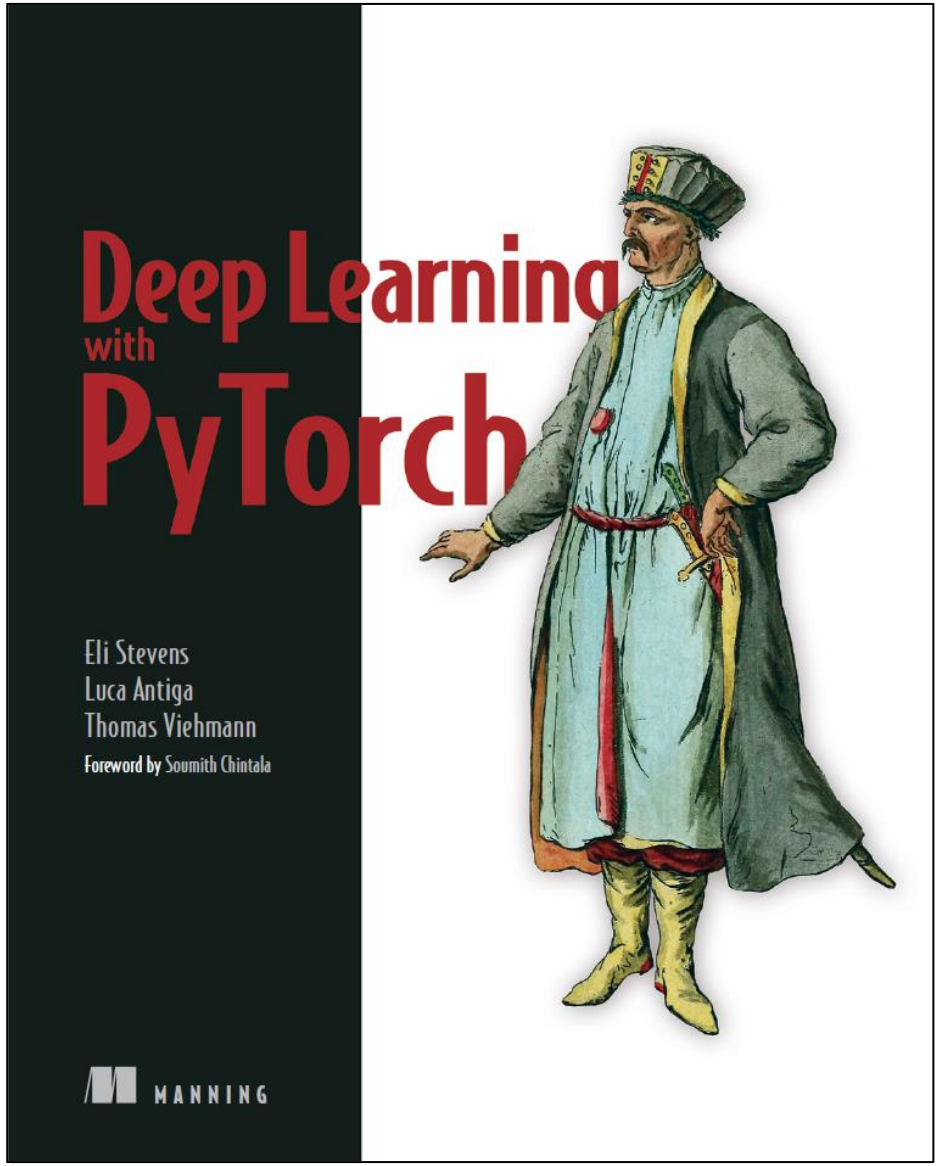
PyTorch Tutorial

王文中

安徽大学计算机学院



https://www.manning.com/books/deep-learning-with-pytorch

START LOCALLY

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, 1.12 builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

PyTorch Build	Stable (1.12.1)	Preview (Nightly)	LTS (1.8.2)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python	C++ / Java		
Compute Platform	CUDA 10.2	CUDA 11.3	CUDA 11.6	~~ROCm 5.1.1~~	CPU
Run this Command:	`conda install pytorch torchvision torchaudio cpuonly -c pytorch`			

https://pytorch.org/get-started/locally/

大纲

- 核心数据结构Tensor
- 多层感知机编程
- 卷积神经网络编程

PyTorch

Python API [-]

torch

torch.nn

torch.nn.functional

torch.Tensor

Tensor Attributes

Tensor Views

torch.amp

torch.autograd

torch.library

torch.cuda

torch.backends

torch.distributed

Libraries [-]

torchaudio

TorchData

TorchRec

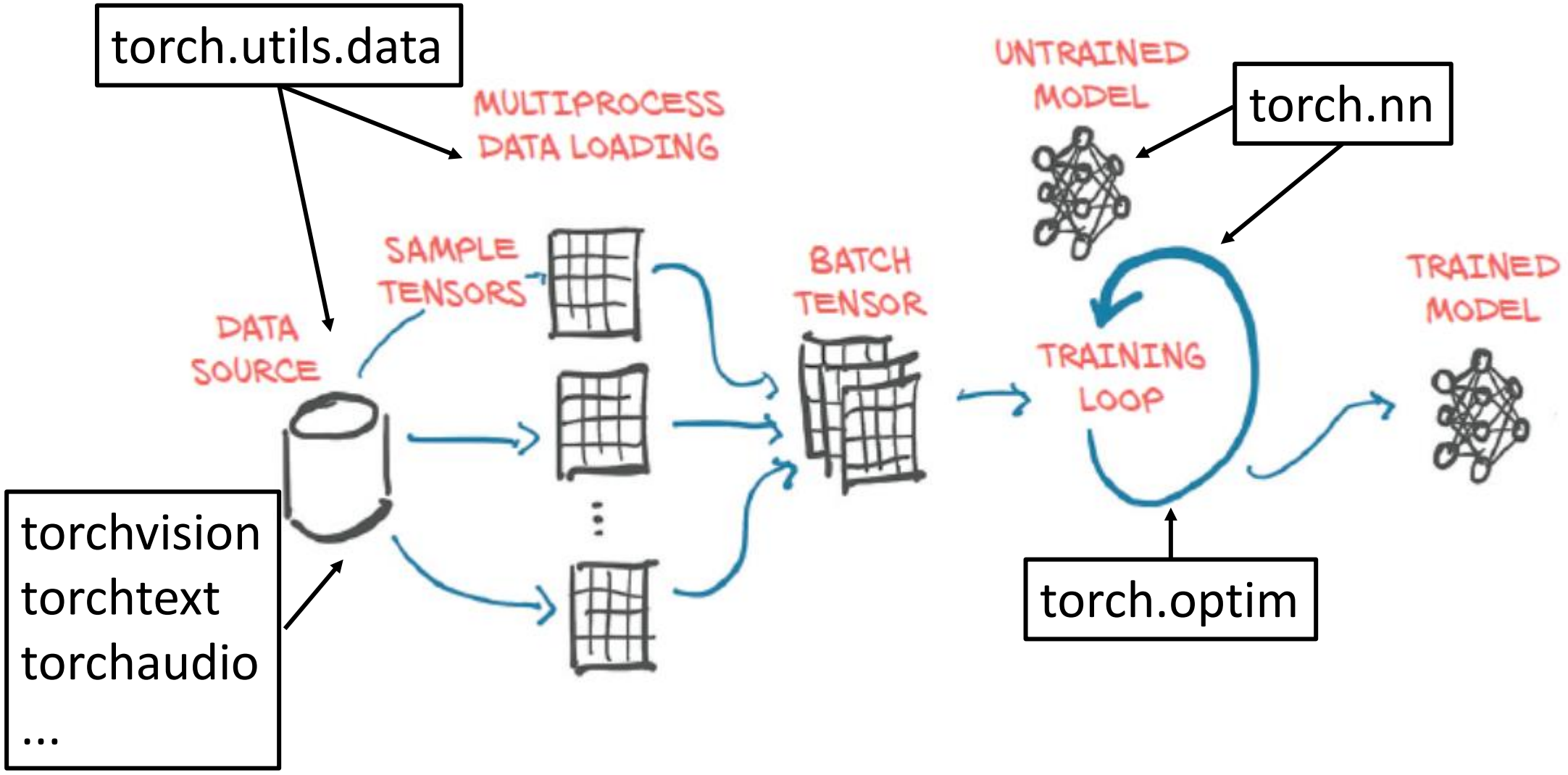
TorchServe

torchtext

torchvision

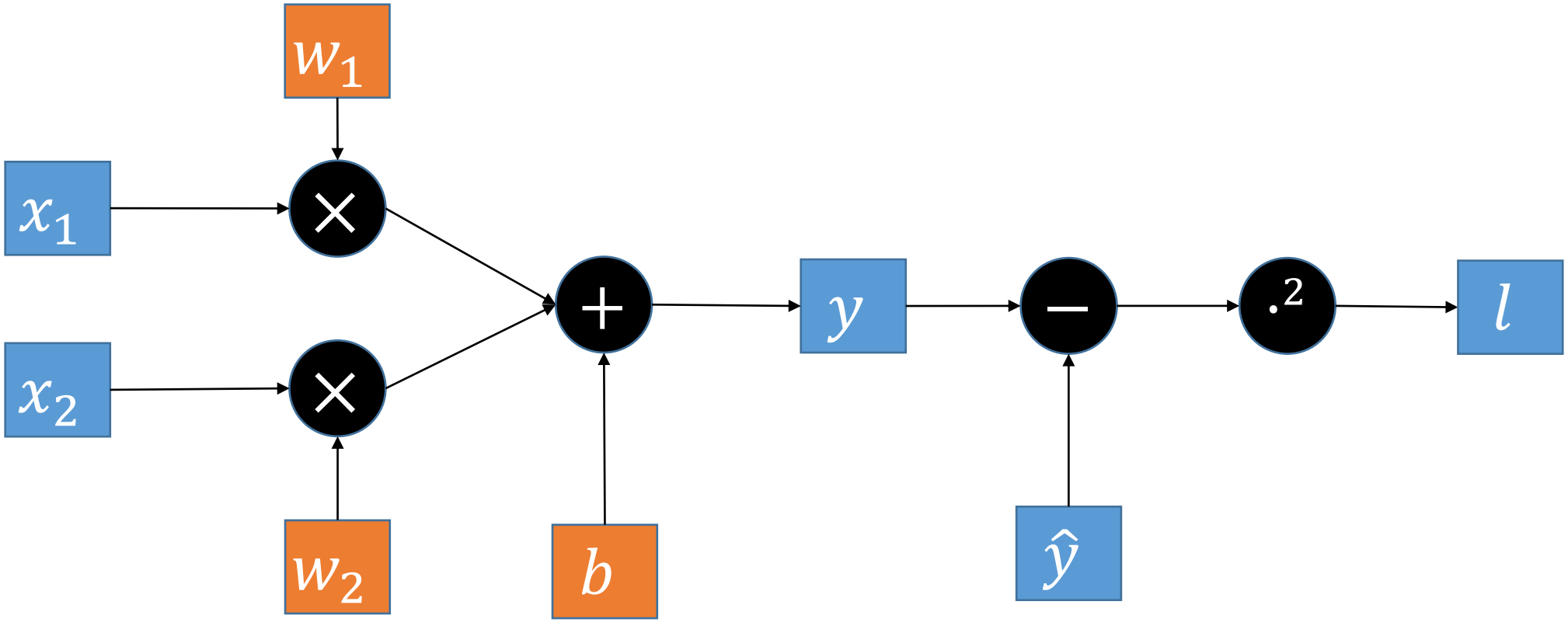
PyTorch on XLA Devices

PyTorch


torch.utils.data
MULTIPROCESS DATA LOADING
UNTRAINED MODEL
torch.nn
SAMPLE TENSORS
DATA SOURCE
BATCH TENSOR
TRAINING LOOP
TRAINED MODEL
torchvision
torchtext
torchaudio
...
torch.optim

计算图

$$y = h(x; w, b) = w_1 \times x_1 + w_2 \times x_2 + b$$
$$l = (y - \hat{y})^2$$



用张量表示数据与参数

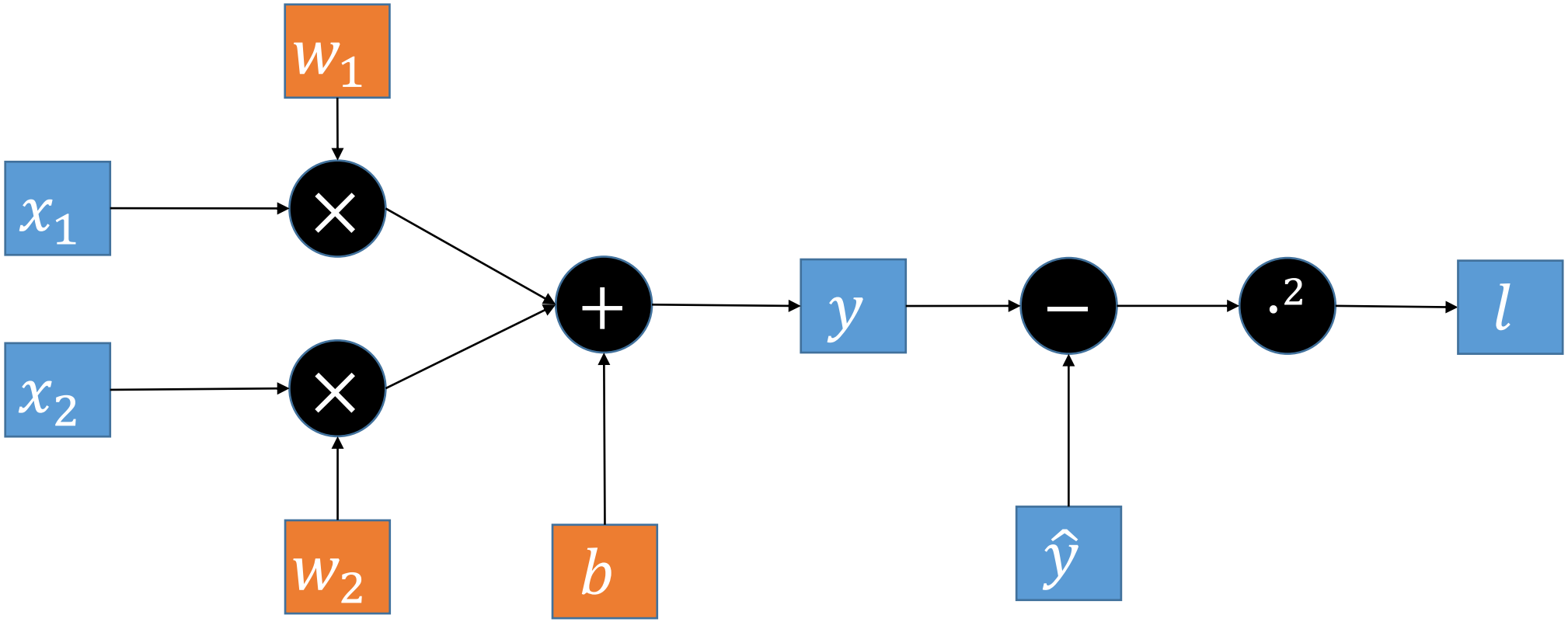
$$y = h(x; w, b) = w_1 \times x_1 + w_2 \times x_2 + b$$
$$l = (y - \hat{y})^2$$

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \mathrm{w} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}, b$$
$$y = w^T x + b$$
$$l = (y - \hat{y})^2$$

$$X = \left(x^{(1)}, x^{(2)}, \dots, x^{(n)}\right)$$
$$\hat{Y} = \left(y^{(1)}, y^{(2)}, \dots, y^{(n)}\right)$$
$$Y = w^T X + b$$
$$L = \left(Y - \hat{Y}\right)^2$$

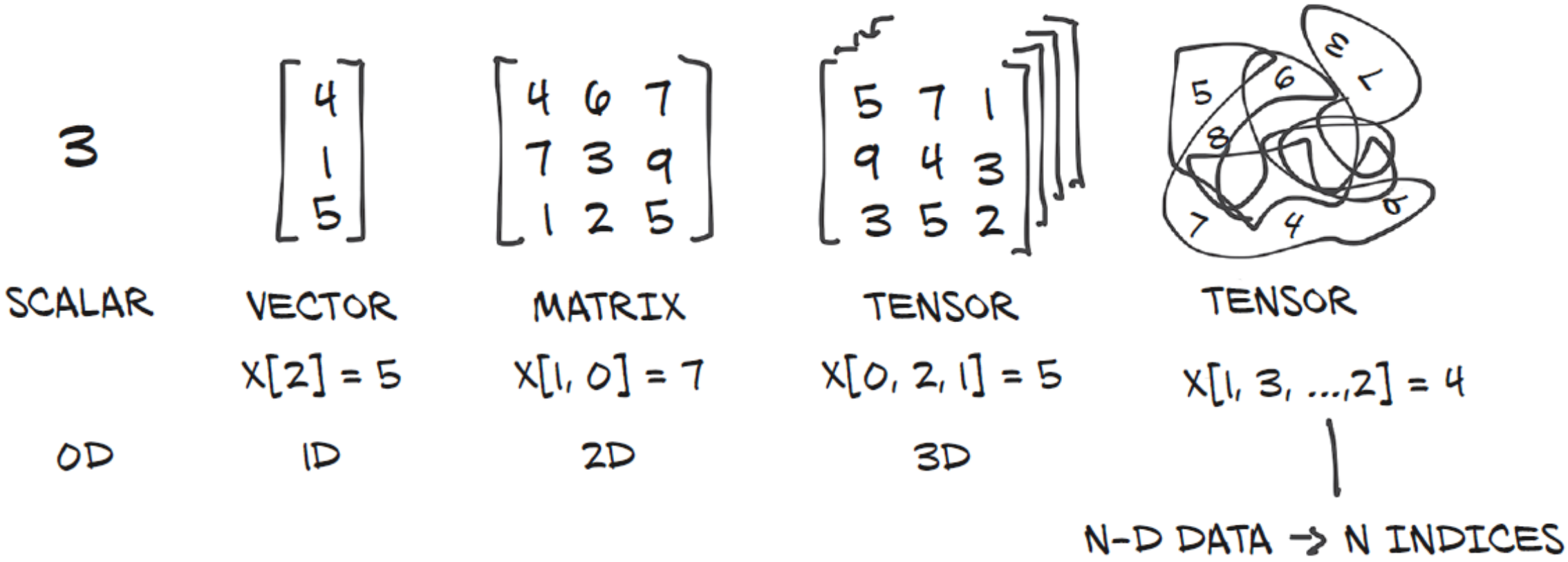
计算:张量流(Tensor Flow)

$$y = h(x; w, b) = w_1 \times x_1 + w_2 \times x_2 + b$$
$$l = (y - \hat{y})^2$$



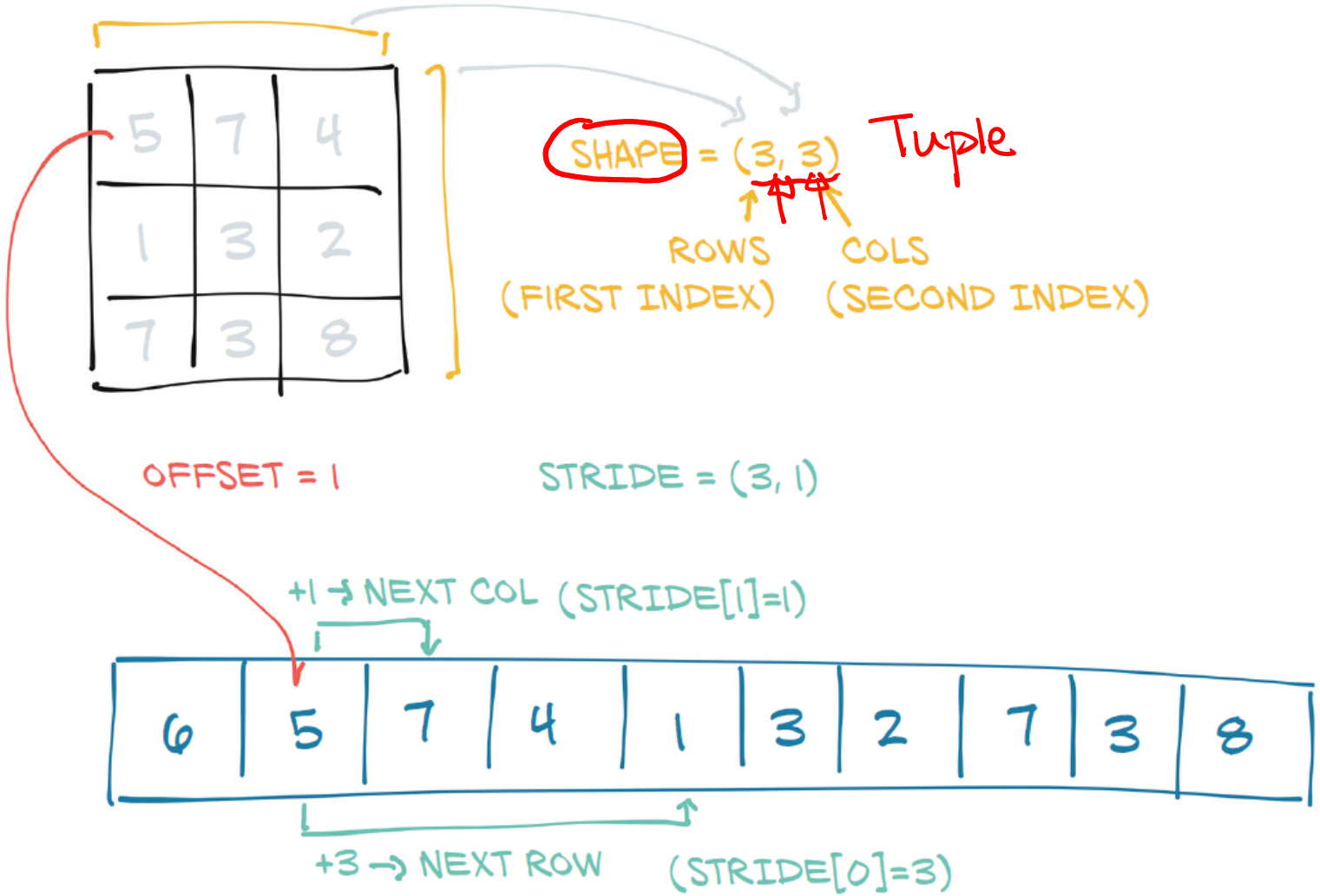
1.核心数据结构:Tensor

Tensor:多维数组


3
4
1
5
4 6 7
7 3 9
1 2 5
5 7 1
9 4 3
3 5 2
SCALAR
VECTOR
X[2] = 5
0D
1D
MATRIX
X[1, 0] = 7
2D
TENSOR
X[0, 2, 1] = 5
3D
TENSOR
X[1, 3, ...,2] = 4
N-D DATA → N INDICES

1.1 创建与操作Tensor

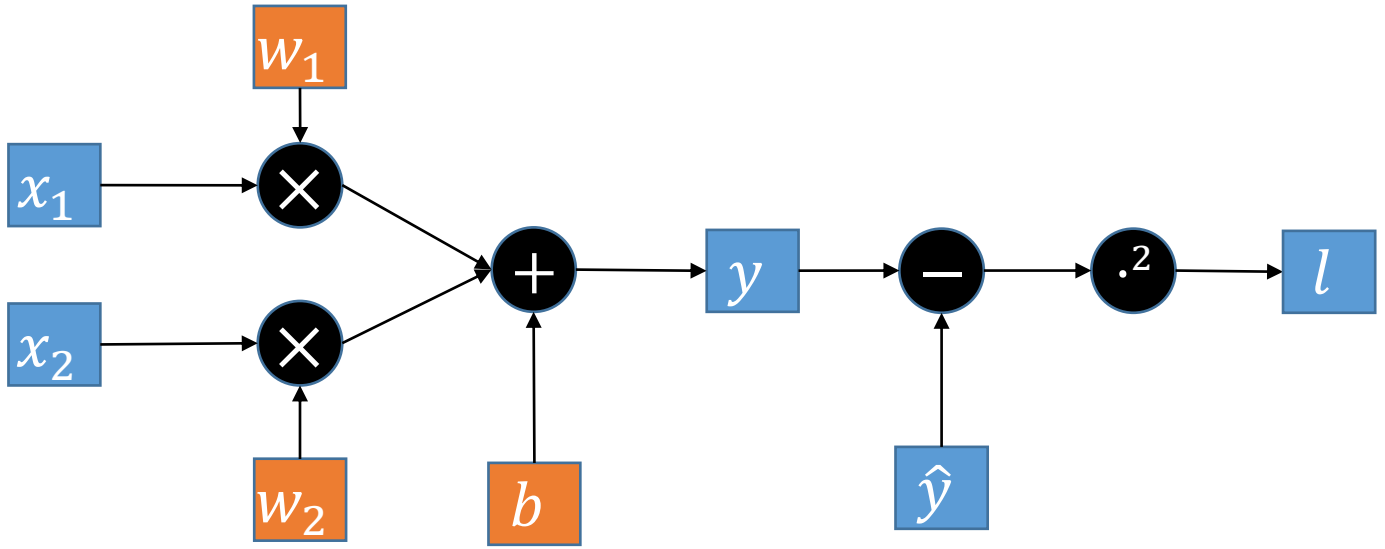
Tensor的属性


5 7 4
1 3 2
7 3 8
SHAPE = (3, 3)
Tuple
ROWS
(FIRST INDEX)
COLS
(SECOND INDEX)
OFFSET = 1
STRIDE = (3, 1)
+1 → NEXT COL (STRIDE[1]=1)
6 5 7 4 1 3 2 7 3 8
+3 → NEXT ROW
(STRIDE[0]=3)

创建一个Tensor

- 1.使用torch的创建算子(Creation Ops)

- 2.通过对Tensor的运算创建



创建一个Tensor

```
a = torch.tensor(1.0)
print(a)
print(a.dim())
print(a.shape)#print(a.size())
print(a.dtype)
print(a.item())
```

```
tensor(1.)
0
torch.Size([])
torch.float32
1.0
```

```
b = torch.tensor([1,2,3])
print(b)
print(b.dim())
print(b.shape)
print(b.dtype)
print(b.numpy())
```

Tensor → ndarray

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

```
x = np.array([0,1])
c = torch.from_numpy(x)
```

```
tensor([0, 1], dtype=torch.int32)
```

创建一个Tensor

```
c = torch.tensor([[1,2],[3,4],[5,6]],dtype = torch.int16)
print(c)
print(c.dim())
print(c.shape)
print(c.dtype)
print(c.numpy())
```

```
tensor([[1, 2],
        [3, 4],
        [5, 6]], dtype=torch.int16)
2
torch.Size([3, 2])
torch.int16
[[1 2]
 [3 4]
 [5 6]]
```

创建一个Tensor

```
d = torch.zeros((3,4))
```

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

```
e = torch.randn((2,2,3,4))
```

```
tensor([[[[-1.0510,  0.3902,  0.1810,  1.0036],
          [-1.4556,  0.4581, -0.5105, -1.3028],
          [-0.8999,  0.3423,  0.0936,  1.3038]],

         [[-0.7691,  0.4797, -1.8662, -0.7550],
          [ 0.0411,  0.5273, -0.1488,  1.0463],
          [ 1.1466, -0.3363,  0.7472, -1.2813]]],


        [[[-0.3448,  0.3213,  0.2527,  1.2449],
          [ 0.7206, -1.5667,  0.2489, -1.9135],
          [ 0.0946, -0.1390,  0.1650,  1.1801]],

         [[-0.9789,  0.7678, -0.2877, -1.1634],
          [ 0.7762,  0.7401,  0.2081,  0.3059],
          [ 0.5780, -0.2145,  0.8056,  0.8838]]]])
```

```
e.stride()
```

```
(24, 12, 4, 1)
```

设置Tensor的设备属性

```
a = torch.tensor([1,2],device = 'cuda')

b = torch.tensor([1,2]).to(device='cuda')
```

访问Tensor的元素

```
a = torch.tensor(1.0)
a[0]
```

```
IndexError                                Traceback (most recent call last)
<ipython-input-68-292e29054dae> in <module>()
      1 a = torch.tensor(1.0)
----> 2 a[0]

IndexError: invalid index of a 0-dim tensor. Use tensor.item() to convert a 0-dim tensor to a Python number
```

```
a.item()
type(a.item())
```

```
1.0
float
```

```
a.numpy()
```

```
array(1., dtype=float32)
```

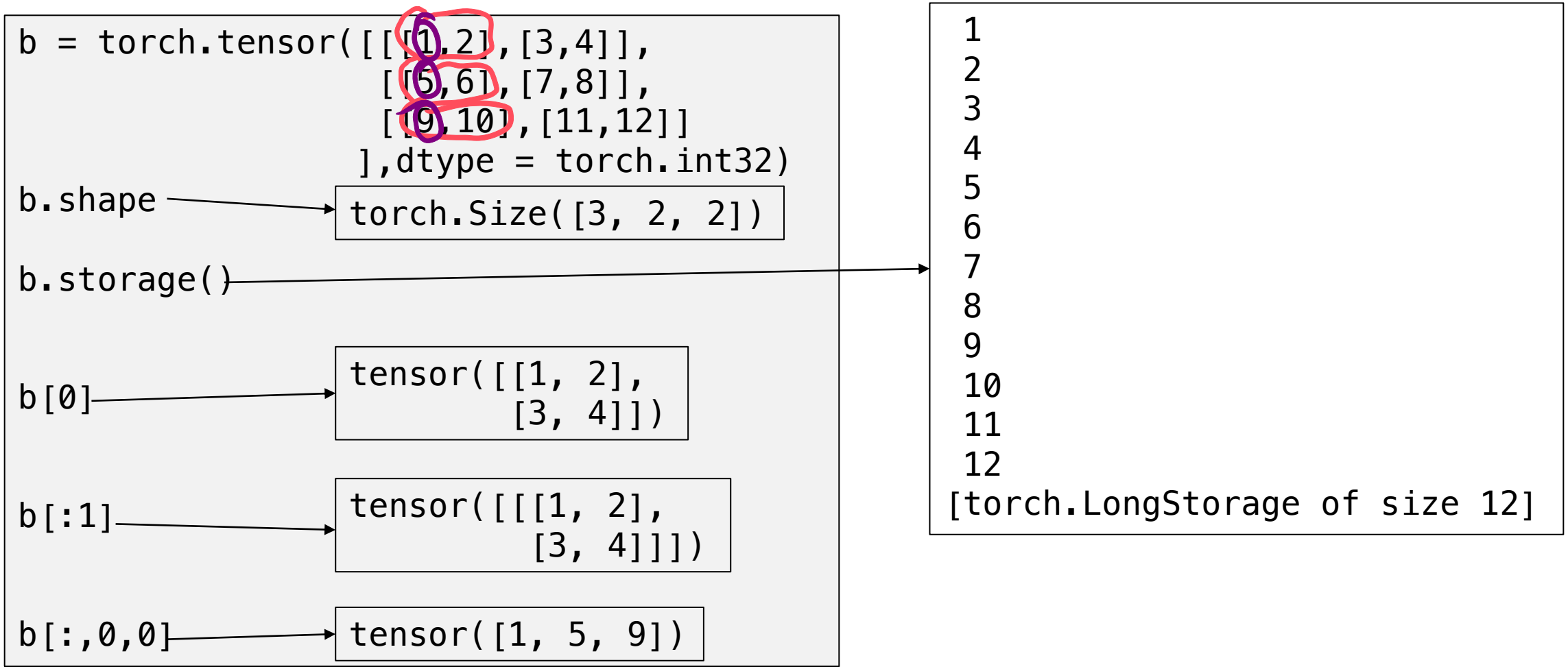
```
a = torch.tensor([1.0,2.0])
a.item()
```

```
a.numpy()
```

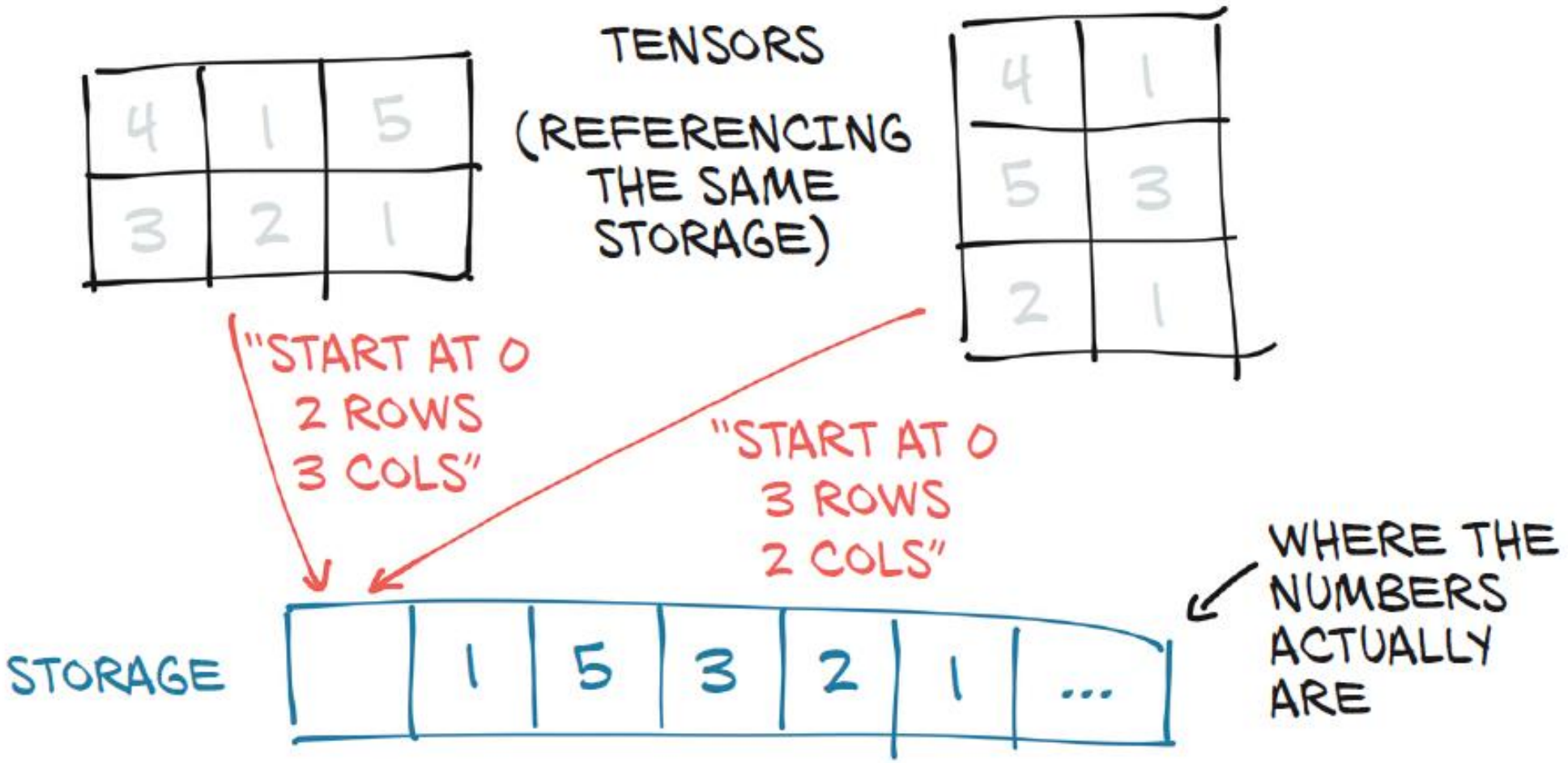
```
array([1., 2.], dtype=float32)
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-121-cd1bda83583f> in <module>()
----> 1 a.item()

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

访问Tensor的元素



访问Tensor的元素



访问Tensor的元素

```
a = torch.ones((3))
b = a[:2]
print(a)
print(b)
```

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

```
b[0] = 2
print(a)
print(b)
```

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

```
c = a[:2].clone()
c[0] = 3
print(a)
print(c)
```

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

访问Tensor的元素:掩膜

```
X = torch.randn((2,3))
mask = torch.randn((2,3))>0
X1 = X[mask]
X2 = torch.masked_select(X,mask)
print('X=',X.numpy())
print('mask=',mask.numpy())
print('X1=',X1.numpy())
print('X2=',X2.numpy())
```

```
X= [[-1.7936143 -1.0867784  0.6989719 ]
 [ 0.32036787 -0.29389462  0.6172063 ]]

mask= [[False  True  True]
 [False  True  True]]

X1= [-1.0867784   0.6989719  -0.29389462  0.6172063 ]

X2= [-1.0867784   0.6989719  -0.29389462  0.6172063 ]
```

访问Tensor的元素:掩膜

```
col_mask = torch.randn((3))>0
print('col_mask=',col_mask.numpy())
X3 = X[:,col_mask]
X4 = torch.masked_select(X,col_mask)
print('X3=',X3.numpy())
print('X4=',X4.numpy())
```

```
X= [[-1.7936143  -1.0867784   0.6989719 ]
 [ 0.32036787 -0.29389462  0.6172063 ]]

col_mask= [False  True  True]

X3= [[-1.0867784   0.6989719 ]
 [-0.29389462  0.6172063 ]]

X4= [-1.0867784   0.6989719  -0.29389462  0.6172063 ]
```

访问Tensor的元素:条件选择

```
x = torch.randn(1, 3)
y = torch.randn(1, 3)
z = torch.where(x>y,x,y)
```

z=x>y?x:y;

```
x tensor([[ 0.0684, -1.5483,  2.1476]])
y tensor([[-1.0761, -0.0047, -2.0822]])
z tensor([[ 0.0684, -0.0047,  2.1476]])
```

Transpose

```
a = torch.tensor([[[1,2],[3,4]],
                  [[5,6],[7,8]],
                  [[9,10],[11,12]]
                 ])
b = a.transpose(0,1)
print(a.stride())
print(b.stride())
```

```
(4, 2, 1)
(2, 4, 1)
```

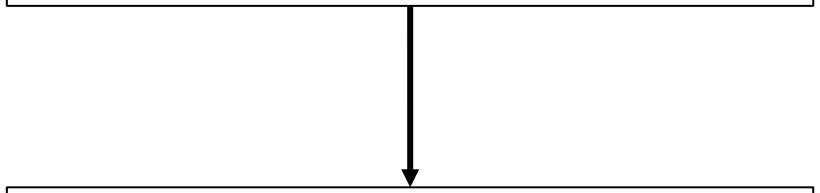
```
tensor([[[ 1,  2],
         [ 5,  6],
         [ 9, 10]],

        [[ 3,  4],
         [ 7,  8],
         [11, 12]]])
```

Transpose

```
a = torch.tensor([[[1,2],[3,4]],
                  [[5,6],[7,8]],
                  [[9,10],[11,12]]
                 ])
b = a.transpose(0,2)
```

3×2×2

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

        [[5,6],[7,8]],

        [[9,10],[11,12]]])
```



```
tensor([[[ 1,  5,  9],
         [ 3,  7, 11]],

        [[ 2,  6, 10],
         [ 4,  8, 12]]])
```

2×2×3

Transpose

```
a = torch.tensor([[3,1,2],[4,1,7]])
b = a.transpose(0,1)
print(a[0,0])
```

```
tensor(3)
```

```
b[0,0] = -1
print(a[0,0])
```

```
tensor(-1)
```

```
a = torch.tensor([[3,1,2],[4,1,7]])
b = a.clone().transpose(0,1)
print(a[0,0])
```

```
tensor(3)
```

```
b[0,0] = -1
print(a[0,0])
```

```
tensor(3)
```

Squeeze/unsqueeze

```
a = torch.tensor([[[1,2,3]]])
b = a.squeeze(dim=0)
c = a.squeeze()
d = b.unsqueeze(dim=1)
```

```
a.size()    torch.Size([1, 1, 3])
b.size()    torch.Size([1, 3])
c.size()    torch.Size([3])
d.size()    torch.Size([3, 1])
```

cat

```
x = torch.randn(2, 3)
x1 = torch.cat((x,x),dim = 0)
x2 = torch.cat((x,x),dim = 1)
```

```
x
tensor([[-0.4173,  0.0633, -1.3320],
        [ 0.5852, -2.0193,  0.2838]])
```

```
x1
tensor([[-0.4173,  0.0633, -1.3320],
        [ 0.5852, -2.0193,  0.2838],
        [-0.4173,  0.0633, -1.3320],
        [ 0.5852, -2.0193,  0.2838]])
```

4x3

```
x2
tensor([[-0.4173,  0.0633, -1.3320, -0.4173,  0.0633, -1.3320],
        [ 0.5852, -2.0193,  0.2838,  0.5852, -2.0193,  0.2838]])
```

2x6

reshape

```
T = torch.arange(0,12)
V1 = T.reshape((2,6))#也可以用torch.reshape(T,(2,6))
V2 = T.reshape((3,4))
print('T=',T)
print('V1=',V1)
print('V2=',V2)
```

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

V1= tensor([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11]])

V2= tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

permute

```
A = torch.arange(24).reshape((2,3,4))
print(A.shape)
print('A=',A)

B = A.permute((2,1,0))
print(B.shape)
print('B=',B)

A[0,0,1] = -5
print('B=',B)#注意B[1,0,0]的值变为-5
```

0,1,2

permutation

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

          [[12, 13, 14, 15],
           [16, 17, 18, 19],
           [20, 21, 22, 23]]])
torch.Size([4, 3, 2])
B= tensor([[[ 0, 12],
           [ 4, 16],
           [ 8, 20]],

          [[ 1, 13],
           [ 5, 17],
           [ 9, 21]],

          [[ 2, 14],
           [ 6, 18],
           [10, 22]],

          [[ 3, 15],
           [ 7, 19],
           [11, 23]]])
```

```
B= tensor([[[ 0, 12],
           [ 4, 16],
           [ 8, 20]],

          [[-5, 13],
           [ 5, 17],
           [ 9, 21]],

          [[ 2, 14],
           [ 6, 18],
           [10, 22]],

          [[ 3, 15],
           [ 7, 19],
           [11, 23]]])
```

张量运算

```
A = torch.tensor([1,2,3],dtype = torch.float32)
B = torch.tensor([-1,1,-1],dtype = torch.float32)

C = A + B
D = A - B
E = A*B
F = A/B
```

element-wise

```
A= [1. 2. 3.]
B= [-1.  1. -1.]
A+B= [0. 3. 2.]
A-B= [2. 1. 4.]
A*B= [-1.  2. -3.]
A/B= [-1.  2. -3.]
```

张量运算

```
A = torch.randn((2,3))
B = torch.randn((3,4))
C = A@B
print(A.shape)
print(B.shape)
print(C.shape)
```

矩阵乘法

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

张量运算

```
A = torch.tensor([[1,-1],[2,3],[4,5]])
B = A**2

print('A=',A.numpy())
print('B=',B.numpy())
```

```
A= [[ 1 -1]
 [ 2  3]
 [ 4  5]]
B= [[ 1  1]
 [ 4  9]
 [16 25]]
```

张量运算

```
W = torch.randn((10,784))
b = torch.randn((10,1))

x = torch.randn((28,28)).reshape((784,1))
y = W@x + b

print(y)
```

```
tensor([[ 29.2908],
        [-43.5128],
        [-27.3771],
        [ 27.4112],
        [ 27.8641],
        [ 26.4428],
        [-36.5835],
        [  5.1869],
        [ 51.5410],
        [-20.5552]])
```

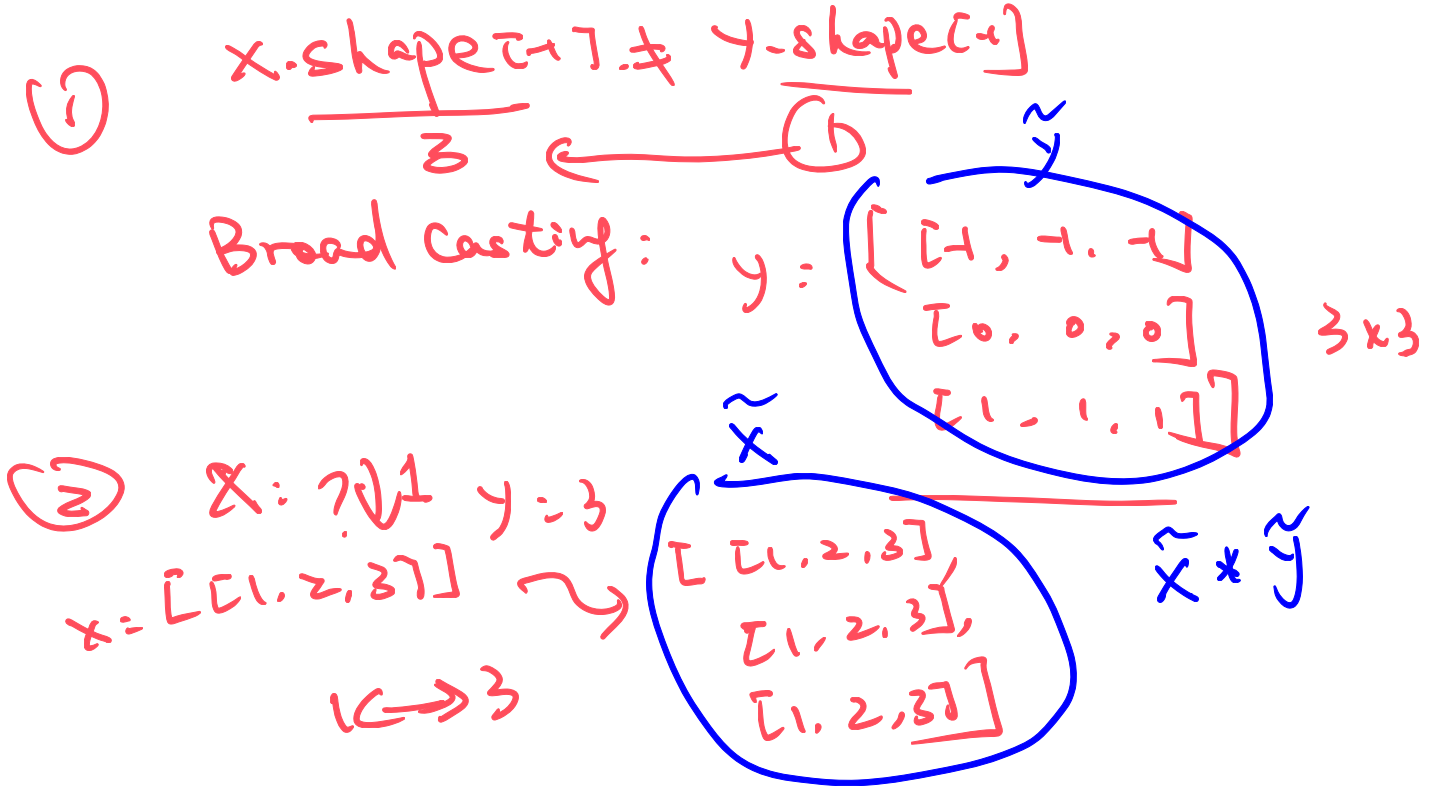
BroadCasting

```
x = torch.tensor([1,2,3])
y = torch.tensor([[-1],[0],[1]])
u = x * y
```

```
u: tensor([[-1, -2, -3],
           [ 0,  0,  0],
           [ 1,  2,  3]])
```

```
x: tensor([1, 2, 3])
y: tensor([[-1],
           [ 0],
           [ 1]])
```

```
x: torch.Size([3])
y: torch.Size([3, 1])

u: torch.Size([3, 3])
```



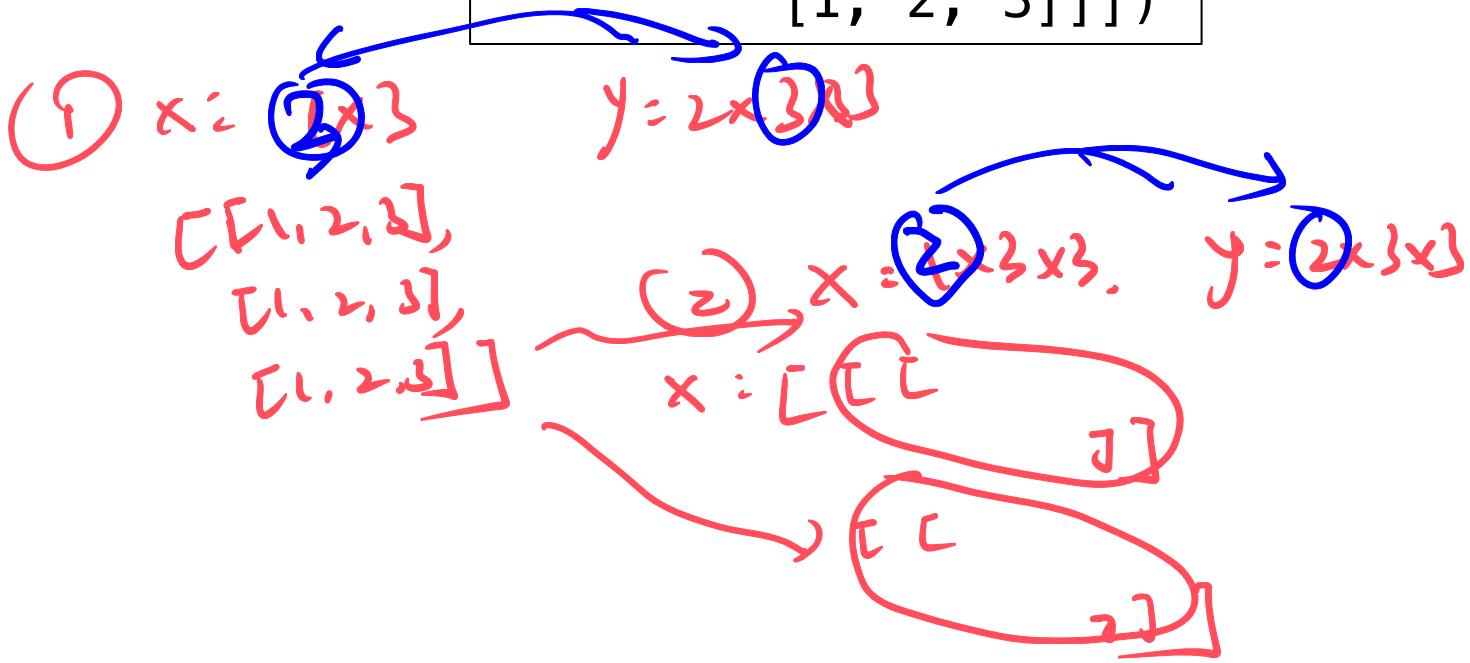
BroadCasting

```
x = torch.tensor([1,2,3])
z = torch.ones((2,3,3),dtype = torch.int64)
v = x * z
```

```
x: tensor([1, 2, 3])
z: tensor([[[1, 1, 1],
           [1, 1, 1],
           [1, 1, 1]],

          [[1, 1, 1],
           [1, 1, 1],
           [1, 1, 1]]])
```

```
x: torch.Size([3])
z: torch.Size([2, 3, 3])

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

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

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



BroadCasting

```
y = torch.tensor([[-1],[0],[1]])
z = torch.ones((2,3,3),dtype = torch.int64)
w = y * z
```

```
y: tensor([[-1],
        [ 0],
        [ 1]])
z: tensor([[[1, 1, 1],
         [1, 1, 1],
         [1, 1, 1]],

        [[1, 1, 1],
         [1, 1, 1],
         [1, 1, 1]]])
```

```
y: torch.Size([3, 1])
z: torch.Size([2, 3, 3])

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

```
w: tensor([[[-1, -1, -1],
         [ 0,  0,  0],
         [ 1,  1,  1]],

        [[-1, -1, -1],
         [ 0,  0,  0],
         [ 1,  1,  1]]])
```

例：彩色图像灰度化

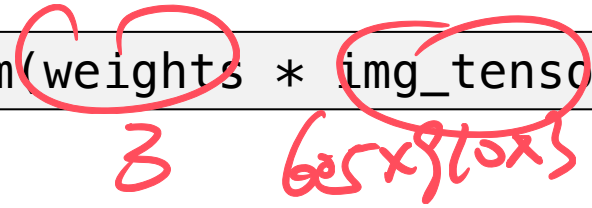
```
from PIL import Image
img = np.asarray(Image.open('balloons.jpg'))
img_tensor = torch.tensor(img).to(dtype = torch.float32)
print(img_tensor.size())
```

```
torch.Size([605, 910, 3])
```

```
weights = torch.tensor([0.2126,0.7152,0.0722])
```

```
gray = torch.sum(weights * img_tensor,dim = 2)
```

```
torch.Size([605, 910])
```




```
img_tensor1 = img_tensor.permute([2,0,1])
```

```
torch.Size([3, 605, 910])
```

```
img_r,img_g,img_b = img_tensor1[0],img_tensor1[1],img_tensor1[2]
```

例2：图像标准化

```
img_files = ['balloons.jpg','happydog.jfif','sunflower.jpg','Woolsthorpe-Manor.jpg']

img_batch = torch.zeros(len(img_files),3,480,640,dtype = torch.uint8)

for i,f in enumerate(img_files):
    img = np.array(Image.open(f).resize((640,480)))
    img_tensor = torch.from_numpy(img)
    img_tensor = img_tensor.permute(2,0,1)
    img_batch[i] = img_tensor
```

```
img_batch : torch.Size([4, 3, 480, 640])
```

例2：图像标准化

```
img_batch = img_batch.float() / 255.0
mean = torch.tensor([0.485, 0.456, 0.406])
std = torch.tensor([0.229, 0.224, 0.225])


print('batch:',img_batch.size())
print('mean:',mean.size())
print('std:',std.size())
```

```
batch: torch.Size([4, 3, 480, 640])
mean: torch.Size([3])
std: torch.Size([3])
```

```
mean_unsqueezed = mean.unsqueeze(dim = 1).unsqueeze(dim=1)
std_unsqueezed = std.unsqueeze(dim = 1).unsqueeze(dim = 1)
print(mean_unsqueezed.size())
```

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

```
img_batch_normalized = (img_batch - mean_unsqueezed) / std_unsqueezed
```

例3：使用张量编写神经元模型

```
class Neuron():
    def __init__(self, in_features):
        self.dim = in_features
        self.W = torch.zeros((1,self.dim))
        self.b = torch.zeros(1)

    def __sigmoid__(self, z):
        return 1/(1 + torch.exp(-z))

    def __transfer__(self, x):
        return self.W@x + self.b
```

例3：使用张量编写神经元模型

```
def __update__(self, dW,db,lr):
    self.W = self.W + lr * dW
    self.b = self.b + lr * db

def __calc_loss__(self, Y,rho):
    loss = -torch.log(rho[Y==1]).sum() - torch.log(1 - rho[Y==0]).sum()
    loss = loss / Y.shape[0]
    return loss
```

例3：使用张量编写神经元模型

```
    def __backward__(self,Y,rho):
        err = Y - rho
        dW = err @ X.T/Y.shape[0]
        db = err.mean()
        return dW,db

    def predict(self, x):
        z = self.__transfer__(x)
        rho = self.__sigmoid__(z)
        return rho
```

例3：使用张量编写神经元模型

```
def fit(self, X,Y,max_iter=100,lr = 0.1):
    n = X.shape[1]
    assert(X.shape[0]==self.dim)
    assert(n==Y.shape[0])

    for iter in range(max_iter):
        rho = self.predict(X).squeeze()
        loss = self.__calc_loss__(Y,rho)
        print('iter=',iter,',loss=',loss.item())
        dW,db = self.__backward__(Y,rho)
        self.__update__(dW,db,lr)
```

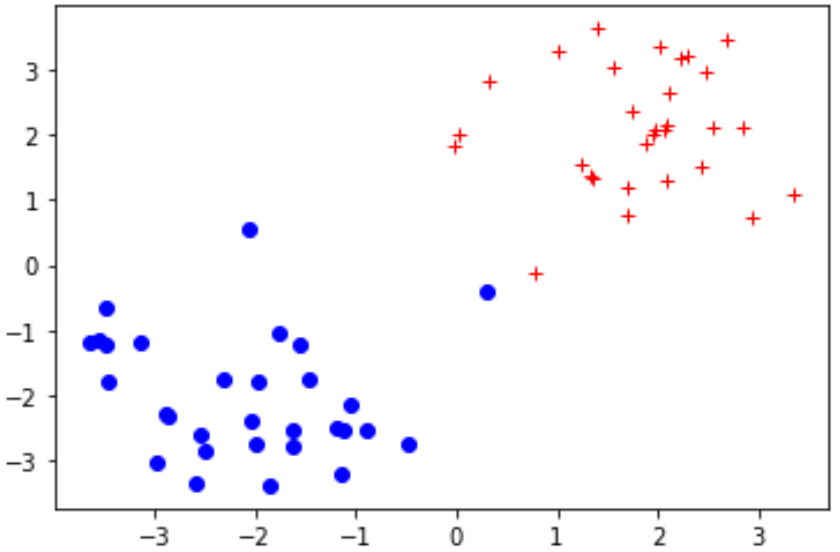
F.P.
calc loss
B.P.
G.D.

例3：使用张量编写神经元模型

```
def GenerateSamples(n):
    x1 = torch.randn((2,n)) + 2
    x2 = torch.randn((2,n)) - 2
    y1 = torch.ones((n))
    y2 = torch.zeros((n))
    x = torch.cat((x1,x2),dim = 1)
    y = torch.cat((y1,y2),dim = 0)


    return x,y
```

例3：使用张量编写神经元模型

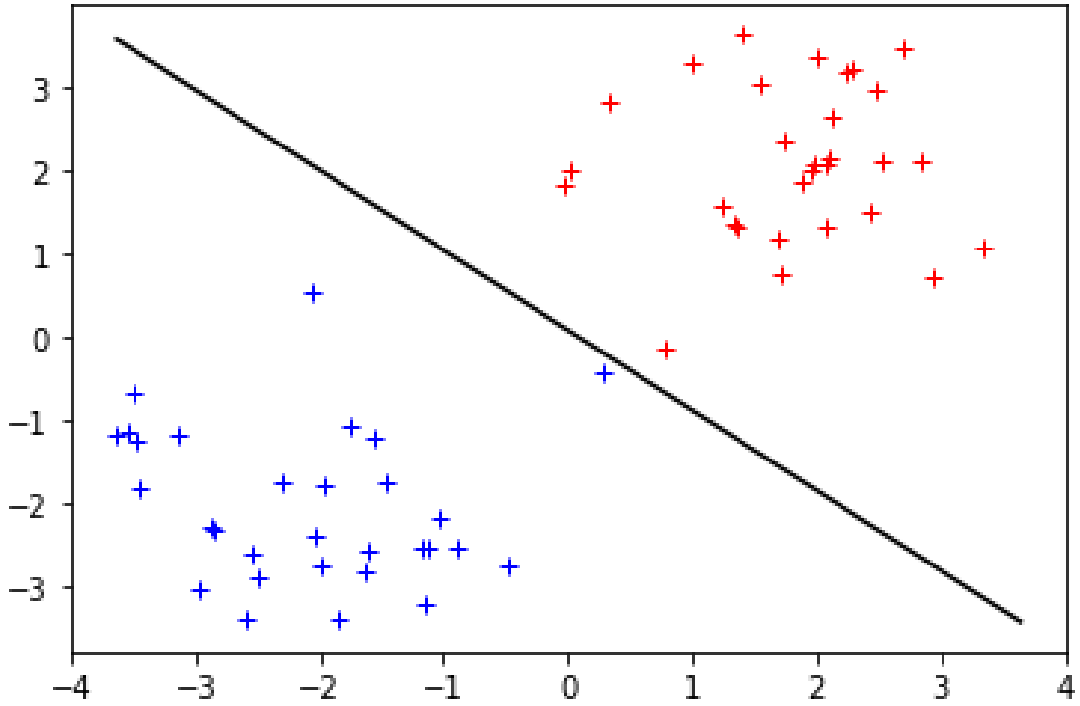
```
X,Y = GenerateSamples(30)
print(X.shape)
plt.plot(X[0,Y==1],X[1,Y==1],'r+')
plt.plot(X[0,Y==0],X[1,Y==0],'bo’)


cell = Neuron(2)
cell.fit(X,Y,100,1)
```




```
iter= 0 ,loss= 0.6931470632553101
iter= 1 ,loss= 0.04438574239611626
iter= 2 ,loss= 0.041226837784051895
iter= 3 ,loss= 0.03874609619379043 6
iter= 4 ,loss= 0.03673584014177322 4
iter= 5 ,loss= 0.03506697143287 66
iter= 6 ,loss= 0.033654589205980 3
iter= 7 ,loss= 0.03244034573435783 4
```

例3：使用张量编写神经元模型

```
w1,w2 = cell.W[0,0],cell.W[0,1]
b = cell.b
minx = torch.min(X,dim = 1)
maxx = torch.max(X,dim = 1)

xs = torch.cat((minx[0],maxx[0]))
ys = -(xs*w1+b)/w2

plt.plot(X[0,Y==1],X[1,Y==1],'r+')
plt.plot(X[0,Y==0],X[1,Y==0],'b+')
plt.plot(xs,ys,'k-')
```

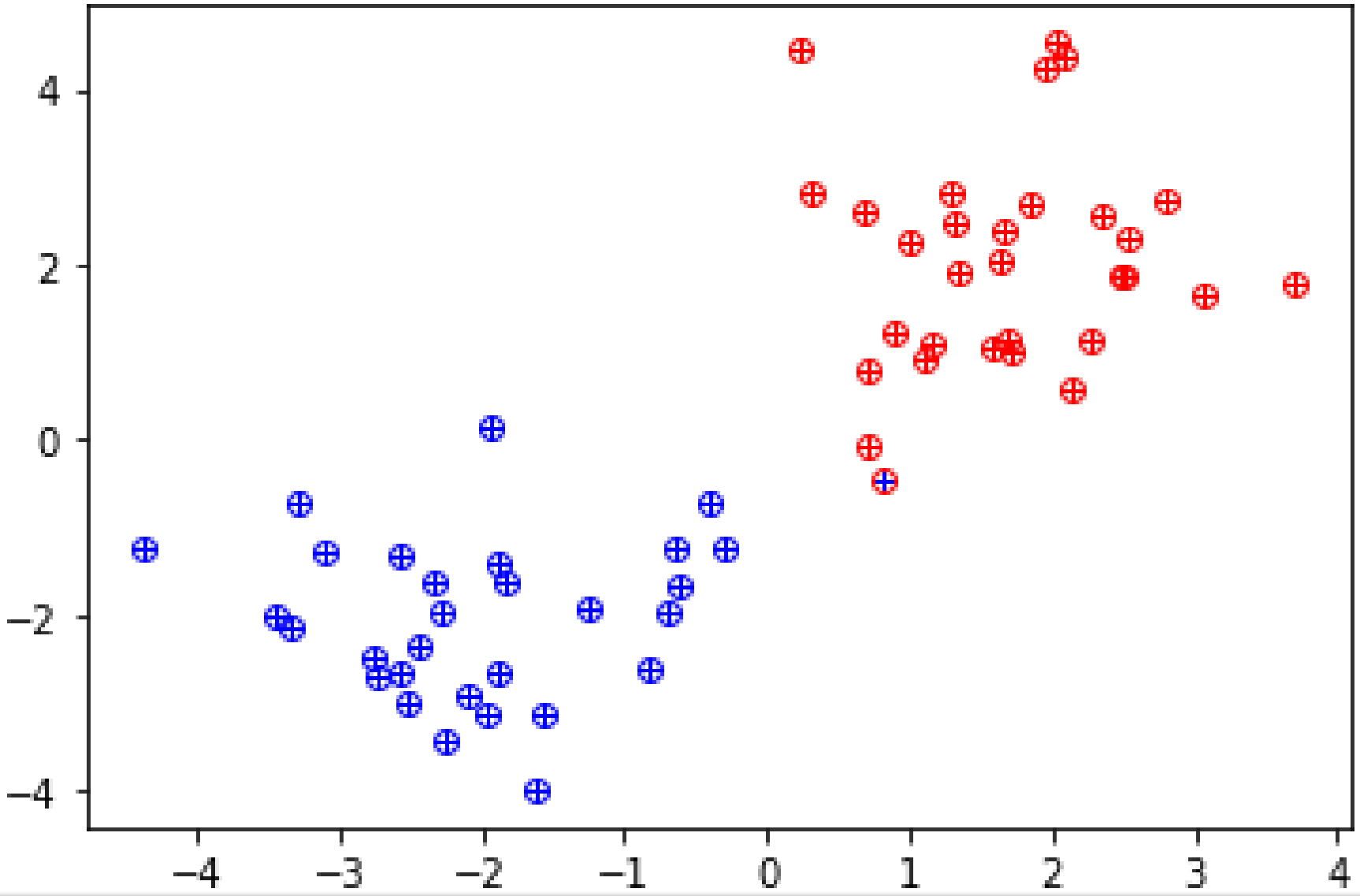


例3：使用张量编写神经元模型

```
#测试集预测效果
X_test,Y_test = GenerateSamples(30)
Y_hat = torch.where(cell.predict(X_test).squeeze()>0.5,1,0)


plt.plot(X_test[0,Y_test==1],X_test[1,Y_test==1],'r+')
plt.plot(X_test[0,Y_test==0],X_test[1,Y_test==0],'b+')
plt.plot(X_test[0,Y_hat==1],X_test[1,Y_hat==1],'ro',fillstyle='none')
plt.plot(X_test[0,Y_hat==0],X_test[1,Y_hat==0],'bo',fillstyle='none')


acc = torch.mean((Y_hat==Y_test).to(torch.float32)).item()
print('Test Acc = ',acc)
```

```
Test Acc = 0.983333492279053
```

例3：使用张量编写神经元模型

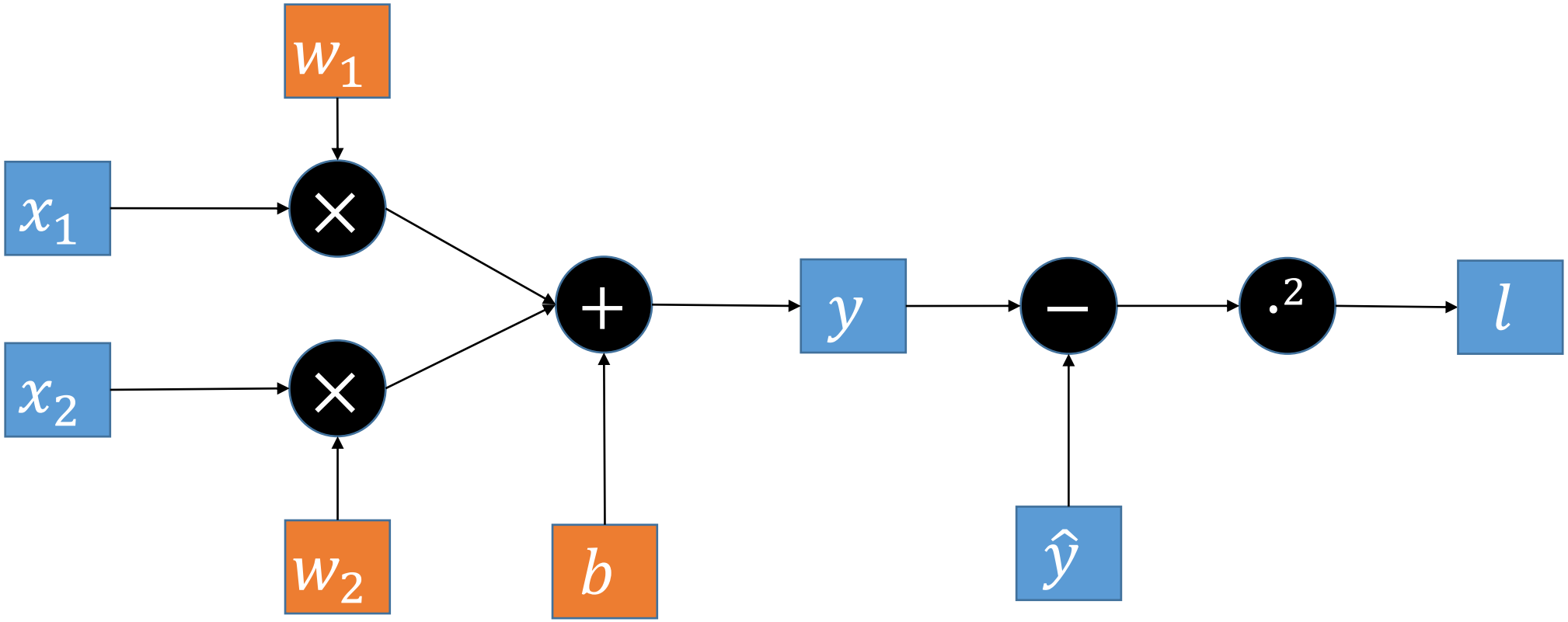


练习一：使用Tensor实现感知器模型

1.2 自动求导(AutoGrad)

计算图

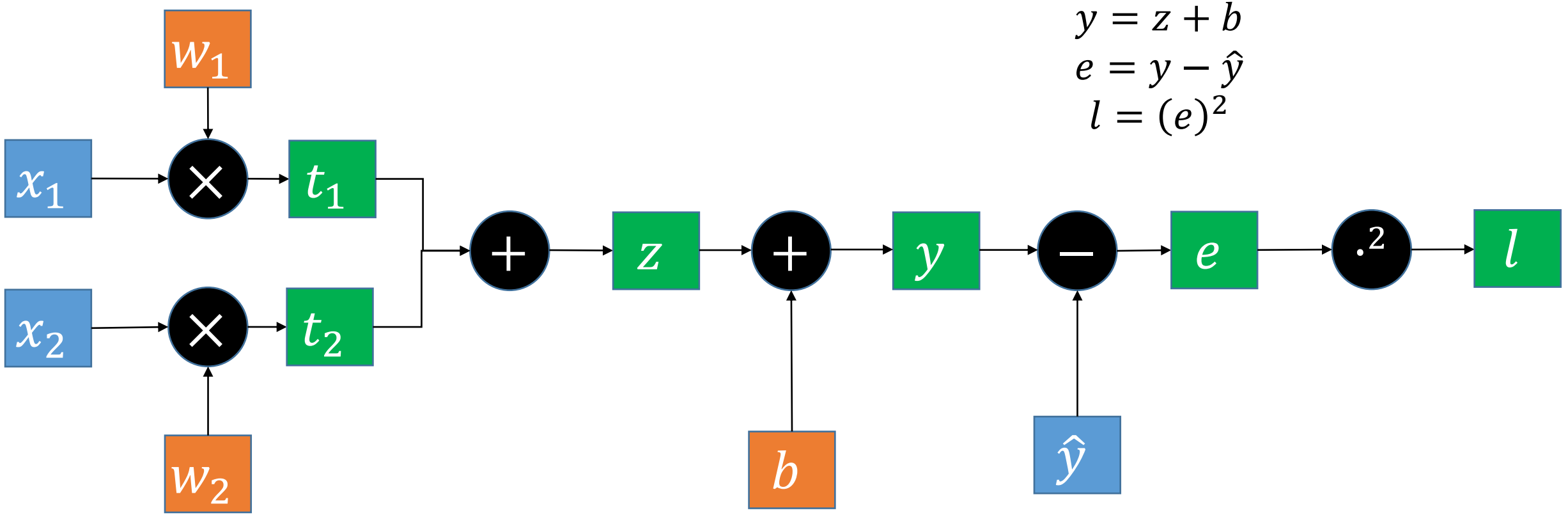
$$y = h(x; w, b) = w_1 \times x_1 + w_2 \times x_2 + b$$
$$l = (y - \hat{y})^2$$



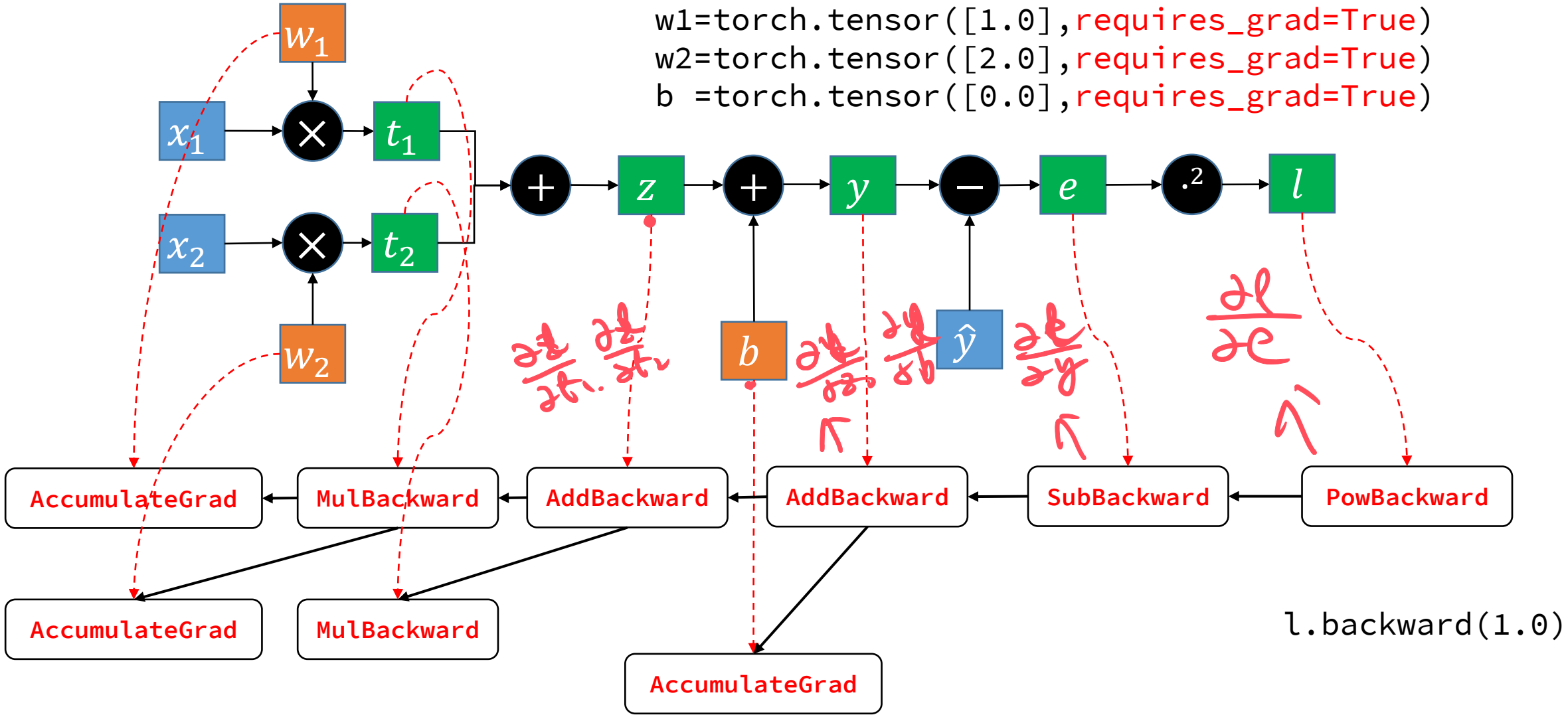
计算图

$$y = h(x; w, b) = w_1 \times x_1 + w_2 \times x_2 + b$$
$$l = (y - \hat{y})^2$$

$$t_1 = w_1 \times x_1$$
$$t_2 = w_2 \times x_2$$
$$z = t_1 + t_2$$
$$y = z + b$$
$$e = y - \hat{y}$$
$$l = (e)^2$$



计算图


w1=torch.tensor([1.0],requires_grad=True)
w2=torch.tensor([2.0],requires_grad=True)
b =torch.tensor([0.0],requires_grad=True)
w_1
x_1
t_1
x_2
t_2
w_2
z
b
y
$\hat{y}$
e
$.^2$
l
AccumulateGrad
MulBackward
AddBackward
AddBackward
SubBackward
PowBackward
AccumulateGrad
MulBackward
AccumulateGrad
l.backward(1.0)

$$y = h(x; w, b) = w * x + b$$
$$loss = (y - \hat{y})^2$$

```
x = torch.tensor([1.0])
w = torch.tensor([1.0],requires_grad = True)
y = w*x
print(y)
```

```
tensor([1.], grad_fn=<MulBackward0>)
```

```
loss = (y - 2)**2
print(loss)
```

```
tensor([1.], grad_fn=<PowBackward0>)
```

```
loss.backward()
print(w.grad)
```

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

```
with torch.no_grad():
    w -= 0.1*w.grad
print(w)
```

```
tensor([1.2000], requires_grad=True)
```

```
x = torch.tensor([1.0])
w = torch.tensor([1.0],requires_grad = True)
y = w*x

for epoch in range(3):
    print('epoch:%d'%(epoch))
    loss = (y - 2)**2
    loss.backward()
    print(w.grad)
```

```
epoch:0
tensor([-2.])
epoch:1
```

```
RuntimeError: Trying to backward through
the graph a second time, but the buffers
have already been freed. Specify
retain_graph=True when calling backward
the first time.
```

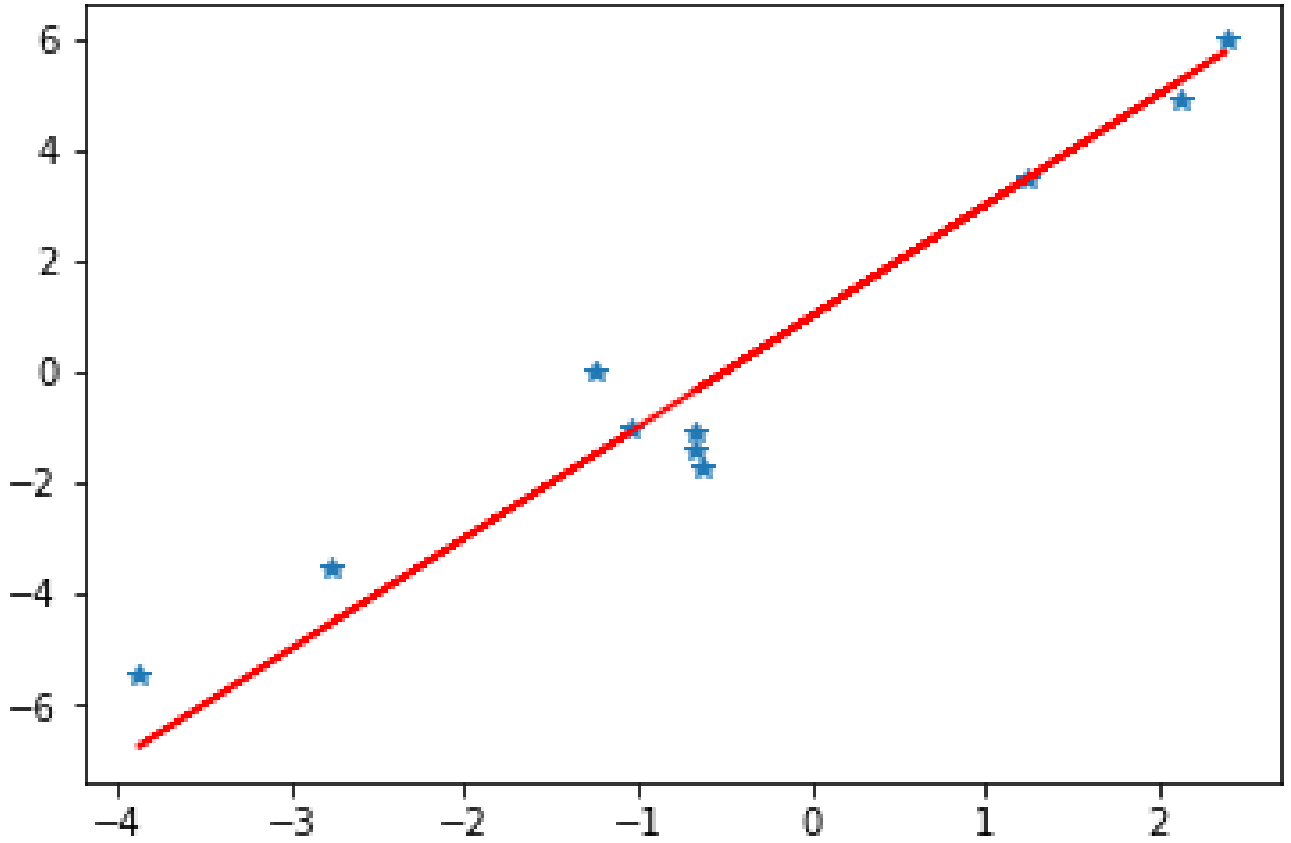
```
x = torch.tensor([1.0])
w = torch.tensor([1.0],requires_grad = True)
y = w*x

for epoch in range(3):
    print('epoch:%d'%(epoch))
    loss = (y - 2)**2
    loss.backward(retain_graph=True)
    print(w.grad)
```

```
epoch:0
tensor([-2.])
epoch:1
tensor([-4.])
epoch:2
tensor([-6.])
```

```
x = torch.tensor([1.0])
w = torch.tensor([1.0],requires_grad = True)
y = w*x

for epoch in range(3):
    print('epoch:%d'%(epoch))
    if w.grad is not None:
        w.grad.zero_()
    loss = (y - 2)**2
    print(w.grad)
    loss.backward(retain_graph=True)
    print(w.grad)
```

```
epoch:0
None
tensor([-2.])
epoch:1
tensor([0.])
tensor([-2.])
epoch:2
tensor([0.])
tensor([-2.])
```

例1：一个简单的一元线性回归问题



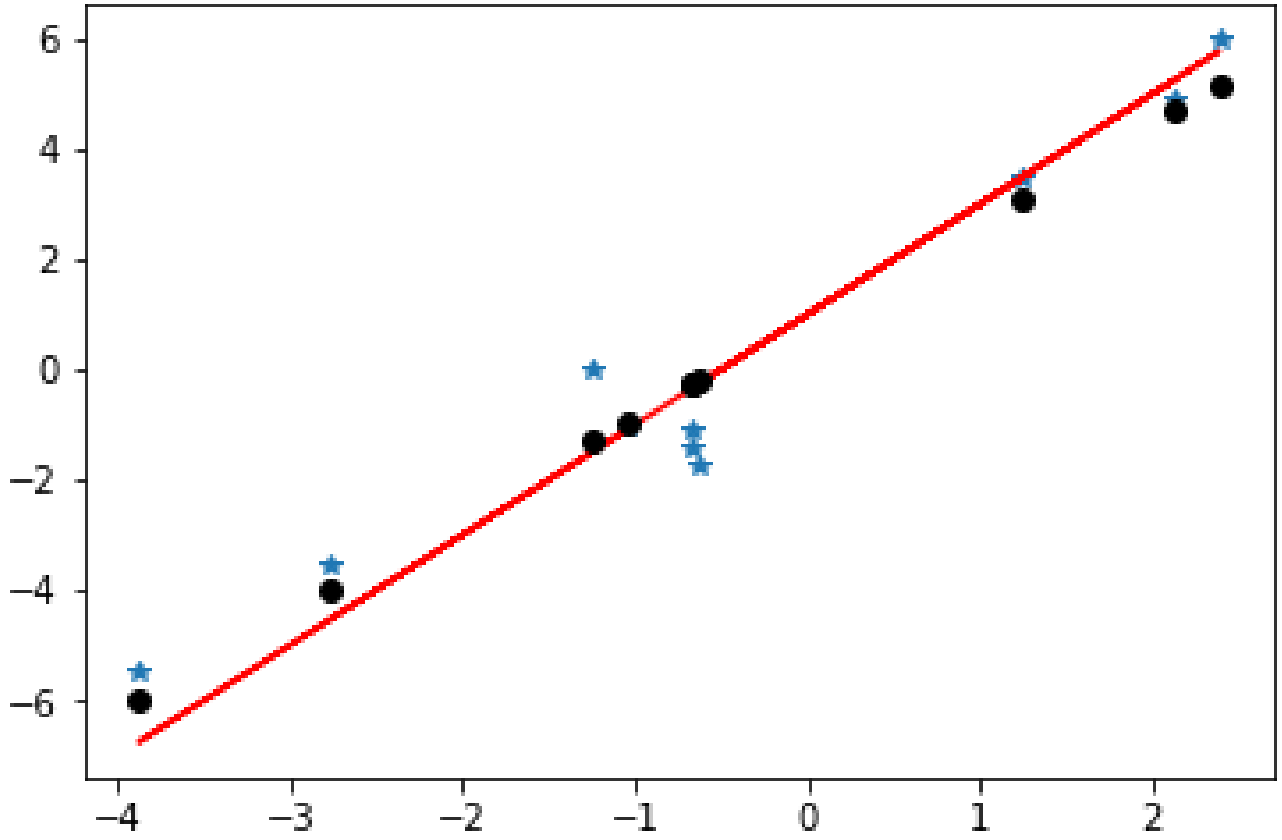
```
#生成训练数据
x = torch.rand(10)*10 - 5 #[-5,5]上的均匀分布随机数
y = 2*x + 1 + torch.randn(10) #受随机噪声污染的y

#定义回归模型
def linearReg(x,w,b):
    y = w * x + b
    return y

#定义损失函数
def lossFn(y,y_hat):
    squared_errors = (y - y_hat)**2
    loss = squared_errors.mean()
    return loss
```

```
#定义训练函数
def trainModel(x,y,w,b,epochs,lr = 0.1):
    for epoch in range(1, epochs + 1):
        if w.grad is not None:
            w.grad.zero_()
        if b.grad is not None:
            b.grad.zero_()

        y_pred = linearReg(x,w,b)

        loss = lossFn(y_pred,y)

        loss.backward()

        with torch.no_grad():
            w -= lr * w.grad
            b -= lr * b.grad

        print('Epoch = %d, Loss = %f, w = %f,b = %f'%(
            epoch,loss.detach().numpy(),w.detach().numpy(),b.detach().numpy()))
    return w,b
```

```
#训练模型
#初始化参数w,b
w = torch.tensor([0.0],requires_grad = True)
b = torch.tensor([0.0],requires_grad = True)
w,b = trainModel(x,y,w,b,epochs = 10, lr = 0.2)
```

```
Epoch = 1, Loss = 12.116911, w = 2.548692,b = 0.003886
Epoch = 2, Loss = 4.572442, w = 1.183241,b = 0.532312
Epoch = 3, Loss = 2.037711, w = 2.024282,b = 0.567515
Epoch = 4, Loss = 1.181779, w = 1.580700,b = 0.762241
Epoch = 5, Loss = 0.891036, w = 1.858682,b = 0.787515
Epoch = 6, Loss = 0.791604, w = 1.714883,b = 0.860059
Epoch = 7, Loss = 0.757336, w = 1.806942,b = 0.873903
Epoch = 8, Loss = 0.745425, w = 1.760451,b = 0.901212
Epoch = 9, Loss = 0.741245, w = 1.791010,b = 0.908000
Epoch = 10, Loss = 0.739764, w = 1.776030,b = 0.918381
```

```
y_pred = linearReg(x,w,b)
plt.plot(x.numpy(),y.numpy(),'*')
plt.plot(x.numpy(),2*x.numpy()+1,'r-')
plt.plot(x.numpy(),y_pred.detach().numpy(),'ko')
plt.show()
```

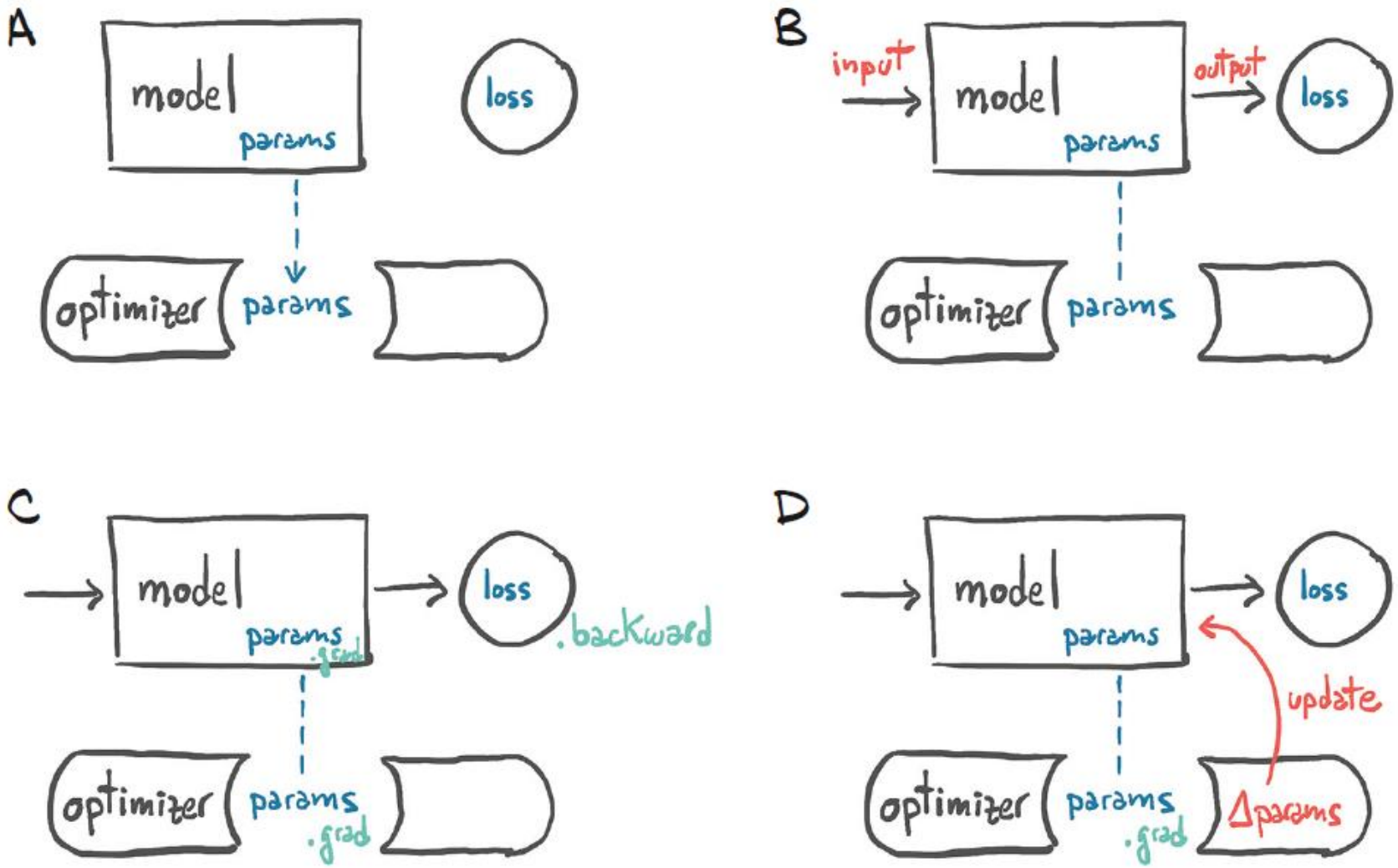


使用torch.optim训练模型

```
#定义训练函数
def trainModel(x,y,w,b,epochs,lr = 0.1):
    for epoch in range(1, epochs + 1):
        if w.grad is not None:
            w.grad.zero_()
        if b.grad is not None:
            b.grad.zero_()

        y_pred = linearReg(x,w,b)

        loss = lossFn(y_pred,y)

        loss.backward()

        with torch.no_grad():
            w -= lr * w.grad
            b -= lr * b.grad

    return w,b
```

```
import torch.optim as optim

def trainModel(x,y,w,b,epochs,lr = 0.1):

    optimizer = optim.SGD([w,b],lr = lr)

    for epoch in range(1, epochs + 1):

        optimizer.zero_grad()

        y_pred = linearReg(x,w,b)

        loss = lossFn(y_pred,y)

        loss.backward()

        optimizer.step()

    return w,b
```

使用torch.optim训练模型



练习二

- 使用Tensor和自动求导编写线性回归器

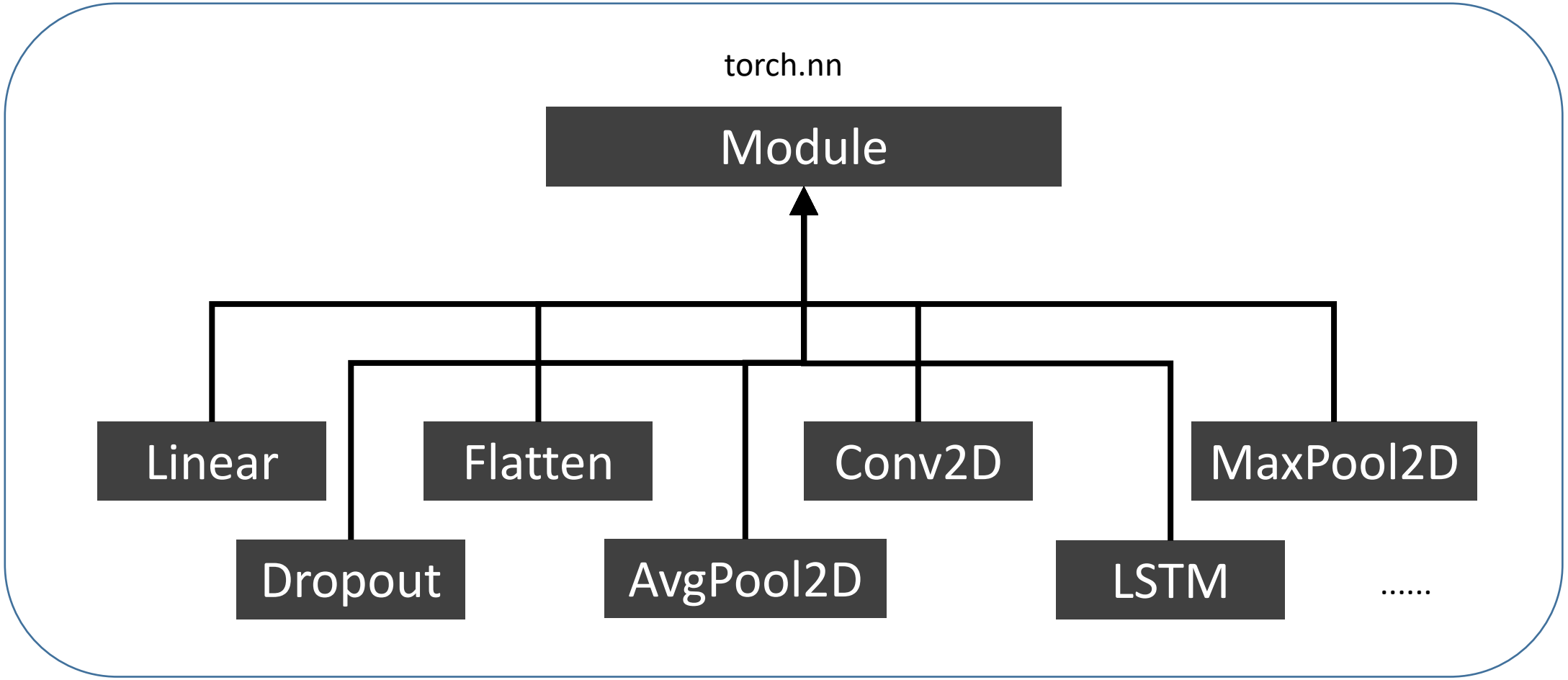
2. 多层感知器编程

torch.nn

多层感知器编程

- 编写网络模型
  - torch.nn.Sequential：编写简单的模型
  - torch.nn.Module：编写复杂的模型
- 训练模型：
  - 数据加载器
  - 损失函数：torch.nn
  - 优化器：torch.nn.optim

torch.nn.Module

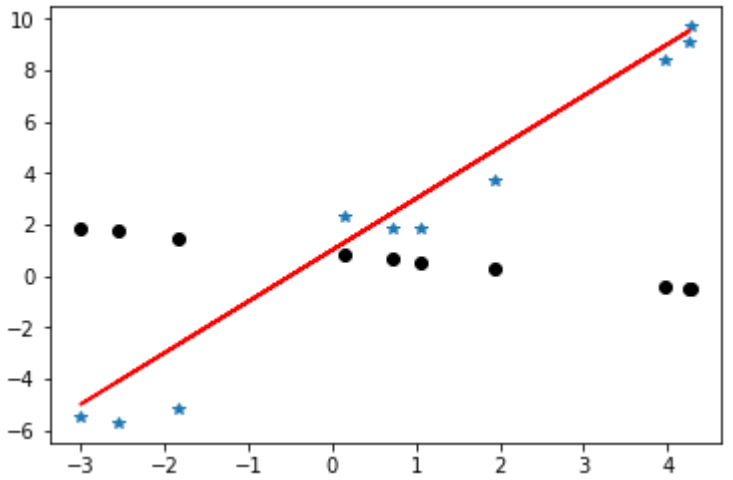


2.1 实现简单的模型

线性回归

```
torch.nn.Linear(in_features: int, out_features: int, bias: bool = True)
```

```
import torch.nn as nn
linReg = nn.Linear(1,1)

x = torch.rand(10)*10 - 5
y = 2*x + 1 + torch.randn(10)

y_pred = linReg(x.view(10,1))

plt.plot(x.numpy(),y.numpy(),'*')
plt.plot(x.numpy(),2*x.numpy()+1,'r-')
plt.plot(x.numpy(),y_pred.detach().numpy(),'ko')
plt.show()
```



线性回归

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

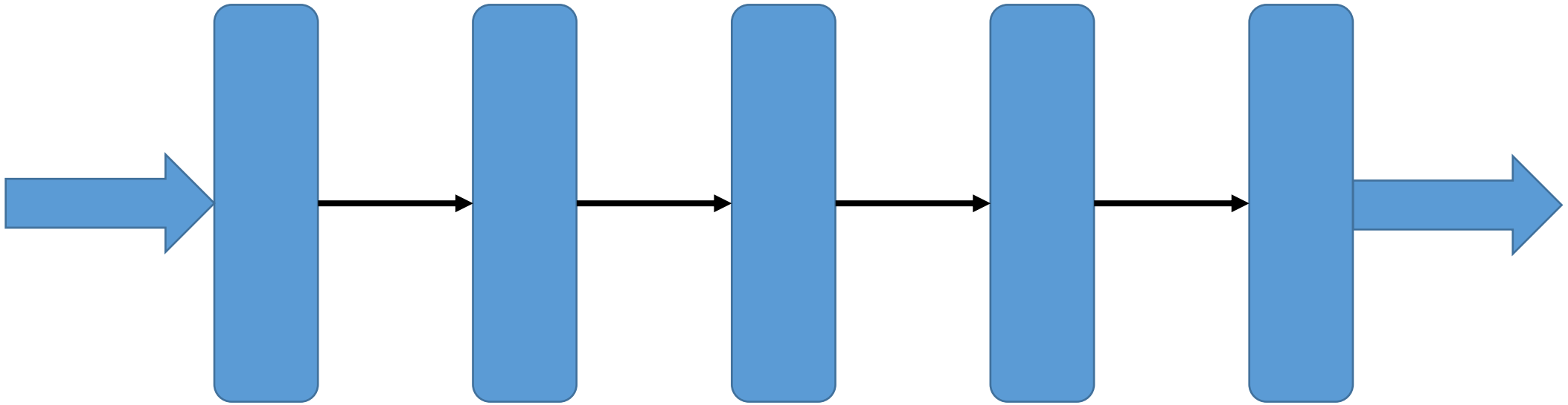
```
linReg = nn.Linear(1,1)
loss_fn = nn.MSELoss(reduction='mean')
linReg.train()
optimizer = optim.SGD(params = linReg.parameters(),lr = 0.1,momentum = 0.9)
for epoch in range(10):
    optimizer.zero_grad()
    y_pred = linReg(x.view(10,1))
    loss = loss_fn(y_pred, y)
    loss.backward()
    optimizer.step()
```

F.P.
Calc Loss
B.P.
G.D.

```
x_train,y_train = readData('circle-train.csv')
x = torch.tensor(x_train,dtype = torch.float32)
y = torch.tensor(y_train[:,np.newaxis],dtype = torch.float)

logReg = nn.Linear(2,1)
loss_fn = nn.BCEWithLogitsLoss()

logReg.train()
optimizer = optim.SGD(params = logReg.parameters(),lr = 0.1,momentum = 0.9)
for epoch in range(10):
    optimizer.zero_grad()
    y_pred = logReg(x)
    loss = loss_fn(y_pred, y)
    loss.backward()
    optimizer.step()
```

2.2 使用Sequential构造神经网络模型

构造序列式神经网络模型

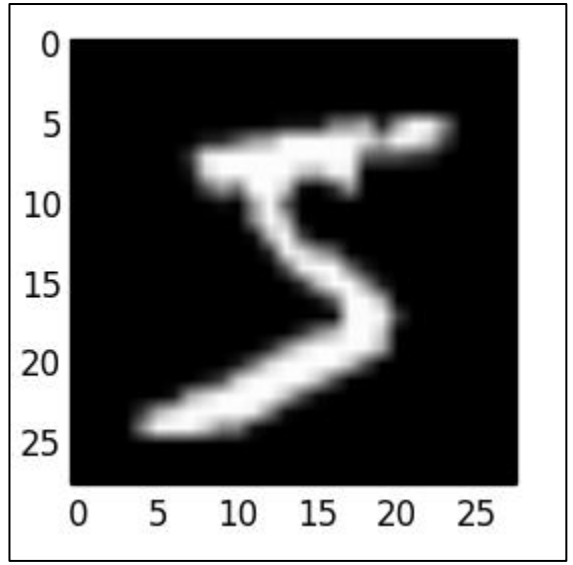


torch.nn.Sequential

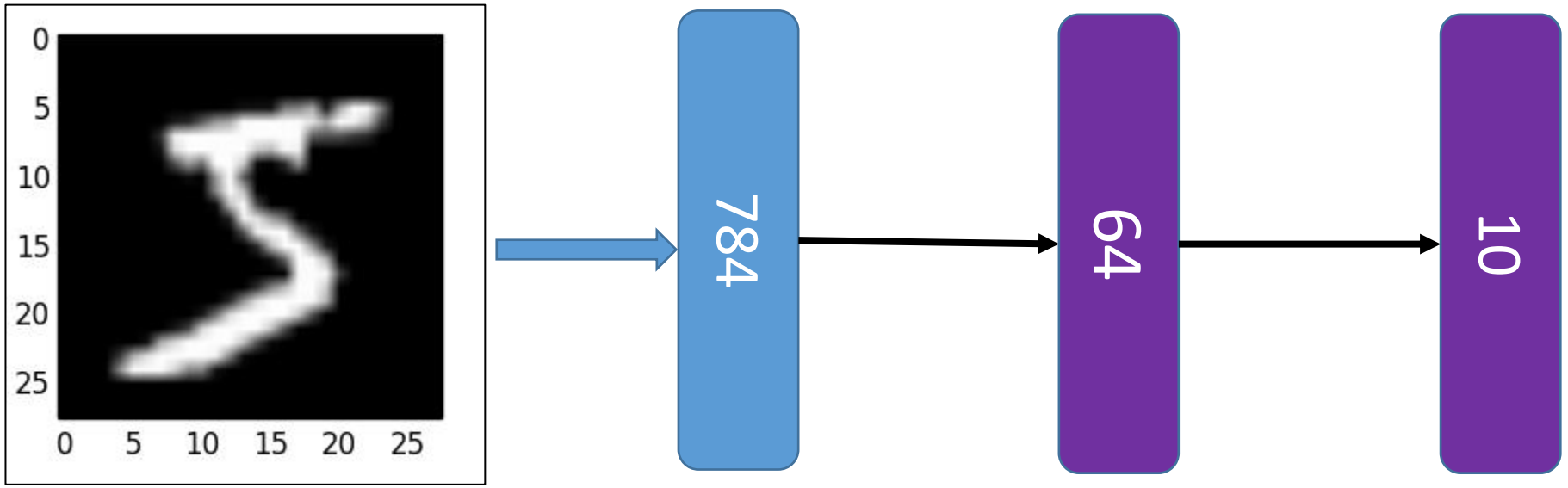
```
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )
```

```
from collections import OrderedDict

model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

MNIST手写体识别



MNIST手写体识别:设计一个单隐层神经网络



$h = \sigma(W_h x + b_h)$ $p = Softmax(W_o h + b_o)$

torch.nn.Linear

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

$$out = W \times input + b$$

torch.nn.Linear

```
x = torch.tensor([1.0,2.0,3.0])
lin = torch.nn.Linear(3,2)
y = lin(x)
print(y)
```

```
tensor([-0.7312,  0.5305], grad_fn=<AddBackward0>)
```

```
for v in lin.parameters():
    print(v)
```

```
Parameter containing:
tensor([[-0.0212,  0.2302, -0.4544],
        [ 0.5579,  0.4064, -0.2442]], requires_grad=True)
Parameter containing:
tensor([ 0.1929, -0.1078], requires_grad=True)
```

torch.nn.Softmax

```
torch.nn.Softmax(dim=None)
```

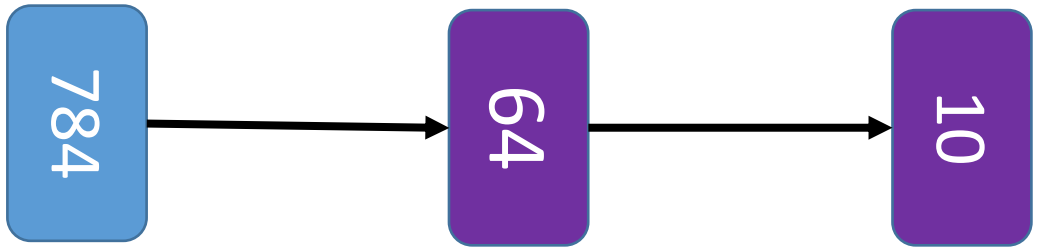
```
x = torch.tensor([[1.0,3.0,5.0],[4.0,5.0,1.0]])
prob1 = torch.nn.Softmax(dim=0)
y1 = prob1(x)
```

```
tensor([[0.0474, 0.1192, 0.9820],
        [0.9526, 0.8808, 0.0180]])
```

```
prob2 = torch.nn.Softmax(dim=1)
y2 = prob2(x)
```

```
tensor([[0.0159, 0.1173, 0.8668],
        [0.2654, 0.7214, 0.0132]])
```

MNIST手写体识别：构造神经网络



$h = \sigma(W_h x + b_h)$　　$z = W_o h + b_o$

```
mnist_mlp = nn.Sequential(nn.Linear(784,64),
                          nn.Sigmoid(),
                          nn.Linear(64,10))

optimizer = optim.SGD(mnist_mlp.parameters(), lr = 0.01)

loss_fn = nn.CrossEntropyLoss()
```

torch.nn.CrossEntropyLoss

$$l(\theta; z, y) = -\log\left(\frac{e^{z_y}}{\sum_{j=1}^{k} e^{z_j}}\right), z = h(x; \theta) = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$

torch.nn.NLLLoss

$$l(\theta; z, y) = -z_y, y \in \{1,2, \dots, k\}$$

$$z_j = \log(\rho_j), \rho_j = \frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_i}}$$

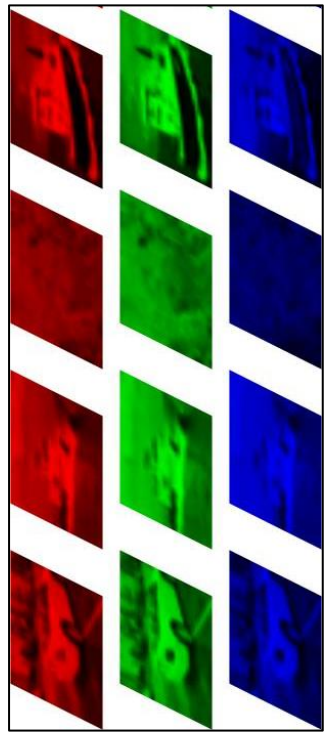
MNIST手写体识别：数据集与加载器

```
from torchvision import datasets,transforms


data_path = '../data/'
mnist = datasets.MNIST(data_path,download=True,
                       transform = transforms.ToTensor())
img,label = mnist[0]
img.shape                                  torch.Size([1, 28, 28])
img = img.view(-1)
img.shape                                  torch.Size([784])

mnist_loader = torch.utils.data.DataLoader(mnist, batch_size = 64,
                                           shuffle = True)
```

MNIST手写体识别：数据集与加载器

```
mnist_loader = torch.utils.data.DataLoader(mnist,
                                           batch_size = 64,
                                           shuffle = True)


for i,data in enumerate(mnist_loader):
    print(data[0].shape)
    print(data[1].shape)
    break
```

```
torch.Size([64, 1, 28, 28])
torch.Size([64])
```



MNIST手写体识别：训练神经网络

```
epochs = 10
mnist_mlp.train()
for epoch in range(epochs):
    for imgs, labels in mnist_loader:
        batch_size = imgs.shape[0]
        logits = mnist_mlp(imgs.view(batch_size,-1))
        loss = loss_fn(logits,labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

MNIST手写体识别：测试模型

```
test_loader = torch.utils.data.DataLoader(mnist_test,
                                          batch_size = 100, shuffle = False)
mnist_mlp.eval()
correct = 0
total = 0
with torch.no_grad():
    for imgs,labels in test_loader:
        batch_size = imgs.shape[0]
        logits = mnist_mlp(imgs.view(batch_size,-1))
        _,predicted = torch.max(logits.data,1)
        total += batch_size
        correct += (predicted == labels).sum().item()
acc = correct / total
```

2.2 子类化 torch.nn.Module

子类化torch.nn.Module

- 从torch.nn.Module派生一个子类
- 编写__init__函数
  - 初始化对象
  - 要调用父类的构造函数__init__
- 编写forward函数
  - 构造网络

使用nn.Module构造网络模型

```
class mlp(nn.Module):

    def __init__(self, in_dim, out_dim):
        super(mlp,self).__init__()
        self.lin1 = nn.Linear(in_features=in_dim, out_features=64)
        self.relu = nn.ReLU(inplace=True)
        self.lin2 = nn.Linear(in_features=64,out_features=out_dim)


    def forward(self, x):
        x = self.lin1(x)
        x = self.relu(x)
        x = self.lin2(x)

        return x
```

```
    mnist_mlp = mlp(784,10)
    print(mnist_mlp)
```

```
mlp(
  (lin1): Linear(in_features=784, out_features=64, bias=True)
  (relu): ReLU(inplace)
  (lin2): Linear(in_features=64, out_features=10, bias=True)
)
```

```
x = torch.randn((1,784),dtype = torch.float32)
y = F.softmax(mnist_mlp(x),dim=1)
print(y.detach().numpy())
```

```
[[0.07569709 0.11641571 0.12871902 0.10927621 0.05882455 0.12559932
  0.11405984 0.07417176 0.0880156  0.1092209 ]]
```

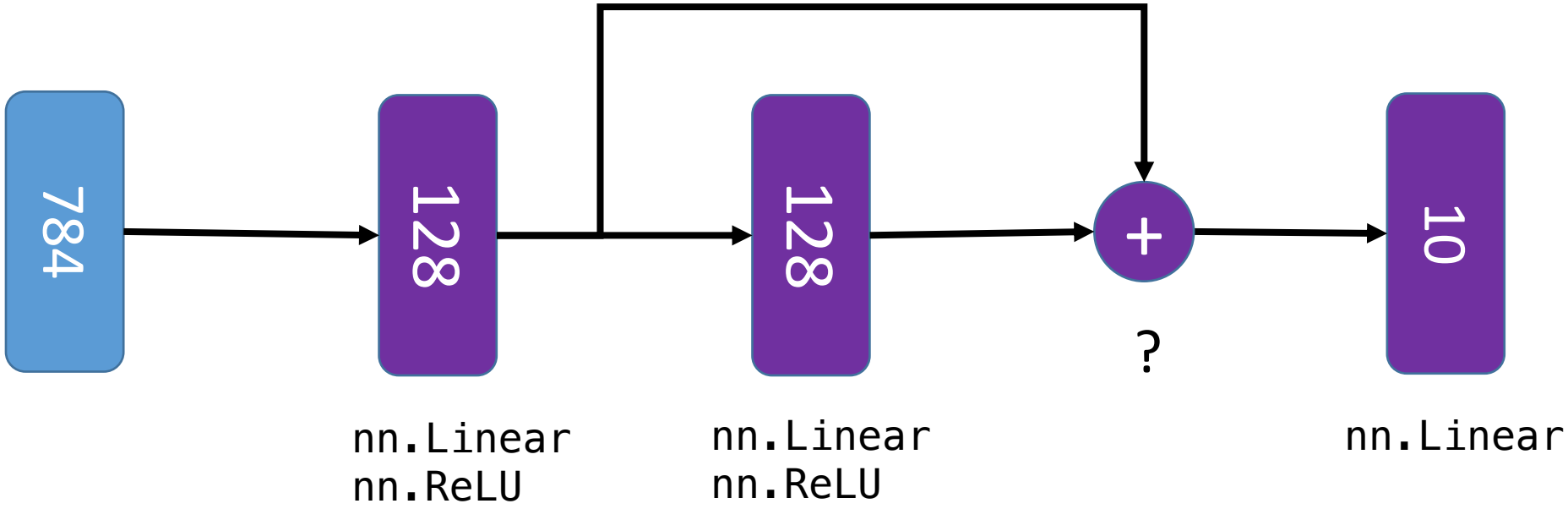
```
print(mnist_mlp._modules)
```

```
OrderedDict([('lin1', Linear(in_features=784, out_features=64,
bias=True)), ('relu', ReLU(inplace)), ('lin2', Linear(in_features=64,
out_features=10, bias=True))])
```

```
x = torch.randn((1,784),dtype = torch.float32)
y = mnist_mlp._modules['lin1'](x)
print(y.detach().numpy())
```

```
[[-0.01645807  0.12257674 -0.3842733  -0.10875969  0.16210961 -0.9262785
  0.3560028  -0.3479771  -0.4386911  -0.37093288 -0.0556463  -0.4703986
  0.54101855 -0.51167864  0.08044843  0.21852194  1.1178634   0.22886397
 -0.6046644  -0.36042506 -0.0121364  -0.37140787  0.41181365  0.9680239
 -0.44987577  0.11921623 -0.55106914  0.18517387  0.56142396 -0.59405994
  0.47401157  1.1164094   0.48247746  0.11364826  0.91904306  0.9782668
 -0.81544924  0.2937653  -1.3910329  -0.10774229 -0.03324178  0.54723066
 -0.43887684  0.12371415  0.03490007  0.62052786 -0.10143616 -0.43207315
  0.42250878  0.49805558 -0.69287837 -0.2873435  -0.5114934   0.6561675
  1.0234826   0.04920949  0.3638496   0.5078668   0.1270785  -0.18455932
 -1.1579272  -0.6274739  -0.23763435 -0.7309237 ]]
```

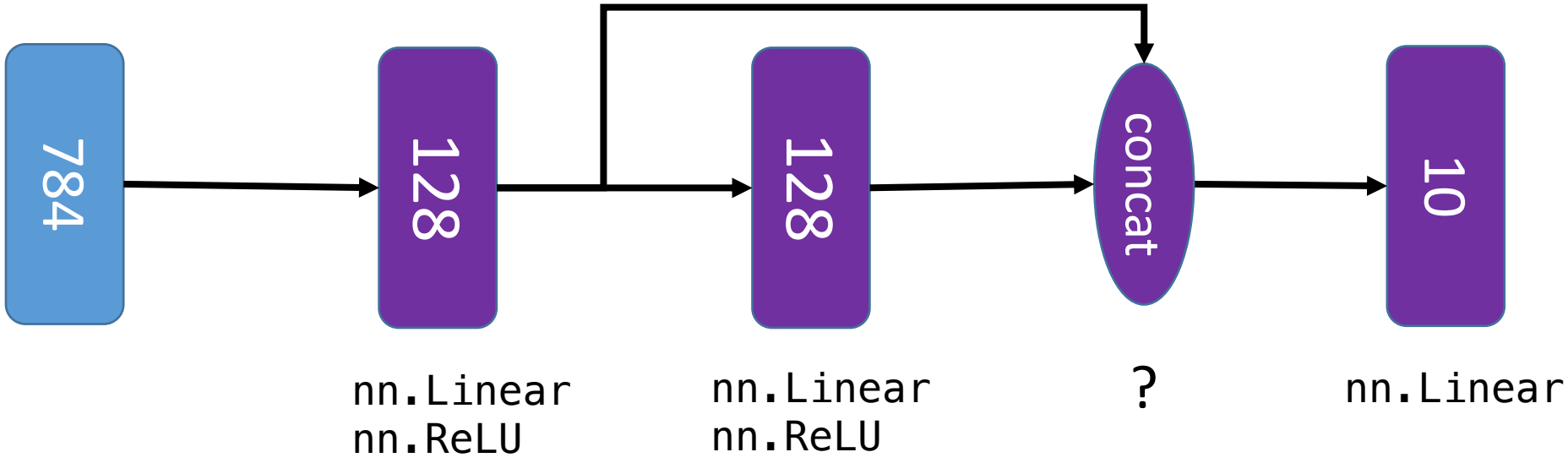
用Module实现复杂的模型



用Module实现复杂的模型

```
import torch.nn.Functional as F

class mlp(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(mlp,self).__init__()
        self.lin1 = nn.Linear(in_features=in_dim, out_features=128)
        self.lin2 = nn.Linear(in_features=128,out_features=128)
        self.lin3 = nn.Linear(in_features=128,out_features=out_dim)

    def forward(self, x):
        x1 = F.relu(self.lin1(x))
        x2 = F.relu(self.lin1(x1))
        x3 = x1+x2
        x4 = self.lin1(x3)
        return x4
```

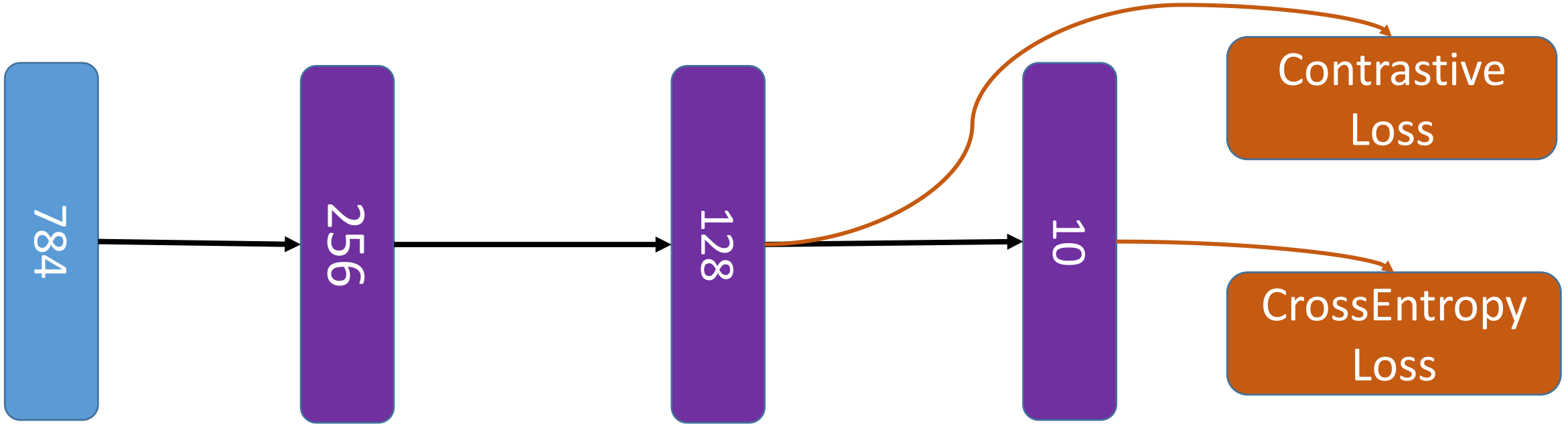
用Module实现复杂的模型



用Module实现复杂的模型

```
import torch.nn.Functional as F

class mlp(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(mlp,self).__init__()
        self.lin1 = nn.Linear(in_features=in_dim, out_features=128)
        self.lin2 = nn.Linear(in_features=128,out_features=128)
        self.lin3 = nn.Linear(in_features=256,out_features=out_dim)

    def forward(self, x):
        x1 = F.relu(self.lin1(x))
        x2 = F.relu(self.lin1(x1))
        x3 = torch.cat((x1,x2),dim = 1)
        x4 = self.lin1(x3)
        return x4
```

用Module实现复杂的模型



$$h_1 = relu(W_{h1}x + b_{h1}) \quad h_2 = relu(W_{h2}h_1 + b_{h2}) \quad z = Softmax(W_o h_2 + b_o)$$

$$\|h_2\| = 1$$

$$contrastive\ loss(x_1, x_2, y) = y \times d(x_1, x_2) + (1 - y) \times \max(\alpha - d(x_1, x_2), 0)$$

用Module实现复杂的模型

```
def contrastive_loss(x,y, margin = 0.3):
    pair_dist = torch.cdist(x,features)

    l1 = torch.unsqueeze(y,dim=0)
    l2 = torch.unsqueeze(y,dim=1)
    pair_mask = torch.logical_not(torch.eye(y.shape[0],dtype=torch.bool))

    pos_mask = torch.logical_and(l1==l2, pair_mask)
    neg_mask = torch.logical_and(l1!=l2, pair_mask)

    pos_dist = torch.masked_select(pair_dist,pos_mask)
    neg_dist = torch.masked_select(pair_dist,neg_mask)

    pos_loss = torch.mean(pos_dist)
    margin = torch.tensor(margin)
    neg_loss = torch.mean(torch.max(margin - neg_dist,torch.tensor(0.0)))
    return pos_loss + neg_loss
```

用Module实现复杂的模型

```
labels = torch.tensor([0,1,2,0,2])
l1 = torch.unsqueeze(labels,dim=0)
l2 = torch.unsqueeze(labels,dim=1)
```

```
l1:tensor([[0, 1, 2, 0, 2]])

l2:tensor([[0],
        [1],
        [2],
        [0],
        [2]])
```

```
pair_mask = torch.logical_not(torch.eye(labels.shape[0],dtype=torch.bool))
pos_mask = torch.logical_and(l1==l2,pair_mask)
neg_mask = torch.logical_and(l1!=l2,pair_mask)
```

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

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

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

```
sim = torch.randn((5,5))
```

```
tensor([[ 0.5344,  0.7295, -0.4729,  0.4417,  1.5117],
        [ 0.3096, -0.1728,  0.4240, -1.8273,  0.6577],
        [-0.0897, -0.8336, -0.2095,  0.4435,  1.1995],
        [-0.5547,  0.6684, -1.2100, -0.2603,  0.6245],
        [ 0.3822,  1.2045,  0.2586,  0.9755, -0.2574]])
```

```
pos_sim = torch.masked_select(sim,pos_mask)
neg_sim = torch.masked_select(sim,neg_mask)
```

```
tensor([ 0.4417,  1.1995, -0.5547,  0.2586])
tensor([ 0.7295, -0.4729,  1.5117,  0.3096,  0.4240, -1.8273,  0.6577, -0.0897,
        -0.8336,  0.4435,  0.6684, -1.2100,  0.6245,  0.3822,  1.2045,  0.9755])
```

用Module实现复杂的模型

```
class mlp(nn.Module):
    def __init__(self,in_features):
        super(mlp, self).__init__()

        self.fnet = nn.Sequential(OrderedDict([
            ('lin1',nn.Linear(in_features,256)),
            ('relu1',nn.ReLU()),
            ('lin2',nn.Linear(256,128)),
            ('relu2',nn.ReLU())]))

        self.fc = nn.Linear(128,10)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        f = F.normalize(self.fnet(x),dim=1)
        logits = self.fc(f)
        p = self.softmax(logits)

        return f,logits,p
```

用Module实现复杂的模型

```
def train(model, data_loader,lr=0.01):
    epochs = 10
    model.train()
    optimizer = optim.RMSprop(model.parameters(), lr = lr)

    identity_loss = torch.nn.CrossEntropyLoss()

    for epoch in range(epochs):
        for imgs, labels in data_loader:
            batch_size = imgs.shape[0]
            features,logits, prob = model(imgs.view(batch_size,-1))

            contr_loss = contrastive_loss(features,labels)
            ident_loss = identity_loss(logits,labels)
            loss = contr_loss + ident_loss

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

小结

使用pytorch开发神经网络模型的基本步骤：

- 1. 构造模型：
  - torch.nn.Sequential类
  - torch.nn.Module类
- 2.训练模型：
  - 损失函数： torch.nn模块
  - 优化算法： torch.optim模块
  - 数据加载： torch.utils.data模块

练习三

- 1. 糖尿病预测（回归）

```
from sklearn.datasets import load_diabetes
diabetes_dataset = load_diabetes()
data = diabetes_dataset['data']
targets = diabetes_dataset['target']
print(data.shape)
print(targets.shape)
```

```
(442, 10)
(442,)
```

- 2. MNIST手写体识别（分类）