

<< [2023-03-19](#) | [2023-03-21](#) >>

If we open a quarrel between past and present, we shall find that we have lost the future.
— Winston Churchill

2. Algorithm

2.1 线性回归

房价预测例子。 [00:02 线性回归](#)

如何在美国买房

- 看中一个房，参观了解
- 估计一个价格，出价

\$5,498,000	7	5	4,865 Sq.Ft.
Price	Beds	Baths	\$1130 / Sq.Ft.

Redfin Estimate: \$5,390,037 On Redfin: 15 days

Virtual Tour

- Branded Virtual Tour
- Virtual Tour (External Link)

Parking Information

- Garage (Minimum): 2
- Garage (Maximum): 2
- Parking Description: Attached Garage, On Street
- Garage Spaces: 2

Multi-Unit Information

- # of Stories: 2

School Information

- Elementary School: El Carmelo Elementary School District: Palo Alto Unified School: Jane Lathrop Elementary School: Palo Alto High School District: Palo Alto Unified School District: Palo Alto Unified

Interior Features

Bedroom Information

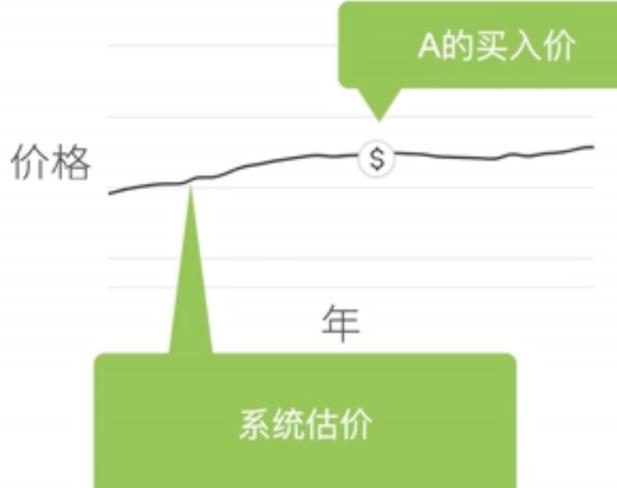
- # of Bedrooms (Minimum): 7
- # of Bedrooms (Maximum): 7



房价预测

很重要，这是真钱

\$100K+ 差价



动手学深度学习 v2 · <https://courses.d2l.ai/zh-v2>

模型简化。03:27 模型简化

1. 假设一：影响房价的关键因素是卧室个数，卫生间个数和居住面积，记为 x_1, x_2, x_3 。
2. 假设二：成交价是关键因素的加权和

$$y = w_1x_1 + w_2x_2 + w_3x_3 + b$$

权重和偏差的实际值在后文给出。

2.1.1 线性模型

2.1.1.1 模型设定



线性模型

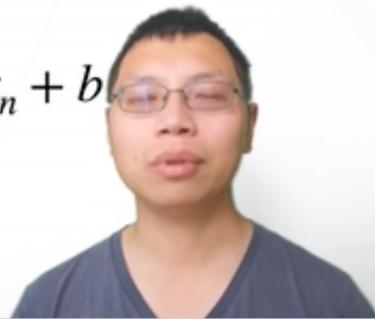
- 给定 n 维输入 $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$
- 线性模型有一个 n 维权重和一个标量偏差

$$\mathbf{w} = [w_1, w_2, \dots, w_n]^T, b$$

- 输出是输入的加权和

$$y = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

向量版本： $y = \langle \mathbf{w}, \mathbf{x} \rangle + b$



动手学深度学习 v2 · <https://courses.d2l.ai/zh-v2>

- 输入： $x = [x_1, x_2, \dots, x_n]^T$
- 线性模型需要确定一个 n 维权重和一个标量偏差

$$\mathbf{w} = [w_1, w_2, \dots, w_n]^T, b$$

- 输出：

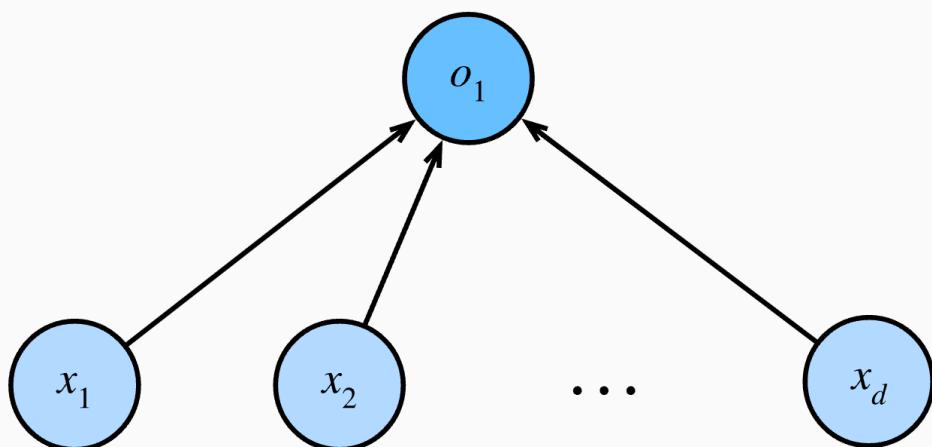
$$y = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

向量版本的是 $y = \langle \mathbf{w}, \mathbf{x} \rangle + b$

Note： 线性模型可以看作是单层神经网络（图片）

输出层

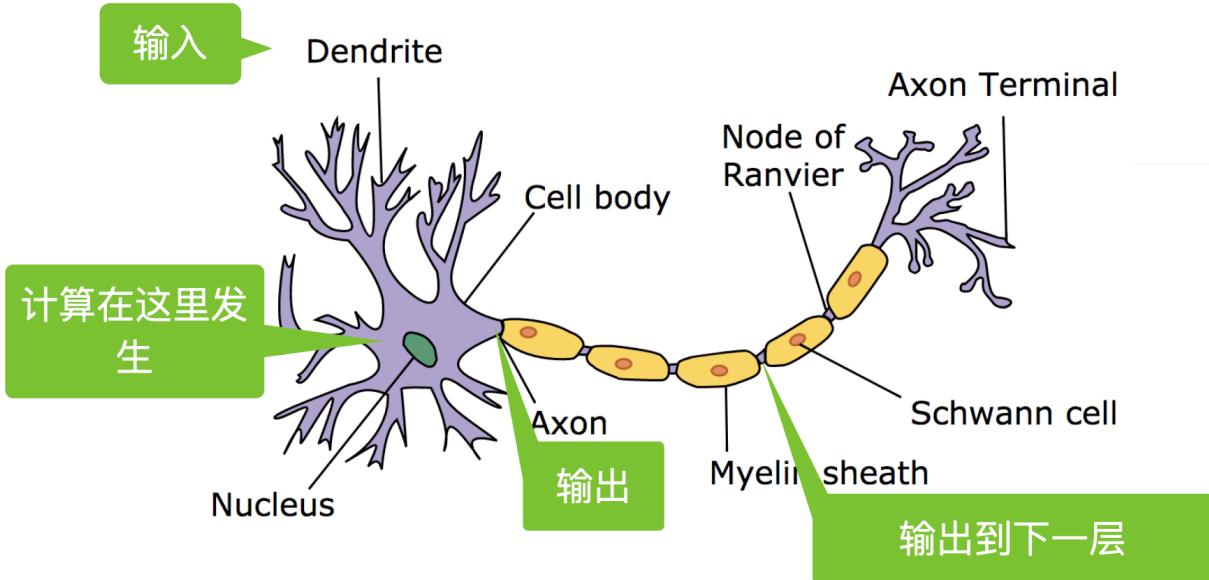
输入层



神经网络源于神经科学：

神经网络源于神经科学

真实的神经元



- 最早的神经网络是源自神经科学的，但是时至今日，很多神经网络已经远远高于神经科学，可解释性也不是很强，不必纠结

2.1.1.2 衡量估计质量

08:08 评价指标

- 需要估计模型的预估值和真实值之间的差距，例如房屋售价和股价
- 假设 y 是真实值， \hat{y} 是估计值，我们可以比较

$$\ell(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

这个叫做**平方损失**

2.1.1.3 训练数据

- 收集一些数据点来决定参数值（权重 ω 和偏差 b ），例如6个月内被卖掉的房子。这称之为训练数据
- 通常越多越好。

Note: 需要注意的是，现实世界的数据都是有限的，但是为了训练出精确的参数往往需要训练数据越多越好，当训练数据不足的时候，我们还需要进行额外处理。

- 假设我们有n个样本，记为

$$X = [x_1, x_2, \dots, x_n]^T, y = [y_1, y_2, \dots, y_n]^T$$

X 的每一行是一个样本， y 的每一行是一个输出的实数值。

2.1.1.4 参数学习

10:41 参数学习

- 训练损失。

训练参数时，需要定义一个损失函数来衡量参数的好坏，应用前文提过的平方损失有公式：

$$\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}, b) = \frac{1}{2n} \sum_{i=1}^n (y_i - \langle \mathbf{x}_i, \mathbf{w} \rangle - b)^2 = \frac{1}{2n} \|\mathbf{y} - \mathbf{X}\mathbf{w} - b\|^2$$

- 最小化损失来学习参数。

训练参数的目的就是使损失函数的值尽可能小（这意味着预估值和真实值更接近）。最后求得的参数值可表示为：

$$\mathbf{w}^*, \mathbf{b}^* = \arg \min_{\mathbf{w}, b} l(\mathbf{X}, \mathbf{y}, \mathbf{w}, b)$$

2.1.1.5 显示解

12:58 显示解

线性回归有显示解，即可以矩阵数学运算，得到参数 \mathbf{w} 和 b 的最优解，而不是用梯度下降，牛顿法等参数优化方式一点点逼近最优解。

推导过程：

- 为了方便矩阵表示和计算，将偏差加入权重， $X \leftarrow [X, 1]$, $\mathbf{w} \leftarrow \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}$

$$\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) = \frac{1}{2n} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 \quad \frac{\partial}{\partial \mathbf{w}} \ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) = \frac{1}{n} (\mathbf{y} - \mathbf{X}\mathbf{w})^T \mathbf{X}$$

- 损失函数是凸函数，最优解满足导数为0，可解出显示解：

$$\frac{\partial}{\partial \omega} l(X, y, \omega) = 0$$

$$\text{有, } \frac{1}{n} (\mathbf{y} - \mathbf{X}\omega)^T \mathbf{X} = 0$$

$$\text{解得: } \omega^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

2.1.1.6 总结

- 线性回归是对 n 维输入的加权，外加偏差
- 使用平方损失来衡量预测值和真实值之间的误差
- 线性回归有显示解
- 线性回归可以看作单层神经网络

2.1.2 基础优化算法

2.1.2.1 梯度下降

当模型没有显示解的时候，应用梯度下降法逼近最优解。梯度下降法的具体步骤：

- 挑选一个初始值 w_0 ,
- 重复迭代参数，迭代公式为：

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \frac{\partial \ell}{\partial \mathbf{w}_{t-1}}$$

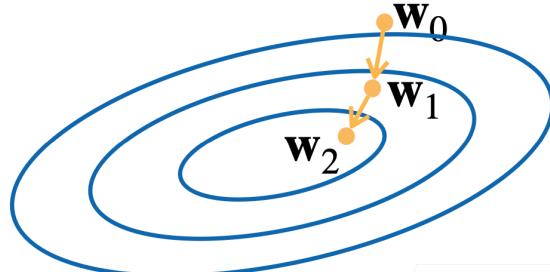
$-\frac{\partial l}{\partial w_{t-1}}$ 为函数值下降最快的方向，学习率 η 为学习步长。

梯度下降

- 挑选一个初始值 \mathbf{w}_0
- 重复迭代参数 $t=1,2,3$

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \frac{\partial \ell}{\partial \mathbf{w}_{t-1}}$$

- 沿梯度方向将增加损失函数值
- 学习率：步长的超参数

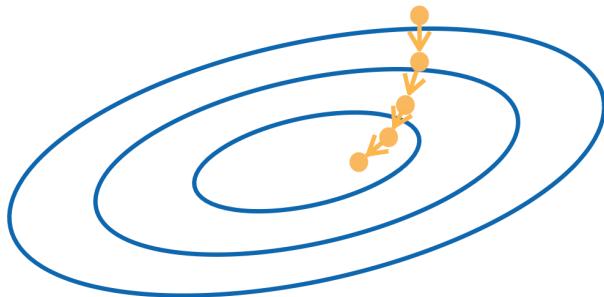


2.1.2.2 选择学习率

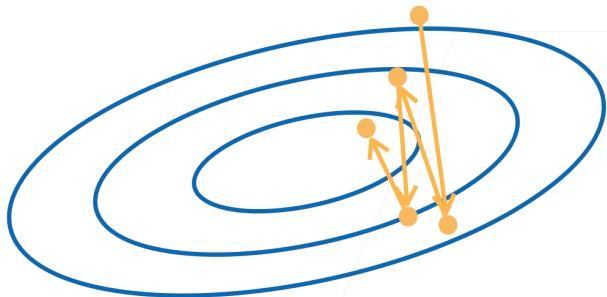
学习率 η 为学习步长，代表了沿负梯度方向走了多远，这是超参数（人为指定的值，不是训练得到的）

选择学习率

不能太小



也不能太大



- Note: 学习率不能太大，也不能太小，需要选取适当。

2.1.2.3 小批量随机梯度下降

04:15 小批量随机梯度下降

在实际应用中，很少直接应用梯度下降法，这是因为在整个训练集上计算梯度代价太大，一个深度神经网络模型可能需要数分钟至数小时。

为了减少运算代价，我们可以 **随机采样** b 个样本 i_1, i_2, \dots, i_b 来近似损失，损失函数为：

$$\frac{1}{b} \sum_{i \in I_b} l(x_i, y_i, \omega)$$

其中 b 是批量大小 **batch size**，也是超参数。

2.1.2.4 选择批量大小

- b 也不能太大：内存消耗增加；浪费计算资源，一个极端的情况是可能会重复选取很多差不多的样本，浪费计算资源
- b 也不能太小：每次计算量太小，很难以并行，不能最大限度利用 **GPU** 资源

2.1.2.5 总结

- 梯度下降通过不断**沿着负梯度方向**更新参数求解
- 小批量随机梯度下降是深度学习默认的求解算法（简单，稳定）
- **两个重要的超参数：批量大小（batch size），学习率（lr）**

2.1.3 线性回归的从零开始实现

00:00 代码实现

在了解线性回归的关键思想之后，可以开始通过代码来动手实现线性回归了。

本节尝试从零开始实现整个方法，
包括数据流水线、模型、损失函数和小批量随机梯度下降优化器。我们将只使用张量和自动求导。
在之后的章节中，会充分利用深度学习框架的优势，介绍更简洁的实现方式。

2.1.3.1 环境配置

```
!pip install d2l  
import numpy as np  
import torch  
from torch.utils import data  
from d2l import torch as d2l
```

language-python

Note: 笔者是 Mac-M1 系统，需要注意的是，此处推荐使用 Python 3.9 环境，如果使用 3.10 版本会报如下错误：

```
ValueError Traceback (most recent call last)  
Input In [1], in <cell line: 4>()  
1 #import torch  
2 #print(torch.__version__)  
----> 4 from d2l import torch as d2l  
  
...  
  
----> 13 from pandas._libs.interval import Interval  
14 from pandas._libs.tslibs import (  
15 NaT,  
16 NaTType  
(...),  
21 iNaT,  
22 )  
  
File pandas/_libs/interval.pyx:1, in init pandas._libs.interval()  
  
ValueError: numpy.ndarray size changed, may indicate binary incompatibility. Expected 96 from  
C header, got 88 from PyObject
```

language-python

解决办法：重装pandas

```
pip install --force-reinstall pandas
```

language-shell

只需要重新安装 miniconda 和 python 版本就好，步骤：

- 从 [官网](#) 下载 3.9 版本的 miniconda，然后运行：

```
# execute the following at the download location:  
sh Miniconda3-py39_23.1.0-1-MacOSX-arm64.sh -b
```

language-shell

- 初始化 miniconda：

```
# initiate the shell  
conda activate ~/miniconda3
```

language-shell

或者（二选一）：

```
~/miniconda3/bin/conda init
```

language-shell

- 创建虚拟环境

```
conda create --name d2l python=3.9 -y
```

language-shell

- 激活虚拟环境

```
conda activate d2l
```

language-shell

- 下载 `torch-gpu` 版本（选）

考虑到后续可能需要 `gpu` 加速训练，因此此处直接下载 `torch-gpu` 版本。

```
pip install --pre torch torchvision torchaudio --extra-index-url  
https://download.pytorch.org/whl/nightly/cpu
```

language-shell

Note: 检查是否安装成功，首先在命令行输入 `python` 进入 `python` 编程环境，然后输入

```
>>> import torch  
>>> print(torch.backends.mps.is_available())  
  
True
```

language-python

若输出结果为 `True`，则表明安装成功。

- 下载 `ipykernel`

考虑到书中给的代码运行环境是 `jupyter`，应该此处安装 `ipykernel`，以确保 `jupyter-notebook` 中可以使用刚刚安装的 `d2l` 虚拟环境的内核。

```
pip install ipykernel
```

language-shell

```
python -m ipykernel install --user --name ENVNAME --display-name DISP_NAME
```

Note: `ENVNAME` 和 `DISP-NAME` 分别为虚拟环境的名字

（此处为 `d2l`）和想要显示的名字。为了简便，此处将两个字段都设置为 `d2l`。

2.1.3.2 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。

任务：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 $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$ 。

使用线性模型参数 $\mathbf{w} = [2, -3.4]^\top$ 、 $b = 4.2$ 和噪声项 ϵ 生成数据集及其标签：

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon$$

可以将 ϵ 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
    @para w 权重
    @para b 偏差
    @para num_examples 样本数量
    @return
        X 随机生成的特征数据, (num_examples, len(w))
        y X对应的标签 (num_examples, 1)

    .....
    X = torch.normal(0, 1, (num_examples, len(w)))#生成均值为0, 方差为1, 数据维度是 (num_examples,
len(w)) 的随机数据作为训练样本
    y = torch.matmul(X, w) + b #生成X对应的预测值
    y += torch.normal(0, 0.01, y.shape)# 加入噪音, 加入的是均值为0, 方差为0.01, 纬度和y.shape一致的噪
音进行干扰
    return X, y.reshape((-1, 1))#返回X, y, y为列向量
```

```
true_w = torch tensor([2, -3.4]) #真实权重
true_b = 4.2 #真实偏差
features, labels = synthetic_data(true_w, true_b, 1000) #随机生成1000组训练数据及标签
```

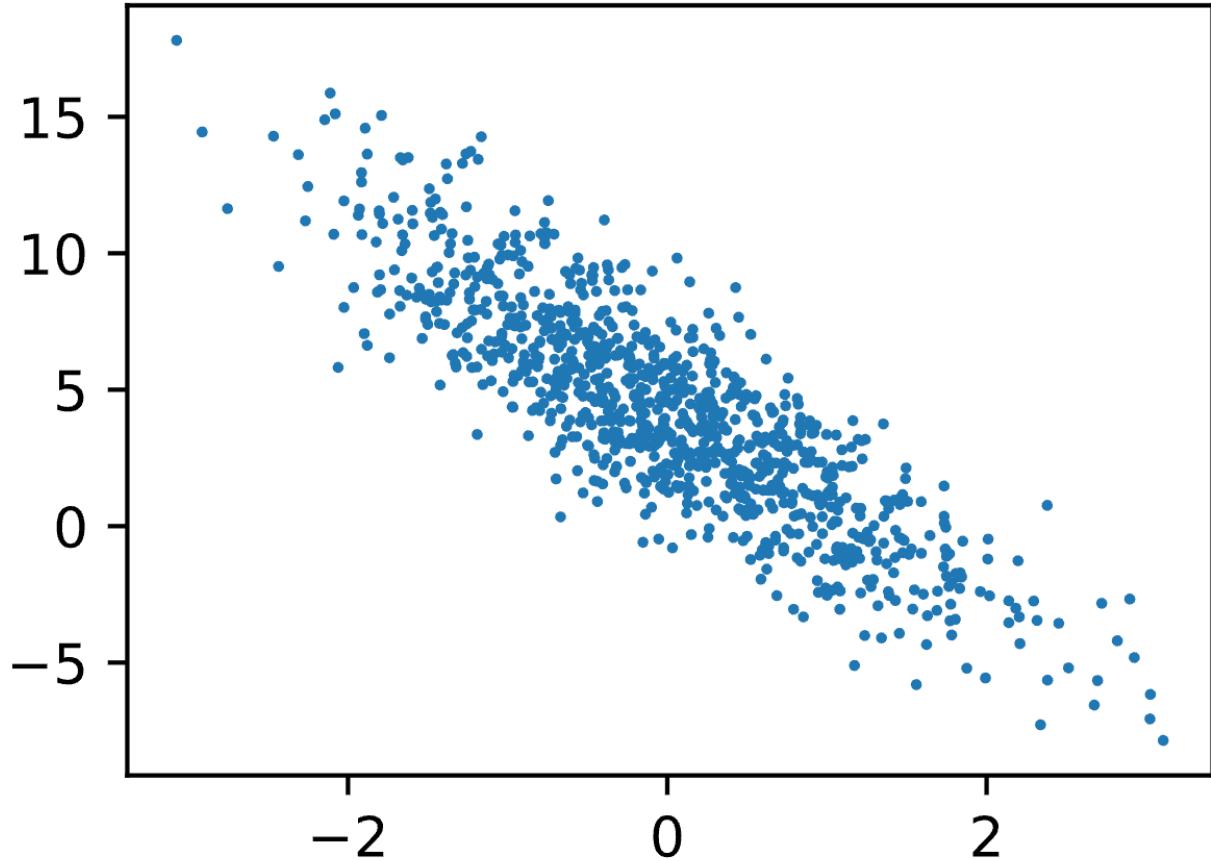
注意：`features`中的每一行都包含一个二维数据样本，`labels`中的每一行都包含一维标签值（一个标量）。

```
>>> print('features:', features[0], '\nlabel:', labels[0]) #打印一个样本数据和对应标签
features: tensor([-1.1678, -0.7740])
label: tensor([4.5072])
```

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。

```
#可视化数据
d2l.set_figsize()
d2l.plt.scatter(features[:, (1)].detach().numpy(), labels.detach().numpy(), 1); #x轴为
features的第一列, y轴为标签值, 正相关
```

Results:



2.1.3.3 读取数据集

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，**定义了一个**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):  
    ....  
    随机获取一小批样本的数据  
    @para batch_size 批量的大小  
    @para features 训练数据  
    @para labels 训练数据对应的标签  
    @return  
        迭代器，每次返回batch_size大小的两组数据，一个是训练样本，一个是对的标签  
    ....  
    num_examples = len(features) #获取样本大小  
    indices = list(range(num_examples)) #获取样本脚标的list  
    # 这些样本是随机读取的，没有特定的顺序  
    random.shuffle(indices) #随机变换indices  
    for i in range(0, num_examples, batch_size): #开始循环  
        batch_indices = torch.tensor(indices[i: min(i + batch_size, num_examples)]) #有可能不能整除，取i + batch_size和num_examples的较小值
```

```
yield features[batch_indices], labels[batch_indices] #相当于是一个迭代器，每次返回  
batch_size个样本
```

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) #X为10 x 2的tensor, y为10 x 1的tensor
    break
```

language=python

```
tensor([[ 0.6301, -0.3610],
       [-1.0436, -0.4737],
       [-0.2955,  0.2614],
       [-0.1482, -0.0150],
       [-0.7068,  1.1198],
       [ 1.3959,  0.2696],
       [ 0.9270,  0.1575],
       [-0.6020,  0.4045],
       [-1.8370,  0.6234],
       [ 0.1756,  0.0142]])

tensor([[ 6.6870],
       [ 3.7135],
       [ 2.7083],
       [ 3.9501],
       [-1.0375],
       [ 6.0819],
       [ 5.5011],
       [ 1.6290],
       [-1.5751],
       [ 4.5169]])
```

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。上面实现的迭代对于教学来说很好，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。

2.1.3.4 初始话模型参数

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，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。

```
# 由于训练的时候需要更细参数，计算梯度，所以requires_grad=True
w = torch.normal(0, 0.01, size=(2,1), requires_grad=True) #w初始化为均值为0, 方差为0.001的符合正态
# 分布的数组，纬度为2 x1
b = torch.zeros(1, requires_grad=True) #b初始化为0, 纬度为1, 就是一个实数
```

language=python

在初始化参数之后，需要更新这些参数，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。因为手动计算梯度很枯燥而且容易出错，所以没有人会手动计算梯度。我们引入自动微分来计算梯度。

2.1.3.5 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 \mathbf{X} 和模型权重 \mathbf{w} 的矩阵-向量乘法后加上偏置 b 。

注意，上面的 $\mathbf{X}\mathbf{w}$ 是一个向量，而 b 是一个标量。

回想一下 [数据操作与数据预处理](#) 中描述的广播机制：

当我们用一个向量加一个标量时，标量会被加到向量的每个分量上。

```
def linreg(X, w, b): #@save
    .....
    线性回归模型
    @para X 训练数据 (num_examples, len(w))
    @para w 权重 (2, 1)
    @para b 偏差 实数
    @return
        模型的预估值
    .....
    return torch.matmul(X, w) + b
```

language=python

2.1.3.6 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。这里使用 [线性回归](#) 中描述的平方损失函数。

Note: 在实现中，我们需要将真实值 y 的形状转换为和预测值 y_{hat} 的形状相同。

```
def squared_loss(y_hat, y): #@save
    .....
    均方损失
    @para y_hat 训练数据的真实值 (num, 1)
    @para y 训练数据的预测值
    @return
        均方误差，没有除以样本数目 (batch_size,1)
    .....
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

language=python

2.1.3.7 定义优化算法

正如我们在 [线性回归](#) 中讨论的，尽管线性回归有解析解，但本书中的其他模型却没有。这里我们介绍小批量随机梯度下降。

- 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 l_r 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
    @para params 参数
    @para lr 学习率, 人为指定
    @para batch_size 批量大小
    @return
    .....
    with torch.no_grad(): #不需要计算梯度
        for param in params:
            param -= lr * param.grad / batch_size #梯度下降法更新参数
            param.grad.zero_() #手动梯度归零

```

language=python

2.1.3.8 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，

你会一遍又一遍地看到几乎相同的训练过程。

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 $\mathbf{g} \leftarrow \partial_{(\mathbf{w}, b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$
 - 更新参数 $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

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。

我们现在忽略这些细节，以后会在 :numref:`chap_optimization` 中详细介绍。

```

lr = 0.003 #学习率
num_epochs = 3 #训练次数
net = linreg #网络，之前定义的线性网络
loss = squared_loss #损失函数，之前定义的平方损失函数

```

language=python

```

#开始训练
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) #计算所有样本的损失函数
        print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')

```

language=python

```

epoch 1, loss 9.117679
epoch 2, loss 4.993863

```

```
epoch 3, loss 2.736007
```

因为我们使用的是自己合成的数据集，所以我们知道真正的参数是什么。因此，可以通过 **比较真实参数和通过训练学到的参数来评估训练的成功程度**。

```
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}')
```

language=python

Results:

```
w的估计误差: tensor([-0.7622, -1.4047], grad_fn=<SubBackward0>)
b的估计误差: tensor([1.7077], grad_fn=<RsubBackward1>)
```

language=python

可以发现，真实参数和通过训练学到的参数确实非常接近。

注：我们不应该想当然地认为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。

2.1.3.9 小结

- 我们学习了深度网络是如何实现和优化的。在这一过程中只使用张量和自动微分，不需要定义层或复杂的优化器。
- 这一节只触及到了表面知识。在下面的部分中，我们将基于刚刚介绍的概念描述其他模型，并学习如何更简洁地实现其他模型。

2.1.4 调用 API 实现

[00:00 简洁实现](#)

在上一节中，我们只运用了：

1. 通过张量来进行数据存储和线性代数；
2. 通过自动微分来计算梯度。

实际上，由于数据迭代器、损失函数、优化器和神经网络层很常用，现代深度学习库也为我们实现了这些组件。

在本节中，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。

2.1.4.1 生成数据

```
true_w = torch.tensor([2, -3.4]) # 真实权重
true_b = 4.2 # 真实偏差
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
# 应用linear-regression-concise里面的函数，生成训练数据

features
```

language=python

Results:

```
tensor([[-1.4684,  0.0977],
       [ 1.7930,  0.9657],
       [ 1.1813,  2.1203],
       ...])
```

language=python

```
[ 0.0698,  0.6992],  
[-1.6323,  2.0169],  
[-0.4198, -1.1236]])
```

2.1.4.2 读取数据集

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  
    @para data_arrays 训练数据  
    @para batch_size 批量大小  
    @para is_train=True 是否训练，选择True会随机选择数据  
    @return  
        迭代器，每次返回batch_size大小的两组数据，一个是训练样本，一个是对应的标签  
    ....  
    dataset = data.TensorDataset(*data_arrays) #数据集  
    return data.DataLoader(dataset, batch_size, shuffle=is_train)
```

Note: dataloader可以理解为数据的一个接口，说明见 [PyTorch中的Data.DataLoader](#),

- 其构造函数为：

```
class torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,  
batch_sampler=None, num_workers=0, collate_fn=<function default_collate>,  
pin_memory=False, drop_last=False, timeout=0, worker_init_fn=None)
```

- `DataLoader` 的参数说明：

- `dataset (Dataset)`: 定义要加载的数据集；
- `batch_size (int, optional)`: 定义`batch_size` 大小，也就是一次加载样本的数量，默认是 `1`；
- `shuffle (bool, optional)`: 在每个epoch开始的时候，是否进行数据重排序，默认 `False`；
- `sampler (Sampler, optional)`: 定义从数据集中取样本的策略，如果进行了指定，那么 `shuffle` 必须是 `False`。
- `num_workers (int, optional)`: 定义加载数据使用的进程数，`0` 代表所有的数据都被装载进主进程，默认是 `0`。
- `drop_last (bool, optional)`: 这个是对最后的未完成的batch来说的，比如你的 `batch_size` 设置为 `64`，而一个epoch 只有 `100` 个样本，如果设置为`True`，那么训练的时候后面的 `36` 个就被扔掉了。如果为 `False`（默认），那么会继续正常执行，只是最后的 `batch_size` 会小一点。

`Dataloader` 的处理逻辑是先通过 `Dataset` 类里面的 `__getitem__` 函数获取单个的数据，然后组合成batch，再使用`collate_fn`所指定的函数对这个batch做一些操作

```
batch_size = 10 # 批量大小  
data_iter = load_array((features, labels), batch_size) # 随机返回的数据
```

使用`data_iter`的方式与我们在 :numref:`sec_linear_scratch`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。

与 :numref:`sec_linear_scratch`不同，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))# iter构造Python迭代器，并使用next从迭代器中获取第一项。
```

language=python

```
[tensor([[ -0.4879,   0.5828],
        [ -0.2912,  -0.0160],
        [ -2.0002,   2.0359],
        [ -0.3812,   1.3547],
        [  2.4167,  -0.4777],
        [  0.3231,  -1.6874],
        [  1.1544,   2.3415],
        [  0.1994,  -0.4488],
        [  0.4064,  -0.7453],
        [  0.6125,   0.6447]]),
 tensor([[ 1.2289,
        [ 3.6639],
        [-6.7296],
        [-1.1639],
        [10.6570],
        [10.5763],
        [-1.4631],
        [ 6.1100],
        [ 7.5428],
        [ 3.2241]]])]
```

language=python

2.1.4.3 定义模型

在上一节 [从零实现线性回归](#) 中实现线性回归时，我们明确定义了模型参数变量，并编写了计算的代码，这样通过基本的线性代数运算得到输出。

但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。

首先定义一个模型变量`net`，它是一个`Sequential`类的实例。

- **Note:** `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。

回顾单层网络架构，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

# Sequential一个容器, list of layers
net = nn.Sequential(nn.Linear(2, 1)) #nn.Linear为线性层函数, 输入的纬度2; 输出的纬度1,
```

language=python

2.1.4.4 初始化模型参数

在使用 `net` 之前，我们需要初始化模型参数——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：

- 权重参数：服从均值为 0、标准差为 0.01 的正态分布
- 偏置参数：初始化为零。

正如在构造 `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) #权重初始化，用normal方式  
>>> net[0].bias.data.fill_(0) #初始化偏差为0  
  
tensor([0.])
```

language=python

2.1.4.5 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() #均方误差函数
```

language=python

2.1.4.6 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。

当实例化一个 `SGD` 实例时，需要指定优化的参数（可通过 `net.parameters()` 从模型中获得）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) #随机梯度下降优化算法SGD，传入参数率lr
```

2.1.4.7 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。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 #迭代周期为3  
for epoch in range(num_epochs):  
    for X, y in data_iter: #返回batch_size大小的数据
```

language=python

```
l = loss(net(X), y) #计算损失函数
trainer.zero_grad() #梯度清零
l.backward() #计算梯度
trainer.step() #更新参数
l = loss(net(features), labels) #计算所有样本的loss
print(f'epoch {epoch + 1}, loss {l:f}')
```

Results:

```
epoch 1, loss 0.000239
epoch 2, loss 0.000095
epoch 3, loss 0.000096
```

language=python

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)
```

language=python

Results:

```
w的估计误差:  tensor([0.0002, 0.0007])
b的估计误差:  tensor([-7.1526e-06])
```

language=python

2.1.5 小结

- 我们可以使用 `PyTorch` 的高级 API 更简洁地实现模型。
- 在 `PyTorch` 中，`data` 模块提供了数据处理工具，`nn` 模块定义了大量的神经网络层和常见损失函数。
- 可以通过 `_` 结尾的方法将参数替换，从而初始化参数。

2.1.6 Q&A

[00:00 Q&A](#)

1. 为什么使用平方损失而不是绝对差值？

其实差别不大，最开始使用平方损失是因为它可导，现在其实都可以使用。

2. 损失为什么要求平均？

本质上没有关系，但是如果不要求平均，梯度的数值会比较大，这时需要学习率除以 n 。如果不除以 n ，可能会随着样本数量的增大而让梯度变得很大。

3. 不管是梯度下降还是随机梯度下降，怎么找到合适的学习率？

- 选择对学习率不敏感的优化方法，比如 `Adam`
- 合理参数初始化

4. 训练过程中，过拟合和欠拟合情况下，学习率和 `batch_size` 应该如何调整？

理论上学习率和 `batch_size` 对最后的拟合结果不会有影响。`batch_size` 越小，最终更容易收敛（一定的噪音对深度学习的泛化性有益）

5. 深度学习上，设置损失函数的时候，需要考虑正则吗？

会考虑，但是和损失函数是分开的，深度学习中正则没有太大的用处，有很多其他的技术可以有正则的效果。

6. 如果样本大小不是批量数的整数倍，需要随机剔除多余的样本吗？

就取多余的样本作为一个批次

直接丢弃

从下一个 epoch 里面补少的样本

2.2 Softmax 回归

2.2.1 Softmax 回归含义

[00:01 Softmax 回归](#)

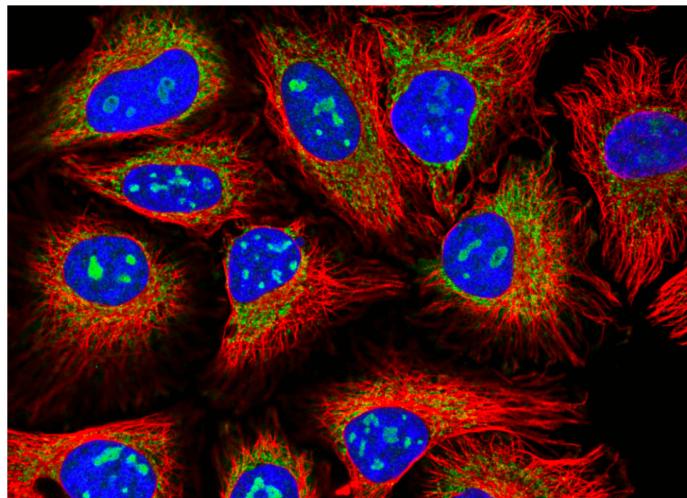
2.2.1.1 回归VS分类：

- 回归估计一个连续值
- 分类预测一个离散类别

典型的分类问题

1. 蛋白质分类 [Link](#)

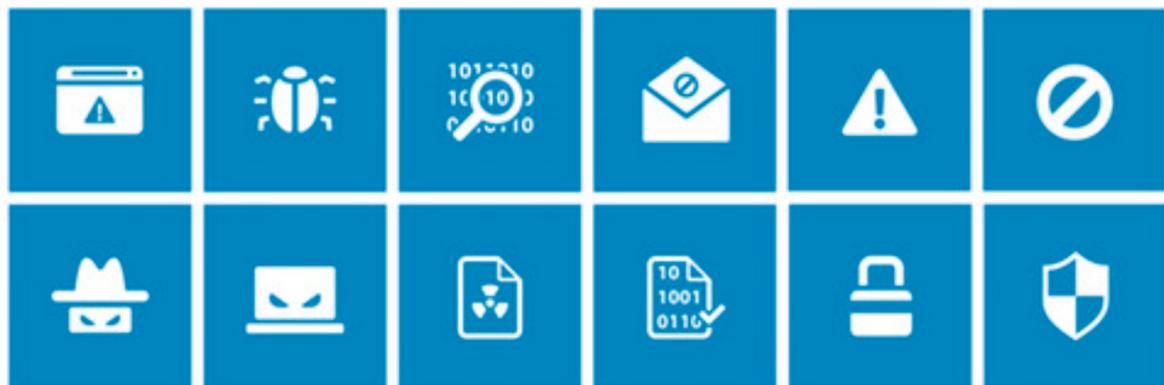
将人类蛋白质显微镜图片分成28类



- 0. Nucleoplasm
- 1. Nuclear membrane
- 2. Nucleoli
- 3. Nucleoli fibrillar
- 4. Nuclear speckles
- 5. Nuclear bodies
- 6. Endoplasmic reticul
- 7. Golgi apparatus
- 8. Peroxisomes
- 9. Endosomes
- 10. Lysosomes
- 11. Intermediate fila
- 12. Actin filaments
- 13. Focal adhesion si
- 14. Microtubules
- 15. Microtubule ends
- 16. Cytokinetic bridg

2. 恶意软件分类 [Link](#)

将恶意软件分成9个类别



3. Wikipedia 评论分类 [Link](#)

将恶意的 Wikipedia 评论分成 7 类

comment_text	toxic	severe_toxic	obscenity
Explanation\nWhy the edits made under my user... n	0	0	0
D'aww! He matches this background colour I'm s... n	0	0	0
Hey man, I'm really not trying to edit war. It... n	0	0	0
"\nMore\nI can't make any real suggestions on ... n	0	0	0
You, sir, are my hero. Any chance you remember... n	0	0	0

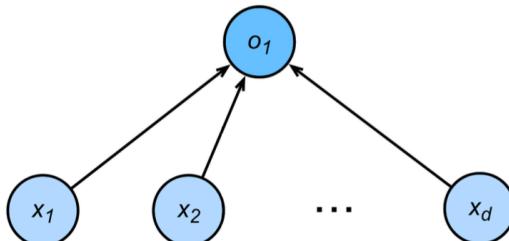
2.2.1.2 从回归到多类分类

[02:37 从回归到分类](#)

从回归到多类分类

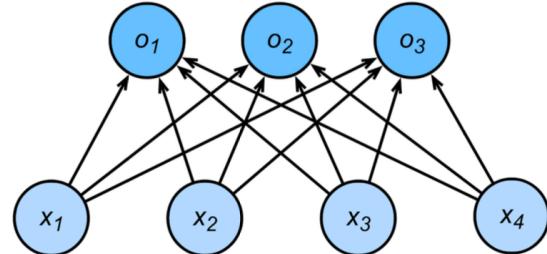
回归

- 单连续数值输出
- 自然区间 \mathbb{R}
- 跟真实值的区别作为损失



分类

- 通常多个输出
- 输出 i 是预测为第 i 类的置信度



1. 回归:

- 单连续数值输出
- 自然区间 \mathbb{R}
- 跟真实值的区别作为损失

2. 分类:

- 通常多个输出
- 输出 i 是预测为第 i 类的置信度

从回归到多类分类——均方损失

- 对类别进行一位有效编码

$$\mathbf{y} = [y_1, y_2, \dots, y_n]^\top$$
$$y_i = \begin{cases} 1 & \text{if } i = y \\ 0 & \text{otherwise} \end{cases}$$

- 使用均方损失训练
- 最大值为预测

$$\hat{y} = \arg \max_i o_i$$

- 需要更置信的识别正确类 (大余量)

$$o_y - o_i \geq \Delta(y, i)$$

从回归到多类分类——校验比例

- 输出匹配概率（非负，和为1）

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$$
$$\hat{y}_i = \frac{\exp(o_i)}{\sum_k \exp(o_k)}$$

- 概率 y 和 \hat{y} 的区别作为损失

2.2.1.3 Softmax和交叉熵损失

- 交叉熵用来衡量两个概率的区别

$$H(\mathbf{p}, \mathbf{q}) = - \sum_i p_i \log(q_i)$$

- 将它作为损失

$$l(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_i y_i \log \hat{y}_i = - \log \hat{y}_y$$

- 其梯度是真实概率和预测概率的区别

$$\partial_{o_i} l(\mathbf{y}, \hat{\mathbf{y}}) = \text{softmax}(\mathbf{o})_i - y_i$$

2.2.1.4 总结

- Softmax回归是一个多类分类模型
- 使用Softmax操作子得到每个类的预测置信度
- 使用交叉熵来衡量和预测标号的区别

2.2.2 损失函数

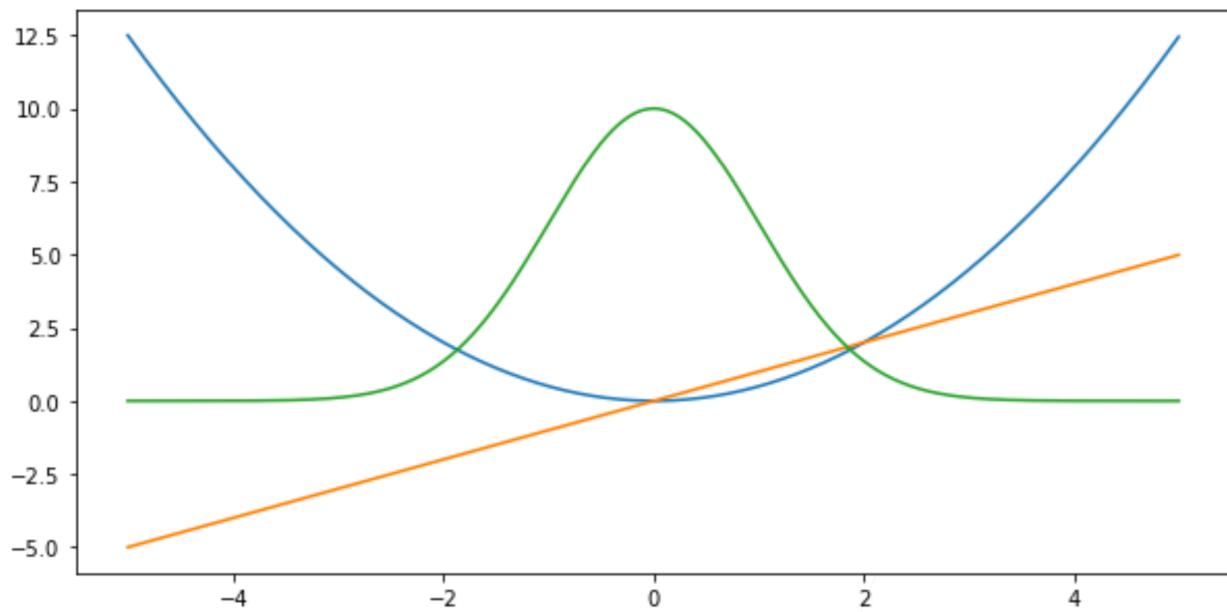
[00:16 常用的损失函数](#)

1. 均方损失 L2 Loss

$$l(y, y') = \frac{1}{2}(y - y')^2$$

L2 Loss

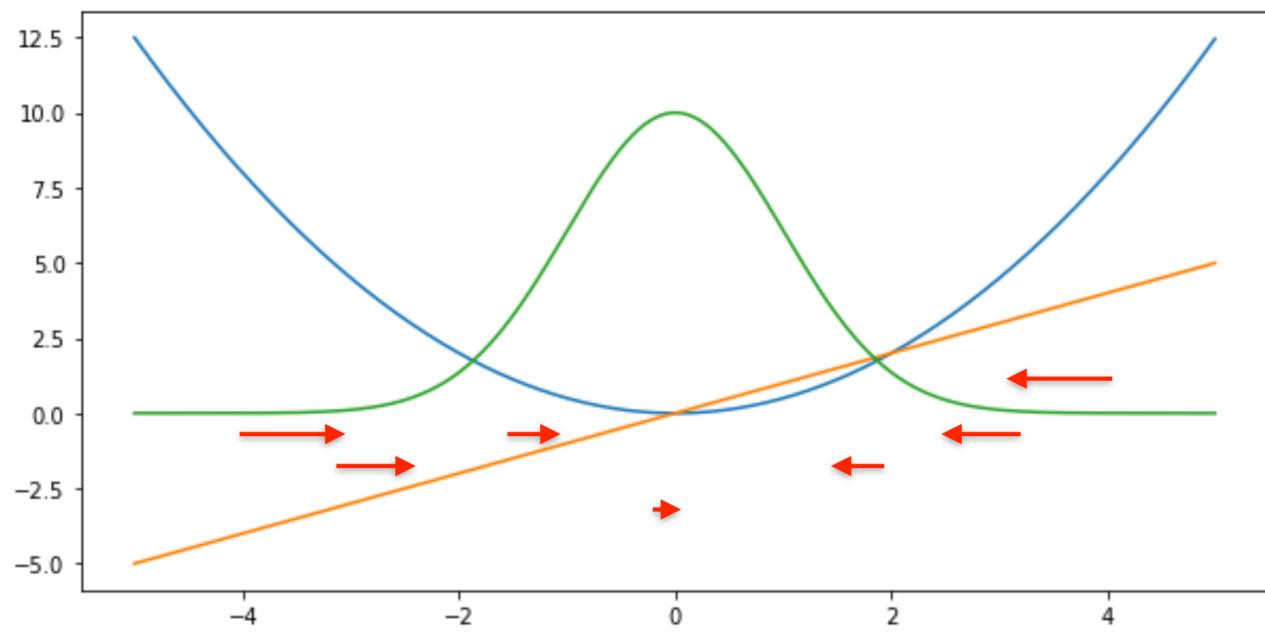
$$l(y, y') = \frac{1}{2}(y - y')^2$$



梯度会随着结果逼近而下降

L2 Loss

$$l(y, y') = \frac{1}{2}(y - y')^2$$

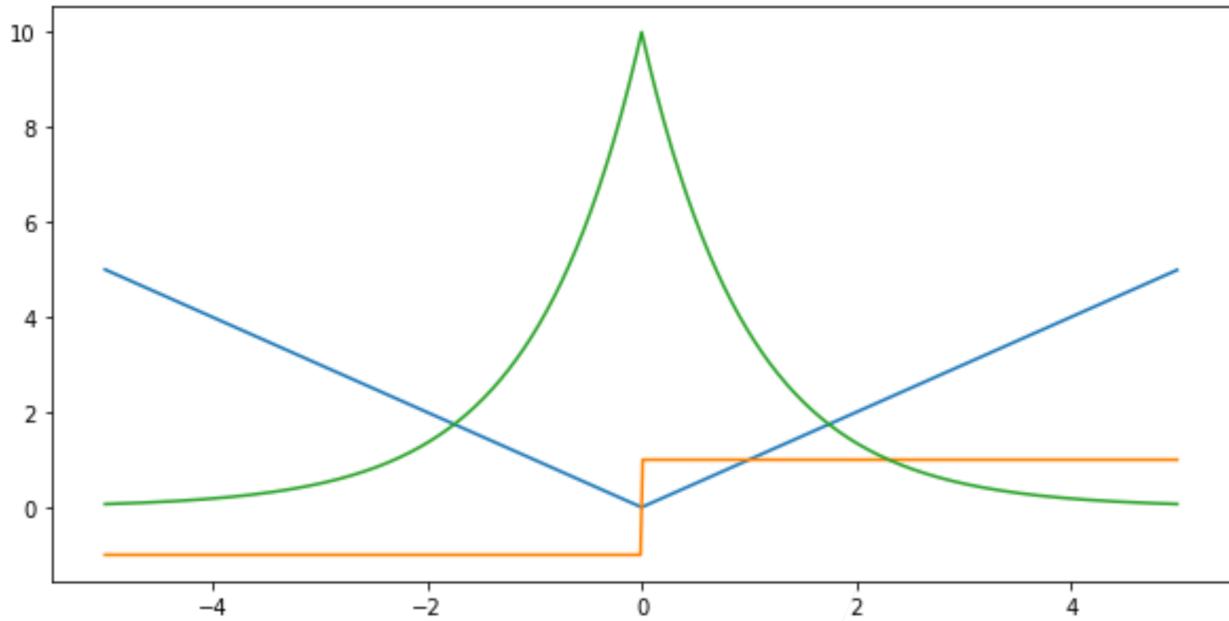


2. L1 Loss

$$l(y, y') = |y - y'|$$

L1 Loss

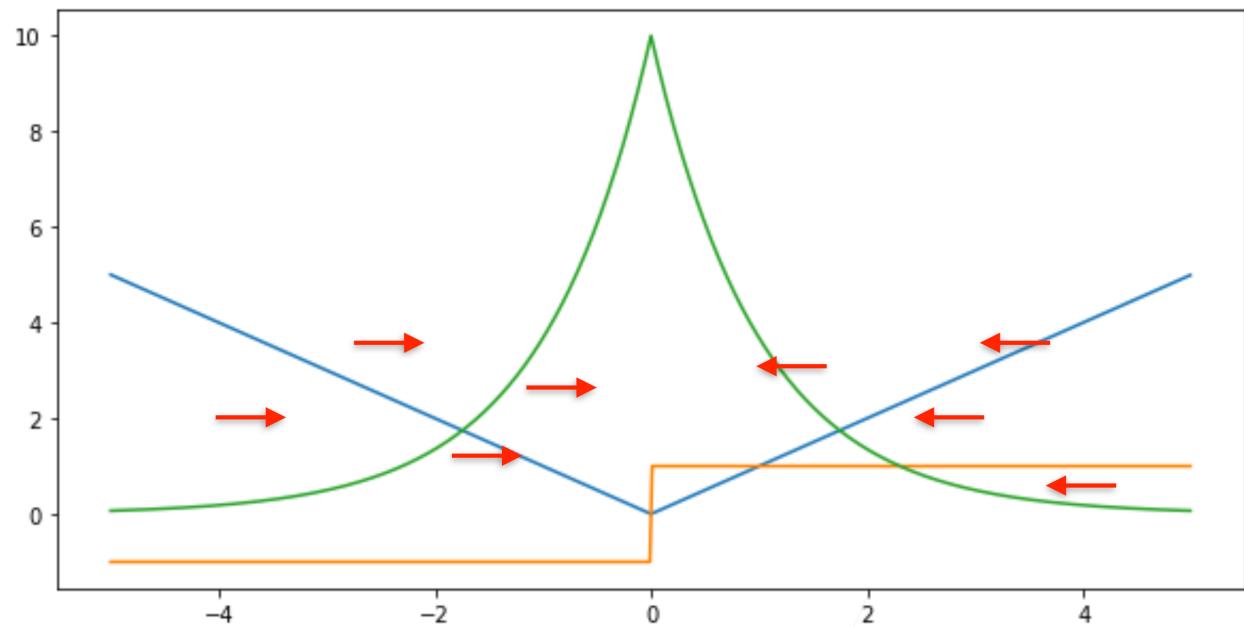
$$l(y, y') = |y - y'|$$



梯度保持不变，但在0处梯度随机

L1 Loss

$$l(y, y') = |y - y'|$$



3. Huber's Robust Loss

$$l(y, y') = \begin{cases} |y - y'| - \frac{1}{2} & \text{if } |y - y'| > 1 \\ \frac{1}{2}(y - y')^2 & \text{otherwise} \end{cases}$$

结合L1 Loss和L2 Loss的优点

2.2.3 图片分类数据集

00:00 图片分类数据集

MNIST 数据集 ([LeCun et al., 1998](#)) 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 ([Xiao et al., 2017](#))：

2.2.3.1 读取数据集

- 导入数据包

```
%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()
```

language=python

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类型转换成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=False)  
mnist_test = torchvision.datasets.FashionMNIST(  
    root="../data", train=False, transform=trans, download=False)
```

language=python

- Note**: 因笔者提前下载了该数据集，因此此处将 `download` 参数设置为 `False`，如果未下载，可以设置为 `True`。

Fashion-MNIST 由 10 个类别的图像组成，

每个类别包含：

- 训练数据集** (train dataset) : 6000张图像
- 测试数据集** (test dataset) : 1000张图像

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)
```

language=python

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])
```

language=python

2.2.3.2 可视化数据集的

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)
- 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]
```

language=python

我们现在可以创建一个函数来可视化这些样本。

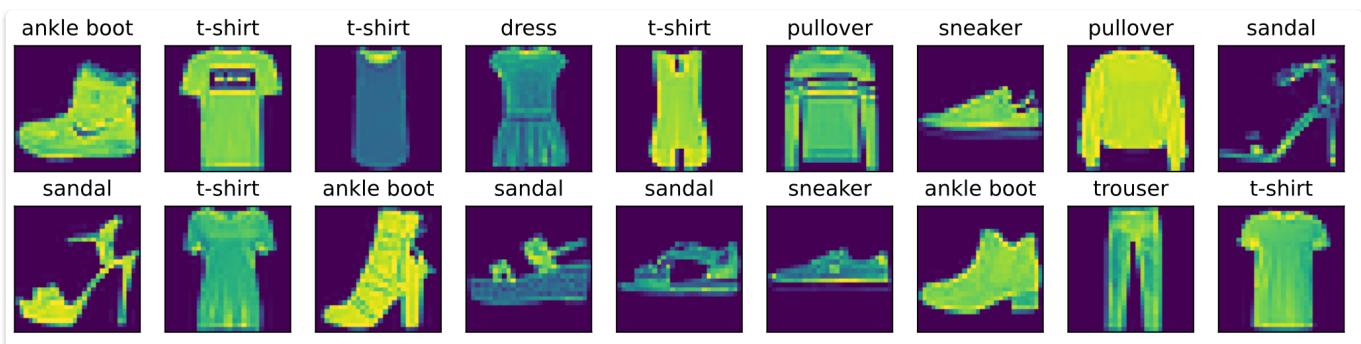
```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5): #@save
    """绘制图像列表"""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        if 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
```

language=python

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))
```

language=python



2.2.3.3 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
language=python

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'
language=python
```

Results:

```
2.03 sec
language=python
```

2.2.3.4 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=False)
    mnist_test = torchvision.datasets.FashionMNIST(
        root="../data", train=False, transform=trans, download=False)
    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
```

language=python

Results:

```
torch.Size([32, 1, 64, 64]) torch.float32 torch.Size([32]) torch.int64
```

language=python

我们现在已经准备好使用Fashion-MNIST数据集，便于下面的章节调用来评估各种分类算法。

2.2.3.5 小结

- Fashion-MNIST是一个服装分类数据集，由10个类别的图像组成。我们将在后续章节中使用此数据集来评估各种分类算法。
- 我们将高度 h 像素，宽度 w 像素图像的形状记为 $h \times w$ 或 (h, w) 。
- 数据迭代器是获得更高性能的关键组件。依靠实现良好的数据迭代器，利用高性能计算来避免减慢训练过程。

2.2.4 从零实现 Softmax回归

[00:35 从零开始实现 Softmax](#)

本节我们将使用上节的 Fashion-MNIST 数据集，并设置数据迭代器的批量大小为256，来实现 softmax 回归的。

```
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)
```

language=python

2.2.4.1 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。此处简单展平每个图像，把它们看作长度为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，但此处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)
```

language=python

2.2.4.2 定义 softmax 操作

```
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)
```

language=python

Results:

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

language=python

- **Note:** 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。

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。

`softmax` 的表达式：

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}.$$

分母或规范化常数，有时也称为 **配分函数**（其对数称为对数-配分函数）。该名称来自 [统计物理学](#) 中一个模拟粒子群分布的方程。

```
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
```

language=python

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)
```

language=python

Results:

```
(tensor([ 0.2135, 0.5900, 0.0449, 0.1064, 0.0453],  
       [ 0.3128, 0.1832, 0.1076, 0.3053, 0.0910])),  
  
 tensor([1.0000, 1.0000]))
```

language=python

Note: 虽然这在数学上看起来是正确的，但我们在代码实现中有点草率。矩阵中的非常大或非常小的元素可能造成数值上溢或下溢，但我们没有采取措施来防止这点。

2.2.4.3 定义模型

[05:37 定义模型](#)

定义操作后，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。

Note: 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)
```

language=python

2.2.4.4 定义损失函数

06:49 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。

Note: 这里不使用 `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]
```

language=python

Results:

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

language=python

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)
```

language=python

Results:

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

language=python

2.2.4.5 分类精度

09:40 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。

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。

1. 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。使用 `argmax` 获得每行中最大元素的索引来获得预测类别。

2. 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。

3. 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())
```

language=python

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)
```

language=python

Results:

0.5

language=python

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]
```

language=python

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):
```

language=python

```

        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)
```

language=python

Results:

0.0508

language=python

2.2.4.6 训练

12:29 训练

在我们看过 [从零开始实现线性回归](#) 中的线性回归实现，softmax 回归的训练过程代码应该看起来非常眼熟。在这里，我们重构训练过程的实现以使其可重复使用。

首先，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(定义见第3章)"""
    # 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]

```

language=python

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=('-', '--', '-.', ':'), nrows=1, ncols=1,
                 figsize=(4.5, 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)

```

language=python

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): #@savet
    """训练模型(定义见第3章)"""
    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)
```

language=python

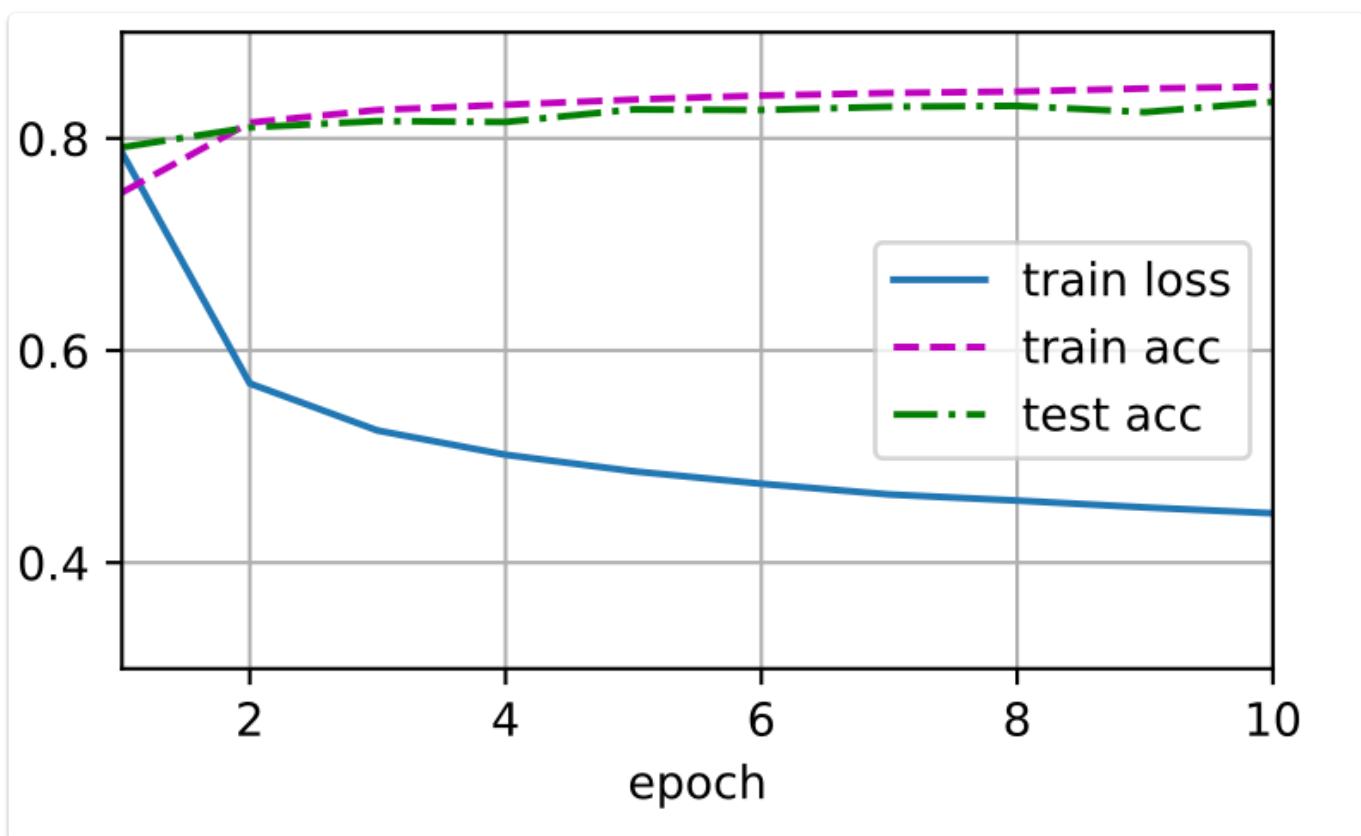
现在，我们训练模型10个迭代周期。

- **Note**: 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)
```

language=python

Results:



2.2.4.7 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（定义见第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))
```

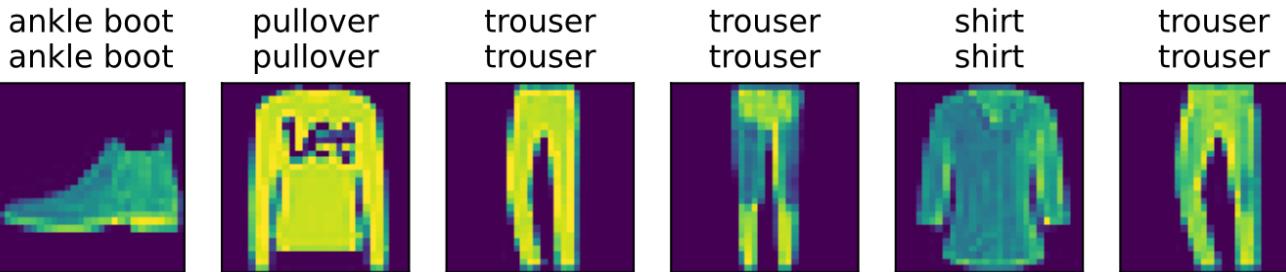
language=python

```

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)

```



2.2.4.8 小结

- 借助 softmax 回归，我们可以训练多分类的模型。
- 训练 softmax 回归循环模型与训练线性回归模型非常相似：先读取数据，再定义模型和损失函数，然后使用优化算法训练模型 python 型。大多数常见的深度学习模型都有类似的训练过程。

2.2.5 Softmax 简洁实现

[00:00 Softmax 简洁实现](#)

在 [简洁实现线性回归](#) 中，我们发现通过深度学习框架的高级API能够使实现线性回归变得更加容易。同样，通过深度学习框架的高级 API 也能更方便地实现 softmax 回归模型。

本节同上一节一样，继续使用 Fashion-MNIST 数据集，并保持批量大小为 256。

```

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)

```

language-python

2.2.5.1 初始化模型参数

如我们在 [Softmax 回归](#) 所述， softmax 回归的输出层是一个全连接层。

因此，为了实现模型，只需在 Sequential 中添加一个带有 10 个输出的全连接层。

- Note**: 同样，在这里 Sequential 并不是必要的，但它是实现深度模型的基础。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);

```

language-python

Note: 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。

2.2.5.2 重新审视Softmax的实现

在前面 [从零实现 Softmax 回归](#) 的例子中，我们计算了模型的输出，然后将此输出送入交叉熵损失。从数学上讲，这是一件完全合理的事情。然而，从计算角度来看，指数可能会造成数值稳定性问题。

回想一下， softmax 函数：

$$\hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$$

其中， \hat{y}_j 是预测的概率分布， o_j 是未规范化的预测 \mathbf{o} 的第 j 个元素。

如果 o_k 中的一些数值非常大，那么 $\exp(o_k)$ 可能大于数据类型容许的最大数字，即 [上溢](#) (overflow)。

这将使分母或分子变为 [inf](#) (无穷大)，

最后得到的是0、[inf](#) 或 [nan](#) (不是数字) 的 \hat{y}_j 。在这些情况下，我们无法得到一个明确定义的交叉熵值。

解决技巧：

- 在继续softmax计算之前，先从所有 o_k 中减去 $\max(o_k)$ 。这里可以看到每个 o_k 按常数进行的移动不会改变 softmax 的返回值：

$$\begin{aligned}\hat{y}_j &= \frac{\exp(o_j - \max(o_k)) \exp(\max(o_k))}{\sum_k \exp(o_k - \max(o_k)) \exp(\max(o_k))} \\ &= \frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}.\end{aligned}$$

- 在减法和规范化步骤之后，可能有些 $o_j - \max(o_k)$ 具有较大的负值。由于精度受限， $\exp(o_j - \max(o_k))$ 将有接近零的值，即 [下溢](#) (underflow)。这些值可能会四舍五入为零，使 \hat{y}_j 为零，并且使得 $\log(\hat{y}_j)$ 的值为 [-inf](#)。反向传播几步后，我们可能会发现自己面对一屏幕可怕的 [nan](#) 结果。

尽管我们要计算指数函数，但我们最终在计算交叉熵损失时会取它们的对数。通过将 softmax 和交叉熵结合在一起，可以避免反向传播过程中可能会困扰我们的数值稳定性问题。

如下面的等式所示，我们避免计算 $\exp(o_j - \max(o_k))$ ，而可以直接使用 $o_j - \max(o_k)$ ，因为 $\log(\exp(\cdot))$ 被抵消了。

$$\begin{aligned}\log(\hat{y}_j) &= \log\left(\frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}\right) \\ &= \log(\exp(o_j - \max(o_k))) - \log\left(\sum_k \exp(o_k - \max(o_k))\right) \\ &= o_j - \max(o_k) - \log\left(\sum_k \exp(o_k - \max(o_k))\right).\end{aligned}$$

我们希望保留传统的 softmax 函数，以用于评估模型输出的概率。但是，我们并没有在交叉熵损失函数中传递 softmax 概率，而是传递了未规范化的预测，并同时计算了 softmax 及其对数。这种方法类似于["LogSumExp技巧"](#)技巧。

2.2.5.3 损失函数

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

language-python

2.2.5.4 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。

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

language=python

2.2.5.5 训练

接下来我们调用上一节中定义的训练函数来训练模型。

```
try:  
    d2l.train_ch3()  
except:  
    print("module 'd2l.torch' has no attribute 'train_ch3'")  
  
try:  
    d2l.train_epoch_ch3()  
except:  
    print("module 'd2l.torch' has no attribute 'train_epoch_ch3'")
```

language=python

Results:

```
module 'd2l.torch' has no attribute 'train_ch3'  
module 'd2l.torch' has no attribute 'train_epoch_ch3'
```

language=python



调用之前定义的训练函数来训练模型

```
In [6]: num_epochs = 10  
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)  
<Figure size 252x180 with 1 Axes>
```



- **Note:** 可以发现，此处直接调用 `d2l.train_ch3` 会报错。这可能是由于笔者安装的 `d2l` 版本与李沐老师在视频中（[02:31 训练模型](#)）展示的代码颇有不同，因此直接调用上一节的部分代码，并进行了改写。

- `train_epoch_ch3` 函数

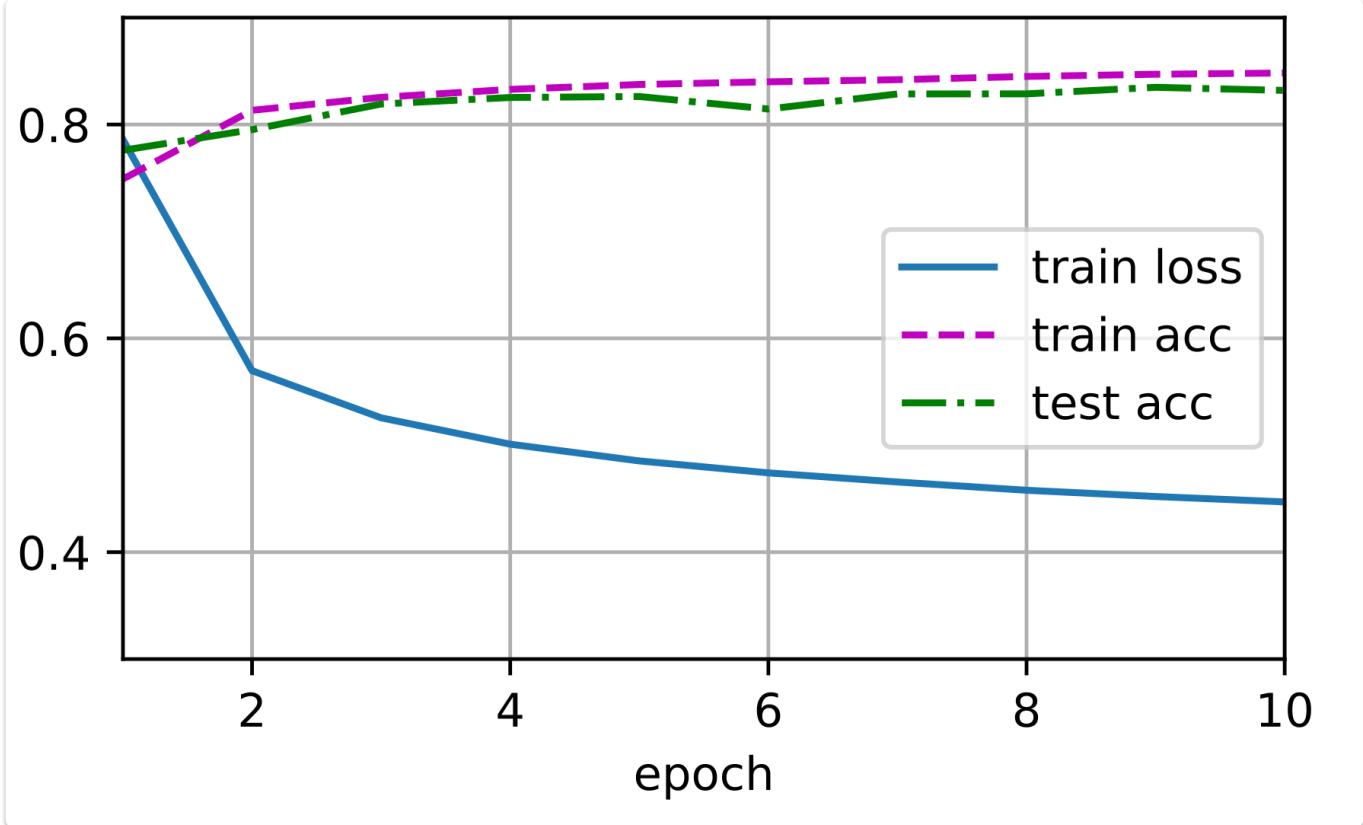
```
def train_epoch_ch3(net, train_iter, loss, updater): #@save      language=python
    """训练模型一个迭代周期（定义见第3章）"""
    # 将模型设置为训练模式
    if isinstance(net, torch.nn.Module):
        net.train()
    # 训练损失总和、训练准确度总和、样本数
    metric = d2l.Accumulator(3)
    for X, y in train_iter:
        # 计算梯度并更新参数
        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()), d2l.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]
```

- `train`

```
num_epochs = 10                                         language=python

def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater): #@save
    """训练模型（定义见第3章）"""
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                            legend=['train loss', 'train acc', 'test acc'], figsize=(4.5, 2.5))
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = d2l.evaluate_accuracy_gpu(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

train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



和以前一样，这个算法使结果收敛到一个相当高的精度，而且这次的代码比之前更精简了。

2.2.5.6小结

- 使用深度学习框架的高级API，我们可以更简洁地实现softmax回归。
- 从计算的角度来看，实现softmax回归比较复杂。在许多情况下，深度学习框架在这些著名的技巧之外采取了额外的预防措施，来确保数值的稳定性。这使我们避免了在实践中从零开始编写模型时可能遇到的陷阱。

2.2.6 softmax回归Q&A

04:09 Q&A

- Q1:softlabel训练策略以及为什么有效？

softmax用指数很难逼近1，softlabel将正例和负例分别标记为0.9和0.1使结果逼近变得可能，这是一个常用的小技巧。

- Q2:softmax回归和logistic回归？

logistic回归为二分类问题，是softmax回归的特例

- Q3:为什么使用交叉熵，而不用相对熵，互信息熵等其他基于信息量的度量？

实际上使用哪一种熵的效果区别不大，所以哪种简单就用哪种

- Q4: $y * \log \hat{y}$ 为什么我们只关心正确类，而不关心不正确的类呢？

并不是不关心，而是不正确的类标号为零，所以算式中不体现，如果使用softlabel策略，就会体现出不正确的类。

- Q5:似然函数曲线是怎么得出来的？有什么参考意义？

最小化损失函数也意味着最大化似然函数，似然函数表示统计概率和模型的拟合程度。

- Q6: 在多次迭代之后如果测试精度出现上升后再下降是过拟合了吗？可以提前终止吗？

很有可能是过拟合，可以继续训练来观察是否持续下降

- Q7:cnn网络主要学习到的是纹理还是轮廓还是所有内容的综合？

目前认为主要学习到的是纹理信息

- Q8:softmax可解释吗？

单纯softmax是可解释的，可以在统计书籍中找到相关的解释。

2.3 多层感知机

00:00 多层感知机

2.3.1 感知机

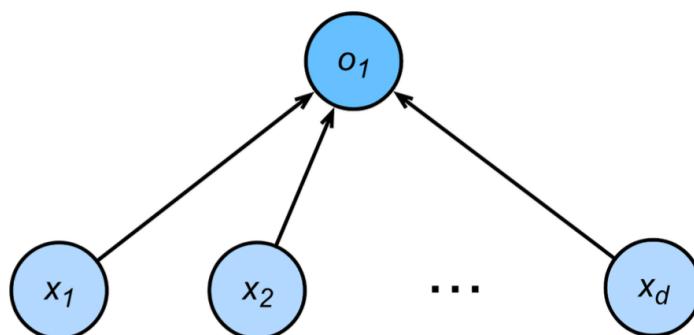
2.3.1.1 定义

从现在的观点来看，感知机实际上就是神经网络中的一个神经单元：

感知机

- 给定输入 \mathbf{x} ，权重 \mathbf{w} ，和偏移 b ，感知机输出：

$$o = \sigma(\langle \mathbf{w}, \mathbf{x} \rangle + b) \quad \sigma(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$



- 给定输入 x ，权重 w 和偏移 b ，感知机的输出为：

$$o = \sigma(\langle \mathbf{w}, \mathbf{x} \rangle + b) \quad \sigma(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

感知机能解决二分类问题，但与线性回归和 softmax 回归有所区别：

- 线性回归与 softmax 回归的输出均为实数
- softmax 回归的输出同时还满足概率公理。

2.3.1.2 训练

[02:37 训练感知机伪代码](#)

训练感知机的伪代码如下：

💡 感知机器训练逻辑 ✓

```

initialize  $w = 0$  and  $b = 0$ 
repeat
    # 此处表达式小于0代表预测结果错误
    if  $y_i[\langle w, x_i \rangle + b] \leq 0$  then
         $w \leftarrow w + y_i x_i$ 
         $b \leftarrow b + y_i$ 
    end if
until all classified

```

可以看出这等价于使用如下损失函数的随机梯度下降 (`batch_size=1`) :

$$\ell(y, \mathbf{x}, \mathbf{w}) = \max(0, -y\langle \mathbf{w}, \mathbf{x} \rangle) = \max(0, -y\mathbf{w}^T \mathbf{x})$$

当预测错误时，偏导数为

$$\frac{\partial \ell}{\partial \mathbf{w}} = -y \cdot \mathbf{x}$$

注：此处为了方便计算，将偏置项 b 归入 w 中的最后一维，并在特征 x 中相应的最后一维加入常数 1。

2.3.1.3 收敛定理

[08:17 收敛定理](#)

收敛定理

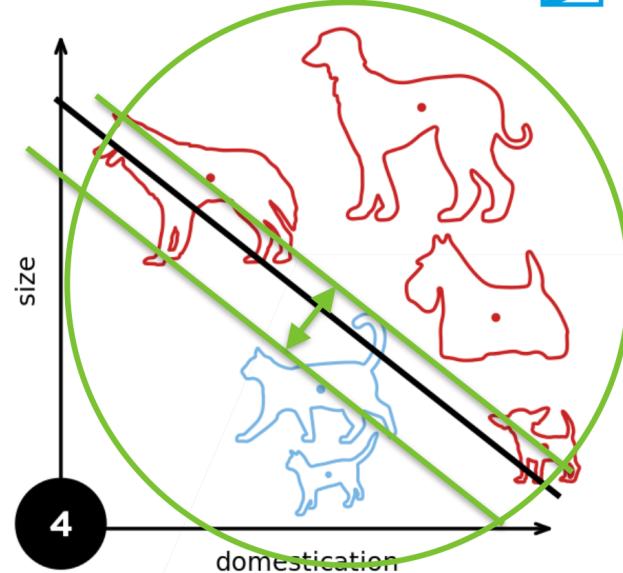


- 数据在半径 r 内
- 余量 ρ 分类两类

$$y(\mathbf{x}^T \mathbf{w} + b) \geq \rho$$

对于 $\|\mathbf{w}\|^2 + b^2 \leq 1$

- 感知机保证在 $\frac{r^2 + 1}{\rho^2}$ 步后收敛



动手学深度学习 v2 • <https://courses.d2l.ai/zh-v2>

设数据在特征空间能被半径为 r 的圆（球）覆盖，并且分类时有余量（即 σ 函数的输入不会取使输出模棱两可的值） $y(\mathbf{x}^T \mathbf{w}) \geq \rho$ ，若初始参数满足 $\|\mathbf{w}\|^2 + b^2 \leq 1$ ，则感知机保证在 $r^2 + 1\rho^2$ 步内收敛。

Note

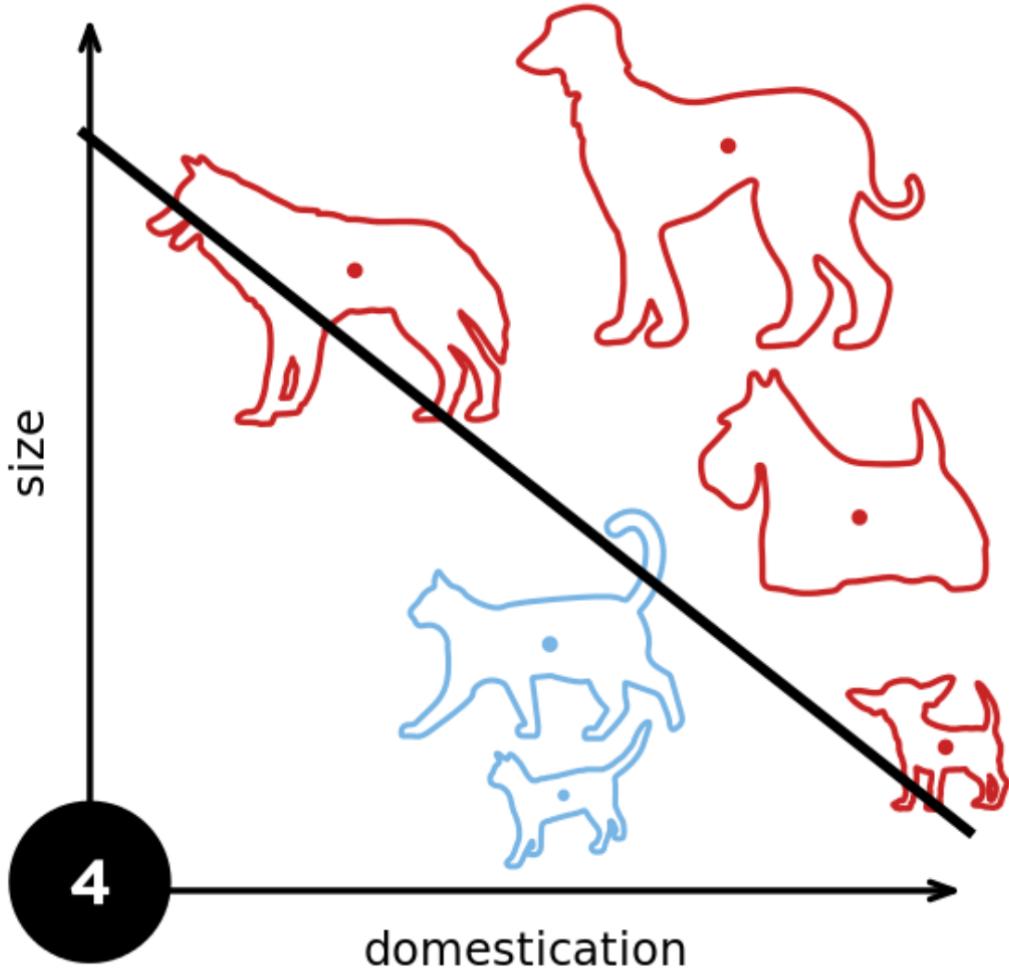
收敛性的证明可以查看：[收敛性的证明](#)

2.3.1.4 线性模型的缺陷

在前面的课程中我们学习了 softmax 回归，线性回归，他们有将输入向量与一个权重向量做内积再与一个偏置相加得到一个值的过程：

$$O = W^T X + b$$

这个过程被称为仿射变换，它是一个带有偏置项的线性变换，它最终产生的模型被称为线性模型，线性模型的特点是只能以线性的方式对特征空间进行划分：



Note

然而，这种线性划分依赖于线性假设，是非常不可靠的

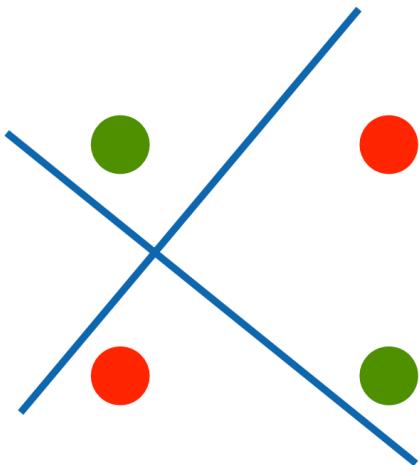
- 线性假设意味着单调假设，这是不可靠的：
 - 对于人体的体温与健康情况的建模，人体在 37°C 时最为健康，过小过大均有风险，然而这不是单调的
- 线性假设意味着特征与预测存在线性相关性，这也是不可靠的：
 - 如果预测一个人偿还债务的可能性，那这个人的资产从0万元增至5万元和从100万元增至105万元对应的偿还债务的可能性的增幅肯定是不相等的，也就是不线性相关的
- 线性模型的评估标准是有位置依赖性的，这是不可靠的：
 - 如果需要判断图片中的动物是猫还是狗，对于图片中一个像素的权重的改变永远是不可靠的，因为如果将图片翻转，它的类别不会改变，但是线性模型不具备这种性质，像素的权重将会失效

XOR 问题：

希望模型能预测出 XOR 分类（分割图片中的一三象限与二四象限）：

XOR 问题 (Minsky & Papert, 1969)

感知机不能拟合 XOR 函数，它只能产生线性分割面



2.3.1.5 总结

Summary

- 感知机是一个二分类模型，是最早的 AI 模型之一；
- 它的求解算法等价于使用批量大小为 1 的梯度下降；
- 它不能你和 XOR 函数，导致第一次 AI 寒冬。

2.3.2 多层感知机

2.3.2.1 XOR问题的多层次解决

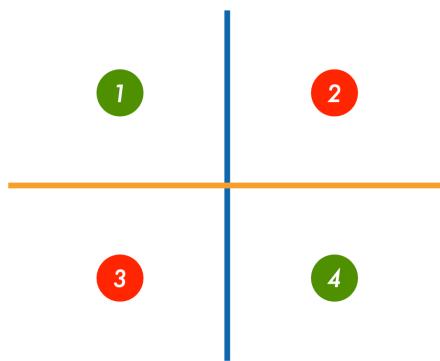
00:02 多层感知机

XOR 问题的一个解决思路是分类两次：

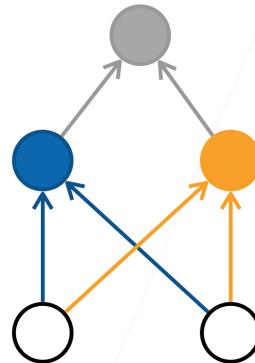
- 先按 x 轴分类为 $+$ 和 $-$ ；
- 再按 y 轴分类为 $+$ 和 $-$ ；
- 最后将两个分类结果相乘， $+$ 即为一三象限， $-$ 即为二四象限：



学习 XOR

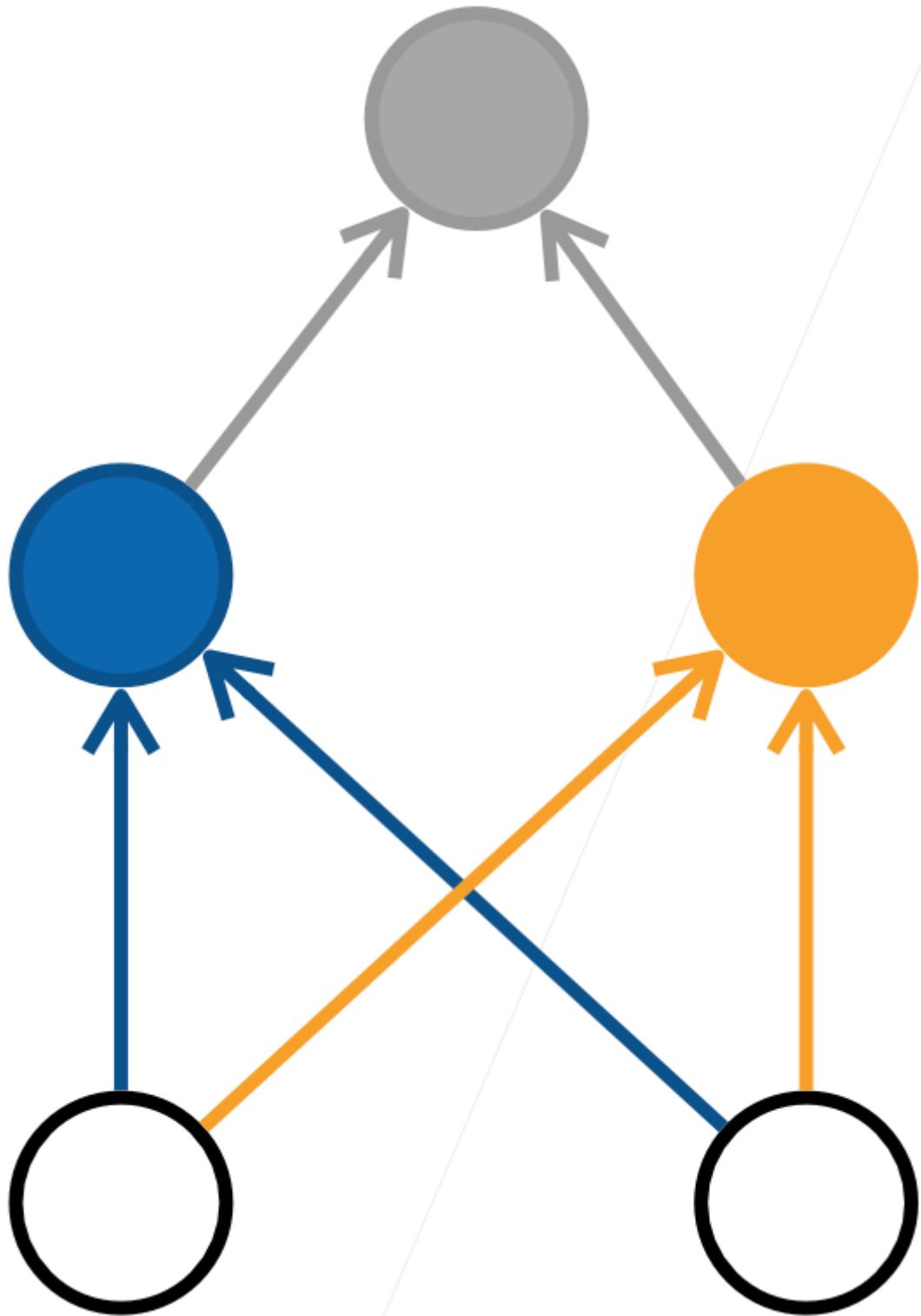


	1	2	3	4
1	+	-	+	-
2	+	+	-	-
product	+	-	-	+



动手学深度学习 v2 · <https://courses.d2l.ai/zh-v2>

这实际上将信息进行了多层次的传递：

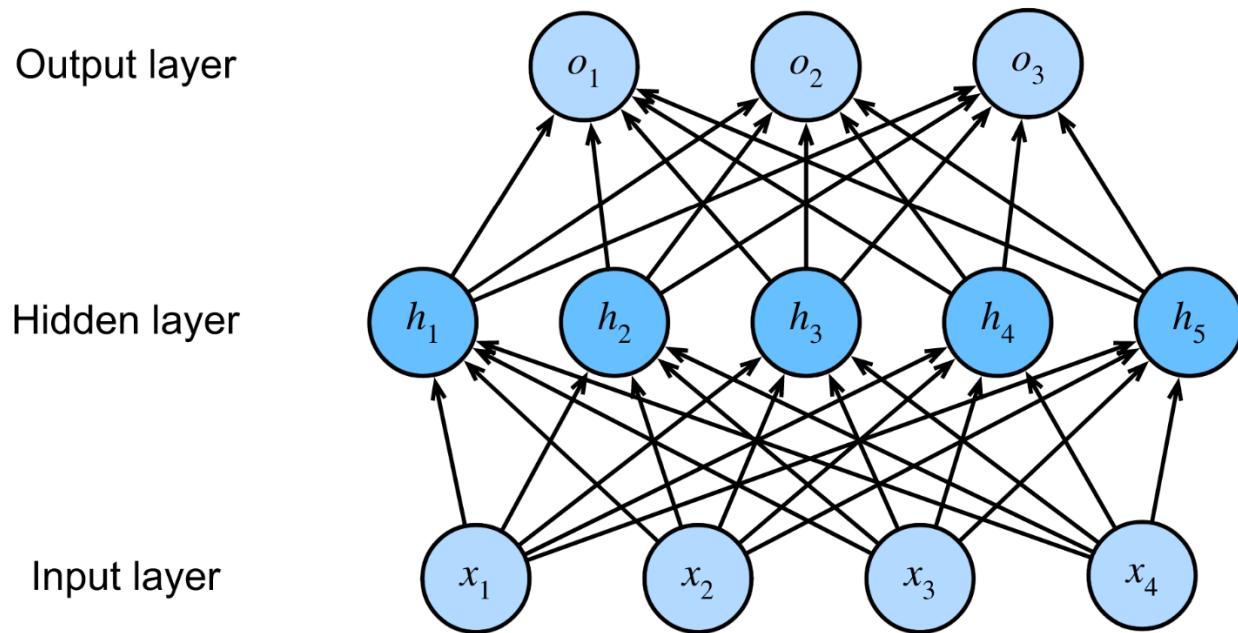


其中蓝色为按 X 坐标的正负进行的分类，橙色为按 Y 坐标的正负进行的分类，灰色为将二者信息的综合，这就实现了用多层次的线性模型对非线性进行预测。

2.3.2.2 多层感知机

[04:22 多层感知机](#)

有了 **XOR** 问题的解决经验，可以想到如果将多个感知机堆叠起来，形成具有多个层次的结构，如下图：



隐藏层大小是超参数

这里的模型称为 **多层感知机**：

- **输入**: x_1, x_2, x_3, x_4 , $\mathbf{x} \in \mathbb{R}^n$
- **隐藏层**: 由 **5** 个感知机构成，均以前一层的信息作为输入。

$$\mathbf{h} = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

其中， $\mathbf{W}_1 \in \mathbb{R}^{m \times n}, \mathbf{b}_1 \in \mathbb{R}^m$ ， σ 是按元素的激活函数。

- **输出层**: 以前一层隐藏层的结果作为输入。

$$o = \mathbf{w}_2^T \mathbf{h} + b_2$$

其中， $\mathbf{w}_2 \in \mathbb{R}^m, b_2 \in \mathbb{R}$

Note

除了输入的信息和最后一层的感知机以外，其余的层均称为隐藏层，隐藏层的设置为模型一个重要的超参数，这里的模型有一个隐藏层。

2.3.2.3 激活函数

[08:23 激活函数](#)

💡 为什么需要非线性激活函数

仅有线性变换是不够的，如果我们简单的将多个线性变换按层次叠加，由于线性变换的结果仍为线性变换，所以最终的结果等价于线性变换，与单个感知机并无区别，反而加大了模型，浪费了资源。
为了防止这个问题，需要对每个单元（感知机）的输出通过激活函数进行处理再交由下一层的感知机进行运算，这些激活函数就是解决非线性问题的关键。

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。主要的激活函数有：

1. Sigmoid 激活函数

Sigmoid 激活函数将输入投影到 $(0, 1)$ ，是一个软的 $\sigma(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$ 。

目前，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。

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}.$$

导数为下面的公式：

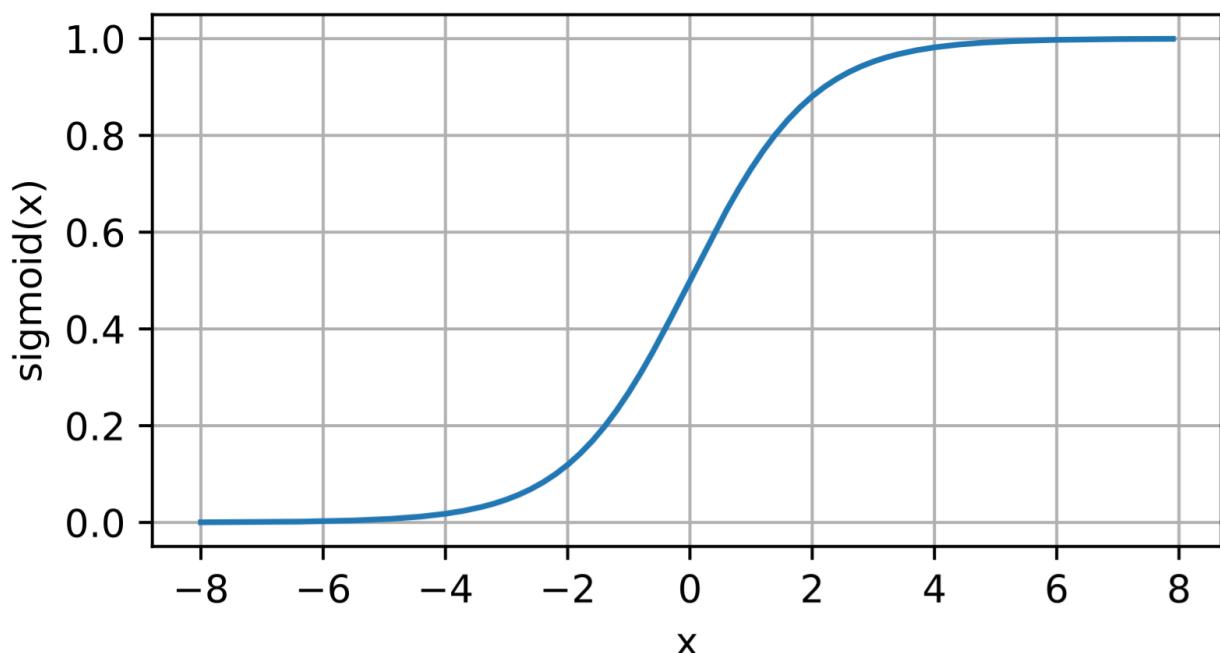
$$\frac{d}{dx} \text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)(1 - \text{sigmoid}(x)).$$

🔗 Sigmoid 函数

绘制sigmoid函数。

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

Results:



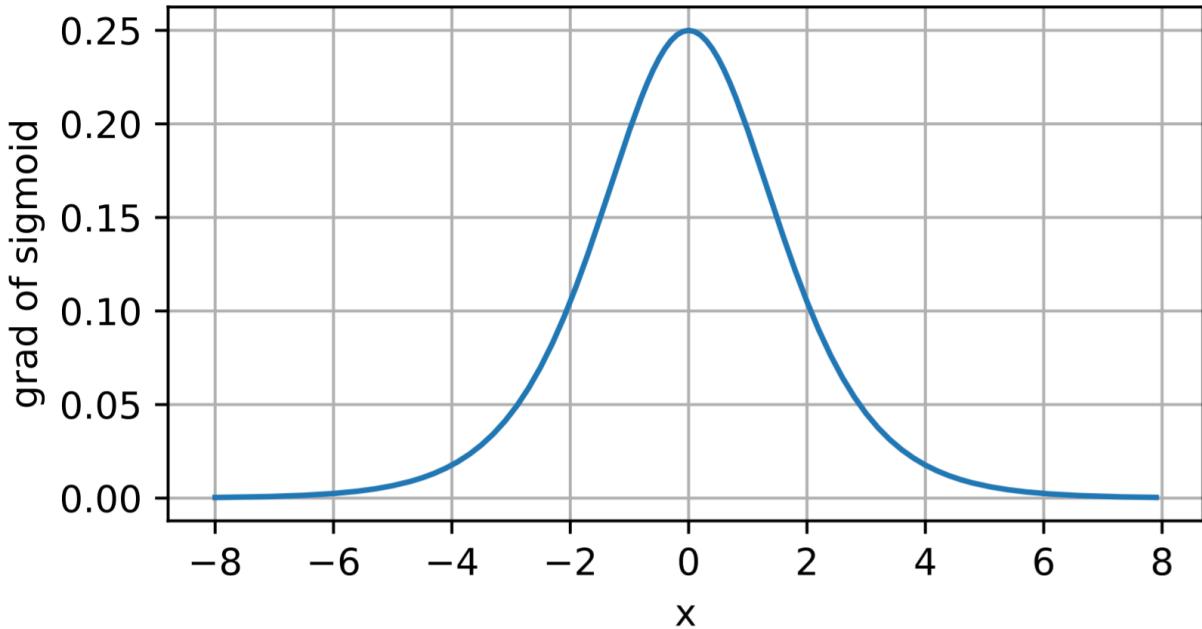
Note: 当输入接近0时，sigmoid函数接近线性变换。

- sigmoid函数的导数：

```
# 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))
```

language=python

Results:



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。

2. 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：

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

tanh函数的导数是：

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

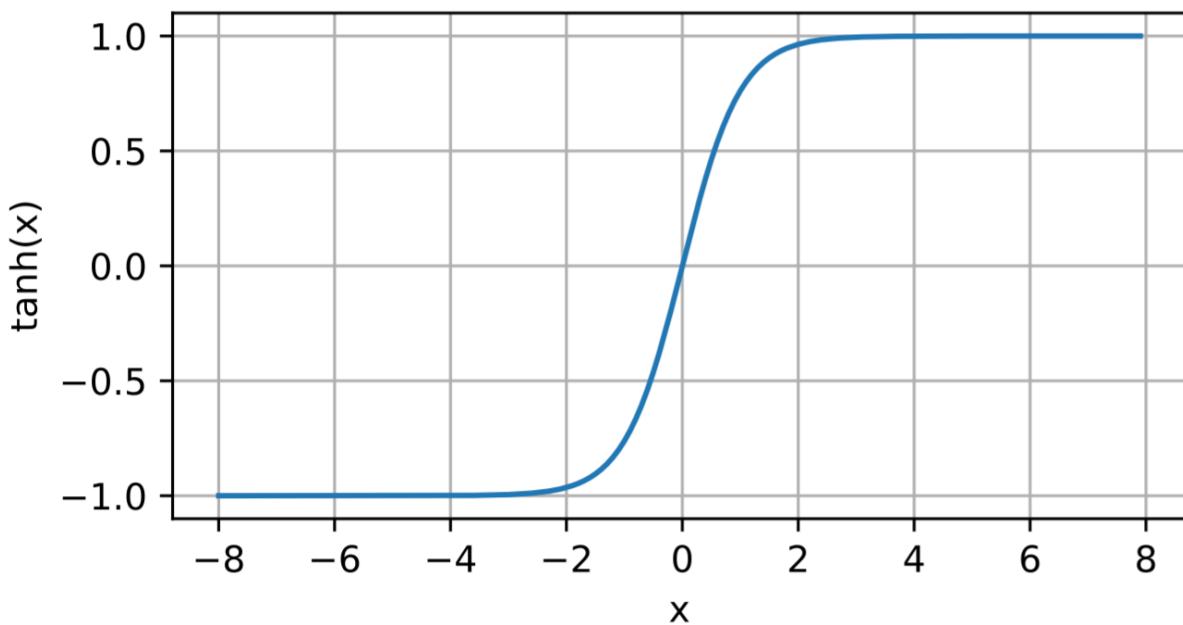
🔗 Tanh 函数

为了直观感受一下，可以画出函数的曲线图。正如从图中所看到，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))
```

language=python

Results:

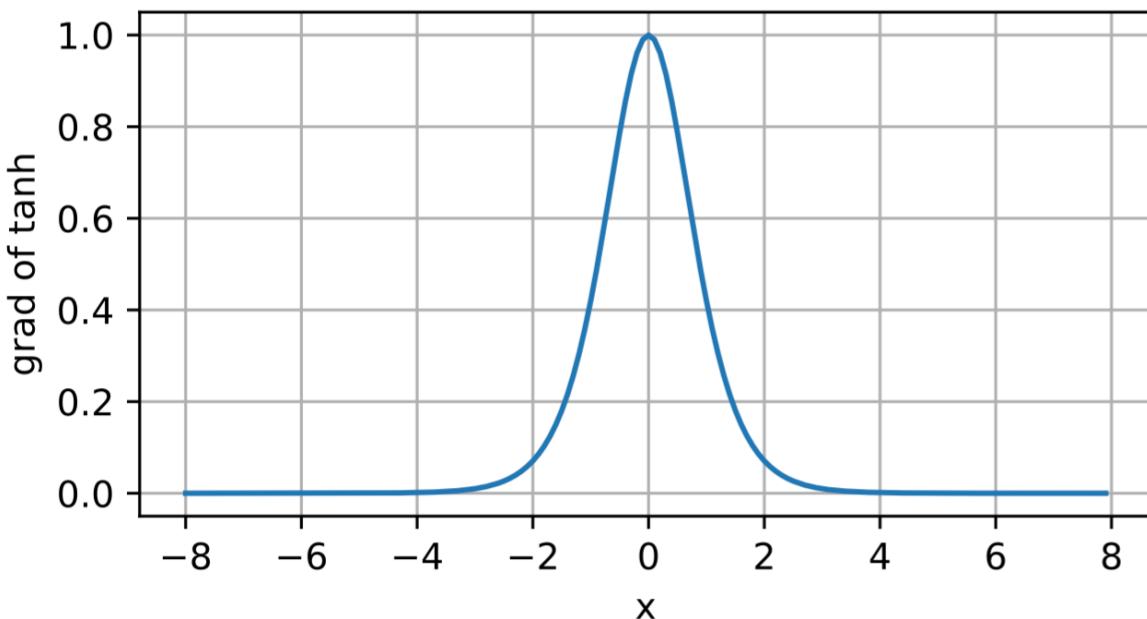


- Tanh 函数的导数

```
# 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))
```

language=python

Results:



3. 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。

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。这使得优化表现的更好，并且ReLU减轻了困扰以往神经网络的梯度消失问题

给定元素 x ，**ReLU** 函数被定义为该元素与 0 的最大值：

$$\text{ReLU}(x) = \max(x, 0)$$

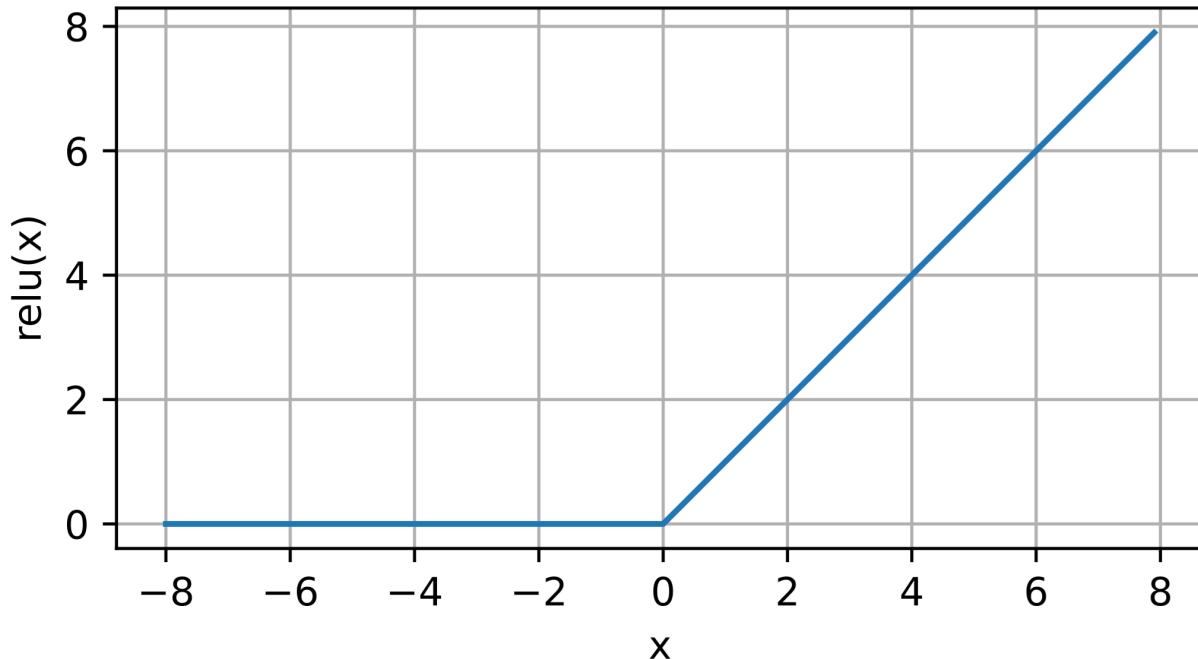
🔗 ReLU函数

为了直观感受一下，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。

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

language-python

Results:



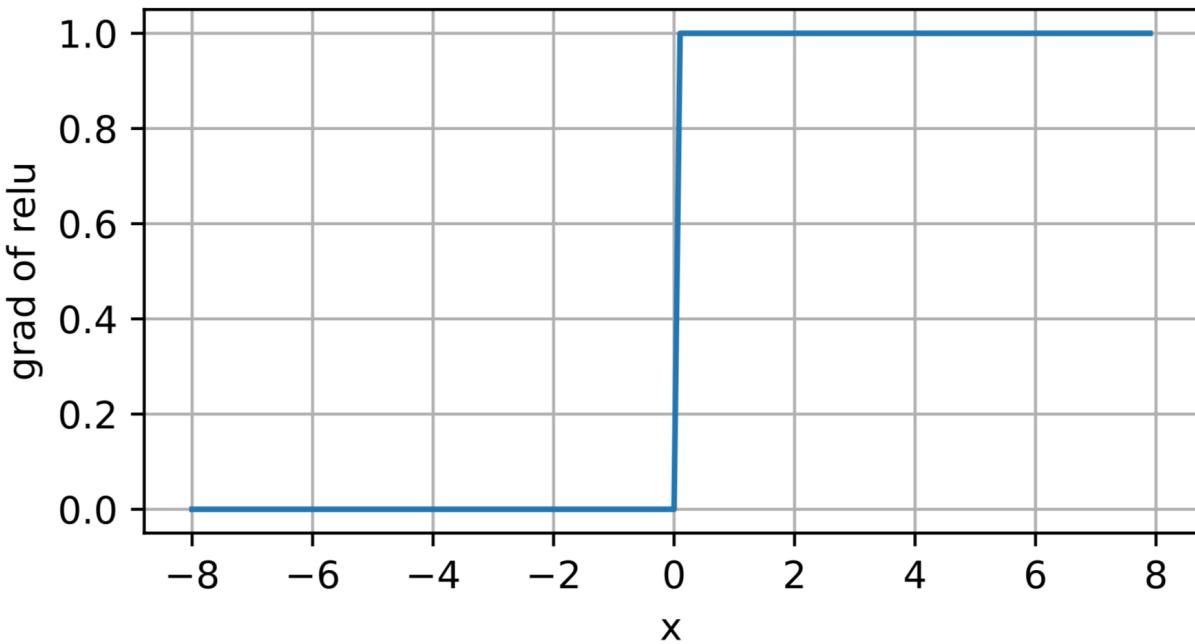
Note:当输入为负时，ReLU函数的导数为0，而当输入为正时，ReLU函数的导数为1。注意，当输入值精确等于0时，ReLU函数不可导。

- 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))
```

language-python

Results:

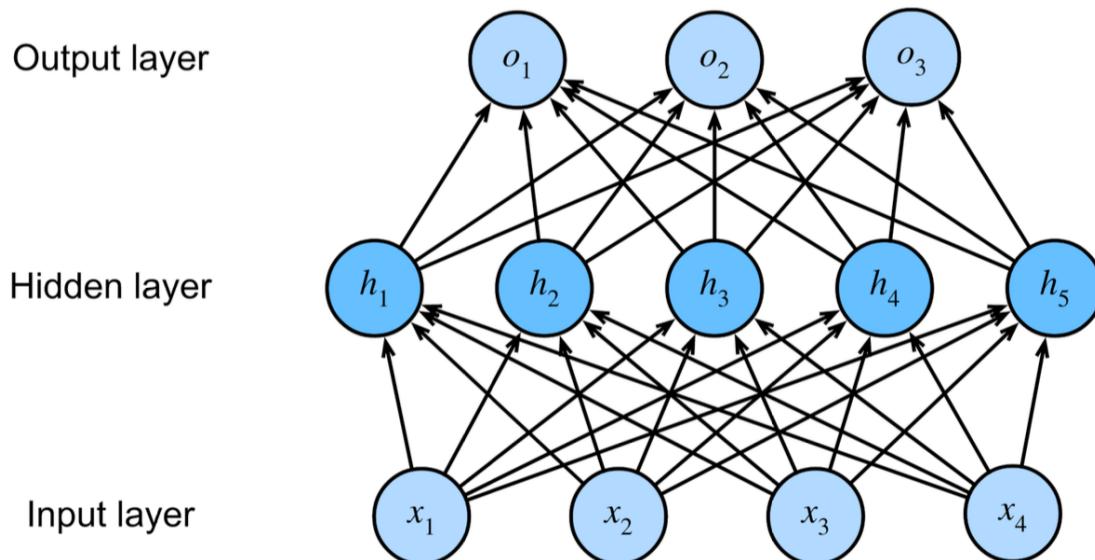


tanh函数的导数图像如下所示。当输入接近0时，tanh函数的导数接近最大值1。与我们在sigmoid函数图像中看到的类似，输入在任一方向上越远离0点，导数越接近0。

2.3.2.4 多类分类

[13:59 多分类](#)

$$y_1, y_2, \dots, y_k = \text{softmax}(o_1, o_2, \dots, o_k)$$



- 输入: $\mathbf{x} \in \mathbb{R}^n$

- 隐藏层:

$$\mathbf{h} = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

其中, $\mathbf{W}_1 \in \mathbb{R}^{m \times n}$, $\mathbf{b}_1 \in \mathbb{R}^m$, σ 是按元素的激活函数。

- **输出层:**

$$o = \mathbf{w}_2^T \mathbf{h} + b_2$$

$$y = \text{softmax}(\mathbf{o})$$

其中, $\mathbf{w}_2 \in \mathbb{R}^{m \times k}$, $b_2 \in \mathbb{R}^k$

2.3.2.5 多隐藏层

[16:11 多隐藏层](#)

多隐藏层

$$\mathbf{h}_1 = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

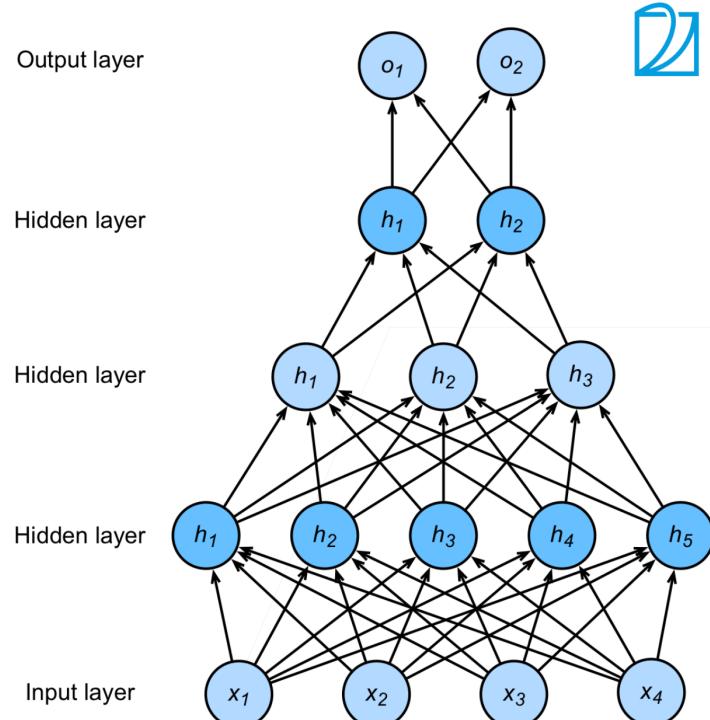
$$\mathbf{h}_2 = \sigma(\mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2)$$

$$\mathbf{h}_3 = \sigma(\mathbf{W}_3 \mathbf{h}_2 + \mathbf{b}_3)$$

$$\mathbf{o} = \mathbf{W}_4 \mathbf{h}_3 + \mathbf{b}_4$$

超参数

- 隐藏层数
- 每层隐藏层的大小



动手学深度学习 v2 • <https://courses.d2l.ai/zh-v2>

$$\mathbf{h}_1 = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{h}_2 = \sigma(\mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2)$$

$$\mathbf{h}_3 = \sigma(\mathbf{W}_3 \mathbf{h}_2 + \mathbf{b}_3)$$

$$\mathbf{o} = \mathbf{W}_4 \mathbf{h}_3 + \mathbf{b}_4$$

超参数:

- 隐藏层数
- 每层隐藏层大小

2.3.2.6 练习

1. 证明一个仅使用ReLU (或pReLU) 的多层感知机构造了一个连续的分段线性函数。[00:08](#)

绘制出ReLU的图像后, 我们可以发现, 输出值在经过下一层隐藏层的计算后, 如果结果小于等于0, 则这个数据被舍弃, 结果大于0则被保留, 类似一个筛选的过程。相当于上一层的输出经过线性变换后在下一层被筛选, 线性变换和上述筛选的过程都是连续的, 因此就会产生连续而且分段的结果。

2. 构建多个超参数的搜索方法。

有四种主要的策略可用于搜索最佳配置。

- 试错
- 网格搜索
- 随机搜索
- 贝叶斯优化

详见 [超参数搜索不够高效？这几大策略了解一下](#)

3. 权重初始化方法

1. 全零初始化：在神经网络中，把w初始化为0是不可以的。这是因为如果把w初始化0，那么每一层的神经元学到的东西都是一样的（输出是一样的），而且在BP的时候，每一层内的神经元也是相同的，因为他们的gradient相同，weight update也相同。
2. 随机初始化
3. Xavier初始化：保持输入和输出的方差一致（服从相同的分布），这样就避免了所有输出值都趋向于0。
4. He initialization：在ReLU网络中，假定每一层有一半的神经元被激活，另一半为0（x负半轴中是不激活的），所以要保持variance不变，只需要在Xavier的基础上再除以2。
5. pre-training

详见 [权重/参数初始化](#)

4. 超参数的调节

- 在mlp中，第一个隐藏的的单元数可能大于输入的个数，每个隐藏层中的单元数由前至后递减，逐渐接近输出的个数。
 -
- 多数情况下，将mlp的深度设置得较深，而每层的单元数相对较少，这样易于训练，不易过拟合，也利于逐步学习样本特征。
 -
- 激活函数种类的选择对训练的影响小于其余的因素。

2.3.2.7 总结

Note

- 多层感知机使用隐藏层和激活函数来得到非线性模型
- 常用激活函数：Sigmoid, Tanh, ReLU
- 使用softmax进行多分类
- 隐藏层数、大小为超参数

2.3.3 从零实现多层感知机

Note

在 [Softmax 简洁实现](#) 中，笔者在尝试运行李沐老师的代码时会报错，此处，对其进行修正。

通过查询发现，笔者安装的 `d2l` 是最新版本：

```
d2l==1.0.0a0
```

language=python

发现这个版本中并没有 `d2l.train_ch3` 函数，因此，考虑将版本进行倒退：

```
pip install d2l==0.17.5
```

language=shell

然而，此时 `import d2l` 包时会报 [环境配置](#) 中出现的错误：

⚠ Bug

```
!pip install d2l
import numpy as np
import torch
from torch.utils import data
from d2l import torch as d2l
```

language=python

会报如下错误：

```
ValueError Traceback (most recent call last)
Input In [1], in <cell line: 4>()
1 #import torch
2 #print(torch.__version__)
----> 4 from d2l import torch as d2l
      ...
      ...
----> 13 from pandas._libs.interval import Interval
14 from pandas._libs.tslibs import (
15     NaT,
16     NaTType,
17     ...
21     iNaT,
22 )

File pandas/_libs/interval.pyx 1, in init pandas._libs.interval()

ValueError: numpy ndarray size changed, may indicate binary incompatibility. Expected
96 from C header, got 88 from PyObject
```

此时只需要重装 `pandas` 就可以完美解决了。

```
!pip install --upgrade numpy
!pip install --force-reinstall pandas
```

language=shell

2.3.3.1 读取数据

[00:03 从零开始实现多层感知机](#)

我们已经在 [上一节](#) 中描述了多层感知机（MLP），接下来让我们尝试自己实现一个多层感知机。

为了与之前 softmax 回归 获得的结果进行比较，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)
```

language=python

2.3.3.2 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。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]
```

language=python

2.3.3.3 模型设定

实现 relu 激活函数：

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

language=python

构建模型：

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)
```

language=python

损失函数：

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

language=python

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（如果运行报错，提示 d2l.train_ch3 不存在，请查看 [本章前沿部分](#) 的解决方案）。

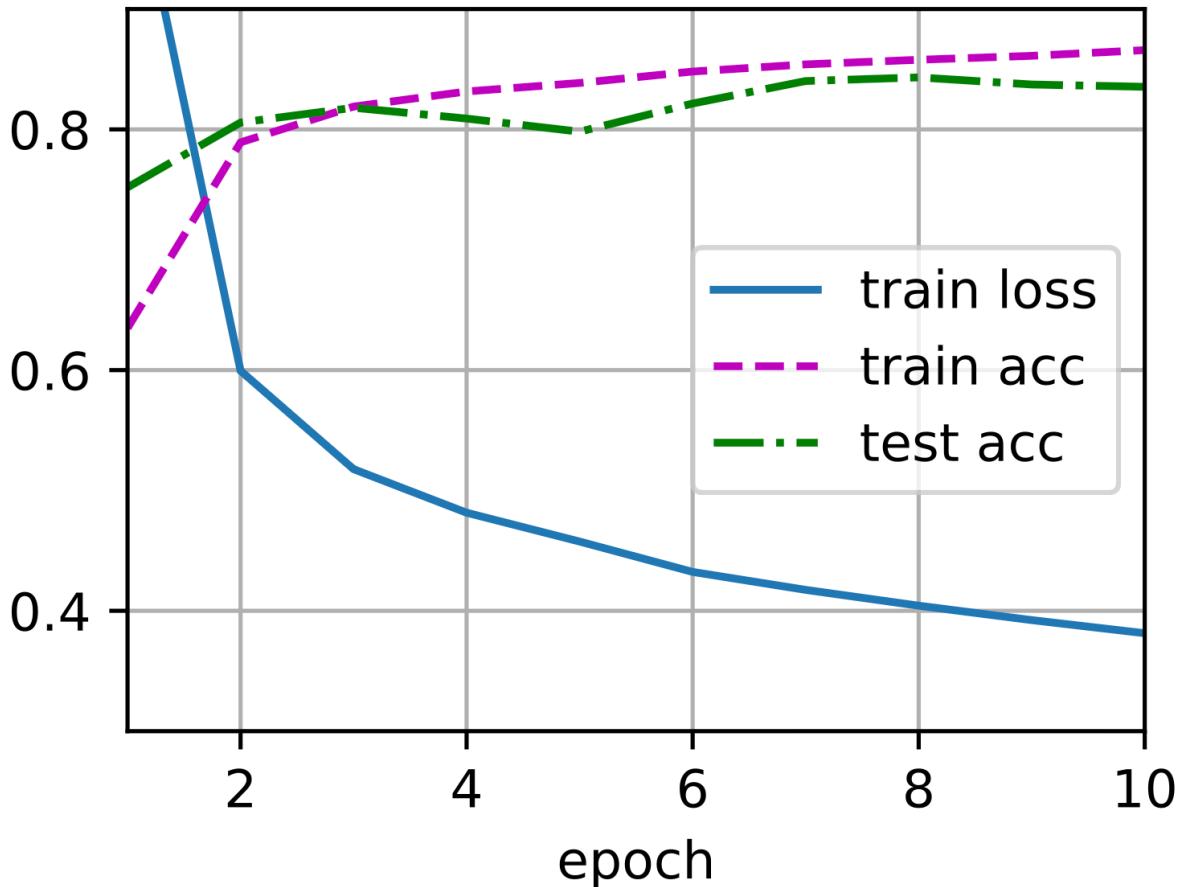
将迭代周期数设置为 10，并将学习率设置为 0.1：

Code

```
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)
```

language-python

Results:



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。

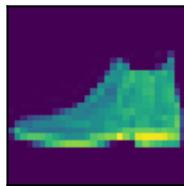
Code

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

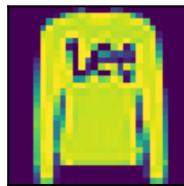
language-python

Results:

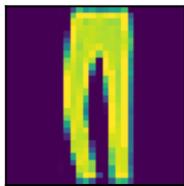
ankle boot
ankle boot



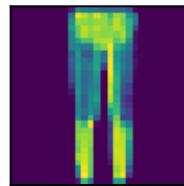
pullover
pullover



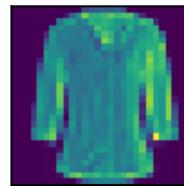
trouser
trouser



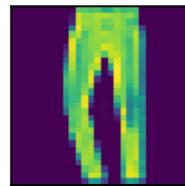
trouser
trouser



shirt
shirt



trouser
trouser



2.3.3.4 小结

Note

手动实现一个简单的多层感知机是很容易的。然而如果有大量的层，从零开始实现多层感知机会变得很麻烦（例如，要命名和记录模型的参数）。

2.3.4 多层感知机Q&A

[00:08 Q&A](#)