Teddy Koker About Résumé

DataLoaders Explained: Building a Multi-Process Data Loader from Scratch

Dec 18, 2020

When training a Deep Learning model, one must often read and pre-process data before it can be passed through the model. Depending on the data source and transformations needed, this step can amount to a non-negligable amount of time, which leads to unecessarily longer training times. This bottleneck is often remedied using a `torch.utils.data.DataLoader` for PyTorch, or a `tf.data.Dataset` for Tensorflow. These structures leverage parallel processing and pre-fetching in order reduce data loading time as much as possible. In this post we will build a simple version of PyTorch's `DataLoader`, and show the benefits of parallel pre-processing.

The full code for this project is available at github.com/teddykoker/tinyloader.

A Naive Base

Before we get to parallel processing, we should build a simple, naive version of our data loader. To initialize our dataloader, we simply store the provided `dataset`, `batch_size`, and `collate_fn`. We also create a variable `self.index` which will store next index that needs to be loaded from the dataset:

```
class NaiveDataLoader:
    def __init__(self, dataset, batch_size=64, collate_fn=default_colla
        self.dataset = dataset
        self.batch_size = batch_size
        self.collate_fn = collate_fn
        self.index = 0
```

The `__iter__` method simply returns the object to be iterated over. Since this method is implicitly called anytime you iterate over the dataloader, we will want to reset `self.index` to 0:

```
    def __iter__(self):
        self.index = 0
```

```
        return self
```

In order for a Python object to be iterable, we must define the `__next__` method, which will provide the next batch from the dataset whenever it is called, by repeatedly calling a `get()` method to fill up the whole batch:

```
    def __next__(self):
        if self.index >= len(self.dataset):
            # stop iteration once index is out of bounds
            raise StopIteration
        batch_size = min(len(self.dataset) - self.index, self.batch_size
        return self.collate_fn([self.get() for _ in range(batch_size)])
```

Lastly, we define the `get()` method which is where we actually load the element at `self.index` from the dataset.

```
    def get(self):
        item = self.dataset[self.index]
        self.index += 1
        return item
```

All the `NaiveDataLoader` does is wrap some indexable `dataset`, allowing it to be iterated in mini-batches, as is usually done when training a model. It can be used like so:

```
>>> dataset = list(range(16))
>>> dataloader = NaiveDataLoader(dataset, batch_size=8)
>>> for batch in dataloader:
...     print(batch)
...
[0 1 2 3 4 5 6 7]
[ 8  9 10 11 12 13 14 15]
```

We now basically have a fully functional data loader; The only issue is that `get()` is loading in one element of dataset at a time, using the same process that would be used for training. This is fine for printing elements from a list, but could become very problematic the loop must stall while waiting to perform some file IO or potentially costly data augmentation.

Introducing Workers

To prevent data loading from blocking training, we can create “workers” that load the data asyncrounously. A simple way of doing this is providing each worker a queue of indicies for that worker load, and an output queue where the worker can place the loaded data. All the worker has to do is repeatedly check its index queue, and load the data if the queue is not empty:

```
def worker_fn(dataset, index_queue, output_queue):
    while True:
        try:
            index = index_queue.get(timeout=0)
        except queue.Empty:
            continue
        if index is None:
            break
        output_queue.put((index, dataset[index]))
```

Python's `multiprocessing.Queue` is perfect for this since it can be shared across processes.

Note: Python does have a `threading` package; however, due to the Global Interpreter Lock (GIL), execution of any Python code is limited to one thread at a time, while all other threads are locked. To circumvent this, we can use `multiprocessing`, which uses subprocesses instead of threads. Since each subprocess has its own memory, we do not have to worry about the GIL.

Multiprocess Data Loader

Using our worker function, we can define a multi-process data loader, subclassing our naive data loader. This data loader will spawn `num_workers` workers upon its initialization:

```
class DataLoader(NaiveDataLoader):
    def __init__(
        self,
        dataset,
        batch_size=64,
        num_workers=1,
        prefetch_batches=2,
        collate_fn=default_collate,
    ):
        super().__init__(dataset, batch_size, collate_fn)

        self.num_workers = num_workers
        self.prefetch_batches = prefetch_batches
        self.output_queue = multiprocessing.Queue()
        self.index_queues = []
        self.workers = []
        self.worker_cycle = itertools.cycle(range(num_workers))
```

```
        self.cache = {}
        self.prefetch_index = 0

        for _ in range(num_workers):
            index_queue = multiprocessing.Queue()
            worker = multiprocessing.Process(
                target=worker_fn, args=(self.dataset, index_queue, self
            )
            worker.daemon = True
            worker.start()
            self.workers.append(worker)
            self.index_queues.append(index_queue)

        self.prefetch()
```

We have a single `output_queue`, that is shared across all of the worker processes, each of which has its own `index_queue`. Additionaly, we will store `self.prefetch_batches`, which will determine how many batches per worker to fetch ahead of time, and `self.prefetch_index`, which denotes index of the next item to prefetch. Using this we can define our `prefetch()` method, which will keep adding indicies to each workers queue (in a round-robin fashion) until two batches of indicies are added:

```
    def prefetch(self):
        while (
            self.prefetch_index < len(self.dataset)
            and self.prefetch_index
            < self.index + 2 * self.num_workers * self.batch_size
        ):
            # if the prefetch_index hasn't reached the end of the datas
            # and it is not 2 batches ahead, add indexes to the index q
            self.index_queues[next(self.worker_cycle)].put(self.prefetc
            self.prefetch_index += 1
```

Now that we have figured out how we are adding indicies to each worker's queue, we need to override our dataloader's `get()` method to retrieve the loaded items.

```
    def get(self):
        self.prefetch()
        if self.index in self.cache:
            item = self.cache[self.index]
            del self.cache[self.index]
        else:
            while True:
```

```
                try:
                    (index, data) = self.output_queue.get(timeout=0)
                except queue.Empty:  # output queue empty, keep trying
                    continue
                if index == self.index:  # found our item, ready to ret
                    item = data
                    break
                else:  # item isn't the one we want, cache for later
                    self.cache[index] = data

        self.index += 1
        return item
```

To start, we call `prefetch()`, which will ensure the next batches are in the process of being loaded. We then check the cache to see if the item we want (with index `self.index`) has already been emptied from the `output_queue`. If it has, we can simply return it; otherwise we must continuesly check the `output_queue` for the item, caching any other items we encounter. This step is necessary, as we cannot guarantee the order in which items are recieved, even if they are prefetched in order.

With the `get()` method overriden, our data loader is almost complete. All that is left is some housekeeping to ensure our data loader can be iterated over multiple times, and does not leave any stray processes running:

```
    def __iter__(self):
        self.index = 0
        self.cache = {}
        self.prefetch_index = 0
        self.prefetch()
        return self
```

Just like our naive data loader, we will use the `__iter__` method to reset the state of our data loader. In addition, we will need to implement a `__del__` method, which is called when the data loader no longer has any references and is garbage-collected. We will use this to safely stop all of the workers:

```
    def __del__(self):
        try:
            # Stop each worker by passing None to its index queue
            for i, w in enumerate(self.workers):
                self.index_queues[i].put(None)
                w.join(timeout=5.0)
            for q in self.index_queues:  # close all queues
                q.cancel_join_thread()
```

```
                q.close()
            self.output_queue.cancel_join_thread()
            self.output_queue.close()
        finally:
            for w in self.workers:
                if w.is_alive():  # manually terminate worker if all el
                    w.terminate()
```

This is our full `DataLoader` implementation! Now we can test it to see if we observe any noticable improvements.

Testing

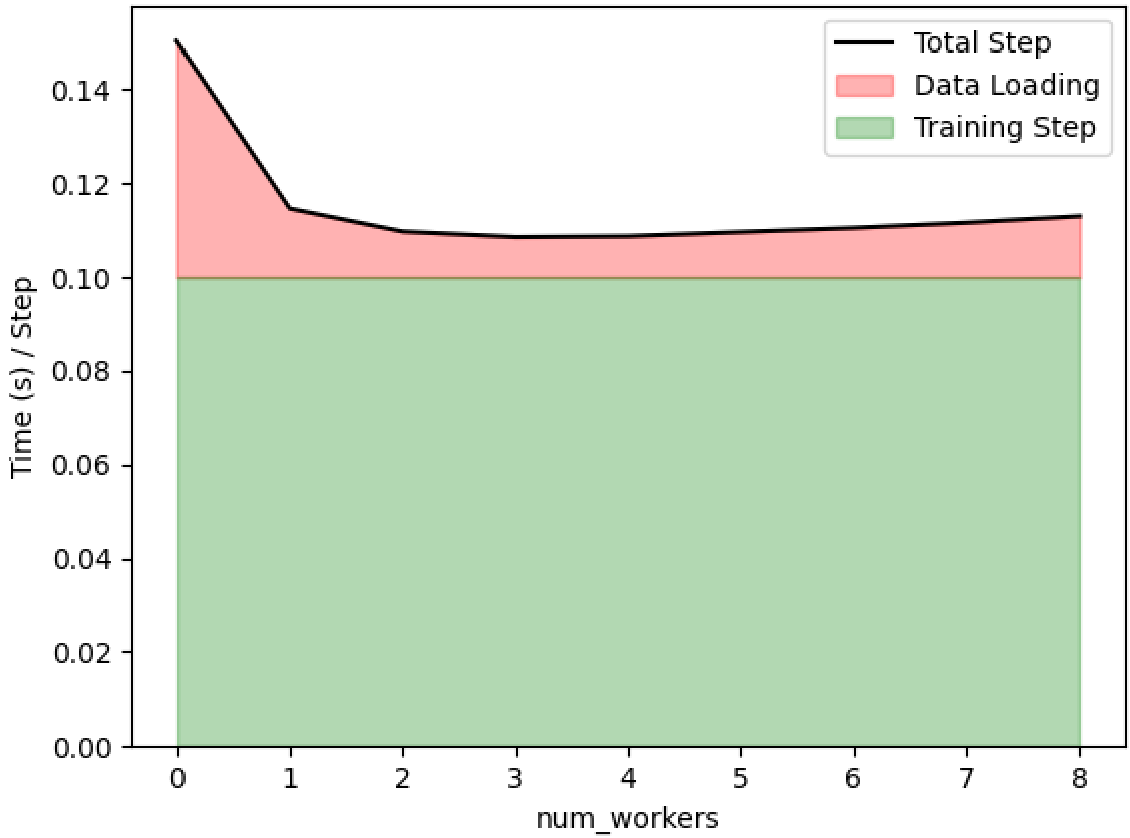
As a simple test, we can mock a dataset that requires some time to load an element simply by calling `time.sleep()` before returning an item:

```
class Dataset:
    def __init__(self, size=2048, load_time=0.0005):
        self.size, self.load_time = size, load_time

    def __len__(self):
        return self.size

    def __getitem__(self, index):
        time.sleep(self.load_time)
        return np.zeros((1, 28, 28)), 1  # return img, label
```

We can also mimic a training loop by iterating through a dataloader, sleeping every step to mock the time it would take to forward propegate, back propegate, and update the weights of a network:

```
def train(dataloader, epochs=10, step_time=0.1):
    steps = 0
    start = time.time()
    for epoch in range(epochs):
        for batch in dataloader:
            # mimic forward, backward, and update step
            time.sleep(step_time)
            steps += 1
    return (time.time() - start) / steps
```

For my contrived experiment, we will make each training step take 0.1 seconds, each individual item 0.0005 seconds to load from the dataset. We will then measure average time needed to perform a step with a batch size of 64, while we vary the number of workers:



The full code needed to reproduce this experiment is available here.

As expected, the naive data loader (`num_workers` = 0) performs far worse, as loading the full batch syncronously blocks the training step. As we increase the number of workers, we notice a steady improvement until 3-4 workers, where the data loading time starts to increase. This is likely the case because the memory overhead of having many processes pre-fetching data. Unfortunately there is no hard-and-fast rule for determining how many workers to use. Some have suggested that using a value equal to 4 times the number of GPUs being used, but I would recommend trying a few values to see what works best.

Overall, the `DataLoader` is a great tool for deep learning, and building one from scratch is a great way to understand how and why it works. As Richard Feynman wrote, “What I cannot create, I do not understand”.

Bonus: PyTorch Lightning

Often when applying deep learning to problems, one of the most difficult steps is loading the data. Once this is done, a great tool for training models is PyTorch Lightning. With Lightning, you simply define your `training_step` and `configure_optimizers`, and it does the rest of the work:

```
import pytorch_lightning as pl
import torch
from torch import nn

class Model(pl.LightningModule):
    def __init__(self):
        super().__init__()
        # define a simple multilayer perceptron
        self.mlp = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28 * 28, 128),
            nn.ReLU(),
            nn.Linear(128, 10),
        )

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.mlp(x)  # forward pass
        loss = nn.functional.cross_entropy(y_hat, y)  # compute loss
        return loss

    def configure_optimizers(self):
        # return the optimizser we want to use
        return torch.optim.Adam(self.mlp.parameters(), lr=1e-3)
```

With the model defined, we can use our own `DataLoader` implementation to train the model, which is very easy using Lightning's `Trainer` class:

```
from torch.utils.data.dataloader import default_collate as torch_collat

ds = Dataset()
dl = DataLoader(ds, collate_fn=torch_collate)
model = Model()
trainer = pl.Trainer(max_epochs=10)
trainer.fit(model, dl)
```

Lightning eliminates the need to rewrite the same training loop code over and over again, and also adds features like mixed-precision training, multi-node training, sharded optimizers, and even training on TPUs just by supplying different arguments to the `Trainer`. You can read more about Lightning on it's documentation.

Teddy Koker

Teddy Koker
teddy.koker@gmail.com

teddykoker
teddykoker
teddykoker

Algorithmic Trading and Machine Learning.