Data Sets and Data Loaders

Introduction

In deep learning, datasets and data loaders play an important role in preparing and supplying data to models efficiently. Deep learning models require large amounts of data, but raw data is often unstructured and cannot be directly fed into a model. Effective data handling ensures:

- **Efficient Memory Usage**: Loading data in chunks rather than keeping it all in memory.
- **Scalability**: Working with datasets that do not fit into RAM.
- **Shuffling**: Avoiding bias due to order dependencies in the dataset.
- **Tranformation and Preprocessing**: Applying transformations such as normalization, resizing, cropping, type conversions etc..
- **Parallelization**: Using multiple CPU cores for faster data loading.

To address these issues, PyTorch provides a convenient way to handle datasets using the `torch.utils.data.Dataset` and `torch.utils.data.DataLoader` classes. These components help manage large datasets, perform transformations, and enable efficient batch processing.

The Dataset Class

In PyTorch, the `Dataset` class serves as a basis for handling and managing data efficiently. It provides a flexible way to load, preprocess, and structure datasets for deep learning models. The `Dataset` class is part of the `torch.utils.data` module and acts as an interface for accessing and retrieving data samples in a structured manner.

The `Dataset` class is particularly useful when working with large datasets that cannot be fully loaded into memory. The implementations of this class can be used to efficiently load data on demand, apply transformations, and manage diverse data formats such as images, text, and structured numerical data.

A dataset in PyTorch is typically an instance of a class that extends `torch.utils.data.Dataset`. It needs to implement three essential methods:

- **`__init__()`**: Initializes the dataset, loads file paths, and performs any necessary setup.
- **`__getitem__(index)`**: Returns a single data sample and its label at the given index.
- **`__len__()`**: Returns the total number of samples in the dataset.

Example: Assuming a set of images stored in a directory with corresponding labels in a CSV file, we create a custom dataset class, which:

- Reads image paths and labels from a CSV file.
- Loads and transforms images.
- Returns an image and its corresponding label.

```
import torch
from torch.utils.data import Dataset
import pandas as pd
from PIL import Image

class CustomImageDataset(Dataset):
    def __init__(self, csv_file, root_dir, transform=None):
        self.annotations = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform

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

    def __getitem__(self, index):
        img_path = f"{self.root_dir}/{self.annotations.iloc[index, 0]}"
        image = Image.open(img_path).convert("RGB")
        label = self.annotations.iloc[index, 1]

        if self.transform:
            image = self.transform(image)

        return image, label
```

The DataLoader Class

PyTorch provides the `DataLoader` class as part of the `torch.utils.data` module to handle the complexities of batch loading, shuffling, and parallel processing of datasets.

The `DataLoader` class acts as a bridge between a dataset and a model by automating the process of fetching mini-batches, applying transformations, and utilizing multiple CPU cores for faster data retrieval. It is particularly useful when working with large datasets that cannot fit into memory all at once.

Manually loading data sample-by-sample would be inefficient, especially for large-scale deep learning tasks. The `DataLoader` class simplifies this process by:

- **Batching Data**: Fetches multiple samples at once, improving computational efficiency.
- **Shuffling**: Randomizes data order to prevent model biases.

- **Parallel Loading**: Uses multiple worker threads to speed up data fetching.
- **Automatic Memory Management**: Helps in handling large datasets efficiently.

Once a dataset is defined, we use `DataLoader` to:

- Efficiently load and batch data.
- Shuffle data.
- Perform parallel processing for faster loading.

```
from torch.utils.data import DataLoader

dataset = CustomImageDataset(csv_file="data/labels.csv", root_dir="data/images")
dataloader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=4)
```

Important parameters of `DataLoader`

- **`batch_size`**: Number of samples per batch.
- **`shuffle`**: Whether to shuffle data before every epoch.
- **`num_workers`**: Number of worker processes for data loading (more workers speed up loading).
- **`drop_last`**: Whether to drop the last incomplete batch (useful in training for batch normalization).

Collation Functions

In PyTorch, a **collation function** is used by the `DataLoader` class to merge individual samples into a batch. This function determines how multiple samples from a dataset are combined into a single batch for training. The default behavior of the `DataLoader` is to simply stack tensors along the first dimension, but this may not always be suitable—especially for variable-length sequences, structured data, or complex custom datasets.

The **collation function** is defined using the `collate_fn` parameter in `DataLoader` and is useful for:

- **Handling Variable-Length Inputs**: When working with text or sequential data of different lengths.
- **Processing Complex Data Structures**: When each sample has multiple components (e.g., dictionaries or nested structures).
- **Applying Custom Preprocessing**: For special transformations or batch-level operations.

By default, the `DataLoader` uses `default_collate()`
from `torch.utils.data._utils.collate`, which:

- Stacks tensors along the first dimension.
- Converts Python lists and tuples into tensors.
- Leaves `None` values as they are.

Custom collation functions may be implemented and specified via the `collate_fn` parameter of a DataLoader. A common reason for implementing cutom collation functions is to ensure that all data in a batch is the same length / size.

Example: collation of a variable-length data set:

```
from torch.nn.utils.rnn import pad_sequence

def collate_variable_length(batch):
    # collates variable-length sequences into a padded batch.
    batch = [torch.tensor(sample) for sample in batch]  # convert to tensors
    return pad_sequence(batch, batch_first=True, padding_value=0)

# example dataset with variable-length sequences
class VariableLengthDataset(Dataset):
    def __init__(self):
        self.data = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]

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

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

dataset = VariableLengthDataset()
dataloader = DataLoader(dataset, batch_size=2, collate_fn=collate_variable_length)

for batch in dataloader:
    print(batch)
```

Data Transformations

PyTorch provides the `torchvision.transforms` module to apply transformations to the data within a `Dataset`.

Common Transformations:

- `transforms.Resize((h, w))`: Resizes images.
- `transforms.ToTensor()`: Converts images to PyTorch tensors.
- `transforms.Normalize(mean, std)`: Normalizes pixel values.
- `transforms.RandomHorizontalFlip()`: Randomly flips images.

Example:

```
from torchvision import transforms

# define a combined transformation composed of multiple simpler ones
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])

# apply o data set via the “transform” parameter
dataset = CustomImageDataset(csv_file="data/labels.csv", root_dir="data/images",
transform=transform)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Standard Datasets

PyTorch provides standard datasets through `torchvision.datasets`.

Example:

```
from torchvision import datasets

mnist_dataset = datasets.MNIST(root="data", train=True,
transform=transforms.ToTensor(), download=True)
mnist_loader = DataLoader(mnist_dataset, batch_size=64, shuffle=True)
```

Other Available Datasets

- `datasets.CIFAR10`
- `datasets.ImageNet`
- `datasets.FashionMNIST`
- `datasets.COCO`

Tasks

1. **Load and explore a built-in PyTorch data set**
   - Load the **MNIST** dataset using `torchvision.datasets.MNIST` with `train=True` and `train=False`.
   - Normalize the dataset so that the data values are from [-1, 1]
   - Use `DataLoader` to create an iterable for training and testing datasets (batch size = 32).
   - Retrieve a batch of images and labels, and visualize the first 10 images with their labels using `matplotlib`.

Hint: use `next(iter(dataloader))` to extract a batch.

2. **Create a custom dataset from CSV**
   - Download or generate a CSV file containing a dataset (e.g., `iris.csv` or any numerical dataset).
   - Implement a custom PyTorch `Dataset` from *iris.csv* to:
     - Load the CSV file.
     - Convert numerical columns into tensors.
     - Separate features and labels.
   - Create a `DataLoader` for mini-batch processing.
   - Print the shape of a single batch.

3. **Apply transformations to the data sets**
   - Load the **CIFAR-10** dataset using `torchvision.datasets.CIFAR10`.
   - Apply the following transformations using `transforms.Compose`:
     - Resize images to **64×64**.
     - Convert images to tensors.
     - Normalize them.
     - Apply **random horizontal flipping**.

    - Apply **random rotation**.
- Visualize some of the transformed images.

4. Implement a custom Data Loader for text data

- Implement a `Dataset` class from the *sentiment.txt* file that:
    - Reads text data.
    - Tokenizes sentences (use `nltk.word_tokenize()` or `torchtext.transforms`).
    - Converts words into tensor indices into the text vocabulary.
    - Converts labels into numerical values.
- Use a `DataLoader` to create batches. Use padding in `collate_fn` for batching variable-length sequences.

5. Combine multiple datasets

- Load both **MNIST** and **FashionMNIST** datasets.
- Ensure that both datasets are **normalized and resized** to the same shape.
- Merge them into a single dataset. Use `ConcatDataset([dataset1, dataset2])` to merge datasets. Ensure both datasets have the same transformations applied.
- Create a `DataLoader` for the merged dataset.
- Retrieve a batch and visualize images from both datasets.

6. Handling variable-size data using a custom *collate* function

- Load the **CIFAR-10** dataset.
- Write a custom `collate_fn` that:
    - Groups images of similar sizes into the same batch.
    - Uses padding to ensure all images in a batch have the same dimensions.
- Pass this function to `DataLoader` and verify batch sizes.

7. Create a custom data set that loads multimodal data (images and text)

- Implement a custom collate function to combine image and text data into data batches
- The training dataset should consist in (image, text, label) instances