Artificial Intelligence Experimental Manual

人工智能实验课程手册（下册）

作者：Yunlong Yu

组织：浙江大学信电学院

时间：July 10, 2023

浙江大学

目录

第 1 章 神经网络

内容提要

前面介绍了不同的线性模型。尽管神经网络涵盖了更多更为丰富的模型，我们依然可以用描述神经网络的方式来描述线性模型，从而把线性模型看作一个神经网络。首先，我们用“层”符号来重写这个模型。

神经网络图

深度学习从业者喜欢绘制图表来可视化模型中正在发生的事情。在图1.1中，我们将线性回归模型描述为一个神经网络。需要注意的是，该图只显示连接模式，即只显示每个输入如何连接到输出，隐去了权重和偏置的值。

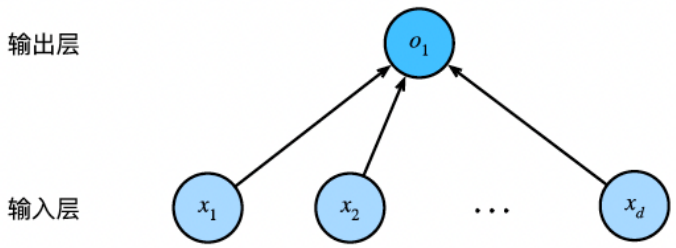


图 1.1: 线性回归是一个单层神经网络。

在图1.1所示的神经网络中，输入为$x_1, ..., x_d$，因此输入层中的输入数（或称为特征维度，feature dimensionality）为 d。网络的输出为 o_1，因此输出层中的输出数是 1。需要注意的是，输入值都是已经给定的，并且只有一个计算神经元。由于模型重点在发生计算的地方，所以通常我们在计算层数时不考虑输入层。也就是说，图1.1中神经网络的层数为 1。我们可以将线性回归模型视为仅由单个人工神经元组成的神经网络，或称为单层神经网络。对于线性回归，每个输入都与每个输出（在本例中只有一个输出）相连，我们将这种变换（图1.1中的输出层）称为全连接层（fully-connected layer）或称为稠密层（dense layer）。

1.1 利用 Pytorch 实现线性回归

```
%matplotlib inline
import random
import torch
from d2l import torch as d2l
```

1.1.1 生成数据集

为了简单起见，我们将根据带有噪声的线性模型构造一个人造数据集。我们的任务是使用这个有限样本的数据集来恢复这个模型的参数。我们将使用低维数据，这样可以很容易地将其可视化。在下面的代码中，我们生成一个包含 1000 个样本的数据集，每个样本包含从标准正态分布中采样的 2 个特征。我们的合成数据集是一个矩阵 $X \in \mathbb{R}^{1000\times 2}$。

我们使用线性模型参数 $w = [2; -3.4]^T$、$b = 4.2$ 和噪声项 ϵ 生成数据集及其标签：

$$y = Xw + b + \epsilon \tag{1.1}$$

ϵ 可以视为模型预测和标签时的潜在观测误差。在这里我们认为标准假设成立，即 ϵ 服从均值为 0 的正态分布。为了简化问题，我们将标准差设为 0.01。下面的代码生成合成数据集。

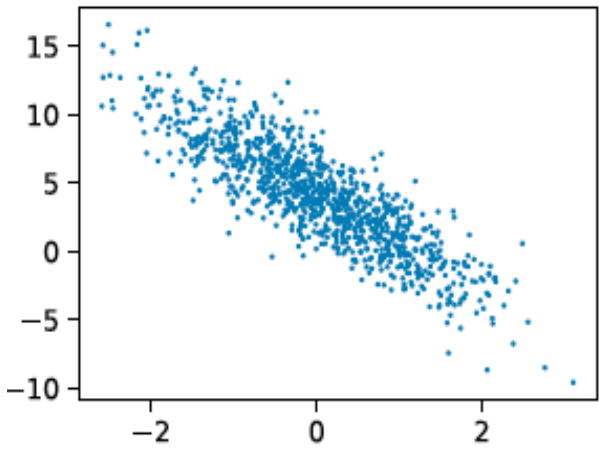
```
def synthetic_data(w, b, num_examples): #@save
    """生成y=Xw+b+噪声"""
    X = torch.normal(0, 1, (num_examples, len(w)))
    y = torch.matmul(X, w) + b
    y += torch.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))

true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
print('features:', features[0],'\nlabel:', labels[0])
```

注意，features 中的每一行都包含一个二维数据样本，labels 中的每一行都包含一维标签值（一个标量）。

```
features: tensor([-0.3679, -1.8471])
label: tensor([9.7361])
```

通过生成第二个特征 features[:, 1] 和 labels 的散点图，可以直观观察到两者之间的线性关系。

```
d2l.set_figsize()
d2l.plt.scatter(features[:, 1].detach().numpy(), labels.detach().numpy(), 1);
```



1.1.2 读取数据集

回想一下，训练模型时要对数据集进行遍历，每次抽取一小批量样本，并使用它们来更新我们的模型。由于这个过程是训练机器学习算法的基础，所以有必要定义一个函数，该函数能打乱数据集中的样本并以小批量方式获取数据。

在下面的代码中，我们定义一个 data_iter 函数，该函数接收批量大小、特征矩阵和标签向量作为输入，生成大小为 batch_size 的小批量。每个小批量包含一组特征和标签。

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # 这些样本是随机读取的，没有特定的顺序
```

```
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = torch.tensor(
            indices[i: min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

通常，我们利用 GPU 并行运算的优势，处理合理大小的“小批量”。每个样本都可以并行地进行模型计算，且每个样本损失函数的梯度也可以被并行计算。GPU 可以在处理几百个样本时，所花费的时间不比处理一个样本时多太多。

我们直观感受一下小批量运算：读取第一个小批量数据样本并打印。每个批量的特征维度显示批量大小和输入特征数。同样的，批量的标签形状与 batch_size 相等。

```
batch_size = 10
for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break

tensor([[ 0.1649, -1.1651],
    [-2.0755, -1.0165],
    [-0.2189, 0.7607],
    [ 0.6833, 0.3537],
    [-0.2736, -2.0485],
    [-0.3026, 0.9771],
    [ 2.4795, 0.6881],
    [-0.2045, -0.8509],
    [-0.1353, 0.5476],
    [ 0.3371, -0.0479]])
tensor([[ 8.4901],
    [ 3.5015],
    [ 1.1779],
    [ 4.3752],
    [10.6125],
    [ 0.2845],
    [ 6.8094],
    [ 6.6776],
    [ 2.0598],
    [ 5.0189]])
```

当我们运行迭代时，我们会连续地获得不同的小批量，直至遍历完整个数据集。上面实现的迭代对教学来说很好，但它的执行效率很低，可能会在实际问题上陷入麻烦。例如，它要求我们将所有数据加载到内存中，并执行大量的随机内存访问。在深度学习框架中实现的内置迭代器效率要高得多，它可以处理存储在文件中的数据和数据流提供的数据。

1.1.3 初始化模型参数

在我们开始用小批量随机梯度下降优化我们的模型参数之前，我们需要先有一些参数。在下面的代码中，我们通过从均值为 0、标准差为 0.01 的正态分布中采样随机数来初始化权重，并将偏置初始化为 0。

```
w = torch.normal(0, 0.01, size=(2,1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```

在初始化参数之后，我们的任务是更新这些参数，直到这些参数足够拟合我们的数据。每次更新都需要计算损失函数关于模型参数的梯度。有了这个梯度，我们就可以向减小损失的方向更新每个参数。

1.1.4 定义模型

接下来，我们必须定义模型，将模型的输入和参数同模型的输出关联起来。回想一下，要计算线性模型的输出，我们只需计算输入特征 X 和模型权重 w 的矩阵-向量乘法后加上偏置 b。注意，上面的 Xw 是一个向量，而 b 是一个标量。

```
def linreg(X, w, b): #@save
    """线性回归模型"""
    return torch.matmul(X, w) + b
```

1.1.5 定义损失函数

因为需要计算损失函数的梯度，所以我们应该先定义损失函数。线性回归利用平方损失函数。

```
def squared_loss(y_hat, y): #@save
    """均方损失"""
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

1.1.6 定义优化算法

小批量随机梯度下降法

在每一步中，使用从数据集中随机抽取的一个小批量，然后根据参数计算损失的梯度。接下来，朝着减少损失的方向更新我们的参数。下面的函数实现小批量随机梯度下降更新。该函数接受模型参数集合、学习速率和批量大小作为输入。每一步更新的大小由学习速率 lr 决定。因为我们计算的损失是一个批量样本的总和，所以我们用批量大小（batch_size）来规范化步长，这样步长大小就不会取决于我们对批量大小的选择。

```
def sgd(params, lr, batch_size): #@save
    """小批量随机梯度下降"""
    with torch.no_grad():
        for param in params:
            param -= lr * param.grad / batch_size
            param.grad.zero_()
```

1.1.7 训练

现在我们已经准备好了模型训练所有需要的要素，可以实现主要的训练过程部分了。理解这段代码至关重要，因为从事深度学习后，相同的训练过程几乎一遍又一遍地出现。在每次迭代中，我们读取一小批量训练样本，并通过我们的模型来获得一组预测。计算完损失后，我们开始反向传播，存储每个参数的梯度。最后，我们调用优化算法 sgd 来更新模型参数。概括一下，我们将执行以下循环：

- 初始化参数
- 重复以下训练，直到完成
  - 计算梯度 $g \leftarrow \partial_{(w,b)} \frac{1}{|B|} \sum_{i \in B} l(x^i, y^i, w, b)$
  - $(w, b) \leftarrow (w, b) - \alpha g$

在每个迭代周期（epoch）中，我们使用 data_iter 函数遍历整个数据集，并将训练数据集中所有样本都使用一次（假设样本数能够被批量大小整除）。这里的迭代周期个数 num_epochs 和学习率 lr 都是超参数，分别设为 3 和 0.03。设置超参数很棘手，需要通过反复试验进行调整

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss
```

```
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        l = loss(net(X, w, b), y) # X和y的小批量损失
        # 因为l形状是(batch_size,1)，而不是一个标量。l中的所有元素被加到一起，
        # 并以此计算关于[w,b]的梯度
        l.sum().backward()
        sgd([w, b], lr, batch_size) # 使用参数的梯度更新参数
    with torch.no_grad():
        train_l = loss(net(features, w, b), labels)
        print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

```
epoch 1, loss 0.043705
epoch 2, loss 0.000172
epoch 3, loss 0.000047
```

```
print(f'w的估计误差: {true_w - w.reshape(true_w.shape)}')
print(f'b的估计误差: {true_b - b}')
```

```
w的估计误差: tensor([ 0.0003, -0.0002], grad_fn=<SubBackward0>)
b的估计误差: tensor([0.0010], grad_fn=<RsubBackward1>)
```

注意，我们不应该想当然地认为我们能够完美地求解参数。在机器学习中，我们通常不太关心恢复真正的参数，而更关心如何高度准确预测参数。幸运的是，即使是在复杂的优化问题上，随机梯度下降通常也能找到非常好的解。其中一个原因是，在深度网络中存在许多参数组合能够实现高度精确的预测。

1.2 线性回归的简洁实现

本节将介绍如何通过使用深度学习框架来简洁地实现线性回归模型。

1.2.1 生成数据集

```
import numpy as np
import torch
from torch.utils import data
from d2l import torch as d2l
```

```
true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

1.2.2 读取数据集

我们可以调用框架中现有的 API 来读取数据。我们将 features 和 labels 作为 API 的参数传递，并通过数据迭代器指定 batch_size。此外，布尔值 is_train 表示是否希望数据迭代器对象在每个迭代周期内打乱数据。

```
def load_array(data_arrays, batch_size, is_train=True): #@save
    """构造一个PyTorch数据迭代器"""
    dataset = data.TensorDataset(*data_arrays)
    return data.DataLoader(dataset, batch_size, shuffle=is_train)
```

```
batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

使用 data_iter 的方式与上节中使用 data_iter 函数的方式相同。为了验证是否正常工作，让我们读取并打印第一个小批量样本。这里我们使用 iter 构造 Python 迭代器，并使用 next 从迭代器中获取第一项。

```
next(iter(data_iter))

[tensor([[ 0.1554, -0.2034],
[-0.2140, 1.0352],
[-0.4209, 0.0428],
[ 0.1887, 0.6141],
[ 0.4987, -0.2314],
[ 0.0653, 1.6406],
[-1.1881, 0.2900],
[-0.2824, 0.5910],
[ 0.9963, -0.1816],
[-1.6830, -1.3963]]),
tensor([[ 5.2116],
[ 0.2479],
[ 3.2188],
[ 2.4845],
[ 5.9884],
[-1.2453],
[ 0.8441],
[ 1.6217],
[ 6.8072],
[ 5.5692]])]
```

1.2.3 定义模型

对于标准深度学习模型，我们可以使用框架的预定义好的层。这使我们只需关注使用哪些层来构造模型，而不必关注层的实现细节。我们首先定义一个模型变量 net，它是一个 Sequential 类的实例。Sequential 类将多个层串联在一起。当给定输入数据时，Sequential 实例将数据传入到第一层，然后将第一层的输出作为第二层的输入，以此类推。在下面的例子中，我们的模型只包含一个层，因此实际上不需要 Sequential。但是由于以后几乎所有的模型都是多层的，在这里使用 Sequential 会让你熟悉“标准的流水线”。

回顾图1.1中的单层网络架构，这一单层被称为全连接层（fully-connected layer），因为它的每一个输入都通过矩阵-向量乘法得到它的每个输出。

在 PyTorch 中，全连接层在 Linear 类中定义。值得注意的是，我们将两个参数传递到 nn.Linear 中。第一个指定输入特征形状，即 2，第二个指定输出特征形状，输出特征形状为单个标量，因此为 1。

```
# nn是神经网络的缩写
from torch import nn
net = nn.Sequential(nn.Linear(2, 1))
```

1.2.4 初始化模型参数

在使用 net 之前，我们需要初始化模型参数。如在线性回归模型中的权重和偏置。深度学习框架通常有预定义的方法来初始化参数。在这里，我们指定每个权重参数应该从均值为 0、标准差为 0.01 的正态分布中随机采样，偏置参数将初始化为零。

正如我们在构造 nn.Linear 时指定输入和输出尺寸一样，现在我们能直接访问参数以设定它们的初始值。我们通过 net[0] 选择网络中的第一个图层，然后使用 weight.data 和 bias.data 方法访问参数。我们还可以使用替换方法 normal_ 和 fill_ 来重写参数值。

```
net[0].weight.data.normal_(0, 0.01)
net[0].bias.data.fill_(0)
```

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

1.2.5 定义损失函数

计算均方误差使用的是 MSELoss 类，也称为平方 L_2 范数。默认情况下，它返回所有样本损失的平均值。

```
loss = nn.MSELoss()
```

1.2.6 定义优化算法

小批量随机梯度下降算法是一种优化神经网络的标准工具，PyTorch 在 optim 模块中实现了该算法的许多变种。当我们实例化一个 SGD 实例时，我们要指定优化的参数（可通过 net.parameters() 从我们的模型中获得）以及优化算法所需的超参数字典。小批量随机梯度下降只需要设置 lr 值，这里设置为 0.03。

```
trainer = torch.optim.SGD(net.parameters(), lr=0.03)
```

1.2.7 训练

通过深度学习框架的高级 API 来实现我们的模型只需要相对较少的代码。我们不必单独分配参数、不必定义我们的损失函数，也不必手动实现小批量随机梯度下降。当我们需要更复杂的模型时，高级 API 的优势将大大增加。当我们有了所有的基本组件，训练过程代码与我们从零开始实现时所做的非常相似。

回顾一下：在每个迭代周期里，我们将完整遍历一次数据集（train_data），不停地从中获取一个小批量的输入和相应的标签。对于每一个小批量，我们会进行以下步骤:

- 通过调用 net(X) 生成预测并计算损失 l（前向传播）。
- 通过进行反向传播来计算梯度。
- 通过调用优化器来更新模型参数。

为了更好的衡量训练效果，我们计算每个迭代周期后的损失，并打印它来监控训练过程。

```
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
```

```
        l = loss(net(X) ,y)
        trainer.zero_grad()
        l.backward()
        trainer.step()
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l:f}')
```

```
epoch 1, loss 0.000183
epoch 2, loss 0.000101
epoch 3, loss 0.000101
```

下面我们比较生成数据集的真实参数和通过有限数据训练获得的模型参数。要访问参数，我们首先从 net 访问所需的层，然后读取该层的权重和偏置。正如在从零开始实现中一样，我们估计得到的参数与生成数据的真实参数非常接近。

```
w = net[0].weight.data
print('w的估计误差：', true_w - w.reshape(true_w.shape))
b = net[0].bias.data
print('b的估计误差：', true_b - b)
```

```
w的估计误差： tensor([-0.0003, -0.0002])
b的估计误差： tensor([8.1062e-06])
```

1.3 softmax 回归

1.3.1 分类架构

我们从一个图像分类问题开始。假设每次输入是一个 2×2 的灰度图像。我们可以用一个标量表示每个像素值，每个图像对应四个特征 $x_1; x_2; x_3; x_4$。此外，假设每个图像属于类别“猫”“鸡”和“狗”中的一个。

接下来，我们要选择如何表示标签。我们有两个明显的选择：最直接的想法是选择 $y \in \{1, 2, 3\}$，其中整数分别代表 f 狗; 猫; 鸡 g。这是在计算机上存储此类信息的有效方法。如果类别间有一些自然顺序，比如说我们试图预测 {婴儿; 儿童; 青少年; 青年人; 中年人; 老年人}，那么将这个问题转变为回归问题，并且保留这种格式是有意义的。

但是一般的分类问题并不与类别之间的自然顺序有关。幸运的是，统计学家很早以前就发明了一种表示分类数据的简单方法：独热编码（one-hot encoding）。独热编码是一个向量，它的分量和类别一样多。类别对应的分量设置为 1，其他所有分量设置为 0。在我们的例子中，标签 y 将是一个三维向量，其中 (1; 0; 0) 对应于“猫”、(0; 1; 0) 对应于“鸡”、(0; 0; 1) 对应于“狗”：

$$y \in \{(1;0;0); (0;1;0); (0;0;1)\}. \tag{1.2}$$

1.3.2 网络架构

为了估计所有可能类别的条件概率，我们需要一个有多个输出的模型，每个类别对应一个输出。为了解决线性模型的分类问题，我们需要和输出一样多的仿射函数（affine function）。每个输出对应于它自己的仿射函数。在我们的例子中，由于我们有 4 个特征和 3 个可能的输出类别，我们将需要 12 个标量来表示权重（带下标的 w），3 个标量来表示偏置（带下标的 b）。下面我们为每个输入计算三个未规范化的预测（logit）：o_1、o_2 和 o_3。

$$o_1 = x_1w_{11} + x_2w_{12} + x_3w_{13} + x_4w_{14} + b_1; \tag{1.3}$$

$$o_2 = x_1w_{21} + x_2w_{22} + x_3w_{23} + x_4w_{24} + b_2; \tag{1.4}$$

$$o_3 = x_1w_{31} + x_2w_{32} + x_3w_{33} + x_4w_{34} + b_3 : \tag{1.5}$$

我们可以用神经网络图图1.2来描述这个计算过程。与线性回归一样，softmax 回归也是一个单层神经网络。由于计算每个输出 $o1$、$o2$ 和 $o3$ 取决于所有输入 $x1$、$x2$、$x3$ 和 $x4$，所以 softmax 回归的输出层也是全连接层。为了更简洁地表达模型，我们仍然使用线性代数符号。通过向量形式表达为 $o = Wx + b$，这是一种更适合数学

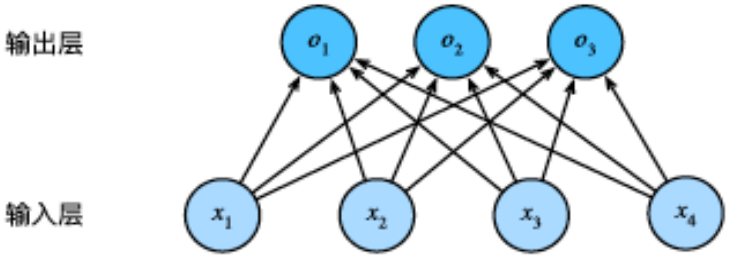


图 1.2: softmax 回归是一种单层神经网络。

和编写代码的形式。由此，我们已经将所有权重放到一个 3×4 矩阵中。对于给定数据样本的特征 x，我们的输出是由权重与输入特征进行矩阵-向量乘法再加上偏置 b 得到的。

1.3.3 softmax 运算

现在我们将优化参数以最大化观测数据的概率。为了得到预测结果，我们将设置一个阈值，如选择具有最大概率的标签。我们希望模型的输出 $\hat{y}_j$ 可以视为属于类 j 的概率，然后选择具有最大输出值的类别 $argmax_j y_j$ 作为我们的预测。例如，如果 $\hat{y}_1$、$\hat{y}_2$ 和 $\hat{y}_3$ 分别为 0.1、0.8 和 0.1，那么我们预测的类别是 2，在我们的例子中代表“鸡”。

公式如下：

$$\hat{\mathbf{y}} = (\mathbf{o}), \quad \hat{y}_j = \frac{exp(o_j)}{\sum_k exp(o_k)} \tag{1.6}$$

这里，对于所有的 j 总有 $0 \leq \hat{y}_j \leq 1$。因此，$\mathbf{y}$ 可以视为一个正确的概率分布。softmax 运算不会改变未规范化的预测 $\mathbf{o}$ 之间的大小次序，只会确定分配给每个类别的概率。因此，在预测过程中，我们仍然可以用下式来选择最有可能的类别。

尽管 softmax 是一个非线性函数，但 softmax 回归的输出仍然由输入特征的仿射变换决定。因此，softmax 回归是一个线性模型（linear model）。

1.4 图像分类数据集

MNIST 数据集是图像分类中广泛使用的数据集之一，但作为基准数据集过于简单。我们将使用类似但更复杂的 Fashion-MNIST 数据集

```
%matplotlib inline
import torch
import torchvision
from torch.utils import data
from torchvision import transforms
from d2l import torch as d2l
```

```
d2l.use_svg_display()
```

1.4.1 读取数据集

我们可以通过框架中的内置函数将 Fashion-MNIST 数据集下载并读取到内存中。

```
# 通过ToTensor实例将图像数据从PIL类型变换成32位浮点数格式，
# 并除以255使得所有像素的数值均在0🄵1之间
trans = transforms.ToTensor()
mnist_train = torchvision.datasets.FashionMNIST(
root="../data", train=True, transform=trans, download=True)
mnist_test = torchvision.datasets.FashionMNIST(
root="../data", train=False, transform=trans, download=True)
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to
    ../
,!data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100.0%
Extracting ../data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ../data/FashionMNIST/raw
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to
    ../
,!data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100.0%
Extracting ../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
    ../
,!data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100.0%
Extracting ../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST/raw
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
    ../
,!data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100.0%Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw
```

Fashion-MNIST 由 10 个类别的图像组成，每个类别由训练数据集（train dataset）中的 6000 张图像和测试数据集（test dataset）中的 1000 张图像组成。因此，训练集和测试集分别包含 60000 和 10000 张图像。测试数据集不会用于训练，只用于评估模型性能。

```
len(mnist_train), len(mnist_test)
(60000, 10000)
```

每个输入图像的高度和宽度均为 28 像素。数据集由灰度图像组成，其通道数为 1。为了简洁起见，本书将高度 h 像素、宽度 w 像素图像的形状记为 $h \times w$ 或（h,w）。

```
mnist_train[0][0].shape
torch.Size([1, 28, 28])
```

Fashion-MNIST 中包含的 10 个类别，分别为 t-shirt（T 恤）、trouser（裤子）、pullover（套衫）、dress（连衣裙）、coat（外套）、sandal（凉鞋）、shirt（衬衫）、sneaker（运动鞋）、bag（包）和 ankle boot（短靴）。以下函数用于在数字标签索引及其文本名称之间进行转换。

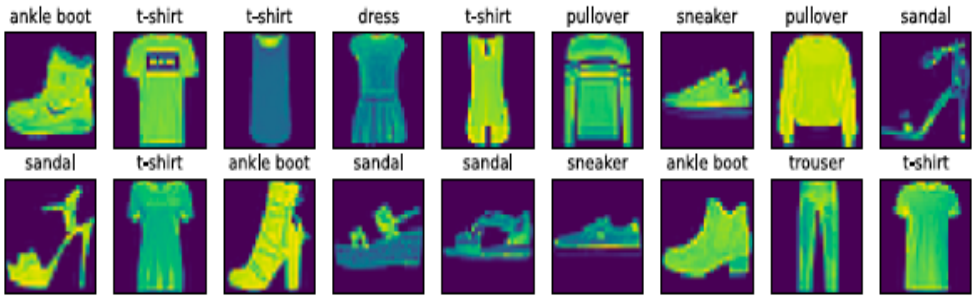
```
def get_fashion_mnist_labels(labels): #@save
"""返回Fashion-MNIST数据集的文本标签"""
text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
return [text_labels[int(i)] for i in labels]
```

我们现在可以创建一个函数来可视化这些样本。

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5): #@save
"""绘制图像列表"""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        if torch.is_tensor(img):
        # 图片张量
            ax.imshow(img.numpy())
        else:
            # PIL图片
            ax.imshow(img)
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

以下是训练数据集中前几个样本的图像及其相应的标签。

```
X, y = next(iter(data.DataLoader(mnist_train, batch_size=18)))
show_images(X.reshape(18, 28, 28), 2, 9, titles=get_fashion_mnist_labels(y));
```



1.4.2 读取小批量

为了使我们在读取训练集和测试集时更容易，我们使用内置的数据迭代器，而不是从零开始创建。回顾一下，在每次迭代中，数据加载器每次都会读取一小批量数据，大小为 batch_size。通过内置数据迭代器，我们可以随机打乱了所有样本，从而无偏见地读取小批量。

```
batch_size = 256
def get_dataloader_workers(): #@save
    """使用4个进程来读取数据"""
    return 4
train_iter = data.DataLoader(mnist_train, batch_size, shuffle=True,
num_workers=get_dataloader_workers())
```

我们看一下读取训练数据所需的时间。

```
timer = d2l.Timer()
for X, y in train_iter:
continue
f'{timer.stop():.2f} sec'
```

1.4.3 整合所有组件

现在我们定义 load_data_fashion_mnist 函数，用于获取和读取 Fashion-MNIST 数据集。这个函数返回训练集和验证集的数据迭代器。此外，这个函数还接受一个可选参数 resize，用来将图像大小调整为另一种形状。

```
def load_data_fashion_mnist(batch_size, resize=None): #@save
"""下载Fashion-MNIST数据集，然后将其加载到内存中"""
    trans = [transforms.ToTensor()]
    if resize:
        trans.insert(0, transforms.Resize(resize))
    trans = transforms.Compose(trans)
    mnist_train = torchvision.datasets.FashionMNIST(
        root="../data", train=True, transform=trans, download=True)
    mnist_test = torchvision.datasets.FashionMNIST(
        root="../data", train=False, transform=trans, download=True)
    return (data.DataLoader(mnist_train, batch_size, shuffle=True, num_workers=get_dataloader_workers
        ()),data.DataLoader(mnist_test, batch_size, shuffle=False, num_workers=get_dataloader_workers
        ()))
```

下面，我们通过指定 resize 参数来测试 load_data_fashion_mnist 函数的图像大小调整功能。

```
train_iter, test_iter = load_data_fashion_mnist(32, resize=64)
for X, y in train_iter:
    print(X.shape, X.dtype, y.shape, y.dtype)
    break
```

1.5 softmax 回归的从零开始实现

就像我们从零开始实现线性回归一样，我们认为 softmax 回归也是重要的基础，因此应该知道实现 softmax 回归的细节。

```
import torch
from IPython import display
from d2l import torch as d2l
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

1.5.1 初始化模型参数

和之前线性回归的例子一样，这里的每个样本都将用固定长度的向量表示。原始数据集中的每个样本都是28×28 的图像。本节将展平每个图像，把它们看作🄴度为 784 的向量。在后面的章节中，我们将讨论能够利用图像空间结构的特征，但现在我们暂时只把每个像素位置看作一个特征。回想一下，在 softmax 回归中，我们的输出与类别一样多。因为我们的数据集有 10 个类别，所以网络输出维度为 10。因此，权重将构成一个 784×10 的矩阵，偏置将构成一个 1×10 的行向量。与线性回归一样，我们将使用正态分布初始化我们的权重 W，偏置初始化为 0。

```
num_inputs = 784
num_outputs = 10
W = torch.normal(0, 0.01, size=(num_inputs, num_outputs), requires_grad=True)
b = torch.zeros(num_outputs, requires_grad=True)
```

1.5.2 定义 softmax 操作

在实现 softmax 回归模型之前，我们简要回顾一下 sum 运算符如何沿着张量中的特定维度工作，给定一个矩阵 X，我们可以对所有元素求和（默认情况下）。也可以只求同一个轴上的元素，即同一列（轴 0）或同一行（轴 1）。如果 X 是一个形状为 (2, 3) 的张量，我们对列进行求和，则结果将是一个具有形状 (3,) 的向量。当调用 sum 运算符时，我们可以指定保持在原始张量的轴数，而不折叠求和的维度。这将产生一个具有形状 (1, 3) 的二维张量。

```
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdim=True), X.sum(1, keepdim=True)
```

```
(tensor([[5., 7., 9.]]),
tensor([[ 6.],
[15.]]))
```

回想一下，实现 softmax 由三个步骤组成：

1. 对每个项求幂（使用 exp）；
2. 对每一行求和（小批量中每个样本是一行），得到每个样本的规范化常数；
3. 将每一行除以其规范化常数，确保结果的和为 1。

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdim=True)
    return X_exp / partition # 这里应用了广播机制
```

正如上述代码，对于任何随机输入，我们将每个元素变成一个非负数。此外，依据概率原理，每行总和为 1。

```
X = torch.normal(0, 1, (2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(tensor([[0.2968, 0.4115, 0.0945, 0.1603, 0.0368],
[0.2128, 0.5422, 0.0865, 0.1104, 0.0481]]),
tensor([1.0000, 1.0000]))
```

1.5.3 定义模型

定义 softmax 操作后，我们可以实现 softmax 回归模型。下面的代码定义了输入如何通过网络映射到输出。注意，将数据传递到模型之前，我们使用 reshape 函数将每张原始图像展平为向量。

```
def net(X):
    return softmax(torch.matmul(X.reshape((-1, W.shape[0])), W) + b)
```

1.5.4 定义损失函数

接下来，我们实现交叉熵损失函数。这可能是深度学习中最常见的损失函数，因为目前分类问题的数量远远超过回归问题的数量。回顾一下，交叉熵采用真实标签的预测概率的负对数似然。这里我们不使用 Python 的 for 循环迭代预测（这往往是低效的），而是通过一个运算符选择所有元素。下面，我们创建一个数据样本 y_hat，其中包含 2 个样本在 3 个类别的预测概率，以及它们对应的标签 y。有了 y，我们知道在第一个样本中，第一类是正确的预测；而在第二个样本中，第三类是正确的预测。然后使用 y 作为 y_hat 中概率的索引，我们选择第一个样本中第一个类的概率和第二个样本中第三个类的概率。

```
y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

现在我们只需一行代码就可以实现交叉熵损失函数。

```
def cross_entropy(y_hat, y):
    return - torch.log(y_hat[range(len(y_hat)), y])
cross_entropy(y_hat, y)
```

```
tensor([2.3026, 0.6931])
```

1.5.5 分类精度

给定预测概率分布 y_hat，当我们必须输出硬预测（hard prediction）时，我们通常选择预测概率最高的类。许多应用都要求我们做出选择。如 Gmail 必须将电子邮件分类为“Primary（主要邮件）”、“Social（社交邮件）”“Updates（更新邮件）”或“Forums（论坛邮件）”。Gmail 做分类时可能在内部估计概率，但最终它必须在类中选择一个。

当预测与标签分类 y 一致时，即是正确的。分类精度即正确预测数量与总预测数量之比。虽然直接优化精度可能很困难（因为精度的计算不可导），但精度通常是我们最关心的性能衡量标准，我们在训练分类器时几乎总会关注它。

为了计算精度，我们执行以下操作。首先，如果 y_hat 是矩阵，那么假定第二个维度存储每个类的预测分数。我们使用 argmax 获得每行中最大元素的索引来获得预测类别。然后我们将预测类别与真实 y 元素进行比较。由于等式运算符“==”对数据类型很敏感，因此我们将 y_hat 的数据类型转换为与 y 的数据类型一致。结果是一个包含 0（错）和 1（对）的张量。最后，我们求和会得到正确预测的数量。

```
def accuracy(y_hat, y): #@save
    """计算预测正确的数量"""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
    y_hat = y_hat.argmax(axis=1)
    cmp = y_hat.type(y.dtype) == y
    return float(cmp.type(y.dtype).sum())
```

我们将继续使用之前定义的变量 y_hat 和 y 分别作为预测的概率分布和标签。可以看到，第一个样本的预测类别是 2（该行的最大元素为 0.6，索引为 2），这与实际标签 0 不一致。第二个样本的预测类别是 2（该行的最大元素为 0.5，索引为 2），这与实际标签 2 一致。因此，这两个样本的分类精度率为 0.5。

```
accuracy(y_hat, y) / len(y)
```

同样，对于任意数据迭代器 data_iter 可访问的数据集，我们可以评估在任意模型 net 的精度。

```
def evaluate_accuracy(net, data_iter): #@save
    """计算在指定数据集上模型的精度"""
    if isinstance(net, torch.nn.Module):
        net.eval() # 将模型设置为评估模式
    metric = Accumulator(2) # 正确预测数、预测总数
    with torch.no_grad():
        for X, y in data_iter:
            metric.add(accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]
```

这里定义一个实用程序类 Accumulator，用于对多个变量进行累加。在上面的 evaluate_accuracy 函数中，我们在 Accumulator 实例中创建了 2 个变量，分别用于存储正确预测的数量和预测的总数量。当我们遍历数据集时，两者都将随着时间的推移而累加。

```
class Accumulator: #@save
    """在n个变量上累加"""
    def __init__(self, n):
        self.data = [0.0] * n
    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]
    def reset(self):
        self.data = [0.0] * len(self.data)
    def __getitem__(self, idx):
        return self.data[idx]
```

由于我们使用随机权重初始化 net 模型，因此该模型的精度应接近于随机猜测。例如在有 10 个类别情况下的精度为 0.1。

```
evaluate_accuracy(net, test_iter)
```

1.5.6 训练

首先，我们定义一个函数来训练一个迭代周期。请注意，updater 是更新模型参数的常用函数，它接受批量大小作为参数。它可以是 d2l.sgd 函数，也可以是框架的内置优化函数。

```
def train_epoch_ch3(net, train_iter, loss, updater): #@save
"""训练模型一个迭代周期"""
# 将模型设置为训练模式
    if isinstance(net, torch.nn.Module):
        net.train()
    # 训练损失总和、训练准确度总和、样本数
    metric = Accumulator(3)
    for X, y in train_iter:
        # 计算梯度并更新参数
        y_hat = net(X)
        l = loss(y_hat, y)
        if isinstance(updater, torch.optim.Optimizer):
            # 使用PyTorch内置的优化器和损失函数
            updater.zero_grad()
            l.mean().backward()
            updater.step()
        else:
            # 使用定制的优化器和损失函数
            l.sum().backward()
            updater(X.shape[0])
        metric.add(float(l.sum()), accuracy(y_hat, y), y.numel())
    # 返回训练损失和训练精度
    return metric[0] / metric[2], metric[1] / metric[2]
```

在展示训练函数的实现之前，我们定义一个在动画中绘制数据的实用程序类 Animator，它能够简化本书其余部分的代码。

```
class Animator: #@save
"""在动画中绘制数据"""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
    ylim=None, xscale='linear', yscale='linear',
    fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
    figsize=(3.5, 2.5)):
        # 增量地绘制多条线
        if legend is None:
            legend = []
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # 使用lambda函数捕获参数
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts
    def add(self, x, y):
        # 向图表中添加多个数据点
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
```

```
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)
```

接下来我们实现一个训练函数，它会在 train_iter 访问到的训练数据集上训练一个模型 net。该训练函数将会运行多个迭代周期（由 num_epochs 指定）。在每个迭代周期结束时，利用 test_iter 访问到的测试数据集对模型进行评估。我们将利用 Animator 类来可视化训练进度。

```
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater): #@save

    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
    legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    assert train_loss < 0.5, train_loss
    assert train_acc <= 1 and train_acc > 0.7, train_acc
    assert test_acc <= 1 and test_acc > 0.7, test_acc
```

作为一个从零开始的实现，我们使用小批量随机梯度下降来优化模型的损失函数，设置学习率为 0.1。

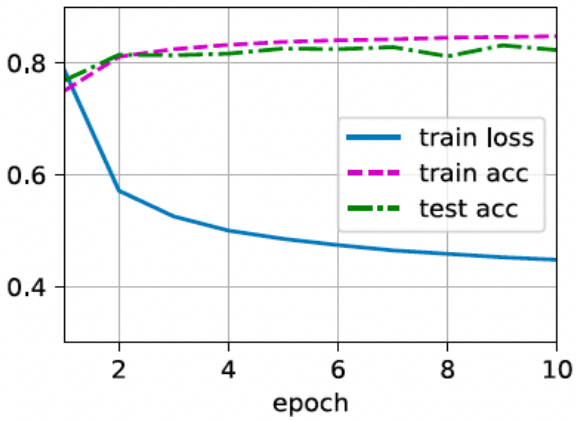
```
lr = 0.1
def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)
```

现在，我们训练模型 10 个迭代周期。请注意，迭代周期（num_epochs）和学习率（lr）都是可调节的超参数。通过更改它们的值，我们可以提高模型的分类精度。

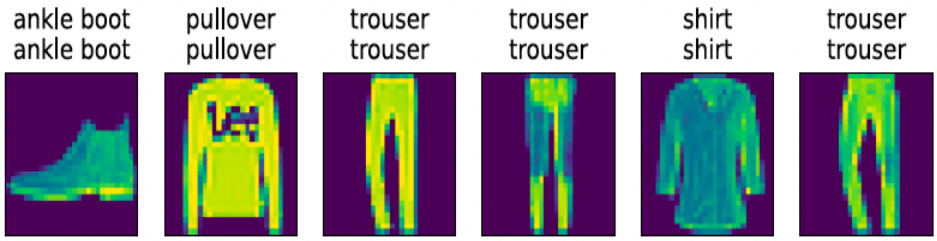
```
num_epochs = 10
train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)
```

1.5.7 预测

现在训练已经完成，我们的模型已经准备好对图像进行分类预测。给定一系列图像，我们将比较它们的实际标签（文本输出的第一行）和模型预测（文本输出的第二行）。




```
def predict_ch3(net, test_iter, n=6): #@save
"""预测标签（定义F第3章）"""
    for X, y in test_iter:
        break
    trues = d2l.get_fashion_mnist_labels(y)
    preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
    titles = [true +'\n' + pred for true, pred in zip(trues, preds)]
    d2l.show_images(
    X[0:n].reshape((n, 28, 28)), 1, n, titles=titles[0:n])
predict_ch3(net, test_iter)
```



1.6 Softmax 回归的简洁实现

```
import torch
from torch import nn
from d2l import torch as d2l
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

1.6.1 初始化模型参数

softmax 回归的输出层是一个全连接层。因此，为了实现我们的模型，我们只需在 Sequential 中添加一个带有 10 个输出的全连接层。同样，在这里 Sequential 并不是必要的，但它是实现深度模型的基础。我们仍然以均值 0 和标准差 0.01 随机初始化权重。

```
# PyTorch不会隐式地调整输入的形状。因此，
# 我们在线性层前定义了展平层（flatten），来调整网络输入的形状
net = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)
net.apply(init_weights);
```

1.6.2 损失函数

```
loss = nn.CrossEntropyLoss(reduction='none')
```

1.6.3 优化算法

在这里，我们使用学习率为 0.1 的小批量随机梯度下降作为优化算法。这与我们在线性回归例子中的相同，这说明了优化器的普适性。

```
    trainer = torch.optim.SGD(net.parameters(), lr=0.1)
```

1.6.4 训练

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```