12

Parallelizing Neural Network Training with PyTorch

In this chapter, we will move on from the mathematical foundations of machine learning and deep learning to focus on PyTorch. PyTorch is one of the most popular deep learning libraries currently available, and it lets us implement **neural networks** (**NNs**) much more efficiently than any of our previous NumPy implementations. In this chapter, we will start using PyTorch and see how it brings significant benefits to training performance.

This chapter will begin the next stage of our journey into machine learning and deep learning, and we will explore the following topics:

- How PyTorch improves training performance
- Working with PyTorch's `Dataset` and `DataLoader` to build input pipelines and enable efficient model training
- Working with PyTorch to write optimized machine learning code
- Using the `torch.nn` module to implement common deep learning architectures conveniently
- Choosing activation functions for artificial NNs

PyTorch and training performance

PyTorch can speed up our machine learning tasks significantly. To understand how it can do this, let's begin by discussing some of the performance challenges we typically run into when we execute expensive calculations on our hardware. Then, we will take a high-level look at what PyTorch is and what our learning approach will be in this chapter.

Performance challenges

The performance of computer processors has, of course, been continuously improving in recent years. That allows us to train more powerful and complex learning systems, which means that we can improve the predictive performance of our machine learning models. Even the cheapest desktop computer hardware that's available right now comes with processing units that have multiple cores.

In the previous chapters, we saw that many functions in scikit-learn allow us to spread those computations over multiple processing units. However, by default, Python is limited to execution on one core due to the **global interpreter lock** (**GIL**). So, although we indeed take advantage of Python's multiprocessing library to distribute our computations over multiple cores, we still have to consider that the most advanced desktop hardware rarely comes with more than 8 or 16 such cores.

You will recall from *Chapter 11*, *Implementing a Multilayer Artificial Neural Network from Scratch*, that we implemented a very simple **multilayer perceptron** (**MLP**) with only one hidden layer consisting of 100 units. We had to optimize approximately 80,000 weight parameters ([784*100 + 100] + [100 * 10] + 10 = 79,510) for a very simple image classification task. The images in MNIST are rather small (28×28), and we can only imagine the explosion in the number of parameters if we wanted to add additional hidden layers or work with images that have higher pixel densities. Such a task would quickly become unfeasible for a single processing unit. The question then becomes, how can we tackle such problems more effectively?

The obvious solution to this problem is to use **graphics processing units** (**GPUs**), which are real workhorses. You can think of a graphics card as a small computer cluster inside your machine. Another advantage is that modern GPUs are great value compared to the state-of-the-art **central processing units** (**CPUs**), as you can see in the following overview:

Specifications	Intel® Core™ i9-11900KB Processor	NVIDIA GeForce® RTX™ 3080 Ti
Base Clock Frequency	3.3 GHz	1.37 GHz
Cores	16 (32 threads)	10240
Memory Bandwidth	45.8 GB/s	912.1 GB/s
Floating-Point Calculations	742 GFLOPS	34.10 TFLOPS
Cost	~ $540.00	~ $1200.00

Figure 12.1: Comparison of a state-of-the-art CPU and GPU

The sources for the information in *Figure 12.1* are the following websites (date accessed: July 2021):

- `https://ark.intel.com/content/www/us/en/ark/products/215570/intel-core-i9-11900kb-processor-24m-cache-up-to-4-90-ghz.html`
- `https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080-3080ti/`

At 2.2 times the price of a modern CPU, we can get a GPU that has 640 times more cores and is capable of around 46 times more floating-point calculations per second. So, what is holding us back from utilizing GPUs for our machine learning tasks? The challenge is that writing code to target GPUs is not as simple as executing Python code in our interpreter. There are special packages, such as CUDA and OpenCL, that allow us to target the GPU. However, writing code in CUDA or OpenCL is probably not the most convenient way to implement and run machine learning algorithms. The good news is that this is what PyTorch was developed for!

What is PyTorch?

PyTorch is a scalable and multiplatform programming interface for implementing and running machine learning algorithms, including convenience wrappers for deep learning. PyTorch was primarily developed by the researchers and engineers from the **Facebook AI Research** (**FAIR**) lab. Its development also involves many contributions from the community. PyTorch was initially released in September 2016 and is free and open source under the modified BSD license. Many machine learning researchers and practitioners from academia and industry have adapted PyTorch to develop deep learning solutions, such as Tesla Autopilot, Uber's Pyro, and Hugging Face's Transformers (`https://pytorch.org/ecosystem/`).

To improve the performance of training machine learning models, PyTorch allows execution on CPUs, GPUs, and XLA devices such as TPUs. However, its greatest performance capabilities can be discovered when using GPUs and XLA devices. PyTorch supports CUDA-enabled and ROCm GPUs officially. PyTorch's development is based on the Torch library (`www.torch.ch`). As its name implies, the Python interface is the primary development focus of PyTorch.

PyTorch is built around a computation graph composed of a set of nodes. Each node represents an operation that may have zero or more inputs or outputs. PyTorch provides an imperative programming environment that evaluates operations, executes computation, and returns concrete values immediately. Hence, the computation graph in PyTorch is defined implicitly, rather than constructed in advance and executed after.

Mathematically, tensors can be understood as a generalization of scalars, vectors, matrices, and so on. More concretely, a scalar can be defined as a rank-0 tensor, a vector can be defined as a rank-1 tensor, a matrix can be defined as a rank-2 tensor, and matrices stacked in a third dimension can be defined as rank-3 tensors. Tensors in PyTorch are similar to NumPy's arrays, except that tensors are optimized for automatic differentiation and can run on GPUs.

To make the concept of a tensor clearer, consider *Figure 12.2*, which represents tensors of ranks 0 and 1 in the first row, and tensors of ranks 2 and 3 in the second row:

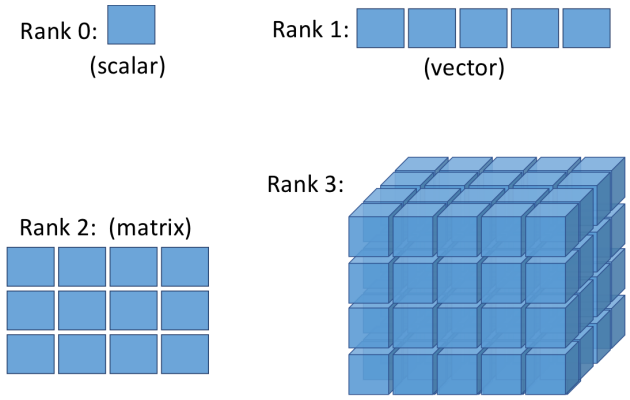

Figure 12.2: Different types of tensor in PyTorch

Now that we know what PyTorch is, let's see how to use it.

How we will learn PyTorch

First, we are going to cover PyTorch's programming model, in particular, creating and manipulating tensors. Then, we will see how to load data and utilize the `torch.utils.data` module, which will allow us to iterate through a dataset efficiently. In addition, we will discuss the existing, ready-to-use datasets in the `torch.utils.data.Dataset` submodule and learn how to use them.

After learning about these basics, the PyTorch neural network `torch.nn` module will be introduced. Then, we will move forward to building machine learning models, learn how to compose and train the models, and learn how to save the trained models on disk for future evaluation.

First steps with PyTorch

In this section, we will take our first steps in using the low-level PyTorch API. After installing PyTorch, we will cover how to create tensors in PyTorch and different ways of manipulating them, such as changing their shape, data type, and so on.

Installing PyTorch

To install PyTorch, we recommend consulting the latest instructions on the official `https://pytorch.org` website. Below, we will outline the basic steps that will work on most systems.

Depending on how your system is set up, you can typically just use Python's `pip` installer and install PyTorch from PyPI by executing the following from your terminal:

```
pip install torch torchvision
```

This will install the latest *stable* version, which is 1.9.0 at the time of writing. To install the 1.9.0 version, which is guaranteed to be compatible with the following code examples, you can modify the preceding command as follows:

```
pip install torch==1.9.0 torchvision==0.10.0
```

If you want to use GPUs (recommended), you need a compatible NVIDIA graphics card that supports CUDA and cuDNN. If your machine satisfies these requirements, you can install PyTorch with GPU support, as follows:

```
pip install torch==1.9.0+cu111 torchvision==0.10.0+cu111 -f https://download.
pytorch.org/whl/torch_stable.html
```

for CUDA 11.1 or:

```
pip install torch==1.9.0 torchvision==0.10.0\  -f https://download.pytorch.org/
whl/torch_stable.html
```

for CUDA 10.2 as of the time of writing.

As macOS binaries don't support CUDA, you can install from source: `https://pytorch.org/get-started/locally/#mac-from-source`.

For more information about the installation and setup process, please see the official recommendations at `https://pytorch.org/get-started/locally/`.

Note that PyTorch is under active development; therefore, every couple of months, new versions are released with significant changes. You can verify your PyTorch version from your terminal, as follows:

```
python -c 'import torch; print(torch.__version__)'
```

Troubleshooting your installation of PyTorch

If you experience problems with the installation procedure, read more about system- and platform-specific recommendations that are provided at `https://pytorch.org/get-started/locally/`. Note that all the code in this chapter can be run on your CPU; using a GPU is entirely optional but recommended if you want to fully enjoy the benefits of PyTorch. For example, while training some NN models on a CPU could take a week, the same models could be trained in just a few hours on a modern GPU. If you have a graphics card, refer to the installation page to set it up appropriately. In addition, you may find this setup guide helpful, which explains how to install the NVIDIA graphics card drivers, CUDA, and cuDNN on Ubuntu (not required but recommended requirements for running PyTorch on a GPU): `https://sebastianraschka.com/pdf/books/dlb/appendix_h_cloud-computing.pdf`. Furthermore, as you will see in *Chapter 17*, *Generative Adversarial Networks for Synthesizing New Data*, you can also train your models using a GPU for free via Google Colab.

Creating tensors in PyTorch

Now, let's consider a few different ways of creating tensors, and then see some of their properties and how to manipulate them. Firstly, we can simply create a tensor from a list or a NumPy array using the `torch.tensor` or the `torch.from_numpy` function as follows:

```
>>> import torch
>>> import numpy as np
>>> np.set_printoptions(precision=3)
>>> a = [1, 2, 3]
>>> b = np.array([4, 5, 6], dtype=np.int32)
>>> t_a = torch.tensor(a)
>>> t_b = torch.from_numpy(b)
>>> print(t_a)
>>> print(t_b)
tensor([1, 2, 3])
tensor([4, 5, 6], dtype=torch.int32)
```

This resulted in tensors `t_a` and `t_b`, with their properties, `shape=(3,)` and `dtype=int32`, adopted from their source. Similar to NumPy arrays, we can also see these properties:

```
>>> t_ones = torch.ones(2, 3)
>>> t_ones.shape
torch.Size([2, 3])
>>> print(t_ones)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

Finally, creating a tensor of random values can be done as follows:

```
>>> rand_tensor = torch.rand(2,3)
>>> print(rand_tensor)
tensor([[0.1409, 0.2848, 0.8914],
        [0.9223, 0.2924, 0.7889]])
```

Manipulating the data type and shape of a tensor

Learning ways to manipulate tensors is necessary to make them compatible for input to a model or an operation. In this section, you will learn how to manipulate tensor data types and shapes via several PyTorch functions that cast, reshape, transpose, and squeeze (remove dimensions).

The `torch.to()` function can be used to change the data type of a tensor to a desired type:

```
>>> t_a_new = t_a.to(torch.int64)
>>> print(t_a_new.dtype)
torch.int64
```

See `https://pytorch.org/docs/stable/tensor_attributes.html` for all other data types.

As you will see in upcoming chapters, certain operations require that the input tensors have a certain number of dimensions (that is, rank) associated with a certain number of elements (shape). Thus, we might need to change the shape of a tensor, add a new dimension, or squeeze an unnecessary dimension. PyTorch provides useful functions (or operations) to achieve this, such as `torch.transpose()`, `torch.reshape()`, and `torch.squeeze()`. Let's take a look at some examples:

- Transposing a tensor:

```
>>> t = torch.rand(3, 5)
>>> t_tr = torch.transpose(t, 0, 1)
>>> print(t.shape, ' --> ', t_tr.shape)
torch.Size([3, 5])  -->  torch.Size([5, 3])
```

- Reshaping a tensor (for example, from a 1D vector to a 2D array):

```
>>> t = torch.zeros(30)
>>> t_reshape = t.reshape(5, 6)
>>> print(t_reshape.shape)
torch.Size([5, 6])
```

- Removing the unnecessary dimensions (dimensions that have size 1, which are not needed):

```
>>> t = torch.zeros(1, 2, 1, 4, 1)
>>> t_sqz = torch.squeeze(t, 2)
>>> print(t.shape, ' --> ', t_sqz.shape)
torch.Size([1, 2, 1, 4, 1])  -->  torch.Size([1, 2, 4, 1])
```

Applying mathematical operations to tensors

Applying mathematical operations, in particular linear algebra operations, is necessary for building most machine learning models. In this subsection, we will cover some widely used linear algebra operations, such as element-wise product, matrix multiplication, and computing the norm of a tensor.

First, let's instantiate two random tensors, one with uniform distribution in the range [–1, 1) and the other with a standard normal distribution:

```
>>> torch.manual_seed(1)
>>> t1 = 2 * torch.rand(5, 2) - 1
>>> t2 = torch.normal(mean=0, std=1, size=(5, 2))
```

Note that `torch.rand` returns a tensor filled with random numbers from a uniform distribution in the range of [0, 1).

Notice that `t1` and `t2` have the same shape. Now, to compute the element-wise product of `t1` and `t2`, we can use the following:

```
>>> t3 = torch.multiply(t1, t2)
>>> print(t3)
tensor([[ 0.4426, -0.3114],
        [ 0.0660, -0.5970],
        [ 1.1249,  0.0150],
        [ 0.1569,  0.7107],
        [-0.0451, -0.0352]])
```

To compute the mean, sum, and standard deviation along a certain axis (or axes), we can use `torch.mean()`, `torch.sum()`, and `torch.std()`. For example, the mean of each column in `t1` can be computed as follows:

```
>>> t4 = torch.mean(t1, axis=0)
>>> print(t4)
tensor([-0.1373,  0.2028])
```

The matrix-matrix product between `t1` and `t2` (that is, $t_1 \times t_2^T$, where the superscript T is for transpose) can be computed by using the `torch.matmul()` function as follows:

```
>>> t5 = torch.matmul(t1, torch.transpose(t2, 0, 1))
>>> print(t5)
tensor([[ 0.1312,  0.3860, -0.6267, -1.0096, -0.2943],
        [ 0.1647, -0.5310,  0.2434,  0.8035,  0.1980],
        [-0.3855, -0.4422,  1.1399,  1.5558,  0.4781],
        [ 0.1822, -0.5771,  0.2585,  0.8676,  0.2132],
        [ 0.0330,  0.1084, -0.1692, -0.2771, -0.0804]])
```

On the other hand, computing $t_1^T \times t_2$ is performed by transposing `t1`, resulting in an array of size 2×2:

```
>>> t6 = torch.matmul(torch.transpose(t1, 0, 1), t2)
>>> print(t6)
tensor([[ 1.7453,  0.3392],
        [-1.6038, -0.2180]])
```

Finally, the `torch.linalg.norm()` function is useful for computing the L^p norm of a tensor. For example, we can calculate the L^2 norm of `t1` as follows:

```
>>> norm_t1 = torch.linalg.norm(t1, ord=2, dim=1)
>>> print(norm_t1)
tensor([0.6785, 0.5078, 1.1162, 0.5488, 0.1853])
```

To verify that this code snippet computes the L^2 norm of `t1` correctly, you can compare the results with the following NumPy function: `np.sqrt(np.sum(np.square(t1.numpy()), axis=1))`.

Split, stack, and concatenate tensors

In this subsection, we will cover PyTorch operations for splitting a tensor into multiple tensors, or the reverse: stacking and concatenating multiple tensors into a single one.

Assume that we have a single tensor, and we want to split it into two or more tensors. For this, PyTorch provides a convenient `torch.chunk()` function, which divides an input tensor into a list of equally sized tensors. We can determine the desired number of splits as an integer using the `chunks` argument to split a tensor along the desired dimension specified by the `dim` argument. In this case, the total size of the input tensor along the specified dimension must be divisible by the desired number of splits. Alternatively, we can provide the desired sizes in a list using the `torch.split()` function. Let's have a look at an example of both these options:

- Providing the number of splits:

  ```
  >>> torch.manual_seed(1)
  >>> t = torch.rand(6)
  >>> print(t)
  tensor([0.7576, 0.2793, 0.4031, 0.7347, 0.0293, 0.7999])
  >>> t_splits = torch.chunk(t, 3)
  >>> [item.numpy() for item in t_splits]
  [array([0.758, 0.279], dtype=float32),
   array([0.403, 0.735], dtype=float32),
   array([0.029, 0.8  ], dtype=float32)]
  ```

  In this example, a tensor of size 6 was divided into a list of three tensors each with size 2. If the tensor size is not divisible by the `chunks` value, the last chunk will be smaller.

- Providing the sizes of different splits:

  Alternatively, instead of defining the number of splits, we can also specify the sizes of the output tensors directly. Here, we are splitting a tensor of size `5` into tensors of sizes `3` and `2`:

```
>>> torch.manual_seed(1)
>>> t = torch.rand(5)
>>> print(t)
tensor([0.7576, 0.2793, 0.4031, 0.7347, 0.0293])
>>> t_splits = torch.split(t, split_size_or_sections=[3, 2])
>>> [item.numpy() for item in t_splits]
[array([0.758, 0.279, 0.403], dtype=float32),
 array([0.735, 0.029], dtype=float32)]
```

Sometimes, we are working with multiple tensors and need to concatenate or stack them to create a single tensor. In this case, PyTorch functions such as `torch.stack()` and `torch.cat()` come in handy. For example, let's create a 1D tensor, `A`, containing 1s with size `3`, and a 1D tensor, `B`, containing 0s with size `2`, and concatenate them into a 1D tensor, `C`, of size `5`:

```
>>> A = torch.ones(3)
>>> B = torch.zeros(2)
>>> C = torch.cat([A, B], axis=0)
>>> print(C)
tensor([1., 1., 1., 0., 0.])
```

If we create 1D tensors `A` and `B`, both with size `3`, then we can stack them together to form a 2D tensor, `S`:

```
>>> A = torch.ones(3)
>>> B = torch.zeros(3)
>>> S = torch.stack([A, B], axis=1)
>>> print(S)
tensor([[1., 0.],
        [1., 0.],
        [1., 0.]])
```

The PyTorch API has many operations that you can use for building a model, processing your data, and more. However, covering every function is outside the scope of this book, where we will focus on the most essential ones. For the full list of operations and functions, you can refer to the documentation page of PyTorch at `https://pytorch.org/docs/stable/index.html`.

Building input pipelines in PyTorch

When we are training a deep NN model, we usually train the model incrementally using an iterative optimization algorithm such as stochastic gradient descent, as we have seen in previous chapters.

As mentioned at the beginning of this chapter, `torch.nn` is a module for building NN models. In cases where the training dataset is rather small and can be loaded as a tensor into the memory, we can directly use this tensor for training. In typical use cases, however, when the dataset is too large to fit into the computer memory, we will need to load the data from the main storage device (for example, the hard drive or solid-state drive) in chunks, that is, batch by batch. (Note the use of the term "batch" instead of "mini-batch" in this chapter to stay close to the PyTorch terminology.) In addition, we may need to construct a data-processing pipeline to apply certain transformations and preprocessing steps to our data, such as mean centering, scaling, or adding noise to augment the training procedure and to prevent overfitting.

Applying preprocessing functions manually every time can be quite cumbersome. Luckily, PyTorch provides a special class for constructing efficient and convenient preprocessing pipelines. In this section, we will see an overview of different methods for constructing a PyTorch `Dataset` and `DataLoader`, and implementing data loading, shuffling, and batching.

Creating a PyTorch DataLoader from existing tensors

If the data already exists in the form of a tensor object, a Python list, or a NumPy array, we can easily create a dataset loader using the `torch.utils.data.DataLoader()` class. It returns an object of the `DataLoader` class, which we can use to iterate through the individual elements in the input dataset. As a simple example, consider the following code, which creates a dataset from a list of values from 0 to 5:

```
>>> from torch.utils.data import DataLoader
>>> t = torch.arange(6, dtype=torch.float32)
>>> data_loader = DataLoader(t)
```

We can easily iterate through a dataset entry by entry as follows:

```
>>> for item in data_loader:
...     print(item)
tensor([0.])
tensor([1.])
tensor([2.])
tensor([3.])
tensor([4.])
tensor([5.])
```

If we want to create batches from this dataset, with a desired batch size of `3`, we can do this with the `batch_size` argument as follows:

```
>>> data_loader = DataLoader(t, batch_size=3, drop_last=False)
>>> for i, batch in enumerate(data_loader, 1):
...     print(f'batch {i}:', batch)
batch 1: tensor([0., 1., 2.])
batch 2: tensor([3., 4., 5.])
```

This will create two batches from this dataset, where the first three elements go into batch #1, and the remaining elements go into batch #2. The optional `drop_last` argument is useful for cases when the number of elements in the tensor is not divisible by the desired batch size. We can drop the last non-full batch by setting `drop_last` to `True`. The default value for `drop_last` is `False`.

We can always iterate through a dataset directly, but as you just saw, `DataLoader` provides an automatic and customizable batching to a dataset.

Combining two tensors into a joint dataset

Often, we may have the data in two (or possibly more) tensors. For example, we could have a tensor for features and a tensor for labels. In such cases, we need to build a dataset that combines these tensors, which will allow us to retrieve the elements of these tensors in tuples.

Assume that we have two tensors, `t_x` and `t_y`. Tensor `t_x` holds our feature values, each of size `3`, and `t_y` stores the class labels. For this example, we first create these two tensors as follows:

```
>>> torch.manual_seed(1)
>>> t_x = torch.rand([4, 3], dtype=torch.float32)
>>> t_y = torch.arange(4)
```

Now, we want to create a joint dataset from these two tensors. We first need to create a `Dataset` class as follows:

```
>>> from torch.utils.data import Dataset
>>> class JointDataset(Dataset):
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
...     def __len__(self):
...         return len(self.x)
...
```

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

A custom `Dataset` class must contain the following methods to be used by the data loader later on:

- `__init__()`: This is where the initial logic happens, such as reading existing arrays, loading a file, filtering data, and so forth.
- `__getitem__()`: This returns the corresponding sample to the given index.

Then we create a joint dataset of `t_x` and `t_y` with the custom `Dataset` class as follows:

```
>>> from torch.utils.data import TensorDataset
>>> joint_dataset = TensorDataset(t_x, t_y)
```

Finally, we can print each example of the joint dataset as follows:

```
>>> for example in joint_dataset:
...     print('  x: ', example[0], '  y: ', example[1])
  x:  tensor([0.7576, 0.2793, 0.4031])   y:  tensor(0)
  x:  tensor([0.7347, 0.0293, 0.7999])   y:  tensor(1)
  x:  tensor([0.3971, 0.7544, 0.5695])   y:  tensor(2)
  x:  tensor([0.4388, 0.6387, 0.5247])   y:  tensor(3)
```

We can also simply utilize the `torch.utils.data.TensorDataset` class, if the second dataset is a labeled dataset in the form of tensors. So, instead of using our self-defined `Dataset` class, `JointDataset`, we can create a joint dataset as follows:

```
>>> joint_dataset = TensorDataset(t_x, t_y)
```

Note that a common source of error could be that the element-wise correspondence between the original features (x) and labels (y) might be lost (for example, if the two datasets are shuffled separately). However, once they are merged into one dataset, it is safe to apply these operations.

If we have a dataset created from the list of image filenames on disk, we can define a function to load the images from these filenames. You will see an example of applying multiple transformations to a dataset later in this chapter.

Shuffle, batch, and repeat

As was mentioned in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, when training an NN model using stochastic gradient descent optimization, it is important to feed training data as randomly shuffled batches. You have already seen how to specify the batch size using the `batch_size` argument of a data loader object. Now, in addition to creating batches, you will see how to shuffle and reiterate over the datasets. We will continue working with the previous joint dataset.

First, let's create a shuffled version data loader from the `joint_dataset` dataset:

```
>>> torch.manual_seed(1)
>>> data_loader = DataLoader(dataset=joint_dataset, batch_size=2, shuffle=True)
```

Here, each batch contains two data records (x) and the corresponding labels (y). Now we iterate through the data loader entry by entry as follows:

```
>>> for i, batch in enumerate(data_loader, 1):
...     print(f'batch {i}:', 'x:', batch[0],
              '\n         y:', batch[1])
batch 1: x: tensor([[0.4388, 0.6387, 0.5247],
        [0.3971, 0.7544, 0.5695]])
         y: tensor([3, 2])
batch 2: x: tensor([[0.7576, 0.2793, 0.4031],
        [0.7347, 0.0293, 0.7999]])
         y: tensor([0, 1])
```

The rows are shuffled without losing the one-to-one correspondence between the entries in `x` and `y`.

In addition, when training a model for multiple epochs, we need to shuffle and iterate over the dataset by the desired number of epochs. So, let's iterate over the batched dataset twice:

```
>>> for epoch in range(2):
>>>     print(f'epoch {epoch+1}')
>>>     for i, batch in enumerate(data_loader, 1):
...         print(f'batch {i}:', 'x:', batch[0],
                  '\n         y:', batch[1])
epoch 1
batch 1: x: tensor([[0.7347, 0.0293, 0.7999],
        [0.3971, 0.7544, 0.5695]])
         y: tensor([1, 2])
batch 2: x: tensor([[0.4388, 0.6387, 0.5247],
        [0.7576, 0.2793, 0.4031]])
         y: tensor([3, 0])
epoch 2
batch 1: x: tensor([[0.3971, 0.7544, 0.5695],
        [0.7576, 0.2793, 0.4031]])
         y: tensor([2, 0])
batch 2: x: tensor([[0.7347, 0.0293, 0.7999],
        [0.4388, 0.6387, 0.5247]])
         y: tensor([1, 3])
```

This results in two different sets of batches. In the first epoch, the first batch contains a pair of values `[y=1, y=2]`, and the second batch contains a pair of values `[y=3, y=0]`. In the second epoch, two batches contain a pair of values, `[y=2, y=0]` and `[y=1, y=3]` respectively. For each iteration, the elements within a batch are also shuffled.

Creating a dataset from files on your local storage disk

In this section, we will build a dataset from image files stored on disk. There is an image folder associated with the online content of this chapter. After downloading the folder, you should be able to see six images of cats and dogs in JPEG format.

This small dataset will show how building a dataset from stored files generally works. To accomplish this, we are going to use two additional modules: `Image` in `PIL` to read the image file contents and `transforms` in `torchvision` to decode the raw contents and resize the images.

The `PIL.Image` and `torchvision.transforms` modules provide a lot of additional and useful functions, which are beyond the scope of the book. You are encouraged to browse through the official documentation to learn more about these functions:

`https://pillow.readthedocs.io/en/stable/reference/Image.html` for `PIL.Image`

`https://pytorch.org/vision/stable/transforms.html` for `torchvision.transforms`

Before we start, let's take a look at the content of these files. We will use the `pathlib` library to generate a list of image files:

```
>>> import pathlib
>>> imgdir_path = pathlib.Path('cat_dog_images')
>>> file_list = sorted([str(path) for path in
... imgdir_path.glob('*.jpg')])
>>> print(file_list)
['cat_dog_images/dog-03.jpg', 'cat_dog_images/cat-01.jpg', 'cat_dog_images/cat-
02.jpg', 'cat_dog_images/cat-03.jpg', 'cat_dog_images/dog-01.jpg', 'cat_dog_
images/dog-02.jpg']
```

Next, we will visualize these image examples using Matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> import os
>>> from PIL import Image
>>> fig = plt.figure(figsize=(10, 5))
>>> for i, file in enumerate(file_list):
...     img = Image.open(file)
...     print('Image shape:', np.array(img).shape)
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
```

```
...     ax.imshow(img)
...     ax.set_title(os.path.basename(file), size=15)
>>> plt.tight_layout()
>>> plt.show()
Image shape: (900, 1200, 3)
Image shape: (900, 1200, 3)
Image shape: (900, 1200, 3)
Image shape: (900, 742, 3)
Image shape: (800, 1200, 3)
Image shape: (800, 1200, 3)
```

Figure 12.3 shows the example images:

Figure 12.3: Images of cats and dogs

Just from this visualization and the printed image shapes, we can already see that the images have different aspect ratios. If you print the aspect ratios (or data array shapes) of these images, you will see that some images are 900 pixels high and 1200 pixels wide (900×1200), some are 800×1200, and one is 900×742. Later, we will preprocess these images to a consistent size. Another point to consider is that the labels for these images are provided within their filenames. So, we extract these labels from the list of filenames, assigning label `1` to dogs and label `0` to cats:

```
>>> labels = [1 if 'dog' in
...               os.path.basename(file) else 0
...                       for file in file_list]
>>> print(labels)
[0, 0, 0, 1, 1, 1]
```

Now, we have two lists: a list of filenames (or paths of each image) and a list of their labels. In the previous section, you learned how to create a joint dataset from two arrays. Here, we will do the following:

```
>>> class ImageDataset(Dataset):
...     def __init__(self, file_list, labels):
...         self.file_list = file_list
```

```
...         self.labels = labels
...
...     def __getitem__(self, index):
...         file = self.file_list[index]
...         label = self.labels[index]
...         return file, label
...
...     def __len__(self):
...         return len(self.labels)

>>> image_dataset = ImageDataset(file_list, labels)
>>> for file, label in image_dataset:
...     print(file, label)


cat_dog_images/cat-01.jpg 0
cat_dog_images/cat-02.jpg 0
cat_dog_images/cat-03.jpg 0
cat_dog_images/dog-01.jpg 1
cat_dog_images/dog-02.jpg 1
cat_dog_images/dog-03.jpg 1
```

The joint dataset has filenames and labels.

Next, we need to apply transformations to this dataset: load the image content from its file path, decode the raw content, and resize it to a desired size, for example, 80×120. As mentioned before, we use the `torchvision.transforms` module to resize the images and convert the loaded pixels into tensors as follows:

```
>>> import torchvision.transforms as transforms
>>> img_height, img_width = 80, 120
>>> transform = transforms.Compose([
...     transforms.ToTensor(),
...     transforms.Resize((img_height, img_width)),
... ])
```

Now we update the `ImageDataset` class with the `transform` we just defined:

```
>>> class ImageDataset(Dataset):
...     def __init__(self, file_list, labels, transform=None):
...         self.file_list = file_list
...         self.labels = labels
...         self.transform = transform
...
```

```
...     def __getitem__(self, index):
...         img = Image.open(self.file_list[index])
...         if self.transform is not None:
...             img = self.transform(img)
...         label = self.labels[index]
...         return img, label
...
...     def __len__(self):
...         return len(self.labels)
>>>
>>> image_dataset = ImageDataset(file_list, labels, transform)
```

Finally, we visualize these transformed image examples using Matplotlib:

```
>>> fig = plt.figure(figsize=(10, 6))
>>> for i, example in enumerate(image_dataset):
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(example[0].numpy().transpose((1, 2, 0)))
...     ax.set_title(f'{example[1]}', size=15)
...
>>> plt.tight_layout()
>>> plt.show()
```

This results in the following visualization of the retrieved example images, along with their labels:

Figure 12.4: Images are labeled

The `__getitem__` method in the `ImageDataset` class wraps all four steps into a single function, including the loading of the raw content (images and labels), decoding the images into tensors, and resizing the images. The function then returns a dataset that we can iterate over and apply other operations that we learned about in the previous sections via a data loader, such as shuffling and batching.

Fetching available datasets from the torchvision.datasets library

The `torchvision.datasets` library provides a nice collection of freely available image datasets for training or evaluating deep learning models. Similarly, the `torchtext.datasets` library provides datasets for natural language. Here, we use `torchvision.datasets` as an example.

The `torchvision` datasets (`https://pytorch.org/vision/stable/datasets.html`) are nicely formatted and come with informative descriptions, including the format of features and labels and their type and dimensionality, as well as the link to the original source of the dataset. Another advantage is that these datasets are all subclasses of `torch.utils.data.Dataset`, so all the functions we covered in the previous sections can be used directly. So, let's see how to use these datasets in action.

First, if you haven't already installed `torchvision` together with PyTorch earlier, you need to install the `torchvision` library via `pip` from the command line:

```
pip install torchvision
```

You can take a look at the list of available datasets at `https://pytorch.org/vision/stable/datasets.html`.

In the following paragraphs, we will cover fetching two different datasets: CelebA (`celeb_a`) and the MNIST digit dataset.

Let's first work with the CelebA dataset (`http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`) with `torchvision.datasets.CelebA` (`https://pytorch.org/vision/stable/datasets.html#celeba`). The description of `torchvision.datasets.CelebA` provides some useful information to help us understand the structure of this dataset:

- The database has three subsets, `'train'`, `'valid'`, and `'test'`. We can select a specific subset or load all of them with the `split` parameter.
- The images are stored in `PIL.Image` format. And we can obtain a transformed version using a custom `transform` function, such as `transforms.ToTensor` and `transforms.Resize`.
- There are different types of targets we can use, including `'attributes'`, `'identity'`, and `'landmarks'`. `'attributes'` is 40 facial attributes for the person in the image, such as facial expression, makeup, hair properties, and so on; `'identity'` is the person ID for an image; and `'landmarks'` refers to the dictionary of extracted facial points, such as the position of the eyes, nose, and so on.

Next, we will call the `torchvision.datasets.CelebA` class to download the data, store it on disk in a designated folder, and load it into a `torch.utils.data.Dataset` object:

```
>>> import torchvision
>>> image_path = './'
>>> celeba_dataset = torchvision.datasets.CelebA(
...     image_path, split='train', target_type='attr', download=True
... )
1443490838/? [01:28<00:00, 6730259.81it/s]
26721026/? [00:03<00:00, 8225581.57it/s]
3424458/? [00:00<00:00, 14141274.46it/s]
6082035/? [00:00<00:00, 21695906.49it/s]
12156055/? [00:00<00:00, 12002767.35it/s]
2836386/? [00:00<00:00, 3858079.93it/s]
```

You may run into a `BadZipFile: File is not a zip file` error, or `RuntimeError: The daily quota of the file img_align_celeba.zip is exceeded and it can't be downloaded. This is a limitation of Google Drive and can only be overcome by trying again later`; it just means that Google Drive has a daily maximum quota that is exceeded by the CelebA files. To work around it, you can manually download the files from the source: `http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`. In the downloaded folder, `celeba/`, you can unzip the `img_align_celeba.zip` file. The `image_path` is the root of the downloaded folder, `celeba/`. If you have already downloaded the files once, you can simply set `download=False`. For additional information and guidance, we highly recommend to see accompanying code notebook at `https://github.com/rasbt/machine-learning-book/blob/main/ch12/ch12_part1.ipynb`.

Now that we have instantiated the datasets, let's check if the object is of the `torch.utils.data.Dataset` class:

```
>>> assert isinstance(celeba_dataset, torch.utils.data.Dataset)
```

As mentioned, the dataset is already split into train, test, and validation datasets, and we only load the train set. And we only use the `'attributes'` target. In order to see what the data examples look like, we can execute the following code:

```
>>> example = next(iter(celeba_dataset))
>>> print(example)
(<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=178x218 at
0x120C6C668>, tensor([0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1,
1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1]))
```

Note that the sample in this dataset comes in a tuple of `(PIL.Image, attributes)`. If we want to pass this dataset to a supervised deep learning model during training, we have to reformat it as a tuple of `(features tensor, label)`. For the label, we will use the `'Smiling'` category from the attributes as an example, which is the 31st element.

Finally, let's take the first 18 examples from it to visualize them with their `'Smiling'` labels:

```
>>> from itertools import islice
>>> fig = plt.figure(figsize=(12, 8))
>>> for i, (image, attributes) in islice(enumerate(celeba_dataset), 18):
...     ax = fig.add_subplot(3, 6, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(image)
...     ax.set_title(f'{attributes[31]}', size=15)
>>> plt.show()
```

The examples and their labels that are retrieved from `celeba_dataset` are shown in *Figure 12.5*:

Figure 12.5: Model predicts smiling celebrities

This was all we needed to do to fetch and use the CelebA image dataset.

Next, we will proceed with the second dataset from `torchvision.datasets.MNIST` (`https://pytorch.org/vision/stable/datasets.html#mnist`). Let's see how it can be used to fetch the MNIST digit dataset:

- The database has two partitions, `'train'` and `'test'`. We need to select a specific subset to load.
- The images are stored in `PIL.Image` format. And we can obtain a transformed version using a custom `transform` function, such as `transforms.ToTensor` and `transforms.Resize`.
- There are 10 classes for the target, from `0` to `9`.

Now, we can download the `'train'` partition, convert the elements to tuples, and visualize 10 examples:

```
>>> mnist_dataset = torchvision.datasets.MNIST(image_path, 'train',
download=True)
>>> assert isinstance(mnist_dataset, torch.utils.data.Dataset)
>>> example = next(iter(mnist_dataset))
>>> print(example)
(<PIL.Image.Image image mode=L size=28x28 at 0x126895B00>, 5)
>>> fig = plt.figure(figsize=(15, 6))
>>> for i, (image, label) in  islice(enumerate(mnist_dataset), 10):
...     ax = fig.add_subplot(2, 5, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(image, cmap='gray_r')
...     ax.set_title(f'{label}', size=15)
>>> plt.show()
```

The retrieved example handwritten digits from this dataset are shown as follows:

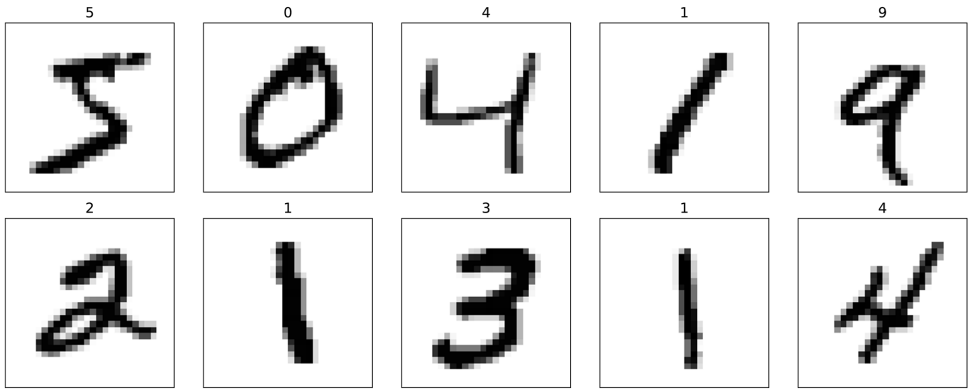

Figure 12.6: Correctly identifying handwritten digits

This concludes our coverage of building and manipulating datasets and fetching datasets from the `torchvision.datasets` library. Next, we will see how to build NN models in PyTorch.

Building an NN model in PyTorch

So far in this chapter, you have learned about the basic utility components of PyTorch for manipulating tensors and organizing data into formats that we can iterate over during training. In this section, we will finally implement our first predictive model in PyTorch. As PyTorch is a bit more flexible but also more complex than machine learning libraries such as scikit-learn, we will start with a simple linear regression model.

The PyTorch neural network module (torch.nn)

`torch.nn` is an elegantly designed module developed to help create and train NNs. It allows easy prototyping and the building of complex models in just a few lines of code.

To fully utilize the power of the module and customize it for your problem, you need to understand what it's doing. To develop this understanding, we will first train a basic linear regression model on a toy dataset without using any features from the `torch.nn` module; we will use nothing but the basic PyTorch tensor operations.

Then, we will incrementally add features from `torch.nn` and `torch.optim`. As you will see in the following subsections, these modules make building an NN model extremely easy. We will also take advantage of the dataset pipeline functionalities supported in PyTorch, such as `Dataset` and `DataLoader`, which you learned about in the previous section. In this book, we will use the `torch.nn` module to build NN models.

The most commonly used approach for building an NN in PyTorch is through `nn.Module`, which allows layers to be stacked to form a network. This gives us more control over the forward pass. We will see examples of building an NN model using the `nn.Module` class.

Finally, as you will see in the following subsections, a trained model can be saved and reloaded for future use.

Building a linear regression model

In this subsection, we will build a simple model to solve a linear regression problem. First, let's create a toy dataset in NumPy and visualize it:

```
>>> X_train = np.arange(10, dtype='float32').reshape((10, 1))
>>> y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0,
...                     6.3, 6.6,7.4, 8.0,
...                     9.0], dtype='float32')
>>> plt.plot(X_train, y_train, 'o', markersize=10)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.show()
```

As a result, the training examples will be shown in a scatterplot as follows:

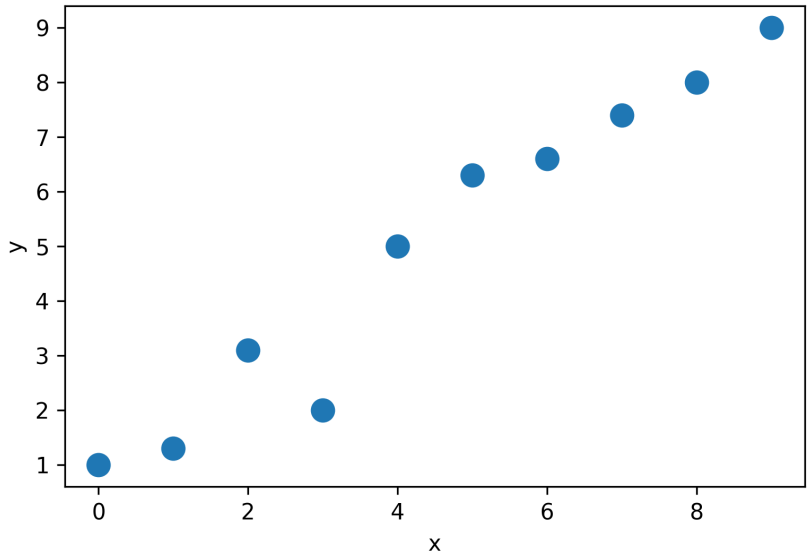

Figure 12.7: A scatterplot of the training examples

Next, we will standardize the features (mean centering and dividing by the standard deviation) and create a PyTorch `Dataset` for the training set and a corresponding `DataLoader`:

```
>>> from torch.utils.data import TensorDataset
>>> X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
>>> X_train_norm = torch.from_numpy(X_train_norm)
>>> y_train = torch.from_numpy(y_train).float()
>>> train_ds = TensorDataset(X_train_norm, y_train)
>>> batch_size = 1
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Here, we set a batch size of `1` for the `DataLoader`.

Now, we can define our model for linear regression as $z = wx + b$. Here, we are going to use the `torch.nn` module. It provides predefined layers for building complex NN models, but to start, you will learn how to define a model from scratch. Later in this chapter, you will see how to use those predefined layers.

For this regression problem, we will define a linear regression model from scratch. We will define the parameters of our model, `weight` and `bias`, which correspond to the weight and the bias parameters, respectively. Finally, we will define the `model()` function to determine how this model uses the input data to generate its output:

```
>>> torch.manual_seed(1)
>>> weight = torch.randn(1)
>>> weight.requires_grad_()
>>> bias = torch.zeros(1, requires_grad=True)
>>> def model(xb):
...     return xb @ weight + bias
```

After defining the model, we can define the loss function that we want to minimize to find the optimal model weights. Here, we will choose the **mean squared error** (**MSE**) as our loss function:

```
>>> def loss_fn(input, target):
...     return (input-target).pow(2).mean()
```

Furthermore, to learn the weight parameters of the model, we will use stochastic gradient descent. In this subsection, we will implement this training via the stochastic gradient descent procedure by ourselves, but in the next subsection, we will use the `SGD` method from the optimization package, `torch.optim`, to do the same thing.

To implement the stochastic gradient descent algorithm, we need to compute the gradients. Rather than manually computing the gradients, we will use PyTorch's `torch.autograd.backward` function. We will cover `torch.autograd` and its different classes and functions for implementing automatic differentiation in *Chapter 13*, *Going Deeper – The Mechanics of PyTorch*.

Now, we can set the learning rate and train the model for 200 epochs. The code for training the model against the batched version of the dataset is as follows:

```
>>> learning_rate = 0.001
>>> num_epochs = 200
>>> log_epochs = 10
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)
...         loss = loss_fn(pred, y_batch.long())
...         loss.backward()
...     with torch.no_grad():
...         weight -= weight.grad * learning_rate
...         bias -= bias.grad * learning_rate
...         weight.grad.zero_()
...         bias.grad.zero_()
...     if epoch % log_epochs==0:
```

```
...         print(f'Epoch {epoch}  Loss {loss.item():.4f}')
Epoch 0  Loss 5.1701
Epoch 10  Loss 30.3370
Epoch 20  Loss 26.9436
Epoch 30  Loss 0.9315
Epoch 40  Loss 3.5942
Epoch 50  Loss 5.8960
Epoch 60  Loss 3.7567
Epoch 70  Loss 1.5877
Epoch 80  Loss 0.6213
Epoch 90  Loss 1.5596
Epoch 100  Loss 0.2583
Epoch 110  Loss 0.6957
Epoch 120  Loss 0.2659
Epoch 130  Loss 0.1615
Epoch 140  Loss 0.6025
Epoch 150  Loss 0.0639
Epoch 160  Loss 0.1177
Epoch 170  Loss 0.3501
Epoch 180  Loss 0.3281
Epoch 190  Loss 0.0970
```

Let's look at the trained model and plot it. For the test data, we will create a NumPy array of values evenly spaced between 0 and 9. Since we trained our model with standardized features, we will also apply the same standardization to the test data:

```
>>> print('Final Parameters:', weight.item(), bias.item())
Final Parameters:  2.669806480407715 4.879569053649902
>>> X_test = np.linspace(0, 9, num=100, dtype='float32').reshape(-1, 1)
>>> X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
>>> X_test_norm = torch.from_numpy(X_test_norm)
>>> y_pred = model(X_test_norm).detach().numpy()
>>> fig = plt.figure(figsize=(13, 5))
>>> ax = fig.add_subplot(1, 2, 1)
>>> plt.plot(X_train_norm, y_train, 'o', markersize=10)
>>> plt.plot(X_test_norm, y_pred, '--', lw=3)
>>> plt.legend(['Training examples', 'Linear reg.'], fontsize=15)
>>> ax.set_xlabel('x', size=15)
>>> ax.set_ylabel('y', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()
```

Figure 12.8 shows a scatterplot of the training examples and the trained linear regression model:

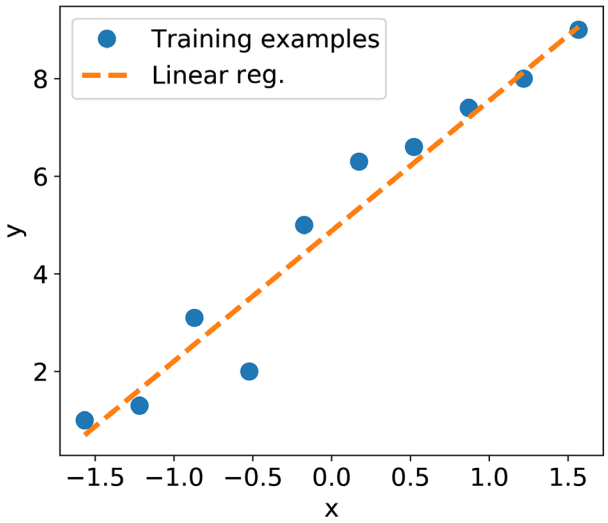

Figure 12.8: The linear regression model fits the data well

Model training via the torch.nn and torch.optim modules

In the previous example, we saw how to train a model by writing a custom loss function `loss_fn()` and applied stochastic gradient descent optimization. However, writing the loss function and gradient updates can be a repeatable task across different projects. The `torch.nn` module provides a set of loss functions, and `torch.optim` supports most commonly used optimization algorithms that can be called to update the parameters based on the computed gradients. To see how they work, let's create a new MSE loss function and a stochastic gradient descent optimizer:

```
>>> import torch.nn as nn
>>> loss_fn = nn.MSELoss(reduction='mean')
>>> input_size = 1
>>> output_size = 1
>>> model = nn.Linear(input_size, output_size)
>>> optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Note that here we use the `torch.nn.Linear` class for the linear layer instead of manually defining it.

Now, we can simply call the `step()` method of the `optimizer` to train the model. We can pass a batched dataset (such as `train_dl`, which was created in the previous example):

```
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         # 1. Generate predictions
...         pred = model(x_batch)[:, 0]
...         # 2. Calculate loss
...         loss = loss_fn(pred, y_batch)
...         # 3. Compute gradients
...         loss.backward()
...         # 4. Update parameters using gradients
...         optimizer.step()
...         # 5. Reset the gradients to zero
...         optimizer.zero_grad()
...     if epoch % log_epochs==0:
...         print(f'Epoch {epoch}  Loss {loss.item():.4f}')
```

After the model is trained, visualize the results and make sure that they are similar to the results of the previous method. To obtain the weight and bias parameters, we can do the following:

```
>>> print('Final Parameters:', model.weight.item(), model.bias.item())
Final Parameters: 2.646660089492798 4.883835315704346
```

Building a multilayer perceptron for classifying flowers in the Iris dataset

In the previous example, you saw how to build a model from scratch. We trained this model using stochastic gradient descent optimization. While we started our journey based on the simplest possible example, you can see that defining the model from scratch, even for such a simple case, is neither appealing nor good practice. PyTorch instead provides already defined layers through `torch.nn` that can be readily used as the building blocks of an NN model. In this section, you will learn how to use these layers to solve a classification task using the Iris flower dataset (identifying between three species of irises) and build a two-layer perceptron using the `torch.nn` module. First, let's get the data from `sklearn.datasets`:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> iris = load_iris()
```

```
>>> X = iris['data']
>>> y = iris['target']
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=1./3, random_state=1)
```

Here, we randomly select 100 samples (2/3) for training and 50 samples (1/3) for testing.

Next, we standardize the features (mean centering and dividing by the standard deviation) and create a PyTorch `Dataset` for the training set and a corresponding `DataLoader`:

```
>>> X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
>>> X_train_norm = torch.from_numpy(X_train_norm).float()
>>> y_train = torch.from_numpy(y_train)
>>> train_ds = TensorDataset(X_train_norm, y_train)
>>> torch.manual_seed(1)
>>> batch_size = 2
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Here, we set the batch size to 2 for the `DataLoader`.

Now, we are ready to use the `torch.nn` module to build a model efficiently. In particular, using the `nn.Module` class, we can stack a few layers and build an NN. You can see the list of all the layers that are already available at `https://pytorch.org/docs/stable/nn.html`. For this problem, we are going to use the `Linear` layer, which is also known as a fully connected layer or dense layer, and can be best represented by $f(w \times x + b)$, where x represents a tensor containing the input features, w and b are the weight matrix and the bias vector, and f is the activation function.

Each layer in an NN receives its inputs from the preceding layer; therefore, its dimensionality (rank and shape) is fixed. Typically, we need to concern ourselves with the dimensionality of output only when we design an NN architecture. Here, we want to define a model with two hidden layers. The first one receives an input of four features and projects them to 16 neurons. The second layer receives the output of the previous layer (which has a size of *16*) and projects them to three output neurons, since we have three class labels. This can be done as follows:

```
>>> class Model(nn.Module):
...     def __init__(self, input_size, hidden_size, output_size):
...         super().__init__()
...         self.layer1 = nn.Linear(input_size, hidden_size)
...         self.layer2 = nn.Linear(hidden_size, output_size)
...     def forward(self, x):
...         x = self.layer1(x)
...         x = nn.Sigmoid()(x)
...         x = self.layer2(x)
...         return x
>>> input_size = X_train_norm.shape[1]
```

```
>>> hidden_size = 16
>>> output_size = 3
>>> model = Model(input_size, hidden_size, output_size)
```

Here, we used the sigmoid activation function for the first layer and softmax activation for the last (output) layer. Softmax activation in the last layer is used to support multiclass classification since we have three class labels here (which is why we have three neurons in the output layer). We will discuss the different activation functions and their applications later in this chapter.

Next, we specify the loss function as cross-entropy loss and the optimizer as Adam:

The Adam optimizer is a robust, gradient-based optimization method, which we will talk about in detail in *Chapter 14, Classifying Images with Deep Convolutional Neural Networks*.

```
>>> learning_rate = 0.001
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Now, we can train the model. We will specify the number of epochs to be `100`. The code of training the flower classification model is as follows:

```
>>> num_epochs = 100
>>> loss_hist = [0] * num_epochs
>>> accuracy_hist = [0] * num_epochs
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist[epoch] += loss.item()*y_batch.size(0)
...         is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
...         accuracy_hist[epoch] += is_correct.sum()
...      loss_hist[epoch] /= len(train_dl.dataset)
...      accuracy_hist[epoch] /= len(train_dl.dataset)
```

The `loss_hist` and `accuracy_hist` lists keep the training loss and the training accuracy after each epoch. We can use this to visualize the learning curves as follows:

```
>>> fig = plt.figure(figsize=(12, 5))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(loss_hist, lw=3)
```

```
>>> ax.set_title('Training loss', size=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(accuracy_hist, lw=3)
>>> ax.set_title('Training accuracy', size=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()
```

The learning curves (training loss and training accuracy) are as follows:

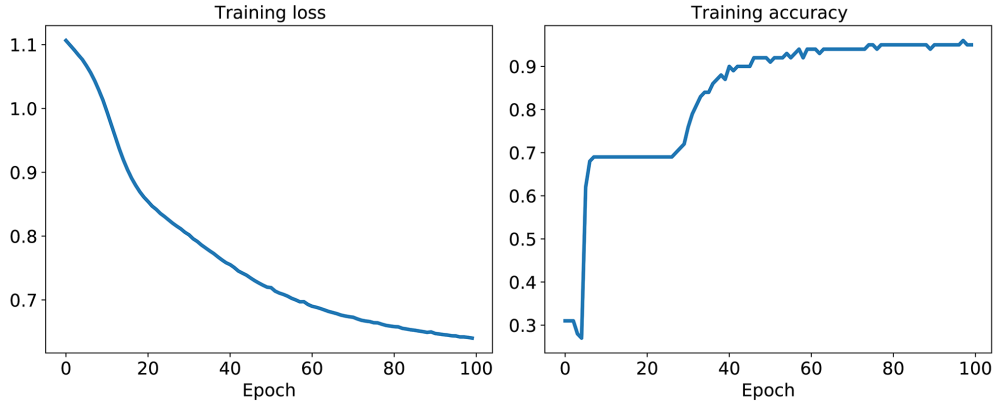

Figure 12.9: Training loss and accuracy curves

Evaluating the trained model on the test dataset

We can now evaluate the classification accuracy of the trained model on the test dataset:

```
>>> X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
>>> X_test_norm = torch.from_numpy(X_test_norm).float()
>>> y_test = torch.from_numpy(y_test)
>>> pred_test = model(X_test_norm)
>>> correct = (torch.argmax(pred_test, dim=1) == y_test).float()
>>> accuracy = correct.mean()
>>> print(f'Test Acc.: {accuracy:.4f}')
Test Acc.: 0.9800
```

Since we trained our model with standardized features, we also applied the same standardization to the test data. The classification accuracy is 0.98 (that is, 98 percent).

Saving and reloading the trained model

Trained models can be saved on disk for future use. This can be done as follows:

```
>>> path = 'iris_classifier.pt'
>>> torch.save(model, path)
```

Calling `save(model)` will save both the model architecture and all the learned parameters. As a common convention, we can save models using a `'pt'` or `'pth'` file extension.

Now, let's reload the saved model. Since we have saved both the model architecture and the weights, we can easily rebuild and reload the parameters in just one line:

```
>>> model_new = torch.load(path)
```

Try to verify the model architecture by calling `model_new.eval()`:

```
>>> model_new.eval()
Model(
  (layer1): Linear(in_features=4, out_features=16, bias=True)
  (layer2): Linear(in_features=16, out_features=3, bias=True)
)
```

Finally, let's evaluate this new model that is reloaded on the test dataset to verify that the results are the same as before:

```
>>> pred_test = model_new(X_test_norm)
>>> correct = (torch.argmax(pred_test, dim=1) == y_test).float()
>>> accuracy = correct.mean()
>>> print(f'Test Acc.: {accuracy:.4f}')
Test Acc.: 0.9800
```

If you want to save only the learned parameters, you can use `save(model.state_dict())` as follows:

```
>>> path = 'iris_classifier_state.pt'
>>> torch.save(model.state_dict(), path)
```

To reload the saved parameters, we first need to construct the model as we did before, then feed the loaded parameters to the model:

```
>>> model_new = Model(input_size, hidden_size, output_size)
>>> model_new.load_state_dict(torch.load(path))
```

Choosing activation functions for multilayer neural networks

For simplicity, we have only discussed the sigmoid activation function in the context of multilayer feedforward NNs so far; we have used it in the hidden layer as well as the output layer in the MLP implementation in *Chapter 11*.

Note that in this book, the sigmoidal logistic function, $\sigma(z) = \frac{1}{1+e^{-z}}$, is referred to as the *sigmoid* function for brevity, which is common in machine learning literature. In the following subsections, you will learn more about alternative nonlinear functions that are useful for implementing multilayer NNs.

Technically, we can use any function as an activation function in multilayer NNs as long as it is differentiable. We can even use linear activation functions, such as in Adaline (*Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*). However, in practice, it would not be very useful to use linear activation functions for both hidden and output layers, since we want to introduce nonlinearity in a typical artificial NN to be able to tackle complex problems. The sum of linear functions yields a linear function after all.

The logistic (sigmoid) activation function that we used in *Chapter 11* probably mimics the concept of a neuron in a brain most closely—we can think of it as the probability of whether a neuron fires. However, the logistic (sigmoid) activation function can be problematic if we have highly negative input, since the output of the sigmoid function will be close to zero in this case. If the sigmoid function returns output that is close to zero, the NN will learn very slowly, and it will be more likely to get trapped in the local minima of the loss landscape during training. This is why people often prefer a hyperbolic tangent as an activation function in hidden layers.

Before we discuss what a hyperbolic tangent looks like, let's briefly recapitulate some of the basics of the logistic function and look at a generalization that makes it more useful for multiclass classification problems.

Logistic function recap

As was mentioned in the introduction to this section, the logistic function is, in fact, a special case of a sigmoid function. You will recall from the section on logistic regression in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, that we can use a logistic function to model the probability that sample x belongs to the positive class (class `1`) in a binary classification task.

The given net input, z, is shown in the following equation:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{i=0}^{m} w_i x_i = w^{\mathrm{T}} x$$

The logistic (sigmoid) function will compute the following:

$$\sigma_{\text{logistic}}(z) = \frac{1}{1 + e^{-z}}$$

Note that w_0 is the bias unit (y-axis intercept, which means $x_0 = 1$). To provide a more concrete example, let's take a model for a two-dimensional data point, x, and a model with the following weight coefficients assigned to the w vector:

```
>>> import numpy as np
>>> X = np.array([1, 1.4, 2.5]) ## first value must be 1
>>> w = np.array([0.4, 0.3, 0.5])
>>> def net_input(X, w):
...     return np.dot(X, w)
>>> def logistic(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> def logistic_activation(X, w):
...     z = net_input(X, w)
...     return logistic(z)
>>> print(f'P(y=1|x) = {logistic_activation(X, w):.3f}')
P(y=1|x) = 0.888
```

If we calculate the net input (z) and use it to activate a logistic neuron with those particular feature values and weight coefficients, we get a value of `0.888`, which we can interpret as an 88.8 percent probability that this particular sample, x, belongs to the positive class.

In *Chapter 11*, we used the one-hot encoding technique to represent multiclass ground truth labels and designed the output layer consisting of multiple logistic activation units. However, as will be demonstrated by the following code example, an output layer consisting of multiple logistic activation units does not produce meaningful, interpretable probability values:

```
>>> # W : array with shape = (n_output_units, n_hidden_units+1)
>>> #     note that the first column are the bias units
>>> W = np.array([[1.1, 1.2, 0.8, 0.4],
...               [0.2, 0.4, 1.0, 0.2],
...               [0.6, 1.5, 1.2, 0.7]])
>>> # A : data array with shape = (n_hidden_units + 1, n_samples)
>>> #     note that the first column of this array must be 1
>>> A = np.array([[1, 0.1, 0.4, 0.6]])
>>> Z = np.dot(W, A[0])
>>> y_probas = logistic(Z)
>>> print('Net Input: \n', Z)
Net Input:
[1.78  0.76  1.65]
>>> print('Output Units:\n', y_probas)
Output Units:
[ 0.85569687  0.68135373  0.83889105]
```

As you can see in the output, the resulting values cannot be interpreted as probabilities for a three-class problem. The reason for this is that they do not sum to 1. However, this is, in fact, not a big concern if we use our model to predict only the class labels and not the class membership probabilities. One way to predict the class label from the output units obtained earlier is to use the maximum value:

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('Predicted class label:', y_class)
Predicted class label: 0
```

In certain contexts, it can be useful to compute meaningful class probabilities for multiclass predictions. In the next section, we will take a look at a generalization of the logistic function, the `softmax` function, which can help us with this task.

Estimating class probabilities in multiclass classification via the softmax function

In the previous section, you saw how we can obtain a class label using the `argmax` function. Previously, in the *Building a multilayer perceptron for classifying flowers in the Iris dataset* section, we determined `activation='softmax'` in the last layer of the MLP model. The `softmax` function is a soft form of the `argmax` function; instead of giving a single class index, it provides the probability of each class. Therefore, it allows us to compute meaningful class probabilities in multiclass settings (multinomial logistic regression).

In `softmax`, the probability of a particular sample with net input z belonging to the ith class can be computed with a normalization term in the denominator, that is, the sum of the exponentially weighted linear functions:

$$p(z) = \sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}}$$

To see `softmax` in action, let's code it up in Python:

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))
>>> y_probas = softmax(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
[ 0.44668973  0.16107406  0.39223621]
>>> np.sum(y_probas)
1.0
```

As you can see, the predicted class probabilities now sum to 1, as we would expect. It is also notable that the predicted class label is the same as when we applied the `argmax` function to the logistic output.

It may help to think of the result of the `softmax` function as a *normalized* output that is useful for obtaining meaningful class-membership predictions in multiclass settings. Therefore, when we build a multiclass classification model in PyTorch, we can use the `torch.softmax()` function to estimate the probabilities of each class membership for an input batch of examples. To see how we can use the `torch.softmax()` activation function in PyTorch, we will convert `Z` to a tensor in the following code, with an additional dimension reserved for the batch size:

```
>>> torch.softmax(torch.from_numpy(Z), dim=0)
tensor([0.4467, 0.1611, 0.3922], dtype=torch.float64)
```

Broadening the output spectrum using a hyperbolic tangent

Another sigmoidal function that is often used in the hidden layers of artificial NNs is the **hyperbolic tangent** (commonly known as **tanh**), which can be interpreted as a rescaled version of the logistic function:

$$\sigma_{\text{logistic}}(z) = \frac{1}{1+e^{-z}}$$

$$\sigma_{\tanh}(z) = 2 \times \sigma_{\text{logistic}}(2z) - 1 = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

The advantage of the hyperbolic tangent over the logistic function is that it has a broader output spectrum ranging in the open interval (–1, 1), which can improve the convergence of the backpropagation algorithm (*Neural Networks for Pattern Recognition*, *C. M. Bishop*, *Oxford University Press*, pages: 500-501, *1995*).

In contrast, the logistic function returns an output signal ranging in the open interval (0, 1). For a simple comparison of the logistic function and the hyperbolic tangent, let's plot the two sigmoidal functions:

```
>>> import matplotlib.pyplot as plt
>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)
>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)
>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('net input $z$')
>>> plt.ylabel('activation $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle=':')
>>> plt.axhline(0.5, color='black', linestyle=':')
>>> plt.axhline(0, color='black', linestyle=':')
```

```
>>> plt.axhline(-0.5, color='black', linestyle=':')
>>> plt.axhline(-1, color='black', linestyle=':')
>>> plt.plot(z, tanh_act,
...          linewidth=3, linestyle='--',
...          label='tanh')
>>> plt.plot(z, log_act,
...          linewidth=3,
...          label='logistic')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

As you can see, the shapes of the two sigmoidal curves look very similar; however, the `tanh` function has double the output space of the `logistic` function:

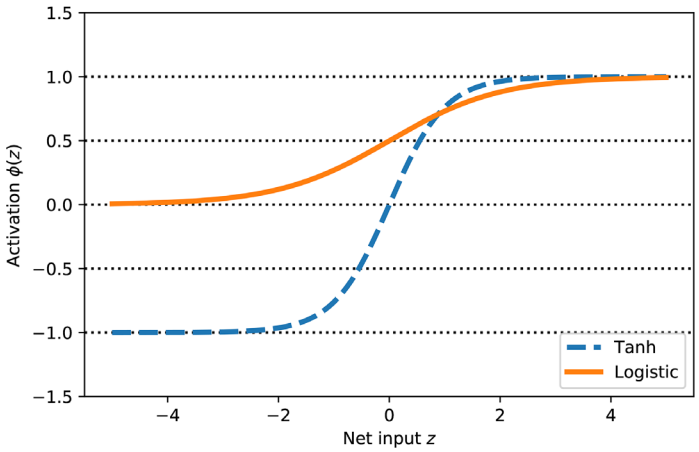

Figure 12.10: A comparison of the tanh and logistic functions

Note that we previously implemented the `logistic` and `tanh` functions verbosely for the purpose of illustration. In practice, we can use NumPy's `tanh` function.

Alternatively, when building an NN model, we can use `torch.tanh(x)` in PyTorch to achieve the same results:

```
>>> np.tanh(z)
array([-0.9999092 , -0.99990829, -0.99990737, ...,  0.99990644,
        0.99990737,  0.99990829])
>>> torch.tanh(torch.from_numpy(z))
tensor([-0.9999, -0.9999, -0.9999,  ...,  0.9999,  0.9999,  0.9999],
       dtype=torch.float64)
```

In addition, the logistic function is available in SciPy's `special` module:

```
>>> from scipy.special import expit
>>> expit(z)
array([0.00669285, 0.00672617, 0.00675966, ..., 0.99320669, 0.99324034,
       0.99327383])
```

Similarly, we can use the `torch.sigmoid()` function in PyTorch to do the same computation, as follows:

```
>>> torch.sigmoid(torch.from_numpy(z))
tensor([0.0067, 0.0067, 0.0068,  ..., 0.9932, 0.9932, 0.9933],
       dtype=torch.float64)
```

Note that using `torch.sigmoid(x)` produces results that are equivalent to `torch.nn.Sigmoid()(x)`, which we used earlier. `torch.nn.Sigmoid` is a class to which you can pass in parameters to construct an object in order to control the behavior. In contrast, `torch.sigmoid` is a function.

Rectified linear unit activation

The **rectified linear unit** (**ReLU**) is another activation function that is often used in deep NNs. Before we delve into ReLU, we should step back and understand the vanishing gradient problem of tanh and logistic activations.

To understand this problem, let's assume that we initially have the net input $z_1 = 20$, which changes to $z_2 = 25$. Computing the tanh activation, we get $\sigma(z_1) = 1.0$ and $\sigma(z_2) = 1.0$, which shows no change in the output (due to the asymptotic behavior of the tanh function and numerical errors).

This means that the derivative of activations with respect to the net input diminishes as z becomes large. As a result, learning the weights during the training phase becomes very slow because the gradient terms may be very close to zero. ReLU activation addresses this issue. Mathematically, ReLU is defined as follows:

$$\sigma(z) = \max(0, z)$$

ReLU is still a nonlinear function that is good for learning complex functions with NNs. Besides this, the derivative of ReLU, with respect to its input, is always 1 for positive input values. Therefore, it solves the problem of vanishing gradients, making it suitable for deep NNs. In PyTorch, we can apply the ReLU activation `torch.relu()` as follows:

```
>>> torch.relu(torch.from_numpy(z))
tensor([0.0000, 0.0000, 0.0000,  ..., 4.9850, 4.9900, 4.9950],
       dtype=torch.float64)
```

We will use the ReLU activation function in the next chapter as an activation function for multilayer convolutional NNs.

Now that we know more about the different activation functions that are commonly used in artificial NNs, let's conclude this section with an overview of the different activation functions that we have encountered so far in this book:

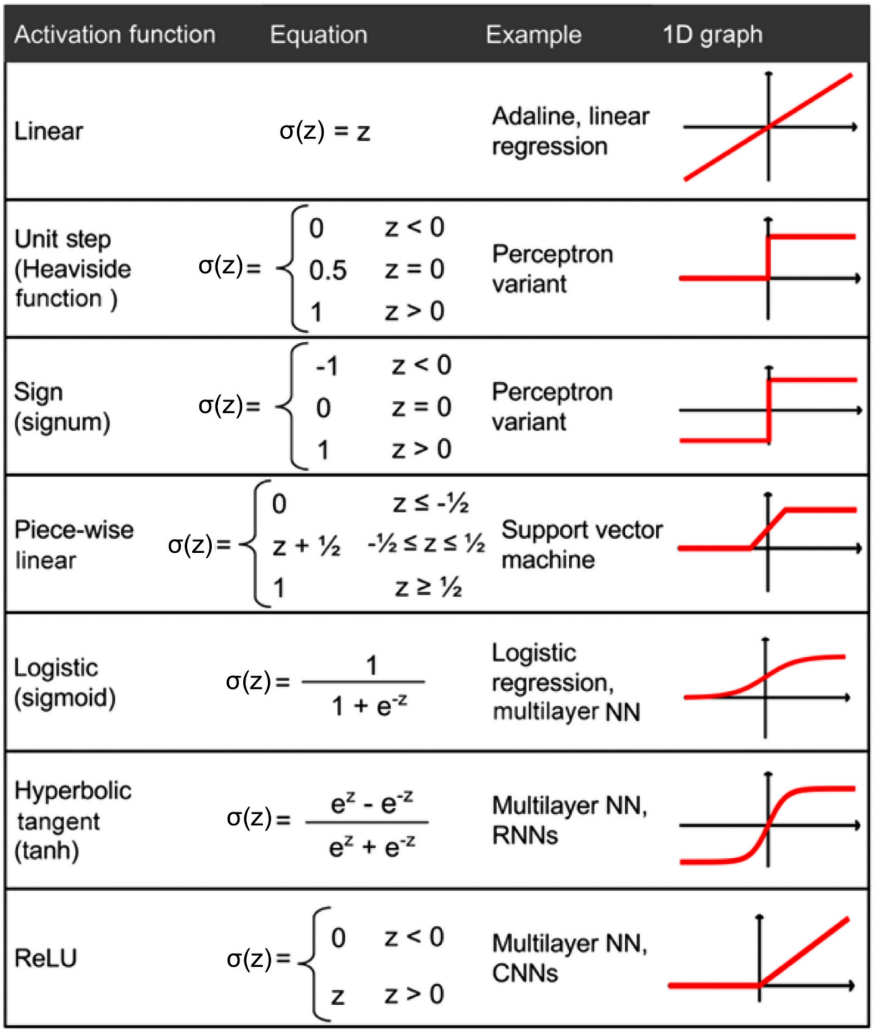

Figure 12.11: The activation functions covered in this book

You can find the list of all activation functions available in the `torch.nn` module at `https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions`.

Summary

In this chapter, you learned how to use PyTorch, an open source library for numerical computations, with a special focus on deep learning. While PyTorch is more inconvenient to use than NumPy, due to its additional complexity to support GPUs, it allows us to define and train large, multilayer NNs very efficiently.

Also, you learned about using the `torch.nn` module to build complex machine learning and NN models and run them efficiently. We explored model building in PyTorch by defining a model from scratch via the basic PyTorch tensor functionality. Implementing models can be tedious when we have to program at the level of matrix-vector multiplications and define every detail of each operation. However, the advantage is that this allows us, as developers, to combine such basic operations and build more complex models. We then explored `torch.nn`, which makes building NN models a lot easier than implementing them from scratch.

Finally, you learned about different activation functions and understood their behaviors and applications. Specifically, in this chapter, we covered tanh, softmax, and ReLU.

In the next chapter, we'll continue our journey and dive deeper into PyTorch, where we'll find ourselves working with PyTorch computation graphs and the automatic differentiation package. Along the way, you'll learn many new concepts, such as gradient computations.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`