SPEECHMATICS

SPEECHMATICS HOW TO SERIES

HOW TO

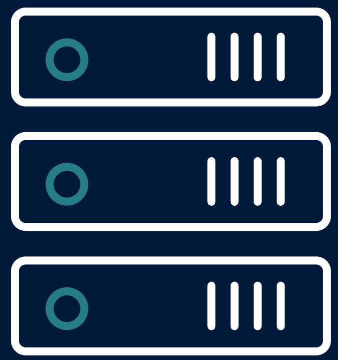
build a streaming dataloader with pytorch

WWW.SPEECHMATICS.COM

The release of PyTorch 1.2 brought with it a new dataset class:

`torch.utils.data.IterableDataset.`

This eBook provides examples of how it can be used to implement a parallel streaming DataLoader in PyTorch, as well as highlighting potential pitfalls to be aware of when using IterableDatasets for sequential data.



Part 1

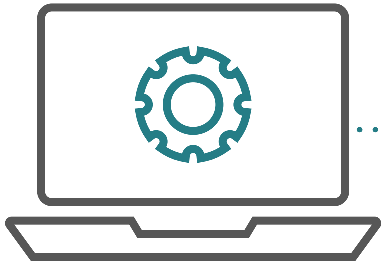
PyTorch Datasets and DataLoaders

PyTorch Datasets are objects that have a single job: to return a single datapoint on request

The exact form of the datapoint varies between tasks: it could be a single image, a slice of a time series, a tabular record or something else entirely. These are then passed on to a DataLoader which handles batching of datapoints and parallelism.

Before PyTorch 1.2

Before PyTorch 1.2 the only available dataset class was the original "map-style" dataset. This simply requires the user to inherit from the `torch.utils.data.Dataset` class and implement the `__len__` and `__getitem__` methods, where `__getitem__` receives an index which is mapped to some item in your dataset.



Let's see a very simple example.

```
from torch.utils.data import Dataset, IterableDataset, Dataloader

class MyMapDataset(Dataset):

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

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

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

This is instantiated and passed to the DataLoader, which is iterated over, returning batches of data to feed into our model.


```
from itertools import islice

data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

map_dataset = MyMapDataset(data)

loader = DataLoader(map_dataset, batch_size=4)

for batch in loader:
    print(batch)
______________________________________________________________

tensor([0,  1,  2,  3])
tensor([4,  5,  6,  7])
tensor([8,  9, 10, 11])
```

Flexible abstraction.

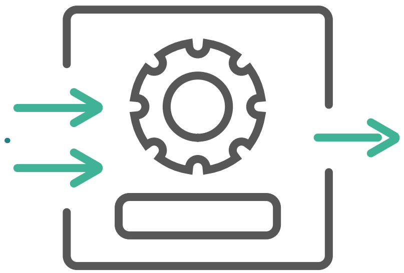
This remains a flexible abstraction, however, the assumption that you can trivially map each data point in your dataset means that it is less suited to situations where the input data is arriving as part of a stream, for example, an audio or video feed. Alternatively, each datapoint might be a subset of a file which is too large to be held in memory and so requires incremental loading during training. These situations can be addressed with more complex logic in our dataset or additional pre-processing of our inputs, but there is now a more natural solution, enter the `IterableDataset`!

```
from itertools import islice

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

      def __iter__(self):
          return iter(self.data)

iterable_dataset = MyIterableDataset(data)

loader = DataLoader(iterable_dataset, batch_size=4)

for batch in loader:
    print(batch)
______________________________________________________________

tensor([0,  1,  2,  3])
tensor([4,  5,  6,  7])
tensor([8,  9, 10, 11])
```

EXAMPLE

For both examples, we are returning the same result. So, what is the practical difference between these objects?

At a high level, each time the DataLoader returns a batch from the "map-style" dataset, it is sampling a set of indices and retrieving them with `map_dataset[idx]`. In contrast, for the IterableDataset the DataLoader is calling `next(iterable_dataset)` until it has built a full batch. One use-case where this latter approach excels is feeding data to a sequential model. A more concrete example is shown on the next page.

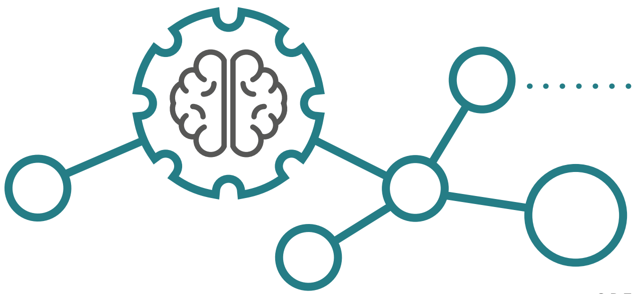


```
from itertools import cycle, islice

class MyIterableDataset(IterableDataset):

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

      def parse_file(self, file_path):
          with open(file_path, 'r') as file_obj:
                 for line in file_obj:
                       tokens = line.strip('\n').split(' ')
                       yield from tokens

      def get_stream(self, file_path):
          return cycle(self.parse_file(file_path))

      def __iter__(self):
          return self.get_stream(self.file_path)

iterable_dataset = MyIterableDataset('file.txt')

loader = DataLoader(iterable_dataset, batch_size=5)

for batch in islice(loader, 8):
    print(batch)

-------------------------------------------------------------

['Far', 'out', 'in', 'the', 'uncharted']
['backwaters', 'of', 'the', 'unfashionable', 'end']
['of', 'the', 'western', 'spiral', 'arm']
['of', 'the', 'Galaxy', 'lies', 'a']
['small', 'unregarded', 'yellow', 'sun.', 'Far']
['out', 'in', 'the', 'uncharted', 'backwaters']
['of', 'the', 'unfashionable', 'end', 'of']
['the', 'western', 'spiral', 'arm', 'of']
```

At each step of our very basic iterator, we are returning a single token from our dataset, which the DataLoader then aggregates into batches (each row of the output is a batch). We are using `itertools.cycle` here to create an infinite iterator, so when we reach the end of an epoch we loop back around to the start. This guarantees consistent batch sizes and avoids having to implement any file-end logic. Note that this example shows only a very short piece of text to illustrate the cycle in action. In practice, you would also want your dataset to return encoded indices instead of raw tokens.

Hopefully, it should now be clear in which scenarios the `IterableDataset` is useful. For the next set of examples, we return to using a numeric dataset with trivial parsing logic as this makes it easier to illustrate key points. Here is the previous example re-implemented.

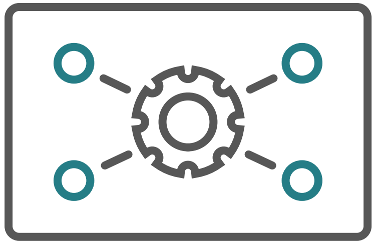
```
class MyIterableDataset(IterableDataset):

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

    def process_data(self, data):
        for x in data:
            yield x

    def get_stream(self, data):
        return cycle(self.process_data(data))

    def __iter__(self):
        return self.get_stream(self.data)


iterable_dataset = MyIterableDataset(data)

loader = DataLoader(iterable_dataset, batch_size=4)

for batch in islice(loader, 8):
    print(batch)

-------------------------------------------------------------

[0, 1, 2, 3]
[4, 5, 6, 7]
[8, 9, 0, 1]
[2, 3, 4, 5]
[6, 7, 8, 9]
[0, 1, 2, 3]
[4, 5, 6, 7]
[8, 9, 0, 1]
```



This example assumes that our entire input is contained within a single file, but we can easily extend our dataset to include multiple files of potentially inconsistent length by passing in a list of data sources and chaining them together.

```
from itertools import chain

class MyIterableDataset(IterableDataset):

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

    def process_data(self, data):
        for x in data:
            yield x

    def get_stream(self, data_list):
        return chain.from_iterable(map(self.process_data, cycle(data_list)))

    def __iter__(self):
        return self.get_stream(self.data_list)


data_list = [
    [12, 13, 14, 15, 16, 17],
    [27, 28, 29],
    [31, 32, 33, 34, 35, 36, 37, 38, 39],
    [40, 41, 42, 43],
]

iterable_dataset = MyIterableDataset(data_list)

loader = DataLoader(iterable_dataset, batch_size=4)

for batch in islice(loader, 8):
    print(batch)

-------------------------------------------------------------------------

[12, 13, 14, 15]
[16, 17, 27, 28]
[29, 31, 32, 33]
[34, 35, 36, 37]
[38, 39, 40, 41]
[42, 43, 12, 13]
[14, 15, 16, 17]
[27, 28, 29, 31]
```

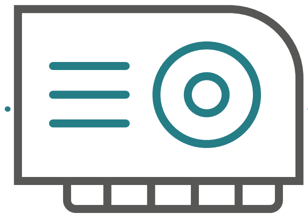
However, there is an issue. Whilst data points between batches are generally assumed to be independent, this is usually not true of a sequential model as persisted hidden state will often assume that the same position in each batch corresponds to a contiguous sequence across batches. In our current example, our sequence continues within a batch, rather than across batches. We can fix this by creating a separate stream for each position in the batch and then zipping them together. We are also required to set `batch_size=None` in the DataLoader to let PyTorch know that we are now handling the batching ourselves.



Remembering that each row is showing us a single batch, we have now achieved our aim. I.e. in the first batch we are returning the first item of the sequence, in the second batch we are returning the next item… But we now have a new problem. We are returning the same data in each batch position. This means that our parameter updates will just be seeing multiple copies of the same data, which is effectively the same as having a batch size of one.

We can fix this by ensuring that the stream in each batch position is different and there are multiple ways to achieve this. If we had a single large file, we could use `itertools.islice` to begin our iteration at a different offset within the file for each stream. If we have multiple files, like in our example, we could partition them into groups and feed each group into a single stream.

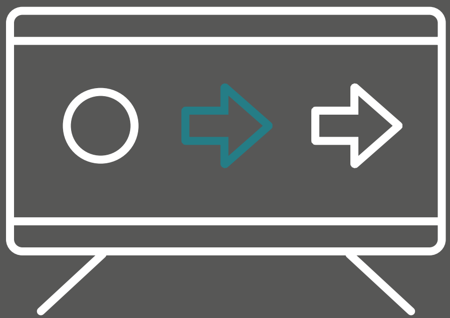
```
class MyIterableDataset(IterableDataset):

    def __init__(self, data_list, batch_size):
        self.data_list = data_list
        self.batch_size = batch_size

    def process_data(self, data):
        for x in data:
            yield x

    def get_stream(self, data_list):
        return chain.from_iterable(map(self.process_data, cycle(data_list)))

    def get_streams(self):
        return zip(*[self.get_stream(self.data_list) for _ in range(self.batch_size)])

    def __iter__(self):
        return self.get_streams()


iterable_dataset = MyIterableDataset(data_list, batch_size=4)

loader = DataLoader(iterable_dataset, batch_size=None)

for batch in islice(loader, 12):
    print(batch)

----------------------------------------------------------------------------------------

[12, 12, 12, 12]
[13, 13, 13, 13]
[14, 14, 14, 14]
[15, 15, 15, 15]
[16, 16, 16, 16]
[17, 17, 17, 17]
[27, 27, 27, 27]
[28, 28, 28, 28]
[29, 29, 29, 29]
[31, 31, 31, 31]
[32, 32, 32, 32]
[33, 33, 33, 33]
```



Alternatively, we could feed all files into every stream but simply shuffle the order of the files. Shuffling has a few advantages. Firstly, we do not have to worry about creating balanced partitions of the file list to spread across the streams. Secondly, we randomise the transition between files across the streams. This makes the model less likely to learn something spurious across these artificial boundaries if our model is not resetting its state, which is often true in language model training.


```
import random

class MyIterableDataset(IterableDataset):

    def __init__(self, data_list, batch_size):
        self.data_list = data_list
        self.batch_size = batch_size

    @property
    def shuffled_data_list(self):
        return random.sample(self.data_list, len(self.data_list))

    def process_data(self, data):
        for x in data:
            yield x

    def get_stream(self, data_list):
        return chain.from_iterable(map(self.process_data, cycle(data_list)))

    def get_streams(self):
        return zip(*[self.get_stream(self.shuffled_data_list)
                     for _ in range(self.batch_size)])

    def __iter__(self):
        return self.get_streams()


iterable_dataset = MyIterableDataset(data_list, batch_size=4)

loader = DataLoader(iterable_dataset, batch_size=None)

for batch in islice(loader, 12):
    print(batch)

----------------------------------------------------------------------------

[31, 40, 12, 27]
[32, 41, 13, 28]
[33, 42, 14, 29]
[34, 43, 15, 12]
[35, 12, 16, 13]
[36, 13, 17, 14]
[37, 14, 40, 15]
[38, 15, 41, 16]
[39, 16, 42, 17]
[27, 17, 43, 40]
[28, 31, 27, 41]
[29, 32, 28, 42]
```

Part 2



Going Parallel

When training a model, the bottleneck for training speed can often be data loading, rather than the forward/backward pass through the model. There aren't many benefits to using a GPU if it just spends most of its time sat around waiting for new data to arrive. We have already established that most use cases for the `IterableDataset` rule out caching our entire dataset in memory, so we will instead look towards data loading in parallel as a potential solution for this.

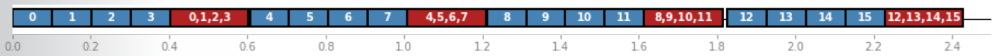
The good news is that Pytorch makes parallel data loading very easy. All you have to do is increase **`num_workers`** on your `DataLoader` object! The not so good news is that there are some caveats which must be considered when calling an `IterableDataset` in parallel to ensure that the data you return is what you are expecting.

To explore this let's go back to basics and revisit a simple map-style `Dataset` and `IterableDataset`, ignoring our earlier modifications for creating sequential streams. Rather than returning our data points, our datasets now time how long it takes to load each data point, as well as returning the id of the parallel worker that executed the task. We have inserted a constant delay into each dataset call to simulate loading a data point. We then plot a timeline which also includes a simulated pass through our model to illustrate the requirement for parallel loading.

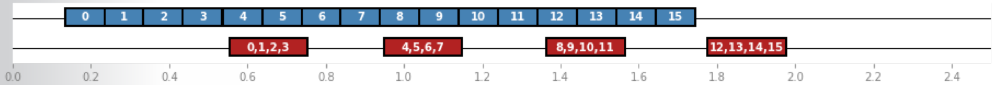
```
from torch.utils.data import DataLoader, Dataset, IterableDataset
import time

class MyMapDataset(Dataset):

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

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

    def __getitem__(self, idx):

        worker = torch.utils.data.get_worker_info()
        worker_id = worker.id if worker is not None else -1

        start = time.time()
        time.sleep(0.1)
        end = time.time()

        return self.data[idx], worker_id, start, end


data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]

map_dataset = MyMapDataset(data)

loader = DataLoader(map_dataset, batch_size=4, num_workers=0)

plot_timings(loader, model_time=0.2, n_batches=4)
```


Here are the timelines for loading our map-style dataset with zero, one and two workers.

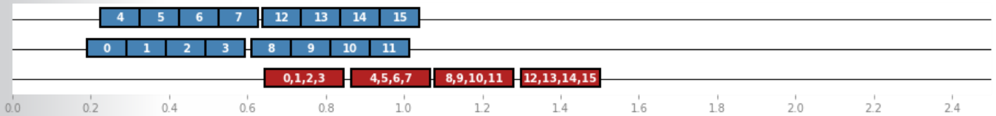
```
loader = DataLoader(map_dataset, batch_size=4, num_workers=0)

plot_timings(loader, model_time=0.2, n_batches=4)
```




```
loader = DataLoader(map_dataset, batch_size=4, num_workers=1)

plot_timings(loader, model_time=0.2, n_batches=4)
```




```
loader = DataLoader(map_dataset, batch_size=4, num_workers=2)

plot_timings(loader, model_time=0.2, n_batches=4)
```



The bottom row in each timeline represents the main Python process. Every other row shows a single subprocess (parallel worker). The red boxes show timings for the model forward/backwards pass while the blue boxes show timings for loading a single data point. Numbers inside boxes show the data point that was loaded, or in the case of the model, the data points that contributed to that batch.

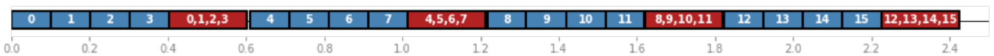
We can see that with `num_workers=0`, the main process takes charge of both the model pass and the data loading. Execution is completely in series with the model pass having to finish before data loading can continue. With `num_workers=1` we now have a separate process which is purely responsible for data loading. This allows us to start loading the next batch whilst the current batch is processed through the model. Note also that there is a slight delay at the start of processing due to the setup time incurred by the worker.

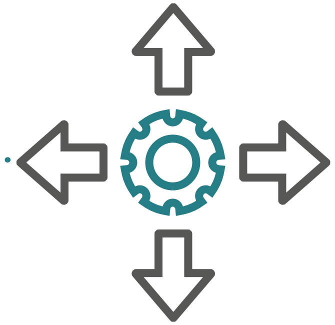
With one worker the model still has waiting time, but this is solved by setting `num_workers=2`. This guarantees that we have enough data loaded and waiting each time the model is ready to receive a batch. Once this state has been achieved, there is no real benefit to further increase the number of workers. In any case, PyTorch will only pre-fetch up to two batches per worker. Once the data queue is saturated the workers will now be in a situation where they are waiting.

We will now replicate these results this time using the `IterableDataset`.

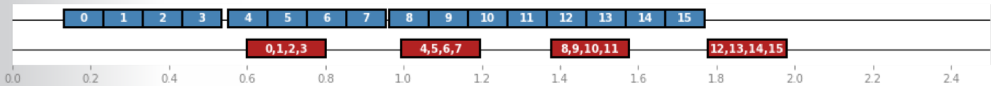
```
class MyIterableDataset(IterableDataset):

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

    def __iter__(self):
        for x in self.data:
            worker = torch.utils.data.get_worker_info()
            worker_id = worker.id if worker is not None else -1

            start = time.time()
            time.sleep(0.1)
            end = time.time()

            yield x, worker_id, start, end

iterable_dataset = MyIterableDataset(data)
```

```
loader = DataLoader(iterable_dataset, batch_size=4, num_workers=0)

plot_timings(loader, model_time=0.2, n_batches=4)
```



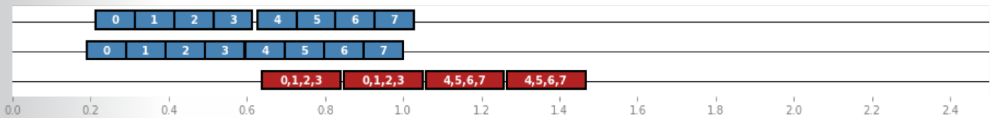


```
loader = DataLoader(iterable_dataset, batch_size=4, num_workers=1)

plot_timings(loader, model_time=0.2, n_batches=4)
```



With zero or one worker, we get the same result as the map-style dataset but look at what happens when we use two workers.

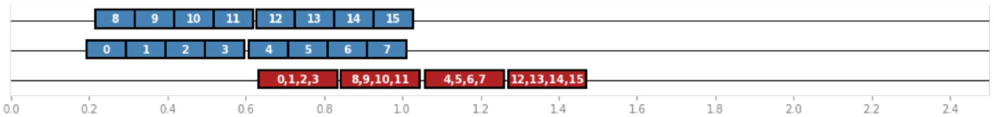
```
loader = DataLoader(iterable_dataset, batch_size=4, num_workers=2)

plot_timings(loader, model_time=0.2, n_batches=4)
```



We are now returning duplicate data across our batches! The reason for this is that when each worker is initialised it receives a copy of the dataset. In the case of the map-style dataset, this is not an issue as the object is stateless and the data to be retrieved is defined by index samples sent to each worker. However, when using IterableDataset each worker iterates over its own separate object which results in a duplicated output. This issue is highlighted in the PyTorch docs and the proposed solution is to add a `worker_init_fn` telling each worker to only process a subset of the data.

We can see a simple example below which divides our data across workers.

```
def worker_init_fn(_):
    worker_info = torch.utils.data.get_worker_info()

    dataset = worker_info.dataset
    worker_id = worker_info.id
    split_size = len(dataset.data) // worker_info.num_workers

    dataset.data = dataset.data[worker_id * split_size:(worker_id + 1) * split_size]


loader = DataLoader(iterable_dataset, batch_size=4, num_workers=2,
                    worker_init_fn=worker_init_fn)

plot_timings(loader, model_time=0.2, n_batches=4)
```

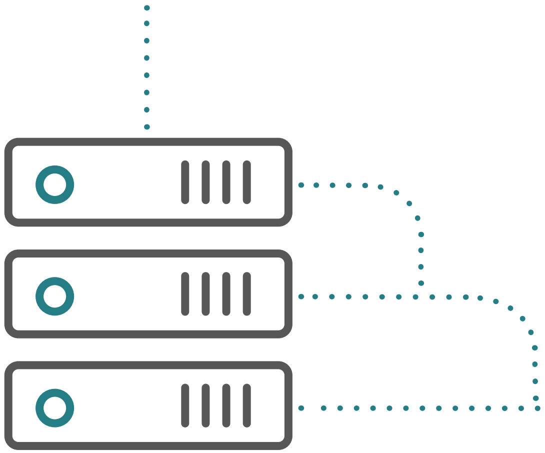


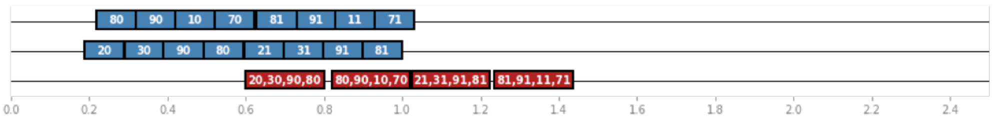
Note the order in which the data is returned: workers alternate in returning a single, entire batch. PyTorch guarantees this cyclic behaviour and we can view the outstanding tasks directly by inspecting the `task_info` dictionary. Each key represents the ordered task id and its corresponding value is the worker which will carry out that task.

```
print(iter(loader).task_info)

---------------------------------------------------------------------

{0: (0,), 1: (1,), 2: (0,), 3: (1,)}
```

Using this knowledge, we will now re-implement our "sequential" IterableDataset which we defined in the previous section and test that it is suitable for parallel execution.


```
import random
from itertools import chain, cycle

class MyIterableDataset(IterableDataset):

    def __init__(self, data_list, batch_size):
        self.data_list = data_list
        self.batch_size = batch_size

    @property
    def shuffled_data_list(self):
        return random.sample(self.data_list, len(self.data_list))

    def process_data(self, data):
        for x in data:
            worker = torch.utils.data.get_worker_info()
            worker_id = worker.id if worker is not None else -1

            start = time.time()
            time.sleep(0.1)
            end = time.time()

            yield x, worker_id, start, end

    def get_stream(self, data_list):
        return chain.from_iterable(map(self.process_data, cycle(data_list)))

    def get_streams(self):
        return zip(*[self.get_stream(self.shuffled_data_list)
                     for _ in range(self.batch_size)])

    def __iter__(self):
        return self.get_streams()

data_list = [
    [10, 11, 12, 13],
    [20, 21, 22, 23],
    [30, 31, 32, 33],
    [40, 41, 42, 43],
    [50, 51, 52, 53],
    [60, 61, 62, 63],
    [70, 71, 72, 73],
    [80, 81, 82, 83],
    [90, 91, 92, 93],
]

iterable_dataset = MyIterableDataset(data_list, batch_size=4)

loader = DataLoader(iterable_dataset, batch_size=None, num_workers=2)

plot_timings(loader, model_time=0.2, n_batches=4)
```



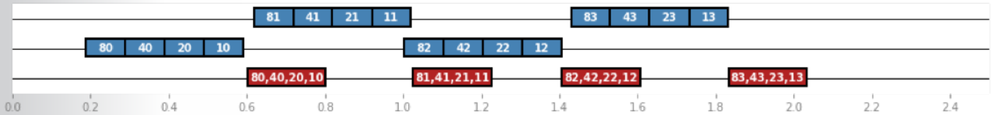


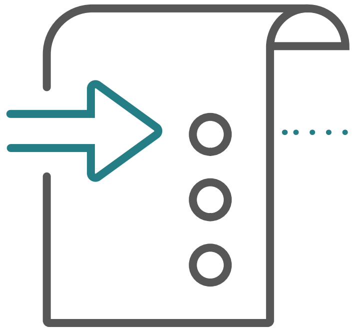
Our shuffled streams have implicitly solved the issue of duplicate batches, as each worker now has its own random seed. However, this does not help our sequential model which requires that **consecutive batches return contiguous items from each stream**. We need each worker to be operating on the same streams but at a different offset so that the returned data is interleaved and in the correct order. One approach to this solution is to fix the random seed of our workers and slice our streams with an offset equal to the number of workers. We can implement this change by modifying the `__iter__` method of our class.

```
def __iter__(self):
        worker_info = torch.utils.data.get_worker_info()

        dataset = worker_info.dataset
        worker_id = worker_info.id
        n_workers = worker_info.num_workers

        random.seed(50)

        return islice(self.get_streams(), worker_id, None, n_workers)
```



```
iterable_dataset = MyIterableDataset(data_list, batch_size=4)

loader = DataLoader(iterable_dataset, batch_size=None, num_workers=2)

plot_timings(loader, model_time=0.2, n_batches=4)
```





This returns our data in the correct order, with each batch position a continuation of the same sequence from the previous batch. However, parallel performance is lost as islice still needs to process each item to increment the iterator for a given worker. In other words, each worker is processing the same data but just returning a subset of it. Herein lies the fundamental problem with parallelism across batches with a streaming input. What we actually require is parallelism within batches where each worker is responsible for loading a subset of a single batch.

As PyTorch assumes that each parallel worker is returning an entire batch we will have to write our own DataLoader to achieve parallelism within batches. Rather than start from scratch, we can utilise subprocess management and data pre-fetching by splitting our batch into subsets, passing each part to a separate PyTorch DataLoader, then zipping the results together.

```python
class MyIterableDataset(IterableDataset):

    def __init__(self, data_list, batch_size):
        self.data_list = data_list
        self.batch_size = batch_size

    @property
    def shuffled_data_list(self):
        return random.sample(self.data_list, len(self.data_list))

    def process_data(self, data):
        for x in data:
            worker = torch.utils.data.get_worker_info()
            worker_id = id(self) if worker is not None else -1

            start = time.time()
            time.sleep(0.1)
            end = time.time()

            yield x, worker_id, start, end

    def get_stream(self, data_list):
        return chain.from_iterable(map(self.process_data, cycle(data_list)))

    def get_streams(self):
        return zip(*[self.get_stream(self.shuffled_data_list)
                     for _ in range(self.batch_size)])

    def __iter__(self):
        return self.get_streams()

    @classmethod
    def split_datasets(cls, data_list, batch_size, max_workers):

        for n in range(max_workers, 0, -1):
            if batch_size % n == 0:
                num_workers = n
                break

        split_size = batch_size // num_workers

        return [cls(data_list, batch_size=split_size)
                for _ in range(num_workers)]
```

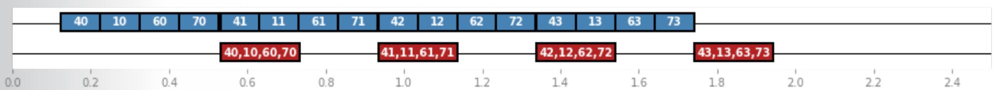
Continued on the next page


```python
class MultiStreamDataLoader:

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

    def get_stream_loaders(self):
        return zip(*[DataLoader(dataset, num_workers=1, batch_size=None)
                     for dataset in datasets])

    def __iter__(self):
        for batch_parts in self.get_stream_loaders():
            yield list(chain(*batch_parts))
```

Our new DataLoader now accepts multiple datasets as input and creates a corresponding DataLoader with exactly one worker. The only modification we have made to our original dataset is to add a factory method which instantiates multiple datasets, each contributing to part of a batch which is passed as input to our DataLoader. Note that rather than defining an absolute number of workers we now set a maximum number of workers and adjust the number of datasets we return accordingly. If our number of workers is not evenly divisible by batch size the workers will receive unbalanced loads. Whilst this is not really a problem, the additional workers are effectively redundant as we require all parts to be processed before returning a batch and so `max_workers` accounts for this by only using as many workers as will provide benefit.

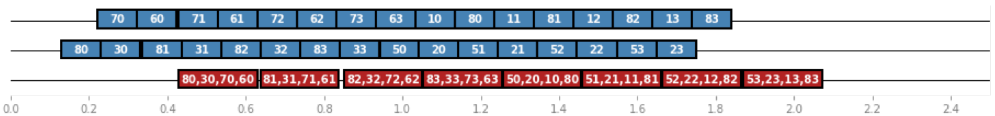
Here is an example which utilises a single worker to build the entire batch.

```python
datasets = MyIterableDataset.split_datasets(data_list, batch_size=4, max_workers=1)

loader = MultiStreamDataLoader(datasets)

plot_timings(loader, model_time=0.2, n_batches=4)
```



The order of the data being returned is correct, however, we still have some waiting time for the model. To rectify this, here is an example which uses two workers, where each worker is processing half of each batch. We also increase the `n_batches` returned to illustrate processing of multiple sequences.

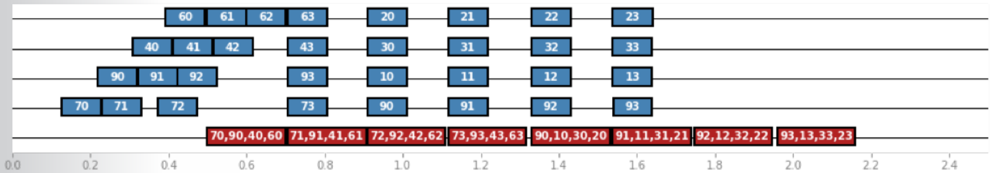
```
datasets = MyIterableDataset.split_datasets(data_list, batch_size=4, max_workers=2)

loader = MultiStreamDataLoader(datasets)

plot_timings(loader, model_time=0.2, n_batches=6)
```



Note that as the internal DataLoaders are instantiated in series there is a slight delay in start time for each worker, just as we were seeing previously when a single DataLoader was created. This, in turn, causes a slight delay in returning the very first batch.

Lastly, we will show an example with four workers, each processing a single item in each batch, although in this scenario having two workers is actually sufficient as we can see from the previous plot that DataLoaders are queuing sufficient data to avoid waiting times for the model pass.

```
datasets = MyIterableDataset.split_datasets(data_list, batch_size=4, max_workers=4)

loader = MultiStreamDataLoader(datasets)

plot_timings(loader, model_time=0.2, n_batches=6)
```



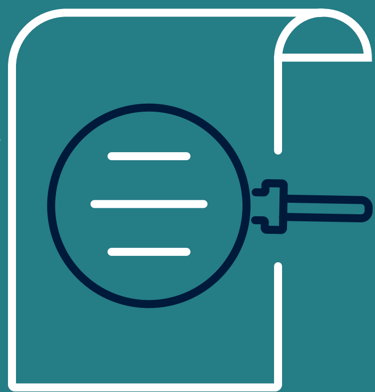
We again see the ramp-on in timings as each internal DataLoader is created and beyond that, we have full batches being fetched in unison across workers.

Summary

One final point which should be emphasised is that parallelism within batches relies heavily on balanced loads as **each batch is only as fast as its slowest worker**. This should not be an issue in the streaming scenario as each batch is processing inputs of constant size, but we should be wary of operations that have an irregular overhead. For example, unzipping or downloading an entire file before feeding it into a generator will have a high cost for specific batches. The effects of which are multiplied when working with multiple streams. In these scenarios, incremental unzipping or iterating over a streaming response should be preferred where possible.

Hopefully, these examples will help you on your way to building your own streaming dataset in PyTorch!

David MacLeod, Machine Learning Engineer, Speechmatics



SPEECHMATICS