2.1 数据操作

为了能够完成各种数据操作，我们需要某种方法来存储和操作数据。 通常，我们需要做两件重要的事：
（1）获取数据；
（2）将数据读入计算机后对其进行处理。 如果没有某种方法来存储数据，那么获取数据是没有意义的。
首先，我们介绍n维数组，也称为张量（tensor）。 使用过Python中NumPy计算包的读者会对本部分很熟悉。 无论使用哪个深度学习框架，它的张量类（在MXNet中为ndarray， 在PyTorch和TensorFlow中为Tensor）都与Numpy的ndarray类似。 但深度学习框架又比Numpy的ndarray多一些重要功能： 首先，GPU很好地支持加速计算，而NumPy仅支持CPU计算； 其次，张量类支持自动微分。 这些功能使得张量类更适合深度学习。 如果没有特殊说明，本书中所说的张量均指的是张量类的实例。

```
import torch
```

张量表示一个由数值组成的数组，这个数组可能有多个维度，具有一个轴的张量对于数学上的向量（vector）,具有两个轴的张量对应数学上的矩阵（matrix）;具有两个轴以上的张量没有特殊的数学名称。
创建一个行向量x 这个行向量包含以0开始的前12个整数，他们默认创建为整数，也可以指定创建类型为浮点数，张量中的每个值都称为张量的元素（element）。例如：张量 x 中有 12 个元素。除非额外指定，新的张量将存储在内存中，并采用基于CPU的计算。

```
x = torch.arange(12)
print(x)

y = torch.arange(0, 12, dtype=float)
print(y)
z = torch.arange(0, 1, 0.1)

print(z)
```

```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.],
       dtype=torch.float64)
tensor([0.0000, 0.1000, 0.2000, 0.3000, 0.4000, 0.5000, 0.6000, 0.7000, 0.8000,
        0.9000])
```

使用shape属性来访问张量（沿每个轴的长度）的形状 。

```
print(x.shape)
print(y.shape)
```

```
torch.Size([12])
torch.Size([12])
```

查看张量中元素的总数，即形状的所有元素乘积，可以检查它的大小（size）。 因为这里在处理的是一个向量，所以它的shape与它的size相同。

```
print(x.numel())
print(y.numel())
```

```
12
12
```

如果想改变一个张量的形状而不改变元素数量的元素值，可以调用reshape 函数

```
x = x.reshape(3, 4)
y = y.reshape(4, 3)
print(x)
print(y)
```

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.],
        [ 6.,  7.,  8.],
        [ 9., 10., 11.]], dtype=torch.float64)
```

```
print(x.shape)
print(y.shape)
```

```
torch.Size([3, 4])
torch.Size([4, 3])
```

我们不需要通过手动指定每个维度来改变形状。 也就是说，如果我们的目标形状是（高度,宽度）， 那么在知道宽度后，高度会被自动计算得出，不必我们自己做除法。 在上面的例子中，为了获得一个3行的矩阵，我们手动指定了它有3行和4列。 幸运的是，我们可以通过-1来调用此自动计算出维度的功能。 即我们可以用x.reshape(-1,4)或x.reshape(3,-1)来取代x.reshape(3,4)。

```
x = torch.arange(0, 12)
x = x.reshape(-1, 4)
print(x)
print(x.shape)
```

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
torch.Size([3, 4])
```

创建全0，全1，其他常量，或者从特定分布中随机采样的数字来初始化矩阵，我们可以创建一个形状为（2，3，4）的张量，其中所有的元素都设置为0

```
x = torch.zeros((2, 3, 4))
print(x)
print(x.shape)
y = torch.ones((2, 3, 4))
print(y)
print(y.shape)
```

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

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
torch.Size([2, 3, 4])
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.]]])
torch.Size([2, 3, 4])
```

有时我们想通过从某个特定的概率分布中随机采样来得到张量中每个元素的值。 例如，当我们构造数组来作为神经网络中的参数时，我们通常会随机初始化参数的值。 以下代码创建一个形状为（3,4）的张量。 其中的每个元素都从均值为0、标准差为1的标准高斯分布（正态分布）中随机采样。

```
z = torch.randn((3, 4))
print(z)
print(z.shape)
```

```
tensor([[ 0.6378, -0.3172, -1.4313,  1.2711],
        [-0.7190, -0.8561,  0.7441,  0.5401],
        [ 2.2458, -0.0089,  1.6704,  0.0695]])
torch.Size([3, 4])
```

我们还可以通过提供包含数值的Python列表（或嵌套列表），来为所需张量中的每个元素赋予确定值。 在这里，最外层的列表对应于轴0，内层的列表对应于轴1。

```
array = [[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]]
x = torch.tensor(array)
print(x)
print(x.shape)
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
torch.Size([3, 4])
```

2.1.2. 运算符

我们的兴趣不仅限于读取数据和写入数据。 我们想在这些数据上执行数学运算，其中最简单且最有用的操作是*按元素*（elementwise）运算。 它们将标准标量运算符应用于数组的每个元素。 对于将两个数组作为输入的函数，按元素运算将二元运算符应用于两个数组中的每对位置对应的元素。 我们可以基于任何从标量到标量的函数来创建按元素函数。

在数学表示法中，我们将通过符号$f: \mathbb{R} \rightarrow \mathbb{R}$来表示一元标量运算符（只接收一个输入）。 这意味着该函数从任何实数（$\mathbb{R}$）映射到另一个实数。 同样，我们通过符号$f: \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$表示二元标量运算符，这意味着该函数接收两个输入，并产生一个输出。 给定同一形状的任意两个向量$\mathbf{u}$和$\mathbf{v}$和二元运算符f，我们可以得到向量$\mathbf{c} = F(\mathbf{u}, \mathbf{v})$。 具体计算方法是$c_i \leftarrow f(u_i, v_i)$，其中$c_i$、$u_i$和$v_i$分别是向量$\mathbf{c}$、$\mathbf{u}$和$\mathbf{v}$中的元素。 在这里，我们通过将标量函数升级为按元素向量运算来生成向量值$F: \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$。

对于任意具有相同形状的张量， 常见的标准算术运算符（+、-、*、/和**）都可以被升级为按元素运算。 我们可以在同一形状的任意两个张量上调用按元素操作。 在下面的例子中，我们使用逗号来表示一个具有5个元素的元组，其中每个元素都是按元素操作的结果。

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
print(x + y)
print(x - y)
print(x * y)
print(x / y)
print(x ** y)  # **运算符是求幂运算
```

```
tensor([ 3.,  4.,  6., 10.])
tensor([-1.,  0.,  2.,  6.])
tensor([ 2.,  4.,  8., 16.])
tensor([0.5000, 1.0000, 2.0000, 4.0000])
tensor([ 1.,  4., 16., 64.])
```

“按元素”方式可以应用更多的计算，包括像求幂这样的一元运算符。

```
print(torch.exp(x))
```

```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

除了按元素计算外，我们还可以执行线性代数运算，包括向量点积和矩阵乘法。 我们将在 2.3节中解释线性代数的重点内容。

我们也可以把多个张量*连结*（concatenate）在一起， 把它们端对端地叠起来形成一个更大的张量。 我们只需要提供张量列表，并给出沿哪个轴连结。 下面的例子分别演示了当我们沿行（轴-0，形状的第一个元素） 和按列（轴-1，形状的第二个元素）连结两个矩阵时，会发生什么情况。 我们可以看到，第一个输出张量的轴-0长度（6）是两个输入张量轴-0长度的总和（$3 + 3$）； 第二个输出张量的轴-1长度（8）是两个输入张量轴-1长度的总和（$4 + 4$）。

```
x = torch.arange(12, dtype=torch.float32).reshape(3, -1)
y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
print(x)
print(y)
# 纵轴链接
z1 = torch.cat((x, y), dim=0)
# 横轴链接
z2 = torch.cat((x, y), dim=1)
print(z1)
print(z2)
print(z1.numel())
print(z2.numel())
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
tensor([[2., 1., 4., 3.],
        [1., 2., 3., 4.],
        [4., 3., 2., 1.]])
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]])
tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]])
24
24
```

有时，我们想通过逻辑运算符构建二元张量。 以X == Y为例： 对于每个位置，如果X和Y在该位置相等，则新张量中相应项的值为1。 这意味着逻辑语句X == Y在该位置处为真，否则该位置为0。

```
print(x == y)
```

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

对张量中的所有元素进行求和，会产生一个单元素张量。

```
print(x.sum())
```

```
tensor(66.)
```

2.1.3. 广播机制

在上面的部分中，我们看到了如何在相同形状的两个张量上执行按元素操作。 在某些情况下，即使形状不同，我们仍然可以通过调用 广播机制（broadcasting mechanism）来执行按元素操作。 这种机制的工作方式如下：

通过适当复制元素来扩展一个或两个数组，以便在转换之后，两个张量具有相同的形状；

对生成的数组执行按元素操作。

在大多数情况下，我们将沿着数组中长度为1的轴进行广播，如下例子：

```
a = torch.arange(3).reshape(3, 1)
b = torch.arange(2).reshape(1, 2)
print(a)
print(b)
```

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

由于a和b分别是3x1和1x2矩阵，如果让它们相加，它们的形状不匹配。 我们将两个矩阵广播为一个更大的3x2矩阵，如下所示：矩阵a将复制列， 矩阵b将复制行，然后再按元素相加。

```
print(a + b)
```

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

2.1.4. 索引和切片

```
print(x[-1])
print(x[1:3])
```

```
tensor([ 8.,  9., 10., 11.])
tensor([[ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

```
print(x)
x[1, 2] = 9
print(x)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5., 77.,  7.],
        [ 8.,  9., 10., 11.]])
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  9.,  7.],
        [ 8.,  9., 10., 11.]])
```

如果我们想为多个元素赋值相同的值，我们只需要索引所有元素，然后为它们赋值。 例如，[0:2, :]访问第1行和第2行，其中":"代表沿轴1（列）的所有元素。 虽然我们讨论的是矩阵的索引，但这也适用于向量和超过2个维度的张量。

```
x[0:2, :] = 12
print(x)
```

```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

2.1.5. 节省内存

运行一些操作可能会导致为新结果分配内存。 例如，如果我们用Y = X + Y，我们将取消引用Y指向的张量，而是指向新分配的内存处的张量。

在下面的例子中，我们用Python的id()函数演示了这一点， 它给我们提供了内存中引用对象的确切地址。 运行Y = Y + X后，我们会发现id(Y)指向另一个位置。 这是因为Python首先计算Y + X，为结果分配新的内存，然后使Y指向内存中的这个新位置。

```
before = id(y)
y = y + x
print(id(y) == before)
```

```
False
```

这可能是不可取的，原因有两个：

1. 首先，我们不想总是不必要地分配内存。在机器学习中，我们可能有数百兆的参数，并且在一秒内多次更新所有参数。通常情况下，我们希望原地执行这些更新；
2. 如果我们不原地更新，其他引用仍然会指向旧的内存位置，这样我们的某些代码可能会无意中引用旧的参数。

```
z = torch.zeros_like(y)
print(z)
before = id(z)
print(before)

z[:] = x + y
print(id(z) == before)
```

```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
1673675734672
True
```

如果在后续计算中没有重复使用X， 我们也可以使用X[:] = X + Y或X += Y来减少操作的内存开销。

```
before = id(x)
x += y
print(id(x) == before)
```

```
True
```

2.1.6. 转换为其他Python对象

将深度学习框架定义的张量转换为NumPy张量（ndarray）很容易，反之也同样容易。 torch张量和numpy数组将共享它们的底层内存，就地操作更改一个张量也会同时更改另一个张量。

```
a = x.numpy()
b = torch.tensor(a)
print("a:{}".format(type(a)))
print("a:{}".format(type(b)))
```

```
a:<class 'numpy.ndarray'>
a:<class 'torch.Tensor'>
```

要将大小为1的张量转换为Python标量，我们可以调用item函数或Python的内置函数。

```
a= torch.tensor([3.5])
print(a)
print(a.item())
print(float(a))
print(int(a))
```

```
tensor([3.5000])
3.5
3.5
3
```

深度学习存储和操作数据的主要接口是张量（n维数组）。它提供了各种功能，包括基本数学运算、广播、索引、切片、内存节省和转换其他Python对象。

2.2 读取数据集

我们首先创建一个人工数据集，并存储在CSV（逗号分隔值）文件 ../data/house_tiny.csv中。 以其他格式存储的数据也可以通过类似的方式进行处理。 下面我们将数据集按行写入CSV文件中。

```
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\n')  # 列名
    f.write('NA,Pave,127500\n')  # 每行表示一个数据样本
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

```
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms Alley   Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000
```

2.2.2. 处理缺失值

注意，“NaN”项代表缺失值。 为了处理缺失的数据，典型的方法包括插值法和删除法， 其中插值法用一个替代值弥补缺失值，而删除法则直接忽略缺失值。 在这里，我们将考虑插值法。

通过位置索引iloc，我们将data分成inputs和outputs， 其中前者为data的前两列，而后者为data的最后一列。 对于inputs中缺少的数值，我们用同一列的均值替换“NaN”项。

```
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

对于inputs中的类别值或离散值，我们将“NaN”视为一个类别。 由于“巷子类型”（“Alley”）列只接受两种类型的类别值“Pave”和“NaN”， pandas可以自动将此列转换为两列“Alley_Pave”和“Alley_nan”。 巷子类型为“Pave”的行会将“Alley_Pave”的值设置为1，“Alley_nan”的值设置为0。 缺少巷子类型的行会将“Alley_Pave”和“Alley_nan”分别设置为0和1。

对于inputs中的类别值或离散值，我们将“NaN”视为一个类别。 由于“巷子类型”（“Alley”）列只接受两种类型的类别值“Pave”和“NaN”， pandas可以自动将此列转换为两列“Alley_Pave”和“Alley_nan”。 巷子类型为“Pave”的行会将“Alley_Pave”的值设置为1，“Alley_nan”的值设置为0。 缺少巷子类型的行会将“Alley_Pave”和“Alley_nan”分别设置为0和1。

```
inputs= pd.get_dummies(inputs,dummy_na=True)
print(inputs)
```

```
   NumRooms  Alley_Pave  Alley_nan
0       NaN        True      False
1       2.0       False       True
2       4.0       False       True
3       NaN       False       True
```

```
import torch

X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(outputs.to_numpy(dtype=float))
X, y
```

```
(tensor([[nan, 1., 0.],
         [2., 0., 1.],
         [4., 0., 1.],
         [nan, 0., 1.]], dtype=torch.float64),
 tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

2.2. 数据预处理

2.2.1. 读取数据集

我们首先创建一个人工数据集，并存储在CSV（逗号分隔值）文件 ../data/house_tiny.csv中。 以其他格式存储的数据也可以通过类似的方式进行处理。 下面我们将数据集按行写入CSV文件中。

```
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\n')  # 列名
    f.write('NA,Pave,127500\n')  # 每行表示一个数据样本
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

```
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms Alley   Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000
```

2.2.2. 处理缺失值

注意，“NaN”项代表缺失值。 为了处理缺失的数据，典型的方法包括插值法和删除法， 其中插值法用一个替代值弥补缺失值，而删除法则直接忽略缺失值。 在这里，我们将考虑插值法。

通过位置索引iloc，我们将data分成inputs和outputs， 其中前者为data的前两列，而后者为data的最后一列。 对于inputs中缺少的数值，我们用同一列的均值替换“NaN”项。

```
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

对于inputs中的类别值或离散值，我们将“NaN”视为一个类别。 由于“巷子类型”（“Alley”）列只接受两种类型的类别值“Pave”和“NaN”， pandas可以自动将此列转换为两列“Alley_Pave”和“Alley_nan”。 巷子类型为“Pave”的行会将“Alley_Pave”的值设置为1，“Alley_nan”的值设置为0。 缺少巷子类型的行会将“Alley_Pave”和“Alley_nan”分别设置为0和1。

对于inputs中的类别值或离散值，我们将“NaN”视为一个类别。 由于“巷子类型”（“Alley”）列只接受两种类型的类别值“Pave”和“NaN”， pandas可以自动将此列转换为两列“Alley_Pave”和“Alley_nan”。 巷子类型为“Pave”的行会将“Alley_Pave”的值设置为1，“Alley_nan”的值设置为0。 缺少巷子类型的行会将“Alley_Pave”和“Alley_nan”分别设置为0和1。

```
inputs= pd.get_dummies(inputs,dummy_na=True)
print(inputs)
```

```
   NumRooms  Alley_Pave  Alley_nan
0       NaN        True      False
1       2.0       False       True
2       4.0       False       True
3       NaN       False       True
```

```
import torch

X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(outputs.to_numpy(dtype=float))
X, y
```

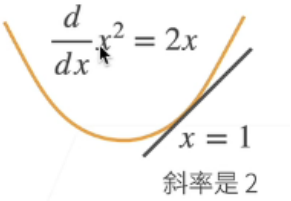
```
(tensor([[nan, 1., 0.],
         [2., 0., 1.],
         [4., 0., 1.],
         [nan, 0., 1.]], dtype=torch.float64),
 tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

标量导数

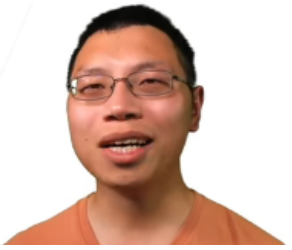
y	a	x^n	$\exp(x)$	$\log(x)$	$\sin(x)$
$\frac{dy}{dx}$	0	nx^{n-1}	$\exp(x)$	$\frac{1}{x}$	$\cos(x)$

a 不是 x 的函数

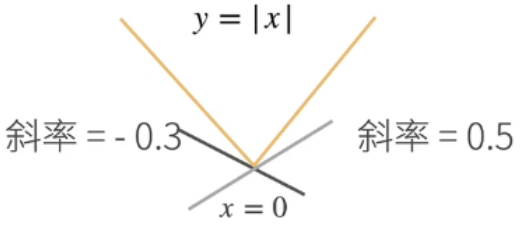
导数是切线的斜率



y	$u+v$	uv	$y=f(u),\ u=g(x)$
$\frac{dy}{dx}$	$\frac{du}{dx}+\frac{dv}{dx}$	$\frac{du}{dx}v+\frac{dv}{dx}u$	$\frac{dy}{du}\frac{du}{dx}$



- 将导数拓展到不可微的函数



$$\frac{\partial |x|}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ a & \text{if } x = 0, \quad a \in [-1,1] \end{cases}$$

另一个例子

$$\frac{\partial}{\partial x}\max(x,0) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \\ a & \text{if } x = 0, \quad a \in [0,1] \end{cases}$$

拓展到矩阵

		标量	向量	矩阵
		x $(1,)$	$\mathbf{x}$ $(n,1)$	$\mathbf{X}$ (n,k)
标量	y $(1,)$	$\frac{\partial y}{\partial x}$ $(1,)$	$\frac{\partial y}{\partial \mathbf{x}}$ $(1,n)$	$\frac{\partial y}{\partial \mathbf{X}}$ (k,n)
向量	$\mathbf{y}$ $(m,1)$	$\frac{\partial \mathbf{y}}{\partial x}$ $(m,1)$	$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ (m,n)	$\frac{\partial \mathbf{y}}{\partial \mathbf{X}}$ (m,k,n)
矩阵	$\mathbf{Y}$ (m,l)	$\frac{\partial \mathbf{Y}}{\partial x}$ (m,l)	$\frac{\partial \mathbf{Y}}{\partial \mathbf{x}}$ (m,l,n)	$\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}$ (m,l,k,n)

向量链式法则

- 标量链式法则

$$y = f(u),\ u = g(x) \qquad \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

- 拓展到向量

$$\frac{\partial y}{\partial \mathbf{x}} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial \mathbf{x}} \qquad \frac{\partial y}{\partial \mathbf{x}} = \frac{\partial y}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{x}} \qquad \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$$

$$(1,n)\quad (1,)\ (1,n) \qquad (1,n)\quad (1,k)\ \ (k,n) \qquad (m,n)\ \ (m,k)\ (k,n)$$

计算图

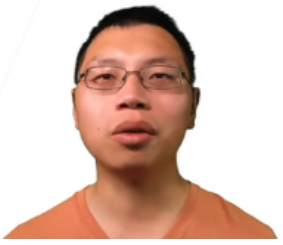
- 将代码分解成操作子
- 将计算表示成一个无环图
- 显示构造

```
from mxnet import sym

a = sym.var()
b = sym.var()
c = 2 * a + b
# bind data into a and b later
```

计算图

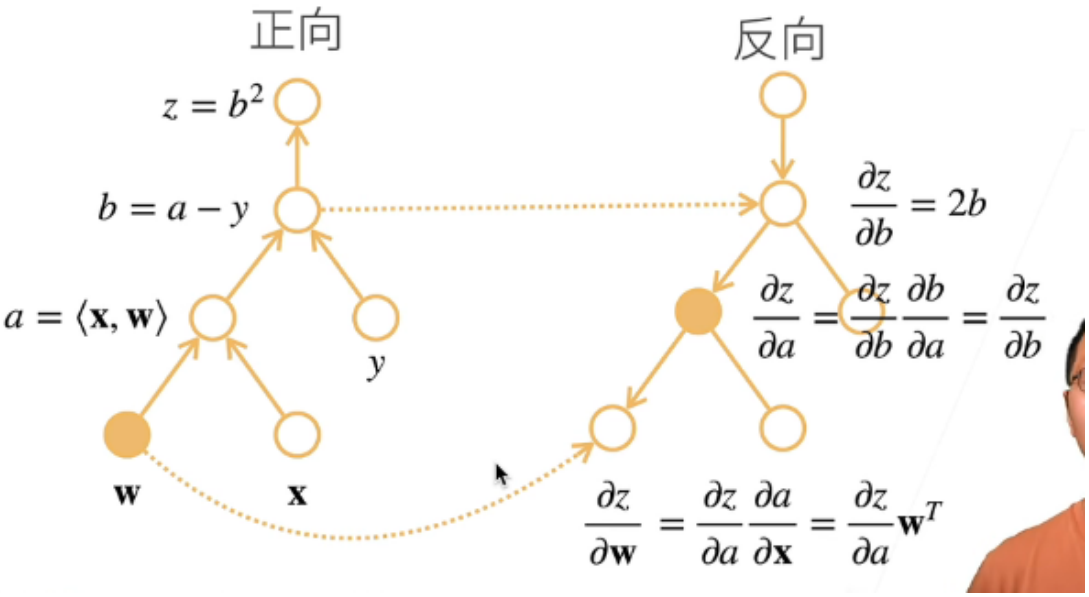
- 将代码分解成操作子
- 将计算表示成一个无环图
- 显式构造
  - Tensorflow/Theano/MXNet
- 隐式构造
  - PyTorch/MXNet

```
from mxnet import autograd, nd

with autograd.record():
    a = nd.ones((2,1))
    b = nd.ones((2,1)
    c = 2 * a + b
```



反向累积

$$z = \left(\langle \mathbf{x}, \mathbf{w} \rangle - y\right)^2$$



反向累积总结

- 构造计算图
- 前向：执行图，存储中间结果
- 反向：从相反方向执行图
  - 去除不需要的枝

复杂度

- 计算复杂度：O(n), n 是操作子个数
  - 通常正向和方向的代价类似
- 内存复杂度: O(n)，因为需要存储正向的所有中间结果
- 跟正向累积对比：
  - O(n) 计算复杂度用来计算一个变量的梯度
  - O(1) 内存复杂度

2.3 自动求导

假设我们想对函数y=(2x^T)x 关于列向量x求导

```
import torch
x= torch.arange(4.0)
x
```

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

在我们计算y关于x的梯度之前，我们需要一个地方来存储梯度

```
x.requires_grad_(True) # 等价于x=torch.arange(4.0,requires_grad=True)
x.grad
```

现在让我们计算y

```
y= 2*torch.dot(x,x)
y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

通过调用反向传播函数来自动计算y关于x每个分量的梯度

```
y.backward()
x.grad
```

```
tensor([ 0.,  4.,  8., 12.])
```

```
x.grad==4*x
```

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

现在让我们计算x的另一个函数

```
# 在默认情况下，pytorch会累积梯度，我们需要清楚之前的值
x.grad.zero_()
y=x.sum()
y.backward()
#向量x求和相当于向量x乘一个单位向量E，那么y对x求导后，就是y'=E
x.grad
```

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

深度学习中，我们的目的不是计算微分矩阵，而是批量中每个样本单独计算的偏导数之和

```
# 对非标量调用`backward`需要传入一个`gradient`参数，该参数指定微分函数是为了把张量对
#张量的求导转换为标量对张量的求导。

x.grad.zero_()
y=x*x
# 等价于y.gradward(torch.ones(len(x)))
y.sum().backward()
x.grad
```

```
tensor([ 0.,  4.,  8., 12.])
```

将某些计算移动到记录的计算图之外

```
x.grad.zero_()
y=x*x

u=y.detach()
z= u*x
z.sum().backward()
# 这里可以理解为对x求偏导 所以需要将u看作为一个常数，.detach() 可以阻止梯度回传
x.grad==u
```

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

```
x.grad.zero_()
y.sum().backward()
x.grad==2*x
x.grad
```

```
tensor([0., 2., 4., 6.])
```

即使构建函数的计算图需要通过python控制流（例如，条件，循环或任意函数调用），我们任然可以计算得到的变量的梯度

```
def f(a):
    b=a*2
    while b.norm()<1000:
        b=b*2
    if b.sum()>0:
        c=b
    else:
        c= 100*b
    return c
a= torch.randn(size=(),requires_grad=True)
print(a)
d=f(a)
d.backward()
a.grad
```

```
tensor(-0.7648, requires_grad=True)

tensor(204800.)
```

2.4. 自动微分

求导是几乎所有深度学习优化算法的关键步骤。 虽然求导的计算很简单，只需要一些基本的微积分。 但对于复杂的模型，手工进行更新是一件很痛苦的事情（而且经常容易出错）。

深度学习框架通过自动计算导数，即自动微分（automatic differentiation）来加快求导。 实际中，根据设计好的模型，系统会构建一个计算图（computational graph）， 来跟踪计算是哪些数据通过哪些操作组合起来产生输出。 自动微分使系统能够随后反向传播梯度。 这里，反向传播（backpropagate）意味着跟踪整个计算图，填充关于每个参数的偏导数。

2.4.1. 例子

假设我们想对函数y=2x^Tx关于列向量x求导，首先我们创建变量x并为其分配一个初始值。

```
import torch

x = torch.arange(4.0)
print(x)
```

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

在我们计算y关于x的梯度之前，需要一个地方来存储梯度。 重要的是，我们不会在每次对一个参数求导时都分配新的内存。 因为我们经常会成千上万次地更新相同的参数，每次都分配新的内存可能很快就会将内存耗尽。 注意，一个标量函数关于向量x的梯度是向量，并且与x具有相同的形状。

```
x.requires_grad_(True)  # 等价于x=torch.arange(4.0,requires_grad=True)
print(x.grad)  # 默认值是None
```

```
None
```

```
y = 2 * torch.dot(x, x)
print(y)
```

```
tensor(28., grad_fn=<MulBackward0>)
```

x是一个长度为4的向量，计算x和x的点积，得到了我们赋值给y的标量输出。 接下来，通过调用反向传播函数来自动计算y关于x每个分量的梯度，并打印这些梯度。

```
y.backward()
print(x.grad)
```

```
tensor([ 0.,  4.,  8., 12.])
```

函数y= 2x^Tx关于x的梯度应为4x,

并计算x的另一个函数。

```
print(x.grad == 4 * x)
# 在默认情况下，PyTorch会累积梯度，我们需要清除之前的值
x.grad.zero_()
y = x.sum()
y.backward()
print(x.grad)
```

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

2.4.2. 非标量变量的反向传播

当y不是标量时，向量y关于向量x的导数的最自然解释是一个矩阵。 对于高阶和高维的y和x，求导的结果可以是一个高阶张量。

然而，虽然这些更奇特的对象确实出现在高级机器学习中（包括深度学习中），但当调用向量的反向计算时，我们通常会试图计算一批训练样本中每个组成部分的损失函数的导数。 这里，我们的目的不是计算微分矩阵，而是单独计算批量中每个样本的偏导数之和。

```
# 对非标量调用backward需要传入一个gradient参数，该参数指定微分函数关于self的梯度。
# 本例只想求偏导数的和，所以传递一个1的梯度是合适的
x.grad.zero_()
y = x * x
# 等价于y.backward(torch.ones(len(x)))
y.sum().backward()
x.grad
```

```
tensor([0., 2., 4., 6.])
```

2.4.3. 分离计算

有时，我们希望将某些计算移动到记录的计算图之外。 例如，假设y是作为x的函数计算的，而z则是作为y和x的函数计算的。 想象一下，我们想计算z关于x的梯度，但由于某种原因，希望将y视为一个常数，并且只考虑到x在y被计算后发挥的作用。

这里可以分离y来返回一个新变量u，该变量与y具有相同的值， 但丢弃计算图中如何计算y的任何信息。 换句话说，梯度不会向后流经u到x。 因此，下面的反向传播函数计算z=u*x关于x的偏导数，同时将u作为常数处理， 而不是z=x*x关于x的偏导数。

```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

z.sum().backward()
x.grad == u
```

由于记录了y的计算结果，我们可以随后在y上调用反向传播， 得到y=x*x关于的x的导数，即2x。

```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

2.4.4. Python控制流的梯度计算

使用自动微分的一个好处是： 即使构建函数的计算图需要通过Python控制流（例如，条件、循环或任意函数调用），我们仍然可以计算得到的变量的梯度。 在下面的代码中，while循环的迭代次数和if语句的结果都取决于输入a的值。

```
def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

```
# 让我们计算梯度。
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

我们现在可以分析上面定义的f函数。 请注意，它在其输入a中是分段线性的。 换言之，对于任何a，存在某个常量标量k，使得f(a)=k*a，其中k的值取决于输入a，因此可以用d/a验证梯度是否正确。

```
a.grad == d / a
```

深度学习框架可以自动计算导数：我们首先将梯度附加到想要对其计算偏导数的变量上，然后记录目标值的计算，执行它的反向传播函数，并访问得到的梯度。