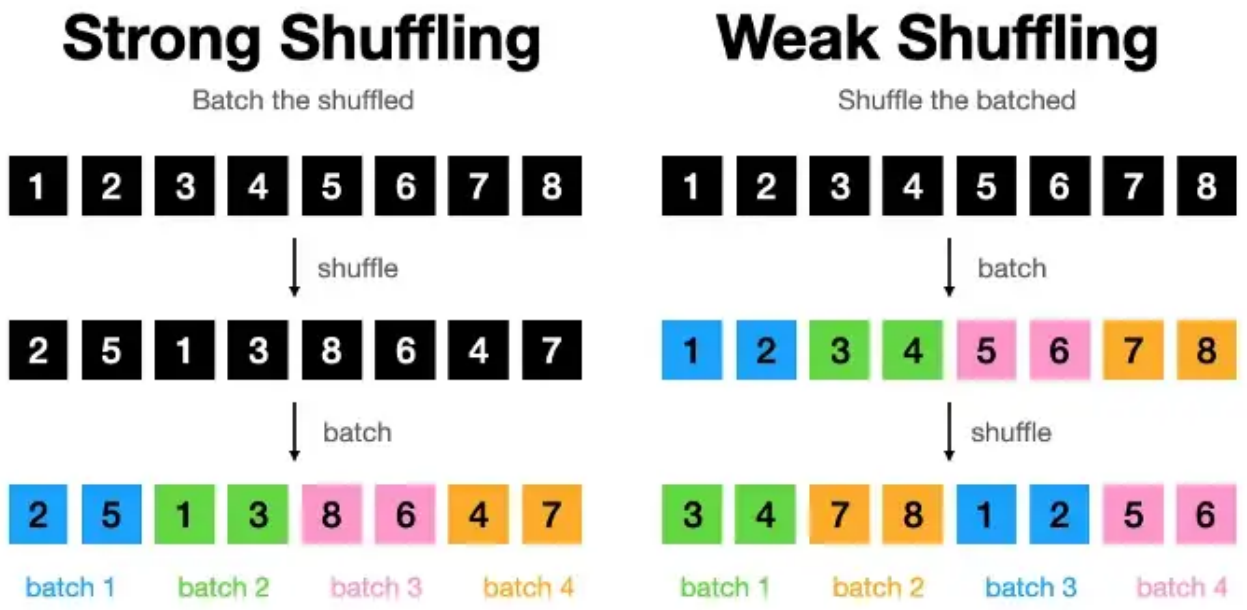
dataloader.py hosted with ❤ by GitHub view raw

Here, each pair of (inputs, targets) for the train loop would be created by the trainloader querying the dataset 32 times (with random indices since shuffle=True). The __getitem__ method is called 32 times, each time with a different index. The trainloader backend then aggregates the individual (inputs, targets) returns from the __getitem__ method, returning an (inputs, targets) pair that contains 32 items for the training loop.

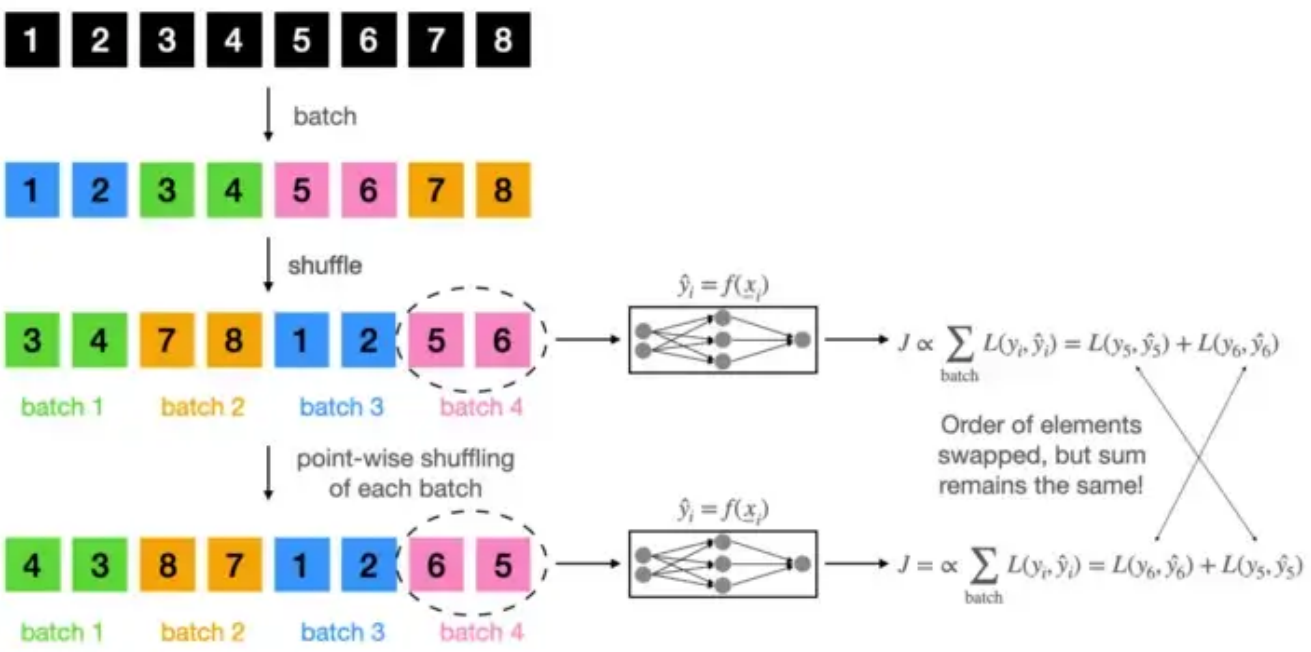
Speed Improvements Using Weak Shuffling

The key point to recognise is that the bottleneck from the .h5 files comes from finding the index at which the data sits. Finding subsequent indices that appear after it takes almost to time at all!

This means that the time complexity for querying the data at an index `i` is almost identical to querying the data from index `i:i+batch_size` . Therefore, if we were to batch, and then shuffle the batches (I call this 'weak shuffling') then we'd speed up the data loading process by a factor of `batch_size` . The diagram below shows how this method differs from the standard way of shuffling.



The clear implication with this method is that there is some bias introduced into the model, since we are not shuffling at point level. Though one may be tempted to shuffle each batch point-wise after the data has been shuffled at batch level in an attempt to decrease bias... this is futile, since the loss function is commutative. This is shown in the image below.



For **m >> batch_size** (where **m** is the number of datapoints) the method of weak shuffling should not greatly impact modelling power.

Weak Shuffling Implementation

The Dataset requires no changes. This is because the `index` parameter from `__getitem__` can be a list of indices. Thus, if we are able to pass a list `list(range(i,`

`i+batch_size))` then we are able to get an entire batch of data with a single query to the .h5 file.

This means that what needs changing is the dataloader. To achieve this, a custom loading scheme had to be created using the `sampler` parameter. This is shown below:

```python
from torch.utils.data import Sampler, BatchSampler, Dataloader
import torch

class RandomBatchSampler(Sampler):
    """Sampling class to create random sequential batches from a given dataset
    E.g. if data is [1,2,3,4] with bs=2. Then first batch, [[1,2], [3,4]] then shuffle b
    This is useful for cases when you are interested in 'weak shuffling'

    :param dataset: dataset you want to batch
    :type dataset: torch.utils.data.Dataset
    :param batch_size: batch size
    :type batch_size: int
    :returns: generator object of shuffled batch indices
    """
    def __init__(self, dataset, batch_size):
        self.batch_size = batch_size
        self.dataset_length = len(dataset)
        self.n_batches = self.dataset_length / self.batch_size
        self.batch_ids = torch.randperm(int(self.n_batches))

    def __len__(self):
        return self.batch_size

    def __iter__(self):
        for id in self.batch_ids:
            idx = torch.arange(id * self.batch_size, (id + 1) * self.batch_size)
            for index in idx:
                yield int(index)
        if int(self.n_batches) < self.n_batches:
            idx = torch.arange(int(self.n_batches) * self.batch_size, self.dataset_lengt
            for index in idx:
                yield int(index)


def fast_loader(dataset, batch_size=32, drop_last=False, transforms=None):
    """Implements fast loading by taking advantage of .h5 dataset
    The .h5 dataset has a speed bottleneck that scales (roughly) linearly with the numbe
    of calls made to it. This is because when queries are made to it, a search is made t
    the data item at that index. However, once the start index has been found, taking th
    does not require any more significant computation. So indexing data[start_index: sta
    is almost the same as just data[start_index]. The fast loading scheme takes advantag
    because the goal is NOT to load the entirety of the data in memory at once, weak shu
    strong shuffling.

    :param dataset: a dataset that loads data from .h5 files
    :type dataset: torch.utils.data.Dataset
    :param batch_size: size of data to batch
    :type batch_size: int
```

```
    :param drop_last: flag to indicate if last batch will be dropped (if size < batch_si
    :type drop_last: bool
    :returns: dataloading that queries from data using shuffled batches
    :rtype: torch.utils.data.DataLoader
    """
    return DataLoader(
        dataset, batch_size=None,  # must be disabled when using samplers
        sampler=BatchSampler(RandomBatchSampler(dataset, batch_size), batch_size=batch_s
    )
```

fastloader.py hosted with ❤ by GitHub view raw

Theoretical Guarantees

To date, I have not found theoretical guarantees for the method of weak shuffling in terms of the bias that we induce in the model. I have asked this question on Cross Validated and Maths Exchange, and will be updating this article if I get responses to them. Till then, I would appreciate any suggestions!

Key Takeaways

- The methods described in this article are valid for cases when you are unable to load the entire data into memory
- Loading batches from .h5 files using standard loading schemes is slow, because the time complexity scales with the number of queries made to the files
- The bottleneck comes from locating the first index, any subsequent indices (that come in order with no gaps in between!) can be loaded at almost no extra cost
- By using batch sampling, we can load the data in batches and thus decrease the load times by a factor of `batch_size`
- This means that the data would be weakly shuffled. The choice of `batch_size` is restricted to **batch_size << m**. Therefore, this solution isn't appropriate for cases when the bottleneck comes from training time.

The entire code for the multi-task-project can be found here. I will be posting more articles regarding the project in the near future, so stay tuned!

. . .

References

[1] Recommend the way to load larger h5 files. PyTorch Forums.

[2] torch.utils.data. PyTorch Documentation.

[3] How to use Batchsampler within a Dataloader. S/O.

[4] Is there a more efficient way of retrieving batches from a hdf5 dataset? S/O.

[5] Reading .h5 file is extremely slow. S/O.

[6] Random batch sampling from Dataloader. PyTorch Forums.

[7] Creating a custom Dataloader in PyTorch. Medium.

[8] Shuffle HDF5 dataset using h5py. S/O.

[9] HDF5 Datasets for PyTorch. Medium,

Datasets

[1] Oxford Pet Dataset. License: Creative Commons Attribution-ShareAlike 4.0 International License.

All images from author unless stated otherwise

Pytorch Python Deep Learning Dataloader Hdf 5

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

Get this newsletter

Medium

About Help Terms Privacy

Get the Medium app