Chapter 1

An Introduction to PyTorch

1.1 A Fun Example

Efficient machine learning processes data in batches, and our model will expect a batch of data.

We use PyTorch's `unsqueeze()` function to add a dimension to our tensor and create a batch of size 1.

The use of `model.to(device)` and `batch.to(device)` sends our model and input data to the GPU if available, and executing `model(batch.to(device))` runs our classifier.

Chapter 2

Tensors

2.1 Creating Tensors

Use torch.arange() when the step size is known. Use torch.linspace() when the number of elements is known. You can use torch.tensor() to create tensors from array-like structures such as lists, NumPy arrays, tuples, and sets. To convert existing tensors to NumPy arrays and lists, use the torch.numpy() and torch.tolist() functions, respectively.

2.1.1 Data Types

To reduce space complexity, you may sometimes want to reuse memory and overwrite tensor values using in-place operations. To perform in-place operations, append the underscore (_) postfix to the function name. For example, the function y.add_(x) adds x to y, but the results will be stored in y.

2.1.2 Creating Tensors from Random Samples

Table: Random sampling functions

2.1.3 Creating Tensors Like Other Tensors

You may want to create and initialize a tensor that has similar properties to another tensor, including the dtype, device, and layout properties to facilitate calculations. Many of the tensor creation operations have a similarity function that allows you to easily do this. The similarity functions will have the postfix _like. For example, torch.empty_like(tensor_a) will create an empty tensor with the dtype, device, and layout properties of tensor_a. Some examples of similarity functions include empty_like(), zeros_like(), ones_like(), full_like(), rand_like(), randn_like(), and rand_int_like().

2.2 Tensor Operations

2.2.1 Indexing, Slicing, Combining, and Splitting Tensors

The following key distinctions and best practices are important to keep in mind:

- item() is an important and commonly used function to return the Python number from a tensor containing a single value.
- Use view() instead of reshape() for reshaping tensors in most cases. Using reshape() may cause the tensor to be copied, depending on its layout in memory. view() ensures that it will not be copied.
- Using x.T or x.t() is a simple way to transpose 1D or 2D tensors. Use transpose() when dealing with multidimensional tensors.

Table 2.1: Tensor creation functions

Function	Description
torch.tensor(data, dtype=None, device=None, requires_grad=False, pin_memory=False)	Creates a tensor from an existing data structure
torch.empty(*size, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)	Creates a tensor from uninitialized elements based on the random state of values in memory
torch.zeros(*size, out=None, dtype=None, layout=torch.strided, evice=None, requires_grad=False)	Creates a tensor with all elements initialized to 0.0
torch.ones(*size, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)	Creates a tensor with all elements initialized to 1.0
torch.arange(start=0, end, step=1, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)	Creates a 1D tensor of values over a range with a common step value
torch.linspace(start, end, steps=100, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)	Creates a 1D tensor of linearly spaced points between the start and end
torch.logspace(start, end, steps=100, base=10.0, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)	Creates a 1D tensor of logarithmically spaced points between the start and end
torch.eye(n, m=None, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)	Creates a 2D tensor with ones on the diagonal and zeros everywhere else
torch.full(size, fill_value, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)	Creates a tensor filled with fill_value
torch.load(f)	Loads a tensor from a serialized pickle file
torch.save(f)	Saves a tensor to a serialized pickle file

- The torch.squeeze() function is used often in deep learning to remove an unused dimension. For example, a batch of images with a single image can be reduced from 4D to 3D using squeeze().
- The torch.unsqueeze() function is often used in deep learning to add a dimension of size 1. Since most PyTorch models expect a batch of data as an input, you could apply unsqueeze() when you only have one data sample. For example, you can pass a 3D image into torch.unsqueeze() to create a batch of one image.

Table: Indexing, slicing, combining, and splitting operations lists some commonly used functions to manipulate tensor elements.

2.2.2 Tensor Operations for Mathematics

Table: Pointwise operations list some commonly used pointwise operations Three different syntaxes can be used for most tensor operations. Tensors support operator overloading, so you can use operators directly, as in z = x + y. Although you can also use PyTorch functions such as torch.add() to do the same thing, this is less common. Lastly, you can perform in-place operations using the underscore (_) postfix. The function y.add_(x) achieves the same results, but they'll be stored in y.

Comparison functions seem pretty straightforward; however, there are a few key points to keep in mind. Common pitfalls include the following:

- The torch.eq() function or == returns a tensor of the same size with a Boolean result for each element. The torch.equal() function tests if the tensors are the same size, and if all elements within the tensor are equal then it returns a single Boolean value.
- The function torch.allclose() also returns a single Boolean value if all elements are close to a specified value.

Tabel: Spectral and other math operations lists some built-in operations for spectrum analysis and other mathematical operations.

Chapter 3

Deep Learning Development with PyTorch

3.1 The Overall Process

3.2 Data Preparation

3.2.1 Data Loading

PyTorch provides powerful built-in classes and utilities, such as the Dataset, DataLoader, and Sampler classes, for loading various types of data. The Dataset class defines how to access and preprocess data from a file or data sources. The Sampler class defines how to sample data from a dataset in order to create batches, while the DataLoader class combines a dataset with a sampler and allows you to iterate over a set of batches.

PyTorch libraries such as Torchvision and Torchtext also provide classes to support specialized data like computer vision and natural language data. The torchvision.datasets module is a good example of how to utilize built-in classes to load data. The torchvision.datasets module provides a number of subclasses to load image data from popular academic datasets.

3.2.2 Data Transforms

The data might need to be adjusted before it is passed into the NN model for training and testing. These adjustments are accomplished by applying transforms. **The beauty of using transforms in PyTorch is that you can define a sequence of transforms and apply it when the data is accessed.**

The transforms are passed to the dataset class during instantiation and become part of the dataset object. The transforms are applied whenever the dataset object is accessed, returning a new result consisting of the transformed data.

3.2.3 Data Batching

When you train your model, you will want to pass in small batches of data at each iteration. Sending data in batches not only allows more efficient training but also takes advantage of the parallel nature of GPUs to accelerate training.

Batch processing can easily be implemented using the torch.utils.data.DataLoader class.

The dataloader object combines a dataset and a sampler, and provides an iterable over the given dataset. In other words, your training loop can use this object to sample your dataset and apply transforms one batch at a time instead of applying them for the complete dataset at once. This considerably improves efficiency and speed when training and testing models.

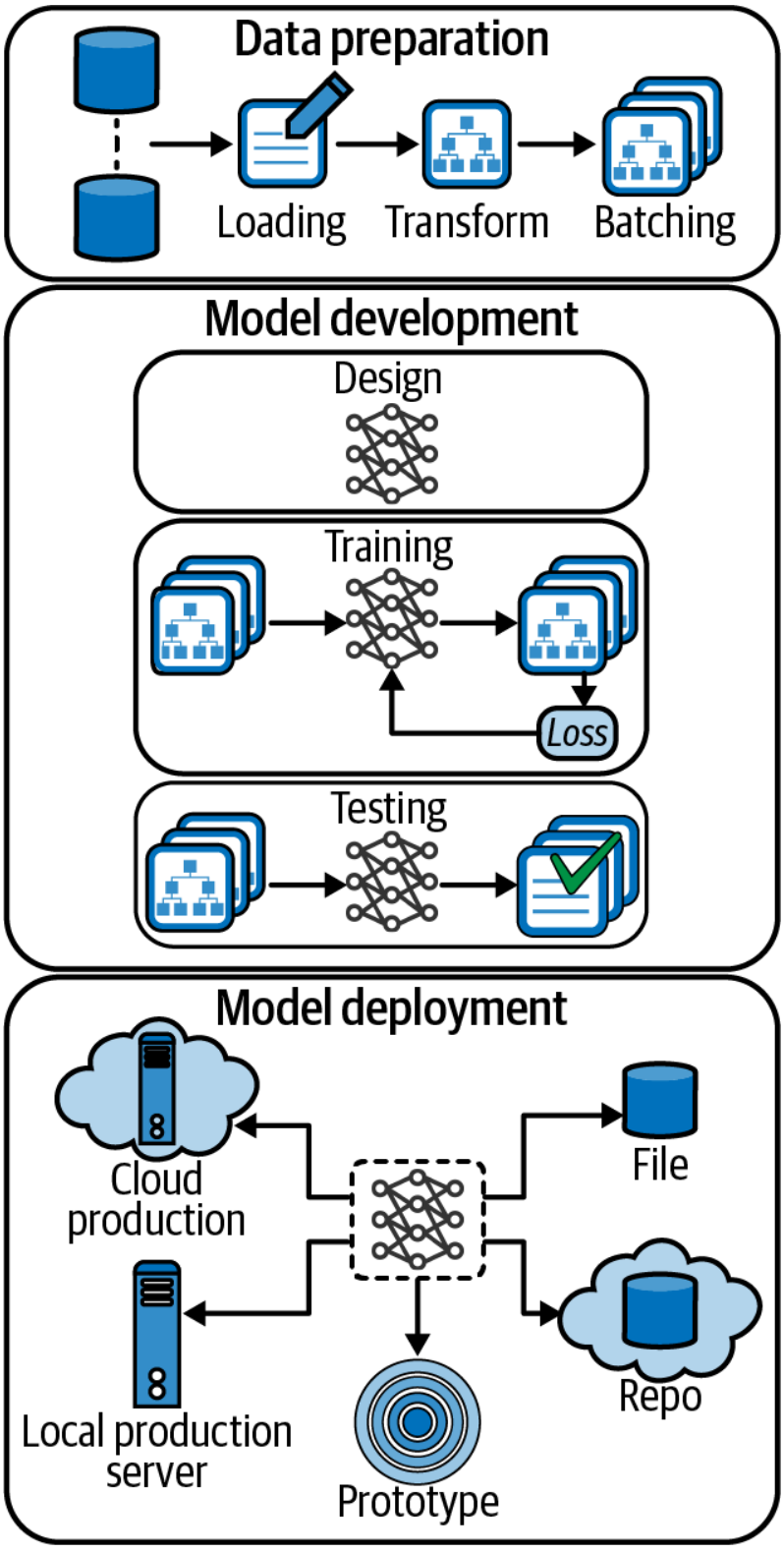

Figure 3.1: The basic deep learning development process

We need to use iter() to cast the trainloader to an iterator and then use next() to iterate over the data one more time. This is only necessary when accessing one batch. As we'll see later, our training loops will access the dataloader directly without the need for iter() and next().

3.2.4 General Data Preparation (torch.utils.data)

You can use PyTorch to prepare other types of data as well. PyTorch libraries such as Torchtext and Torchaudio provide dataset and dataloader classes for text and audio data, and new external libraries are being developed all the time.

PyTorch also provides a submodule called torch.utils.data that you can use to create your own dataset and dataloader classes like the ones you saw in Torchvision. It consists of Dataset, Sampler, and DataLoader classes.

Dataset classes

PyTorch supports map- and iterable-style dataset classes. A map-style dataset is derived from the abstract class torch.utils.data.Dataset. It implements the getitem() and len() functions, and represents a map from (possibly nonintegral) indices/keys to data samples. For example, such a dataset, when accessed with dataset[idx], could read the idx-th image and its corresponding label from a folder on the disk. Map-style datasets are more commonly used than iterable-style datasets, and all datasets that represent a map made from keys or data samples should use this subclass.

> **TIP**
>
> The simplest way to create your own dataset class is to subclass the map-style torch.utils.data.Dataset class and override the `getitem()` and `len()` functions with your own code.

All subclasses should overwrite getitem(), which fetches a data sample for a given key. Subclasses can also optionally overwrite len(), which returns the size of the dataset by many Sampler implementations and the default options of DataLoader.

An iterable-style dataset, on the other hand, is derived from the torch.utils.data.IterableDataset abstract class. It implements the iter() protocol and represents an iterable over data samples. This type of dataset is typically used when reading data from a database or a remote server, as well as data generated in real time. Iterable datasets are useful when random reads are expensive or uncertain, and when the batch size depends on fetched data.

PyTorch's torch.utils.data submodule also provides dataset operations to convert, combine, or split dataset objects. These operations include the following:

- `TensorDataset(tensors)`: Creates a dataset object from a tensor
- `ConcatDataset(datasets)`: Creates a dataset from multiple datasets
- `ChainDataset(datasets)`: Chains multiple IterableDatasets
- `Subset(dataset, indices)`: Creates a subset of a dataset from specified indices

Sampler classes

In addition to dataset classes PyTorch also provides sampler classes, which offer a way to iterate over indices of dataset samples. Sampler are derived from the torch.utils.data.Sampler base class.

Every Sampler subclass needs to implement an iter() method to provide a way to iterate over indices of dataset elements and a len() method that returns the length of the returned iterators. Table 3.1 provides a list of available samplers for your reference.

Samplers are usually not used directly. They are often passed to dataloaders to define the way the dataloader samples the dataset.

Table 3.1: Dataset samplers (torch.utils.data)

Sampler	Description
SequentialSampler(data_source)	Samples data in sequence
RandomSampler(data_source, replacement=False, num_samples=None, generator=None)	Samples data randomly
SubsetRandomSampler(indices, generator=None)	Samples data randomly from a subset of the dataset
WeightedRandomSampler(weights, num_samples, replacement=True, generator=None)	Samples randomly from a weighted distribution
BatchSampler(sampler, batch_size, drop_last)	Returns a batch of samples
distributed.DistributedSampler(dataset, num_replicas=None, rank=None, shuffle=True, seed=0)	Samples across distributed datasets

DataLoader classes

The Dataset class returns a dataset object that includes data and information about the data. The Sampler class returns the actual data itself in a specified or random fashion. The Data Loader class combines a dataset with a sampler and returns an iterable.

The dataset and sampler objects are not iterables, meaning you cannot run a for loop on them. The dataloader object solves this problem.