实验 1：PyTorch 基本操作及前馈神经网络

2024 年 3 月 12 日

实验 1：PyTorch 基本操作及前馈神经网络

1 实验简介

本实验介绍 PyTorch 及其他常用 Python 工具包的使用方法。通过本实验，你将掌握以下技能：

- PyTorch 的基本数据操作
- 数据预处理方法
- PyTorch 自动微分
- 搭建并训练全连接神经网络

2 安装 PyTorch

首先，你需要安装完成本实验所需的工具包。为了加快安装速度，请确保 `pip` 包管理器已切换到国内源。

[1]:
```
import pkgutil

# 检查 PyTorch 是否已安装
if pkgutil.find_loader('torch'):
    print('PyTorch is available.')
else:
    !pip install torch -i https://pypi.tuna.tsinghua.edu.cn/simple/  # 安装
PyTorch 最新的 CPU 稳定版本

# 检查 torchvision 是否已安装
if pkgutil.find_loader('torchvision'):
```

```
    print('torchvision is available.')
else:
    !pip install torchvision -i https://pypi.tuna.tsinghua.edu.cn/simple/  # 安装 torchvision
```

```
PyTorch is available.
torchvision is available.
```

如果你希望安装 GPU 版本，请参阅[Link]。

3 PyTorch 数据操作

首先，我们导入 torch。请注意，虽然框架名称为 PyTorch，但我们应该导入 torch 而不是 pytorch。

[2]:
```
import torch
```

PyTorch 的基本计算单元是张量（Tensor）。张量表示由一个数值组成的数组，这个数组可能有多个维度。

定义张量的方法很多，我们这里定义一个从 0 开始，间隔为 1，长度为 12 的等差数列张量。

[3]:
```
x = torch.arange(12)
```

[4]:
```
x
```

[4]:
```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

3.1 查看张量形状和元素数量

可以通过张量的 shape 属性来访问张量的形状和张量中元素的总数。

[5]:
```
x.shape  # 查看张量 x 的形状
```

[5]:
```
torch.Size([12])
```

[6]:
```
x.numel()  # 查看张量 x 中元素的个数（Number of elements）
```

[6]:
```
12
```

使用 len 函数返回的是张量最外层的维度。

[7]:
```
len(x)
```

[7]:
```
12
```

3.2 改变张量形状

要改变一个张量的形状而不改变元素数量和元素值，可以调用 reshape 函数。

[8]:
```
X = x.reshape(3, 4)
```

[9]:
```
X
```

[9]:
```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

3.3 常见的张量构造方法

我们经常需要使用全 0、全 1、其他常量或者从特定分布中随机采样的数字构建张量，构建方法如下。

[10]:
```
torch.zeros((2, 3, 4))
```

[10]:
```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

[11]:
```
torch.ones((2, 3, 4))
```

[11]:
```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],
```

```
        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

[12]:
```
torch.randn(3, 4)  # 表示构建一个形状为 3×4，每个元素随机采样，且采样得到的随机数服从标准正态分布的张量
```

[12]:
```
tensor([[ 0.0933,  0.5417,  2.4882,  0.3283],
        [-2.0056, -0.2062, -1.1585,  1.3966],
        [ 1.4770,  0.4409, -0.2156, -0.4812]])
```

[13]:
```
torch.rand(3, 4)  # 表示构建一个形状为 3×4，每个元素随机采样，且采样得到的随机数服从均匀分布、元素范围从 0 到 1 的张量
```

[13]:
```
tensor([[0.5731, 0.9141, 0.0293, 0.4724],
        [0.3077, 0.9250, 0.4063, 0.8057],
        [0.0961, 0.9126, 0.1881, 0.9734]])
```

通过提供包含数值的 Python 列表（或嵌套列表）来为所需张量中的每个元素赋予确定值。

[14]:
```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])  # 也就是手工构造一个张量
```

[14]:
```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

3.4 按元素计算

对张量使用常见的标准算术运算符（加、减、乘、除、幂运算等），可实现按元素运算。

[15]:
```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x**y
```

[15]:
```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
```

```
 tensor([ 1.,  4., 16., 64.]))
```

[16]:
```
torch.exp(x)
```

[16]:
```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

两个矩阵的按元素乘法称为哈达玛积（Hadamard product）（数学符号 ）。

[17]:
```
A = torch.arange(9).reshape(3, 3)
B = torch.randn(3, 3)
A, B
```

[17]:
```
(tensor([[0, 1, 2],
         [3, 4, 5],
         [6, 7, 8]]),
 tensor([[-0.6051,  1.1670, -0.9322],
         [ 0.2977, -0.3831, -0.3464],
         [ 0.8506,  1.1780, -0.9286]]))
```

[18]:
```
A * B
```

[18]:
```
tensor([[-0.0000,  1.1670, -1.8643],
        [ 0.8932, -1.5322, -1.7320],
        [ 5.1035,  8.2460, -7.4289]])
```

3.5 张量的连结

我们也可以通过 `torch.cat` 方法把多个张量连结（concatenate）在一起，且通过 `dim` 参数可指定在哪个维度进行连结。

[19]:
```
X = torch.arange(12, dtype=torch.float32).reshape((3, 4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
X, Y
```

[19]:
```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]),
 tensor([[2., 1., 4., 3.],
         [1., 2., 3., 4.],
```

```
        [4., 3., 2., 1.]]))
```

[20]:
```
torch.cat((X, Y), dim=0)
```

[20]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]])
```

[21]:
```
torch.cat((X, Y), dim=1)
```

[21]:
```
tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]])
```

3.6 张量逻辑运算

通过张量之间的逻辑运算，可构建二元张量。

[22]:
```
X == Y
```

[22]:
```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

3.7 张量元素求和

对张量中的所有元素进行求和会产生一个只有一个元素的张量。

[23]:
```
X.sum()
```

[23]:
```
tensor(66.)
```

当然，我们也可以沿某根轴进行求和。

[24]:
```
X.sum(axis=0)
```

[24]:
```
tensor([12., 15., 18., 21.])
```

[25]:
```
X.sum(axis=1)
```

[25]:
```
tensor([ 6., 22., 38.])
```

[26]:
```
X.sum(axis=[0, 1])
```

[26]:
```
tensor(66.)
```

一个与求和相关的量是平均值（mean 或 average）。

[27]:
```
X.mean()
```

[27]:
```
tensor(5.5000)
```

[28]:
```
X.mean(axis=0)
```

[28]:
```
tensor([4., 5., 6., 7.])
```

指定 `keepdims=True`，能够在计算总和或均值时保持轴数不变。

[29]:
```
X.sum(axis=1, keepdims=True)
```

[29]:
```
tensor([[ 6.],
        [22.],
        [38.]])
```

沿某个轴计算元素的累积总和。

[30]:
```
X.cumsum(axis=0)
```

[30]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  6.,  8., 10.],
        [12., 15., 18., 21.]])
```

3.8 广播机制

即使张量形状不同，我们仍然可以通过调用广播机制（broadcasting mechanism）来执行按元素操作。

[31]:
```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

[31]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

[32]:
```
a + b
```

[32]:
```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

思考：请你通过多次尝试，总结广播机制的计算规则。

3.9 点积和矩阵乘法

点积是相同位置的按元素乘积的和。

[33]:
```
x = torch.arange(4, dtype=torch.float32)
y = torch.ones(4, dtype=torch.float32)
x, y, torch.dot(x, y)
```

[33]:
```
(tensor([0., 1., 2., 3.]), tensor([1., 1., 1., 1.]), tensor(6.))
```

`torch.mm` 可实现矩阵乘法。

[34]:
```
A = torch.randn(3, 4)
B = torch.ones(4, 3)
A, B
```

[34]:
```
(tensor([[-0.3240,  0.8319,  0.0197, -0.1307],
         [ 0.1737, -0.8415,  0.2636,  1.2021],
         [-1.6382, -0.7520, -0.8513,  0.3146]]),
 tensor([[1., 1., 1.],
         [1., 1., 1.],
         [1., 1., 1.],
```

```
        [1., 1., 1.]]))
```

[35]:
```
torch.mm(A, B)
```

[35]:
```
tensor([[ 0.3969,  0.3969,  0.3969],
        [ 0.7979,  0.7979,  0.7979],
        [-2.9269, -2.9269, -2.9269]])
```

3.10 切片

张量可以方便地进行切片操作，即取出张量的某个部分。

注意，PyTorch 的索引规则与 Python 保持一致，索引从 0 开始。

例如，可以用 `[-1]` 选择最后一个元素，可以用 `[1:3]` 选择第二个和第三个元素。

[36]:
```
X
```

[36]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

[37]:
```
X[-1], X[1:3]
```

[37]:
```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

[38]:
```
X[0, 1]
```

[38]:
```
tensor(1.)
```

[39]:
```
X[0, 1:3]
```

[39]:
```
tensor([1., 2.])
```

[40]:
```
X[:2, 1:3]
```

[40]:
```
tensor([[1., 2.],
        [5., 6.]])
```

[41]:
```
X[:, 1:3]
```

[41]:
```
tensor([[ 1.,  2.],
        [ 5.,  6.],
        [ 9., 10.]])
```

除读取外，我们还可以通过指定索引来将元素写入张量。

[42]:
```
X
```

[42]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

[43]:
```
X[1, 2] = 9
X
```

[43]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  9.,  7.],
        [ 8.,  9., 10., 11.]])
```

为多个元素赋值相同的值，我们只需要索引所有元素，然后为它们赋值。

[44]:
```
X[0:2, :] = 12
X
```

[44]:
```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

3.11 原地操作（In-place operation）

运行一些操作可能会导致为新结果分配内存。

[45]:
```
before = id(Y)  # id 函数返回对象的唯一 ID，或者说（虚拟）内存地址
Y = Y + X
id(Y) == before
```

[45]:
```
False
```

执行原地操作则能够避免为结果重新分配内存。

[46]:
```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y  # 切片操作能够避免重新分配内存
print('id(Z):', id(Z))
```

```
id(Z): 2513883935984
id(Z): 2513883935984
```

如果在后续计算中没有重复使用 X，我们也可以使用 X[:] = X + Y 或 X += Y 来减少操作的内存开销。

[47]:
```
before = id(X)
X += Y
id(X) == before
```

[47]:
```
True
```

使用 clone 函数能够开辟新的内存空间，并得到一个数值上完全相同的张量。

[48]:
```
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
B = A.clone()
A, B
```

[48]:
```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.],
         [16., 17., 18., 19.]]),
 tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.],
         [16., 17., 18., 19.]]))
```

[49]:
```
id(A) == id(B)
```

[49]:
```
False
```

3.12 与 NumPy 数组或 Python 标量的相互转换

与 NumPy 数组的转换。

[50]:
```
A = X.numpy()
B = torch.tensor(A)
type(A), type(B)
```

[50]:
```
(numpy.ndarray, torch.Tensor)
```

将大小为 1 的张量转换为 Python 标量。

[51]:
```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

[51]:
```
(tensor([3.5000]), 3.5, 3.5, 3)
```

4 数据预处理

首先，我们创建一个人工数据集，并存储在 csv（逗号分隔值）文件。

[52]:
```
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\n')
    f.write('NA,Pave,127500\n')
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

从创建的 csv 文件中加载原始数据集。

[53]:
```
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms Alley   Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000
```

4.1 处理缺失值

为了处理缺失的数据，典型的方法包括插值和删除，这里，我们将考虑插值。

[54]:
```
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
```

[55]:
```
inputs
```

[55]:
```
   NumRooms Alley
0       NaN  Pave
1       2.0   NaN
2       4.0   NaN
3       NaN   NaN
```

[56]:
```
inputs = inputs.fillna(inputs.mean(numeric_only=True))
inputs
```

[56]:
```
   NumRooms Alley
0       3.0  Pave
1       2.0   NaN
2       4.0   NaN
3       3.0   NaN
```

4.2 类别值转换为数值

对于 inputs 中的类别值或离散值，我们将 “NaN” 视为一个类别。

[57]:
```
inputs = pd.get_dummies(inputs, dummy_na=True)
inputs
```

[57]:
```
   NumRooms  Alley_Pave  Alley_nan
0       3.0           1          0
```

```
1       2.0           0          1
2       4.0           0          1
3       3.0           0          1
```

现在 inputs 和 outputs 中的所有条目都是数值类型，它们可以转换为张量格式

[58]:
```
X, y = torch.tensor(inputs.values), torch.tensor(outputs.values)
X, y
```

[58]:
```
(tensor([[3., 1., 0.],
         [2., 0., 1.],
         [4., 0., 1.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500, 106000, 178100, 140000]))
```

5 自动微分

假设我们想对函数 $y = 2\mathbf{x}^{\mathrm{T}}\mathbf{x}$ 关于列向量 $\mathbf{x}$ 求导。

[59]:
```
x = torch.arange(4.0)
x
```

[59]:
```
tensor([0., 1., 2., 3.])
```

首先，我们要为 x 开启存储梯度的功能。

[60]:
```
x.requires_grad_(True)
print(x.grad)
```

```
None
```

现在让我们计算 y。

[61]:
```
y = 2 * torch.dot(x, x)
y
```

[61]:
```
tensor(28., grad_fn=<MulBackward0>)
```

通过调用反向传播函数来自动计算 y 关于 x 每个分量的梯度。

[62]:
```
y.backward()
print(x.grad)
```

```
tensor([ 0.,  4.,  8., 12.])
```

[63]:
```
x.grad == 4 * x
```

[63]:
```
tensor([True, True, True, True])
```

现在让我们计算 x 的另一个函数。

[64]:
```
# 在默认情况下，PyTorch 会累积梯度，我们需要清除之前的值
x.grad.zero_()  # 方法后以'_'结尾代表原地修改值
y = x.sum()
y.backward()
print(x.grad)
```

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

在 PyTorch 中有个简单的规定，不允许向量对向量求导，只允许标量对向量求导。因此，一个向量调用 backward()，则需要传入一个 gradient 参数。传入这个参数就是为了把向量对向量的求导转换为标量对向量的求导。

[65]:
```
x.grad.zero_()
y = x * x  # 向量元素相乘，得到的仍是向量，此时存在向量对向量求导的情况
y.sum().backward()  # 等价于 y.backward(torch.ones(len(x))) ；这里我们首先将向量 y 降维成标量
print(x.grad)
```

```
tensor([0., 2., 4., 6.])
```

将某些计算移动到记录的计算图之外。

[66]:
```
x.grad.zero_()
y = x * x
u = y.detach()  # 把 y 的值赋给 u，u 不进入计算图，作为常数处理
z = u * x

z.sum().backward()
x.grad == u
```

[66]:
```
tensor([True, True, True, True])
```

[67]:
```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

[67]:
```
tensor([True, True, True, True])
```

即使构建函数的计算图需要通过 Python 控制流（例如，条件、循环或任意函数调用），我们仍然可以计算得到的变量的梯度。

[68]:
```
def f(a):
    b = a * 2
    while b.norm() < 1000:  # b.norm: 求 L2 范数
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c

a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()

a.grad == d / a
```

[68]:
```
tensor(True)
```

6 搭建全连接神经网络

在本节中，我们将搭建一个全连接神经网络，用于解决图像的分类问题。

6.1 数据集下载

本实验将使用Fasion-MNIST数据集。该数据集是一个旨在替代原始的 MNIST 手写数字数据集的简易图像分类数据集，由 Zalando（一家德国的时尚科技公司）旗下的研究部门提供，包含 10 个类

别、共 7 万张不同商品的正面图片。Fashion-MNIST 的大小、格式和训练集/测试集划分与原始的 MNIST 完全一致，即包含 60000 张训练图片，10000 张测试图片，每张图片均为 28x28 的灰度图。

首先，导入必要的包。

[69]:
```
import torch
from torch import nn, optim
import torch.nn.functional as F
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

import os
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"
```

定义一个图像预处理器，用于将图像数据标准化，使得每个像素的灰度值从 0 到 1 转换为-1 到 +1 之间。

[70]:
```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
↪5,), (0.5,))])
```

接下来，下载数据并定义训练和测试集数据加载器（DataLoader）。数据集下载约需几分钟时间。

如果下载速度太慢，可通过以下链接手动下载[Link]（提取码：dju3），并将 `train-images-idx3-ubyte.gz` 等 4 个文件放在 `dataset/FashionMNIST/raw` 目录下，之后重新执行下面的代码。

[71]:
```
# 下载 Fashion-MNIST 训练集数据，构建训练集数据加载器 trainloader，每次从训练集中
载入 64 张图片，每次载入都打乱顺序
trainset = datasets.FashionMNIST(root='dataset/', download=True, train=True,␣
↪transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

# 下载 Fashion-MNIST 测试集数据，构建测试集数据加载器 testloader，每次从测试集中载
入 64 张图片，每次载入都打乱顺序
testset = datasets.FashionMNIST(root='dataset/', download=True, train=False,␣
↪transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=True)
```

6.2 查看数据样例

[72]:
```
image, label = next(iter(trainloader))
print(image.shape)
print(label)
```

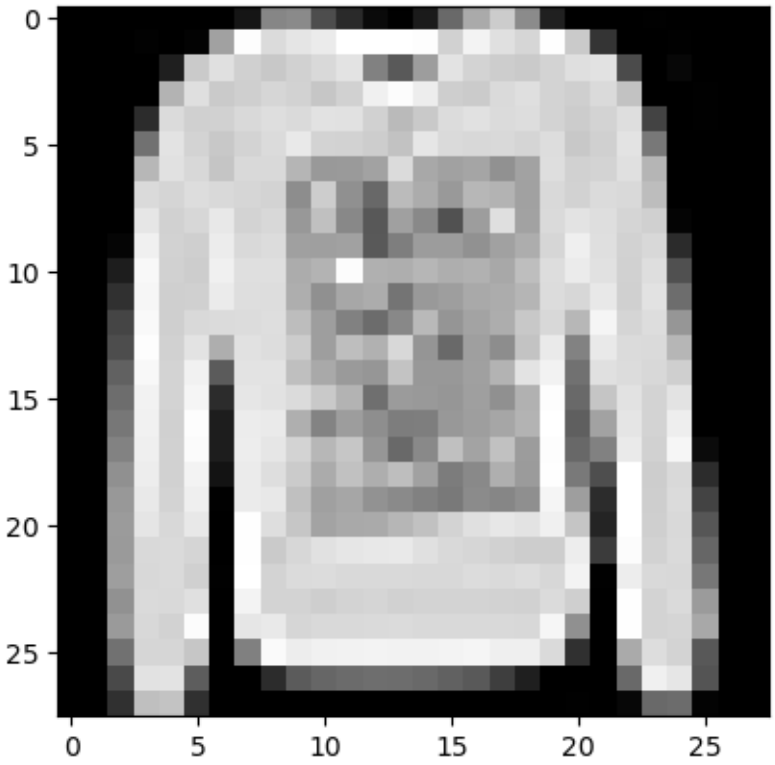
```
torch.Size([64, 1, 28, 28])
tensor([2, 0, 0, 1, 7, 8, 0, 4, 7, 5, 7, 3, 6, 4, 1, 6, 8, 7, 8, 3, 2, 8, 2, 7,
        3, 2, 4, 2, 2, 1, 3, 9, 3, 2, 0, 4, 4, 0, 5, 2, 7, 4, 0, 3, 6, 3, 4, 2,
        4, 2, 4, 3, 0, 0, 7, 1, 0, 3, 8, 7, 2, 9, 6, 4])
```

label 的形式为 0-9 的整数，分别代表以下 10 个类别：

标注编号	描述
0	T-shirt/top（T 恤）
1	Trouser（裤子）
2	Pullover（套衫）
3	Dress（裙子）
4	Coat（外套）
5	Sandal（凉鞋）
6	Shirt（汗衫）
7	Sneaker（运动鞋）
8	Bag（包）
9	Ankle boot（踝靴）

[73]:
```
# image 图片中有 64 张图片，我们查看索引为 0 的图片
imagedemo = image[0]
imagedemolabel = label[0]

imagedemo = imagedemo.reshape((28,28))

plt.imshow(imagedemo, cmap='gray')

labellist = ['T 恤','裤子','套衫','裙子','外套','凉鞋','汗衫','运动鞋','包','靴子']
print(f'这张图片对应的标签是 {labellist[imagedemolabel]}')
```

```
这张图片对应的标签是 套衫
```



6.3 搭建神经网络

让我们搭建一个简单的全连接神经网络，实现 Fashion-MNIST 数据集的图像分类任务。

首先，设计神经网络的结构，我们这里定义一个 4 层全连接神经网络：

- **输入层**：由于图片长、宽均为 28 像素，则输入元素数量为 28 * 28 = 784；
- **隐藏层 1**：包含 256 个神经元；
- **隐藏层 2**：包含 128 个神经元；
- **隐藏层 3**: 包含 64 个神经元；
- **输出层**：输出 10 个结果，对应图片属于 10 个分类的概率。

[74]:
```
class Classifier(nn.Module):
    """
    图像分类器，其结构为 4 层全连接神经网络。
    """
    def __init__(self):
```

```
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 10)

    def forward(self, x):
        # 将输入张量展平到 1 维
        x = x.view(x.shape[0], -1)

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.log_softmax(self.fc4(x), dim=1)

        return x
```

6.4 训练模型

定义损失函数，并采用适当的优化器，对模型进行训练。

[75]:
```
# 对上面定义的 Classifier 类进行实例化
model = Classifier()

# 定义损失函数为负对数损失函数
criterion = nn.NLLLoss()

# 优化方法为 Adam 梯度下降方法，学习率为 0.003
optimizer = optim.Adam(model.parameters(), lr=0.003)

# 对训练集的全部数据学习 15 遍，这个数字越大，训练时间越长
epochs = 15

# 将每次训练的训练误差和测试误差存储在这两个列表里，后面绘制误差变化折线图用
train_losses, test_losses = [], []
```

```
print('开始训练')
for e in range(epochs):
    running_loss = 0

    # 对训练集中的所有图片都过一遍
    for images, labels in trainloader:
        # 将优化器中的求导结果都设为 0，否则会在每次反向传播之后叠加之前的梯度
        optimizer.zero_grad()

        # 对 64 张图片进行推断，计算损失函数，反向传播优化权重，将损失求和
        log_ps = model(images)
        loss = criterion(log_ps, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    # 每次学完一遍数据集，都进行以下测试操作
    else:
        test_loss = 0
        accuracy = 0
        # 测试的时候不需要开自动求导和反向传播
        with torch.no_grad():
            # 关闭 Dropout
            model.eval()

            # 对测试集中的所有图片都过一遍
            for images, labels in testloader:
                # 对传入的测试集图片进行正向推断、计算损失，accuracy 为测试集一万张
图片中模型预测正确率
                log_ps = model(images)
                test_loss += criterion(log_ps, labels)
                ps = torch.exp(log_ps)
                top_p, top_class = ps.topk(1, dim=1)
                equals = top_class == labels.view(*top_class.shape)

                # 等号右边为每一批 64 张测试图片中预测正确的占比
```

```
            accuracy += torch.mean(equals.type(torch.FloatTensor))
        # 恢复 Dropout
        model.train()
        # 将训练误差和测试误差存在两个列表里，后面绘制误差变化折线图用
        train_losses.append(running_loss/len(trainloader))
        test_losses.append(test_loss/len(testloader))

        print("训练集学习次数: {}/{}; ".format(e+1, epochs),
              "训练误差: {:.3f}; ".format(running_loss/len(trainloader)),
              "测试误差: {:.3f}; ".format(test_loss/len(testloader)),
              "模型分类准确率: {:.3f}.".format(accuracy/len(testloader)))
```

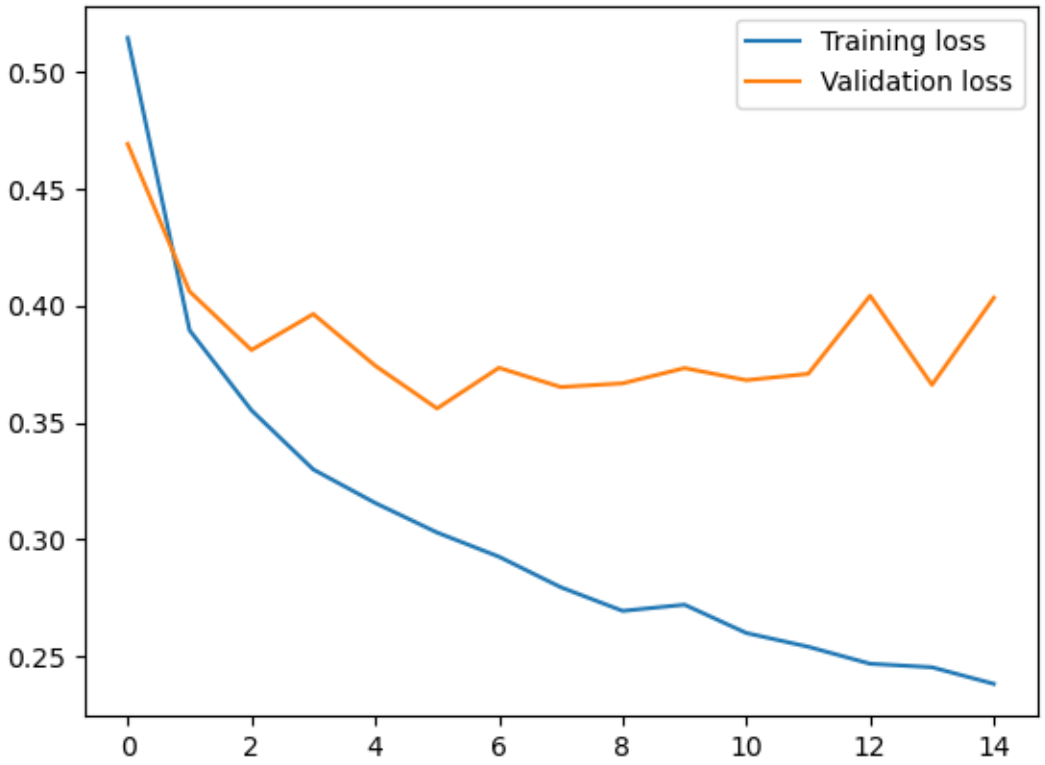
开始训练

```
训练集学习次数: 1/15..  训练误差: 0.514..  测试误差: 0.469..  模型分类准确率: 0.831
训练集学习次数: 2/15..  训练误差: 0.389..  测试误差: 0.406..  模型分类准确率: 0.854
训练集学习次数: 3/15..  训练误差: 0.355..  测试误差: 0.381..  模型分类准确率: 0.860
训练集学习次数: 4/15..  训练误差: 0.330..  测试误差: 0.396..  模型分类准确率: 0.859
训练集学习次数: 5/15..  训练误差: 0.316..  测试误差: 0.374..  模型分类准确率: 0.869
训练集学习次数: 6/15..  训练误差: 0.303..  测试误差: 0.356..  模型分类准确率: 0.874
训练集学习次数: 7/15..  训练误差: 0.293..  测试误差: 0.373..  模型分类准确率: 0.868
训练集学习次数: 8/15..  训练误差: 0.280..  测试误差: 0.365..  模型分类准确率: 0.872
训练集学习次数: 9/15..  训练误差: 0.270..  测试误差: 0.367..  模型分类准确率: 0.875
训练集学习次数: 10/15..  训练误差: 0.272..  测试误差: 0.373..  模型分类准确率: 0.
↪879
训练集学习次数: 11/15..  训练误差: 0.260..  测试误差: 0.368..  模型分类准确率: 0.
↪874
训练集学习次数: 12/15..  训练误差: 0.254..  测试误差: 0.371..  模型分类准确率: 0.
↪878
训练集学习次数: 13/15..  训练误差: 0.247..  测试误差: 0.404..  模型分类准确率: 0.
↪858
训练集学习次数: 14/15..  训练误差: 0.246..  测试误差: 0.366..  模型分类准确率: 0.
↪878
训练集学习次数: 15/15..  训练误差: 0.238..  测试误差: 0.403..  模型分类准确率: 0.
↪871
```

可视化训练和测试损失。

[76]:
```
plt.plot(train_losses, label='Training loss')
plt.plot(test_losses, label='Validation loss')
plt.legend()
```

[76]:
```
<matplotlib.legend.Legend at 0x24953ecf340>
```

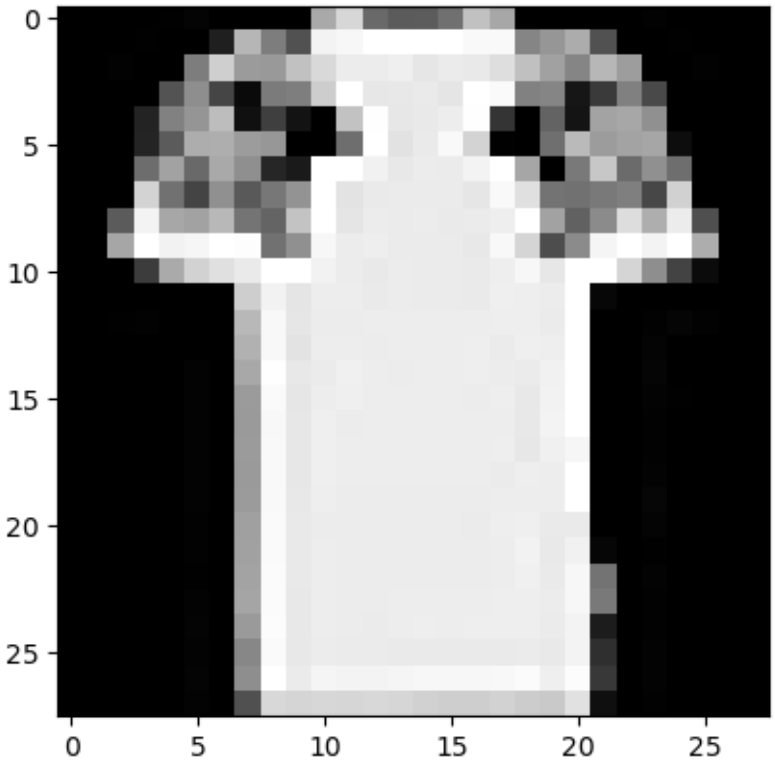


6.5 基于模型进行推理

[77]:
```
dataiter = iter(testloader)
images, labels = next(dataiter)
img = images[0]
img = img.reshape((28,28)).numpy()
plt.imshow(img, cmap='gray')

# 将测试图片转为一维的列向量
img = torch.from_numpy(img)
```

```
img = img.view(1, 784)

# 进行正向推断，预测图片所在的类别
with torch.no_grad():
    output = model.forward(img)
ps = torch.exp(output)

top_p, top_class = ps.topk(1, dim=1)
labellist = ['T 恤','裤子','套衫','裙子','外套','凉鞋','汗衫','运动鞋','包','靴子']
prediction = labellist[top_class]
probability = float(top_p)
print(f'神经网络猜测图片里是 {prediction}，概率为{probability*100}%')
print(f'正确标签为 {labellist[labels[0]]}')
```

神经网络猜测图片里是 T 恤，概率为 99.88006353378296%
正确标签为 T 恤



反复运行上面的代码，你会发现大多数情况下模型都给出了正确的预测，但也有一定概率推断错误，模型性能还有很大的提升空间。在本课程后续实验中，你将学习更加强大的模型，通过这些模型，能够进一步提升分类准确率。