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

内部资料，请勿传播!!!

目录

第 1 章 深度神经网络

1.1 多层感知机

在前面的章节中，我们介绍了 softmax 回归，然后我们从零开始实现了 softmax 回归，接着使用高级 API 实现了算法，并训练分类器从低分辨率图像中识别 10 类服装。在这个过程中，我们学习了如何处理数据，如何将输出转换为有效的概率分布，并应用适当的损失函数，根据模型参数最小化损失。我们已经在简单的线性模型背景下掌握了这些知识，现在我们可以开始对深度神经网络的探索。

1.1.1 隐藏层

线性模型可能会出错

例如，线性意味着单调假设：任何特征的增大都会导致模型输出的增大（如果对应的权重为正），或者导致模型输出的减小（如果对应的权重为负）。有时这是有道理的。例如，如果我们试图预测一个人是否会偿还贷款。我们可以认为，在其他条件不变的情况下，收入较高的申请人比收入较低的申请人更有可能偿还贷款。但是，虽然收入与还款概率存在单调性，但它们不是线性相关的。收入从 0 增加到 5 万，可能比从 100 万增加到 105 万带来更大的还款可能性。处理这一问题的一种方法是对我们的数据进行预处理，使线性变得更合理，如使用收入的对数作为我们的特征。

然而我们可以很容易找出违反单调性的例子。例如，我们想要根据体温预测死亡率。对体温高于 37 摄氏度的人来说，温度越高风险越大。然而，对体温低于 37 摄氏度的人来说，温度越高风险就越低。在这种情况下，我们也可以通过一些巧妙的预处理来解决问题。例如，我们可以使用与 37 摄氏度的距离作为特征。

但是，如何对猫和狗的图像进行分类呢？增加位置 (13; 17) 处像素的强度是否总是增加（或降低）图像描绘狗的似然？对线性模型的依赖对应于一个隐含的假设，即区分猫和狗的唯一要求是评估单个像素的强度。在一个倒置图像后依然保留类别的世界里，这种方法注定会失败。

与我们前面的例子相比，这里的线性很荒谬，而且我们难以通过简单的预处理来解决这个问题。这是因为任何像素的重要性都以复杂的方式取决于该像素的上下文（周围像素的值）。我们的数据可能会有一种表示，这种表示会考虑到我们在特征之间的相关交互作用。在此表示的基础上建立一个线性模型可能会是合适的，但我们不知道如何手动计算这么一种表示。对于深度神经网络，我们使用观测数据来联合学习隐藏层表示和应用于该表示的线性预测器。

在网络中加入隐藏层

我们可以通过在网络中加入一个或多个隐藏层来克服线性模型的限制，使其能处理更普遍的函数关系类型。要做到这一点，最简单的方法是将许多全连接层堆叠在一起。每一层都输出到上面的层，直到生成最后的输出。我们可以把前 $L-1$ 层看作表示，把最后一层看作线性预测器。这种架构通常称为多层感知机（multilayer perceptron），通常缩写为 MLP。下面，我们以图的方式描述了多层感知机

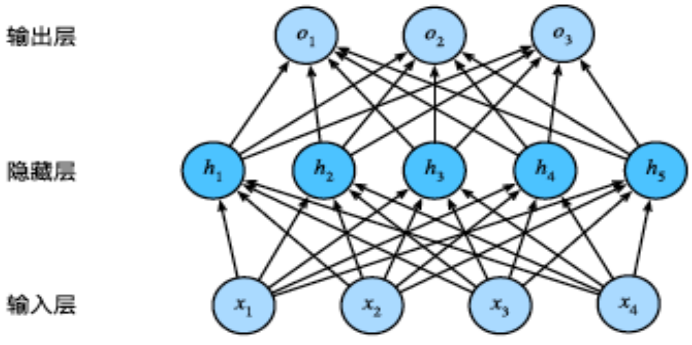


图 1.1: 一个单隐藏层的多层感知机，具有 5 个隐藏单元。

这个多层感知机有 4 个输入，3 个输出，其隐藏层包含 5 个隐藏单元。输入层不涉及任何计算，因此使用此网络产生输出只需要实现隐藏层和输出层的计算。因此，这个多层感知机中的层数为 2。注意，这两个层都是全连接的。每个输入都会影响隐藏层中的每个神经元，而隐藏层中的每个神经元又会影响输出层中的每个神经元。

从线性到非线性

同之前的章节一样，我们通过矩阵 $X \in \mathbb{R}^{n\times d}$ 来表示 n 个样本的小批量，其中每个样本具有 d 个输入特征。对于具有 h 个隐藏单元的单隐藏层多层感知机，用 $H \in \mathbb{R}^{n\times h}$ 表示隐藏层的输出，称为隐藏表示（hidden representations）。在数学或代码中，H 也被称为隐藏层变量（hidden-layer variable）或隐藏变量（hidden variable）。因为隐藏层和输出层都是全连接的，所以我们有隐藏层权重 $W^{(1)} \in \mathbb{R}^{d\times h}$ 和隐藏层偏置 $b^{(1)} \in \mathbb{R}^{n\times h}$ 以及输出层权重 $W^{(2)} \in \mathbb{R}^{h\times q}$ 和输出层偏置 $b^{(2)} \in \mathbb{R}^{1\times q}$。形式上，我们按如下方式计算单隐藏层多层感知机的输出 $O \in \mathbb{R}^{n\times q}$：

$$H = XW^{(1)} + b^{(1)} \tag{1.1}$$

$$O = XW^{(2)} + b^{(2)} \tag{1.2}$$

注意在添加隐藏层之后，模型现在需要跟踪和更新额外的参数。可我们能从中得到什么好处呢？在上面定义的模型里，我们没有好处！原因很简单：上面的隐藏单元由输入的仿射函数给出，而输出（softmax 操作前）只是隐藏单元的仿射函数。仿射函数的仿射函数本身就是仿射函数，但是我们之前的线性模型已经能够表示任何仿射函数。

我们可以证明这一等价性，即对于任意权重值，我们只需合并隐藏层，便可产生具有参数 $W = W^{(1)}W^{(2)}$ 和 $b = b^{(1)}W^{(2)} + b^{(2)}$ 的等价单层模型：

$$O = (XW^{(1)} + b^{(1)})W^{(2)} + b^{(2)} = XW^{(1)}W^{(2)} + b^{(1)}W^{(2)} + b^{(2)} = XW + b\,: \tag{1.3}$$

为了发挥多层架构的潜力，我们还需要一个额外的关键要素：在仿射变换之后对每个隐藏单元应用非线性的激活函数（activation function）σ。激活函数的输出（例如，$\sigma(\cdot)$）被称为活性值（activations）。一般来说，有了激活函数，就不可能再将我们的多层感知机退化成线性模型：

$$H = \sigma(XW^{(1)} + b^{(1)}) \tag{1.4}$$

$$O = XW^{(2)} + b^{(2)} \tag{1.5}$$

由于 X 中的每一行对应于小批量中的一个样本，出于记号习惯的考量，我们定义非线性函数 σ 也以按行的方式作用于其输入，即一次计算一个样本。我们可以使用了 softmax 符号来表示按行操作。但是本节应用于隐藏层的激活函数通常不仅按行操作，也按元素操作。这意味着在计算每一层的线性部分之后，我们可以计算每个活性值，而不需要查看其他隐藏单元所取的值。对于大多数激活函数都是这样。

为了构建更通用的多层感知机，我们可以继续堆叠这样的隐藏层，例如 $H^{(1)} = \sigma_1((XW^{(1)} + b^{(1)}))$ 和 $H^{(2)} = \sigma_2(H^{(1)}W^{(2)} + b^{(2)})$，一层叠一层，从而产生更有表达能力的模型。

1.1.2 激活函数

激活函数（activation function）通过计算加权和并加上偏置来确定神经元是否应该被激活，它们将输入信号转换为输出的可微运算。大多数激活函数都是非线性的。由于激活函数是深度学习的基础，下面简要介绍一些常[E]的激活函数。

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

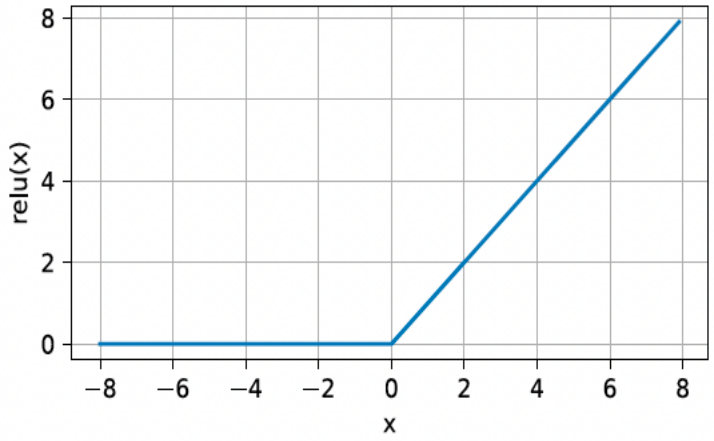
ReLU 函数

最受欢迎的激活函数是修正线性单元（Rectified linear unit，ReLU），因为它实现简单，同时在各种预测任务中表现良好。ReLU 提供了一种非常简单的非线性变换。给定元素 x，ReLU 函数被定义为该元素与 0 的最大值：

$$\mathrm{ReLU}(x) = \max(x; 0)\,. \tag{1.6}$$

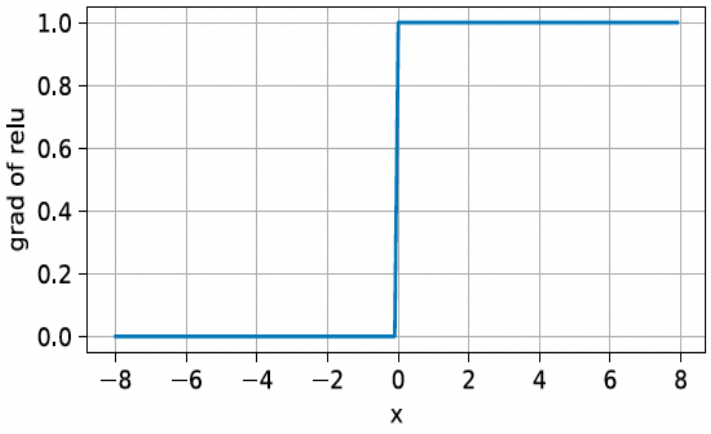
通俗地说，ReLU 函数通过将相应的活性值设为 0，仅保留正元素并丢弃所有负元素。为了直观感受一下，我们可以画出函数的曲线图。正如从图中所看到，激活函数是分段线性的。

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



当输入为负时，ReLU 函数的导数为 0，而当输入为正时，ReLU 函数的导数为 1。注意，当输入值精确等于 0 时，ReLU 函数不可导。在此时，我们默认使用左侧的导数，即当输入为 0 时导数为 0。我们可以忽略这种情况，因为输入可能永远都不会是 0。这里引用一句古老的谚语，“如果微妙的边界条件很重要，我们很可能是在研究数学而非工程”，这个观点正好适用于这里。下面我们绘制 ReLU 函数的导数。

```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



使用 ReLU 的原因是，它求导表现得特别好：要么让参数消失，要么让参数通过。这使得优化表现得更好，并且 ReLU 减轻了困扰以往神经网络的梯度消失问题。

sigmoid 函数

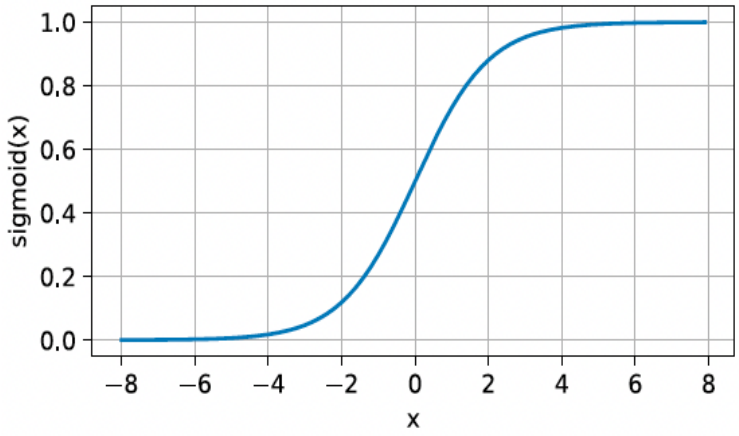
对于一个定义域在 R 中的输入，sigmoid 函数将输入变换为区间 (0, 1) 上的输出。因此，sigmoid 通常称为挤压函数（squashing function）：它将范围（-inf, inf）中的任意输入压缩到区间（0, 1）中的某个值：

$$sigmoid(x) = \frac{1}{1 + exp(x)} \tag{1.7}$$

当人们逐渐关注到到基于梯度的学习时，sigmoid 函数是一个自然的选择，因为它是一个平滑的、可微的阈值单元近似。当我们想要将输出视作二元分类问题的概率时，sigmoid 仍然被广泛用作输出单元上的激活函数（sigmoid 可以视为 softmax 的特例）。然而，sigmoid 在隐藏层中已经较少使用，它在大部分时候被更简单、更容易训练的 ReLU 所取代。

下面，我们绘制 sigmoid 函数。注意，当输入接近 0 时，sigmoid 函数接近线性变换。

```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

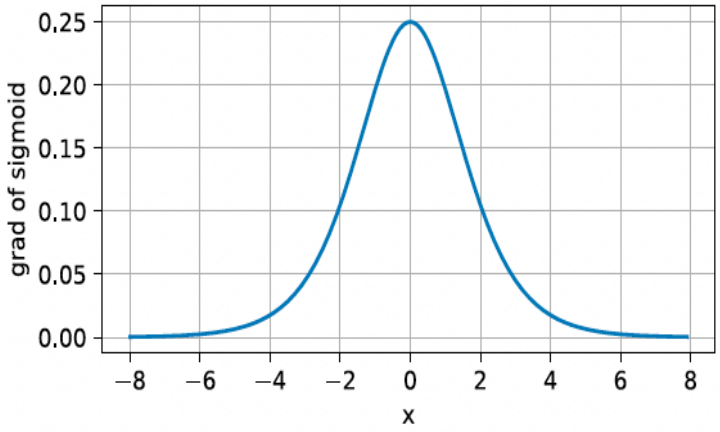


sigmoid 函数的导数为下面的公式：

$$\frac{d}{dx} sigmoid(x) = \frac{exp(-x)}{(1 + exp(-x))^2} = sigmoid(x)(1 - sigmoid(x)) \tag{1.8}$$

sigmoid 函数的导数图像如下所示。注意，当输入为 0 时，sigmoid 函数的导数达到最大值 0.25；而输入在任一方向上越远离 0 点时，导数越接近 0。

```
# 清除以前的梯度
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```

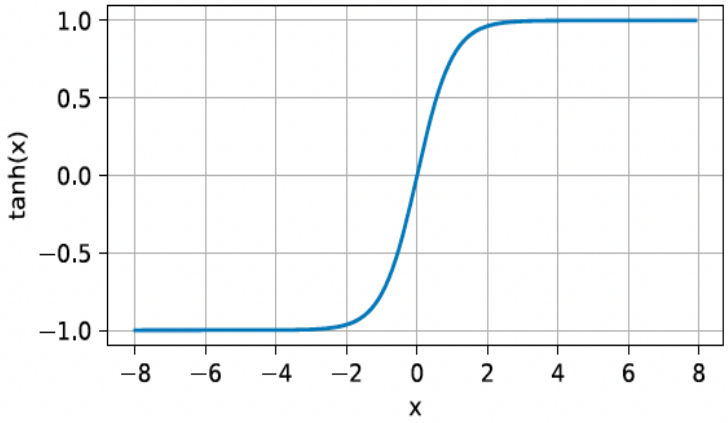


tanh 函数与 sigmoid 函数类似，tanh(双曲正切) 函数也能将其输入压缩转换到区间 (-1, 1) 上。tanh 函数的公式如下：

$$tanh(x) = \frac{1 - exp(-2x)}{1 + exp(-2x)} \tag{1.9}$$

下面我们绘制 tanh 函数。注意，当输入在 0 附近时，tanh 函数接近线性变换。函数的形状类似于 sigmoid 函数，不同的是 tanh 函数关于坐标系原点中心对称。

```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```

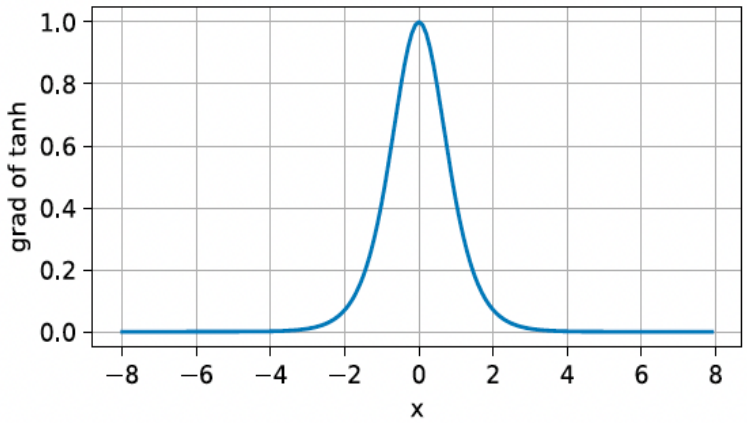


tanh 函数的导数是：

$$\frac{d}{dx}tanh(x) = 1 - tanh^2(x) \tag{1.10}$$

tanh 函数的导数图像如下所示。当输入接近 0 时，tanh 函数的导数接近最大值 1。与我们在 sigmoid 函数图像中看到的类似，输入在任一方向上越远离 0 点，导数越接近 0。

```
# 清除以前的梯度
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



1.2 多层感知机的从零开始实现

我们将继续使用 Fashion-MNIST 图像分类数据集

```
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)
```

1.2.1 初始化模型参数

Fashion-MNIST 中的每个图像由 $28 \times 28 = 784$ 个灰度像素值组成。所有图像共分为 10 个类别。忽略像素之间的空间结构，我们可以将每个图像视为具有 784 个输入特征和 10 个类的简单分类数据集。首先，我们将实现一个具有单隐藏层的多层感知机，它包含 256 个隐藏单元。注意，我们可以将这两个变量都视为超参数。通常，我们选择 2 的若干次幂作为层的宽度。因为内存在硬件中的分配和寻址方式，这么做往往可以在计算上更高效。

我们用几个张量来表示我们的参数。注意，对于每一层我们都要记录一个权重矩阵和一个偏置向量。跟以前一样，我们要为损失关于这些参数的梯度分配内存。

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256
W1 = nn.Parameter(torch.randn(
num_inputs, num_hiddens, requires_grad=True) * 0.01)
b1 = nn.Parameter(torch.zeros(num_hiddens, requires_grad=True))
W2 = nn.Parameter(torch.randn(
num_hiddens, num_outputs, requires_grad=True) * 0.01)
b2 = nn.Parameter(torch.zeros(num_outputs, requires_grad=True))

params = [W1, b1, W2, b2]
```

1.2.2 激活函数

为了确保我们对模型的细节了如指掌，我们将实现 ReLU 激活函数，而不是直接调用内置的 relu 函数。

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

1.2.3 模型

因为我们忽略了空间结构，所以我们使用 reshape 将每个二维图像转换为一个长度为 num_inputs 的向量。只需几行代码就可以实现我们的模型。

```
def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(X@W1 + b1) # 这里“@”代表矩阵乘法
    return (H@W2 + b2)
```

1.2.4 损失函数

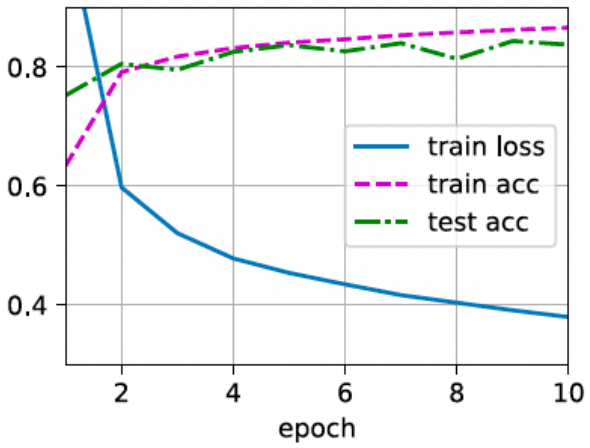
在这里我们直接使用高级 API 中的内置函数来计算 softmax 和交叉熵损失。

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

1.2.5 训练

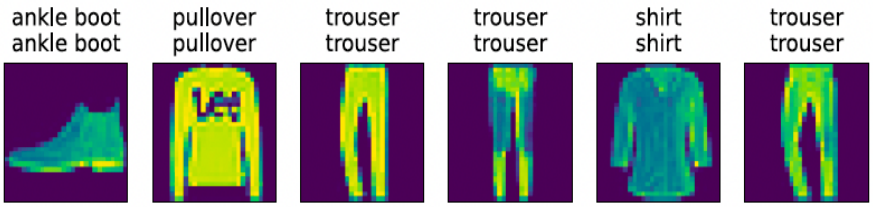
多层感知机的训练过程与 softmax 回归的训练过程完全相同。可以直接调用 d2l 包的 train_ch3 函数，将迭代周期数设置为 10，并将学习率设置为 0.1.

```
num_epochs, lr = 10, 0.1
updater = torch.optim.SGD(params, lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)
```



为了对学习到的模型进行评估，我们将在一些测试数据上应用这个模型。

```
d2l.predict_ch3(net, test_iter)
```



1.3 多层感知机的简洁实现

本节将介绍通过高级 API 更简洁地实现多层感知机。

```
import torch
from torch import nn
from d2l import torch as d2l
```

1.3.1 模型

与 softmax 回归的简洁实现相比，唯一的区别是我们添加了 2 个全连接层（之前我们只添加了 1 个全连接层）。第一层是隐藏层，它包含 256 个隐藏单元，并使用了 ReLU 激活函数。第二层是输出层。

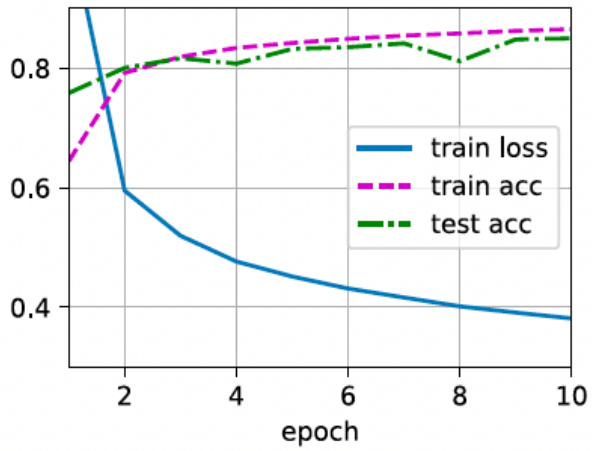
```
net = nn.Sequential(nn.Flatten(), nn.Linear(784, 256), nn.ReLU(), nn.Linear(256, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)
net.apply(init_weights);
```

训练过程的实现与我们实现 softmax 回归时完全相同，这种模块化设计使我们能够将与模型架构有关的内容独立出来。

```
batch_size, lr, num_epochs = 256, 0.1, 10
loss = nn.CrossEntropyLoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=lr)
```

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



1.4 层和块

之前首次介绍神经网络时，我们关注的是具有单一输出的线性模型。在这里，整个模型只有一个输出。注意，单个神经网络（1）接受一些输入；（2）生成相应的标量输出；（3）具有一组相关参数（parameters），更新这些参数可以优化某目标函数。

然后，当考虑具有多个输出的网络时，我们利用矢量化算法来描述整层神经元。像单个神经元一样，层（1）接受一组输入，（2）生成相应的输出，（3）由一组可调整参数描述。当我们使用 softmax 回归时，一个单层本身就是模型。然而，即使我们随后引入了多层感知机，我们仍然可以认为该模型保留了上面所说的基本架构。

对于多层感知机而言，整个模型及其组成层都是这种架构。整个模型接受原始输入（特征），生成输出（预测），并包含一些参数（所有组成层的参数集合）。同样，每个单独的层接收输入（由前一层提供），生成输出（到下一层的输入），并且具有一组可调参数，这些参数根据从下一层反向传播的信号进行更新。

事实证明，研究讨论“比单个层大”但“比整个模型小”的组件更有价值。例如，在计算机视觉中广泛流行的 ResNet-152 架构就有数百层，这些层是由层组（groups of layers）的重复模式组成。这个 ResNet 架构赢得了

2015 年 ImageNet 和 COCO 计算机视觉比赛的识别和检测任务。目前 ResNet 架构仍然是许多视觉任务的首选架构。在其他的领域，如自然语言处理和语音，层组以各种重复模式排列的类似架构现在也是普遍存在。

为了实现这些复杂的网络，我们引入了神经网络块的概念。块（block）可以描述单个层、由多个层组成的组件或整个模型本身。使用块进行抽象的一个好处是可以将一些块组合成更大的组件，这一过程通常是递归的，如图1.2所示。通过定义代码来按需生成任意复杂度的块，我们可以通过简洁的代码实现复杂的神经网络。

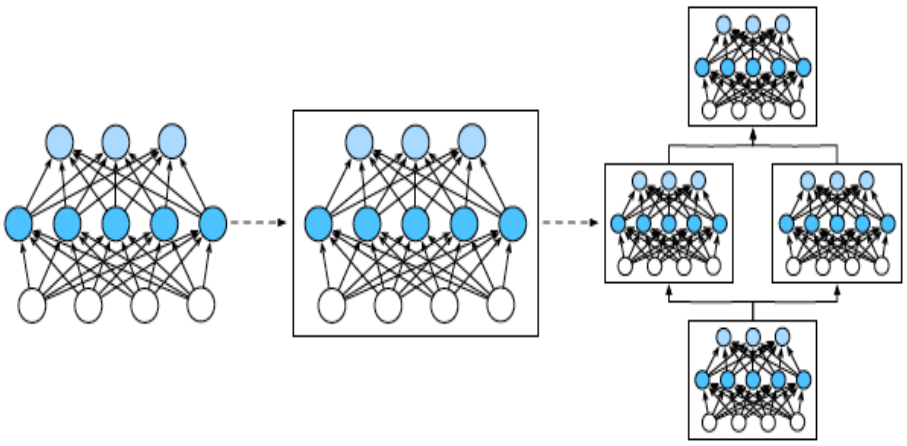


图 1.2: 多个层被组合成块，形成更大的模型。

从编程的角度来看，块由类（class）表示。它的任何子类都必须定义一个将其输入转换为输出的前向传播函数，并且必须存储任何必需的参数。注意，有些块不需要任何参数。最后，为了计算梯度，块必须具有反向传播函数。在定义我们自己的块时，由于自动微分提供了一些后端实现，我们只需要考虑前向传播函数和必需的参数。

在构造自定义块之前，我们先回顾一下多层感知机的代码。下面的代码生成一个网络，其中包含一个具有 256 个单元和 ReLU 激活函数的全连接隐藏层，然后是一个具有 10 个隐藏单元且不带激活函数的全连接输出层。

```
import torch
from torch import nn
from torch.nn import functional as F
net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
X = torch.rand(2, 20)

net(X)

tensor([[ 0.2275, -0.0931, 0.0570, 0.0855, -0.0524, 0.0325, 0.1676, -0.1014,
0.0794, -0.1586],
[ 0.2249, -0.0549, -0.0864, -0.1089, -0.1159, -0.0089, 0.1040, -0.2234,
0.1652, -0.2738]], grad_fn=<AddmmBackward0>)
```

在这个例子中，我们通过实例化 nn.Sequential 来构建我们的模型，层的执行顺序是作为参数传递的。简而言之，nn.Sequential 定义了一种特殊的 Module，即在 PyTorch 中表示一个块的类，它维护了一个由 Module 组成的有序列表。注意，两个全连接层都是 Linear 类的实例，Linear 类本身就是 Module 的子类。另外，到目前为止，我们一直在通过 net(X) 调用我们的模型来获得模型的输出。这实际上是 net.__call__(X) 的简写。这个前向传播函数非常简单：它将列表中的每个块连接在一起，将每个块的输出作为下一个块的输入。

1.4.1 自定义块

要想直观地了解块是如何工作的，最简单的方法就是自己实现一个。在实现我们自定义块之前，我们简要总结一下每个块必须提供的基本功能。

1. 将输入数据作为其前向传播函数的参数。
2. 通过前向传播函数来生成输出。请注意，输出的形状可能与输入的形状不同。例如，我们上面模型中的第一个全连接的层接收一个 20 维的输入，但是返回一个维度为 256 的输出。
3. 计算其输出关于输入的梯度，可通过其反向传播函数进行访问。通常这是自动发生的。
4. 存储和访问前向传播计算所需的参数。
5. 根据需要初始化模型参数。

在下面的代码片段中，我们从零开始编写一个块。它包含一个多层感知机，其具有 256 个隐藏单元的隐藏层和一个 10 维输出层。注意，下面的 MLP 类继承了表示块的类。我们的实现只需要提供我们自己的构造函数（Python 中的 __init__ 函数）和前向传播函数。

```
class MLP(nn.Module):
    # 用模型参数声明层。这里，我们声明两个全连接的层
    def __init__(self):
        # 调用MLP的父类Module的构造函数来执行必要的初始化。
        # 这样，在类实例化时也可以指定其他函数参数，例如模型参数params（稍后将介绍）
        super().__init__()
        self.hidden = nn.Linear(20, 256) # 隐藏层
        self.out = nn.Linear(256, 10) # 输出层

    # 定义模型的前向传播，即如何根据输入X返回所需的模型输出
    def forward(self, X):
        # 注意，这里我们使用ReLU的函数版本，其在nn.functional模块中定义。
        return self.out(F.relu(self.hidden(X)))
```

我们首先看一下前向传播函数，它以 X 作为输入，计算带有激活函数的隐藏表示，并输出其未规范化的输出值。在这个 MLP 实现中，两个层都是实例变量。要了解这为什么是合理的，可以想象实例化两个多层感知机（net1 和 net2），并根据不同的数据对它们进行训练。当然，我们希望它们学到两种不同的模型。

接着我们实例化多层感知机的层，然后在每次调用前向传播函数时调用这些层。注意一些关键细节：首先，我们定制的 __init__ 函数通过 super().__init__() 调用父类的 __init__ 函数，省去了重复编写模版代码的痛苦。然后，我们实例化两个全连接层，分别为 self.hidden 和 self.out。注意，除非我们实现一个新的运算符，否则我们不必担心反向传播函数或参数初始化，系统将自动生成这些。

我们来试一下这个函数：

```
net = MLP()
net(X)
```

```
tensor([[ 0.1981, 0.1234, 0.0592, 0.0819, 0.0648, 0.3781, 0.1214, 0.1906,
-0.0062, 0.1415],
[ 0.2254, 0.0934, -0.0539, 0.1771, 0.0199, 0.2174, 0.2251, 0.3334,
-0.0966, 0.2420]], grad_fn=<AddmmBackward0>)
```

块的一个主要优点是它的多功能性。我们可以子类化块以创建层（如全连接层的类）、整个模型（如上面的 MLP 类）或具有中等复杂度的各种组件。我们在接下来的章节中充分利用了这种多功能性，比如在处理卷积神经网络时。

1.4.2 顺序块

现在我们可以更仔细地看看 Sequential 类是如何工作的，回想一下 Sequential 的设计是为了把其他模块串起来。为了构建我们自己的简化的 MySequential，我们只需要定义两个关键函数：

1. 一种将块逐个追加到列表中的函数；
2. 一种前向传播函数，用于将输入按追加块的顺序传递给块组成的“链条”。

下面的 MySequential 类提供了与默认 Sequential 类相同的功能。

```
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
            # 这里，module是Module子类的一个实例。我们把它保存在'Module'类的成员
            # 变量_modules中。_module的类型是OrderedDict
            self._modules[str(idx)] = module

    def forward(self, X):
        # OrderedDict保证了按照成员添加的顺序遍历它们
        for block in self._modules.values():
            X = block(X)
        return X
```

__init__ 函数将每个模块逐个添加到有序字典 _modules 中。读者可能会好奇为什么每个 Module 都有一个 _modules 属性？以及为什么我们使用它而不是自己定义一个 Python 列表？简而言之，_modules 的主要优点是：在模块的参数初始化过程中，系统知道在 _modules 字典中查找需要初始化参数的子块。当 MySequential 的前向传播函数被调用时，每个添加的块都按照它们被添加的顺序执行。现在可以使用我们的 MySequential 类重新实现多层感知机。

```
net = MySequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
net(X)
```

```
tensor([[ 0.0270, -0.0326, -0.0883, 0.0289, 0.0240, -0.2199, 0.1712, -0.1201,
0.1149, 0.1823],
[-0.0254, -0.0283, -0.1461, 0.1541, 0.2295, -0.3174, -0.0860, 0.0275,
0.0732, 0.1548]], grad_fn=<AddmmBackward0>)
```

请注意，MySequential 的用法与之前为 Sequential 类编写的代码相同

1.4.3 在前向传播函数中执行代码

Sequential 类使模型构造变得简单，允许我们组合新的架构，而不必定义自己的类。然而，并不是所有的架构都是简单的顺序架构。当需要更强的灵活性时，我们需要定义自己的块。例如，我们可能希望在前向传播函数中执行 Python 的控制流。此外，我们可能希望执行任意的数学运算，而不是简单地依赖预定义的神经网络层。

到目前为止，我们网络中的所有操作都对网络的激活值及网络的参数起作用。然而，有时我们可能希望合并既不是上一层的结果也不是可更新参数的项，我们称之为常数参数（constant parameter）。例如，我们需要一个计算函数 $f(x;w) = c \cdot wx$ 的层，其中 x 是输入，w 是参数，c 是某个在优化过程中没有更新的指定常量。因此我们实现了一个 FixedHiddenMLP 类，如下所示：

```
class FixedHiddenMLP(nn.Module):
    def __init__(self):
```

```
        super().__init__()
        # 不计算梯度的随机权重参数。因此其在训练期间保持不变
        self.rand_weight = torch.rand((20, 20), requires_grad=False)
        self.linear = nn.Linear(20, 20)
    def forward(self, X):
        X = self.linear(X)
        # 使用创建的常量参数以及relu和mm函数
        X = F.relu(torch.mm(X, self.rand_weight) + 1)
        # 复用全连接层。这相当于两个全连接层共享参数
        X = self.linear(X)
        # 控制流
        while X.abs().sum() > 1:
            X /= 2
        return X.sum()
```

在这个 FixedHiddenMLP 模型中，我们实现了一个隐藏层，其权重（self.rand_weight）在实例化时被随机初始化，之后为常量。这个权重不是一个模型参数，因此它永远不会被反向传播更新。然后，神经网络将这个固定层的输出通过一个全连接层。

注意，在返回输出之前，模型做了一些不寻常的事情：它运行了一个 while 循环，在 L1 范数大于 1 的条件下，将输出向量除以 2，直到它满足条件为止。最后，模型返回了 X 中所有项的和。注意，此操作可能不会常用于在任何实际任务中，我们只展示如何将任意代码集成到神经网络计算的流程中。

```
net = FixedHiddenMLP()
net(X)
```

```
tensor(-0.2160, grad_fn=<SumBackward0>)
```

我们可以混合搭配各种组合块的方法。在下面的例子中，我们以一些想到的方法嵌套块。

```
class NestMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(20, 64), nn.ReLU(),
        nn.Linear(64, 32), nn.ReLU())
        self.linear = nn.Linear(32, 16)
    def forward(self, X):
        return self.linear(self.net(X))
chimera = nn.Sequential(NestMLP(), nn.Linear(16, 20), FixedHiddenMLP())
chimera(X)
```

```
tensor(-0.4822, grad_fn=<SumBackward0>)
```

1.4.4 参数管理

在选择了架构并设置了超参数后，我们就进入了训练阶段。此时，我们的目标是找到使损失函数最小化的模型参数值。经过训练后，我们将需要使用这些参数来做出未来的预测。此外，有时我们希望提取参数，以便在其他环境中复用它们，将模型保存下来，以便它可以在其他软件中执行，或者为了获得科学的理解而进行检查。

之前的介绍中，我们只依靠深度学习框架来完成训练的工作，而忽略了操作参数的具体细节。本节，我们将介绍以下内容：

1. 访问参数，用于调试、诊断和可视化；
2. 参数初始化；
3. 在不同模型组件间共享参数。

我们首先看一下具有单隐藏层的多层感知机。

```
import torch
from torch import nn
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 1))
X = torch.rand(size=(2, 4))
net(X)
```

```
tensor([[-0.0619],
[-0.0489]], grad_fn=<AddmmBackward0>)
```

1.4.5 参数访问

我们从已有模型中访问参数。当通过 Sequential 类定义模型时，我们可以通过索引来访问模型的任意层。这就像模型是一个列表一样，每层的参数都在其属性中。如下所示，我们可以检查第二个全连接层的参数。

```
print(net[2].state_dict())
```

```
OrderedDict([('weight', tensor([[ 0.3016, -0.1901, -0.1991, -0.1220, 0.1121, -0.1424, -0.3060, 0.
,!3400]])), ('bias', tensor([-0.0291]))])
```

输出的结果告诉我们一些重要的事情：首先，这个全连接层包含两个参数，分别是该层的权重和偏置。两者都存储为单精度浮点数（float32）。注意，参数名称允许唯一标识每个参数，即使在包含数百个层的网络中也是如此。

目标参数

注意，每个参数都表示为参数类的一个实例。要对参数执行任何操作，首先我们需要访问底层的数值。有几种方法可以做到这一点。有些比较简单，而另一些则比较通用。下面的代码从第二个全连接层（即第三个神经网络层）提取偏置，提取后返回的是一个参数类实例，并进一步访问该参数的值。

```
print(type(net[2].bias))
print(net[2].bias)
print(net[2].bias.data)
```

```
<class 'torch.nn.parameter.Parameter'>
Parameter containing:
tensor([-0.0291], requires_grad=True)
tensor([-0.0291])
```

参数是复合的对象，包含值、梯度和额外信息。这就是我们需要显式参数值的原因。除了值之外，我们还可以访问每个参数的梯度。在上面这个网络中，由于我们还没有调用反向传播，所以参数的梯度处于初始状态。

```
net[2].weight.grad == None
```

```
True
```

一次性访问所有参数

当我们需要对所有参数执行操作时，逐个访问它们可能会很麻烦。当我们处理更复杂的块（例如，嵌套块）时，情况可能会变得特别复杂，因为我们需要递归整个树来提取每个子块的参数。下面，我们将通过演示来比较访问第一个全连接层的参数和访问所有层。

```
print(*[(name, param.shape) for name, param in net[0].named_parameters()])
print(*[(name, param.shape) for name, param in net.named_parameters()])
```

这为我们提供了另一种访问网络参数的方式，如下所示。

```
net.state_dict()['2.bias'].data
```

```
tensor([-0.0291])
```

从嵌套块收集参数

让我们看看，如果我们将多个块相互嵌套，参数命名约定是如何工作的。我们首先定义一个生成块的函数（可以说是“块工厂”），然后将这些块组合到更大的块中。

```
def block1():
    return nn.Sequential(nn.Linear(4, 8), nn.ReLU(),nn.Linear(8, 4), nn.ReLU())
```

```
def block2():
    net = nn.Sequential()
    for i in range(4):
        # 在这里嵌套
        net.add_module(f'block {i}', block1())
    return net
rgnet = nn.Sequential(block2(), nn.Linear(4, 1))
rgnet(X)
```

```
tensor([[-0.3078],
[-0.3078]], grad_fn=<AddmmBackward0>)
```

设计了网络后，我们看看它是如何工作的。

```
print(rgnet)
```

因为层是分层嵌套的，所以我们也可以像通过嵌套列表索引一样访问它们。下面，我们访问第一个主要的块中、第二个子块的第一层的偏置项。

```
rgnet[0][1][0].bias.data
```

```
tensor([-0.2539, 0.4913, 0.3029, -0.4799, 0.2022, 0.3146, 0.0601, 0.3757])
```

1.4.6 参数初始化

知道了如何访问参数后，现在我们看看如何正确地初始化参数。我们前面讨论了良好初始化的必要性。深度学习框架提供默认随机初始化，也允许我们创建自定义初始化方法，满足我们通过其他规则实现初始化权重。

默认情况下，PyTorch 会根据一个范围均匀地初始化权重和偏置矩阵，这个范围是根据输入和输出维度计算出的。PyTorch 的 nn.init 模块提供了多种预置初始化方法。

内置初始化

让我们首先调用内置的初始化器。下面的代码将所有权重参数初始化为标准差为 0.01 的高斯随机变量，且将偏置参数设置为 0。

```
def init_normal(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, mean=0, std=0.01)
        nn.init.zeros_(m.bias)
net.apply(init_normal)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([-0.0128, -0.0141, 0.0062, 0.0028]), tensor(0.))
```

我们还可以将所有参数初始化为给定的常数，比如初始化为 1。

```
def init_constant(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 1)
        nn.init.zeros_(m.bias)
net.apply(init_constant)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([1., 1., 1., 1.]), tensor(0.))
```

我们还可以对某些块应用不同的初始化方法。例如，下面我们使用 Xavier 初始化方法初始化第一个神经网络层，然后将第三个神经网络层初始化为常量值 42。

```
def init_xavier(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
def init_42(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 42)
net[0].apply(init_xavier)
net[2].apply(init_42)
print(net[0].weight.data[0])
print(net[2].weight.data)
```

```
tensor([ 0.3809, 0.5354, -0.4686, -0.2376])
tensor([[42., 42., 42., 42., 42., 42., 42., 42.]])
```

1.4.7 参数绑定

有时我们希望在多个层间共享参数：我们可以定义一个稠密层，然后使用它的参数来设置另一个层的参数。

```
shared = nn.Linear(8, 8)
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),shared, nn.ReLU(), shared, nn.ReLU(), nn.Linear(8, 1))
net(X)
# 检查参数是否相同
print(net[2].weight.data[0] == net[4].weight.data[0])
net[2].weight.data[0, 0] = 100
# 确保它们实际上是同一个对象，而不只是有相同的值
print(net[2].weight.data[0] == net[4].weight.data[0])
```

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

1.5 自定义层

深度学习成功背后的一个因素是神经网络的灵活性：我们可以用创造性的方式组合不同的层，从而设计出适用于各种任务的架构。例如，研究人员发明了专用于处理图像、文本、序列数据和执行动态规划的层。有时我们会遇到或要自己发明一个现在在深度学习框架中还不存在的层。在这些情况下，必须构建自定义层。本节将展示如何构建自定义层。

1.5.1 不带参数的层

首先，我们构造一个没有任何参数的自定义层。下面的 CenteredLayer 类要从其输入中减去均值。要构建它，我们只需继承基础层类并实现前向传播功能。

```
import torch
import torch.nn.functional as F
from torch import nn

class CenteredLayer(nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self, X):
        return X - X.mean()
```

让我们向该层提供一些数据，验证它是否能按预期工作。

```
layer = CenteredLayer()
layer(torch.FloatTensor([1, 2, 3, 4, 5]))
```

```
tensor([-2., -1., 0., 1., 2.])
```

现在，我们可以将层作为组件合并到更复杂的模型中。

```
net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
```

作为额外的健全性检查，我们可以在向该网络发送随机数据后，检查均值是否为 0。由于我们处理的是浮点数，因为存储精度的原因，我们仍然可能会看到一个非常小的非零数。

```
Y = net(torch.rand(4, 8))
Y.mean()
```

```
tensor(-1.3970e-09, grad_fn=<MeanBackward0>)
```

1.5.2 带参数的层

以上我们知道了如何定义简单的层，下面我们继续定义具有参数的层，这些参数可以通过训练进行调整。我们可以使用内置函数来创建参数，这些函数提供一些基本的管理功能。比如管理访问、初始化、共享、保存和加载模型参数。这样做的好处之一是：我们不需要为每个自定义层编写自定义的序列化程序。

现在，让我们实现自定义版本的全连接层。回想一下，该层需要两个参数，一个用于表示权重，另一个用于表示偏置项。在此实现中，我们使用修正线性单元作为激活函数。该层需要输入参数：in_units 和 units，分别表示输入数和输出数。

```
class MyLinear(nn.Module):
    def __init__(self, in_units, units):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(in_units, units))
        self.bias = nn.Parameter(torch.randn(units,))
    def forward(self, X):
        linear = torch.matmul(X, self.weight.data) + self.bias.data
        return F.relu(linear)
```

接下来，我们实例化 MyLinear 类并访问其模型参数。

```
linear = MyLinear(5, 3)
linear.weight
```

```
Parameter containing:
tensor([[ 1.9094, -0.8244, -1.6846],
[ 0.6850, 0.8366, -1.3837],
[ 0.0289, 2.0976, 1.3855],
[-0.8574, -0.3557, -0.4109],
[ 2.2963, -1.3008, 1.2173]], requires_grad=True)
```

我们可以使用自定义层直接执行前向传播计算。

```
linear(torch.rand(2, 5))
```

```
tensor([[0.0984, 0.5687, 2.8316],
[2.2558, 0.0000, 1.8880]])
```

我们还可以使用自定义层构建模型，就像使用内置的全连接层一样使用自定义层。

```
net = nn.Sequential(MyLinear(64, 8), MyLinear(8, 1))
net(torch.rand(2, 64))
```

```
tensor([[7.5465],
[4.6817]])
```

1.6 读写文件

到目前为止，我们讨论了如何处理数据，以及如何构建、训练和测试深度学习模型。然而，有时我们希望保存训练的模型，以备将来在各种环境中使用（比如在部署中进行预测）。此外，当运行一个耗时较🄴的训练过程时，最佳的做法是定期保存中间结果，以确保在服务器电源被不小心断掉时，我们不会损失几天的计算结果。因此，现在是时候学习如何加载和存储权重向量和整个模型了。

1.6.1 加载和保存张量

对于单个张量，我们可以直接调用 load 和 save 函数分别读写它们。这两个函数都要求我们提供一个名称，save 要求将要保存的变量作为输入。

```
import torch
from torch import nn
from torch.nn import functional as F
x = torch.arange(4)
torch.save(x, 'x-file')
```

```
x2 = torch.load('x-file')
x2
```

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

我们可以存储一个张量列表，然后把它们读回内存。

```
y = torch.zeros(4)
torch.save([x, y],'x-files')
x2, y2 = torch.load('x-files')
(x2, y2)
```

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

我们甚至可以写入或读取从字符串映射到张量的字典。当我们要读取或写入模型中的所有权重时，这很方便。

```
mydict = {'x': x, 'y': y}
torch.save(mydict, 'mydict')
mydict2 = torch.load('mydict')
mydict2
```

```
{'x': tensor([0, 1, 2, 3]), 'y': tensor([0., 0., 0., 0.])}
```

1.6.2 加载和保存模型参数

保存单个权重向量（或其他张量）确实有用，但是如果我们想保存整个模型，并在以后加载它们，单独保存每个向量则会变得很麻烦。毕竟，我们可能有数百个参数散布在各处。因此，深度学习框架提供了内置函数来保存和加载整个网络。需要注意的一个重要细节是，这将保存模型的参数而不是保存整个模型。例如，如果我们有一个 3 层多层感知机，我们需要单独指定架构。因为模型本身可以包含任意代码，所以模型本身难以序列化。因此，为了恢复模型，我们需要用代码生成架构，然后从磁盘加载参数。让我们从熟悉的多层感知机开始尝试一下。

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.output = nn.Linear(256, 10)
    def forward(self, x):
        return self.output(F.relu(self.hidden(x)))

net = MLP()
X = torch.randn(size=(2, 20))
Y = net(X)
```

接下来，我们将模型的参数存储在一个叫做“mlp.params”的文件中。

```
torch.save(net.state_dict(), 'mlp.params')
```

为了恢复模型，我们实例化了原始多层感知机模型的一个备份。这里我们不需要随机初始化模型参数，而是直接读取文件中存储的参数。

```
clone = MLP()
clone.load_state_dict(torch.load('mlp.params'))
clone.eval()
```

1.7 GPU

我们先看看如何使用单个 NVIDIA GPU 进行计算。首先，确保至少安装了一个 NVIDIA GPU。然后，下载 NVIDIA 驱动和 CUDA80 并按照提示设置适当的路径。当这些准备工作完成，就可以使用 nvidia-smi 命令来查看显卡信息。

```
nvidia-smi
```

在 PyTorch 中，每个数组都有一个设备（device），我们通常将其称为环境（context）。默认情况下，所有变量和相关的计算都分配给 CPU。有时环境可能是 GPU。当我们跨多个服务器部署作业时，事情会变得更加棘手。通过智能地将数组分配给环境，我们可以最大限度地减少在设备之间传输数据的时间。例如，当在带有 GPU 的服务器上训练神经网络时，我们通常希望模型的参数在 GPU 上。

1.7.1 计算设备

我们可以指定用于存储和计算的设备，如 CPU 和 GPU。默认情况下，张量是在内存中创建的，然后使用 CPU 计算它。

在 PyTorch 中，CPU 和 GPU 可以用 torch.device('cpu') 和 torch.device('cuda') 表示。应该注意的是，cpu 设备意味着所有物理 CPU 和内存，这意味着 PyTorch 的计算将尝试使用所有 CPU 核心。然而，gpu 设备只代表一个卡和相应的显存。如果有多个 GPU，我们使用 torch.device(f'cuda:i') 来表示第 i 块 GPU（i 从 0 开始）。另外，cuda:0 和 cuda 是等价的。

```
import torch
from torch import nn
torch.device('cpu'), torch.device('cuda'), torch.device('cuda:1')
```

```
(device(type='cpu'), device(type='cuda'), device(type='cuda', index=1))
```

我们可以查询可用 gpu 的数量。

```
torch.cuda.device_count()
```

现在我们定义了两个方便的函数，这两个函数允许我们在不存在所需所有 GPU 的情况下运行代码。

```
def try_gpu(i=0): #@save
    """如果存在，则返回gpu(i)，否则返回cpu()"""
    if torch.cuda.device_count() >= i + 1:
        return torch.device(f'cuda:{i}')
    return torch.device('cpu')
```

```
def try_all_gpus(): #@save
    """返回所有可用的GPU，如果没有GPU，则返回[cpu(),]"""
    devices = [torch.device(f'cuda:{i}') for i in range(torch.cuda.device_count())]
    return devices if devices else [torch.device('cpu')]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(device(type='cuda', index=0),
device(type='cpu'),
[device(type='cuda', index=0), device(type='cuda', index=1)])
```

1.7.2 张量与 GPU

我们可以查询张量所在的设备。默认情况下，张量是在 CPU 上创建的。

```
x = torch.tensor([1, 2, 3])
x.device
```

```
device(type='cpu')
```

需要注意的是，无论何时我们要对多个项进行操作，它们都必须在同一个设备上。例如，如果我们对两个张量求和，我们需要确保两个张量都位于同一个设备上，否则框架将不知道在哪里存储结果，甚至不知道在哪里执行计算。

存储在 GPU 上

有几种方法可以在 GPU 上存储张量。例如，我们可以在创建张量时指定存储设备。接下来，我们在第一个 gpu 上创建张量变量 X。在 GPU 上创建的张量只消耗这个 GPU 的显存。我们可以使用 nvidia-smi 命令查看显存使用情况。一般来说，我们需要确保不创建超过 GPU 显存限制的数据。

```
X = torch.ones(2, 3, device=try_gpu())
X
```

```
tensor([[1., 1., 1.], [1., 1., 1.]], device='cuda:0')
```

假设我们至少有两个 GPU，下面的代码将在第二个 GPU 上创建一个随机张量。

```
Y = torch.rand(2, 3, device=try_gpu(1))
Y
```

```
tensor([[0.3821, 0.5270, 0.4919],
[0.9391, 0.0660, 0.6468]], device='cuda:1')
```

1.7.3 神经网络与 GPU

类似地，神经网络模型可以指定设备。下面的代码将模型参数放在 GPU 上。

```
net = nn.Sequential(nn.Linear(3, 1))
net = net.to(device=try_gpu())
```

在接下来的几章中，我们将看到更多关于如何在 GPU 上运行模型的例子，因为它们将变得更加计算密集。

当输入为 GPU 上的张量时，模型将在同一 GPU 上计算结果。

```
net(X)
```

```
tensor([[-0.0605],[-0.0605]], device='cuda:0', grad_fn=<AddmmBackward0>)
```

让我们确认模型参数存储在同一个 GPU 上。

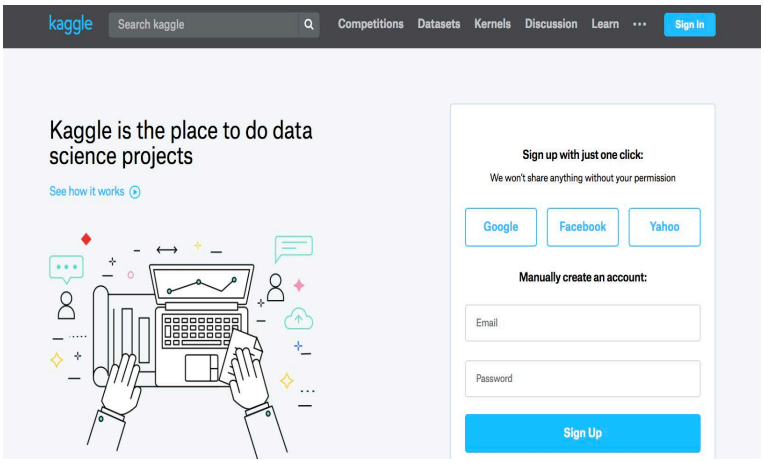
```
net[0].weight.data.device
```

```
device(type='cuda', index=0)
```

1.8 作业

实战 Kaggle 比赛：预测房价

Kaggle是一个当今流行举办机器学习比赛的平台，每场比赛都以至少一个数据集为中心。许多比赛有赞助方，他们为获胜的解决方案提供奖金。该平台帮助用🄴通过论坛和共享代码进行互动，促进协作和竞争。虽然排行榜的追逐往往令人失去理智：有些研究人员短视地专注于预处理步骤，而不是考虑基础性问题。但一个客观的平台有巨大的价值：该平台促进了竞争方法之间的直接定量比较，以及代码共享。这便于每个人都可以学习哪些方法起作用，哪些没有起作用。如果我们想参加 Kaggle 比赛，首先需要注册一个账户。



下载数据

读取数据

```
    #读取csv数据
train_data = pandas.read_csv("train.csv")
test_data = pandas.read_csv("test.csv")

#把去掉id的数据拼在一起，去掉id的目的是为了防止模型通过记住编号得到对应房价。
all_features = pandas.concat(( train_data.iloc[:,1:-1], test_data.iloc[:,1:]))

print("train_data.shape:",train_data.shape)
print("test_data.shape:",test_data.shape)
print("all_features:",all_features.shape)
print(train_data.iloc[:5,:8])
```

处理缺省值和属性

```
#提取全是数字的特征名字
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index

#对数据做标准化处理,对应位置赋值
all_features[numeric_features] = all_features[numeric_features].apply(lambda x: (x - x.mean()) / (x.
    std()))

# 在标准化数据之后，将缺失值设置为0
all_features[numeric_features] = all_features[numeric_features].fillna(0)

#‘Dummy_na=True‘ 将 “na” （缺失值）视为有效的特征值，并为其创建指示符特征。
# pandas.get_dummies把特征为类别值或离散值分成每一个特征为一个类别。
all_features = pandas.get_dummies(all_features, dummy_na = True)
print("all_features.shape:",all_features.shape)
```

分成训练数据和测试数据

```
 #把数据分成训练数据和测试数据
n_train = train_data.shape[0]
train_features = torch.tensor(all_features[:n_train].values, dtype = torch.float32)
test_features = torch.tensor(all_features[n_train:].values, dtype = torch.float32)
train_labels = torch.tensor(train_data.SalePrice.values.reshape(-1, 1), dtype = torch.float32)
print("train_features.shape:", train_features.shape)
print("train_features.shape:", test_features.shape)
print("train_labels:", train_labels.shape)
```

数据分批

```
   batch_size = 32
dataset = torch.utils.data.TensorDataset(train_features, train_labels)
train_loader = torch.utils.data.DataLoader(dataset,            # 数据
                                    batch_size = batch_size, # 每个batch大小
                                    shuffle = True,          # 是否打乱数据
                                    num_workers = 0,         # 工作线程
                                    pin_memory = True)
print(f"每一批{len(next(iter(train_loader))[0])}个，一共{len(train_loader)}批")
```

定义网络

```
class MyNet(torch.nn.Module):
    def __init__(self, in_put, hidden, hidden1, out_put):
        super().__init__()

        # Codes

    def forward(self, data):

        # Codes
        return x
```

初始化神经网络

```
#取出输入特征个数


#损失函数 loss(xi,yi)=(xiyi)2


#梯度优化算法
```

训练神经网络

```
 epochs = 200
def train(train_loader):
    train_ls = []

     for epoch in range(epochs):
        # Codes
```

输出 loss 图像

测试神经网络生成提交数据

```
def test(test_features):
    test_features = test_features.to(device)
    preds = model(test_features).detach().to("cpu").numpy()
    print(preds.squeeze().shape)

    #pandas.Series 创建新维度
    test_data['SalePrice'] = pandas.Series(preds.squeeze())

    #axis选择拼接的维度
    return pandas.concat([test_data['Id'], test_data['SalePrice']], axis = 1)
submission = test(test_features)
submission.to_csv('submission.csv', index=False)
```