The fast train loops

Nasy

Jun 14, 2024

Table of Contents

Introduction

Question

Which of the following is the fastest loop for iterating over 1000 epochs and 300,000 samples with batch size 300?

1. Manual loop with pure python
2. PyTorch Dataset and Dataloader w/ multiple workers
3. Tensorflow Dataset
4. Manual loop with numpy

Manual loop with pure python

```
from random import sample
from tqdm import tqdm

dataset = list(range(300_000))
for i in tqdm(range(1000)):
  rd = sample(dataset, k=len(dataset))
  for ii in range(0, len(rd), 300):
    rd[ii:ii+300]
    continue
```

PyTorch version

```
import torch
from torch.utils.data import DataLoader
from tqdm import tqdm

dataset = torch.arange(300_000)
dl = DataLoader(dataset, batch_size=300, shuffle=True,
                num_workers=10, prefetch_factor=16,
                pin_memory=True, persistent_workers=True)

for i in tqdm(range(1000)):
  for ii in dl:
    continue
```

Tensorflow version

```
import tensorflow as tf
from tqdm import tqdm

dataset = tf.data.Dataset.range(300_000)
dl = (dataset.shuffle(buffer_size=dataset.cardinality(),
                      reshuffle_each_iteration=True)
             .batch(300)
             .prefetch(tf.data.experimental.AUTOTUNE))

for i in tqdm(range(1000)):
  for _,ii in zip(range(1000), dl):
    continue
```

Manual loop with numpy

```
import numpy as np
from tqdm import tqdm

dataset = np.arange(300_000)
rng = np.random.default_rng()

for i in tqdm(range(1000)):
  rd = rng.permutation(dataset).reshape(-1, 300)
  for ii in rd:
    continue
```

Why Slow? Why Fast?

Inside PyTorch Dataset and DataLoader

The basic torch dataset:

```
class D(Dataset):

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

  def __getitem__(self, idx):
    return self.data[idx]

  def __len__(self):
    return len(self.data)
```

Every batch, it will use call batch size times `__getitem__` method. which is supper slow.

Example

```
In [14]: xx = torch.arange(300_000)
Performance: 2ms, 103µs, 334ns
Process: 9ms, 14µs

In [15]: %timeit torch.stack([xx[i] for i in range(100)])
122 µs ± 695 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
Performance: 9s, 890ms, 857µs, 875ns
Process: 9s, 824ms, 205µs
```

100 times `__getitem__` call will use around 122 μs.

How to solve?

Base idea

PyTorch support one time get multiple items. Thus, we only need to run one time `__getitem__` call. Down the time from 122 μs to 3.42 μs.

```
In [15]: %timeit torch.stack([xx[i] for i in range(100)])
122 µs ± 695 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
Performance: 9s, 890ms, 857µs, 875ns
Process: 9s, 824ms, 205µs

In [16]: %timeit idxs = torch.arange(100); xx[idxs]
3.42 µs ± 50.7 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Performance: 2s, 802ms, 937µs, 916ns
Process: 2s, 787ms, 333µs
```

Inside Dataset

Subclasses could also optionally implement :meth:`__getitems__`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Custom Dataset

```
class FD(D):

  def __getitems__(self, idxs):
    return self[idxs]

d = D(torch.arange(300_000))
dl = DataLoader(d, batch_size=300, shuffle=True)
fd = FD(torch.arange(300_000))
fdl = DataLoader(fd, batch_size=300, shuffle=True, collate_fn=lambda x: x)

for i in tqdm(range(1000), ncols=40):
  for ii in dl:
    pass
```

Or A Custom DataLoader

```
class FastDataloader:

  def __init__(self, dataset, batch_size, shuffle=True):
    self.data = dataset
    self.batch_size = batch_size
    self.shuffle = shuffle

  def __iter__(self):
    self.idx = torch.randperm(len(self.data)) if self.shuffle else torch.arange(len(self.data))
    self.i = 0
    return self

  def __next__(self):
    if self.i >= len(self.data):
      raise StopIteration
    batch = self.idx[self.i:self.i+self.batch_size]
    self.i += self.batch_size
    return self.data[batch]
```

Other Tips

Random number generator

It is slow to generate random number in each epoch for big array.

```
In [124]: for i in range(1000):
              rng.permutation(np.arange(300_000))
Performance: 3s, 42ms, 600µs, 500ns
Process: 3s, 18ms, 318µs
```

Fast Random Number Generator

We can split the indexes into groups, and shuffle the groups and indexes inside the groups.

```
In [134]: def f(xs):
        2     shape = xs.shape
        3     xs = xs.reshape(-1, 3000)
        4     return rng.permutation(rng.permutation(xs, axis=1)).reshape(*shape)
Performance: 686µs, 708ns
Process: 666µs

In [135]: for i in range(1000):
        2     f(np.arange(300_000))
Performance: 1s, 225ms, 565µs, 625ns
Process: 1s, 191ms, 648µs
```

Fuse operator

Pointwise operations such as elementwise addition, multiplication, and math functions like sin(), cos(), sigmoid(), etc., can be combined into a single kernel. This fusion helps reduce memory access and kernel launch times.

```
In [17]: xs = torch.from_numpy(rng.random((1000, 300_000)))

In [18]: def gelu(x):
    ...:     return x * 0.5 * (1.0 + torch.erf(x / 1.41421))
    ...:

In [19]: _ = gelu(xs)

In [20]: %timeit gelu(xs)
1.81 s ± 249 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [21]: cgelu = torch.compile(gelu)

In [22]: _ = cgelu(xs)

In [23]: %timeit cgelu(xs)
390 ms ± 35.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

To Devices Problem

How to move a dict of tuple/list/dict of tensors to devices?

```
xs = {
  "im": {
    "label": torch.rand(1, 224, 224),
    "image": torch.rand(3, 224, 224),
  },
  "cim": torch.rand(3, 224, 224),
  "pim": [torch.rand(3, 224, 224), torch.rand(3, 224, 224)],
}
```

Automatic Solution

- Use DataParallel
- Use DistributedDataParallel

General Solution

```
def move_to(obj, device):
  if torch.is_tensor(obj):
    return obj.to(device):
  elif isinstance(obj, dict):
    res = {}
    for k, v in obj.items():
      res[k] = move_to(v, device)
    return res
  elif isinstance(obj, list):
    res = []
    for v in obj:
      res.append(move_to(v, device))
    return res
  else:
    raise TypeError("Invalid type for move_to")
```

PyTree Structure

There are three common structures for pytorch:

- Torch PyTree: https://github.com/pytorch/pytorch/blob/main/torch/utils/_pytree.py
- tensordict & tensorclass: https://pytorch.org/tensordict/stable/index.html
- deepmind tree: https://github.com/google-deepmind/tree.git
- optree: https://github.com/metaopt/optree.git

PyTree

This a simple version of PyTree:

```
from torch.utils._pytree import tree_map

xs = tree_map(lambda x: x.to(device), xs)
```

Tensordict

If you only need to store Tensors, you can use Tensordict

```
from tensordict import TensorDict

txs = TensorDict.from_dict(xs)
txs = txs.to(device)
```

optree

Full support for PyTree structure.

```
import optree

txs = optree.tree_map(lambda x: x.to(device), xs)
```

IO

PyTorch can load saved model to a specific device.

```
import io

temp = io.BytesIO()
torch.save(xs, temp)
temp.seek(0)
lxs = torch.load(temp, map_location=device)
```

JAX

You don't need to manually move the data to devices. However, you can still use `jax.device_put` to put any structure to devices.

Conclusion

Conclusion

- Dataloader Loops
  - Idea: less `__getitem__`
  - Custom Dataset
  - Custom Dataloader
- Other tips
  - Random number generator
  - Fuse operator
  - To device
    - DataParallel & DistributedDataParallel
    - Tree Structure
    - IO