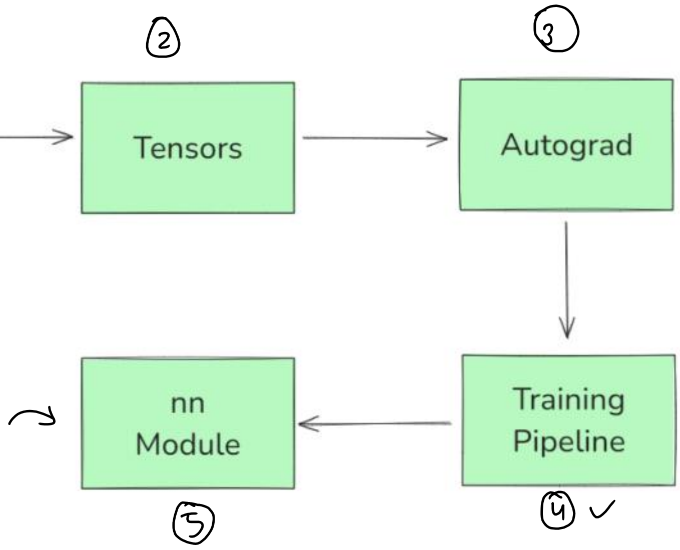
Recap

10 December 2024 17:04



Problems

1. Memory inefficient
2. Better Convergence

Solution - using batches of data to train the model

1000 rows

↳ 10 batch

↓

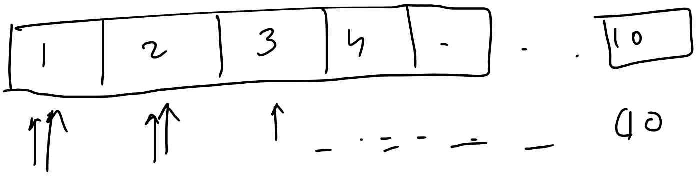
100 rows → forwa / loss / grad / grad

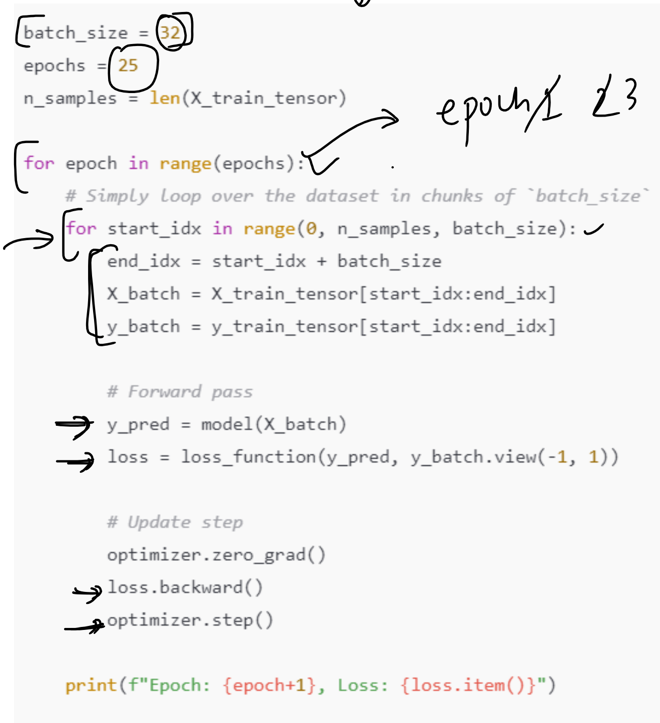
100 rows →

Simple Solution

12 December 2024 08:55

$\frac{320}{32} = 10$





```
batch_size = 32
epochs = 25
n_samples = len(X_train_tensor)

for epoch in range(epochs):
    # Simply loop over the dataset in chunks of `batch_size`
    for start_idx in range(0, n_samples, batch_size):
        end_idx = start_idx + batch_size
        X_batch = X_train_tensor[start_idx:end_idx]
        y_batch = y_train_tensor[start_idx:end_idx]

        # Forward pass
        y_pred = model(X_batch)
        loss = loss_function(y_pred, y_batch.view(-1, 1))

        # Update step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
```

epoch 1 2 3 . .

problems →

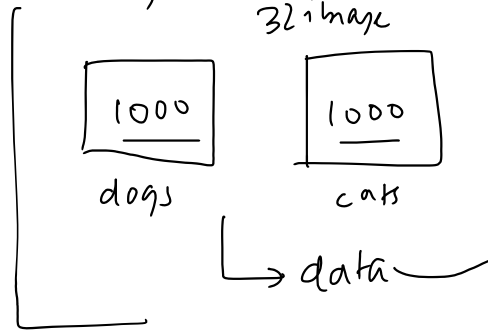
Problem with this approach!

12 December 2024 16:22

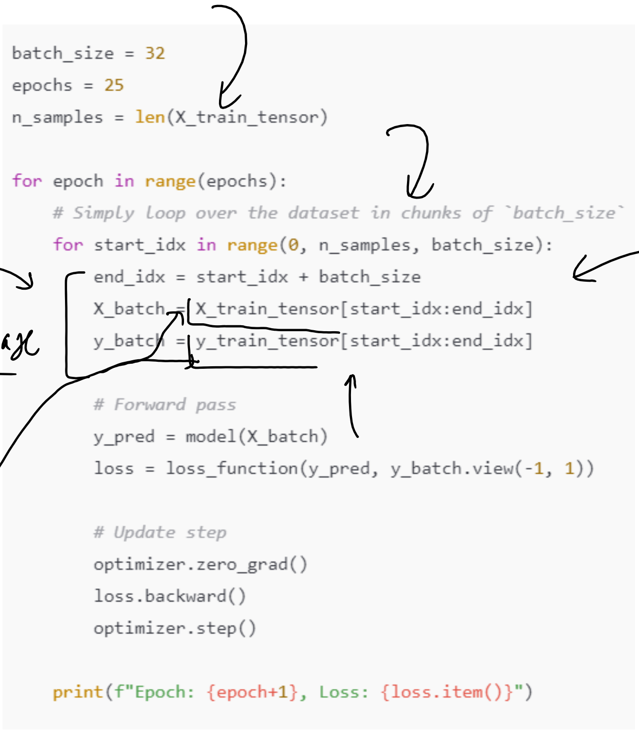
1. No standard interface for data
2. No easy way to apply transformations
3. Shuffling and sampling
4. Batch management & Parallelization

image

dogs vs cat
32 image



32 image



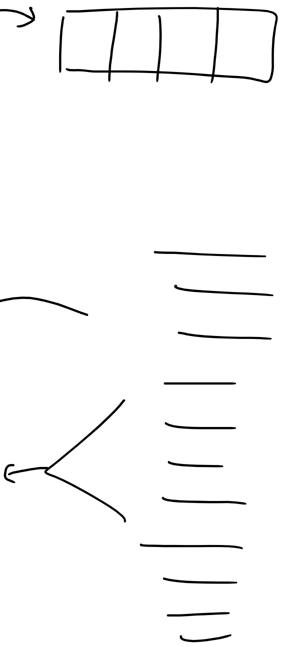
```
batch_size = 32
epochs = 25
n_samples = len(X_train_tensor)

for epoch in range(epochs):
    # Simply loop over the dataset in chunks of `batch_size`
    for start_idx in range(0, n_samples, batch_size):
        end_idx = start_idx + batch_size
        X_batch = X_train_tensor[start_idx:end_idx]
        y_batch = y_train_tensor[start_idx:end_idx]

        # Forward pass
        y_pred = model(X_batch)
        loss = loss_function(y_pred, y_batch.view(-1, 1))

        # Update step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch: {epoch+1}, Loss: {loss.item()}")
```

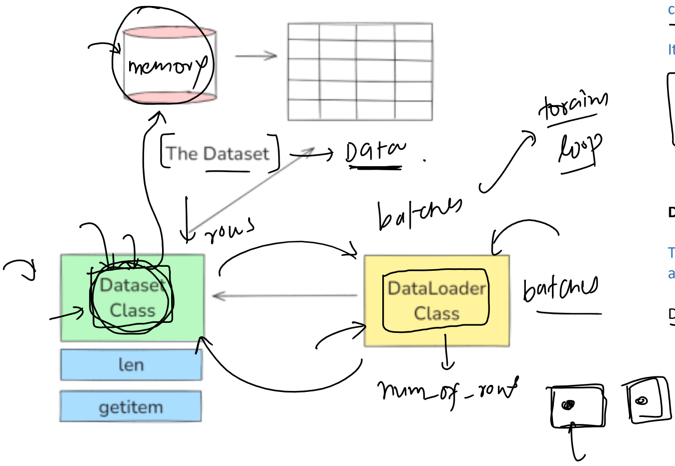


Dataset
Dataloader → mini batch
GD

The Dataset and Dataloader Classes

12 December 2024 08:47

Dataset and DataLoader are core abstractions in PyTorch that decouple how you define your data from how you efficiently iterate over it in training loops.



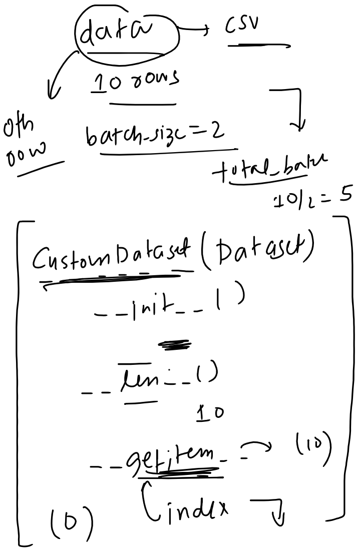
abstract class

Dataset Class

The Dataset class is essentially a blueprint. When you create a custom Dataset, you decide how data is loaded and returned.

It defines:

- __init__() which tells how data should be loaded.
- __len__() which returns the total number of samples.
- __getitem__(index) which returns the data (and label) at the given index.

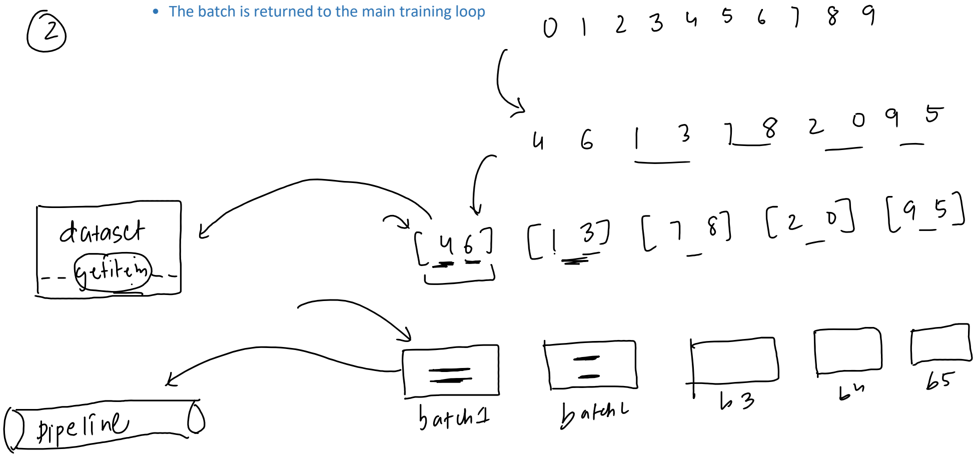


DataLoader Class

The DataLoader wraps a Dataset and handles batching, shuffling, and parallel loading for you.

DataLoader Control Flow:

- At the start of each epoch, the DataLoader (if shuffle=True) shuffles indices(using a sampler).
- It divides the indices into chunks of batch_size.
- for each index in the chunk, data samples are fetched from the Dataset object
- The samples are then collected and combined into a batch (using collate_fn)
- The batch is returned to the main training loop



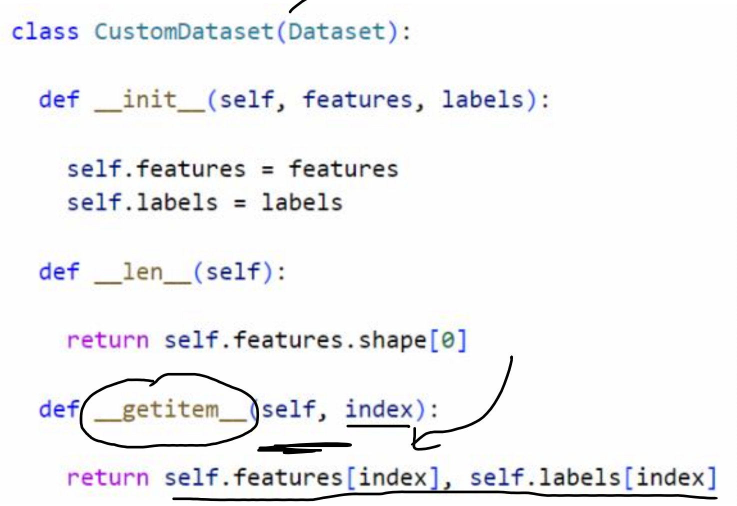
A Simple Example

12 December 2024 08:48

A note about data transformations

12 December 2024 17:51

Dataset



```
class CustomDataset(Dataset):

  def __init__(self, features, labels):

    self.features = features
    self.labels = labels

  def __len__(self):

    return self.features.shape[0]

  def __getitem__(self, index):

    return self.features[index], self.labels[index]
```

transformation

image
- resize
- black
- data aug

textual
- lower
- lem.
- stop

A note about Parallelization

12 December 2024 18:00

Imagine the entire data loading and training process for one epoch with num_workers=4:

Assumptions:

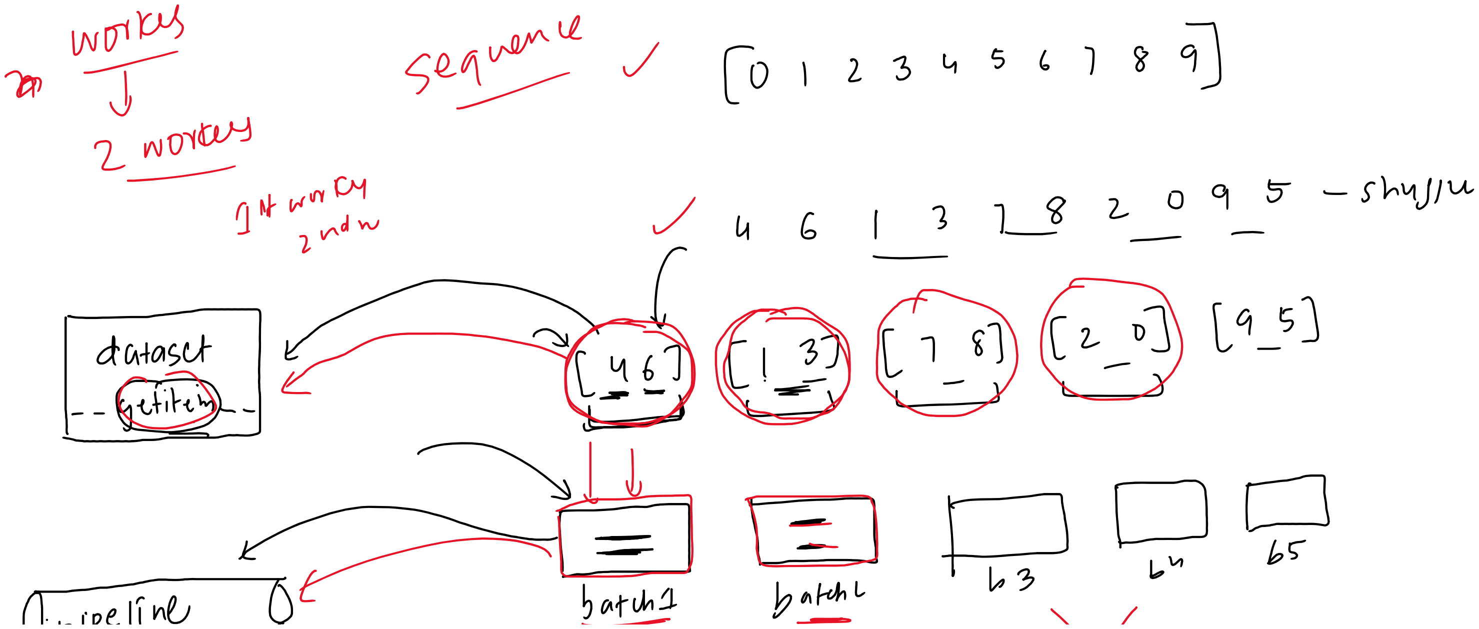
- Total samples: 10,000
- Batch size: 32
- Workers (num_workers): 4
- Approximately 312 full batches per epoch (10000 / 32 ≈ 312).

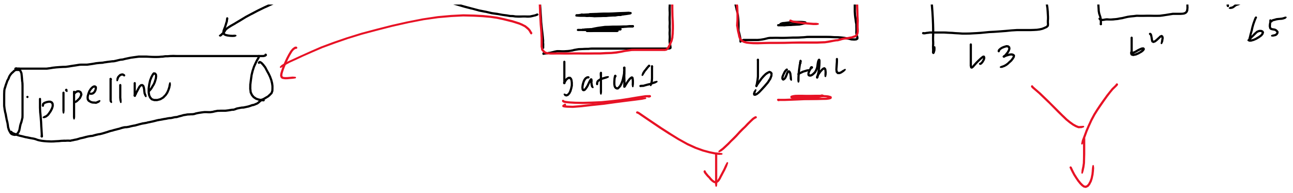
Workflow:

1. **Sampler and Batch Creation (Main Process):**
   Before training starts for the epoch, the DataLoader's sampler generates a shuffled list of all 10,000 indices. These are then grouped into 312 batches of 32 indices each. All these batches are queued up, ready to be fetched by workers.
2. **Parallel Data Loading (Workers):**
   - At the start of the training epoch, you run a training loop like:

     python
     Copy code
     ```
     for batch_data, batch_labels in dataloader:
         # Training logic
     ```
   - Under the hood, as soon as you start iterating over dataloader, it dispatches the first four batches of indices to the four workers:
     - Worker #1 loads batch 1 (indices [batch_1_indices])
     - Worker #2 loads batch 2 (indices [batch_2_indices])
     - Worker #3 loads batch 3 (indices [batch_3_indices])
     - Worker #4 loads batch 4 (indices [batch_4_indices])
       Each worker:
   - Fetches the corresponding samples by calling __getitem__ on the dataset for each index in that batch.
   - Applies any defined transforms and passes the samples through collate_fn to form a single batch tensor.
3. **First Batch Returned to Main Process:**
   - Whichever worker finishes first sends its fully prepared batch (e.g., batch 1) back to the main process.
   - As soon as the main process gets this first prepared batch, it yields it to your training loop, so your codefor batch_data, batch_labels in dataloader: receives (batch_data, batch_labels) for the first batch.
4. **Model Training on the Main Process:**
   - While you are now performing the forward pass, computing loss, and doing backpropagation on the first batch, the other three workers are still preparing their batches in parallel.
   - By the time you finish updating your model parameters for the first batch, the DataLoader likely has the second, third, or even more batches ready to go (depending on processing speed and hardware).
5. **Continuous Processing:**
   - As soon as a worker finishes its batch, it grabs the next batch of indices from the queue.
   - For example, after Worker #1 finishes with batch 1, it immediately starts on batch 5. After Worker #2 finishes batch 2, it takes batch 6, and so forth.
   - This creates a pipeline effect: at any given moment, up to 4 batches are being prepared concurrently.
6. **Loop Progression:**
   - Your training loop simply sees:

     python
     Copy code
     ```
     for batch_data, batch_labels in dataloader:
         # forward pass
         # loss computation
         # backward pass
         # optimizer step
     ```
   - Each iteration, it gets a new, ready-to-use batch without long I/O waits, because the workers have been pre-loading and processing data in parallel.
7. **End of the Epoch:**
   - After ~312 iterations, all batches have been processed. All indices have been consumed, so the DataLoader has no more batches to yield.
   - The epoch ends. If shuffle=True, on the next epoch, the sampler reshuffles indices, and the whole process repeats with workers again loading data in parallel.




pipeline
batch1
batch2
b3
b4
b5

A note about samplers

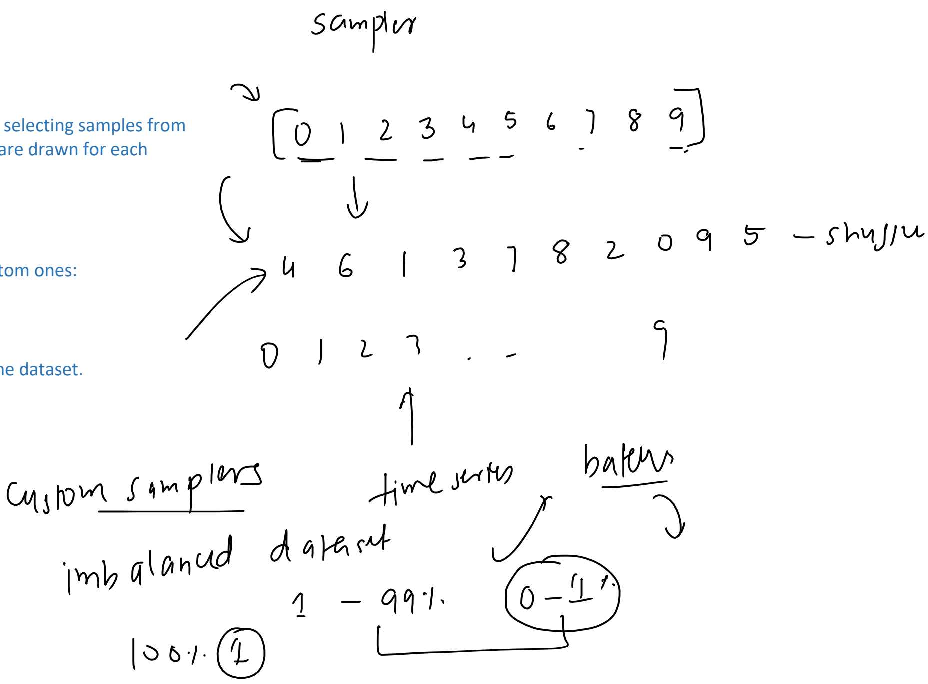
12 December 2024 17:51

In PyTorch, the sampler in the DataLoader determines the strategy for selecting samples from the dataset during data loading. It controls how indices of the dataset are drawn for each batch.

Types of Samplers

PyTorch provides several predefined samplers, and you can create custom ones:

1. SequentialSampler:
   - Samples elements sequentially, in the order they appear in the dataset.
   - Default when shuffle=False.
2. RandomSampler:
   - Samples elements randomly without replacement.
   - Default when shuffle=True.



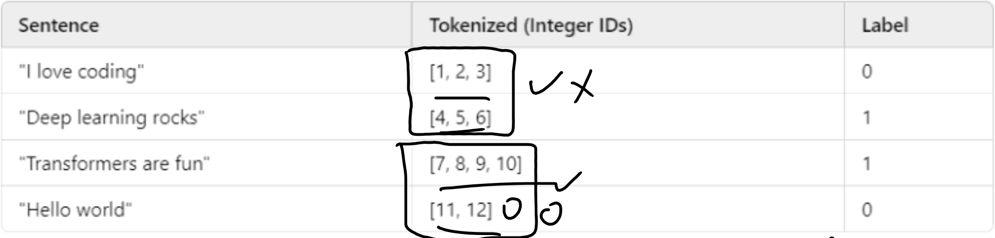
A note about collate_fn

12 December 2024 17:51

The collate_fn in PyTorch's DataLoader is a function that specifies how to combine a list of samples from a dataset into a single batch. By default, the DataLoader uses a simple batch collation mechanism, but collate_fn allows you to customize how the data should be processed and batched.

batch_size=2

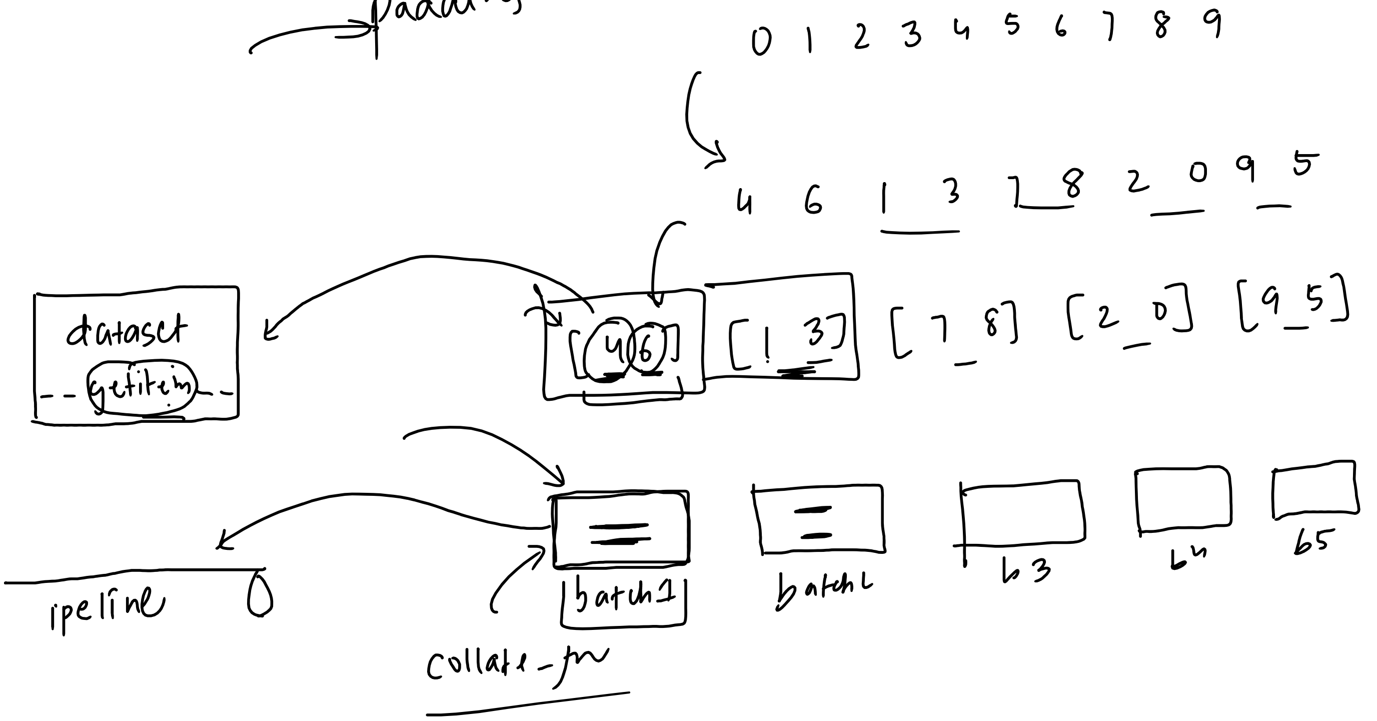
text



Sentence	Tokenized (Integer IDs)	Label
"I love coding"	[1, 2, 3]	0
"Deep learning rocks"	[4, 5, 6]	1
"Transformers are fun"	[7, 8, 9, 10]	1
"Hello world"	[11, 12]	0

padding

collate_fn

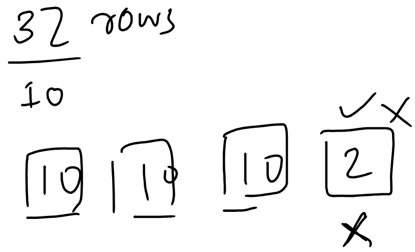


DataLoader Important Parameters

12 December 2024 18:01

The DataLoader class in PyTorch comes with several parameters that allow you to customize how data is loaded, batched, and preprocessed. Some of the most commonly used and important parameters include:

1. dataset (mandatory):
   - The Dataset from which the DataLoader will pull data.
   - Must be a subclass of torch.utils.data.Dataset that implements __getitem__ and __len__.
2. batch_size:
   - How many samples per batch to load.
   - Default is 1.
   - Larger batch sizes can speed up training on GPUs but require more memory.
3. shuffle:
   - If True, the DataLoader will shuffle the dataset indices each epoch.
   - Helpful to avoid the model becoming too dependent on the order of samples.
4. num_workers:
   - The number of worker processes used to load data in parallel.
   - Setting num_workers > 0 can speed up data loading by leveraging multiple CPU cores, especially if I/O or preprocessing is a bottleneck.
5. pin_memory: — gpu
   - If True, the DataLoader will copy tensors into pinned (page-locked) memory before returning them.
   - This can improve GPU transfer speed and thus overall training throughput, particularly on CUDA systems.
6. drop_last:
   - If True, the DataLoader will drop the last incomplete batch if the total number of samples is not divisible by the batch size.
   - Useful when exact batch sizes are required (for example, in some batch normalization scenarios).
7. collate_fn:
   - A callable that processes a list of samples into a batch (the default simply stacks tensors).
   - Custom collate_fn can handle variable-length sequences, perform custom batching logic, or handle complex data structures.
8. sampler:
   - sampler defines the strategy for drawing samples (e.g., for handling imbalanced classes, or custom sampling strategies).
   - batch_sampler works at the batch level, controlling how batches are formed.
   - Typically, you don't need to specify these if you are using batch_size and shuffle. However, they provide lower-level control if you have advanced requirements.



Improving our existing code

12 December 2024 08:49