4 Pytorch简介

中国矿业大学 计算机科学与技术学院

概述

简洁优雅

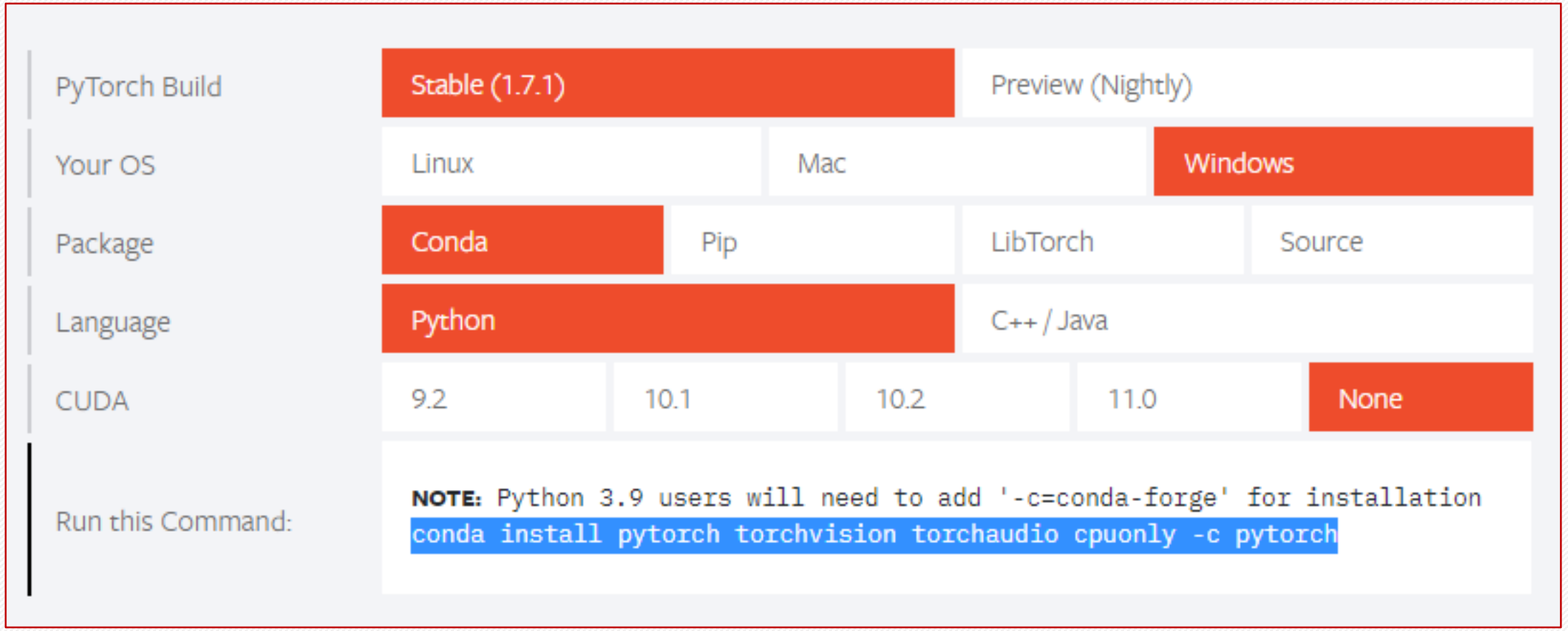
容易上手

速度快

发展趋势

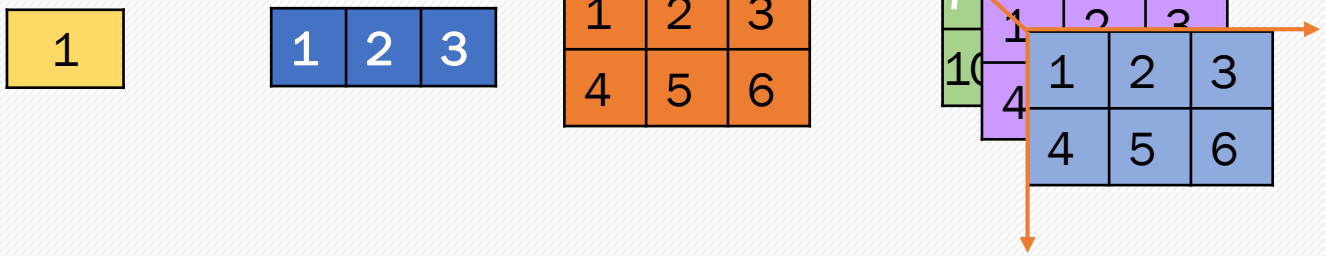
anaconda下安装pytorch

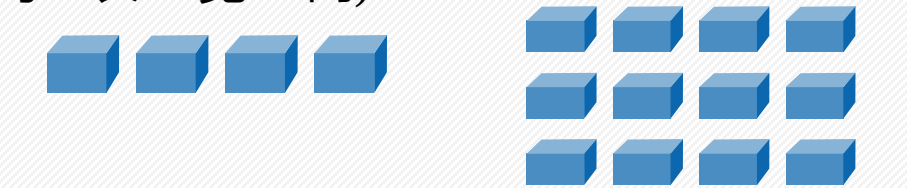
■**https://pytorch.org/**



什么是Tensor（张量）？

	标量	向量	矩阵	三维矩阵	四维矩阵	五维矩阵
数学:						
用途:	类标签	特征向量	特征矩阵	多通道图像	一组RGB图像(批大小×长 × 宽 × 高)	一组RGB视频(批大小×时间 ×长 × 宽 × 高)





值:	1	[1, 2, 3]	[[1, 2, 3], [4, 5, 6]]	[[[1,2,3],[4,5,6]], [[7,8,9],[10,11,12]]]
维度:	0	1	2	3
尺寸:	torch.Size([])	torch.Size([3])	torch.Size([2, 3])	torch.Size([2, 2, 3])
元素数:	1	3	6	12

a = torch.tensor(1)

b= torch.tensor([1, 2, 3])

c = torch.tensor([[1, 2, 3], [4, 5, 6]])

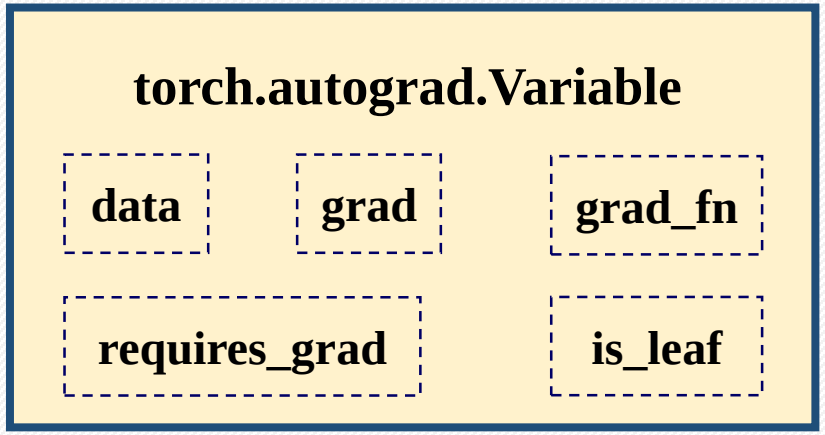
d= torch.tensor([[[1, 2, 3],[4, 5, 6]], [[7, 8, 9],[10, 11,12]]])

```
import torch
print(a)
print(a.dtype)
print(a.device)
print(a.shape)
print(b.size())
print(b.dim())
print(c.ndim)
print(c.numel())
print(c.nelement())
```

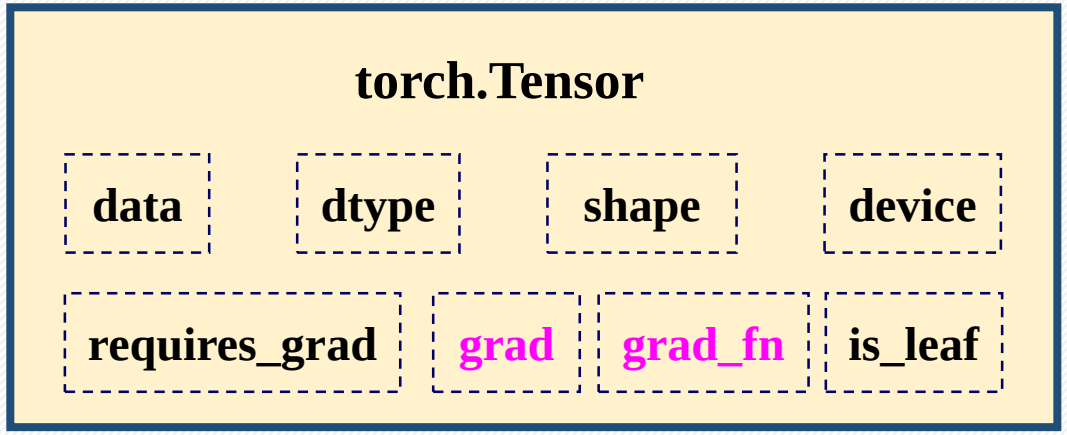
```
tensor(1)
torch.int64
cpu
torch.Size([])
torch.Size([3])
1
2
6
6
```

Tensor 与 Variable

PyTorch 0.4.0 之前，有Variable数据类型，主要用于封装 Tensor，实现自动求导。



PyTorch 0.4.0 之后，Variable 并入Tensor。



Tensor数据类型

序号	描述	数据类型	CPU tensor	GPU tensor
1	32浮点数	torch.float32或torch.float	torch.FloatTensor	torch.cuda.FloatTensor
2	64浮点数	torch.float64或torch.double	torch.DoubleTensor	torch.cuda.DoubleTensor
3	64位复数	torch.complex64或torch.cfloat		
4	128位复数	torch.complex128或torch.cdouble		
5	16浮点数1	torch.float16或torch.half	torch.HalfTensor	torch.cuda.HalfTensor
6	16浮点数2	torch.bfloat16	torch.BFloat16Tensor	torch.cuda.BFloat16Tensor
7	8位无符号整数	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
8	8位有符号整数	torch.int8	torch.CharTensor	torch.cuda.CharTensor
9	16位有符号整数	torch.int16或torch.short	torch.ShortTensor	torch.cuda.ShortTensor
10	32位有符号整数	torch.int32或torch.int	torch.IntTensor	torch.cuda.IntTensor
11	64位有符号整数	torch.int64或torch.long	torch.LongTensor	torch.cuda.LongTensor
12	布尔数据	torch.bool	torch.BoolTensor	torch.cuda.BoolTensor

创建张量

1）利用已有数据直接创建张量

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

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

```
b= torch.tensor((1, 2, 3))
```

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

```
c=torch.tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]])
```

```
tensor([[ 0.1000,  1.2000],
        [ 2.2000,  3.1000],
        [ 4.9000,  5.2000]])
```

```
d = torch.tensor(np.array([1, 2, 3]))
```

```
tensor([1, 2, 3], dtype=torch.int32)
```

```
e=torch.tensor([3])
```

```
tensor([3])
```

```
f=torch.tensor(3)
```

```
tensor(3)
```

```
g=torch.tensor([])
```

```
tensor([])
```

```
h=torch.tensor()
```

```
TypeError!
```

利用函数创建

```
a=torch.ones(2, 3)
```

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

```
b=torch.ones(5)
```

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

```
c = torch.zeros(2, 3, dtype=torch.long)
```

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

```
d=torch.ones_like(a)
```

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

```
e= torch.empty(2, 3)
```

```
tensor([[1.6689e-07, 1.0186e-11, 2.1441e-07],
        [2.1353e+20, 1.0720e-08, 2.0450e+23]])
```

```
f= torch.eye(3)
```

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

```
g=torch.full((2, 3), 3.141592)
```

```
tensor([[3.1416, 3.1416, 3.1416],
        [3.1416, 3.1416, 3.1416]])
```

利用函数创建（续）

```
a = torch.arange(1, 4)
```

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

```
b = torch.arange(1, 4, 0.5)
```

```
tensor([1.0000, 1.5000, 2.0000,
        2.5000, 3.0000, 3.5000])
```

```
c = torch.linspace(3, 10, steps=5)
```

```
tensor([3.0000,  4.7500,  6.5000,  8.2500,  10.0000])
```

```
d=torch.logspace(start=-10, end=10, steps=5)
```

```
tensor([ 1.0000e-10,  1.0000e-05,
         1.0000e+00,  1.0000e+05,  1.0000e+10])
```

用函数创建含有随机值的张量

```
torch.manual_seed(123)
```

```
a=torch.rand(2, 3)
```

```
tensor([[ 0.8237,  0.5781,  0.6879],
        [ 0.3816,  0.7249,  0.0998]])
```

```
b=torch.randn(2, 3)
```

```
tensor([[ 1.5954,  2.8929, -1.0923],
        [ 1.1719, -0.4709, -0.1996]])
```

```
c=torch.rand_like(a)
```

```
tensor([[0.2429, 0.2219],
        [0.7826, 0.7959],
        [0.8887, 0.9916]])
```

```
d=torch.randint(3, 10, (2, 2))
```

```
tensor([[4, 5],
        [6, 7]])
```

```
e=torch.randint(10, (2, 2))
```

```
tensor([[0, 2],
        [5, 5]])
```

```
f=torch.randperm(4)
```

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

```
g= torch.rand(100, 28, 28, device='cpu')
```

```
……
```

用函数创建含有随机值的张量

```
torch.normal(mean=0.0,std=torch.tensor(1.0))
```

```
tensor(-0.0124)
```

```
torch.normal(3, 0.1, (3, 4))
```

```
tensor([[2.9346, 3.0394, 3.1113, 3.0019],
        [3.1380, 3.0129, 3.0432, 2.9252],
        [3.0169, 3.0448, 2.9652, 2.9941]])
```

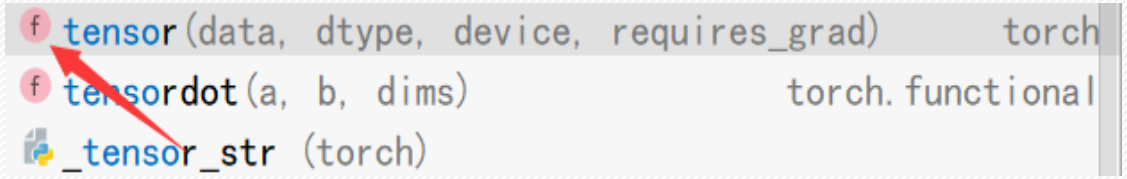
构造张量的函数

函数名	张量中元素的内容
torch.tensor()	内容为传入的数据
torch.zeros()、torch.zero_like()	各元素全为0
torch.ones()、torch.ones_like()	各元素全为l
torch.full()、torch.full_like()	各元素全为指定的值
torch.empty()、torch.empty_like()	未指定元素的值
torch.eye()	对角线为1，其他为0
torch.arange()、torch.range()、torch.linspace()	各元素等差
torch.logspaceO	各元素等比
torch.rand()、torch.rand_like()	各元素独立服从标准均匀分布
torch.randn()、torch.randn_like()、torch.normal()	各元素独立服从标准正态分布
torch.randint()、torch.randint_like()	各元素独立服从离散均匀分布
torch.bernoulli()	{0,1}上的两点分布
torch.normal()	元素服从指定均值和方差的正态分布
torch.multrinomial()	{0, l, …, n-l}上的多项式分布
torch.randperm()	各元素为{0,1,⋯,n-1}的一个随机排列

torch.tensor与torch.Tensor的区别

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

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




```
a.type()
a.dtype
```

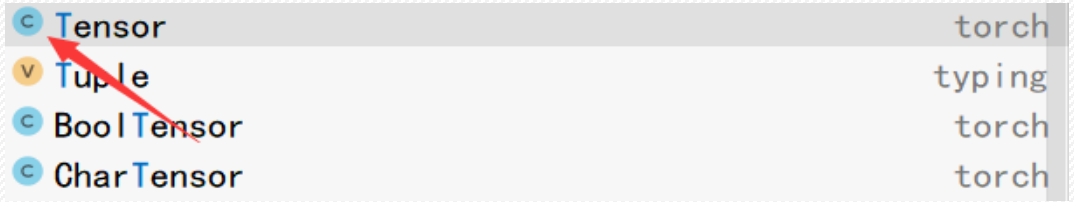
```
torch.LongTensor
torch.int64
```

```
c=torch.Tensor()
d=torch.tensor()
```

```
tensor([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: tensor() missing 1 required positional arguments: "data"
```

```
b = torch.Tensor([1, 2, 3])
```

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




```
b.type()
b.dtype
```

```
torch.FloatTensor
torch.float32
```

```
c=torch.Tensor(3)
d=torch.tensor(3)
```

```
tensor([9.1477e-41, 4.4766e+00, 1.1210e-44])
tensor(3)
```

访问元素

```
x = torch.Tensor([[1,2,3], [4,5,6], [7,8,9]])
print(x[0][2])
print(x[0, 2])
print(x[0])
print(x[:, 2])
print(x[1:2,1:2])
print(x>5)
print(x[x>5])
```

```
tensor(3.)
tensor(3.)
tensor([1., 2., 3.])
tensor([3., 6., 9.])
tensor([[5.]])
tensor([[False, False, False],
        [False, False,  True],
        [True,  True,  True]])
tensor([6., 7., 8., 9.])
```

```
x = torch.randn(1)
print(x)
print(x.item())
```

```
tensor([2.3466])
2.346638202667236 3
```

```
x = torch.randn(2)
print(x)
print(x.item())
```

```
tensor([ 1.4132, -0.0699])
---------------------------------------------------------------------------
ValueError                Traceback (most recent call last)
<ipython-input-55-783106dbbcef> in <module>
      1 x = torch.randn(2)
      2 print(x)
----> 3 print(x.item())

ValueError: only one element tensors can be converted to Python scalars
```

访问元素

```
a = torch.tensor([[0.6, 0.0, 0.0, 0.0],
                  [0.0, 0.4, 0.0, 0.0],
                  [0.0, 0.0, 1.2, 0.0],
                  [0.0, 0.0, 0.0, -0.7]])
```

```
b = torch.nonzero(a)
```

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

```
c = torch.nonzero(a, as_tuple=True)
```

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

```
d = a[a.nonzero(as_tuple=True)]
```

```
tensor([ 0.6000,  0.4000,  1.2000, -0.7000])
```

```
torch.where(a>0.5, torch.full_like(a, 60), a)
```

```
tensor([[60.0000,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.4000,  0.0000,  0.0000],
        [ 0.0000,  0.0000, 60.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000, -0.7000]])
```

索引

```
t=torch.arange(9).reshape(3,3)
#注意idx的dtype不能指定为torch.float
idx=torch.tensor([0, 2], dtype=torch.long)
#取出第0行和第2行
row=torch.index_select(t, dim=0, index=idx)
col=torch.index_select(t, dim=1, index=idx)
```

t:

```
tensor([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]])
```

row

```
tensor([[0, 1, 2],
        [6, 7, 8]])
```

col

```
tensor([[0, 2],
        [3, 5],
        [6, 8]])
```

```
x = torch.randn(3, 4)
```

```
tensor([[ 0.3552, -2.3825, -0.8297,  0.3477],
        [-1.2035,  1.2252,  0.5002,  0.6248],
        [ 0.1307, -2.0608,  0.1244,  2.0139]])
```

```
mask = x.ge(0.5)
等价于
mask=torch.ge(x, 0.5)
```

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

```
torch.masked_select(x, mask)
```

```
tensor([ 1.2252,  0.5002,  0.6248,  2.0139])
```

类型转换

```
a = torch.Tensor(2, 3)
print(a.type())
```

```
torch.FloatTensor
```

```
b = a.double()
print(b.dtype)
```

```
torch.float64
```

```
c = a.type(torch.IntTensor)
print(c.dtype)
```

```
torch.int32
```

```
d = a.type_as(b)
print(d.dtype)
```

```
torch.float64
```

Tensor与numpy的转换

从numpy到tensor的转换：torch.from_numpy(a)
从tensor到numpy的转换：a.numpy()

```
a=np.ones([2, 3])
```

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

```
b1=torch.from_numpy(
b2=torch.as_tensor(a)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]],
dtype=torch.float64)
```

```
b1.numpy()
```

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

```
b1.tolist()
```

```
[[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]]
```

数据拷贝 vs 共享内存

■ 数据拷贝

```
a = np.ones((3, 3))
print('dtype of ndarray:', a.dtype)
t= torch.tensor(a)
print('dtype of tensor:', t.dtype)
t[0, 0]=-100
print('numpy object, a:', a)
print("tensor object, t", t)
```

```
dtype of ndarray: float64
dtype of tensor: torch.float64
numpy object, a: [[1. 1. 1.]
                  [1. 1. 1.]
                  [1. 1. 1.]]
tensor object, t tensor([[-100.,    1.,    1.],
        [   1.,    1.,    1.],
        [   1.,    1.,    1.]], dtype=torch.float64)
```

■ 共享内存

```
a = np.ones((3, 3))
print('dtype of ndarray:', a.dtype)
t= torch.from_numpy(a)
print('dtype of tensor:', t.dtype)
t[0, 0]=-100
print('numpy object, a:', a)
print("tensor object, t", t)
```

```
dtype of ndarray: float64
dtype of tensor: torch.float64
numpy object, a: [[-100.    1.    1.]
                  [   1.    1.    1.]
                  [   1.    1.    1.]]
tensor object, t tensor([[-100.,    1.,    1.],
                         [   1.,    1.,    1.],
                         [   1.,    1.,    1.]],
dtype=torch.float64)
```

Tensor的内存共享

2）切片

```
a=torch.Tensor([[1, 2], [3, 4]])
c=a[1,:]
c[0]=100
print(a)
print(c)
```

a
```
tensor([[  1.,   2.],
        [100.,   4.]])
```

c
```
tensor([100.,   4.])
```

3）原地操作符

```
a=torch.Tensor([[1, 2], [3, 4]])
a.add_(a)
a.resize_(4)
```

```
tensor([[2., 4.],
        [6., 8.]])
```

```
tensor([2., 4., 6., 8.])
```

内存分配

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
y = y + x
print(id(y) == id_before)
```

False

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
y[:] = y + x
print(id(y) == id_before)
```

True

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
torch.add(x, y, out=y)
print(id(y) == id_before)
```

True

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
y += x
print(id(y) == id_before)
```

True

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
y.add_(x)
print(id(y) == id_before)
```

True

变形

```
x= torch.rand(4, 3)
x_cp = x.clone().view(3,4)
x -= 1
print(x)
print(x_cp)  # x_cp无变化
```

```
tensor([[-0.6106, -0.0303, -0.6859],
        [-0.2682, -0.2081, -0.5147],
        [-0.0501, -0.3383, -0.0395],
        [-0.5852, -0.3352, -0.2284]])
tensor([[0.3894, 0.9697, 0.3141, 0.7318],
        [0.7919, 0.4853, 0.9499, 0.6617],
        [0.9605, 0.4148, 0.6648, 0.7716]])
```

```
x.size()
y=x.view(2, 6)
y.size()
z=x.reshape(-1, 1)
z.size()
```

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

torch.Size([2, 6])

torch.Size([12, 1])
```

```
x += 6
print(x)
print(y) # y也加了6
```

```
tensor([[5.3894, 5.9697, 5.3141],
        [5.7318, 5.7919, 5.4853],
        [5.9499, 5.6617, 5.9605],
        [5.4148, 5.6648, 5.7716]])
tensor([[5.3894, 5.9697, 5.3141, 5.7318, 5.7919, 5.4853],
        [5.9499, 5.6617, 5.9605, 5.4148, 5.6648, 5.7716]])
```

张量计算——基本运算

```
x = torch.rand(5, 3)
```

```
tensor([[0.0494, 0.9550, 0.4300],
        [0.1464, 0.9270, 0.4361],
        [0.9718, 0.9294, 0.5718],
        [0.3580, 0.7507, 0.8444],
        [0.7232, 0.1770, 0.9898]])
```

```
y = torch.ones(5, 3)
```

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

等价

```
z = x + y
```

```
z = torch.add(x, y)
```

```
z = torch.empty(5, 3)
torch.add(x, y, out=z)
```

```
y.add_(x)
```

```
tensor([[1.0494, 1.9550, 1.4300],
        [1.1464, 1.9270, 1.4361],
        [1.9718, 1.9294, 1.5718],
        [1.3580, 1.7507, 1.8444],
        [1.7232, 1.1770, 1.9898]])
```

torch的部分常用操作

add()	clamp()	sigmoid()	prod() cumprod()
mul()	clamp_min() clamp_max()	sign()	sum() cumsum()
div()	frac()	sqrt()	max()
fmod() remainder()	neg()	rsqrt()	min()
abs()	reciprocal()	dist()	sin()
ceil()	log	mean() median()	cos()
floor	pow() **	std()	tan()
round()	exp()	norm()	arcsin()

张量计算——基本运算

```
A=torch.arange(6).reshape(2,3)
torch.sqrt(A)
```

```
tensor([[0.0000, 1.0000, 1.4142],
        [1.7321, 2.0000, 2.2361]])
```

```
torch.exp(A)
```

```
tensor([[  1.0000,   2.7183,   7.3891],
        [ 20.0855,  54.5981, 148.4132]])
```

```
torch.log(A)
```

```
tensor([[  -inf, 0.0000, 0.6931],
        [1.0986, 1.3863, 1.6094]])
```

```
torch.pow(A,3)
```

```
tensor([[  0,   1,   8],
        [ 27,  64, 125]])
```

```
torch.clamp_max(A, 4)
```

```
tensor([[0, 1, 2],
        [3, 4, 4]])
```

```
torch.clamp_min(A, 2)
```

```
tensor([[2, 2, 2],
        [3, 4, 5]])
```

```
torch.clamp(A, 2.5, 4)
```

```
tensor([[2, 2, 2],
        [3, 4, 4]])
```

三角函数

```
A=torch.arange(12).reshape(3,4)
```

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

```
torch.tril(A)
```

```
tensor([[ 0,  0,  0,  0],
        [ 4,  5,  0,  0],
        [ 8,  9, 10,  0]])
```

```
torch.tril(A, diagonal=1)
```

```
tensor([[ 0,  1,  0,  0],
        [ 4,  5,  6,  0],
        [ 8,  9, 10, 11]])
```

```
torch.triu(A)
```

```
tensor([[ 0,  1,  2,  3],
        [ 0,  5,  6,  7],
        [ 0,  0, 10, 11]])
```

```
torch.diag(A)
```

```
tensor([ 0,  5, 10])
```

```
torch.diag(A,diagonal=1)
```

```
tensor([ 1,  6, 11])
```

```
torch.diag(torch.tensor([0,  5, 10]))
```

```
tensor([[ 0,  0,  0],
        [ 0,  5,  0],
        [ 0,  0, 10]])
```

张量计算——比较大小

函数	功能
torch.allclose()	比较两个元素是否接近
torch.eq()	逐元素比较是否相等
torch.equal()	判断两个张量是否具有相同的形状和元素
torch.ge()	逐元素比较大于等于
torch.gt()	逐元素比较大于
torch.le()	逐元素比较小于等于
torch.lt()	逐元素比较小于
torch.ne()	逐元素比较不等于
torch.isnan()	逐元素判断是否为缺失值

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

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

```
torch.equal(torch.tensor([1, 2]), torch.tensor([1, 2]))
```

```
True
```

代数运算

```
a = torch.Tensor([1, 2, 3])
b = torch.Tensor([2, 3, 4])
torch.dot(a, b)
```

```
tensor(20.)
```

```
a = torch.Tensor([[1,2,3], [2,3,4], [3,4,5]])
b = torch.Tensor([1,2,3])
torch.mv(a, b)
torch.matmul(a, b)
```

```
tensor([14., 20., 26.])
```

```
a = torch.Tensor([[1,2,3], [2,3,4], [3,4,5]])
b = torch.Tensor([[2,3,4], [3,4,5],[4,5,6]])
torch.mm(a, b)
torch.matmul(a, b)
```

```
tensor([[20., 26., 32.],
        [29., 38., 47.],
        [38., 50., 62.]])
```

代数运算

```
A = torch.arange(1., 10.).view(3, 3)
```

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

```
t = torch.trace(A)
```

```
tensor(15.)
```

```
torch.linalg.svd(A)
```

```
torch.return_types.linalg_svd(
U=tensor([[-0.2148,  0.8872,  0.4082],
          [-0.5206,  0.2496, -0.8165],
          [-0.8263, -0.3879,  0.4082]]),
S=tensor([1.6848e+01, 1.0684e+00, 2.3721e-07]),
Vh=tensor([[-0.4797, -0.5724, -0.6651],
           [-0.7767, -0.0757,  0.6253],
           [-0.4082,  0.8165, -0.4082]]))
```

```
b = torch.rand(4, 4)
```

```
tensor([[0.2026, 0.6480, 0.8136, 0.4816],
        [0.8259, 0.7942, 0.9691, 0.7807],
        [0.3255, 0.2887, 0.1268, 0.9332],
        [0.6734, 0.2121, 0.9896, 0.0847]])
```

```
b1 = torch.inverse(b)
b2 = torch.linalg.inv(b)
```

```
tensor([[-2.2495,  2.0649, -0.5573, -0.1014],
        [-0.0841,  2.6503, -1.9674, -2.2743],
        [ 1.4968, -1.8630,  0.6480,  1.5214],
        [ 0.6072, -1.2871,  1.7866,  0.5323]])
```

统计相关

```
a = torch.randn(3, 4)
```

```
tensor([[ 0.0372, -1.4255,  0.5897,  0.0728],
        [-0.7950,  0.1134, -1.0111, -0.7277],
        [-0.0435,  1.5029,  1.1200,  0.2160]])
```

```
torch.max(a)
```

```
tensor(1.5029)
```

```
torch.max(a, dim=1)
```

```
torch.return_types.max(
  values=tensor([0.5897, 0.1134, 1.5029]),
  indices=tensor([2, 1, 1]))
```

```
torch.max(a, dim=0)
```

```
torch.return_types.max(
  values=tensor([0.0372, 1.5029, 1.1200, 0.2160]),
  indices=tensor([0, 2, 2, 2]))
```

```
torch.argmax(a)
```

```
tensor(9)
```

```
torch.argmax(a, dim=1)
```

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

```
torch.argmax(a, dim=0)
```

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

统计相关

```
A=torch.randint(1, 10, size=(3,4))
```

```
tensor([[4, 7, 6, 5],
        [1, 3, 2, 5],
        [9, 9, 1, 6]])
```

```
torch.topk(A, 2)
```

```
torch.return_types.topk(
values=tensor([[7, 6],
               [5, 3],
               [9, 9]]),
indices=tensor([[1, 2],
                [3, 1],
                [0, 1]]))
```

```
torch.topk(A, 2, dim=0)
```

```
torch.return_types.topk(
values=tensor([[9, 9, 6, 6],
               [4, 7, 2, 5]]),
indices=tensor([[2, 2, 0, 2],
                [0, 0, 1, 0]]))
```

```
torch.kthvalue(A,3)
```

```
torch.return_types.kthvalue(
  values=tensor([6, 3, 9]),
  indices=tensor([2, 1, 0]))
```

组合

```
a = torch.tensor([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10,11,12]])
b = torch.tensor([[13, 14, 15, 16],
                  [17, 18, 19, 20],
                  [21, 22, 23, 24]])
c0=torch.cat((a, b), 0)
c1=torch.cat((a, b), 1)

s0=torch.stack((a, b), 0)
s1=torch.stack((a, b), 1)
s2=torch.stack((a, b), 2)
```

a

```
tensor([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])
```

dim:(3,4)

b

```
tensor([[13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]])
```

dim:(3,4)

c0 dim:(6,4)

```
tensor([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12],
        [13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]])
```

c1 dim:(3,8)

```
tensor([[ 1,  2,  3,  4, 13, 14, 15, 16],
        [ 5,  6,  7,  8, 17, 18, 19, 20],
        [ 9, 10, 11, 12, 21, 22, 23, 24]])
```

s0 dim:(2,3,4)

```
tensor([[[ 1,  2,  3,  4],
         [ 5,  6,  7,  8],
         [ 9, 10, 11, 12]],

        [[13, 14, 15, 16],
         [17, 18, 19, 20],
         [21, 22, 23, 24]]])
```

s1 dim:(3,2,4)

```
tensor([[[ 1,  2,  3,  4],
         [13, 14, 15, 16]],

        [[ 5,  6,  7,  8],
         [17, 18, 19, 20]],

        [[ 9, 10, 11, 12],
         [21, 22, 23, 24]]])
```

s2 dim:(3,4,2)

```
tensor([[[ 1, 13],
         [ 2, 14],
         [ 3, 15],
         [ 4, 16]],

        [[ 5, 17],
         [ 6, 18],
         [ 7, 19],
         [ 8, 20]],

        [[ 9, 21],
         [10, 22],
         [11, 23],
         [12, 24]]])
```

组合（续）

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
torch.hstack((a,b))
```

```
tensor([1, 2, 3, 4, 5, 6])
```

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
torch.vstack((a,b))
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
a = torch.tensor([[1],[2],[3]])
b = torch.tensor([[4],[5],[6]])
torch.hstack((a,b))
```

```
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

```
a = torch.tensor([[1],[2],[3]])
b = torch.tensor([[4],[5],[6]])
torch.vstack((a,b))
```

```
tensor([[1],
        [2],
        [3],
        [4],
        [5],
        [6]])
```

分块

```
a= torch.Tensor([[1, 2, 3], [4, 5, 6]])
c0=torch.chunk(a, 2, 0)
c1=torch.chunk(a, 2, 1)
```

a
```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

c0
```
(tensor([[1., 2., 3.]]), tensor([[4., 5., 6.]]))
```

c1
```
(tensor([[1., 2.],
         [4., 5.]]),
 tensor([[3.],
         [6.]]))
```

Note: If such division is not possible, this function may return less than the specified number of chunks.

```
s0=torch.split(a, 2, 0)
s1=torch.split(a, 2, 1)
s2=torch.split(a, [1, 2], 1)
```

s0
```
(tensor([[1., 2., 3.],
         [4., 5., 6.]]),)
```

s1
```
(tensor([[1., 2.],
         [4., 5.]]),
 tensor([[3.],
         [6.]]))
```

s2
```
(tensor([[1.],
         [4.]]),
 tensor([[2., 3.],
         [5., 6.]]))
```

transpose()和permute()

```
x = torch.randn(2, 3)
```

```
tensor([[ 1.0028, -0.9893,  0.5809],
        [-0.1669,  0.7299,  0.4942]])
```

```
torch.transpose(x, 0, 1)
```

```
tensor([[ 1.0028, -0.1669],
        [-0.9893,  0.7299],
        [ 0.5809,  0.4942]])
```

```
x = torch.randn(2, 3, 5)
x.size()
```

```
torch.Size([2, 3, 5])
```

```
x.permute(2, 0, 1).size()
```

```
torch.Size([5, 2, 3])
```

```
y=torch.tensor([[1,2,3],
                [4,5,6]])
y.t()
```

```
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

Tensor.permute(a, b, c, d, ...)：可以对任意高维矩阵进行转置：

```
torch.randn(2, 3, 4, 5).permute(3, 2, 0, 1).shape
```

```
torch.randn(2, 3, 4, 5).transpose(3, 0).transpose(2, 1).transpose(3, 2).shape
```

```
torch.Size([5, 4, 2, 3])
```

squeeze()与unsqueeze()

```
x = torch.zeros(2, 1, 3, 1, 5)
x.size()
```
```
torch.Size([2, 1, 3, 1, 5])
```

```
y = torch.squeeze(x)
y.size()
```
```
torch.Size([2, 3, 5])
```

```
y = torch.squeeze(x, 0)
y.size()
```
```
torch.Size([2, 1, 3, 1, 5])
```

```
y = torch.squeeze(x, 1)
y.size()
```
```
torch.Size([2, 3, 1, 5])
```

```
x = torch.tensor([1, 2, 3, 4])
y = torch.unsqueeze(x, 0)
```
```
tensor([1, 2, 3, 4])
tensor([[1, 2, 3, 4]])
```

```
x.size()
y.size()
```
```
torch.Size([4])
torch.Size([1, 4])
```

```
z=torch.unsqueeze(x, 1)
```
```
tensor([[ 1],
        [ 2],
        [ 3],
        [ 4]])
```

```
z.size()
```
```
torch.Size([4, 1])
```

扩展

```
a = torch.tensor([[1], [2], [3]])
a.size()
a1=a.expand(3, 4)
a2=a.expand(-1, 5)
```

a1

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

a

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

a.size()

```
torch.Size([3, 1])
```

a2

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

```
b=torch.tensor([[2, 2], [3, 3], [5, 5]])
b.size()
b0=a.expand_as(b)
```

b

```
tensor([[2, 2],
        [3, 3],
        [4, 4]])
```

b.size()

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

b0

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

a

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

```
x = torch.tensor([1, 2, 3])
y = x.repeat(4, 2)
```

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

```
x = torch.tensor([1, 2, 3])
z = x.repeat(4, 2, 1)
```

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

        [[1, 2, 3],
         [1, 2, 3]],

        [[1, 2, 3],
         [1, 2, 3]],

        [[1, 2, 3],
         [1, 2, 3]]])
```

广播机制

```
a=torch.ones(3, 1, 2)
b=torch.ones(2, 1)
(a+b).size()
```

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

```
x = torch.arange(1, 3).view(1, 2)
y = torch.arange(1, 4).view(3, 1)
print(x + y)
```

x

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

y

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

x+y

```
tensor([[2, 3],
        [3, 4],
        [4, 5]])
```

```
c=torch.ones(5, 3)
(a+c).size()
```

```
RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at non-singleton dimension 2
```

```
d=torch.ones(5, 2)
(a+d).size()
```

```
torch.Size([3, 5, 2])
```

25.txt

排序

```
a = torch.randn(3, 4)
```

a

```
tensor([[-0.7017, -1.3189,  0.7294, -0.9685],
        [ 0.6198,  0.0102, -0.3996,  0.8234],
        [ 0.1210, -0.8738, -1.1875, -0.9900]])
```

```
s0, i0 = torch.sort(a, 0)
```

s0

```
tensor([[-0.7017, -1.3189, -1.1875, -0.9900],
        [ 0.1210, -0.8738, -0.3996, -0.9685],
        [ 0.6198,  0.0102,  0.7294,  0.8234]])
```

i0

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

```
s1, i1 = torch.sort(a)
```

s0

```
tensor([[-1.3189, -0.9685, -0.7017,  0.7294],
        [-0.3996,  0.0102,  0.6198,  0.8234],
        [-1.1875, -0.9900, -0.8738,  0.1210]])
```

i0

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

向量化提高计算效率

```
from time import time
LENGTH=1000
a = torch.ones(LENGTH, LENGTH)
b = torch.ones(LENGTH, LENGTH)
```

```
start = time()
c = a + b
print(time() - start)
```

```
0.00699472427368164
```

```
start = time()
c = torch.zeros(LENGTH, LENGTH)
for i in range(LENGTH):
    for j in range(LENGTH):
        d[i] = a[i] + b[i]
print(time() - start)
```

```
44.121182680130005
```

GPU张量

Tensor与Numpy数组的最大不同：Tensor可以在**GPU**上运算

```
x = np.ones([2, 3])
y = np.ones([2, 3])
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
print(x + y)
```

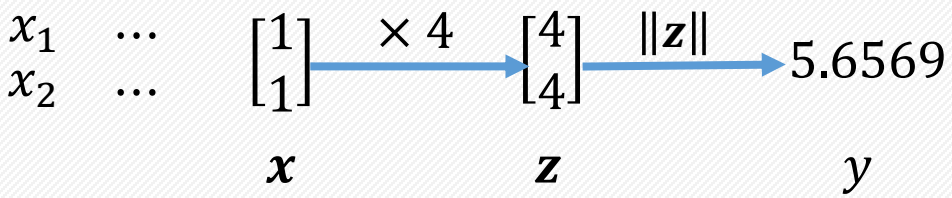
例，a fast calculator

```python
import numpy as np
import torch

# Task: compute matrix multiplication C = AB
d = 3000

# using numpy
A = np.random.rand(d, d).astype(np.float32)
B = np.random.rand(d, d).astype(np.float32)
C = A.dot(B)

# using torch with gpu
A = torch.rand(d, d).cuda()
B = torch.rand(d, d).cuda()
C = torch.mm(A, B)
```

350 ms

0.1 ms

因变量y对自变量x求导



从向量x到标量y的计算过程

y关于x的表达式

$$y = \|z\| = \sqrt{z_1^2 + z_2^2} = \sqrt{(4x_1)^2+(4x_2)^2} = 4\sqrt{x_1^2 + x_2^2}$$

y关于x的微分

$$\frac{\partial y}{\partial x_1} = \frac{\partial(4\sqrt{x_1^2 + x_2^2})}{\partial x_1} = 4 \times \frac{1}{2} \times (x_1^2 + x_2^2)^{-\frac{1}{2}} \times 2x_1$$

$$= 4 \times \frac{1}{2} \times 2^{-\frac{1}{2}} \times 2$$

$$= 2.8284$$

$$\frac{\partial y}{\partial x_2} = \frac{\partial(4\sqrt{x_1^2 + x_2^2})}{\partial x_2}$$

$$= 4 \times \frac{1}{2} \times (x_1^2 + x_2^2)^{-\frac{1}{2}} \times 2x_2$$

$$= 2.8284$$

自动求导（微分）

■Pytorch提供自动求导机制autograd，将前向传播的计算记录成计算图，自动完成求导。

■张量在自动微分方面的三个重要属性

1. requires_grad：布尔值，默认为False。为True时，表示该张量需要自动微分。
2. grad：用于存储Tensor的梯度值，且与Tensor同维度。
3. grad_fn：用于存储张量的微分函数，即该Tensor经过了什么操作，用作反向传播的梯度计算，如果由用户创建，则该属性为None。

当叶子节点的requires_grad为True时，信息流经过该节点时，所有中间节点的requires_grad属性都会变成True，只要在输出节点调用反向传播函数。

前向传播

```
x=torch.ones(2)
x.requires_grad
```

```
x tensor([1., 1.])
x.requires_grad False
```

```
x.requires_grad=True
```

```
x tensor([1., 1.], requires_grad=True)
```

```
x.grad
x.grad_fn
```

```
None
None
```

```
z=4*x
```

```
z tensor([4., 4.], grad_fn=<MulBackward0>)
```

```
y=z.norm()
```

```
y tensor(5.6569, grad_fn=<CopyBackwards>)
```

反向传播

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

x.grad

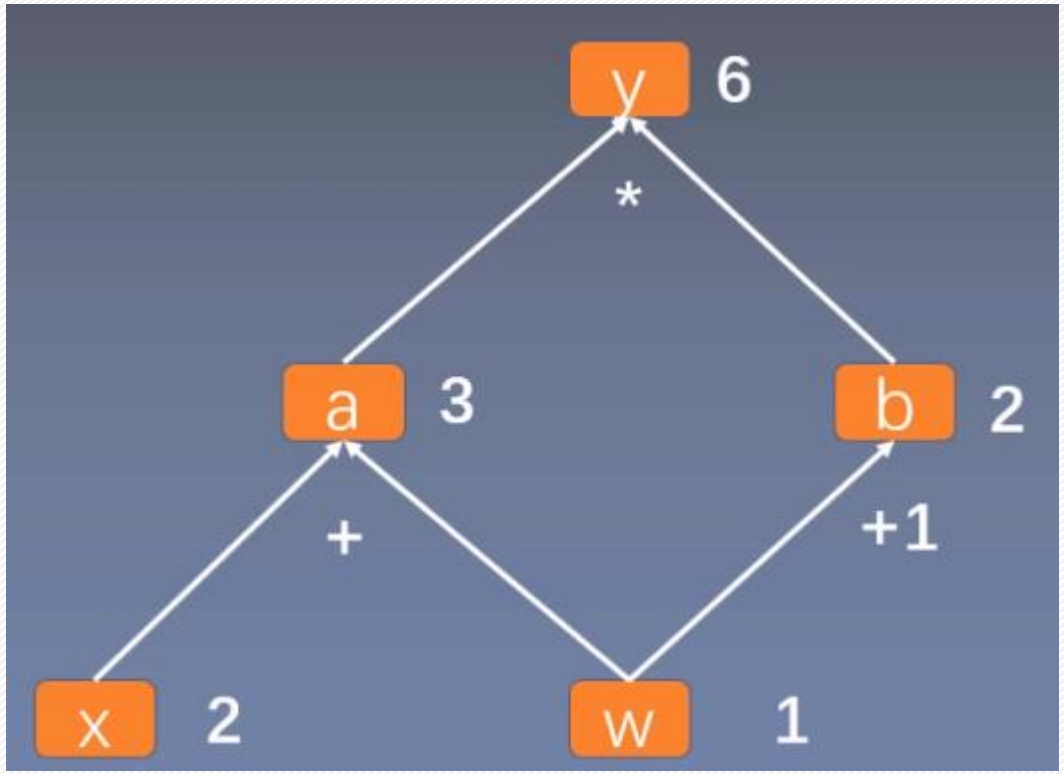
```
tensor([2.8284, 2.8284])
```

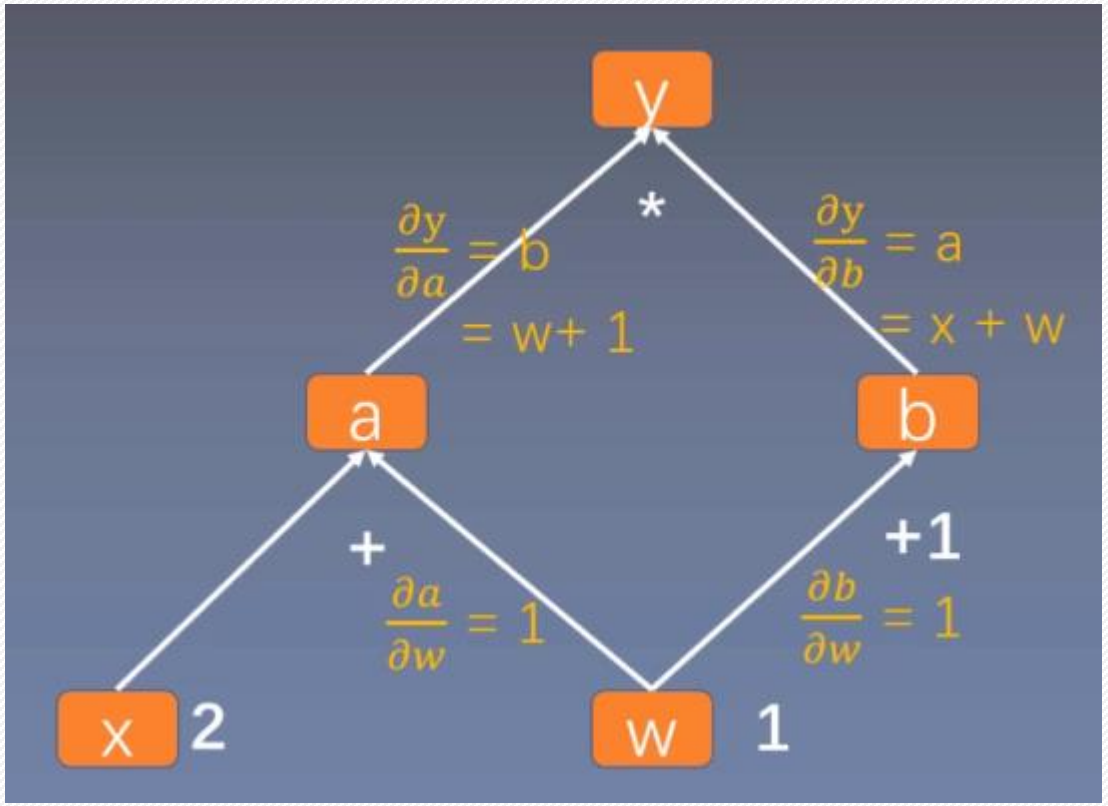
```
z.grad
y.grad
```

```
<ipython-input-79-97cf09c739bf>:1: UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute won't be populated during autograd.backward(). If you indeed want the gradient for a non-leaf Tensor, use .retain_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access the leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more informations.
  z.grad
<ipython-input-79-97cf09c739bf>:2: UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute won't be populated during autograd.backward(). If you indeed want the gradient for a non-leaf Tensor, use .retain_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access the leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more informations.
  y.grad
```

计算图

- 用来描述运算的有向无环图，有两个主要元素：节点 (Node) 和边 (Edge)。节点表示数据，如向量、矩阵、张量。边表示运算，如加减、乘除、卷积等。
- 表达式：$y=(x+w)*(w+1)$，看作$y=a*b$，其中 $a=x+w$，$b=w+1$





自动求导

```
w= torch.tensor([1.], requires_grad=True)
x = torch.tensor([2.])

a = torch.add(w, x)
b = torch.add(w, 1)
y = torch.mul(a, b)
#y求导
y.backward()
#打印w的梯度，即y对w的导数
```

```
#查看梯度
print("w.grad_fn = ", w.grad_fn)
print("x.grad_fn = ", x.grad_fn)
print("a.grad_fn = ", a.grad_fn)
print("b.grad_fn = ", b.grad_fn)
print("y.grad_fn = ", y.grad_fn)
```

```
w.grad_fn =  None
x.grad_fn =  None
a.grad_fn =  <AddBackward0 object at 0x0000015D232CC388>
b.grad_fn =  <AddBackward0 object at 0x0000015D232CC688>
y.grad_fn =  <MulBackward0 object at 0x0000015D232CC388>
```

```
#并查看叶子结点
print("is_leaf:\n", w.is_leaf, x.is_leaf, a.is_leaf, b.is_leaf, y.is_leaf)
#查看梯度
print("gradient:\n", w.grad, x.grad, a.grad, b.grad, y.grad)
```

```
is_leaf:
 True True False False False
gradient:
 tensor([5.]) None None None None
```

自动求导

```
w= torch.tensor([1.], requires_grad=True)
x = torch.tensor([2.])
a = torch.add(w, x)
b = torch.add(w, 1)
y = torch.mul(a, b)
#第一次执行梯度求导
y.backward()
print(w.grad)
#第二次执行梯度求导，出错！
y.backward()
```

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

```
w= torch.tensor([1.], requires_grad=True)
x = torch.tensor([2.])
a = torch.add(w, x)
b = torch.add(w, 1)
y = torch.mul(a, b)
#第一次求导，设置参数，保留计算图
y.backward(retain_graph=True)
print(w.grad)
#第二次求导，成功
y.backward()
print(w.grad)
```

```
tensor([5.])
tensor([10.])
```

```
RuntimeError: Trying to backward through the graph a second time (or directly access saved variables after they have already been freed). Saved intermediate values of the graph are freed when you call .backward() or autograd.grad(). Specify retain_graph=True if you need to backward through the graph a second time or if you need to access saved variables after calling backward.
```

非标量（向量）输出

```
w = torch. tensor([5.], requires_grad=True)
x = torch. tensor([6.])
a = torch.add(w, x)
b = torch.add(w, 3)
y0 = torch.mul(a, b) # y0 = (x+w)*(w+3)
y1 = torch.add(a, b) # y1 = (x+w)+(w+3)
y=[y0, y1]

#把两个loss拼接都到一起
loss = torch.cat(y, dim=0)  # [y0, y1]

#设置两个loss的权重: y0的权重是1,y1的权重是2
grad_tensors = torch.tensor([1., 2.])
loss.backward(gradient=grad_tensors)

print(w.grad) #最终的w的导数由两部分组成。
```

tensor([23.])

$w = 5, x = 6$

$y_0 = (x + w) * (w + 3) = w^2 + (x + 3)w + 3x$

$y_1 = (x + w) + (w + 3) = 2w + x + 3$

$\frac{\partial y_0}{\partial w} = 2w + (x + 3) = 19$

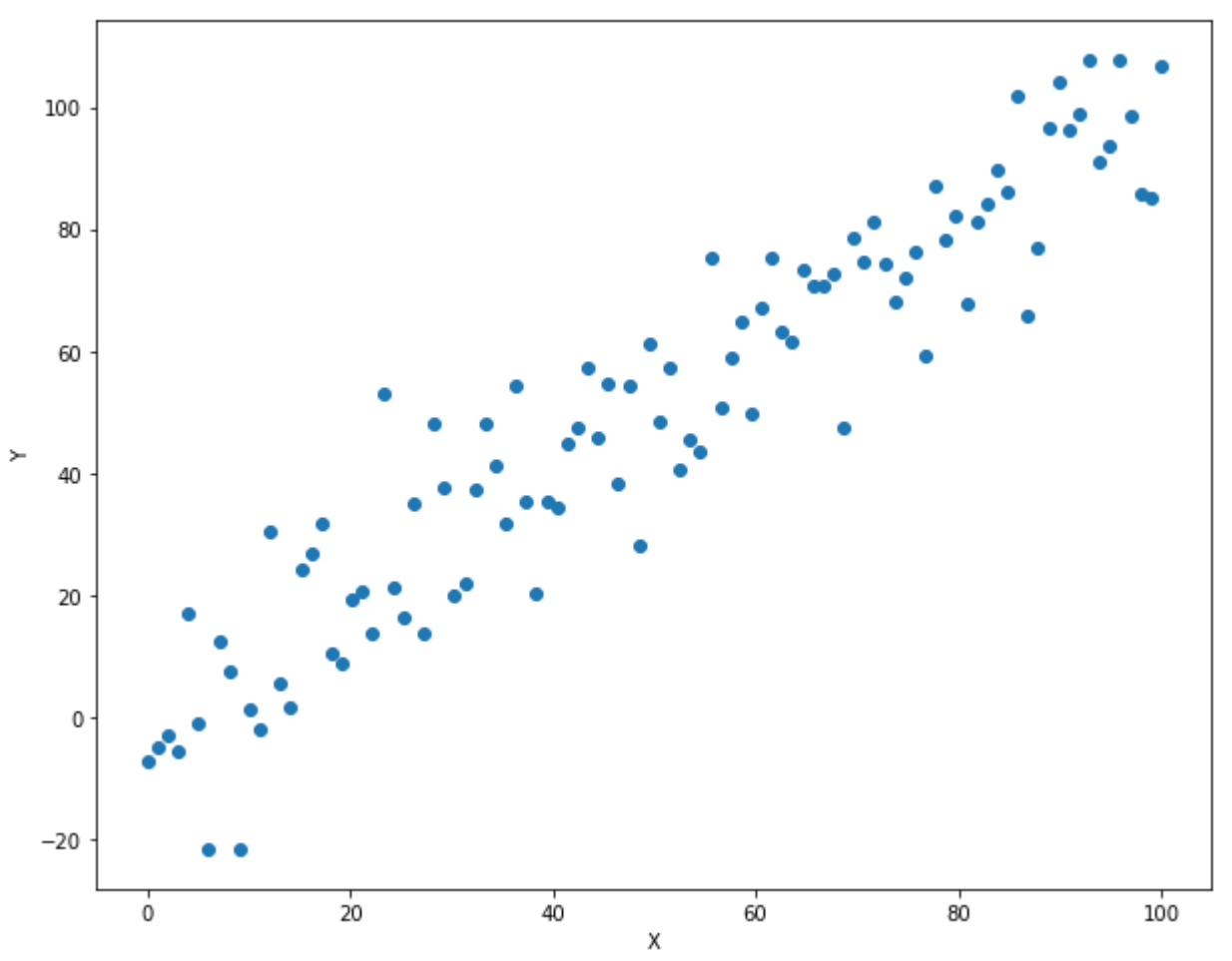
$\frac{\partial y_1}{\partial w} = 2$

$\frac{\partial loss}{\partial w} = \frac{\partial y_0}{\partial w} * 1 + \frac{\partial y_1}{\partial w} * 2$

$= 19 * 1 + 2 * 2$

$= 23$

机器学习程序：线性回归



找到一条直线，使得所有点到直线的距离之和最小。

$$L = \frac{1}{N}\sum_{i=1}^{N}(\omega X_i + b - Y_i)^2$$

$$\min_{\omega,b} L(\omega, b)$$

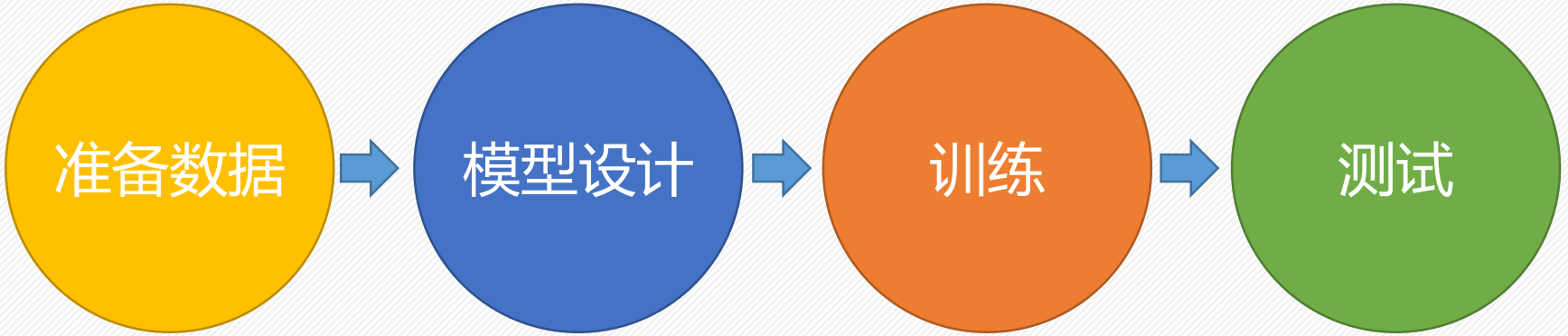
计算梯度： $\frac{\delta L}{\delta \omega}, \frac{\delta L}{\delta b}$

梯度下降法：

$$\omega_{t+1} = \omega_t - \eta \frac{\delta L}{\delta \omega}$$

$$b_{t+1} = b_t - \eta \frac{\delta L}{\delta b}$$

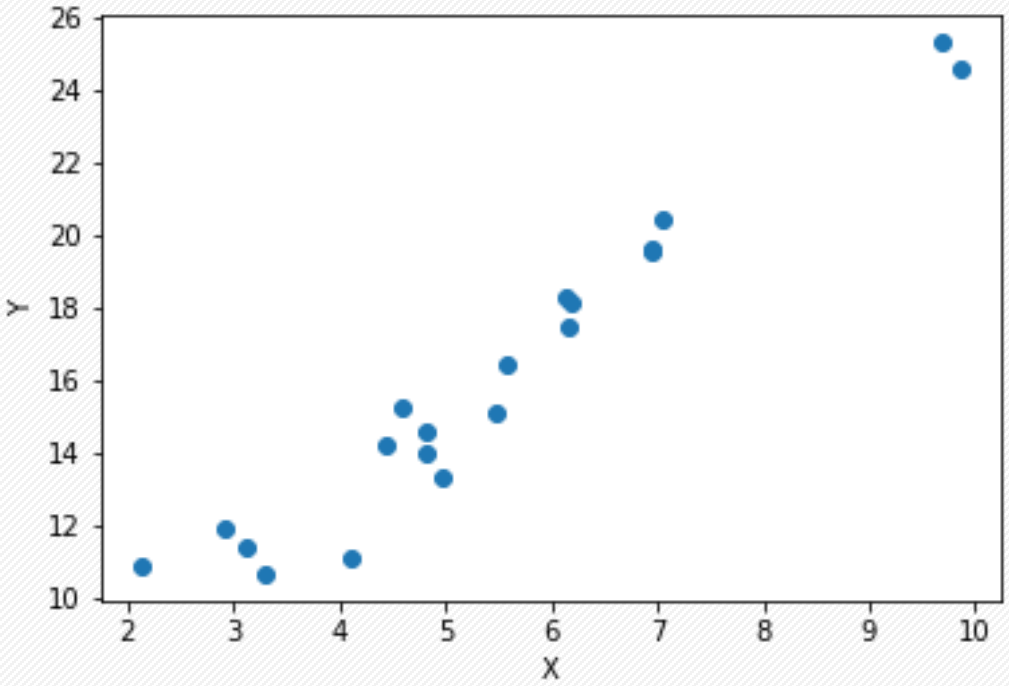
步骤



准备数据：手工生成数据

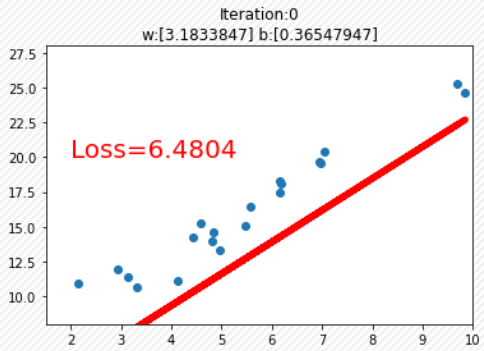
```
%matplotlib inline
import torch
import matplotlib.pyplot as plt

torch.manual_seed(10)
lr = 0.05   #学习率

#创建训练数据
x = torch.rand(20, 1) * 10
y = 2 * x + (5 + torch.randn(20, 1))
```

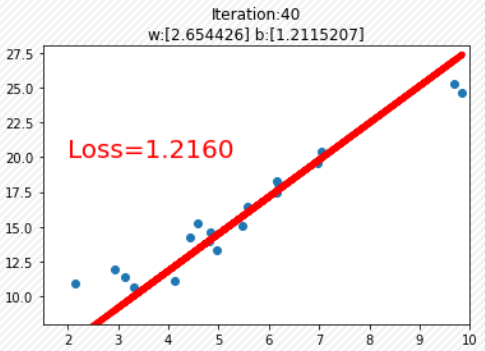
```
import matplotlib.pyplot as plt
plt.plot(x.data.numpy(), y.data.numpy(),'o')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

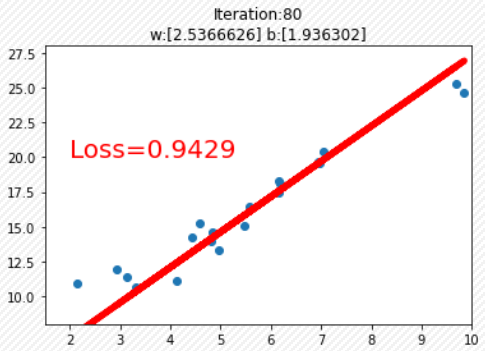


```
#构建线性回归参数
w=torch.randn((1), requires_grad=True)
b=torch.randn((1), requires_grad=True)

#送代训练1000 次
for iteration in range(1000):
    #前向传播，计算预测值
    wx= torch.mul(w, x)
    y_pred =torch.add(wx, b)

    #计算MSE loss
    loss= (0.5 * (y - y_pred) ** 2).mean()

    #反向传播
    loss.backward()

    #更新参数
    b.data.sub_(lr * b.grad)
    w.data.sub_(lr * w.grad)

    #每次更斯参数后，梯度都要清零
    w.grad.zero_()
    b.grad.zero_()
```

```
    #绘图，每隔20次重新绘制直线
    if iteration % 20 == 0:
        plt.scatter(x.data.numpy(), y.data.numpy())
        plt.plot(x.data.numpy(), y_pred.data.numpy(), 'r-', lw=5)
        plt.text(2, 20, 'Loss=%.4f' % loss.data.numpy(), fontdict={'size':20,
'color':'red'})
        plt.xlim(1.5, 10)
        plt.ylim(8, 28)
        plt.title("Iteration:{}\nw:{} b:{}".format(iteration,w.data.numpy(),
b.data.numpy()))
        plt.pause(0.5)

        if loss.data.numpy()< 1:
            break;
```







测试

```
x_test = torch.FloatTensor([1, 2, 10, 100, 1000])
predictions = w.expand_as(x_test) * x_test + b.expand_as(x_test)
```

```
tensor([4.4730,    7.0096,   27.3029,  255.6026, 2538.5989],
       grad_fn=<AddBackward0>)
```


torch.nn
nn.Parameter
Tensor 子类，表示可学习参数，如 weight，bias
nn.Module
所有模型的基类，用于管理网络的属性
nn.functional
函数具体实现，如卷积，池化，激活函数等
nn.init
网络参数初始化方法

神经网络工具箱

torch.autograd库虽然实现了自动求导与梯度反向传播，但如果要完成一个模型的训练，仍需要手写参数的自动更新，训练过程的控制等。
Pytorch进一步提供了集成度更高的模块化接口torch.nn。
torch.nn构建于autograd之上，提供了网络模组、优化器和初始化策略等一系列功能。

■ nn.Module

- 所有神经网络的基类，实现了网络各层的定义及前向计算与反向传播机制。
- 如果想要实现某个神经网络，只需继承nn.Module，在__init__()中定义模型结构与参数，在函数forward()中编写网络前向过程。

nn.Module

```
import torch
from torch import nn
# 先建立一个全连接的子module，继承nn.module
class Linear(nn.Module):
    def __init__ (self, in_dim, out_dim):
        super(Linear, self).__init__() #调用nn.module的构造函数
        # 使用Parameter来构造需要学习的参数
        self.w = nn.Parameter(torch.randn(in_dim, out_dim))
        self.b = nn.Parameter(torch.randn(out_dim))
    def forward(self, x): # 在forward中实现前向传播过程
        x = x.matmul(self.w)  #使用matmul矩阵相乘
        y = x + self.b.expand_as(x)  #expand_as使形状一致
        return y
```

```
#构建感知机类，调用linear的子module
class Perception(nn.Module):
    def __init__ (self, in_dim, hid_dim, out_dim):
        super(Perception,self).__init__()
        self.layer1=Linear(in_dim,hid_dim)
        self.layer2=Linear(hid_dim,out_dim)
    def forward(self, x):
        x=self.layer1(x)
        y=torch.sigmoid(x)
        y=self.layer2(y)
        y=torch.sigmoid(y)
        return y
```

```
perception = Perception(3, 4, 2)
print(perception)
for name, parameter in perception.named_parameters():
    print(name, parameter)
```

```
feature=torch.randn(4,3)
print(feature)
output=perception(feature)
print(output)
```

torch.nn.Linear

CLASS torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)

- **parameters**
  - in_features – size of each input sample
  - out_features – size of each output sample
  - bias – If set to False, the layer will not learn an additive bias. Default: True
- **variables**
  - weight (torch.Tensor) – the learnable weights of the module of shape (out_features, in_features).
  - bias – the learnable bias of the module of shape (out_features)

```
model = nn.Linear(20, 30)
input = torch.randn(128, 20)
output = model(input)
print(output.size())
```

```
torch.Size([128, 30])
```

神经网络工具箱

■1. nn.Parameter函数

- 在类的__init__()中定义学习参数，在此使用 nn.Parameter()函数定义了全连接中的*w*和*b*，这是一种特殊的Tensor构造方法，默认需要求导，即requires_grad为True。

■2. forward()函数

- 用来进行网络的前向传播，并需要传入相应的 Tensor。
- 例如，perception(data)即是直接调用了forward()。

■3. 反向传播

- nn.Module可以自动利用autograd机制实现反向传播，不需要自己手动实现。

神经网络工具箱

■4. 多个module的嵌套

- 在Module搭建时，可以嵌套子Module，上例的Perception中调用了Linear类。可以使网络模块化，提升代码复用性。

■5 nn.Module与nn.functional库

- nn.Xxx 和 nn.functional.xxx 功能相同，比如 nn.functional.conv2d 和 nn.Conv2d都能实现卷积操作。
- nn.functional.xxx是函数接口，而nn.Xxx是nn.functional.xxx的类封装，并且nn.Xxx都继承于共同祖先nn.Module。nn.Xxx除了具有nn.functional.xxx功能之外，内部附带了nn.Module相关的属性和方法，例train(), eval(), load_state_dict, state_dict 等。
- PyTorch官方推荐：具有学习参数的（例，conv2d, linear, batch_norm)采用nn.Xxx方式，没有学习参数的（例，maxpool, loss func, activation func）等选择使用nn.functional.xxx或者nn.Xxx方式。

神经网络工具箱

■6. nn.Sequential()模块

- 当模型是简单前馈网络时，即上一层输出直接作为下一层输入，可采用**nn.Sequential()搭建模型，不必手动在forward()中一层一层地前向传播。**

```
class Perception(nn.Module):
    def __init__(self, in_dim, hid_dim, out_dim):
        # 利用nn.Sequenctial()快速搭建网络模块
        super(Perception, self).__init__()
        self.layer = nn.Sequential(
            nn.Linear(in_dim, hid_dim),
            nn.Sigmoid(),
            nn.Linear(hid_dim, out_dim),
            nn.Sigmoid()
        )
    def forward(self, x):
        y = self.layer(x)
        return y
```

```
perception = Perception(3,4,2)
print(perception)
for name, parameter in perception.named_parameters():
    print(name, parameter)

data=torch.randn(4, 3)
print(data)
output=perception(data)
print(output)
```

损失函数

PyTorch 中常用损失函数

类	算法名称	适用问题类型
torch.nn.L1Loss()	平均绝对值误差损失	回归
torch.nn.MSELoss()	均方误差损失	回归
torch.nn.CrossEntropyLoss()	交叉熵损失	多分类
torch.nn.NLLLoss()	负对数似然函数损失	多分类
torch.nn.NLLLoss2d()	图片负对数似然函数损失	图像分割
torch.nn.KLDivLoss()	KL散度损失	回归
torch.nn.BCELoss()	二分类交叉熵损失	二分类
torch.nn.MarginRankingLoss()	评价相似度的损失	
torch.nn.MuliL abelMarginLoss()	多标签分类的损失	多标签分类
torch.nn.SmoothLlLoss()	平滑的L1损失	回归
torch.nn.SoftMarginLoss()	多标签二分类问题的损失	多标签二分类

损失函数

均方误差损失

torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')

参数:

- size_average: 默认为True。 计算的损失为每个batch的均值，否则为每个batch的和。以后将会弃用该参数，可以通过设置reduction来代替该参数的效果。
- reduce:默认为True， 此时计算的损失会根据size_ _average参数设定，是计算每个batch的均值或和。以后将会弃用该参数。
- reduction:通过指定参数取值为none、mean、sum来判断损失的计算方式。默认为mean，即计算的损失为每个batch的均值；如果为sum，则计算的损失为每个batch的和；如果设置为none，则表示不使用该参数。

对模型的预测输出x和目标y计算均方误差损失方式为

$$loss(x,y) = \frac{1}{N}\ (x_i - y_i)^2$$

如果reduction的取值为sum，则不除以N。

损失函数

■PyTorch在torch.nn及torch.nn.functional中均提供有各种损失函数，通常来讲，由于损失函数不含有可学习的参数，因此两者在功能上基本没有区别。

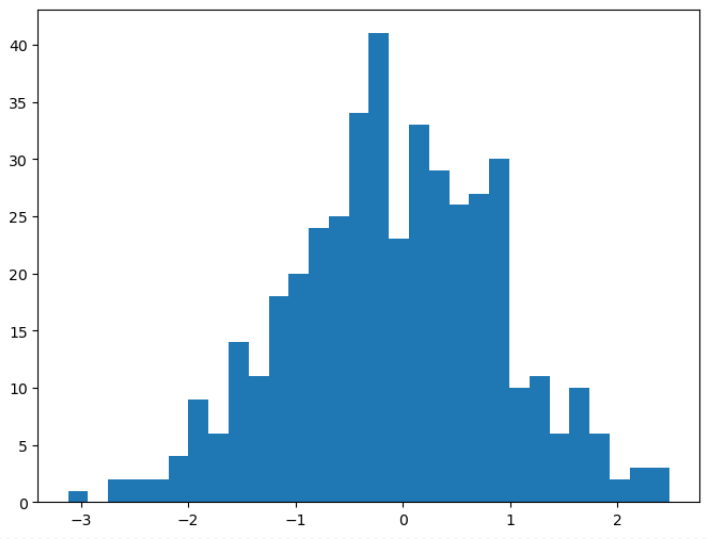
```
from torch import nn
import torch.nn.functional as F
label=torch.Tensor([0, 1, 1, 0]).long()
criterion=nn.CrossEntropyLoss()
loss_nn=criterion(output, label)
print(loss_nn)

loss_functional=F.cross_entropy(output, label)
print(loss_functional)
```

网络参数初始化

方法(类)	功能
nn.init.uniform_(tensor, a=0.0, b=1.0)	从均匀分布U(a, b)中生成值，填充输入的张量或变量
nn.init.normal_(tensor, mean=0.0, std=1.0)	从给定均值mean和标准差std的正态分布中生成值，填充输入的张量或变量
nn.init.constant (tensor, val)	用val的值填充输入的张量或变量
nn.init.eye_(tensor)	用单位矩阵来填充输入二维张量或变量
nn.init.dirac_(tensor)	用Dirac delta函数来填充{3, 4, 5}维输入张量或变量。在卷积层尽可能多地保存输入通道特性
nn.init.xavier_uniform_(tensor, gain=1.0)	使用Glorot initialization 方法均匀分布生成值填充张量
nn.init.xavier_normal(tensor, gain=1.0)	使用Glorot initialization 方法正态分布生成值填充张量
nn.init.kaiming_uniform_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu')	使用He initialization 方法均匀分布生成值填充张量
n.init.kaiming.normal_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu')	使 用He initialization 方法正态分布生成值填充张量
nn.init.orthogonal (tensor, gain=1)	使用正交矩阵填充张量进行初始化

例，初始化示例

```
#针对一个层的权重初始化方法
conv1 = torch.nn.Conv2d(3,16,3)
#使用标准正态分布初始化权重
torch.manual_seed(12) #随机数初始化种子
torch.nn.init.normal(conv1.weight, mean=0, std=1)
#使用直方图可视化convl.weight的分布情况
import matplotlib.pyplot as plt
plt.figure(figsize=(8,6))
plt.hist(conv1.weight.data.numpy().reshape((-1, 1)), bins = 30)
plt.show()
```



针对一个网络的权重初始化方法

```
class TestNet (nn Module):
    def __init__(self):
        super(TestNet,self).__init__()
        self. convl1=nn.Conv2d(3,16,3)
        self.hidden =nn. Sequential(
                nn. Linear (100,100),
                nn. ReLU(),
                nn. Linear (100,50),
                nn. ReLU())
        self.cla = nn.Linear (50,10)
    #定义网络的前向传播路径
    def forward(self, x):
        x = self.conv1 (x)
        x = x.view (x.shape[0],-1)
        x = self.hidden (x)
        output = self.cla(x)
        return output
```

```
testnet = TestNet ()
```

```
Out[3] :TestNet(
(conv1): Conv2d(3, 16, kernel size=(3, 3)，  stride=(1, 1))
(hidden): Sequential(
(0): Linear (in_ features=100, out_ features=100, bias=True)
1): ReLU()
(2): Linear(in features=100，  out_ features=50, bias=True)
(3): ReLU()
(cla): Linear(in features=50, out_ features=10, bias=True)
```

针对一个网络的权重初始化方法

```
def init_weights (m) :
    ##如果是卷积层
    if type(m) == nn.Conv2d:
        torch.nn.init.normal (m.weight, mean=0,std=0.5)
    ##如果是全连接层
    if type(m) == nn.Linear:
        torch.nn.init.uniform (m.weight,a=-0.1,b=0.1)
         m.bias.data.fill_ (0.01)
```

```
##使用网络的apply方法进行权重初始化
torch.manual_seed (13)
##随机数初始化种子
testnet.apply(init_weights)
```

Dataset和DataLoader

1.torch.utils.data.Dataset

- 将数据包装为Dataset类。
- **抽象类**，不能实例化，需要构造抽象类的子类。
- 子类必须实现__len__()和__getitem__()函数，前者给出数据集的大小，后者是用于查找数据和标签。

2. torch.utils.data.DataLoader

- 构建可迭代的数据加载器。
- 每次根据参数生成一个batch的数据

■ **联系**：Dataset是一个包装类，用来将数据包装为Dataset类，然后传入DataLoader中，使用DataLoader可以更加快捷地对数据进行操作。

Dataset示例

```
import torch
from torch.utils.data import Dataset
import numpy as np

#创建子类
class subDataset(Dataset):
    #初始化，定义数据内容和标签
    def __init__(self, Data, Label):
        self.Data = Data
        self.Label = Label
    #返回数据集大小
    def __len__(self):
        return len(self.Data)
    #得到数据内容和标签
    def __getitem__(self, index):
        data = torch.Tensor(self.Data[index])
        label = torch.Tensor(self.Label[index])
        return data, label
```

```
#在类外对Data和Label赋值
Data = np.asarray([[1, 2], [3, 4],[5, 6], [7, 8]])
Label = np.asarray([[0], [1], [0], [2]])

dataset = subDataset(Data, Label)

print(dataset)
print('dataset大小为：', dataset.__len__())
print(dataset.__getitem__(0))
print(dataset[0])
```

```
<__main__.subDataset object at 0x000001F301814708>
dataset大小为： 4
(tensor([1., 2.]), tensor([0.]))
(tensor([1., 2.]), tensor([0.]))
```

DataLoader

```
from torch.utils.data import DataLoader

#创建DataLoader迭代器
dataloader = DataLoader(dataset, batch_size= 2, shuffle = False, num_workers= 0)

for i, item in enumerate(dataloader):
        print('i:', i)
        data, label = item
        print('data:', data)
        print('label:', label)
```

```
i: 0
data: tensor([[1., 2.],
        [3., 4.]])
label: tensor([[0.],
        [1.]])
i: 1
data: tensor([[5., 6.],
        [7., 8.]])
label: tensor([[0.],
        [2.]])
```

torch.utils.data.TensorDataset示例

```
# 处理分类数据
import torch.utils.data as Data
from sklearn.datasets import load_iris

#读取iris数据集(Numpy数组)
iris_x, iris_y = load_iris(return_X_y=True)
print("iris_x.dtype:",iris_x. dtype)
print ("iris_y:", iris_y.dtype)
```

```
iris_x.dtype: float64
iris_y: int32
```

```
#训练集转化为张量
train_xt = torch.from_numpy(iris_x.astype(np.float32))
train_yt =torch.from_numpy(iris_y.astype(np.int64))
print("train_xt.dtype:",train_xt.dtype)
print("train_yt.dtype:",train_yt.dtype)
```

```
train_xt.dtype: torch.float32
train_yt.dtype: torch.int64
```

torch.utils.data.TensorDataset示例

```
#将训练集转化为张量后，使用TensorDataset将x和y整理到一起
train_data = Data.TensorDataset(train_xt, train_yt)
#定义一个数据加载器将训练数据集进行批量处理
train_loader = Data.DataLoader(dataset = train_data, #使用的数据集
                               batch_size=10, #批处理样本大小
                               shuffle = True, #每次迭代前打乱数据
                               num_workers = 1,
                              )
#检查训练数据集的一个batch样本的维度是否正确
for step, (b_x, b_y) in enumerate (train_loader):
    if step>0:
        break
#输出训练图像的尺寸和标签的尺寸与数据类型
print("b_x.shape:",b_x.shape)
print("b_y.shape:",b_y.shape)
print("b_x.dtype:",b_x.dtype)
print("b_y.dtype:",b_y.dtype)
```

```
b_x.shape: torch.Size([10, 4])
b_y.shape: torch.Size([10])
b_x.dtype: torch.float32
b_y.dtype: torch.int64
```

Epoch, Iteration, Batchsize

Epoch：所有训练样本都已经输入到模型中，称为一个 Epoch。
Iteration：一批样本输入到模型中，称为一个 Iteration。
Batchsize：批大小，决定一个 iteration 有多少样本，也决定了一个 Epoch 有多少个 Iteration。

设样本总数有 80， Batchsize 为 8，则共有 80/8=10个 Iteration。这里 1 Epoch = 10 Iteration。
设样本总数有 86，设置 Batchsize 为 8。如果drop_last=True则共有 10 个 Iteration；如果drop_last=False则共有 11 个 Iteration。

示例——网络定义与训练

```
class MLPmodel (nn.Module) :
    def __init__ (self) :
        super (MLPmodel,self)._ init__ ()
        #定义第一个隐藏层
        self.hiddenl = nn.Linear (
              in_features = 13,  #第一个隐藏层的输入，数据的特征数
              out_ features = 10, #第一个隐藏层的输出，神经元的数量
             bias=True,   #默认会有偏置
         self.active1 = nn.ReLU()  #定义第一个隐藏层
         self.hidden2 = nn.Linear(10,10)
         self.active2 = nn.ReLU()   #定义预测回归层
         self.regression = nn.Linear(10,1)  #定义网络的前向传播路径
    def forward(self, x):
         x = self.hidden1 (x)
         self.activel (x)
         x=self.hidden2 (x)
         x = self.active2(x)
         output = self.regression(x)
         return output  #输出为output
```

```
mlp1 = MLPmodel ()
print (mlp1)
```

```
MLPmodel (
(hidden1): Linear (in features=13, out_ features=10, bias-True)
(activel): ReLU()
(hidden2): Linear (in_ features=10, out_ features=10, bias=True)
(active2): ReLU()
(regression): Linear (in features=10, out_ features=1, bias=True)
```

示例——网络定义与训练

```
optimizer = SGD (mlp1.parameters(), 1r=0.001)
loss_func = nn.MSELoss()  #最小均方根误差
train_loss_all= []   #输出每个批次训练的损失函数
#进行训练，并输出每次迭代的损失函数
for epoch in range (30) :
    #对训练数据的加载器进行迭代计算
    for step, (b_x, b_y) in enumerate (train_loader):
        output = mlp1(b_x) .flatten()
        ## MLP在训练batch上的输出
        train_loss = loss_func(output,b_y) #均方根误差
        optimizer.zero_grad()   #每个达代步的梯度初始化为0
        train_loss .backward()   #损失的后向传播，计算梯度
        optimizer.step()    #使用梯度进行优化
        train_loss_all.append(train_loss.item())
```

```
plt.figure()
plt.plot(train_loss_all,"r-")
plt.title("Train loss per iteration")
plt.show()
```

计算机视觉工具包torchvision

torchvision.transforms: 包括常用的图像预处理方法
torchvision.datasets: 包括常用数据集如 mnist、CIFAR-10、Image-Net 等
torchvision.models: 包括常用的预训练好的模型，如 AlexNet、VGG、ResNet、GoogleNet 等

torchvision中图像的变换操作

数据集对应的类描述	
transforms.Compose()	将多个transform组合起来使用
transforms.Scale()	按照指定的图像尺寸对图像进行调整
transforms.CenterCrop()	将图像进行中心切割，得到给定的大小
transforms.RandomCrop()	切割中心点的位置随机选取
transforms.RandomHorizontalFlip()	图像随机水平翻转
transforms.RandomVerticalFlip()	图像随机垂直翻转
transforms.RandomSizedCrop()	将给定的图像随机切割，然后再变换为给定大小
transforms.Pad()	将图像所有边用给定的pad value填充
transforms.ToTensor()	把一个取值范围是[0,255]的PIL图像或形状为(H,W,C)的数组，转换成形状为[C,H,W],取值范围是[0, 1.0]的张量(torch.FloatTensor)
transforms.Normalize()	将给定的图像进行规范化操作
transforms.Lambda(lambd)	使用lambd作为转化器，可自定义图像操作方式

图像随机垂直翻转

```
import matplotlib.pyplot as plt
import torchvision. transforms as T
from PIL import Image
path = "d:/tmp/Lenna.png"
plt.figure()

img_array = Image.open(path)
plt.subplot(1,2,1)
plt.imshow(img_array)
plt.axis('off')

new_img_array = T.RandomVerticalFlip(0.5)(img_array)
plt.subplot(1,2,2)
plt.imshow(new_img_array)

plt.axis('off')
plt.show()
```

常用的图像预处理方法

```
# 设置训练集的数据增强和转化
train_transform = transforms.Compose([
    transforms.Resize((32, 32)),  #缩放
    transforms.RandomCrop(32, padding=4), #裁剪
    transforms.ToTensor(), #转为张量，同时归一化
    transforms.Normalize(norm_mean, norm_std), #标准化
])

# 设置验证集的数据增强和转化，不需要 RandomCrop
valid_transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize(norm_mean, norm_std),
])
```

torchvision提供的数据集

数据集对应的类	描述
datasets.MNIST()	手写体数据集
datasets.FashionMNIST()	(衣服、鞋子、包等)10类数据集
datasets.KMNIST()	一些文字的灰度数据
datasets.CocoCaptions()	用于图像标注的MS COCO数据
datasets.CocoDetection()	用于检测的MS COCO数据
datasets.LSUN()	10个场景和20个目标的分类数据集
datasets.CIFAR10()	CIFAR10类数据集
datasets.CIFAR100()	CIFAR100类数据集
datasets.STL100()	包含10类的分类数据集和大量的未标记数据
datasets.ImageFolder()	一个通用的数据加载器从文件夹中读取数据

例， FashiomMNIST示例

```
#使用FashiomMNIST 数据，准备训练数据集
train data= FashionMNIST(
    root="./data/FashionMNIST", #数据的路径
    train=True, #只使用训练数据集
    transform= train_transform,
    download= False #因为数据已经下载过，所以不再下载
)

#定义一个数据加载器
train_loader = Data.DataLoader(dataset = train_data, #使用的数据集
                               batch_size=64, #批处理样本大小
                               shufle = True, #每次迭代前打乱数据
                               num workers = 2, #使用两个进程
)

#计算train loader 有多少个batch
print("train_loader 的batch数量为： ", len(train_loader))
```

train_laoder的batch数量为：938

模型处理

1）网络模型库：torchvision.models

提供了众多经典的网络结构与预训练模型，例如VGG、ResNet和Inception等，利用这些模型可以快速搭建网络，不需要逐层手动实现。

```
from torch import nn
from torchvision import models
# 通过torchvision.model直接调用VGG16的网络结构
vgg = models.vgg16()
# VGG16的特征层包括13个卷积、 13个激活函数ReLU、 5个池化， 一共31层
print(len(vgg.features))
# VGG16的分类层包括3个全连接、 2个ReLU、 2个Dropout， 一共7层
print(len(vgg.classifier))
# 可以通过出现的顺序直接索引每一层
print(vgg.classifier[-1])
# 也可以选取某一部分， 如下代表了特征网络的最后一个卷积模组
print(vgg.features[24:])
```

模型处理

• 2）加载预训练模型

- 对于计算机视觉的任务，通常很难拿到很大的数据集，在这种情况下重新训练一个新的模型是比较复杂的，并且不容易调整，因此，Fine-tune（微调）是一个常用的选择。所谓Fine-tune是指利用别人在一些数据集上训练好的预训练模型，在自己的数据集上训练自己的模型。
- 在具体使用时，通常有两种情况，第一种是直接利用torchvision.models中自带的预训练模型，只需要在使用时赋予**pretrained参数为True**即可。

```
from torch import nn
from torchvision import models

# 通过torchvision.model直接调用VGG16的网络结构
vgg = models.vgg16(pretrained=True)
```

模型处理

```
import torch
from torch import nn
from torchvision import models

#通过torchvision.model直接调用VGG16的网络结构
vgg = models.vgg16()
static_dict = torch.load(" your model path")

#利用load_state_dict，遍历预训练模型的关键字，如果出现在VGG中，则加载预训练参数
vgg.load_state_dict({k:v for k,v in state_dict.items() if k in vgg.state_dict()})
```

```
for layer in range(10):
   for p in vgg[layer].parameters():
      p.requires_grad = False
```

模型保存与加载

方法1：保存整个模型

```
#保存整个模型
torch.save(mlp2, "data/chap3/mlp2.pt")
#导入保存的模型
mlp2load = torch.load("data/chap3/mlp2.pt")
```

方法2：只保存模型的参数

```
#只保存模型参数
torch.save(mlp2.state_dict(),"data/chap3/mlp2_param.pt")
#导入保存的模型参数
mip2param=torch.load("data/chap3/mlp2_param.pt")
```

The end