线性代数

1. 标量

如果你曾经在餐厅支付餐费，那么应该已经知道一些基本的线性代数，比如在数字间相加或相乘。 例如，北京的温度为$52^\circ F$（华氏度，除摄氏度外的另一种温度计量单位）。 严格来说，仅包含一个数值被称为*标量*（scalar）。 如果要将此华氏度值转换为更常用的摄氏度， 则可以计算表达式$c = \frac{5}{9}(f - 32)$，并将f赋为52。 在此等式中，每一项（5、9和32）都是标量值。 符号c和f称为*变量*（variable），它们表示未知的标量值。

本书采用了数学表示法，其中标量变量由普通小写字母表示（例如，x、y和z）。 本书用$\mathbb{R}$表示所有（连续）*实数*标量的空间，之后将严格定义*空间*（space）是什么， 但现在只要记住表达式$x \in \mathbb{R}$是表示x是一个实值标量的正式形式。 符号$\in$称为“属于”，它表示“是集合中的成员”。 例如$x, y \in \{0, 1\}$可以用来表明x和y是值只能为0或1的数字。

标量由只有一个元素的张量表示。 下面的代码将实例化两个标量，并执行一些熟悉的算术运算，即加法、乘法、除法和指数。

```
import torch

x = torch.tensor(3.0)
y = torch.tensor(2.0)
print(x + y)
print(x - y)
print(x * y)
print(x / y)
print(x ** y)
```

```
tensor(5.)
tensor(1.)
tensor(6.)
tensor(1.5000)
tensor(9.)
```

2. 向量

向量可以被视为标量值组成的列表。 这些标量值被称为向量的元素（element）或分量（component）。 当向量表示数据集中的样本时，它们的值具有一定的现实意义。 例如，如果我们正在训练一个模型来预测贷款违约风险，可能会将每个申请人与一个向量相关联， 其分量与其收入、工作年限、过往违约次数和其他因素相对应。 如果我们正在研究医院患者可能面临的心脏病发作风险，可能会用一个向量来表示每个患者， 其分量为最近的生命体征、胆固醇水平、每天运动时间等。 在数学表示法中，向量通常记为粗体、小写的符号 （例如x,y和z).

人们通过一维张量表示向量。一般来说，张量可以具有任意长度，取决于机器的内存限制。

```
x = torch.arange(4)
print(x)
```

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

我们可以使用下标来引用向量的任一元素，例如可以通过x(i)来引用第i个元素。 注意x(i)元素是一个标量，所以我们在引用它时不会加粗。 大量文献认为列向量是向量的默认方向，在本书中也是如此。 在数学中，向量
x可以写为：

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

其中x(1)....x(n)是向量的元素。在代码中，我们通过张量的索引来访问任一元素。

```
print(x[3])
```

```
tensor(3)
```

2.1 长度,维度和形状

向量只是一个数字数组，就像每个数组都有一个长度一样，每个向量也是如此。 在数学表示法中，如果我们想说一个向量$\mathbf{x}$由n个实值标量组成， 可以将其表示为$\mathbf{x} \in \mathbb{R}^n$。 向量的长度通常称为向量的*维度*（dimension）。

与普通的Python数组一样，我们可以通过调用Python的内置`len()`函数来访问张量的长度。

当用张量表示一个向量（只有一个轴）时，我们也可以通过.shape属性访问向量的长度。 形状（shape）是一个元素组，列出了张量沿每个轴的长度（维数）。 对于只有一个轴的张量，形状只有一个元素。
请注意，维度（dimension）这个词在不同上下文时往往会有不同的含义，这经常会使人感到困惑。 为了清楚起见，我们在此明确一下： 向量或轴的维度被用来表示向量或轴的长度，即向量或轴的元素数量。 然而，张量的维度用来表示张量具有的轴数。 在这个意义上，张量的某个轴的维数就是这个轴的长度。

```
print(len(x))
print(x.shape)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

Cell In[1], line 1
----> 1 print(len(x))
      2 print(x.shape)
```

```
NameError: name 'x' is not defined
```

2.3 矩阵

正如向量将标量从零阶推广到一阶，矩阵将向量从一阶推广到二阶。 矩阵，我们通常用粗体、大写字母来表示（例如，$\mathbf{X}$、$\mathbf{Y}$和$\mathbf{Z}$），在代码中表示为具有两个轴的张量。

数学表示法使用$\mathbf{A} \in \mathbb{R}^{m \times n}$来表示矩阵$\mathbf{A}$，其由$m$行和$n$列的实值标量组成。 我们可以将任意矩阵$\mathbf{A} \in \mathbb{R}^{m \times n}$视为一个表格， 其中每个元素$a_{ij}$属于第$i$行第$j$列：

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{2.3.2}$$

对于任意$\mathbf{A} \in \mathbb{R}^{m \times n}$， $\mathbf{A}$的形状是（m,n）或$m \times n$。 当矩阵具有相同数量的行和列时，其形状将变为正方形； 因此，它被称为*方阵*（square matrix）。

当调用函数来实例化张量时， 我们可以通过指定两个分量m和n来创建一个形状为$m \times n$的矩阵。

```
A = torch.arange(20).reshape(5, 4)
print(A)
```

我们可以通过行索引（i）和列索引（j）来访问矩阵中的标量元素a_{ij}， 例如$[\mathbf{A}]_{ij}$。 如果没有给出矩阵$\mathbf{A}$的标量元素，如在 (2.3.2)那样， 我们可以简单地使用矩阵$\mathbf{A}$的小写字母索引下标a_{ij}来引用$[\mathbf{A}]_{ij}$。 为了表示起来简单，只有在必要时才会将逗号插入到单独的索引中， 例如$a_{2,3j}$和$[\mathbf{A}]_{2i-1,3}$。

当我们交换矩阵的行和列时，结果称为矩阵的*转置*（transpose）。 通常用$\mathbf{a}^\top$来表示矩阵的转置，如果$\mathbf{B} = \mathbf{A}^\top$， 则对于任意$i$和$j$，都有$b_{ij} = a_{ji}$。 因此，在 (2.3.2)中的转置是一个形状为$n \times m$的矩阵：

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}. \tag{2.3.3}$$

现在在代码中访问矩阵的转置。

```
print(A.T)
```

```
B = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
print(B)

B_T = B.T
print(B_T)
print(B == B_T)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

Cell In[2], line 1
----> 1 B = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
      2 print(B)
      4 B_T = B.T
```

```
NameError: name 'torch' is not defined
```

2.4 张量

当我们开始处理图像时，张量将变得更加重要，图像以
维数组形式出现， 其中3个轴对应于高度、宽度，以及一个通道（channel）轴， 用于表示颜色通道（红色、绿色和蓝色）。 现在先将高阶张量暂放一边，而是专注学习其基础知识。

```
X = torch.arange(24).reshape(2, 3, 4)
print(X)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

Cell In[3], line 1
----> 1 X = torch.arange(24).reshape(2, 3, 4)
      2 print(X)
```

```
NameError: name 'torch' is not defined
```

2.5 张量算法的基本性质

标量、向量、矩阵和任意数量轴的张量（本小节中的“张量”指代数对象）有一些实用的属性。 例如，从按元素操作的定义中可以注意到，任何按元素的一元运算都不会改变其操作数的形状。 同样，给定具有相同形状的任意两个张量，任何按元素二元运算的结果都将是相同形状的张量。 例如，将两个相同形状的矩阵相加，会在这两个矩阵上执行元素加法。

```
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
B = A.clone()  # 通过分配新内存，将A的一个副本分配给B
print(A)
print(A + B)
print(A * 2)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

Cell In[4], line 1
----> 1 A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
      2 B = A.clone()  # 通过分配新内存，将A的一个副本分配给B
      3 print(A)
```

```
NameError: name 'torch' is not defined
```

具体而言，两个矩阵的按元素乘法称为*Hadamard积*（Hadamard product）（数学符号$\odot$）。对于矩阵$\mathbf{B} \in \mathbb{R}^{m \times n}$，其中第$i$行和第$j$列的元素是$b_{ij}$。矩阵$\mathbf{A}$（在(2.3.2)中定义）和$\mathbf{B}$的Hadamard积为：

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \tag{2.3.4}$$

将张量乘以或加上一个标量不会改变张量的形状，其中张量的每个元素都将与标量相加或相乘。

```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
print(a + X)
print((a * X).shape)
print(A * B)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

Cell In[5], line 2
      1 a = 2
----> 2 X = torch.arange(24).reshape(2, 3, 4)
      3 print(a + X)
      4 print((a * X).shape)
```

```
NameError: name 'torch' is not defined
```

2.6. 降维

我们可以对任意张量进行的一个有用的操作是计算其元素的和，数学表示法使用∑符号表示求和,为了表示长度为d的向量中元素的综合，可以记为∑(^d)(_i=1){x_i},在代码中可以调用起算求和的函数。

我们可以表示任意形状张量的元素和。例如，矩阵$\mathbf{A}$中元素的和可以记为$\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$。

默认情况下，调用求和函数会沿所有的轴降低张量的维度，使它变为一个标量。我们还可以指定张量沿哪一个轴来通过求和降低维度。以矩阵为例，为了通过求和所有行的元素来降维（轴0），可以在调用函数时指定axis=0。由于输入矩阵沿0轴降维以生成输出向量，因此输入轴0的维数在输出形状中消失。

```
x = torch.arange(4, dtype=torch.float32)
x, x.sum()
A_sum_axis0 = A.sum(axis=0)
A_sum_axis1 = A.sum(axis=1)
print(A)
print(A_sum_axis0)
print(A_sum_axis0.shape)
print(A_sum_axis1)
print(A_sum_axis1.shape)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
tensor([40., 45., 50., 55.])
torch.Size([4])
tensor([ 6., 22., 38., 54., 70.])
torch.Size([5])
```

沿着行和列对矩阵求和，等价于对矩阵的所有元素进行求和。

```
A_sum_axis01 = A.sum(axis=[0, 1])  # 结果和A.sum()相同
print(A_sum_axis01)
```

```
tensor(190.)
```

一个与求和相关的量是平均值（mean或average）。 我们通过将总和除以元素总数来计算平均值。 在代码中，我们可以调用函数来计算任意形状张量的平均值。
mean():求平均
numel():元素个数

```
print(A.mean())
print(A.sum() / A.numel())
```

```
tensor(9.5000)
tensor(9.5000)
```

同样，计算平均值的函数也可以沿指定轴降低张量的维度。

```
print(A)
print(A.sum(axis=0))
print(A.mean(axis=0))
print(A.shape)
print(A.sum(axis=0) / A.shape[0])
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
tensor([40., 45., 50., 55.])
tensor([ 8.,  9., 10., 11.])
torch.Size([5, 4])
tensor([ 8.,  9., 10., 11.])
```

2.6 非降维求和

但是，有时在调用函数来计算总和或均值时保持轴数不变会很有用。
例如，由于sum_A在对每行进行求和后仍保持两个轴，我们可以通过广播将A除以sum_A。

```
print(A)
sum_A = A.sum(axis=1, keepdims=True)
print(sum_A)
print(sum_A.shape)

A_sum_A = A / sum_A
print(A_sum_A)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
tensor([[ 6.],
        [22.],
        [38.],
        [54.],
        [70.]])
torch.Size([5, 1])
tensor([[0.0000, 0.1667, 0.3333, 0.5000],
        [0.1818, 0.2273, 0.2727, 0.3182],
        [0.2105, 0.2368, 0.2632, 0.2895],
        [0.2222, 0.2407, 0.2593, 0.2778],
        [0.2286, 0.2429, 0.2571, 0.2714]])
```

如果我们想沿某个轴计算A元素的累积总和， 比如axis=0（按行计算），可以调用cumsum函数。 此函数不会沿任何轴降低输入张量的维度。

```
print(A)
A_cumsum = A.cumsum(axis=0)
print("--------------------")
print(A_cumsum)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
--------------------
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  6.,  8., 10.],
        [12., 15., 18., 21.],
        [24., 28., 32., 36.],
        [40., 45., 50., 55.]])
```

2.7 点积（Dot Product）

我们已经学习了按元素操作、求和及平均值。 另一个最基本的操作之一是点积。 给定两个向量$\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$， 它们的*点积*（dot product）$\mathbf{x}^\top \mathbf{y}$ （或$\langle \mathbf{x}, \mathbf{y} \rangle$） 是相同位置的按元素乘积的和：$\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$。

点积计算规则： 对应元素相乘的和
注意，我们可以通过执行按元素乘法，然后进行求和来表示两个向量的点积

点积在很多场合都很有用。 例如，给定一组由向量$\mathbf{x} \in \mathbb{R}^d$表示的值， 和一组由$\mathbf{w} \in \mathbb{R}^d$表示的权重。 $\mathbf{x}$中的值根据权重$\mathbf{w}$的加权和， 可以表示为点积$\mathbf{x}^\top \mathbf{w}$。 当权重为非负数且和为1（即$\left(\sum_{i=1}^{d} w_i = 1\right)$）时， 点积表示*加权平均*（weighted average）。 将两个向量规范化得到单位长度后，点积表示它们夹角的余弦。 本节后面的内容将正式介绍*长度*（length）的概念。

```
y = torch.ones(4, dtype=torch.float32)
print(x)
print("-----------")
print(y)
print("-----------")
print(torch.dot(x, y))
print("-----------")
print(torch.sum(x * y))
```

```
tensor([0., 1., 2., 3.])
-----------
tensor([1., 1., 1., 1.])
-----------
tensor(6.)
-----------
tensor(6.)
```

3 矩阵-向量积

现在我们知道如何计算点积，可以开始理解*矩阵-向量积*（matrix-vector product）。 回顾分别在 (2.3.2)和 (2.3.1) 中定义的矩阵$\mathbf{A} \in \mathbb{R}^{m \times n}$和向量$\mathbf{x} \in \mathbb{R}^n$。 让我们将矩阵$\mathbf{A}$用它的行向量表示：

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \tag{2.3.5}$$

其中每个$\mathbf{a}_i^\top \in \mathbb{R}^n$都是行向量，表示矩阵的第$i$行。 矩阵向量积$\mathbf{A}\mathbf{x}$是一个长度为$m$的列向量， 其第$i$个元素是点积$\mathbf{a}_i^\top \mathbf{x}$:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \tag{2.3.6}$$

我们可以把一个矩阵$\mathbf{A} \in \mathbb{R}^{m \times n}$乘法看作一个从$\mathbb{R}^n$到$\mathbb{R}^m$向量的转换。 这些转换是非常有用的，例如可以用方阵的乘法来表示旋转。 后续章节将讲到，我们也可以使用矩阵-向量积来描述在给定前一层的值时， 求解神经网络每一层所需的复杂计算。

在代码中使用张量表示矩阵-向量积，我们使用mv函数。 当我们为矩阵A和向量x调用torch.mv(A, x)时，会执行矩阵-向量积。 注意，A的列维数（沿轴1的长度）必须与x的维数（其长度）相同。

```
A_mv_x = torch.mv(A, x)
print(A_mv_x)
print("---------等同于------------------")
Ax_sum = (A * x).sum(axis=1)
print(Ax_sum)
```

```
tensor([ 14.,  38.,  62.,  86., 110.])
---------等同于------------------
tensor([ 14.,  38.,  62.,  86., 110.])
```

3.1 矩阵-矩阵乘法

在掌握点积和矩阵-向量积的知识后， 那么**矩阵-矩阵乘法**（matrix-matrix multiplication）应该很简单。

假设有两个矩阵$\mathbf{A} \in \mathbb{R}^{n \times k}$和$\mathbf{B} \in \mathbb{R}^{k \times m}$:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \tag{2.3.7}$$

用行向量$\mathbf{a}_i^\top \in \mathbb{R}^k$表示矩阵$\mathbf{A}$的第$i$行，并让列向量$\mathbf{b}_j \in \mathbb{R}^k$作为矩阵$\mathbf{B}$的第$j$列。要生成矩阵积$\mathbf{C} = \mathbf{AB}$，最简单的方法是考虑$\mathbf{A}$的行向量和$\mathbf{B}$的列向量:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}. \tag{2.3.8}$$

当我们简单地将每个元素c_{ij}计算为点积$\mathbf{a}_i^\top \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}. \tag{2.3.9}$$

我们可以将矩阵-矩阵乘法$\mathbf{AB}$看作简单地执行m次矩阵-向量积，并将结果拼接在一起，形成一个$n \times m$矩阵。 在下面的代码中，我们在A和B上执行矩阵乘法。 这里的A是一个5行4列的矩阵，B是一个4行3列的矩阵。 两者相乘后，我们得到了一个5行3列的矩阵。

矩阵乘法要求： 矩阵A(n*k) 和矩阵B(*k*m)他们两个中 矩阵A的列数需要等于矩阵B的行数

矩阵-矩阵乘法可以简单地称为矩阵乘法，不应与“Hadamard积”混淆。

```
B = torch.ones(4, 3)
A = A
print("A:", A.shape)
print("B:", B.shape)
mm_AB = torch.mm(A, B)

print(mm_AB)
```

```
A: torch.Size([5, 4])
B: torch.Size([4, 3])
tensor([[ 6.,  6.,  6.],
        [22., 22., 22.],
        [38., 38., 38.],
        [54., 54., 54.],
        [70., 70., 70.]])
```

3.2 范数

线性代数中最有用的一些运算符是范数（norm）。 非正式地说，向量的范数是表示一个向量有多大。这里考虑的大小（size）概念不涉及维度，而是分量的大小。

在线性代数中，向量范数是将向量映射到标量的函数f。 给定任意向量$\mathbf{x}$，向量范数要满足一些属性。 第一个性质是：如果我们按常数因子α缩放向量的所有元素， 其范数也会按相同常数因子的*绝对值*缩放：

$$f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x}). \tag{2.3.10}$$

第二个性质是熟悉的三角不等式:

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}). \tag{2.3.11}$$

第三个性质简单地说范数必须是非负的:

$$f(\mathbf{x}) \geq 0. \tag{2.3.12}$$

这是有道理的。因为在大多数情况下，任何东西的最小的*大小*是0。 最后一个性质要求范数最小为0，当且仅当向量全由0组成。

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \tag{2.3.13}$$

范数听起来很像距离的度量。 欧几里得距离和毕达哥拉斯定理中的非负性概念和三角不等式可能会给出一些启发。事实上，欧几里得距离是一个L_2范数： 假设n维向量$\mathbf{x}$中的元素是$x_1, \ldots, x_n$，其L_2*范数*是向量元素平方和的平方根：

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}, \tag{2.3.14}$$

其中，在L_2范数中常常省略下标2，也就是说$\|\mathbf{x}\|$等同于$\|\mathbf{x}\|_2$。 在代码中，我们可以按如下方式计算向量的L_2范数。

深度学习中更经常地使用L_2范数的平方，也会经常遇到L_1范数，它表示为向量元素的绝对值之和：

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|. \tag{2.3.15}$$

与L_2范数相比，L_1范数受异常值的影响较小。 为了计算L_1范数，我们将绝对值函数和按元素求和组合起来。

L_2范数和L_1范数都是更一般的L_p范数的特例：

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}. \tag{2.3.16}$$

类似于向量的L_2范数，矩阵$\mathbf{X} \in \mathbb{R}^{m \times n}$的*Frobenius范数*（Frobenius norm）是矩阵元素平方和的平方根：

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}^2}. \tag{2.3.17}$$

Frobenius范数满足向量范数的所有性质，它就像是矩阵形向量的L_2范数。 调用以下函数将计算矩阵的Frobenius范数。

```
u = torch.tensor([3.0, -4.0])
norm = torch.norm(u)
print(norm)
print("------------")
abs_sum = torch.abs(u).sum()
print(abs_sum)
print("------------")
frobenius = torch.norm(torch.ones((4, 9)))
print(frobenius)
```

```
tensor(5.)
------------
tensor(7.)
------------
tensor(6.)
```

3.3 范数和目标

在深度学习中，我们经常试图解决优化问题： 最大化分配给观测数据的概率; 最小化预测和真实观测之间的距离。 用向量表示物品（如单词、产品或新闻文章），以便最小化相似项目之间的距离，最大化不同项目之间的距离。 目标，或许是深度学习算法最重要的组成部分（除了数据），通常被表达为范数。

```
# 练习

import torch

a= torch.ones([2,5,4])
print(a)
print(a.shape)
```

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

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

```
print(a.sum(axis=1))
```

```
tensor([[5., 5., 5., 5.],
        [5., 5., 5., 5.]])
```

```
print(a.sum(axis=0))
```

```
tensor([[2., 2., 2., 2.],
        [2., 2., 2., 2.],
        [2., 2., 2., 2.],
        [2., 2., 2., 2.],
        [2., 2., 2., 2.]])
```

```
print(a.sum(axis=2))
```

```
tensor([[4., 4., 4., 4., 4.],
        [4., 4., 4., 4., 4.]])
```

```
print(a.sum(axis=1,keepdims=True))
```

```
tensor([[[5., 5., 5., 5.]],

        [[5., 5., 5., 5.]]])
```

微积分

在2500年前，古希腊人把一个多边形分成三角形，并把它们的面积相加，才找到计算多边形面积的方法。 为了求出曲线形状（比如圆）的面积，古希腊人在这样的形状上刻内接多边形。 如 图2.4.1所示，内接多边形的等长边越多，就越接近圆。 这个过程也被称为*逼近法*（method of exhaustion）。

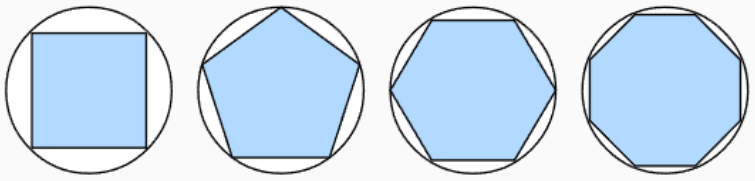

图2.4.1 用逼近法求圆的面积

事实上，逼近法就是*积分*（integral calculus）的起源。 2000多年后，微积分的另一支，*微分*（differential calculus）被发明出来。 在微分学最重要的应用是优化问题，即考虑如何把事情做到最好。 正如在 2.3.10.1节中讨论的那样， 这种问题在深度学习中是无处不在的。

在深度学习中，我们"训练"模型，不断更新它们，使它们在看到越来越多的数据时变得越来越好。 通常情况下，变得更好意味着最小化一个*损失函数*（loss function）， 即一个衡量"模型有多糟糕"这个问题的分数。 最终，我们真正关心的是生成一个模型，它能够在从未见过的数据上表现良好。 但"训练"模型只能将模型与我们实际能看到的数据相拟合。 因此，我们可以将拟合模型的任务分解为两个关键问题：

- *优化*（optimization）： 用模型拟合观测数据的过程；
- *泛化*（generalization）： 数学原理和实践者的智慧，能够指导我们生成出有效性超出用于训练的数据集本身的模型。

为了帮助读者在后面的章节中更好地理解优化问题和方法， 本节提供了一个非常简短的入门教程，帮助读者快速掌握深度学习中常用的微分知识。

事实上，逼近法就是积分（integral calculus）的起源。 2000多年后，微积分的另一支，微分（differential calculus）被发明出来。 在微分学最重要的应用是优化问题，即考虑如何把事情做到最好。 正如在 2.3.10.1节中讨论的那样， 这种问题在深度学习中是无处不在的。

在深度学习中，我们"训练"模型，不断更新它们，使它们在看到越来越多的数据时变得越来越好。 通常情况下，变得更好意味着最小化一个损失函数（loss function）， 即一个衡量"模型有多糟糕"这个问题的分数。 最终，我们真正关心的是生成一个模型，它能够在从未见过的数据上表现良好。 但"训练"模型只能将模型与我们实际能看到的数据相拟合。 因此，我们可以将拟合模型的任务分解为两个关键问题：

1. 优化（optimization）： 用模型拟合观测数据的过程；
2. 泛化（generalization）： 数学原理和实践者的智慧，能够指导我们生成出有效性超出用于训练的数据集本身的模型。

为了帮助读者在后面的章节中更好地理解优化问题和方法， 本节提供了一个非常简短的入门教程，帮助读者快速掌握深度学习中常用的微分知识。

1 导数和微分

我们首先讨论导数的计算，这是几乎所有深度学习优化算法的关键步骤。 在深度学习中，我们通常选择对于模型参数可微的损失函数。 简而言之，对于每个参数， 如果我们把这个参数增加或减少一个无穷小的量，可以知道损失会以多快的速度增加或减少，

假设我们有一个函数$f: \mathbb{R} \to \mathbb{R}$，其输入和输出都是标量。 如果$f$的*导数*存在，这个极限被定义为

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{2.4.1}$$

如果$f'(a)$存在，则称f在a处是*可微*（differentiable）的。 如果f在一个区间内的每个数上都是可微的，则此函数在此区间中是可微的。 我们可以将 (2.4.1)中的导数$f'(x)$解释为$f(x)$相对于x的*瞬时*（instantaneous）变化率。 所谓的瞬时变化率是基于x中的变化h，且h接近0。

为了更好地解释导数，让我们做一个实验。 定义$u = f(x) = 3x^2 - 4x$如下：

```
%matplotlib inline
import numpy as np
from matplotlib_inline import backend_inline
from d2l import torch as d2l

def f(x):
    return 3 * x ** 2 - 4 * x
```

通过令$x = 1$并让h接近0， (2.4.1)中$\frac{f(x+h) - f(x)}{h}$的数值结果接近2。 虽然这个实验不是一个数学证明，但稍后会看到，当$x = 1$时，导数u'是2。

```
def numerical_lim(f, x, h):
    return (f(x + h) - f(x)) / h


h = 0.1
for i in range(5):
    print(f'h={h:.5f}, numerical limit={numerical_lim(f, 1, h):.5f}')
    h *= 0.1
```

```
h=0.10000, numerical limit=2.30000
h=0.01000, numerical limit=2.03000
h=0.00100, numerical limit=2.00300
h=0.00010, numerical limit=2.00030
h=0.00001, numerical limit=2.00003
```

让我们熟悉一下导数的几个等价符号。 给定$y = f(x)$，其中x和y分别是函数f的自变量和因变量。以下表达式是等价的：

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f(x) = Df(x) = D_x f(x), \tag{2.4.2}$$

其中符号$\frac{d}{dx}$和D是*微分运算符*，表示*微分*操作。 我们可以使用以下规则来对常见函数求微分：

- $DC = 0$（C是一个常数）
- $Dx^n = nx^{n-1}$（*幂律*（power rule），n是任意实数）
- $De^x = e^x$
- $D\ln(x) = 1/x$

为了微分一个由一些常见函数组成的函数，下面的一些法则方便使用。 假设函数f和g都是可微的，C是一个常数，则：

常数相乘法则

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x), \tag{2.4.3}$$

加法法则

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \tag{2.4.4}$$

乘法法则

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \tag{2.4.5}$$

除法法则

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \tag{2.4.6}$$

现在我们可以应用上述几个法则来计算$u' = f'(x) = 3\frac{d}{dx}x^2 - 4\frac{d}{dx}x = 6x - 4$。 令$x = 1$，我们有$u' = 2$：在这个实验中，数值结果接近2， 这一点得到了在本节前面的实验的支持。 当$x = 1$时，此导数也是曲线$u = f(x)$切线的斜率。

为了对导数的这种解释进行可视化，我们将使用matplotlib， 这是一个Python中流行的绘图库。 要配置matplotlib生成图形的属性，我们需要定义几个函数。 在下面，use_svg_display函数指定matplotlib软件包输出svg图表以获得更清晰的图像。

注意，注释#@save是一个特殊的标记，会将对应的函数、类或语句保存在d2l包中。 因此，以后无须重新定义就可以直接调用它们（例如，d2l.use_svg_display()）。

注意，注释#@save是一个特殊的标记，会将对应的函数、类或语句保存在d2l包中。 因此，以后无须重新定义就可以直接调用它们（例如，d2l.use_svg_display()）。

```
def use_svg_display():  #@save
    """使用svg格式在Jupyter中显示绘图"""
    backend_inline.set_matplotlib_formats('svg')
```

我们定义set_figsize函数来设置图表大小。 注意，这里可以直接使用d2l.plt，因为导入语句 from matplotlib import pyplot as plt已标记为保存到d2l包中。

```
def set_figsize(figsize=(3.5, 2.5)):  #@save
    """设置matplotlib的图表大小"""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

下面的set_axes函数用于设置由matplotlib生成图表的轴的属性。

```
#@save
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """设置matplotlib的轴"""
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()
```

通过这三个用于图形配置的函数，定义一个plot函数来简洁地绘制多条曲线， 因为我们需要在整个书中可视化许多曲线。

```
#@save
def plot(X, Y=None, xlabel=None, ylabel=None, legend=None, xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None):
    """绘制数据点"""
    if legend is None:
        legend = []

    set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()

    # 如果X有一个轴，输出True
    def has_one_axis(X):
        return (hasattr(X, "ndim") and X.ndim == 1 or isinstance(X, list)
                and not hasattr(X[0], "__len__"))

    if has_one_axis(X):
        X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        if len(x):
            axes.plot(x, y, fmt)
        else:
            axes.plot(y, fmt)
```

```
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
```

```
x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], 'x', 'f(x)', legend=['f(x)', 'Tangent line (x=1)'])
```

2 偏导数

到目前为止，我们只讨论了仅含一个变量的函数的微分。 在深度学习中，函数通常依赖于许多变量。 因此，我们需要将微分的思想推广到*多元函数*（multivariate function）上。

设$y = f(x_1, x_2, \ldots, x_n)$是一个具有$n$个变量的函数。 y关于第i个参数x_i的*偏导数*（partial derivative）为：

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{2.4.7}$$

为了计算$\frac{\partial y}{\partial x_i}$， 我们可以简单地将$x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$看作常数， 并计算$y$关于$x_i$的导数。 对于偏导数的表示，以下是等价的：

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{2.4.8}$$

3 梯度

我们可以连结一个多元函数对其所有变量的偏导数，以得到该函数的*梯度*（gradient）向量。 具体而言，设函数$f: \mathbb{R}^n \to \mathbb{R}$的输入是 一个$n$维向量$\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$，并且输出是一个标量。 函数$f(\mathbf{x})$相对于$\mathbf{x}$的梯度是一个包含$n$个偏导数的向量:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right]^\top, \tag{2.4.9}$$

其中$\nabla_{\mathbf{x}} f(\mathbf{x})$通常在没有歧义时被$\nabla f(\mathbf{x})$取代。

假设$\mathbf{x}$为n维向量，在微分多元函数时经常使用以下规则:

- 对于所有$\mathbf{A} \in \mathbb{R}^{m \times n}$，都有$\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$
- 对于所有$\mathbf{A} \in \mathbb{R}^{n \times m}$，都有$\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$
- 对于所有$\mathbf{A} \in \mathbb{R}^{n \times n}$，都有$\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$
- $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$

同样，对于任何矩阵$\mathbf{X}$，都有$\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$。 正如我们之后将看到的，梯度对于设计深度学习中的优化算法有很大用处。

4 链式法则

然而，上面方法可能很难找到梯度。 这是因为在深度学习中，多元函数通常是*复合*（composite）的， 所以难以应用上述任何规则来微分这些函数。 幸运的是，链式法则可以被用来微分复合函数。

让我们先考虑单变量函数。假设函数$y = f(u)$和$u = g(x)$都是可微的，根据链式法则：

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}. \tag{2.4.10}$$

现在考虑一个更一般的场景，即函数具有任意数量的变量的情况。 假设可微分函数y有变量$u_1, u_2, \ldots, u_m$，其中每个可微分函数u_i都有变量$x_1, x_2, \ldots, x_n$。 注意，y是$x_1, x_2， \ldots, x_n$的函数。 对于任意$i = 1, 2, \ldots, n$，链式法则给出：

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial u_1}\frac{\partial u_1}{\partial x_i} + \frac{\partial y}{\partial u_2}\frac{\partial u_2}{\partial x_i} + \cdots + \frac{\partial y}{\partial u_m}\frac{\partial u_m}{\partial x_i} \tag{2.4.11}$$