第 2 章
PyTorch

Python 是深度学习领域的首选编程语言。构建某些大型的复杂神经网络时，经常需要使用深度学习框架来实现快速、有效的开发，这些深度学习框架大部分都是基于 Python 的，常用的框架包括 PyTorch、TensorFlow、Keras、Caffe2 等。其中，PyTorch 和 TensorFlow 是应用范围最广、热度最高的两个深度学习框架，本书将对这两个框架进行重点介绍。

本章主要对 PyTorch 进行简要的介绍。

2.1 PyTorch 概述

2.1.1 什么是 PyTorch

什么是 PyTorch？其实，PyTorch 可以拆分成两部分：Py 和 Torch。Py 就是 Python，Torch 是一个有大量机器学习算法支持的科学计算框架。Lua 语言简洁高效，但由于其过于小众，用的人不是很多。考虑到 Python 在人工智能领域的领先地位，以及其生态的完整性和接口的易用性，几乎任何框架都不可避免地要提供 Python 接口。终于，2017 年，Torch 的幕后团队使用 Python 重写了 Torch 的很多内容，推出了 PyTorch，并提供了 Python 接口。此后，PyTorch 成为最流行的深度学习框架之一。

直白地说，PyTorch 可以看成一个 Python 库，可以像 NumPy、Pandas 一样被 Python 所调用。PyTorch 与 NumPy 的功能是类似的，可以把 PyTorch 看成应用在神经网络里的 NumPy，而且是加入了 GPU 支持的 NumPy。

2.1.2　为什么使用 PyTorch

深度学习框架那么多，为什么我们这么青睐 PyTorch 呢？我们先来看看有哪些机构在使用 PyTorch 吧，如图 2-1 所示。

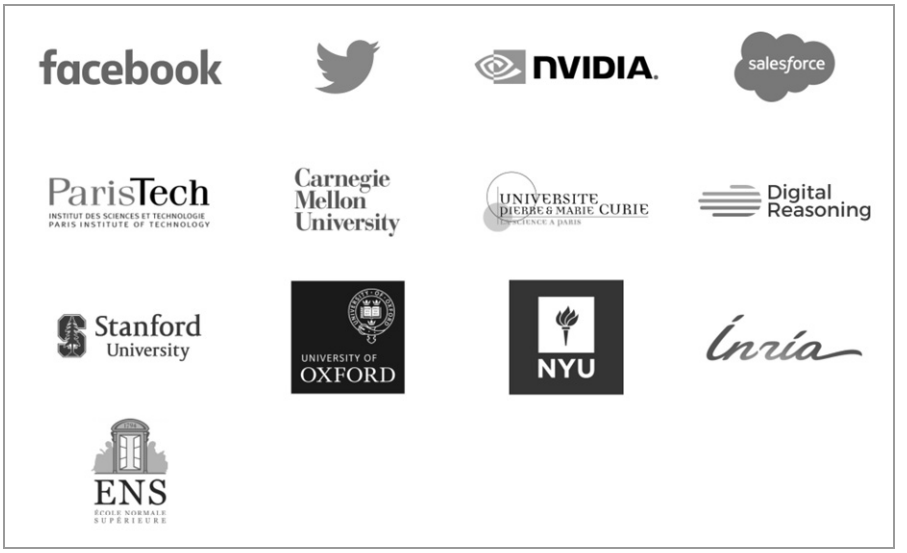

图 2-1　使用 PyTorch 的机构

由图 2-1 可知，PyTorch 已经被 Twitter、CMU 和 Salesforce 等多个机构使用。这足以说明 PyTorch 是好用的，而且是值得推广的。

PyTorch 实际上是 NumPy 的替代工具，而且支持 GPU，带有高级功能，可以用来搭建和训练深度神经网络。只要熟悉 NumPy、Python 以及基础的深度学习概念，PyTorch 就非常容易上手。这就是 PyTorch 备受青睐的重要原因之一。

2.2　PyTorch 的安装

我们在第 1 章中已经介绍了 Anaconda 的安装和使用，下面将介绍在 Anaconda 环境下安装 PyTorch 的方法。

首先，我们为 PyTorch 创建一个虚拟环境。创建虚拟环境是一个好的编程习惯，因为在实际项目开发过程中，我们通常会根据自己的需求去下载各种框架和库，但是可能每个项目使用的框架和库并不一样，或使用的版本不一样，这时就需要根据需求不断地更新或卸载相应的库，

管理起来相当麻烦。创建虚拟环境相当于为不同的项目创建一个独立的空间，在这个空间里安装的任何库和框架都是独立的，不会影响到外部环境。

因为安装了 Anaconda，所以创建虚拟环境变得很简单，可以使用 Anaconda Prompt 来创建。

首先打开 Anaconda Prompt。在命令行窗口中输入以下代码：

```
> conda create --name pytorch python=3.7
```

注意，这里的 pytorch 是虚拟环境的名称，可以自由命名。创建完成之后，可以输入以下命令，进入虚拟环境 pytorch：

```
> activate pytorch
```

注意，不想使用 PyTorch 时，可以输入 deactivate pytorch 关闭当前虚拟环境。

进入该虚拟环境之后，我们就可以安装 PyTorch 了。首先在浏览器中输入 https://pytorch.org/，进入 PyTorch 的官网。然后单击 Get Started，进入下载页面。PyTorch 支持 Windows、Mac、Linux 操作系统，只需按照提示，选择 PyTorch 版本、操作系统、Python 版本、CUDA 版本等选项，然后 Run this Command 区域就会显示需要的安装命令，如图 2-2 所示。

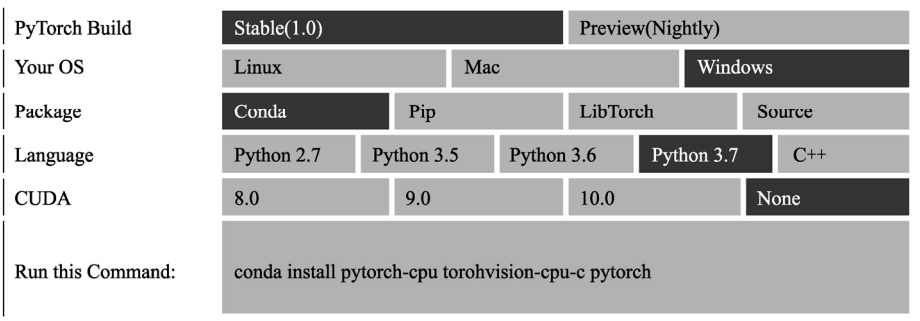

图 2-2　PyTorch 的下载

这里需要注意的是，如果想选择安装 GPU（Graphics Processing Unit）版本的 PyTorch，CUDA 就不能设置为 None。安装 GPU 版本的 PyTorch，首先计算机需要有一块 NVIDIA 的 GPU 显卡并安装了显卡驱动。在安装 PyTorch 之前，需要提前安装 CUDA 和 CUDNN。这里就不再对 CUDA 和 CUDNN 的安装进行介绍了，感兴趣的读者可自行学习安装方法。

因为安装 GPU 版本的 PyTorch 需要有硬件支持，而且准备工作较多，因此，本书推荐安装 CPU 版本的 PyTorch。其实，只有比较复杂的神经网络，GPU 版本和 CPU 版本 PyTorch 的运行速度差异较大。一般规模的神经网络，两者的运行速度并无较大的区别。注意，本书的所有代码都可以在 CPU 版本的 PyTorch 上运行，有的运行时间较慢，但并无大碍。当然，如果计算机上有 GPU，也可以安装 GPU 版本的 PyTorch。

以 Windows 操作系统为例，安装 CPU 版本的 PyTorch 时，Run this Command 中显示的命令

如下：

```
conda install pytorch-cpu torchvision-cpu -c pytorch
```

接下来，我们只需要在虚拟环境 pytorch 中输入上面这条命令，就可以顺利完成 PyTorch 的安装。

注意，新建的虚拟环境 pytorch 是没有 Jupyter 的，所以，我们需要在该环境下输入以下命令安装 Jupyter：

```
conda install jupyter
```

除了 Jupyter 之外，还可以根据需要使用 conda 命令安装其他的 Python 库。

安装工作完成后，如何验证 PyTorch 是否安装正确呢？打开 Anaconda Navigator，因为 PyTorch 是安装在虚拟环境 pytorch 中的，所以在 Anaconda Navigator 界面的 Applications on 下拉列表框中选择 pytorch，然后启动该环境下的 Jupyter Notebook，如图 2-3 所示。

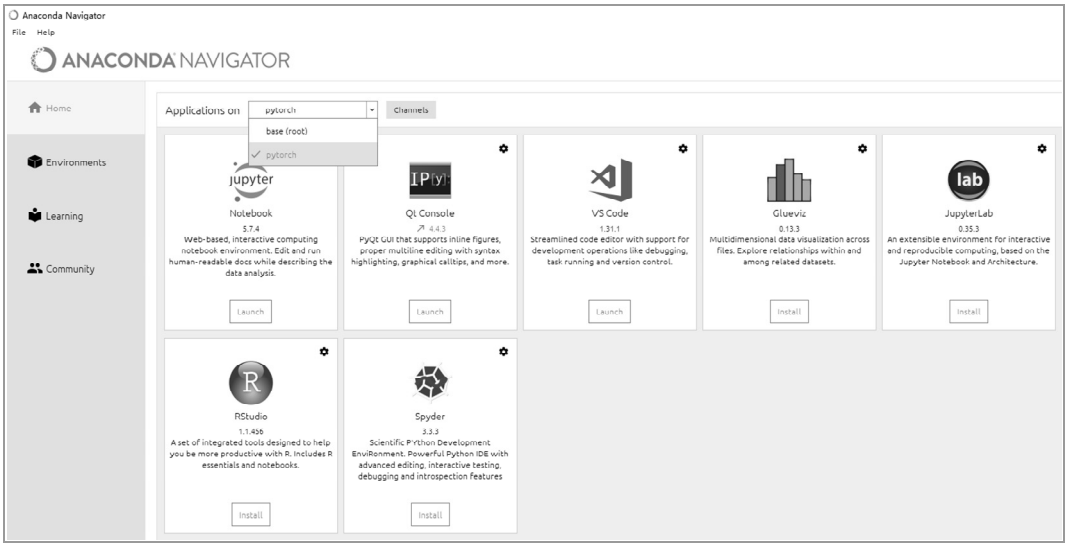

图 2-3　切换至虚拟环境 pytorch

打开 Jupyter Notebook 之后，输入 import 语句，如果没有报错，就说明 PyTorch 已经安装成功了。也可以在 Jupyter Notebook 中查看安装的 PyTorch 的版本，如图 2-4 所示。

图 2-4　导入 PyTorch 库并查看 PyTorch 的版本

这里，可能有的读者会问为什么有 torch 和 torchvision？torch 就是 PyTorch 的核心库，torchvision 包是服务于 PyTorch 深度学习框架的，用来生成图片、视频数据集、一些流行的模型类和预训练模型。简单来说，torchvision 由 torchvision.datasets、torchvision.models、torchvision.transforms、torchvision.utils 四个模块组成。安装的时候，我们同时安装了 PyTorch 和 torchvision。后面的章节中，我们还会介绍。

2.3 张量

2.3.1 张量的创建

张量（tensor）是 PyTorch 里基础的运算单位，类似于 NumPy 中的数组。但是，张量可以在 GPU 版本的 PyTorch 上运行，而 NumPy 中的数组只能在 CPU 版本的 PyTorch 上运行。因此，张量的运算速度更快。

下面，我们来创建一个张量。

```
>>> x = torch.randn(2,2)
>>> print(x)
tensor([[-2.2093,  0.1976],
        [-0.9493,  1.2901]])
```

上面代码创建的是 2×2 的随机初始化张量，可以看出 PyTorch 的语法和函数与 NumPy 是非常类似的。还可以使用以下代码创建不同的张量。

根据 Python 列表创建张量，代码如下：

```
>>> x = torch.tensor([[1, 2], [3, 4]])
>>> print(x)
tensor([[1, 2],
        [3, 4]])
```

创建一个全零的张量，代码如下：

```
>>> x = torch.zeros(2,2)
>>> print(x)
tensor([[0., 0.],
        [0., 0.]])
```

基于现有的张量创建新的张量，代码如下：

```
>>> x = torch.zeros(2,2)
>>> y = torch.ones_like(x)
>>> print(x)
tensor([[0., 0.],
        [0., 0.]])
```

```
>>> print(y)
tensor([[1., 1.],
        [1., 1.]])
```

在创建张量的时候，还可以指定数据类型，例如创建一个长整型张量，代码如下：

```
>>> x = torch.ones(2, 2, dtype=torch.long)
>>> print(x)
tensor([[1, 1],
        [1, 1]])
```

2.3.2　张量的数学运算

张量的数学运算与数组的数学运算相同，与 NumPy 的数学运算类似。下面通过几个例子来说明。

两个张量相加，代码如下：

```
>>> x = torch.ones(2,2)
>>> y = torch.ones(2,2)
>>> z = x + y
>>> print(z)
tensor([[2., 2.],
        [2., 2.]])
```

也可以使用 torch.add()实现张量相加，代码如下：

```
>>> x = torch.ones(2,2)
>>> y = torch.ones(2,2)
>>> z = torch.add(x, y)
>>> print(z)
tensor([[2., 2.],
        [2., 2.]])
```

还可以使用.add_()实现张量的替换，代码如下：

```
>>> x = torch.ones(2,2)
>>> y = torch.ones(2,2)
>>> y.add_(x)
tensor([[2., 2.],
        [2., 2.]])
>>> print(y)
tensor([[2., 2.],
        [2., 2.]])
```

张量的乘法有两种形式，第一种是对应元素相乘，代码如下：

```
>>> x = torch.tensor([[1, 2], [3, 4]])
>>> y = torch.tensor([[1, 2], [3, 4]])
>>> x.mul(y)
tensor([[ 1,  4],
        [ 9, 16]])
```

第二种是矩阵相乘，这种形式更加常用，代码如下：

```
>>> x = torch.tensor([[1, 2], [3, 4]])
>>> y = torch.tensor([[1, 2], [3, 4]])
>>> x.mm(y)
tensor([[ 7, 10],
        [15, 22]])
```

2.3.3 张量与 NumPy 数组

PyTorch 中的张量与 NumPy 数组可以互相转化，而且两者共享内存位置，如果一个发生改变，另一个也随之改变。

1. 张量转换为 NumPy 数组

通过简单的.numpy()就可以将张量转换为 NumPy 数组，代码如下：

```
>>> a = torch.ones(2,2)
>>> b = a.numpy()
>>> print(type(a))
<class 'torch.Tensor'>
>>> print(type(b))
<class 'numpy.ndarray'>
```

此时，如果张量发生改变，对应的 NumPy 数组也有相同的变化，代码如下：

```
>>> a.add_(1)
tensor([[2., 2.],
        [2., 2.]])
>>> print(b)
[[2. 2.]
 [2. 2.]]
```

2. NumPy 数组转换为张量

对于 NumPy 数组，可以通过 torch.from_numpy()转化为张量，代码如下：

```
>>> a = np.array([[1, 1], [1, 1]])
>>> b = torch.from_numpy(a)
>>> print(type(a))
<class 'numpy.ndarray'>
>>> print(type(b))
<class 'torch.Tensor'>
```

同样，如果 NumPy 数组发生改变，对应的张量也有相同的变化，代码如下：

```
>>> np.add(a, 1, out=a)
array([[2, 2],
       [2, 2]])
>>> print(b)
tensor([[2, 2],
        [2, 2]], dtype=torch.int32)
```

2.3.4　CUDA 张量

新建的张量默认保存在 CPU 里，如果安装了 GPU 版本的 PyTorch，就可以将张量移动到 GPU 里，代码如下：

```
>>> a = torch.ones(2,2)
>>> if torch.cuda.is_available():
...     a_cuda = a.cuda()
...     print(a_cuda)
tensor([[1., 1.],
        [1., 1.]], device='cuda:0')
```

2.4　自动求导

深度学习的算法在本质上是通过反向传播求导数（后面的章节将会详细介绍），此功能由 PyTorch 的自动求导（autograd）模块实现。关于张量的所有操作，自动求导模块都能为它们自动提供微分，避免了手动计算导数的复杂过程，可以节约大量的时间。

如果要让张量使用自动求导功能，只需要在定义张量的时候设置参数 tensor.requires_grad=True 即可。默认情况下，张量是没有自动求导功能的。

```
>>> x = torch.ones(2, 2, requires_grad=True)
>>> y = torch.ones(2, 2, requires_grad=True)
>>> print(x.requires_grad)
True
>>> print(y.requires_grad)
True
```

新建的张量 x 和 y 的 requires_grad 参数值均为 True。

2.4.1　返回值是标量

我们定义输出 $z=\frac{1}{4}\sum_i x_i+y_i$，代码如下：

```
>>> z = x + y
>>> z = z.mean()
>>> print(z)
tensor(2., grad_fn=<MeanBackward1>)
```

在 PyTorch 中，每个通过函数计算得到的变量都有一个.grad_fn 属性。因为 z 是通过 x 和 y 运算而来，所以具有 grad_fn 属性。如果现在要计算 z 对 x 的偏导数 $\frac{\partial z}{\partial x}$ 以及 z 对 y 的偏导数 $\frac{\partial z}{\partial y}$，

首先要对 z 使用.backward()来定义反向传播，代码如下：

```
>>> z.backward()
```

然后直接使用 x.grad 来计算 $\frac{\partial z}{\partial x}$，使用 y.grad 来计算 $\frac{\partial z}{\partial y}$，代码如下：

```
>>> print(x.grad)
tensor([[0.2500, 0.2500],
        [0.2500, 0.2500]])
>>> print(y.grad)
tensor([[0.2500, 0.2500],
        [0.2500, 0.2500]])
```

可以看到，只需要简单的两行语句，PyTorch 的自动求导功能就可以计算出 $\frac{\partial z}{\partial x}$ 和 $\frac{\partial z}{\partial y}$。

2.4.2 返回值是张量

2.4.1 小节的例子中，输出 z 是一个标量，不需要在 backward()中指定任何参数。如果 z 是一个多维张量，则需要在 backward()中指定参数，匹配相应的尺寸。

我们来看下面这个例子。因为返回值 z 不是一个标量，所以需要输入一个大小相同的张量作为参数，这里我们用 ones_like()函数根据 z 生成一个张量。

```
>>> x = torch.ones(2, 2, requires_grad=True)
>>> y = torch.ones(2, 2, requires_grad=True)

>>> z = 2 * x + 3 * y

>>> z.backward(torch.ones_like(z))

>>> print(x.grad)
tensor([[2., 2.],
        [2., 2.]])

>>> print(y.grad)
tensor([[3., 3.],
        [3., 3.]])
```

2.4.3 禁止自动求导

我们可以使用 with torch.no_grad()来禁止已经设置 requires_grad=True 的张量进行自动求导，这个方法在测试集测试准确率的时候会经常用到。例如：

```
>>> print(x.requires_grad)
True
>>> print((2 * x).requires_grad)
```

```
True
>>> with torch.no_grad():
...     print((2 * x).requires_grad)
...
False
```

2.5　torch.nn 和 torch.optim

PyTorch 中，训练神经网络离不开两个重要的包：torch.nn 和 torch.optim。到目前为止本书还没有介绍神经网络，此处可以先来了解这两个非常重要的包，便于对后面内容的理解。注意，本小节的内容不会涉及神经网络的具体知识，仅介绍 torch.nn 和 torch.optim 的整体框架，便于读者理解。学习本节内容之前，最好有机器学习的基础，如线性回归、梯度下降算法等。

2.5.1　torch.nn

torch.nn 是专门为神经网络设计的模块化接口，构建于自动求导模块的基础上，可用来定义和运行神经网络。可以理解为，torch.nn 用于搭建一个模型，因此使用之前需要导入这个库，代码如下：

```
>>> import torch
>>> import torch.nn as nn
```

使用 torch.nn 来搭建一个模型的方法是定义一个类，代码如下：

```
class net_name(nn.Module):
    def __init__(self):
        super(net_name, self).__init__()
        self.fc = nn.Linear(1, 1)
        # 其他层

    def forward(self, x):
        out = self.fc(x)
        return out
```

上面的代码中，net_name 就是类名，名字可以自由拟定。该类继承于父类 nn.Module。nn.Module 是 torch.nn 中十分重要的类，包含网络各层的定义及 forward 方法，避免从底层搭建网络的麻烦。self.fc = nn.Linear(1, 1)表示对模型的搭建，模型仅仅是一个全连接层（fc），也叫线性层。（1,1）中的数字分别表示输入和输出的维度。这里，令输入和输出的维度都为 1。学习神经网络的时候，可以在此处构建更加复杂的网络。该类还包含一个 forward 函数，因为模型仅仅是一层全连接层，所以 out=self.fc(x)。最后，函数返回 out。上面这段代码其实就是一个简单的

线性回归模型。

使用该线性回归模型的时候，可以直接新建一个该模型的对象，代码如下：

```
net = net_name()
```

2.5.2 torch.optim

torch.optim 是一个实现了各种优化算法的库，包括最简单的梯度下降（Gradient Descent，GD）、随机梯度下降（Stochastic Gradient Descent，SGD）及其他更复杂的优化算法。直接输入以下命令即可导入 torch.optim：

```
>>> import torch.optim as optim
```

我们首先来了解 PyTorch 中的损失函数是如何定义的。损失函数用于计算每个实例的输出与真实样本的标签是否一致，并评估差距的大小。利用 torch.nn 可以很方便地定义损失函数。torch.nn 包有多种不同的损失函数，最简单的是 nn.MSELoss()，可以计算预测值与真实值的均方误差。

```
criterion = nn.MSELoss()
loss = criterion(output, target)
```

其中，output 是预测值，target 是真实值。

反向传播过程中，一般通过 loss.backward()计算梯度。注意，每次迭代时梯度要先清零，否则会被累加计算。

优化算法有很多，最简单的就是使用随机梯度下降算法，只需要一行语句即可：

```
optimizer = optim.SGD(net.parameters(), lr=0.01)
```

其中，net.parameters()表示模型参数，即待求参数。lr 为学习率，这里设置为 0.01。

总结一下，一次完整的前向传播加上反向传播需要用到 torch.nn 和 torch.optim 包。单次迭代对应的代码如下：

```
optimizer.zero_grad()      # 梯度清零

output = net(input)

loss = criterion(output, target)
loss.backward()

optimizer.step()           # 完成更新
```

这里有一点需要注意，每次迭代开始，梯度都要被清零，即执行 optimizer.zero_grad()，如果不清零的话，上次梯度会被累加。

2.6　线性回归

本章的前 5 节介绍了 PyTorch 的基本内容和语法，本节将通过一个简单的线性回归实例，介绍如何使用 PyTorch 编写一个完整的模型并验证它的好坏。

2.6.1　线性回归的基本原理

线性回归是一个最基本、最简单的机器学习算法，相信读者对它的基本原理已经非常熟悉了，本小节仅做简要介绍。

线性回归一般用于数值预测，如房屋价格预测、信用卡额度预测等。线性回归算法就是要找出这样一条拟合线或拟合面，能够最大限度地拟合真实的数据分布，如图 2-5 所示。

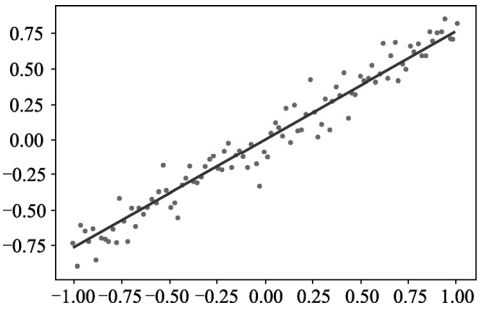

图 2-5　线性回归算法示意

这条直线可以表示为

$$\hat{y} = w_0 + w_1 x \tag{2-1}$$

式中，$\hat{y}$ 是预测值，w_0 和 w_1 是直线参数，正是需要去求的两个值。

如何确定这条直线以及相关的参数 w_0 和 w_1 呢？我们希望预测值 $\hat{y}$ 与真实值 y 越接近越好，因此引入代价函数。代价函数是定义在整个训练集上的，是所有样本误差的平均，也就是所有样本损失函数的平均。其实，代价函数与损失函数的唯一区别在于前者针对整个训练集，后者针对单个样本。代价函数越小，表明直线拟合得越好。

此时，代价函数通过均方差的计算而得到，计算公式为：

$$J = \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{2-2}$$

式中，m 表示总的样本个数，y_i 表示第 i 个样本的真实值，$\hat{y}_i$ 表示第 i 个样本的预测值。分母是 $2m$ 而不是 m 仅仅是为了平方求导的方便。

接下来要求最小化代价函数 J 时对应的参数 w_0 和 w_1。如何最小化代价函数 J 呢？最简单的方法就是使用梯度下降算法，其核心思想是在函数曲线上的某一点，函数沿梯度方向具有最大的变化率，那么沿着负梯度方向移动会不断逼近最小值，这样一个迭代的过程可以最终实现代价函数的最小化目标。

梯度下降算法中，w_0 和 w_1 迭代更新的表达式为：

$$w_0 = w_0 - \alpha \frac{\partial J}{\partial w_0} \tag{2-3}$$

$$w_1 = w_1 - \alpha \frac{\partial J}{\partial w_1} \tag{2-4}$$

式中，α 表示学习率。

这样，经过多次的迭代更新，J 会不断接近全局最小值。此时，就可以得到参数 w_0 和 w_1，直线也就确定了。

2.6.2 线性回归的 PyTorch 实现

1. 数据集

首先，我们要构造一些数据集，代码如下：

```
# y=3x+10，后面加上 torch.randn()函数制造噪音
x = torch.unsqueeze(torch.linspace(-1, 1, 50), dim=1)
y = 3*x + 10 + 0.5 * torch.randn(x.size())
```

显然，原始的数据集中，y 是由直线 $10+3x$ 加上一些随机噪声而得到的。原始数据的分布如图 2-6 所示。

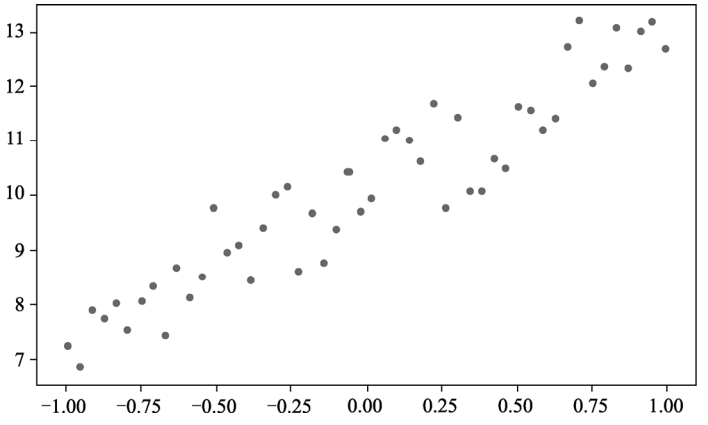

图 2-6　原始数据的分布

2. 模型定义

定义线性回归模型，代码如下：

```
class LinearRegression(nn.Module):
    def __init__(self):
        super(LinearRegression, self).__init__()
        self.fc = nn.Linear(1, 1)

    def forward(self, x):
        out = self.fc(x)
        return out

model = LinearRegression()
```

接下来，我们定义损失函数和优化函数，这里使用均方误差作为损失函数，使用梯度下降算法进行优化，代码如下：

```
# 定义损失函数和优化函数
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=5e-3)
```

3. 模型训练

开始进行模型的训练，代码如下：

```
num_epochs = 1000                # 遍历整个训练集的次数
for epoch in range(num_epochs):
    # forward
    out = model(x)               # 前向传播
    loss = criterion(out, y)     # 计算损失函数
    # backward
    optimizer.zero_grad()        # 梯度归零
    loss.backward()              # 反向传播
    optimizer.step()             # 更新参数

    if (epoch+1) % 20 == 0:
        print('Epoch[{}/{}], loss: {:.6f}'.format(epoch+1, num_epochs, loss.detach().
numpy()))
```

上面的模型训练代码中，迭代的次数为 1000 次，是遍历整个数据集的次数。先进行前向传播计算代价函数，然后向后传播计算梯度，这里需要注意的是，每次计算梯度前都要将梯度归零，不然梯度会累加到一起造成结果不收敛。为了便于观察结果，每 20 次迭代之后输出当前的均方差损失。

4. 模型测试

最后，我们通过 model.eval()函数将模型由训练模式变为测试模式，将数据放入模型中进行预测。最后，通过绘图工具 Matplotlib 判断拟合的直线与原始数据的贴近程度，代码如下：

```
model.eval()
y_hat = model(x)
plt.scatter(x.numpy(), y.numpy(), label='原始数据')
plt.plot(x.numpy(), y_hat.detach().numpy(), c='r', label='拟合直线')
# 显示图例
plt.legend()
plt.show()
```

y_hat 就是训练好的线性回归模型的预测值。注意，y_hat.detach().numpy()中，.detach()用于停止对张量的梯度跟踪。模型训练阶段需要跟踪梯度，但是模型测试的时候就不需要梯度跟踪了。最后显示的拟合直线如图 2-7 所示。

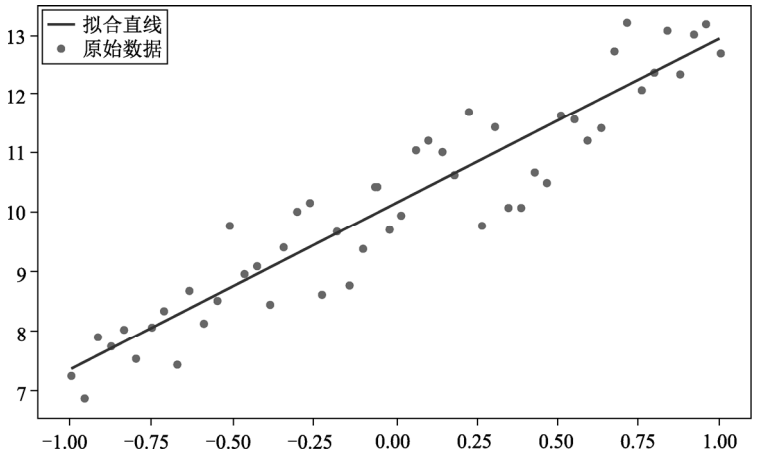

图 2-7　原始数据与拟合直线显示效果

可以看到，线性回归模型与原始数据拟合得非常好。我们可以使用下面的代码来查看这条直线的参数 w_0 和 w_1：

```
>>> list(model.named_parameters())
[('fc.weight', Parameter containing:
  tensor([[2.7447]], requires_grad=True)), ('fc.bias', Parameter containing:
  tensor([10.1136], requires_grad=True))]
```

通过参数查询可得 w_0=10.1136，w_1=2.7447，与构造数据时使用的直线 y=10+3x 非常接近。

至此，我们已经介绍了 PyTorch 的基本用法，并使用 PyTorch 实现了一个简单的线性回归模型。本书后面的神经网络章节中，我们将会学习更多、更重要的与 PyTorch 有关的知识，并使用这些知识来处理更复杂的机器视觉（Computer Vision，CV）和自然语言处理（Natural Language Processing，NLP）问题。