```
%matplotlib inline
import random
import torch
from d2l import torch as d2l
# 根据带有噪音的线性模型构建一个人造数据集，
#我们使用线性模型参数w=[2,-3.4]^T b=4.2 和噪音项c 生成数据集及其标签

def synthetic_data(w, b, num_examples):
    """生成y=Xw+b+噪音。"""
    # 均值方程
    X = torch.normal(0, 1, (num_examples, len(w)))
    y = torch.matmul(X, w) + b
    y += torch.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))


true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
d2l.set_figsize()
d2l.plt.scatter(features[:, 1].detach().numpy(), labels.detach().numpy(), 1)
```

```
# 定义一个data_iter函数，该函数接收批量大小，特征矩阵和标签向量作为输入，生成大小为batch_size
的小批量

def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = torch.tensor(indices[i:min(i + batch_size,
num_examples)])
        # print(batch_indices)
        yield features[batch_indices], labels[batch_indices]


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

# 定义初始化模型参数
w = torch.normal(0, 0.01, size=(2, 1), requires_grad=True)

b = torch.zeros(1, requires_grad=True)


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


# 定义损失函数 loss
```

```python
def squared_loss(y_hat, y):
    """均方误差"""
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2


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


lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        l = loss(net(X, w, b), y)  #`x`和`y`的小批量损失
        # 因为`l`的形状是`(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_1 = loss(net(features, w, b), labels)
        print(f"epoch:{epoch + 1},loss:{float(train_1.mean()):f}")
        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}")
```

```
tensor([[ 1.2500,  1.3090],
        [ 1.1490, -0.5451],
        [ 0.7693, -2.0294],
        [ 0.2937, -1.2719],
        [ 3.1034, -0.3622],
        [ 0.1710,  0.5130],
        [-2.3583, -1.0385],
        [ 2.0013,  1.2128],
        [-1.1262,  1.1793],
        [-1.9892, -0.2712]])
 tensor([[ 2.2365],
        [ 8.3491],
        [12.6233],
        [ 9.1010],
        [11.6282],
        [ 2.8082],
        [ 3.0157],
        [ 4.0825],
        [-2.0717],
        [ 1.1508]])
epoch:1,loss:0.027258
w的估计误差:tensor([ 0.0548, -0.1262])
b的估计误差:tensor([0.1812])
epoch:2,loss:0.000097
```

```
w的估计误差:tensor([ 0.0015, -0.0044])
b的估计误差:tensor([0.0080])
epoch:3,loss:0.000052
w的估计误差:tensor([ 0.0003, -0.0002])
b的估计误差:tensor([-7.1526e-05])
```

```python
# 通过使用深度学习框架来简介地实现线性回归模型生成数据集

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

true_w = torch.tensor([2, -3.4])
ture_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1500)


def load_array(data_arrays, batch_size, is_train=True):
    """构建一个Pytorch数据迭代器"""
    dataset = data.TensorDataset(*data_arrays)
    return data.DataLoader(dataset, batch_size, shuffle=is_train)


batch_size = 10
data_iter = load_array((features, labels), batch_size)
# next(iter(data_iter))
# 使用框架的预定义好的层
# `nn`是神经网络的缩写
from torch import nn

net = nn.Sequential(nn.Linear(2, 1))
# 初始化模型参数(初始化权重和偏差)
net[0].weight.data.normal_(0, 0.01)
net[0].bias.data.fill_(0)
# 计算均方误差使用的是MSELoss类 ,也称为平方l2范数
loss = nn.MSELoss()
# 实例化SGD实例
trainer = torch.optim.SGD(net.parameters(), lr=0.03)
# 训练
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        l = loss(net(X), y)
        trainer.zero_grad()
        l.backward()
        trainer.step()
    l = loss(net(features), labels)
    print(f"epoch:{epoch + 1},loss:{l:f}")
```

```
epoch:1,loss:0.000103
epoch:2,loss:0.000102
epoch:3,loss:0.000102
```

逻辑回归(Logistic Regression)

Logistic Regression 虽然被称为回归，但其实际上是分类模型，并常用于二分类。Logistic Regression 因其简单、可并行化、可解释强深受工业界喜爱。

Logistic 回归的本质是：假设数据服从这个分布，然后使用极大似然估计做参数的估计。

$logistic$分布是一种连续型的概率分布，其中分布函数和密度函数分别是：

$$F(x) = P(X <= x) = \frac{1}{1 + e^{-(x-u)/\gamma}}$$

$$F(x) = F^{'}(X <= x) = \frac{e^{-(x-u)/\gamma}}{\gamma(1 + e^{-(x-u)/\gamma})^2}$$

其中，u表示位置参数$\gamma > 0$为形状参数。

$Logistic$分布是由其位置和尺度参数定义的连续分布。$Logistic$分布的形状与正态分布的形状相似，

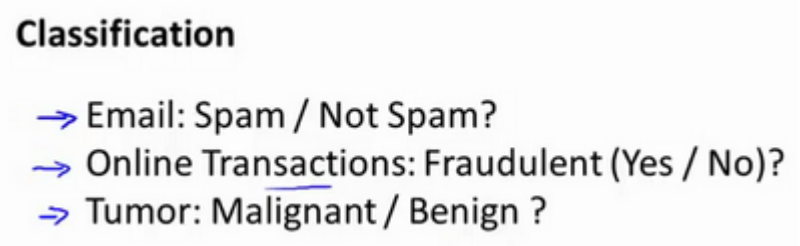
但是$Logistic$分布的尾部更长，所以我们可以使用$Logistic$分布来建模比正态分布具有更长尾部和更高波峰的数据分布。在深度学习中常用到的$Sigmoid$函数就是$Logistic$的分布函数在$u = 0, \gamma = 1$的特殊形式。

1 分类问题

在分类问题中，你要预测的变量y是离散的值，我们讲学习一种叫做

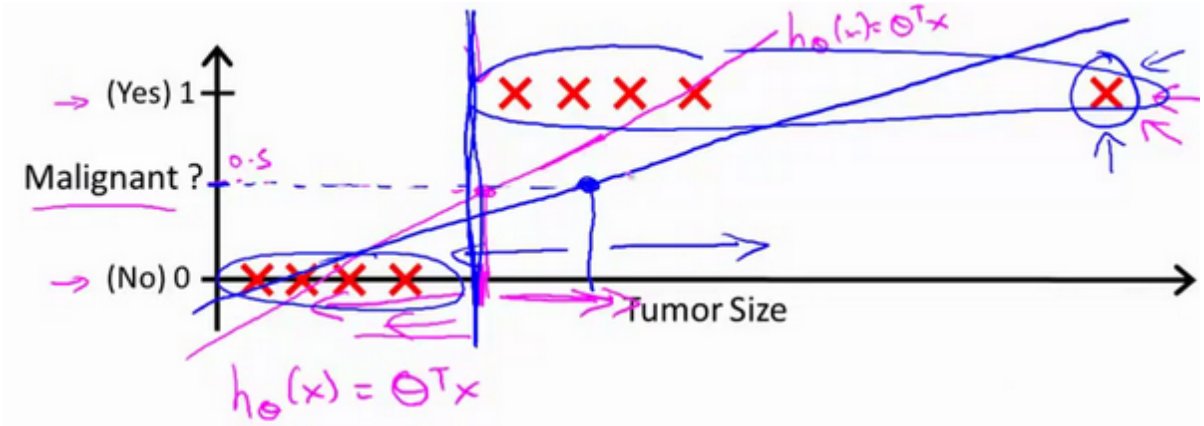
逻辑回归$(LogisticRegression)$的算法，这是目前最流行使用最广泛的一种学习算法。

在分类问题中，我们尝试预测的是结果是否属于某一个类（例如正确或错误）。分类问题的例子有：判断一封电子邮件是否是垃圾邮件；判断一次金融交易是否是欺诈；之前我们也谈到了肿瘤分类问题的例子，区别一个肿瘤是恶性的还是良性的。



二元分类问题：

我们将因变量$(dependentvariable)$可能属于的两个类分别称为负向类（$negativeclass$）和正向类（$positiveclass$），则因变量$y \in 0,1$，其中0表示负向类，1表示正向类。



→ Threshold classifier output $h_\theta(x)$ at 0.5:

→ If $h_\theta(x) \geq 0.5$, predict "y = 1"

If $h_\theta(x) < 0.5$, predict "y = 0"

Classification: y = 0 or 1

$h_\theta(x)$ can be > 1 or < 0

Logistic Regression: $0 \leq h_\theta(x) \leq 1$

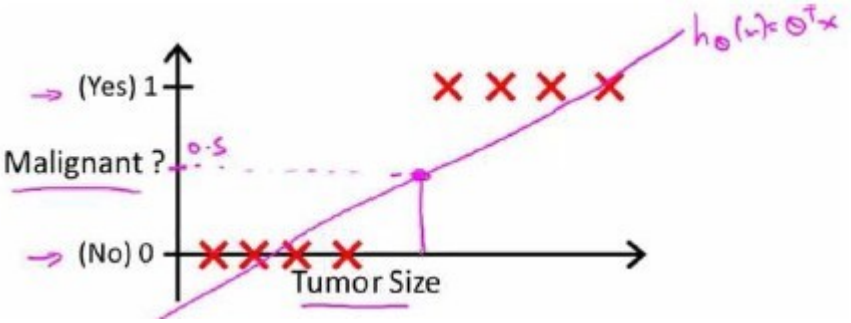
如果我们要用线性回归算法来解决一个分类问题，对于分类，y取值为0或1，但如果你使用的是线性回归,那么假设函数的输出值可能大于1，或者远小于1，即使所有训练样本的标签y都等于0或1。尽管我们知道标签应该取值0 或者1，但是如果算法得到的值远大于1或者远小于0的话，就会感觉很奇怪。所以\我们在接下来的要研究的算法就叫做逻辑回归算法，这个算法的性质是：它的输出值永远在0到 1 之间。

逻辑回归算法是分类算法，我们将它作为分类算法使用。有时候可能因为这个算法的名字中出现了“回归”使你感到困惑，但逻辑回归算法实际\上是一种分类算法，它适用于标签 取值离散的情况，如：1 0 0 1。

2 假说表示

假设函数的表达式，在分类问题中，要用什么样的函数来表示我们的假设。此前我们说过，希望我们的分类器的输出值在0和1之间，因此，我们希望想出一个满足某个性质的假设函数，这个性质是它的预测值要在0和1之间。

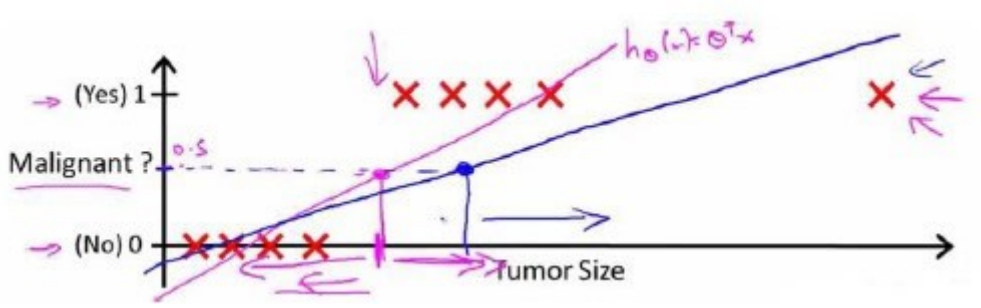
回顾在一开始提到的乳腺癌分类问题，我们可以用线性回归的方法求出适合数据的一条直线：



根据线性回归模型我们只能预测连续的值，然而对于分类问题，我们需要输出0或1，我们可以预测：

$$当h_\theta(x) >= 0.5时，预测y = 1$$
$$当h_\theta(x) < 0.5时，预测y = 0$$

对于上图所示的数据，这样的一个线性模型似乎能很好地完成分类任务。假使我们又观测到一个非常大尺寸的恶性肿瘤，将其作为实例加入到我们的训练集中来，这将使得我们获得一条新的直线。



这时，再使用0.5作为阀值来预测肿瘤是良性还是恶性便不合适了。可以看出，线性回归模型，因为其预测的值可以超越[0,1]的范围，并不适合解决这样的问题。

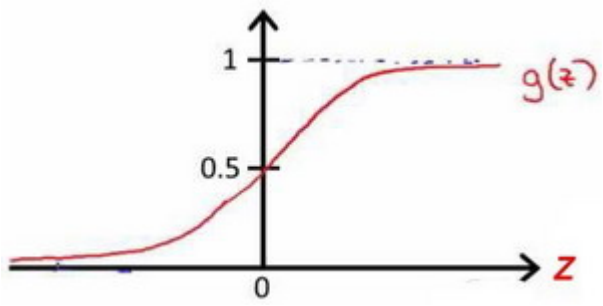
2.1 sigmoid 函数

$$这里我们现在引入一个新的模型。逻辑回归，该模型的输出变量范围始终在0-1之间，$$
$$逻辑回归的假设是：h_\theta(x) = g(\theta^T X)，其中：X：代表特征向量,g代表逻辑函数$$
$$（logistic function）是一个常用的逻辑函数为s形函数（sigmoid function），$$
$$公式为：g(z) = \frac{1}{1+e^{-z}}$$

python代码：

```
import numpy as np
def sigmoid(z):
   return 1 / (1 + np.exp(-z))
```

sigmoid 函数图像



$h_\theta(x)$的作用是，对于给定的输入变量，根据选择的参数计算输出变量=1的可能性（**estimated probablity**）即$h_\theta(x) = P(y=1|x;\theta)$ 例如，如果对于给定的x，通过已经确定的参数计算得出$h_\theta(x) = 0.7$，则表示有70%的几率y为正向类，相应地y为负向类的几率为1-0.7=0.3。

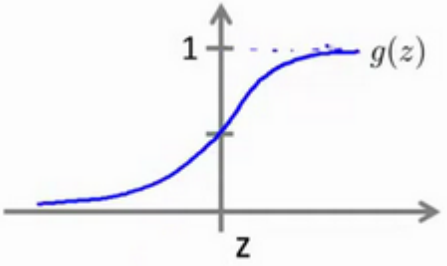
3 判定边界

在讲下决策边界(**decision boundary**)的概念。这个概念能更好地帮助我们理解逻辑回归的假设函数在计算什么。

Logistic regression

$$h_\theta(x) = g(\theta^T x)$$

$$g(z) = \frac{1}{1+e^{-z}}$$



在逻辑回归中，我们预测：

当$h_\theta(x) >= 0.5$时，预测$y = 1$; 当$h_\theta(x) < 0.5$时，预测$y = 0$

根据上面绘制出的s形函数图像，我们知道：

$z = 0$时$g(z) = 0.5$

$z > 0$时$g(z) > 0.5$

$z < 0$时$g(z) < 0.5$

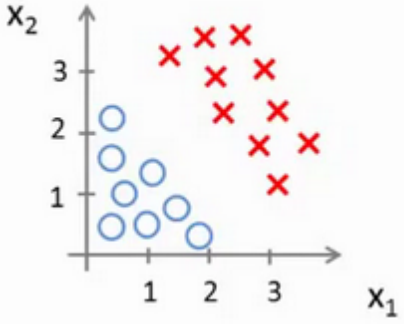
又$z = \theta^T x$

即：$\theta^T x >= 0$时，预测$y = 1$

$\theta^T x < 0$时，预测$y = 0$

现在假设我们有一个模型：

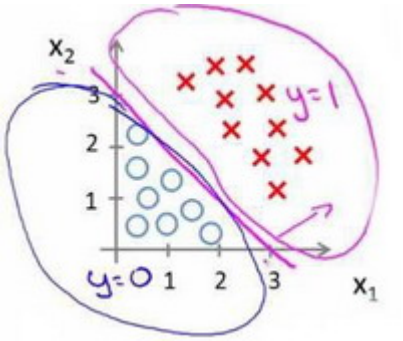
Decision Boundary



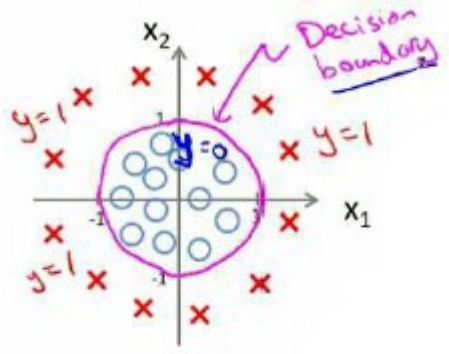
$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

-3

并且参数θ是向量$[-3, 1, 1]$, 则当$-3 + x_1 + x_2 >= 0$时，即$x_1 + x_2 >= 3$时，模型加个预测$y = 1$，我们可以绘制直线$x_1 + x_2 = 3$, 这条直线便是我们模型的分割线，将预测为1的区域和预测为01的区域分隔开。



假使我们的数据呈现这样的分布情况，怎样的模型才能适合呢?



因为需要使用曲线才能分割$y=0$的区域和$y=1$的区域，我们需要二次方特征：$h_0(x)=g(\theta_0+\theta_1x_1+\theta_2x_2+\theta_3x_1^2+\theta_4x_2^2)$是$[-1,0,0,1,1]$,则我们得到的判定边界恰好是圆点在原点且半径为$1$的圆形,我们可以用非常复杂的模型来适应非常复杂形状的判定边界。

4 代价函数

如何拟合逻辑函数回归模型的参数{\theta},具体来说，我要定义用来拟合参数的优化目标或者叫做代价函数，这便是监督学习问题中的逻辑回归模型的拟合问题。

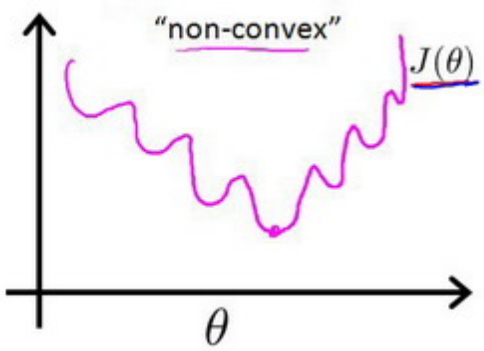
Training set: $\{(x^{(1)},y^{(1)}),(x^{(2)},y^{(2)}),\cdots,(x^{(m)},y^{(m)})\}$

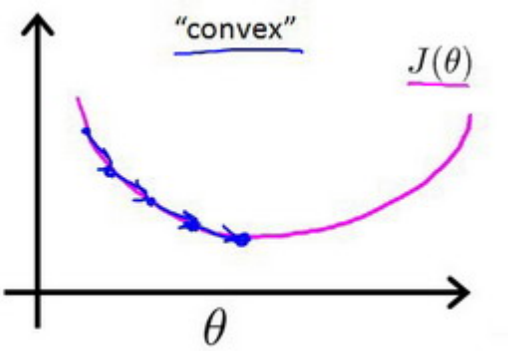
m examples $\quad x\in\begin{bmatrix}x_0\\x_1\\\cdots\\x_n\end{bmatrix}\quad x_0=1,y\in\{0,1\}$

$$h_\theta(x)=\frac{1}{1+e^{-\theta^Tx}}$$

How to choose parameters θ ?

对于线性回归模型，我们定义的代价函数是所有模型误差的平方和。理论上来说，我们也可以对逻辑回归模型沿用这个定义，但是问题在于，当我们将$h_\theta(x)=\frac{1}{1+e^{-\theta^Tx}}$带入到这样定义了的代价函数中时，我们可以得到的代价函数将是一个非凸函数（**non-convexfunction**）





这意味着我们的代价函数有许多局部最小值，这将影响梯度下降算法寻找全局最小值。

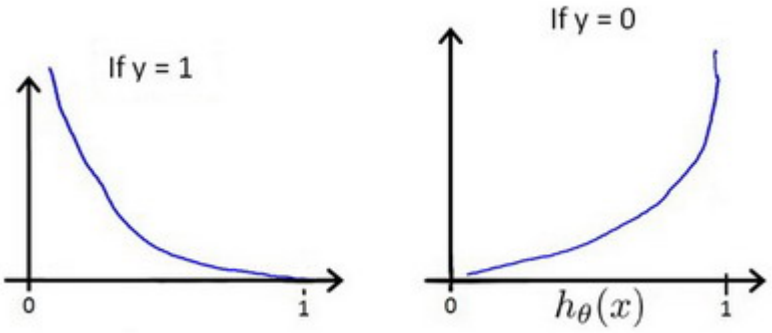
线性回归的代价函数为：$J\left(\theta\right)=\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)^2$。我们定义逻辑回归的代价函数为：

$$J\left(\theta\right)=\frac{1}{m}\sum_{i=1}^{m}Cost\left(h_\theta\left(x^{(i)}\right),y^{(i)}\right)$$

其中

$$\text{Cost}(h_\theta(x),y)=\begin{cases}-\log(h_\theta(x)) & \text{if } y=1\\ -\log(1-h_\theta(x)) & \text{if } y=0\end{cases}$$

$h_\theta\left(x\right)$与21$Cost\left(h_\theta\left(x\right),y\right)$之间的关系如下图所示：



线性回归

回归（regression）是能为一个或多个自变量与因变量之间关系建模的一类方法。 在自然科学和社会科学领域，回归经常用来表示输入和输出之间的关系。

在机器学习领域中的大多数任务通常都与预测（prediction）有关。 当我们想预测一个数值时，就会涉及到回归问题。 常见的例子包括：预测价格（房屋、股票等）、预测住院时间（针对住院病人等）、 预测需求（零售销量等）。 但不是所有的预测都是回归问题。 在后面的章节中，我们将介绍分类问题。分类问题的目标是预测数据属于一组类别中的哪一个。

1.线性模型

线性假设是指目标（房屋价格）可以表示为特征（面积和房龄）的加权和，如下面的式子：

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b. \quad (3.1.1)$$

warea和wage 称为权重（weight），权重决定了每个特征对我们预测值的影响。b称为偏置（bias）、 偏移量（offset）或截距（intercept）。

偏置是指当所有特征都取值为0时，预测值应该为多少。即使现实中不 会有任何房子的面积是0或房龄正好是0年，我们仍然需要偏置项。如果没有偏置项，我们模型的表达能力将 受到限制。严格来说，(3.1.1)是输入特征的一个仿射变换（affinetransformation）,仿射变换的特点是通过 加权和对特征进行线性变换（lineartransformation），并通过偏置项来进行平移（translation）。

当我们的输入 包含d个特征时，我们将预测结果ˆ y（通常使用"尖角"符号表示y的估计值）表示为：

$$\hat{y} = w_1 x_1 + ... + w_d x_d + b.$$

将所有特征放到向量x∈Rd中，并将所有权重放到向量w∈Rd中，我们可以用点积形式来简洁地表达模型： $\hat{y} = \mathbf{w}^\top \mathbf{x} + b.$

向量x对应于单个数据样本的特征。用符号表示的矩阵X∈Rn×d可以很方便地引用我们整个数 据集的n个样本。其中，X的每一行是一个样本，每一列是一种特征。 对于特征集合X，预测值ˆ y∈Rn可以通过矩阵-向量乘法表示为： $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$

即使确信特征与标签的潜在关系是线性的，我们也会加入一个噪声项来考虑观测误差带 来的影响。在开始寻找最好的模型参数（modelparameters）w和b之前，我们还需要两个东西：

（1）一种模型质量的度 量方式；

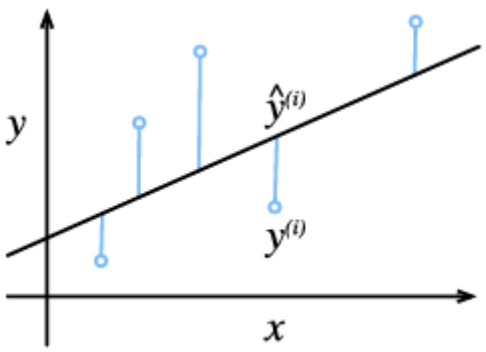
（2）一种能够更新模型以提高模型预测质量的方法。

2. 损失函数

损失函数（lossfunction） 能够量化目标的实际值与预测值之间的差距。通常我们会选择非负数作为损失，且数值越小表示损失越小， 完美预测时的损失为0。回归问题中最常用的损失函数是平方误差函数。当样本i的预测值为ˆ y(i)，其相应的真 实标签为y(i)时，平方误差可以定义为以下公式：

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2.$$

常数1/2不会带来本质的差别，但这样在形式上稍微简单一些（因为当我们对损失函数求导后常数系数为1）。由 于训练数据集并不受我们控制，所以经验误差只是关于模型参数的函数。为了进一步说明，来看下面的例子。 我们为一维情况下的回归问题绘制图像，如图所示:



由于平方误差函数中的二次方项，估计值ˆ y(i)和观测值y(i)之间较大的差异将导致更大的损失。为了度量模型 在整个数据集上的质量，我们需计算在训练集n个样本上的损失均值（也等价于求和）：

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2.$$

在训练模型时，我们希望寻找一组参数（w∗,b∗），这组参数能最小化在所有训练样本上的总损失。

如下式：$$\mathbf{w}^*, b^* = \operatorname*{argmin}_{\mathbf{w}, b}\ L(\mathbf{w}, b).$$

3. 解析解

线性回归刚好是一个很简单的优化问题。与其他大部分模型不同，线性回归的解可以用一个公式简单地表达出来，这类解叫作**解析解（analyticalsolution）**。首先，我们将偏置b合并到参数w中， 合并方法是在包含所有参数的矩阵中附加一列。我们的预测问题是最小||y−X||^2。这在损失平面上只有 一个临界点，这个临界点对应于整个区域的损失极小点。将损失关于w的导数设为0，得到解析解：

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}.$$

像线性回归这样的简单问题存在解析解，但并不是所有的问题都存在解析解。解析解可以进行很好的数学分 析，但解析解对问题的限制很严格，导致它无法广泛应用在深度学习里。

4. 随机梯度下降

即使在我们无法得到解析解的情况下，我们仍然可以有效地训练模型。在许多任务上，那些难以优化的模型 效果要更好。因此，弄清楚如何训练这些难以优化的模型是非常重要的。

这里我们用到一种名为梯度下降（gradientdescent）的方法，这种方法几乎可以优化所有深度学习模型。 它通过不断地在损失函数递减的方向上更新参数来降低误差。 梯度下降最简单的用法是计算损失函数（数据集中所有样本的损失均值）关于模型参数的导数（在这里也可 以称为梯度）。但实际中的执行可能会非常慢：因为在每一次更新参数之前，我们必须遍历整个数据集。因此，我们通常会在每次需要计算更新的时候随机抽取一小批样本，这种变体叫做小批量随机梯度下降（minibatch stochastic gradient descent）。

在每次迭代中，我们首先**随机抽样一个小批量B，它是由固定数量的训练样本组成的。然后，我们计算小批量的平均损失关于模型参数的导数（也可以称为梯度）。最后，我们将梯度乘以一个预先确定的正数η，并从 当前参数的值中减掉。** 我们用下面的数学公式来表示这一更新过程（∂表示偏导数）

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b).$$

总结一下，算法的步骤如下：

（1）初始化模型参数的值，如随机初始化；

（2）从数据集中随机抽取小批量样 本且在负梯度的方向上更新参数，并不断迭代这一步骤。对于平方损失和仿射变换，我们可以明确地写成如下形式:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \mathbf{x}^{(i)}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right),$$

$$b \leftarrow b - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \partial_{b} l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right).$$

公式中的w和x都是向量。在这里，更优雅的向量表示法比系数表示法（如w1,w2,...,wd）更具可读性。**|B|表示每个小批量中的样本数**，这也称为批量大小（batchsize）。**η表示学习率（learningrate）**，批量 大小和学习率的值通常是手动预先指定，而不是通过模型训练得到的。这些可以调整但不在训练过程中更新 的参数称为超参数（hyperparameter）。调参（hyperparametertuning）是选择超参数的过程。超参数通常是我们根据训练迭代结果来调整的，而训练迭代结果是在独立的验证数据集（validationdataset）上评估得到的。

在训练了预先确定的若干迭代次数后（或者直到满足某些其他停止条件后），我们记录下模型参数的估计值， 表示为ˆ w,ˆ b。但是，即使我们的函数确实是线性的且无噪声，这些估计值也不会使损失函数真正地达到最小 值。因为算法会使得损失向最小值缓慢收敛，但却不能在有限的步数内非常精确地达到最小值。 线性回归恰好是一个在整个域中只有一个最小值的学习问题。但是对像深度神经网络这样复杂的模型来说， 损失平面上通常包含多个最小值。深度学习实践者很少会去花费大力气寻找这样一组参数，使得在训练集上 的损失达到最小。事实上，更难做到的是找到一组参数，这组参数能够在我们从未见过的数据上实现较低的 损失，这一挑战被称为泛化（generalization）。

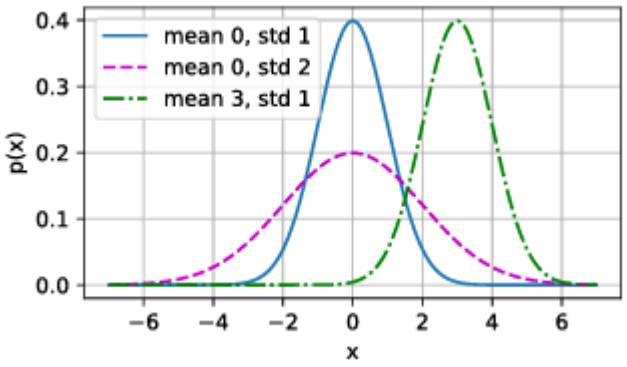
5. 正态分布与平方误差

正态分布和线性回归之间的关系很密切。正态分布（normaldistribution），也称为高斯分布（Gaussian distribution），最早由德国数学家高斯（Gauss）应用于天文学研究。简单的说，若随机变量x具有均值μ和方 差σ2（标准差σ），其正态分布概率密度函数如下：

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right).$$

```
def normal(x, mu, sigma):
 p = 1 / math.sqrt(2 * math.pi * sigma**2)
 return p * np.exp(-0.5 / sigma**2 * (x- mu)**2)

# 再次使用numpy进行可视化
x = np.arange(-7, 7, 0.01)
 # 均值和标准差对
params = [(0, 1), (0, 2), (3, 1)]
 d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
 ylabel='p(x)', figsize=(4.5, 2.5),
 legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



就像我们所看到的，改变均值会产生沿x轴的偏移，增加方差将会分散分布、降低其峰值。

均方误差损失函数（简称均方损失）可以用于线性回归的一个原因是：我们假设了观测中包含噪声，其中噪 声服从正态分布。噪声正态分布如下式： $y = \mathbf{w}^\top \mathbf{x} + b + \epsilon,$ 其中 $\epsilon \sim \mathcal{N}(0, \sigma^2)$。

因此，我们现在可以写出通过给定的x观测到特定y的似然（likelihood）：

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right).$$

现在，根据极大似然估计法，参数w和b的最优值是使整个数据集的似然最大的值：

$$P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}).$$

根据极大似然估计法选择的估计量称为极大似然估计量。虽然使许多指数函数的乘积最大化看起来很困难， 但是我们可以在不改变目标的前提下，通过最大化似然对数来简化。由于历史原因，优化通常是说最小化而 不是最大化。我们可以改为最小化负对数似然−logP(y|X)。由此可以得到的数学公式是：

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2.$$

现在我们只需要假设σ是某个固定常数就可以忽略第一项，因为第一项不依赖于w和b。现在第二项除了常 数1 σ2 外，其余部分和前面介绍的均方误差是一样的。上面式子的解并不依赖于σ。因此，在高斯噪声的假设下，最小化均方误差等价于对线性模型的极大似然估计。

6. 从线性回归到深度网络

到目前为止，我们只谈论了线性模型。 尽管神经网络涵盖了更多更为丰富的模型，我们依然可以用描述神经网络的方式来描述线性模型， 从而把线性模型看作一个神经网络。 首先，我们用“层”符号来重写这个模型。

7. 神经网络图

深度学习从业者喜欢绘制图表来可视化模型中正在发生的事情。 在 图3.1.2中，我们将线性回归模型描述为一个神经网络。 需要注意的是，该图只显示连接模式，即只显示每个输入如何连接到输出，隐去了权重和偏置的值。

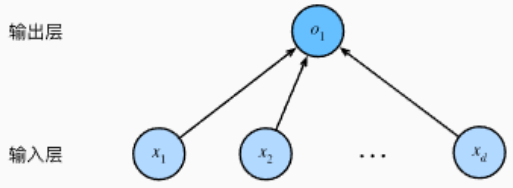


图3.1.2 线性回归是一个单层神经网络。

在 图3.1.2所示的神经网络中，输入为$x_1, \ldots, x_d$， 因此输入层中的*输入数*（或称为*特征维度*，feature dimensionality）为d。 网络的输出为o_1，因此输出层中的*输出数*是1。 需要注意的是，输入值都是已经给定的，并且只有一个*计算*神经元。 由于模型重点在发生计算的地方，所以通常我们在计算层数时不考虑输入层。 也就是说， 图3.1.2中神经网络的*层数*为1。 我们可以将线性回归模型视为仅由单个人工神经元组成的神经网络，或称为单层神经网络。

对于线性回归，每个输入都与每个输出（在本例中只有一个输出）相连， 我们将这种变换（ 图3.1.2中的输出层）称为*全连接层*（fully-connected layer）或称为*稠密层*（dense layer）。 下一章将详细讨论由这些层组成的网络。

1. 生物学

线性回归发明的时间（1795年）早于计算神经科学，所以将线性回归描述为神经网络似乎不合适。 当控制学家、神经生物学家沃伦·麦库洛奇和沃尔特·皮茨开始开发人工神经元模型时， 他们为什么将线性模型作为一个起点呢？ 我们来看一张图片 图3.1.3： 这是一张由树突（dendrites，输入终端）、 细胞核（nucleus，CPU）组成的生物神经元图片。 轴突（axon，输出线）和轴突端子（axon terminal，输出端子） 通过突触（synapse）与其他神经元连接。

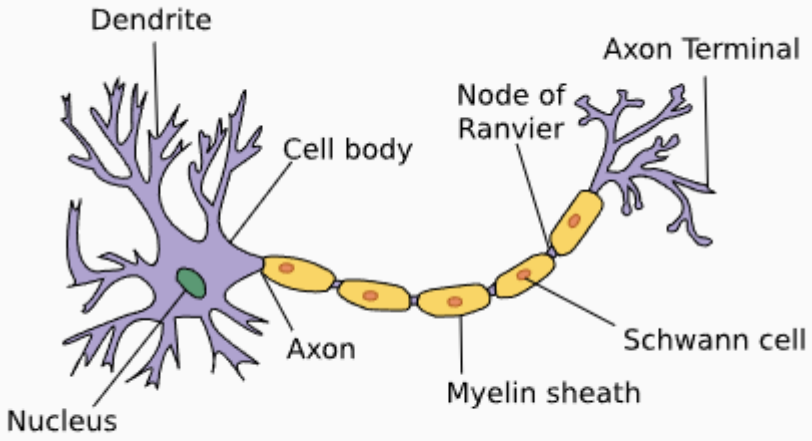


图3.1.3 真实的神经元。

树突中接收到来自其他神经元（或视网膜等环境传感器）的信息x_i。 该信息通过*突触权重*w_i来加权，以确定输入的影响（即，通过$x_i w_i$相乘来激活或抑制）。 来自多个源的加权输入以加权和$y = \sum_i x_i w_i + b$的形式汇聚在细胞核中， 然后将这些信息发送到轴突y中进一步处理，通常会通过$\sigma(y)$进行一些非线性处理。 之后，它要么到达目的地（例如肌肉），要么通过树突进入另一个神经元。

当然，许多这样的单元可以通过正确连接和正确的学习算法拼凑在一起， 从而产生的行为会比单独一个神经元所产生的行为更有趣、更复杂， 这种想法归功于我们对真实生物神经系统的研究。

当今大多数深度学习的研究几乎没有直接从神经科学中获得灵感。 我们援引斯图尔特·罗素和彼得·诺维格在他们的经典人工智能教科书 *Artificial Intelligence:A Modern Approach* (Russell and Norvig, 2016) 中所说的：虽然飞机可能受到鸟类的启发，但几个世纪以来，鸟类学并不是航空创新的主要驱动力。 同样地，如今在深度学习中的灵感同样或更多地来自数学、统计学和计算机科学。