5.4 PyTorch 深度学习软件包

PyTorch 起源于 Facebook 公司的深度学习框架——Torch，在底层 Torch 框架的基础上，使用 Python 对其进行了重写，使得 PyTorch 在支持 GPU 的基础上实现了与 Numpy 的无缝衔接。另外，PyTorch 还提供 torchaudio（用于处理音频），torchtext（用于处理文本），torchvision（用于处理视频）等专门的库，内置大量已经预训练的深度神经网络和高质量的训练数据集，这些库为直接使用经典神经网络进行项目开发和迁移学习提供了便利。

PyTorch 的内容博大精深，本书不可能覆盖所有方面，本小节仅针对在后文中要使用到的内容及关键知识点进行简单介绍。本书的使用的 PyTorch 版本是 1.9.0。

5.4.1 数据类型及类型的转换

1. Tensor 数据类型

PyTorch 的基本数据结构是张量（Tensor），所有的计算都是通过张量来进行的。PyTorch 中的张量和 Numpy 中的数组（ndarray）具有极高的相似度，二者可以相互转换。唯一不同的是，Tensor 可以在 GPU 上运行，而 Numpy 中的 ndarray 只能在 CPU 上运行。PyTorch 的这种设计是为了充分利用 Numpy 丰富的数组处理函数，同时又兼顾了 GPU 的高计算性能。

PyTorch 支持的数据类型包括浮点型、复数型、整型和布尔型，它们的相关信息如表 5-3 所示。

表 5-3 PyTorch 数据类型

数据类型	dtype	CPU tensor	GPU tensor
32-bit floating point	torch.float32 或 torch.float	torch.FloatTensor	torch.cuda.FloatTensor
64-bit floating point	torch.float64 或 torch.double	torch.DoubleTensor	torch.cuda.DoubleTensor
16-bit floating point	torch.float16 或 torch.half	torch.HalfTensor	torch.cuda.HalfTensor
32-bit complex	torch.complex32		
64-bit complex	torch.complex64		
128-bit complex	torch.complex128 或 torch.cdouble		
8-bit integer (unsigned)	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
8-bit integer (signed)	torch.int8	torch.CharTensor	torch.cuda.CharTensor

数据类型	dtype	CPU tensor	GPU tensor
16-bit integer (signed)	torch.int16 或 torch.short	torch.ShortTensor	torch.cuda.ShortTensor
32-bit integer (signed)	torch.int32 或 torch.int	torch.IntTensor	torch.cuda.IntTensor
64-bit integer (signed)	torch.int64 或 torch.long	torch.LongTensor	torch.cuda.LongTensor
Boolean	torch.bool	torch.BoolTensor	torch.cuda.BoolTensor

PyTorch 在构造张量时，浮点型默认使用 torch.float32，整型默认使用 torch.int64，也可以在定义时指定数据类型，可以通 dtype 属性来查看张量的数据类型，如：

```
import torch
import numpy as np

# In[Tensor 默认数据类型]
a = torch.rand((3,))
print(a.dtype)

b = torch.randint(1,10,(3,))
print(b.dtype)

# In[Tensor 指定数据类型]
a = torch.rand((3,),dtype=torch.float64)
print(a.dtype)

b = torch.randint(1,10,(3,),dtype=torch.int32)
print(b.dtype)
```

运行结果如下：

```
torch.float32
torch.int64
torch.float64
torch.int32
```

2. Tensor 数据类型转换

可以通过在 Tensor 后面加上数据类型的方式来改变 Tensor 的数据类型，如：

```
# In[Tensor 数据类型转换]
a = torch.rand((3,))
print(a,a.dtype)

b = a.int()
print(b,b.dtype)

c = b.float()
print(c,c.dtype)
```

运行结果如下：

```
tensor([0.9472, 0.5125, 0.2198]) torch.float32
tensor([0, 0, 0], dtype=torch.int32) torch.int32
```

```
tensor([0., 0., 0.]) torch.float32
```

值得注意的是，从 torch.float32 转成 torch.int32 时，只保留了原来数据的整数部分，所以再转回 torch.float32 后就和原来的数据就不相等了，从上面运行结果也可以看出 $a \neq c$ 。

1. Tensor 和 ndarray 相互转换

PyTorch 提供了 Numpy 的 ndarray 数据和 Tensor 相互转换的工具，这在实际编程中非常适用和重要，因为大部分原始和生成数据最初都是 ndarray 格式的，在将它们灌入深度神经网络进行训练之前需要先将其转换成 Tensor；另外，从深度神经网络输出的数据都是 Tensor，要对它们进行一般的计算，又要先转换成 ndarray 格式。

使用 from_numpy 可以将由 Numpy 生成的 ndarray 数据转换成相应的 Tensor，也可以使用 torch.FloatTensor()函数来生成 ndarray 数组对应的 Tensor，如：

```
# In[ndarray 转 Tensor]
a = np.array([1.,2.,3.])
tensor_a1 = torch.from_numpy(a)
tensor_a2 = torch.FloatTensor(a)
print(a,a.dtype)
print(tensor_a1,tensor_a1.dtype)
print(tensor_a2,tensor_a2.dtype)
```

运行结果如下：

```
[1. 2. 3.] float64
tensor([1., 2., 3.], dtype=torch.float64) torch.float64
tensor([1., 2., 3.]) torch.float32
```

使用 Tensor 的 numpy 功能函数可以将 Tensor 转换成相应的 ndarray 数据格式，也可以直接使用 numpy.array()函数来生成与 Tensor 相应的数组，如：

```
# In[Tensor 转 ndarray]
t = torch.rand((3,))
array_t1 = t.numpy()
array_t2 = np.array(t)
print(t,t.dtype)
print(array_t1,array_t1.dtype)
print(array_t2,array_t2.dtype)
```

运行结果如下：

```
tensor([0.2360, 0.0514, 0.6417]) torch.float32
[0.23601848 0.05141479 0.64165056] float32
[0.23601848 0.05141479 0.64165056] float32
```

Tensor 和 ndarray 数据格式的相互转换是在深度神经网络编程中经常用到，但又非常容易出错的部分，在编程过程中一定要清楚各个数据是 Tensor 还是 ndarray，以及它们的位数。一般只在神经网络内部使用 Tensor，在其它地方均使用 ndarray，浮点数位数一般设置为 float32。

5.4.2 张量的维度和重组操作

1. 维度的定义

在 PyTorch 中，张量的维度是一个重要概念，许多操作都和维度有关。从形式上看，张量和 Numpy 中的多维数组一样，其中 0 维张量表示标量，即一个数；1 维张量表示向量，即 1 维数组；2 维张量表示矩阵，即 2 维数组；多维张量相当于多维数组。张量的维度计数从 0 维开始，自外向里计数，在 Tensor 中的关键字是 axis 或 dim。

这里需要注意的是张量的维度和每一维上的分量数是两个概念，但是人们一般都将两者都称为维数。为了区别，本书将张量每一维上的分量的个数称为分量数。

以一个 3 维张量为例，如：

```
t = torch.Tensor(np.arange(24)).reshape((2,3,4))
```

运行结果为：

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

        [[12., 13., 14., 15.],
         [16., 17., 18., 19.],
         [20., 21., 22., 23.]]])
```

这里 t 是一个 3 维张量，第 0，1，2 维上的分量数分别为 2，3，4。

可以采用剥掉中括号的方法来确认张量各维度的分量具体是什么。将 t 最外层的中括号剥掉，得到两个尺寸为 3×4 的 2 维张量，即

```
[[ 0.,  1.,  2.,  3.],
  [ 4.,  5.,  6.,  7.],
  [ 8.,  9., 10., 11.]],

[[12., 13., 14., 15.],
  [16., 17., 18., 19.],
  [20., 21., 22., 23.]]
```

这就是 t 的第 0 维上的两个分量。若继续将第 1 个 2 维张量的最外层中括号剥掉，则得到 3 个分量数为 4 的 1 维张量，即

```
[ 0.,  1.,  2.,  3.],
[ 4.,  5.,  6.,  7.],
[ 8.,  9., 10., 11.]
```

这就是 t 的第 1 维上的分量。若继续将第 1 个 1 维张量的最外层中括号剥掉，则得到 4 个标量，即 0 维张量，即

```
0.,  1.,  2.,  3.
```

这就是 t 的第 2 维上的分量。

弄清楚张量各维度具体如何得到以后，与张量维度有关的操作就容易理解了。比如对 t

的第 0 维求和，就是将第 0 维上的两个 2 维张量相加，即

```
print(t.sum(dim=0)) # print(t.sum(axis=0))
```

运行结果为：

```
tensor([[12., 14., 16., 18.],
        [20., 22., 24., 26.],
        [28., 30., 32., 34.]])
```

可见得到的是一个尺寸为 3×4 的 2 维张量。若对 t 的第 1 维求和，就是将第两个 2 维张量上的 1 维张量分别相加，即

```
print(t.sum(dim=1))
```

运行结果为：

```
tensor([[12., 15., 18., 21.],
        [48., 51., 54., 57.]])
```

可见得到的是一个尺寸为 2×4 的 2 维张量。若对 t 的第 2 维求和，就是将第 2 维上的标量分别相加，即

```
print(t.sum(dim=2))
```

运行结果为：

```
tensor([[ 6., 22., 38.],
        [54., 70., 86.]])
```

可见得到的是一个尺寸为 2×3 的 2 维张量。

总而言之，对张量某一维的操作要先弄清这一维上的分量是什么才不至于出错。值得注意的是，Tensor 和 ndarray 都没有行向量和列向量的概念，这和 MATLAB 是很不一样的，熟悉 MATLAB 编程的读者要注意区分。

2. 张量的维度重组

张量的维度重组使用 view 或 reshape 函数，这两个函数的功能基本相同，如

```
# In[张量的维度重组]
t = torch.Tensor(np.arange(24)).reshape((2,3,4))
t1 = t.view(2,2,6)
t2 = t.reshape(8,-1)
print(t)
print(t1)
print(t2)
```

运行结果如下：

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

        [[12., 13., 14., 15.],
         [16., 17., 18., 19.],
         [20., 21., 22., 23.]]])
tensor([[[ 0.,  1.,  2.,  3.,  4.,  5.],
         [ 6.,  7.,  8.,  9., 10., 11.]],
```

```
       [[12., 13., 14., 15., 16., 17.],
        [18., 19., 20., 21., 22., 23.]]])
tensor([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.],
       [ 9., 10., 11.],
       [12., 13., 14.],
       [15., 16., 17.],
       [18., 19., 20.],
       [21., 22., 23.]])
```

可以看出，数据重组的方式是原始数据按照最后一维依次填入重组后的张量。t2 中的-1 会自动计算剩下维度的分量数，即 $(2\times3\times4)/8$ 。

3. 张量的维度添加和压缩

在神经网络数据流动过程中经常需要将张量的维度进行对齐，这就需要对张量的维度进行添加或压缩。

维度添加使用 unsequeeze 函数，用于在张量的指定维度上添加一维；维度压缩使用 squeeze，用于压缩分量数为 1 的维度；如：

```
# In[维的添加和压缩]
t = torch.Tensor(np.arange(6)).reshape((2,3))
t1 = t.unsqueeze(dim=0)
t2 = t1.unsqueeze(dim=3)
t3 = t1.squeeze()
t4 = t2.squeeze()
print(t,t.shape)
print(t1,t1.shape)
print(t2,t2.shape)
print(t3,t3.shape)
print(t4,t4.shape)
```

运行结果为：

```
tensor([[0., 1., 2.],
       [3., 4., 5.]]) torch.Size([2, 3])
tensor([[[0., 1., 2.],
        [3., 4., 5.]]]) torch.Size([1, 2, 3])
tensor([[[[0.],
         [1.],
         [2.]],

        [[3.],
         [4.],
         [5.]]]]) torch.Size([1, 2, 3, 1])
tensor([[0., 1., 2.],
       [3., 4., 5.]]) torch.Size([2, 3])
tensor([[0., 1., 2.],
       [3., 4., 5.]]) torch.Size([2, 3])
```

上例中 t 是一个尺寸为 2×3 的 2 维张量；t1 在 t 的第 0 维上添加 1 维，故 t1 是一个尺寸为 1×2×3 的 3 维张量；t2 再在 t1 的第 3 维上增加一维，将 t1 的每一个标量元素变成 1 维张量，故 t2 是一个尺寸为 1×2×3×1 的 4 维张量；最后 t3 和 t4 分别将 t1 和 t2 的所有分量数为 1 的

维度压缩。squeeze 压缩维度时也可以指定压缩某一维度，如：

```
t5 = t2.squeeze(dim=3)
print(t5,t5.shape)
```

运行结果如下：

```
tensor([[[0., 1., 2.],
         [3., 4., 5.]]]) torch.Size([1, 2, 3])
```

可以看出，只压缩了 t2 的第 3 维。

4. 张量的转置

用于张量转置的有 3 个函数 t，transpose 和 permute。t 只能用于 2 维张量，和矩阵转置一样，如：

```
t = torch.Tensor(np.arange(6)).reshape((2,3))
t_t = t.t()
print(t_t,t_t.shape)
```

运行结果如下：

```
tensor([[0., 3.],
        [1., 4.],
        [2., 5.]]) torch.Size([3, 2])
```

transpose 函数用于张量的某两个维度的转置，如：

```
t = torch.Tensor(np.arange(24)).reshape((2,3,4))
t_trans = t.transpose(0,1) # 第 0 和 1 维转置
print(t,t.shape)
print(t_trans,t_trans.shape)
```

运行结果如下：

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

        [[12., 13., 14., 15.],
         [16., 17., 18., 19.],
         [20., 21., 22., 23.]]]) torch.Size([2, 3, 4])
tensor([[[ 0.,  1.,  2.,  3.],
         [12., 13., 14., 15.]],

        [[ 4.,  5.,  6.,  7.],
         [16., 17., 18., 19.]],

        [[ 8.,  9., 10., 11.],
         [20., 21., 22., 23.]]]) torch.Size([3, 2, 4])
```

permute 给出维度转置的一个排列方式，如：

```
t_perm = t.permute((1,0,2)) # 将维度按照 1,0,2 方式转置，即（3,2,4）
print(t_perm,t_perm.shape)
```

运行结果如下：

```
tensor([[[ 0.,  1.,  2.,  3.],
         [12., 13., 14., 15.]],
```

```
       [[ 4.,  5.,  6.,  7.],
        [16., 17., 18., 19.]],

       [[ 8.,  9., 10., 11.],
        [20., 21., 22., 23.]]]) torch.Size([3, 2, 4])
```

张量的转置比较容易造成数据混乱，在实际中应尽量减少使用。

5. 张量的广播

张量的常规加减乘除运算只有在相同尺寸的张量上才能够进行，但在实际计算中经常会遇到一个 2 维张量加一个 1 维张量的问题，这就需要先将 1 维张量进行复制，得到一个和 2 维张量尺寸一样的张量以后，再进行加法运算，这就是张量的广播机制，如

```
t1 = torch.Tensor(np.arange(6)).reshape((2,3))
t2 = torch.ones((3,))
t3 = t1+t2
print(t3)
```

运行结果如下：

```
tensor([[1., 2., 3.],
       [4., 5., 6.]])
```

这里 t1 是一个尺寸为 2×3 的 2 维张量，但 t2 是一个分量数为 3 的 1 维张量，所以需要先将 t2 复制两次，成为一个尺寸也为 2×3 的 2 为张量，即

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

才能和 t1 相加。值得注意的是，只能对分量数为 1 的维度进行广播，而且被广播的张量维度也要合适才行。比如，若上例中 t2 是一个分量数为 4 的 1 维张量，则会报错。

5.4.3 组装神经网络的模块

用 PyTorch 构建神经网络就像搭积木一样，只需要使用已经封装好的模块，按照约定的结构搭建即可。这些封装好的模块都放在 torch.nn 模块库中，包括卷积层（Convolution Layers），池化层（Pooling Layers），边界填充层（Padding Layers），激活函数（Activation Functions），线性层（Linear Layers），等。本小节选择介绍一些后文中要用到的模块。

torch.nn 中的模块都是以类的形式给出的，在使用它们时要先创建一个类的实例，然后传入参数进行使用。

1. 线性层

线性层是神经网络最基本的映射层，用公式表示是

$$y = xA^T + b$$

线性层使用的模块类是 Linear，调用语法为

torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)

其中

- in_features：输入 x 的维度
- out_features：输出 y 的维度
- bias：是否添加偏置，默认添加
- device：计算设备为 CPU 还是 GPU，默认为 CPU
- dtype：数据类型，默认为 torch.float32

线性层支持批量数据运算，可以一次性输入一个批量的数据，如

```
# In[线性层]
B = 10                                  # batch-size
linear_layer = nn.Linear(20,30)         # 创建线性层实例
x = torch.randn(B,20)                   # 输入批量数据，单个输入维度为 20
y = linear_layer(x)                     # 线性层映射，输出 y
print(y.size(),y.dtype)                 # 查看输出的尺寸
```

运行结果为：

```
torch.Size([10, 30]) torch.float32
```

线性层默认支持的数据类型是 torch.float32。可以在 dtype 关键字中修改数据类型，比如修改为 dtype=torch.float64，这时 x 在创建时也要使用相同的数据类型，否则会报错。使用默认数据类型对于实际问题来说精度已经足够了。

上例中创建的 linear_layer 是 torch.nn.Linear 类的一个实体，有自己的属性和功能函数，可以通过 dir 函数查询这些属性和功能函数名，一般用得较多的是查询其权重和偏置，如：

```
dir(linear_layer)               # 查询 linear_layer 的所有属性和功能函数名
print(linear_layer.weight)      # 查询权重
print(linear_layer.bias)        # 查询偏置
```

2. 激活函数

激活函数是神经网络非线性性的唯一来源，不可或缺。关于常见激活函数的具体表达式和导数已经在第 5.3.1 节中详细介绍，不再赘述。此处以最常用的激活函数 ReLU 来介绍激活函数的使用方法。

ReLU 函数的调用语法为

torch.nn.ReLU(inplace=False)

其中

- inplace：输出数据是否直接使用输入数据的内存，默认为 False，即使用新的内存输出数据。inplace=True 可以节省内存空间，省去了反复申请和释放内存的时间，但会覆盖输入数据。

ReLU 函数是单变量函数，它会分别作用于输入张量的每一个标量数据，所以输出数据和输入数据具有相同的尺寸，如

```
# In[ReLU 激活函数]
B = 10                          # batch-size
relu = nn.ReLU()                # 创建 ReLU 函数实体
```

```
x = torch.randn(B,20)          # 输入批量数据，单个输入维度为 20
y = relu(x)                    # ReLU 函数映射
print(x.shape,y.shape)         # 输入输出尺寸一样
```

运行结果为

```
torch.Size([10, 20]) torch.Size([10, 20])
```

可见输入数据和输出数据的尺寸一样。

3. 损失函数

损失函数是在训练神经网络时必须的模块，也放在 torch.nn 模块库中。常见损失函数的具体原理和表达式已经在第 5.3.2 节详细介绍，不再赘述。此处以常用的均方误差损失函数为例介绍损失函数的使用方法。

均方误差损失函数的调用语法为

torch.nn.MSELoss(reduction='mean')

其中

- reduction：取'none'，'mean'或'sum'，若 reduction='mean'，则输出批量运算结果之和按输入尺度平均后的标量；若 reduction='sum'，则输出批量运算结果之和按输入尺度相加后的标量和；若 reduction='none'，则只输出批量和，在输入尺度上不做处理，输入数据和输出数据有相同的尺度。默认为 reduction='mean'。

  均方误差损失函数也支持批量运算，可以输入一个批量的数据，如

```
# In[MSELoss 函数]
B = 10                                    # batch-size
loss = nn.MSELoss()                       # 创建 MSELoss 函数，reduction='mean'
loss_sum = nn.MSELoss(reduction='sum') # 创建 MSELoss 函数，reduction='sum'
loss_none = nn.MSELoss(reduction='none')# 创建 MSELoss 函数，reduction='none'
y_hat = torch.randn((3,5))                # 预测输出
y_tar = torch.randn((3,5))                # 目标输出
out = loss(y_hat,y_tar)                   # 损失函数值
out_sum = loss_sum(y_hat,y_tar)
out_none = loss_none(y_hat,y_tar)
print(out,out.shape)
print(out_sum,out_sum.shape)
print(out_none,out_none.shape)
```

运行结果为：

```
tensor(1.4108) torch.Size([])
tensor(21.1614) torch.Size([])
tensor([[1.1905e-02, 4.8586e-04, 1.0308e+00, 1.2013e+00, 9.7473e-02],
        [1.5531e+00, 6.4626e-01, 8.4639e-01, 3.3398e+00, 7.0021e+00],
        [1.6918e+00, 2.0903e+00, 2.6002e-02, 1.3863e+00, 2.3753e-01]])
torch.Size([3, 5])
```

上例中输入尺度是 3×5，所以 out_sum/15=out，而 out_none 的所有元素之和等于 out_sum。

5.4.4 自动梯度计算

在神经网络的误差反向传播过程中，梯度计算是一个必不可少的步骤，PyTorch 针对这一步骤专门开发了一个自动梯度计算引擎 torch.autograd，它支持任何计算图的自动梯度计算。本小节以一个简单的函数为例来介绍相关概念及操作。

1. 计算图

考虑二次函数

$$y = x^T W x + b^T x + b^T b \tag{5-49}$$

式（5-49）的计算过程可以用如图 5-15 所示的一个计算流程图来表示

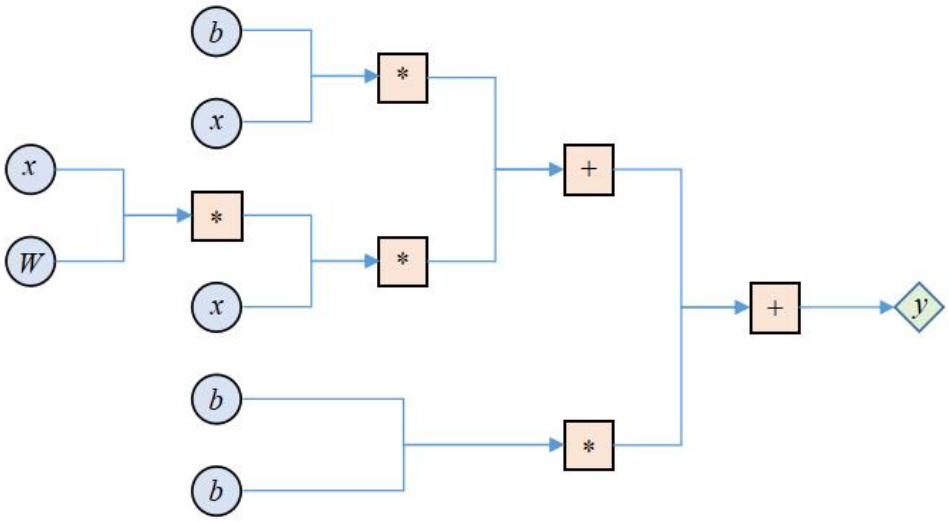

图 5-15 计算图

在图 5-15 中，圆圈内的元素表示输入到流程图的计算单元，称为叶节点（Leaf Node）；方框内的符号表示各计算单元的运算方式，称为计算节点（Computation Node）；菱形内的元素是计算流程的最后结果，称为根节点（Root Node）。

torch.autograd 引擎计算梯度的过程是这样的：首先，在前向传播过程中建立计算图，并保留一些计算梯度需要的中间结果；然后，根据计算图自动计算用链式法则计算变量梯度所需要的中间函数；最后，利用这些中间函数和前向传播中保留的中间结果根据链式法则计算各变量的梯度。整个过程代码如下：

```
import torch

# In[搭建计算图]
x = torch.ones((5,),requires_grad=True)          # 变量，需要计算梯度
W = torch.ones((5,5),requires_grad=False)        # 参数，不需要计算梯度
b = torch.ones((5,))                             # 参数，默认不需要计算梯度
Q = torch.matmul(torch.matmul(x,W),x)            # 二次项，中间结果
L = torch.matmul(b,x)                            # 一次项，中间结果
C = torch.matmul(b,b)                            # 常数项
y = Q+L+C                                        # 前向传播，建立计算图
```

```
                                                    # 查看需要求梯度的量
print(x.requires grad,Q.requires grad,C.requires grad,y.requires grad)
print(y.grad fn) # 对最终结果 y 的梯度函数
print(Q.grad fn) # 对中间结果 Q 的梯度函数
print(C.grad_fn) # 对中间结果 C 的梯度函数
print(x.grad_fn) # 对叶节点 x 的梯度函数
```

运行结果如下：

```
True True False True
<AddBackward0 object at 0x0000023A5DCA4208>
<DotBackward object at 0x0000023A5DCA4208>
None
None
```

每一个张量都有一个 requires_grad 属性，为 True 时表示该张量需要计算梯度，为 False 即为该张量不需要计算梯度。输入张量 x，W，b 的 requires_grad 属性是在输入时给定的，默认为 False；中间结果和输出结果是否需要求梯度要根据链式法则来定，比如在上例中 Q 和 y 需要求梯度，但作为常数项的 C 就不需要求梯度。

同样，根据求梯度的链式法则，要对 x 求梯度，首先要求输出结果 y 的梯度函数，然后要求中间结果 Q 和 L 的梯度函数，但 C 不需要求梯度函数，因为 C 是常数项，不参与梯度计算，x 也不需要求梯度函数，因为 x 已经是叶节点了。从上例中后 4 行打印出的结果也能清楚地看到这一点。

2. 自动梯度计算

计算图搭建好以后，就可以用 backward 函数进行梯度计算了，用 grad 属性可以查看计算好的梯度，如：

```
# In[梯度计算和查看]
y.backward()          # 自动梯度计算
print(x.grad)         # 查看 y 对于 x 的梯度
print(b.grad)         # 查看 y 对于 b 的梯度
print(Q.grad)         # 查看 y 对于 Q 的梯度
```

运行结果为：

```
tensor([11., 11., 11., 11., 11.])
None
None
__main__:4: UserWarning: The .grad attribute of a Tensor that is not a leaf
Tensor is being accessed. Its .grad attribute won't be populated during
autograd.backward(). If you indeed want the gradient for a non-leaf Tensor,
use .retain grad() on the non-leaf Tensor. If you access the non-leaf Tensor
by mistake, make sure you access the leaf Tensor instead. See
github.com/pytorch/pytorch/pull/30531 for more information.
```

可以看出，只有 y 关于 x 的梯度输出了有效值，y 关于的 b 的梯度未输出是因为 b 的 requires_grad 属性为 False，不需要计算梯度，而 y 关于 Q 的梯度未输出是因为 Q 是中间变量，规定中间变量的梯度不能获取。

值得注意的是，PyTorch 中的计算图是动态图，在前向传播时构建，梯度计算完毕后释

放。因此，若在以上代码中再次执行 y.backward()命令，就会出现以下错误提醒：

```
RuntimeError: Trying to backward through the graph a second time (or directly
access saved variables after they have already been freed). Saved intermediate
values of the graph are freed when you call .backward() or autograd.grad().
Specify retain_graph=True if you need to backward through the graph a second
time or if you need to access saved variables after calling backward.
```

也就是说，在第一次执行 y.backward()命令后，计算图就被释放了，如果再次执行该命令，因为已经不存在计算图，所以就不能顺利计算梯度了。保存计算图的方法是在 backward 函数中传入 retain_graph=True 参数，即 y.backward(retain_graph=True)。也就是在执行 y.backward()命令后，计算图被保留下来。

PyTorch 这种在前向传播时构建计算图，梯度计算完成后释放计算图的范式叫做动态计算图，相较于 TensorFlow 的静态计算图而言。动态计算图的优点是灵活，可以随时改变计算图的结构，这在训练一些动态神经网络或有分叉的神经网络中很有用处，缺点是构建和释放计算图需要一些时间和计算资源。

另外，y.backward()函数求出的梯度是在原来的梯度上的累加值，如

```
y.backward(retain_graph=True)
print(x.grad)
```

运行结果为

```
tensor([22., 22., 22., 22., 22.])
```

可见 y 关于 x 的梯度变成了原来的 2 倍，这是因为之前已经求过一次梯度，第二次求出的梯度值要累加第一次求出的结果。为避免这种情况的出现，在每次求梯度之前需要先使用 zero_函数将梯度归零，如

```
x.grad.zero_()
print(x.grad)
y.backward(retain_graph=True)
print(x.grad)
```

运行结果为：

```
tensor([0., 0., 0., 0., 0.])
tensor([11., 11., 11., 11., 11.])
```

可见在使用了 zero_以后，y 关于 x 的梯度就归零了，再次计算梯度以后输出值恢复正常。

3. 关闭自动梯度计算

自动梯度计算通常用在模型训练中，但在前向传播过程中并不需要计算梯度，这时关闭自动梯度计算会让计算效率更高。有两种方式可以局部关闭自动梯度计算，一种是用 with torch.no_grad 块将本来需要计算梯度的代码包起来，如：

```
y = torch.matmul(b,x)
print(y.requires_grad)

with torch.no_grad():
    y1 = torch.matmul(b,x)
print(y1.requires_grad)
```

运行结果如下：

```
True
False
```

可以看出在 with torch.no_grad 块中构建的计算图是不能计算梯度的。

另一种方法是用张量的 detach 函数，如

```
y2 = y.detach()
print(y2.requires_grad)
```

运行结果为：

```
False
```

5.4.5 训练数据自由读取

为了更方便的管理和读取训练数据，PyTorch 提供了 torch.utils.data.Datasets 和 torch.utils.data.DataLoader 两个类。torch.utils.data.Datasets 是 torchvision 中所有预存训练数据的基类，保存着训练数据集的基本信息，比如数据和标签等。用户也可以用 Dataset 基类自定义训练数据。DataLoader 是一个用于加载训练数据的类，它可以对原始训练数据进行打乱顺序，划分小批量，和循环加载等操作。以下以著名的图像处理数据集 FashionMNIST 为例介绍数据读取和加载。

1. 数据下载

首先从云端下载 FashionMNIST 数据集，代码如下：

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt

# In[下载数据集]
training data = datasets.FashionMNIST(
       root='data',
       train=True,
       download=True,
       transform=ToTensor()
       )

test data = datasets.FashionMNIST(
       root='data',
       train=False,
       download=True,
       transform=ToTensor()
       )
```

关键字中 root 表示数据集存储的地址，程序运行后会在当前目标下生成一个名为 data 的文件，里面就是下载的 FashionMNIST 数据；train=True 表示下载训练集，train=False 表示下载测试集；download=True 表示如果本地没有该数据集则从网上下载；transform 接收用于将原

始数据转化成训练所需要的数据类型的函数，如 ToTensor 函数将 ndarray 数据转成转成浮点型张量，并标准化。

2. 查看数据

可以用 dir 函数查看 training_data 和 test_data 的相关属性和功能函数，常用的是 data 和 targets 属性，分别代表数据本身和对应的标签，所有的标签名可以用 classes 属性读取，如：

```
# In[查看数据结构]
print(training_data.data.shape,training_data.targets.shape)
print(test_data.data.shape,test_data.data.shape)
print(training_data.data.dtype,training_data.targets.dtype)
print(training_data.classes)
```

运行结果如下：

```
torch.Size([60000, 28, 28]) torch.Size([60000])
torch.Size([10000, 28, 28]) torch.Size([10000, 28, 28])
torch.uint8 torch.int64
['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt',
'Sneaker', 'Bag', 'Ankle boot']
```

可见 FashionMNIST 数据集一共包括 60000 条训练数据和 10000 条测试数据，每条训练数据是一个 dtype 为 torch.unit8 的 28×28 维的矩阵，数据标签一共有 10 类，用 torch.int64 数据表示。

3. 训练数据加载

用 torch.utils.data.DataLoader 加载训练数据的语法如下

dataloader = torch.utils.data.DataLoader(dataset, batch_size=1,shuffle=False)

其中：

- dataset：表示要加载的 torch.utils.data.Dataset 类的训练数据，比如上例中下载的 FashionMNIST 数据集；
- batch_size: 小批量数据的批量尺度；
- shuffle：将数据划分成批量时，是否要先打乱数据顺序，默认为不打乱。

将上例中已经下载好的 FashionMNIST 训练数据用 torch.util.data.DataLoader 加载的代码如下：

```
# In[训练数据加载]
batch_size = 32
training_data_loader = DataLoader(training_data,batch_size,shuffle=True)
test_data_loader = DataLoader(test_data,batch_size,shuffle=True)

for X,y in training_data_loader:
    print('Shape of X is ',X.shape)
    print('Shape of y is ',y.shape)
    break
```

运行结果如下：

```
Shape of X is  torch.Size([32, 1, 28, 28])
Shape of y is  torch.Size([32])
```

可见所有训练数据已经被分成了 32 条一份的小批量数据集，数据标签也做了相应的划分。

5.4.6 模型的搭建、训练和测试

本小节用一个完整的案例来介绍深度神经网络模型的搭建、训练和测试。以 torchvision 中已经预存的 FashionMNIST 训练数据为例。

1. 数据准备

数据准备包括下载和加载数据，代码如下：

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose
import matplotlib.pyplot as plt

# In[数据准备]
training_data = datasets.FashionMNIST(
       root='data',
       train=True,
       download=True,
       transform=ToTensor()
       )
test_data = datasets.FashionMNIST(
       root='data',
       train=False,
       download=True,
       transform=ToTensor()
       )
batch_size = 64
train_dataloader = DataLoader(training_data,batch_size=batch_size)
test_dataloader = DataLoader(test_data,batch_size=batch_size)
```

2. 构建模型

一个神经网络模型包括两个基本部分：拓扑结构和 forward 函数。拓扑结构是由线性层、激活函数、卷积层等基本模块搭建而成，封装在 nn.Sequential 类中，forward 函数用于前向传播计算过程。用户自定义神经网络模型一般继承 nn.Module 类，如：

```
# In[构建模型]
device = 'cuda' if torch.cuda.is_available() else 'cpu'

class NeuNet(nn.Module):
   def __init__(self):
       nn.Module.__init__(self)
       self.flatten = nn.Flatten()      # 将多维张量拉直成 1 维张量
                                        # 定义网络拓扑
       self.linear_relu_stack = nn.Sequential(
              nn.Linear(28*28,512),
              nn.ReLU(),
              nn.Linear(512,512),
              nn.ReLU(),
              nn.Linear(512,10),
```

```
            )
    def forward(self,x):                # 前向传播函数
        x=self.flatten(x)
        logits = self.linear relu stack(x)
        return logits

model =NeuNet().to(device)              # 创建一个神经网络实体
print(model)
```

运行结果如下：

```
NeuNet(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

从打印的结果可以清楚地看到该神经网络的拓扑结构。

3. 损失函数和优化器

因为 FashionMNIST 数据集是一个多分类问题，所以使用交叉熵损失函数和 SGD 优化器，代码如下：

```
# In[损失函数和优化器]
loss = nn.CrossEntropyLoss(reduction='mean')
opt = torch.optim.SGD(model.parameters(),lr=1e-3)
```

优化器定义时的两个必须参数分别表示需要优化的模型参数和学习率。

4. 训练函数

训练函数的主要任务是误差反向传播和调整参数，代码如下：

```
# In[训练函数]
def train(dataloader,model,loss fn,optimizer):
    size = len(dataloader.dataset)
    model.train()                       # 声明以下是训练环境
    for batch, (X,y) in enumerate(dataloader):
        X,y = X.to(device),y.to(device)
        pred = model(X)                 # 计算预测值
        loss = loss_fn(pred,y)          # 计算损失函数
        opt.zero_grad()                 # 梯度归零
        loss.backward()                 # 误差反向传播
        opt.step()                      # 调整参数

        if batch%100 == 0:              # 每隔 100 批次打印训练进度
            loss,current = loss.item(),batch*len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

代码中 model.train()是用于声明下面的代码是训练环境，与此对应，测试环境的声明方式是 model.eval()。这个声明在本例中是没有任何意义的，可加可不加，但若训练过程中使用了 Dropout 层就必须加了，因为在训练过程中 Dropout 层是要起作用的，但测试过程中却不能

启动 Dropout 层。

5. 测试函数

测试函数的主要任务是用测试数据测试训练出的模型的泛化性能，可以根据测试结果来修改模型或者训练过程。测试函数代码如下：

```
# In[测试函数]
def test(dataloader,model,loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()                                    # 声明模型评估状态
    test_loss,correct = 0,0
    with torch.no_grad():
        for X,y in dataloader:
            X,y = X.to(device),y.to(device)
            pred= model(X)                          # 计算预测值
            test_loss += loss_fn(pred,y).item()  # 计算误差
            correct += (pred.argmax(1)==y).type(torch.float).sum().item()
    correct /= size                                 # 预测分类正确率
    test_loss /= num_batches                        # 测试数据凭据误差
    print('Accuracy is {}, Average loss is {}'.format(correct,test_loss))
```

6. 模型训练和测试

训练和测试代码如下：

```
# In[训练和测试]
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, opt)
    test(test_dataloader, model, loss_fn)
print("Done!")
```

超参数 epochs 是指训练和测试的回合数，每一回合训练后都会进行测试。运行结果如下（为节约空间，仅输出第一回合训练过程结果和每回合测试结果）：

```
Epoch 1
-------------------------------
loss: 2.163912  [    0/60000]
loss: 2.159833  [ 6400/60000]
loss: 2.095972  [12800/60000]
loss: 2.120173  [19200/60000]
loss: 2.076414  [25600/60000]
loss: 2.007832  [32000/60000]
loss: 2.031443  [38400/60000]
loss: 1.956872  [44800/60000]
loss: 1.955904  [51200/60000]
loss: 1.886647  [57600/60000]
Accuracy is 0.6081, Average loss is 1.891194589578422
Epoch 2
-------------------------------
Accuracy is 0.6186, Average loss is 1.5148332749202753
Epoch 3
-------------------------------
Accuracy is 0.6345, Average loss is 1.2477369035125538
Epoch 4
```

```
------------------------------
Accuracy is 0.6474, Average loss is 1.0845678323393415
Epoch 5
------------------------------
Accuracy is 0.6607, Average loss is 0.9791825618713524
Done!
```

可以看出正确率在增加，而平均误差在减小。

5.4.7 模型的保存和重载

模型的保存和重载是模型应用的重要工具。一个训练好的模型需要保存起来才能继续使用；如果想获得模型训练的一些中间结果，也需要进行模型保存；另外迁移学习中要继续训练已经有过预训练的模型，也需要保存和重载原有模型。

1. 保存模型或模型参数

接着上一节中的代码，使用 torch.save 函数将训练好的模型和模型参数保存成.pth 文件，代码如下：

```
# In[保存模型或模型参数]
torch.save(model,'model')                                    # 保存整个模型
torch.save(model.state_dict(),'model_parameter.pth')    # 保存模型参数
```

这里 model.state_dict()是一个字典，保存着 model 模型的所有参数信息，可以通过键值访问神经网络每一层参数的具体值，比如：

```
model.state_dict()['linear_relu_stack.4.bias']
```

就可以访问第 4 层的偏置参数值，运行结果如下：

```
tensor([-0.0567,  0.0019, -0.0235,  0.0068, -0.0487,  0.1576, -0.0035,
0.0590,-0.0523, -0.0582])
```

2. 重载模型或模型参数

重载模型或模型参数是用 torch.load 函数，代码如下：

```
# In[重载模型和模型参数]
model1 = torch.load('model')  # 直接重载整个模型，包括网络拓扑和参数
model2 = NeuNet()             # 只重载参数需要先创建一个相同网络拓扑的初始模型
                              # 重载模型参数
model2.load_state_dict(torch.load('model_parameter.pth'))

with torch.no_grad():
    for X,y in train_dataloader:
        print(model(X)[0])
        print(model1(X)[0])
        print(model2(X)[0])
        break
```

若重载整个模型，则包括模型的网络拓扑重载和参数重载，不需要另外创建模型；若只重载模型参数，则需要预先创建一个具有相同网络拓扑的模型作为模型参数的载体。程序运行结果如下：

```
tensor([-2.4643, -4.8268, -0.6808, -3.1310, -1.1310,  2.9940, -1.2384,
2.7196,  3.1995,  4.8576])
tensor([-2.4643, -4.8268, -0.6808, -3.1310, -1.1310,  2.9940, -1.2384,
2.7196,3.1995,  4.8576])
tensor([-2.4643, -4.8268, -0.6808, -3.1310, -1.1310,  2.9940, -1.2384,
2.7196,3.1995,  4.8576])
```

可见，三个模型是完全一样的。

3. torchvision 中的模型

torchvision 库用已经预存了许多已经训练好的经典神经网络模型，比如 alexnet，densenet，resnet 等，用户可以在项目中直接使用这些模型，也可以基于这些模型进行迁移学习，如：

```
# In[torchvision 预训练模型加载]
import torchvision.models as models
model_vgg16 = models.vgg16(pretrained=True) # 创建一个已经训练好的 Vgg16 网络
                                            # 保存模型参数
torch.save(model_vgg16.state_dict(),'model_vgg16_parameter')
model_vgg16_1 = models.vgg16()              # 创建一个未训练的 vgg16 网络
                                            # 加载模型参数
model_vgg16_1.load_state_dict(torch.load('model_vgg16_parameter'))
model_vgg16.eval()            # 评估模型
model_vgg16_1.eval()          # 评估模型
```

5.5 深度学习案例

一个完整的深度学习过程主要包括构建网络、定义训练函数、定义测试函数、训练和测试、结果展示几个过程。本节给出两个基于 PyTorch 的深度学习案例，来展示基于 PyTorch 的深度学习全过程。

5.5.1 函数近似

本例用一个深度神经网络来近似一元二次函数，待近似的一元二次函数为

$$y = x^2 + 3x + 4 \tag{5-50}$$

使用全连接前馈式深度神经网络，各层节点数为输入层：1，第 1 隐藏：20，第 2 隐层：40，第 3 隐层 20，输出层：1；损失函数使用均方误差损失（MSE）；优化器使用随机梯度下降（SGD），全部代码如下：

【代码 5-1】深度神经网络逼近一元二次函数代码

```
# In[导入包]
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

# In[超参数]
LR = 1e-3
```

```
BATCH SIZE = 32
EPOCHS = 40

# In[原函数]
def fun(x):
    return x*x+3*x+4

x = np.linspace(-np.pi,np.pi,100)
y = fun(x)

# In[创建神经网络]
class NeuNet(nn.Module):
    def   init  (self,in size,out size):
        nn.Module.  init  (self)
        self.flatten = nn.Flatten()
        self.layers = nn.Sequential(
                    nn.Linear(in size,20),
                    nn.ReLU(),
                    nn.Linear(20,40),
                    nn.ReLU(),
                    nn.Linear(40,20),
                    nn.ReLU(),
                    nn.Linear(20,out_size),
                    )
    def forward(self,x):
        self.flatten(x)
        return self.layers(x)

model = NeuNet(1,1)

# In[损失函数和优化器]
loss = torch.nn.MSELoss()
opt = torch.optim.SGD(model.parameters(),lr=LR)

# In[训练函数]
def train(model,loss,opt):
    x_batch = -np.pi+2*np.pi*np.random.rand(BATCH_SIZE,1) # 训练输入
    y_tar_batch = fun(x_batch)                            # 目标输出

    x_batch = torch.from_numpy(x_batch).float()           # 数据格式转换
    y_tar_batch = torch.from_numpy(y_tar_batch).float()  # 数据格式转换
    y_pre_batch = model(x_batch).float()                  # 预测输入

    loss_fn = loss(y_tar_batch,y_pre_batch)               # 损失函数

    model.train()      # 声明训练
    opt.zero_grad()    # 梯度归零
    loss_fn.backward()# 误差反向传播
    opt.step()         # 参数调整

# In[测试函数]
def test(model):
    model.eval()
    with torch.no_grad():
        y_pre_test = model(torch.from_numpy(x).float().unsqueeze(dim=1))
        loss_value = loss(torch.from_numpy(y).float(),y_pre_test.float())
    print('loss_fn = ',loss_value)
```

```
    return loss value

# In[训练和测试]
Loss = []
for i in range(EPOCHS):
    print('EPOCH {}--------------'.format(i))
    train(model,loss,opt)
    loss value = test(model)
    Loss.append(loss value)
print('DONE')

# In[作图比较]
with torch.no grad():
    y test = model(torch.from numpy(x).float().unsqueeze(dim=1))
    y test = y test.squeeze().numpy()

plt.figure(1)
plt.plot(Loss)
plt.xlabel('EPOCHS')
plt.ylabel('Loss')
plt.title('Loss via EPOCHS')
plt.savefig('loss.jpg')

plt.figure(2)
plt.plot(x,y,label='real')
plt.plot(x,y_test,label='approximated')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Real vs approximated graph')
plt.legend()
plt.savefig('graph.jpg')
plt.show()
```

使用 print(model)命令可以查看构建的神经网络结构如下：

```
NeuNet(
  (flatten): Flatten(start dim=1, end dim=-1)
  (layers): Sequential(
    (0): Linear(in features=1, out features=20, bias=True)
    (1): ReLU()
    (2): Linear(in features=20, out features=40, bias=True)
    (3): ReLU()
    (4): Linear(in_features=40, out_features=20, bias=True)
    (5): ReLU()
    (6): Linear(in_features=20, out_features=1, bias=True)
  )
)
```

程序运行结果如图 5-16 所示：

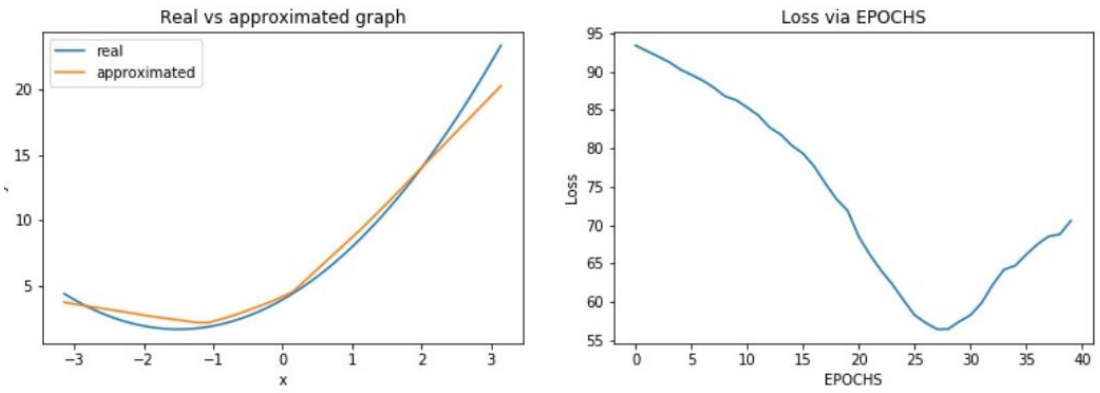

图 5-16 函数逼近运行结果

5.5.2 数字图片识别

本小节给出一个用深度神经网络识别 FashionMNIST 图片库的案例。深度神经网络共有 4 层，节点数分别为 28*28（这也是 FashionMNIST 图片的像素尺寸），512，512 和 10；因为是分类问题，损失函数使用交叉熵损失函数（CrossEntropyLoss），优化器使用随机梯度下降（SGD），全部代码如下：

【代码 5-2】深度神经网络数字图片识别代码

```
# In[导入包]
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt

# In[超参数]
BATCH_SIZE = 64
LR = 1e-3
EPOCHS = 5

# In[数据下载]
training_data = datasets.FashionMNIST(
       root="data",
       train=True,
       download=True,
       transform=ToTensor(),
       )

test_data = datasets.FashionMNIST(
       root="data",
       train=False,
       download=True,
       transform=ToTensor(),
       )
```

```
# In[数据加载]
train dataloader = DataLoader(training data,batch size=BATCH SIZE)
test dataloader = DataLoader(test data,batch size=BATCH SIZE)

# In[创建网络]
device = 'cuda' if torch.cuda.is available() else 'cpu'

class NeuralNetwork(nn.Module):
    def   init  (self):
        nn.Module.  init  (self)
        self.flatten = nn.Flatten()
        self.linear relu stack=nn.Sequential(
               nn.Linear(28*28,512),
               nn.ReLU(),
               nn.Linear(512,512),
               nn.ReLU(),
               nn.Linear(512,10)
               )

    def forward(self,x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)

# In[损失函数和优化器]
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),lr=LR)

# In[训练函数]
def train(dataloader,model,loss_fn,optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X,y) in enumerate(dataloader):
        X,y = X.to(device),y.to(device)

        pred = model(X)
        loss = loss_fn(pred,y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch%100 ==0:
            loss, current = loss.item(),batch*len(X)
            print(f"loss: {loss:>7f} [{current:>5d}/{size:>5d}]")

# In[测试函数]
def test(dataloader,model,loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss,correct = 0,0
    with torch.no_grad():
        for X,y in dataloader:
```

```
            X,y = X.to(device),y.to(device)
            pred = model(X)
            test loss += loss fn(pred,y).item()
            correct += (pred.argmax(1)==y).type(torch.float).sum().item()
    test loss /= num batches
    correct /= size
    print(f"Accuracy: {(100*correct):>0.1f}%, Avg loss: {test loss:>8f}")

# In[模型训练和测试]
for t in range(EPOCHS):
    print(f"Epoch {t+1}\n--------------------")
    train(train dataloader,model,loss fn,optimizer)
    test(test dataloader,model,loss fn)
print("Done!")

# In[训练结果展示]
classes = training data.classes
model.eval()
x,y = test_data[0][0],test_data[0][1]
with torch.no_grad():
    pred = model(x)
    predicted,actual = classes[pred[0].argmax(0)],classes[y]
print(f'Predicted: "{predicted}", Actual: "{actual}"')
```

使用 print(model)函数可以得到神经网络模型的拓扑结构如下：

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

程序运行结果如下：

```
------Epoch 1------
loss: 2.308854 [    0/60000]
loss: 2.285936 [ 6400/60000]
loss: 2.274783 [12800/60000]
loss: 2.276982 [19200/60000]
loss: 2.243695 [25600/60000]
loss: 2.230343 [32000/60000]
loss: 2.230508 [38400/60000]
loss: 2.203054 [44800/60000]
loss: 2.198021 [51200/60000]
loss: 2.176432 [57600/60000]
Accuracy: 46.0%, Avg loss: 2.161742
------Epoch 2------
Accuracy: 58.9%, Avg loss: 1.907152
------Epoch 3------
Accuracy: 61.6%, Avg loss: 1.540300
------Epoch 4------
Accuracy: 63.3%, Avg loss: 1.265945
------Epoch 5------
```

```
Accuracy: 64.5%, Avg loss: 1.097099
Done!
Predicted: "Ankle boot", Actual: "Ankle boot"
```