torch官方文档

2024年4月24日 15:54

https://pytorch.org/docs/stable/_modules/torch/nn/modules/linear.html

标量 张量

2024年4月10日 9:51

https://blog.csdn.net/ztf312/article/details/72859014?ops_request_misc=%7B%22request%5Fid%22%3A%221712712713790168002130742228%22%2C%22scm%22%3A%2220140713.130102334..%22%7D&request_id=171271371379016800213074228&biz_id=0&spm=1018.2226.3001.4187

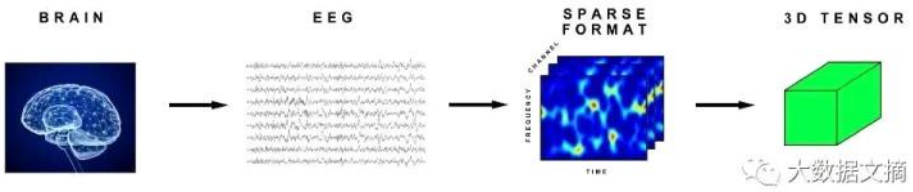
标量是一个数字，0维张量

1维张量 数组，向量，1个坐标轴
2维张量，矩阵，2个坐标轴
3维张量，如时间序列
4维张量，如图像，因为样本量占据张量的第4个字段。

- 例如，一个图像可以用三个字段表示：

  (width, height, color_depth) = 3D

  但是，在机器学习工作中，我们经常要处理不止一张图片或一篇文档——我们要处理一个集合。我们可能有10,000张郁金香的图片，这意味着，我们将用到4D张量，就像这样：

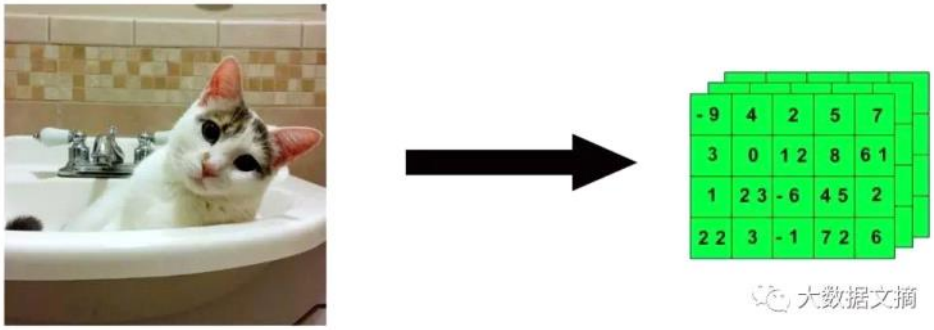


  (sample_size, width, height, color_depth) = 4D

- 再如彩色图片
  彩色图片有不同的颜色深度，这取决于它们的色彩（注：跟分辨率没有关系）编码。一张典型的JPG图片使用RGB编码，于是它的颜色深度为3，分别代表红、绿、蓝。

  这是一张我美丽无边的猫咪（Dove）的照片，750 x750像素，这意味着我们能用一个3D张量来表示它：

  (750,750,3)



  然后，如果我们有一大堆不同类型的猫咪图片（虽然都没有Dove美），也许是

100,000张吧，不是DOVE它的，750 x750像素的。我们可以在Keras中用4D张量来这样定义：

(10000,750,750,3)

5维张量，视频

5D张量可以用来存储视频数据。TensorFlow中，视频数据将如此编码：

（sample_size, frames, width, height, color_depth)

如果我们考察一段5分钟（300秒），1080pHD（1920 x 1080像素），每秒15帧（总共4500帧），颜色深度为3的视频，我们可以用4D张量来存储它：

(4500,1920,1080,3)

当我们有多段视频的时候，张量中的第五个维度将被使用。如果我们有10段这样的视频，我们将得到一个5D张量：

(10,4500,1920,1080,3)

如何判断数组形状

2024年4月29日 9:22

1. 确定维度数：首先，看你有多少层括号。每一层括号都代表一个维度。例如，[[ ]] 是一个二维数组，[[[ ]]] 是一个三维数组，以此类推。
2. 计算每个维度的大小：然后，对于每一层括号，数一下里面有多少个元素或子数组。这个数就是该维度的大小。

例1：a = [[1, 2, 3], [4, 5, 6]]

- 维度大小：第一层括号里有两个子数组，所以第一维大小是2；每个子数组里有三个元素，所以第二维大小是3。
- 形状：(2, 3)

例2：b = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]

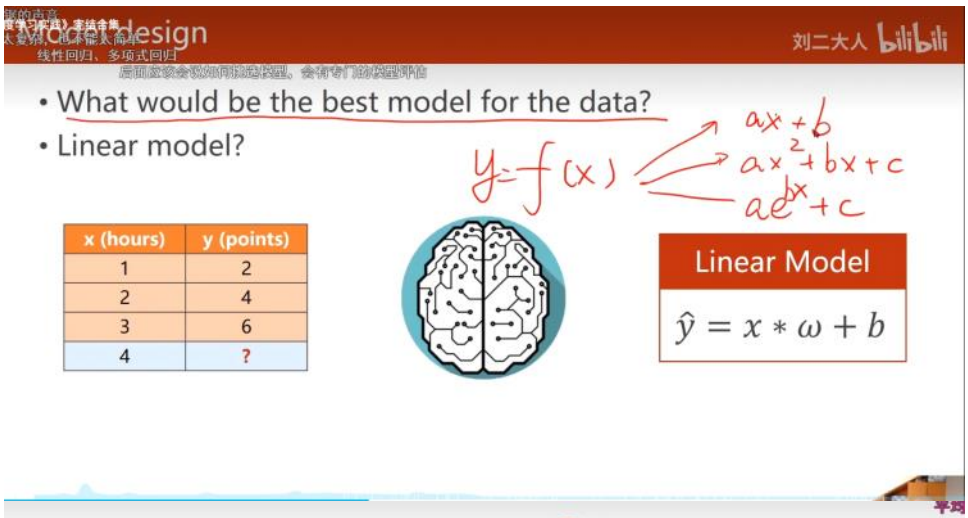
- 维度数：有三层括号，所以是三维数组。
- 维度大小：最外层括号里有两个子数组，所以第一维大小是2；每个子数组里又有两个子数组，所以第二维大小是2；每个最内层的子数组里有两个元素，所以第三维大小是2。
- 形状：(2, 2, 2)

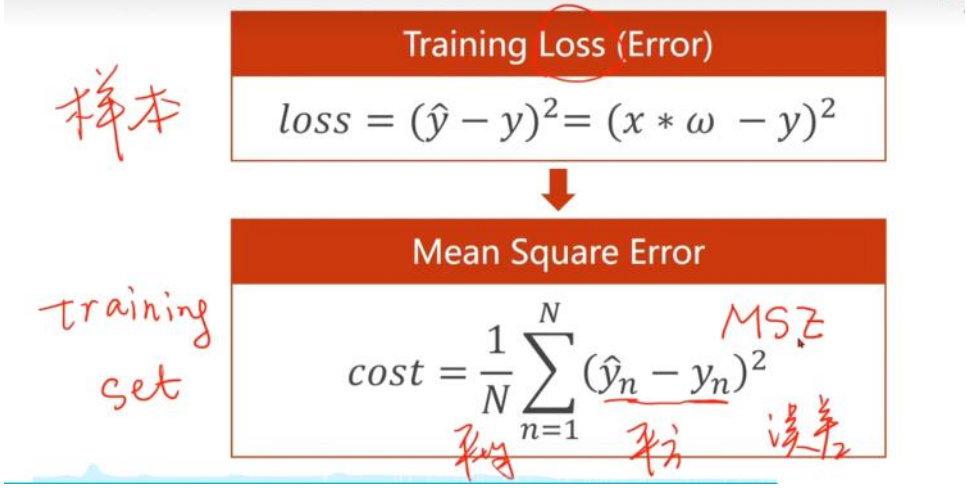
例3：c = torch.Tensor([[0.2, 0.1, -0.1]])

- 维度数：有两层括号，所以是二维张量。
- 维度大小：最外层括号里有一个子数组（尽管它看起来像一个单独的数组，但它仍然被一层括号包围），所以第一维大小是1；这个子数组里有三个元素，所以第二维大小是3。
- 形状：(1, 3)

线性模型

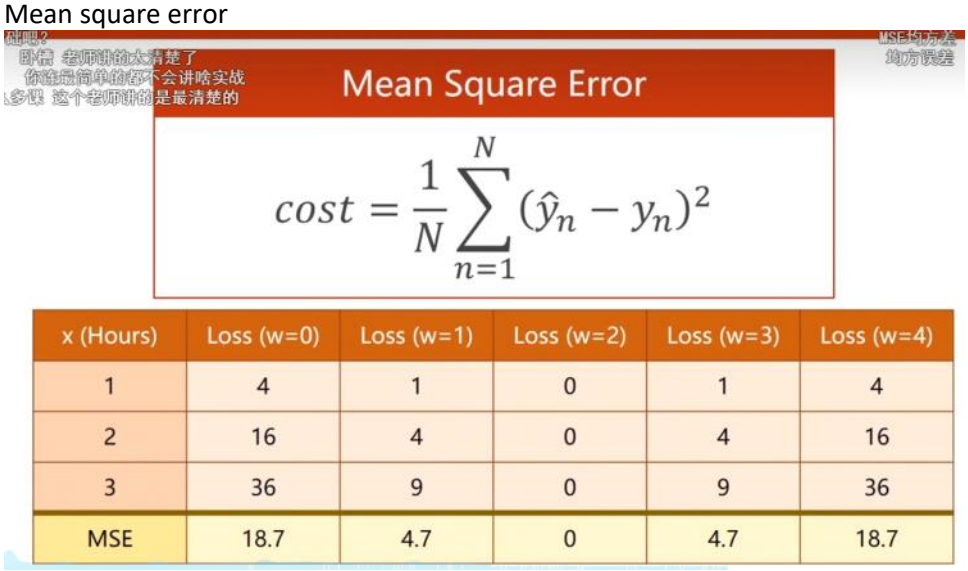
2024年4月6日 19:49





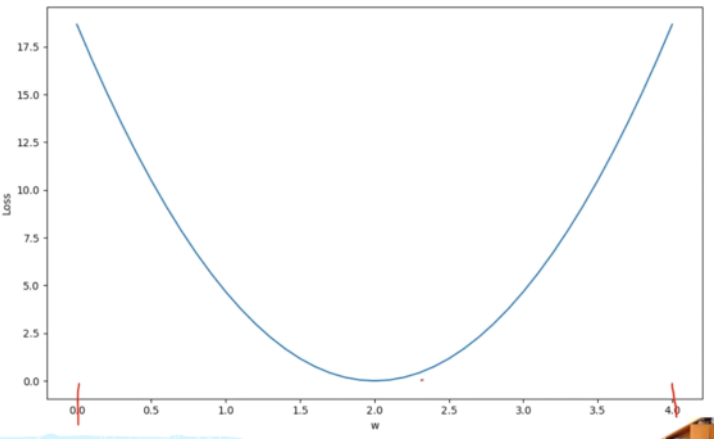
MSE

Mean square error



x (Hours)	Loss (w=0)	Loss (w=1)	Loss (w=2)	Loss (w=3)	Loss (w=4)
1	4	1	0	1	4
2	16	4	0	4	16
3	36	9	0	9	36
MSE	18.7	4.7	0	4.7	18.7

It can be found that when $\omega = 2$, the cost will be minimal.




```
import numpy as np
import matplotlib.pyplot as plt

x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

def forward(x):
    return x * w

def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) * (y_pred - y)

w_list = []
mse_list = []
for w in np.arange(0.0, 4.1, 0.1):
    print('w=', w)
    l_sum = 0
    for x_val, y_val in zip(x_data, y_data):
        y_pred_val = forward(x_val)
        loss_val = loss(x_val, y_val)
        l_sum += loss_val
        print('\t', x_val, y_val, y_pred_val, loss_val)
    print('MSE=', l_sum / 3)
    w_list.append(w)
    mse_list.append(l_sum / 3)
```

```
import numpy as np
import matplotlib.pyplot as plt
```

Import necessary library to draw the graph.

MSE

2024年4月26日 9:46

在均方误差（MSE）的计算中，除以2的原因主要是为了数学上的便利，特别是在求导和梯度下降的过程中。MSE的定义是预测值与真实值之间差的平方的平均值。如果不除以2，MSE的公式就是：

MSE=$\frac{1}{n}\sum_{i=1}^{n}\left(y_i-\hat{y}_i\right)^2$

其中，n 是样本数量，yi 是第 i 个样本的真实值，$\hat{y}_i$ 是模型对第 i 个样本的预测值。

当我们对MSE进行求导时，平方项会导致导数中出现2的系数。为了简化计算，通常在定义MSE时就除以2，这样求导后就不会再出现这个系数，使得梯度下降等优化算法的计算更为简洁。因此，MSE的另一种常见定义是：

MSE = $\frac{1}{2n}\sum_{i=1}^{n}\left(y_i-\hat{y}_i\right)^2$

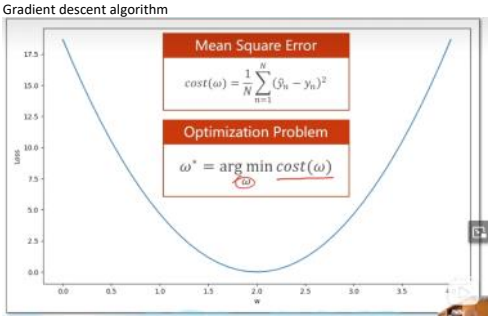
在这个定义下，对于单个样本的损失就是$\left(y_i-\hat{y}_i\right)^2$。所以，在前面的例子中，当预测价格为190万元而实际价格为200万元时，单个样本的MSE损失就是 $(200-190)^2/2=100/2=50$。

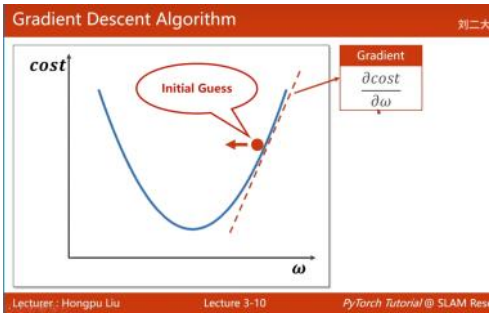
这样做并不会改变MSE作为损失函数的基本性质，只是为了让数学运算更加简洁。在机器学习和深度学习的实践中，这种除以2的操作是很常见的，特别是在定义损失函数时。

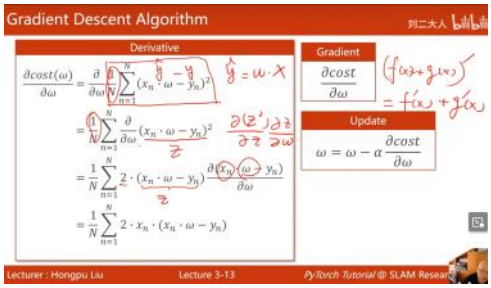
梯度下降算法

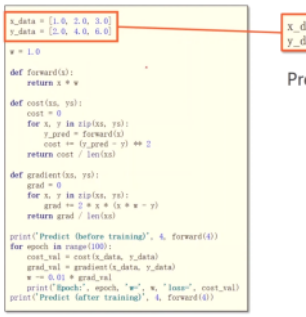
2024年4月7日 9:03

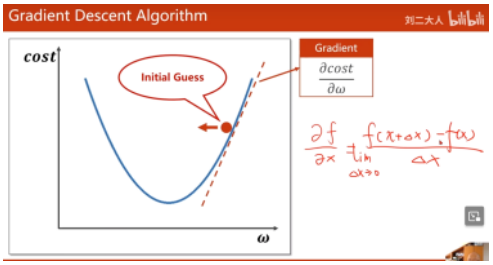
Gradient descent algorithm

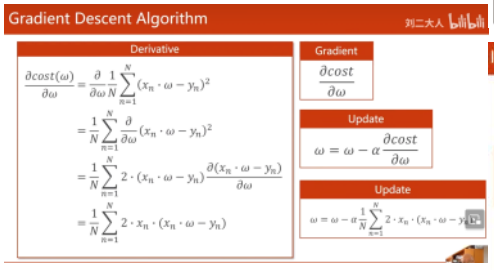


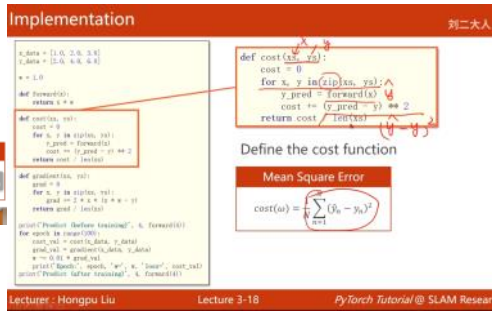






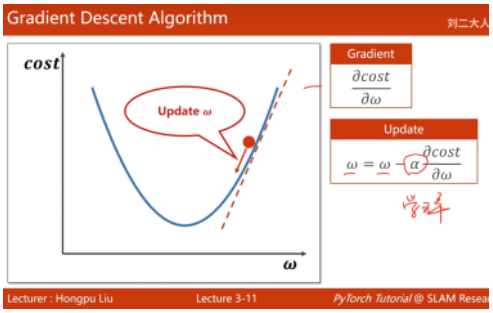






要找斜率往小的方向走

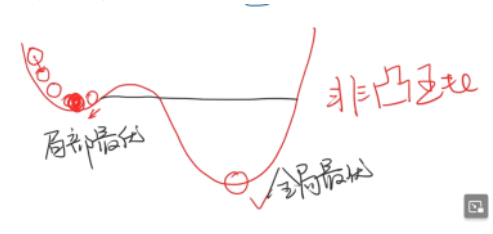
学习率要小一点，不然步子迈得太大



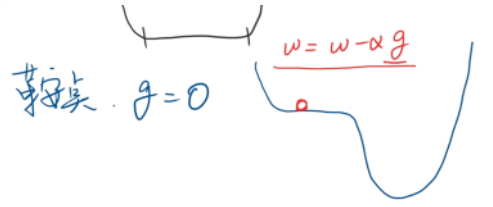
每次迭代的时候都是朝着函数cost下降的方向更新w权重

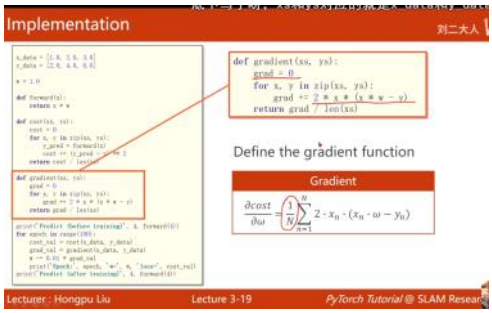
有可能出现这种情况

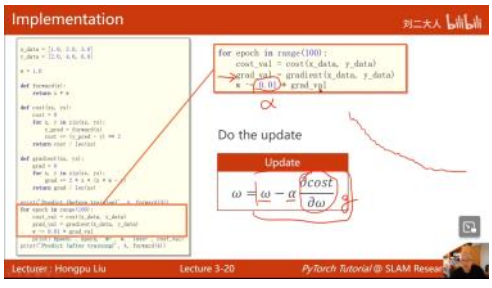
局部最优 全局最优

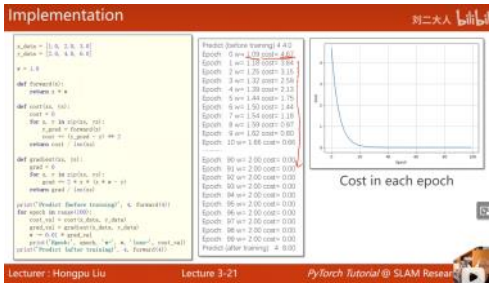


鞍点





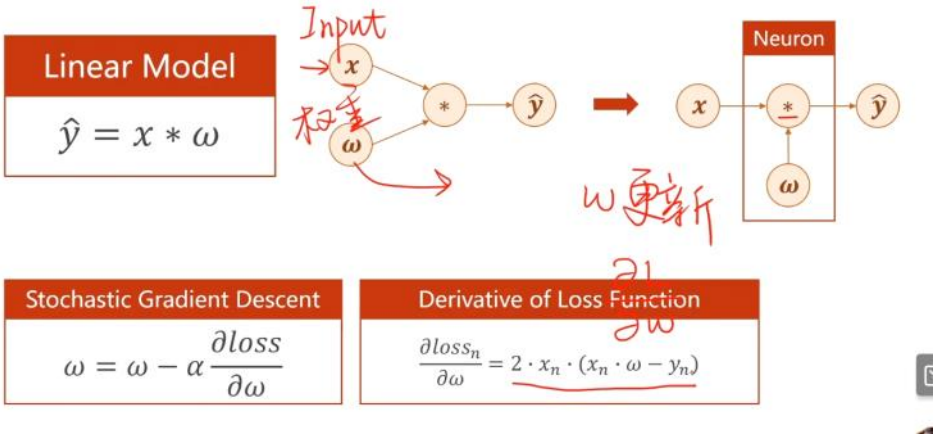




反向传播

2024年4月8日 9:18

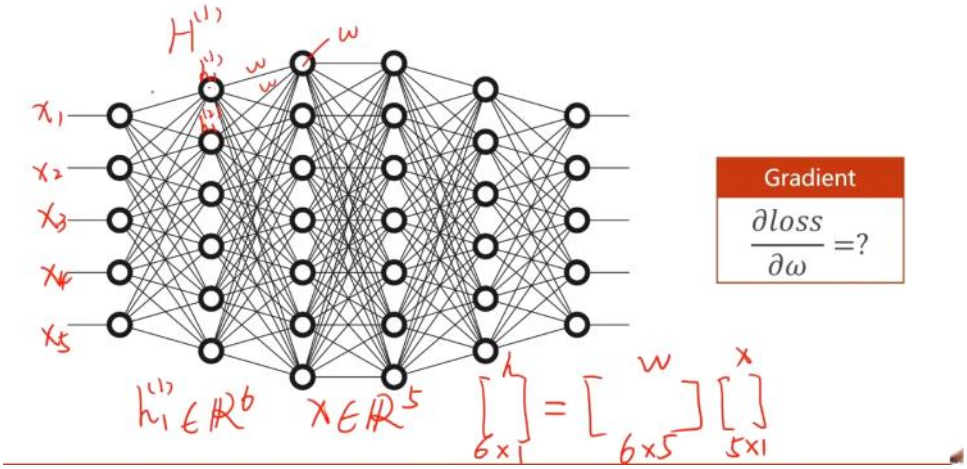
Back propagation



x:输入的变量 下图中x是五维向量

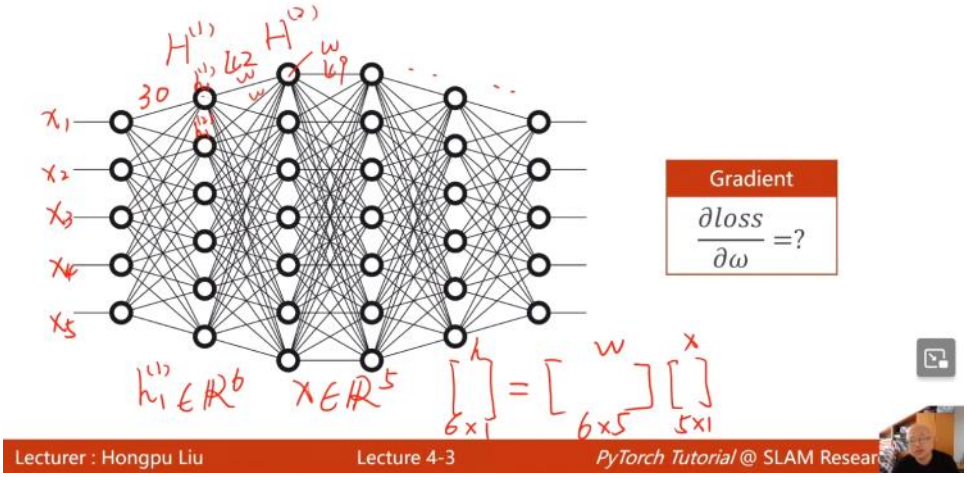
h:hidden隐层 H_1是六维向量

h矩阵=w矩阵*x矩阵

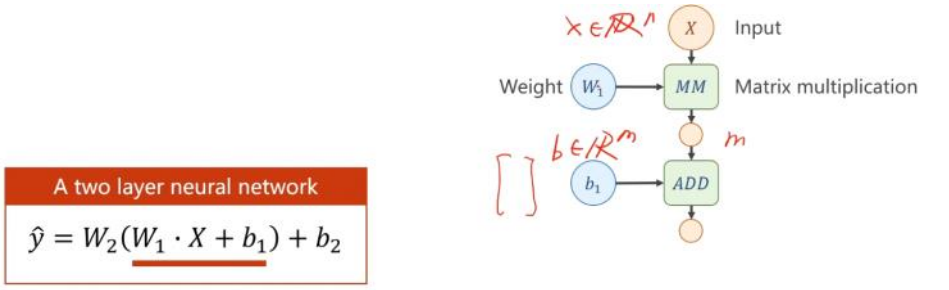


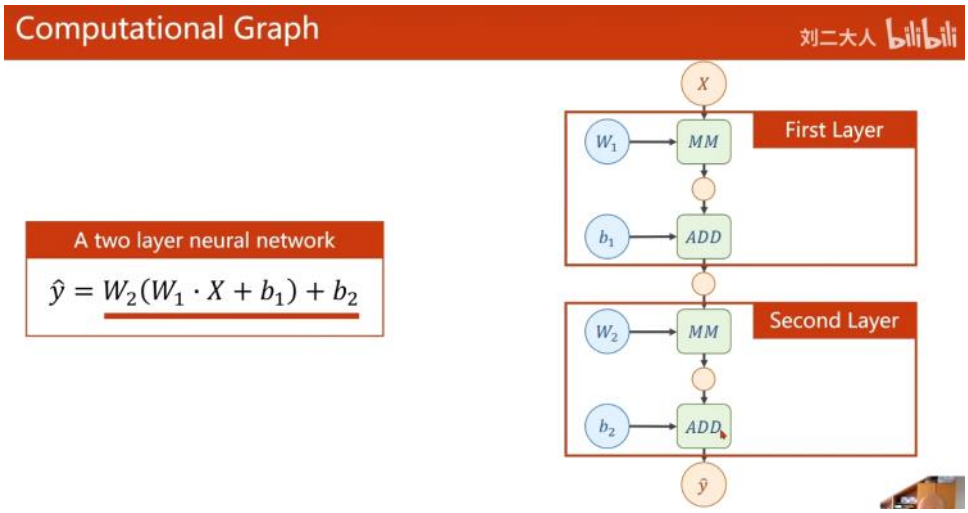
由此判断出w的矩阵维度

比如x1和h1之间是6行5列的w矩阵，30个权重，以此类推

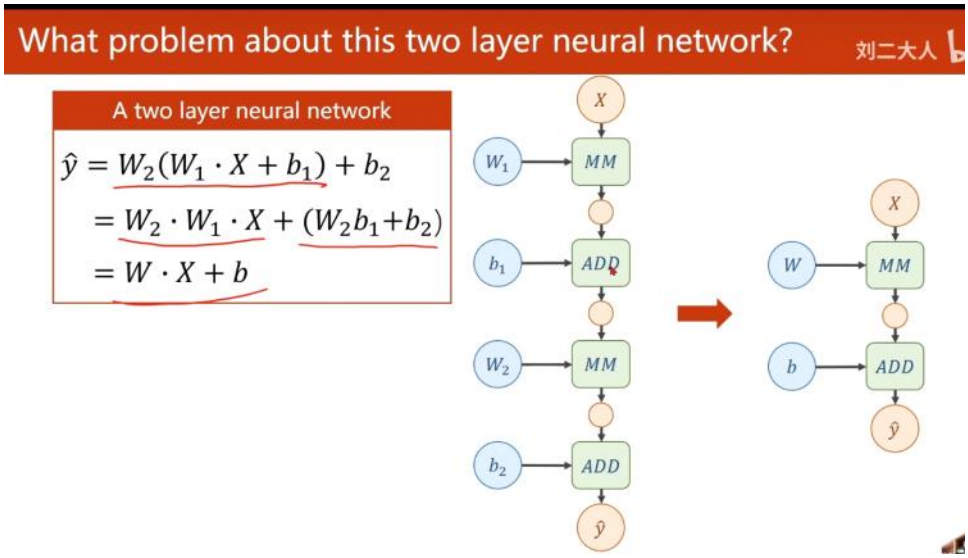


以简单的两层神经网络为例，绿模块代表计算过程





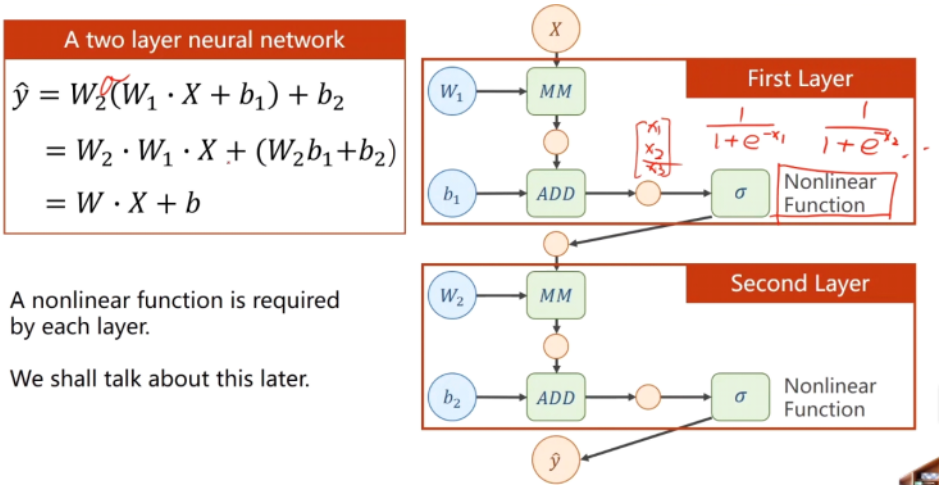
Matrix-cookbook矩阵求导

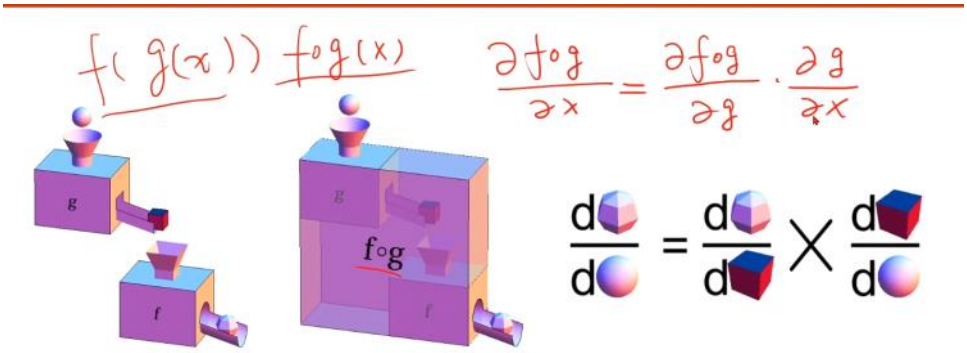


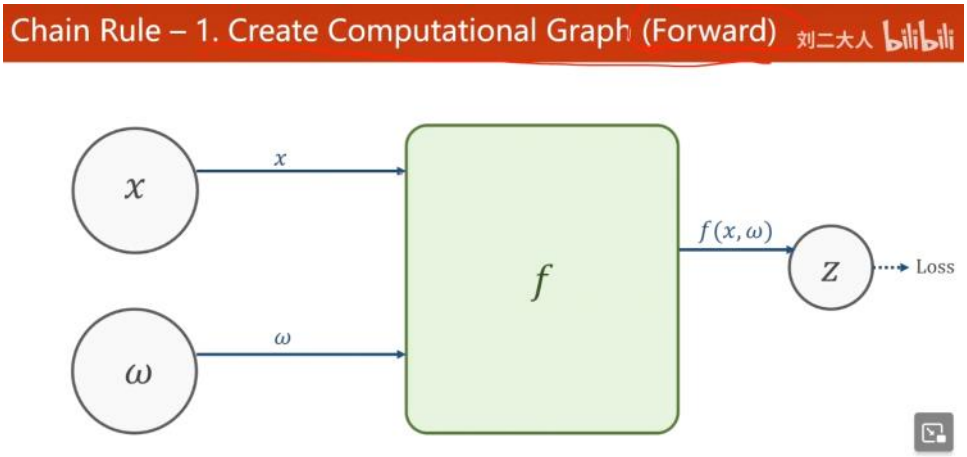
激活函数（比如sigmod）：引入非线性的函数，就不能线性展开了

正因为有了激活函数，使中间的得数有了变化，深度的增加也就有了意义

不加激活函数多少层都和一层没区别

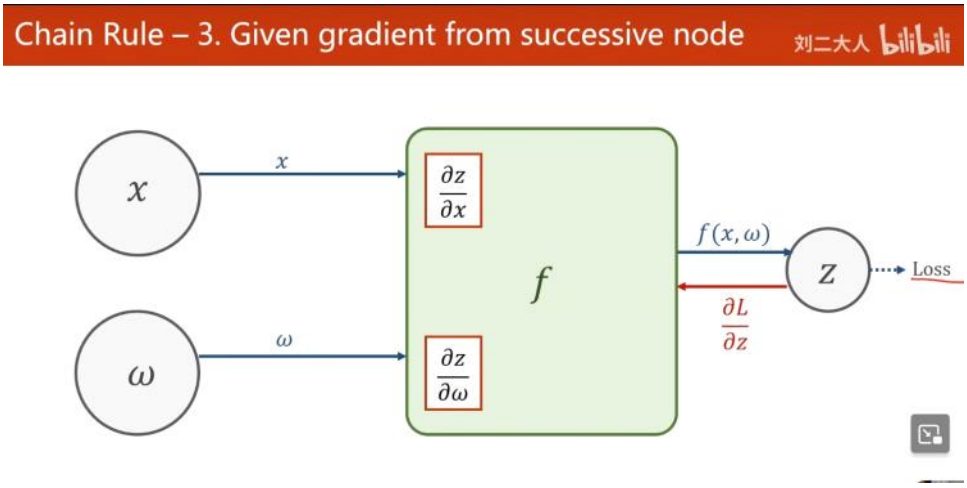


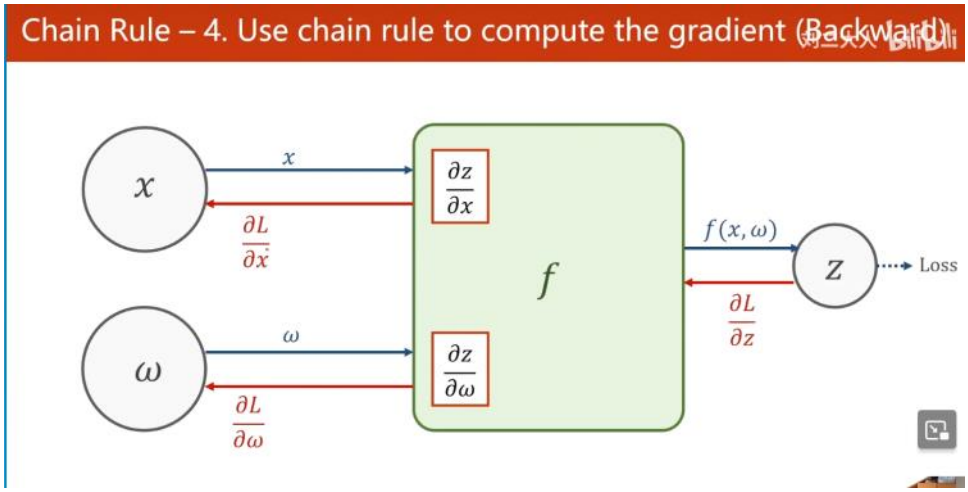


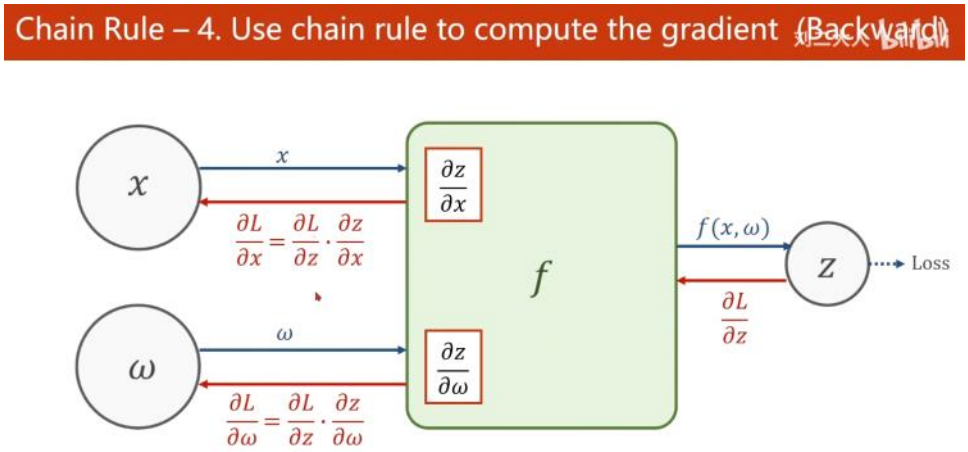


f-前馈运算

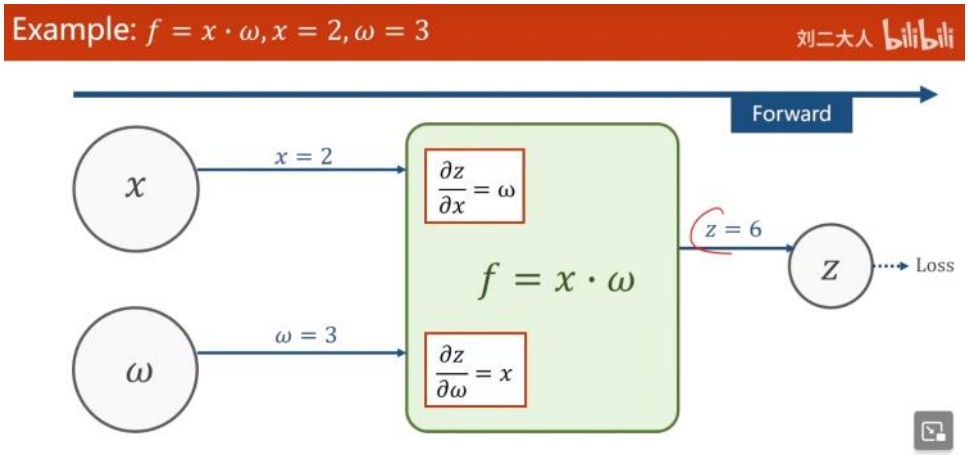
再算LOSS对z的偏导



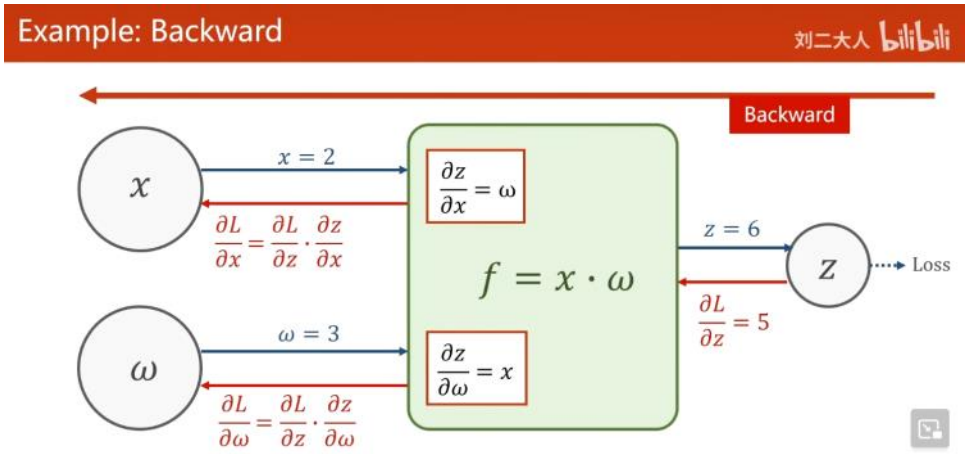

Chain Rule – 4. Use chain rule to compute the gradient (Backward)
x
ω
f
z
Loss
$f(x,\omega)$
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial \omega}$
$\frac{\partial L}{\partial x}$
$\frac{\partial L}{\partial \omega}$
$\frac{\partial L}{\partial z}$

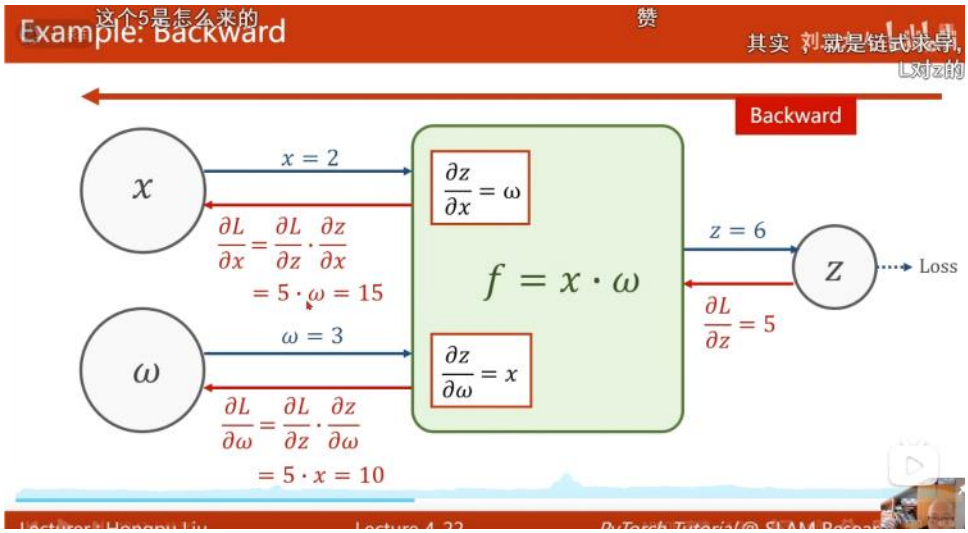

Chain Rule – 4. Use chain rule to compute the gradient (Backward)
x
ω
f
z
Loss
$f(x,\omega)$
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial \omega}$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x}$
$\frac{\partial L}{\partial \omega} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial \omega}$
$\frac{\partial L}{\partial z}$

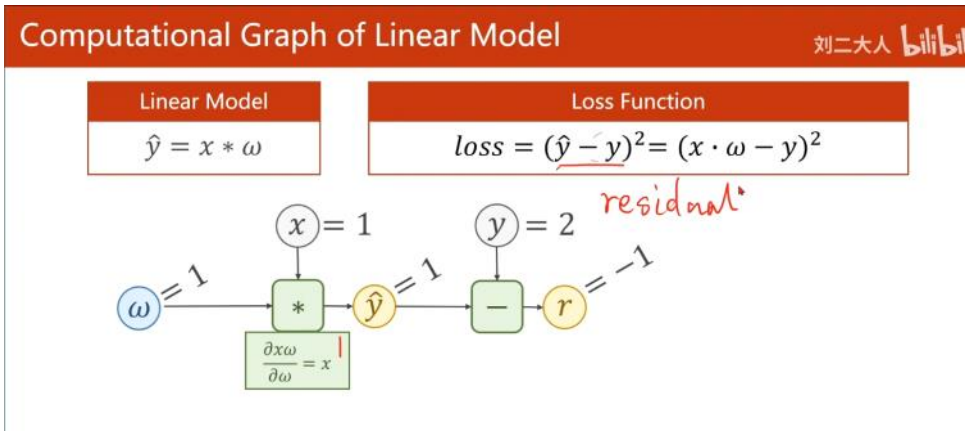
举例:


Example: $f = x \cdot \omega, x = 2, \omega = 3$
Forward
x
ω
f
z
Loss
$x = 2$
$\omega = 3$
$\frac{\partial z}{\partial x} = \omega$
$\frac{\partial z}{\partial \omega} = x$
$f = x \cdot \omega$
$z = 6$

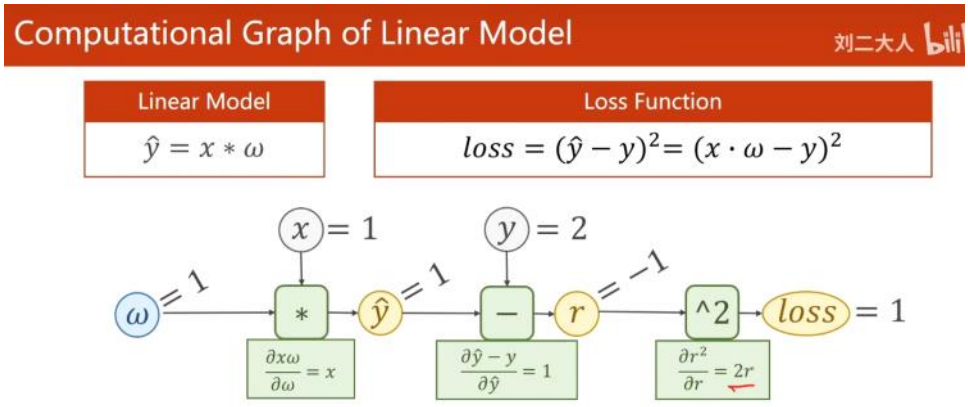
假设为5，反馈

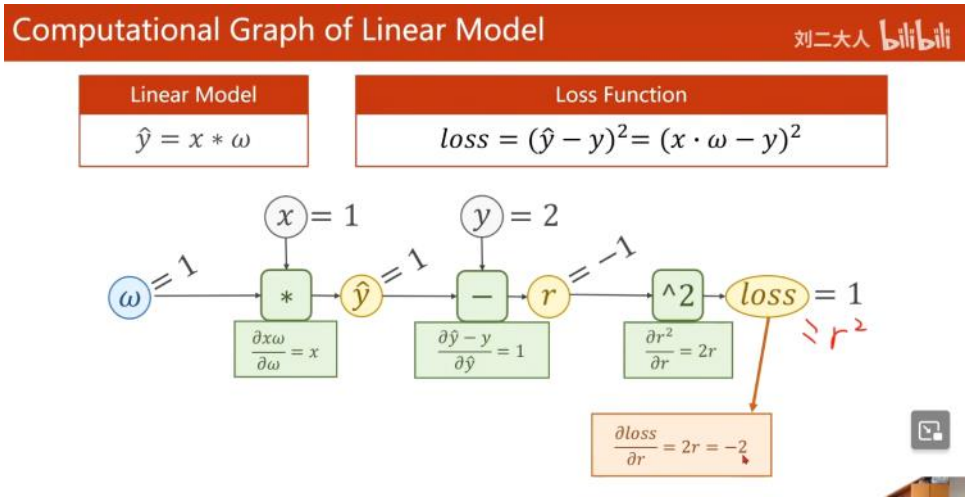


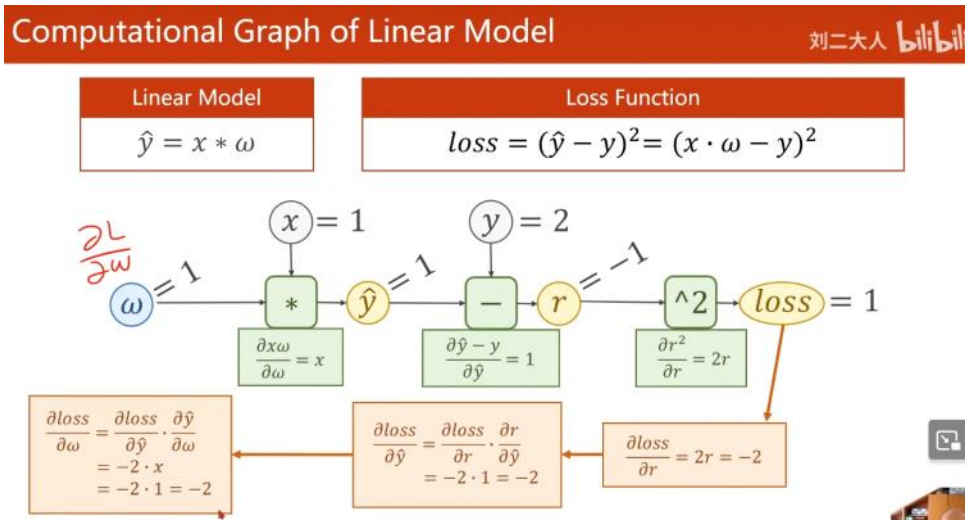




r-residual 残差项

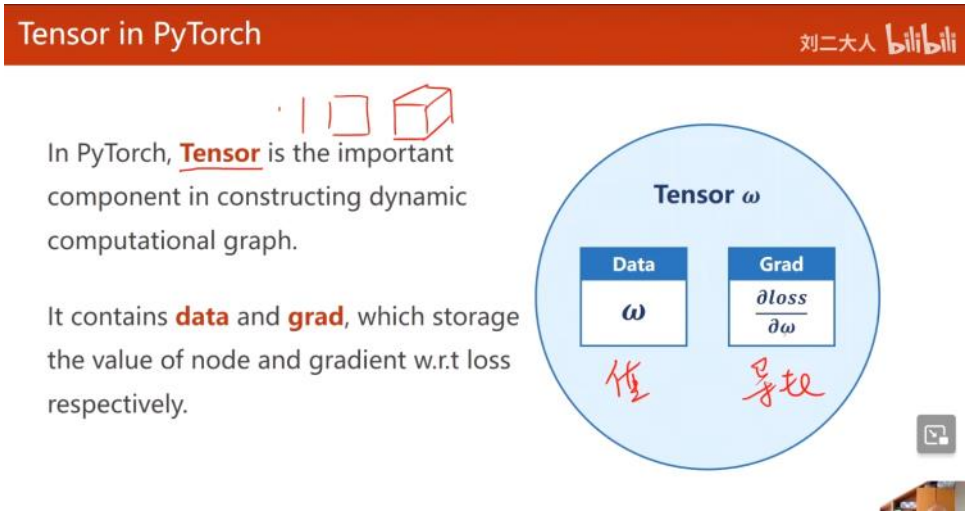






反馈回来，对梯度进行更新，我们更新的时候需要的就是梯度

tensor张量



Implementation of linear model with PyTorch

刘二大人 bilibili

```
import torch

x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = torch.Tensor([1.0])
w.requires_grad = True
```

If **autograd mechanics** are required, the element variable **requires_grad** of **Tensor** has to be set to **True**.

$\hat{y} = x \cdot w$

x*w是两个tensor之间的数乘

Implementation of linear model with PyTorch

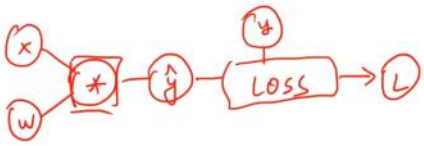
刘二大人 bili

```
def forward(x):
    return x * w

def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) ** 2
```

Define the loss function:

Loss Function

$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$



前馈

```
print("predict (before training)", 4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data

        w.grad.data.zero_()

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).item())
```

Forward, compute the loss.

反馈

```
print("predict (before training)", 4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data

        w.grad.data.zero_()

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).item())
```

Backward, compute grad for Tensor whose **requires_grad** set to True

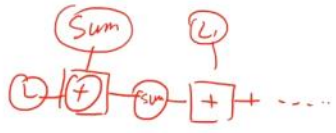
不能用sum的原因，sum是张量，反复构建计算图，吃内存

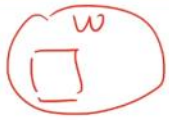
Implementation of linear model with PyTorch

刘二大人 bilibili

```
print("predict (before training)", 4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data
        w.grad.data.zero_()

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).item())
```

The **grad** is utilized to update weight.





Lecturer : Hongpu Liu Lecture 4-41 *PyTorch Tutorial* @ SLAM Resear

```
print("predict (before training)", 4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data

        w.grad.data.zero_()

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).i
```

NOTICE:

The grad computed by *.backward()* will be **accumulated**.

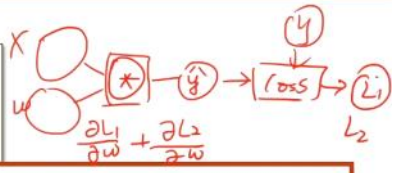
So after update, remember set the grad to **ZERO**!!!

必须梯度清零，不然计算的是和

Implementation of linear model with PyTorch

刘二大人 bilibili

```
print("predict (before training)", 4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data

        w.grad.data.zero_()

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).i
```

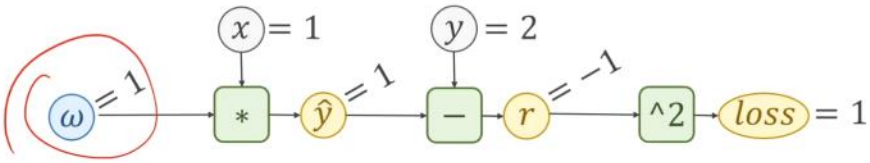


NOTICE:

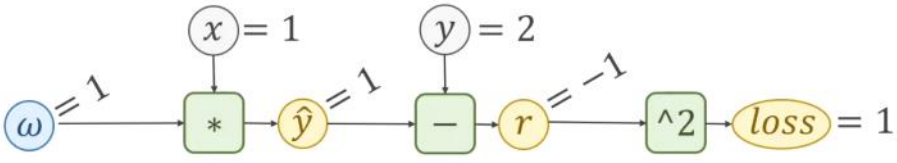
The grad computed by *.backward()* will be **accumulated**.

So after update, remember set the grad to **ZERO**!!!

Lecturer : Hongpu Liu Lecture 4-42 *PyTorch Tutorial* @ SLAM Resear




```
w = torch.Tensor([1.0])
w.requires_grad = True

l = loss(x, y)
```




```
l.backward()
```

$$\frac{\partial loss}{\partial \omega} = w.grad$$

计算图

2024年4月10日 10:23

一、计算图

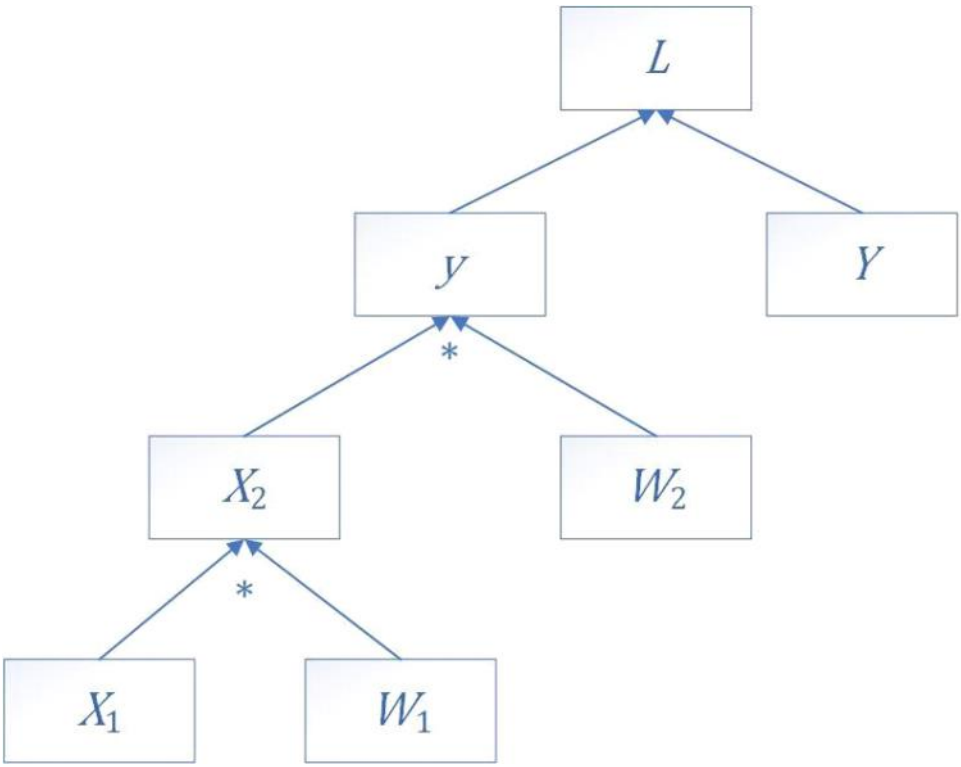
我们假设一个计算过程，其中$\mathbf{X_1}$、$\mathbf{W_1}$、$\mathbf{W_2}$、$\mathbf{Y}$都是N维向量。

$$\mathbf{X_2} = \mathbf{W_1}\mathbf{X_1}$$

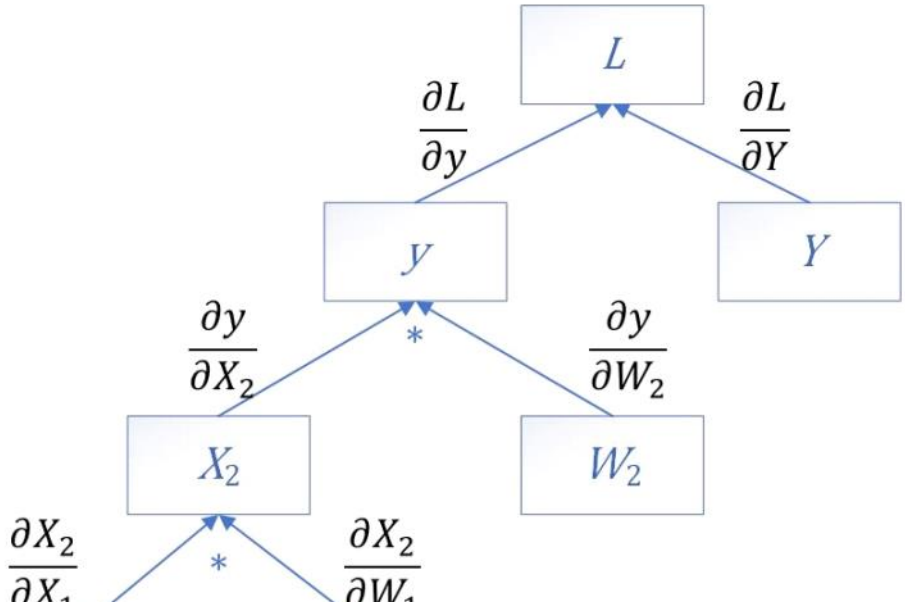
$$\mathbf{y} = \mathbf{W_2}\mathbf{X_2}$$

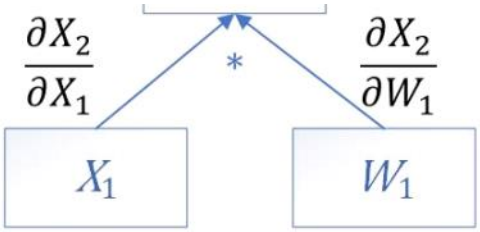
$$\mathrm{L} = \sum_{\mathrm{i}=1}^{\mathrm{N}} (\mathrm{Y_i} - \mathrm{y_i})^2$$

上述过程，用计算图表现出来，就是下图。



1. X1、W1、W2、Y是直接创建的，是叶子节点，X2、y、L是经过计算得到的，不是叶子节点。
2. 计算导数





如果我们想跨越若干层计算导数，如计算 $\frac{\partial \mathrm{L}}{\partial \mathbf{X_1}}$ 的值，则需要根据求导的链式法则，一层一层的计算下去。

$$\frac{\partial \mathrm{L}}{\partial \mathbf{X_1}} = \frac{\partial \mathrm{L}}{\partial \mathrm{y}} \frac{\partial \mathrm{y}}{\partial \mathbf{X_2}} \frac{\partial \mathbf{X_2}}{\partial \mathbf{X_1}}$$
$$\frac{\partial \mathrm{L}}{\partial \mathbf{W_1}} = \frac{\partial \mathrm{L}}{\partial \mathrm{y}} \frac{\partial \mathrm{y}}{\partial \mathbf{X_2}} \frac{\partial \mathbf{X_2}}{\partial \mathbf{W_1}}$$
$$\frac{\partial \mathrm{L}}{\partial \mathbf{W_2}} = \frac{\partial \mathrm{L}}{\partial \mathrm{y}} \frac{\partial \mathrm{y}}{\partial \mathbf{W_2}}$$

backward函数

2024年4月10日 11:12

PyTorch提供了autograd包来自动根据输入和前向传播构建计算图，其中，backward函数可以很轻松的计算出梯度。

Tensor在pytorch中用来表示张量，上例中的 X1、 W1、 W2都是张量，且均为直接被我们创建的。如果我们想使用autograd包让它们参与梯度计算，则需要在创建它们的时候，将.requires_grad属性指定为true。

注意，在pytorch中，只有浮点类型的数才有梯度，因此在定义张量时一定要将类型指定为float型。

```
x1 = torch.tensor([2, 3, 4, 5], dtype=torch.float, requires_grad=True)
print(x1)
```

```
tensor([2., 3., 4., 5.], requires_grad=True)
```

当然对于没有指定.requires_grad属性的向量，也可以在后续进行指定，或者使用requires_grad_函数进行指定。

```
w1 = torch.ones(4)
w1.requires_grad = True
print(w1)

w2 = torch.Tensor([1, 2, 3, 4])
w2.requires_grad_(True)
print(w2)
```

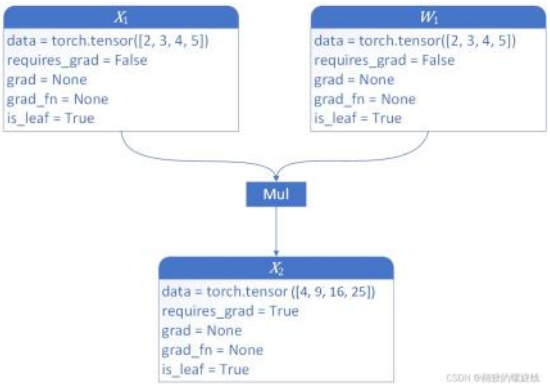
```
tensor([1., 1., 1., 1.], requires_grad=True)
tensor([1., 2., 3., 4.], requires_grad=True)
```

接下来进行运算操作。

```
x2 = x1 * w1
```

```
tensor([2., 3., 4., 5.], grad_fn=<MulBackward0>)
```

在这个运算中，pytorch会构建一个动态地创建一个计算图，即动态计算图（Dynamic Computation Graph, DCG），计算图中的每一个节点，都会封装若干个属性。下图为 X2=X1W1这一计算的计算图。



然后继续计算直到得到 L。这个过程是正向传播的过程，会继续动态的生成计算图。

```
y = x2 * w2
Y = torch.ones(2, 2, requires_grad=True)
L = (Y - y).mean()
```

在本例中，如果我们想求$\frac{\partial L}{\partial X_1}$，则需要首先使用backward()函数对L做反向传播。backward()实际上是通过DCG图从根张量追溯到每一个叶子节点，然后计算将计算出的梯度存入每个叶子节点的.grad属性中。由于L是一个标量，因此backward()函数中不需要传入任何参数。代码如下

```
L.backward()
```

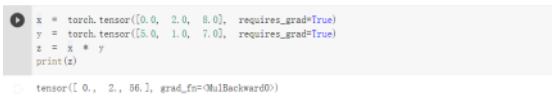
这一步后L针对每一个变量的梯度都会被求出，并存放在对应节点的.grad属性中。如果我们想要$\frac{\partial L}{\partial X_1}$，只需要读取x1.grad即可。

```
print(x1.grad)
```

上面的例子中，作为输出的L是一个标量，即神经网络只有一个输出，backward不需要传入参数。

但如果输出是一个向量，计算梯度需要传入参数。例如

```
x = torch.tensor([0.0, 2.0, 8.0], requires_grad=True)
y = torch.tensor([5.0, 1.0, 7.0], requires_grad=True)
z = x * y
print(z)
```



如果想求z对x或y的梯度，则需要将一个外部梯度传递给z.backward()函数。这个额外被传入的张量就是grad_tensor。

```
z.backward(torch.FloatTensor([1.0, 1.0, 1.0]))
```

另外，.grad属性在反向传播过程中是累加的，每一次反向传播梯度都会累加之前的梯度。因此每次重新计算梯度前都要将梯度清零。

```
x.grad.data.zero_()
```

data：存储的Tensor的值。

- requires_grad：该节点是否参与反向传播图的计算，如果为True，则参与计算；如果为False则不参与。
- grad：存储梯度值。requires_grad为False时，该属性为None；requires_grad为True且在调用过其他节点的backward后，grad保存对这个节点的梯度值，否则为None。
- grad_fn：表示用于计算梯度的函数，即创建该Tensor的Function。如果该Tensor不是通过计算得到的，则grad_fn为None；如果是通过计算得到的，则返回该运算相关的对象。

  举个例子，a为直接创建的Tensor，b和c由计算得到，则a、b、c的grad_fn如下所示。

```
a = torch.ones(2, 2, requires_grad=True)
print(a.grad_fn)
b = a + 2
print(b.grad_fn)
c = b * b * 3
print(c.grad_fn)
```

```
None
<AddBackward0 object at 0x7fde5fbe76d0>
<MulBackward0 object at 0x7fde5fbe7790>
```

- is_leaf：用True和False表示是否为叶子节点。

Torch.tensor/Tensor

2024年4月10日 9:39

PyTorch 是一个流行的开源机器学习库，用于深度学习应用。
在 PyTorch 中，tensor 是其核心数据结构，用于存储和操作多维数组（类似于 NumPy 的 ndarray）。
Torch.tensor是默认的tensor类型（torch.FloatTensor）的简称
https://blog.csdn.net/qq_42676511/article/details/121414784?ops_request_misc=&request_id=&biz_id=102&spm=1018.2226.3001.4187

- **torch.Tensor() 和 torch.tensor() 的区别：**
  torch.Tensor是一种主要的 tensor 类型，是torch.FloatTensor()的别名。所有的 tensor 都是torch.Tensor的实例。
  而torch.tensor()是一个函数，函数原型是：

  torch.tensor(data, dtype=None, device=None, requires_grad=False)
  其中 data 可以是：list、 tuple、 NumPy ndarray、 scalar和其他类型。
  torch.tensor会从data 中的数据部分做拷贝（而不是直接引用），根据原始数据类型生成相应的torch.LongTensor、 torch.FloatTensor和torch.DoubleTensor或者根据dtype 的值生成相应数据类型。

- **torch.Tensor(data):将输入的data转化为torch.FloatTensor()**
- Torch.tensor(data):将data转化为torch.FloatTensor、 torch.LongTensor、 torch.DoubleTensor等类型，转化类型依据于data的类型或者dtyoe的值
- Torch.Tensor()可以创建一个空的FloatTensor，使用torch.tensor()时会报错

1. 创建 Tensor
   - torch.tensor(data, dtype=None, device=None, requires_grad=False): 从数据创建 tensor。
   - torch.zeros(*size, dtype=None, device=None, requires_grad=False): 创建全为 0 的 tensor。
   - torch.ones(*size, dtype=None, device=None, requires_grad=False): 创建全为 1 的 tensor。
   - torch.eye(*size, dtype=None, device=None, requires_grad=False): 创建单位矩阵。
   - torch.rand(*size, dtype=None, device=None, requires_grad=False): 创建元素在 [0, 1) 区间内均匀分布的 tensor。
   - torch.randn(*size, dtype=None, device=None, requires_grad=False): 创建元素符合标准正态分布的 tensor。
2. Tensor 属性
   - .shape: 返回 tensor 的形状（维度）。
   - .size(): 返回 tensor 的大小（各维度上的元素数量）。
   - .numel(): 返回 tensor 中的元素总数。
   - .dtype: 返回 tensor 的数据类型。
   - .device: 返回 tensor 所在的设备（CPU 或 GPU）。
3. Tensor 操作
   - 数学运算：如 +, -, *, /, ** 等，都支持 element-wise 操作。

- 索引和切片：类似于 NumPy 的 ndarray，可以使用索引和切片来访问和修改 tensor 的元素。
- 重塑：view(), reshape(), resize_() 等函数用于改变 tensor 的形状。
- 转换：to(dtype=None, device=None, non_blocking=False, copy=True) 用于改变 tensor 的数据类型或将其移动到另一个设备。
- 扩展：expand(), expand_as(), repeat() 等函数用于扩展 tensor 的维度或重复其元素。
- 聚合操作：如 sum(), mean(), max(), min() 等，用于计算 tensor 的统计信息。

4. 自动微分

- requires_grad=True: 在创建 tensor 时设置此参数，以便 PyTorch 跟踪该 tensor 上的所有操作，从而计算梯度。
- .backward(): 对一个 scalar tensor（或具有单个元素的 tensor）调用此函数，以计算其关于之前所有设置 requires_grad=True 的 tensor的梯度。
- .grad: 存储计算得到的梯度。

在 PyTorch 中，backward 函数是自动微分引擎的关键部分，它用于计算张量的梯度。当你在模型中进行前向传播时，PyTorch 会跟踪所有的操作，这样你就可以随后调用 backward 来自动计算梯度。

示例：

```
import torch
# 创建一个需要计算梯度的张量
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
# 定义一个简单的函数 y = x * 2
y = x * 2

# 调用 backward 计算梯度
y.backward()

# 输出梯度 dy/dx，它应该是 [2., 2., 2.] 因为 y = 2x
print(x.grad)
```

注意：

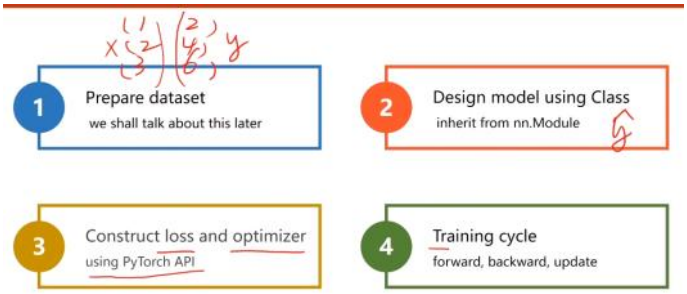
1.只有设置了 requires_grad=True 的张量才能计算梯度。如果你尝试对一个没有设置这个属性的张量调用 backward，你会得到一个错误。

2.默认情况下，梯度是累积的。如果你不希望这样，需要在每次调用 backward 之后使用 zero_() 方法清零梯度。

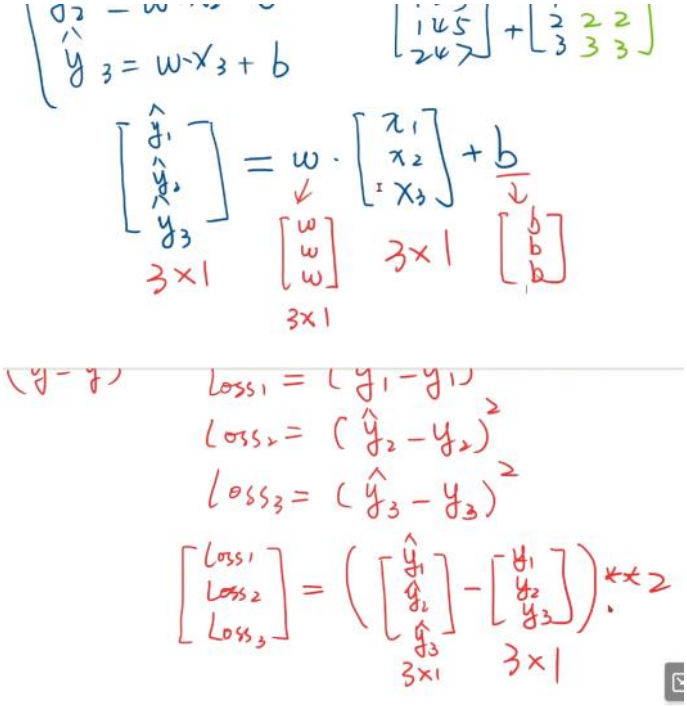
用pytorch实现线性回归

2024年4月9日　9:33

步骤

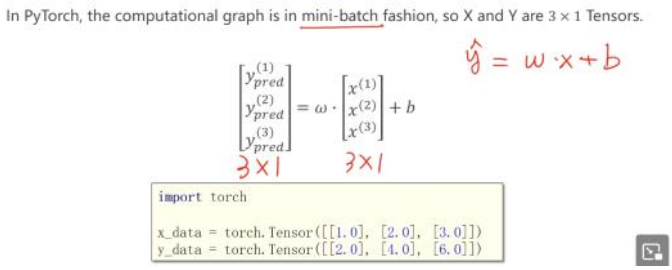


原理



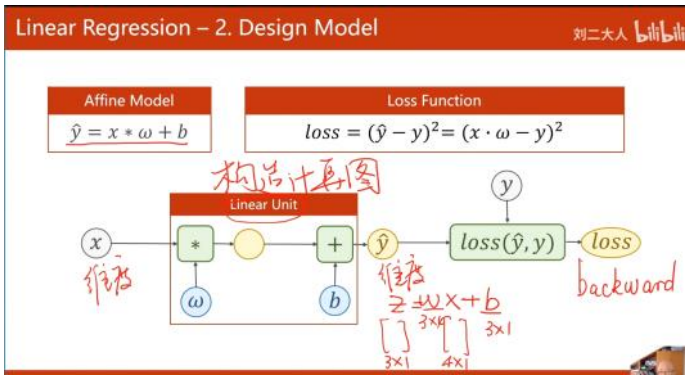
代码实现

1.数据准备

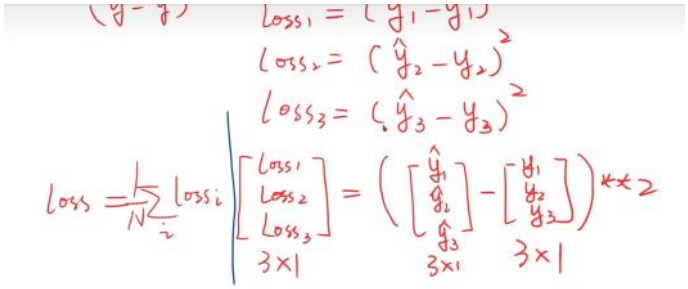


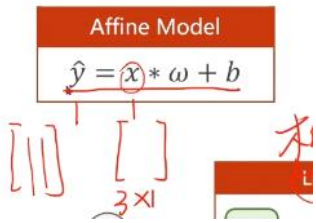
2.构造计算图 构造linear unit线性单元

为了求得W权重和b偏置的维度，从x和y hat的维度判断得出



本来的loss是一个矩阵，但是为了得到标量，（标量才能做backward）就得求和/求均值

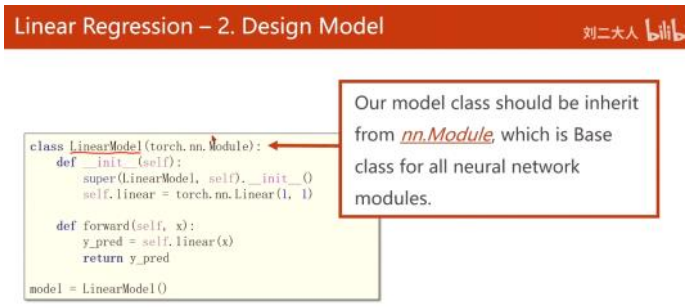




本例都是1维

计算图构造出的Loss是3*1的矩阵，
但是最后得到的loss必须是一个标量，所以必须求和（也可以再求均值），
这样才能用backward(其实向量也可以用，见上一节笔记)

向量是一维张量，矩阵是二维张量

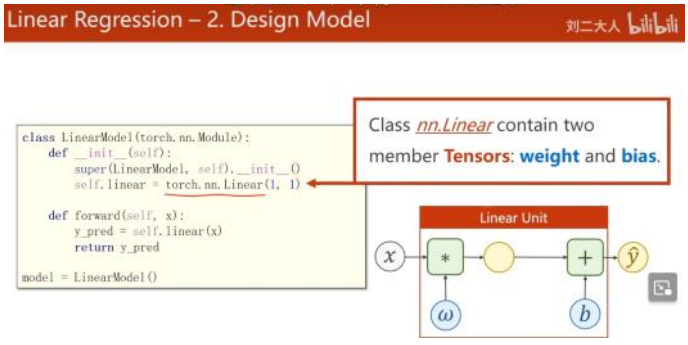


这里的LinearModel和self之间的逗号（,）是super()函数旧式调用的语法的一部分。在Python 2和早期版本的Python 3中，super()的调用通常是这样的：
super(CurrentClassName, self).__init__()
从Python 3开始，super()的调用可以简化为：
super().__init__()调用父类的构造

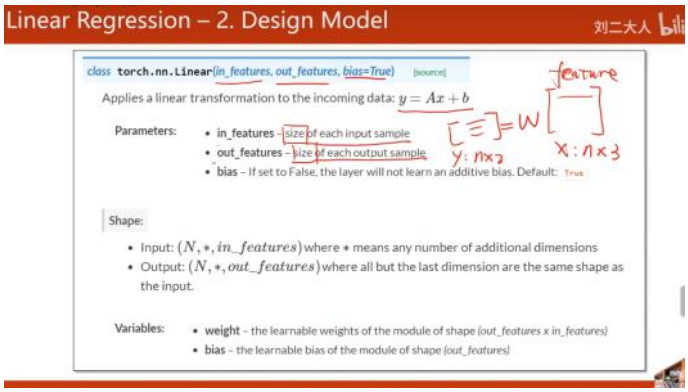
self.linear构造了一个对象 torch.nn.Linear(1, 1)
torch.nn.Linear(1, 1) 创建了一个线性层，该层有一个输入特征和一个输出特征。
这意味着它执行以下线性变换：
y = x * weight + bias
其中：

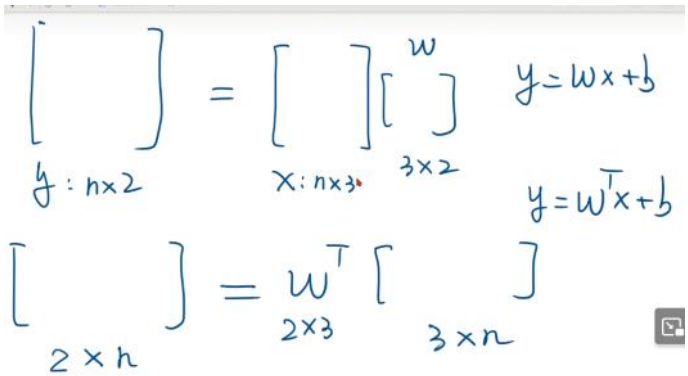
- x 是输入数据（一个标量，因为输入特征数为 1）。
- weight 是该层的权重参数（一个标量，因为输入和输出特征数都是 1）。
- bias 是该层的偏置参数（一个标量）。
- y 是输出数据（也是一个标量，因为输出特征数为 1）。

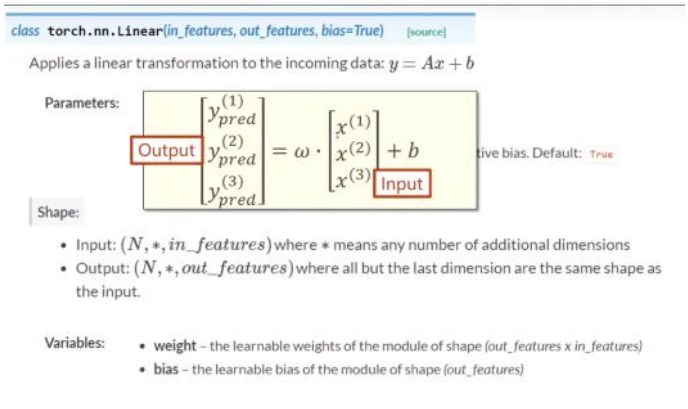
model会自动backward,所以不需要backward

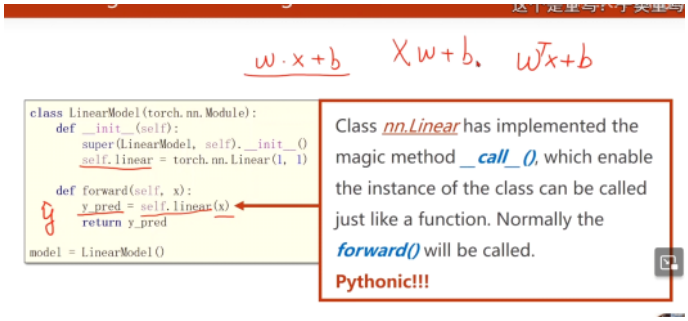


输入x维度
输出y维度

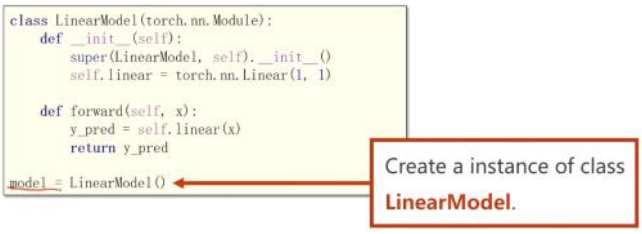


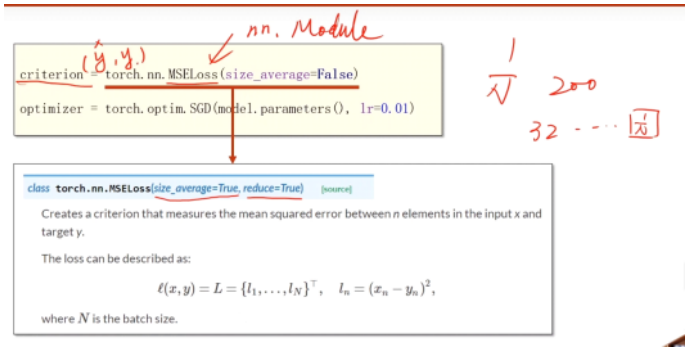


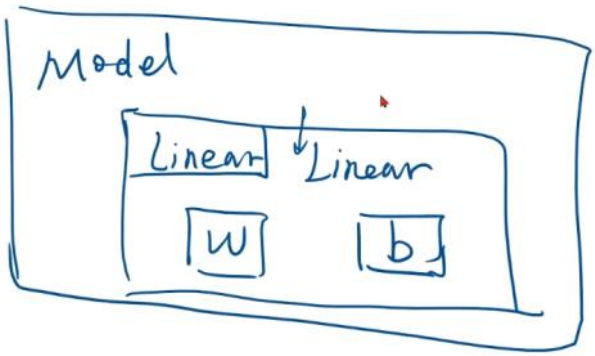


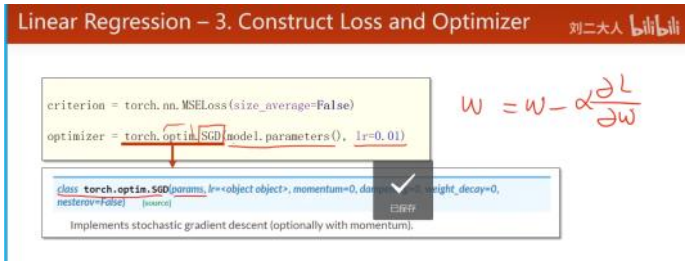


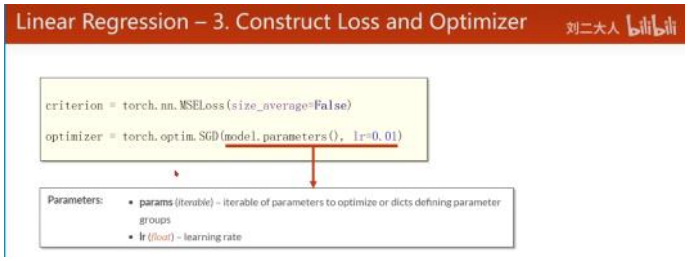
执行model() 会自动调用 LinearModel 的__call__ 然后__call__又会被调用实现的forward()

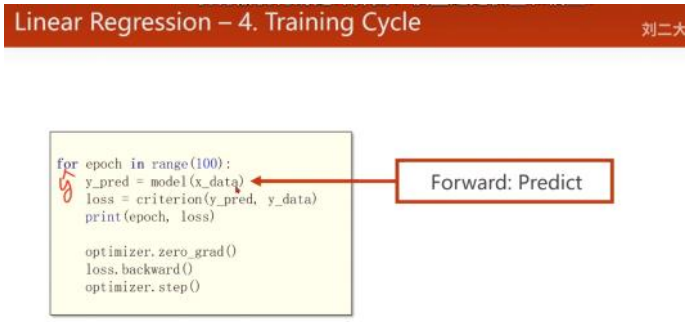




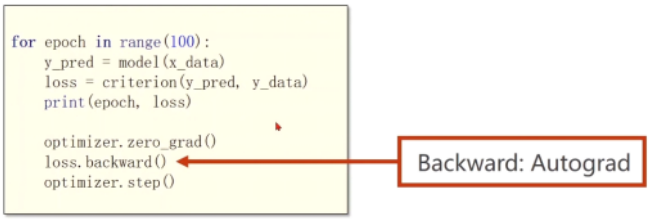




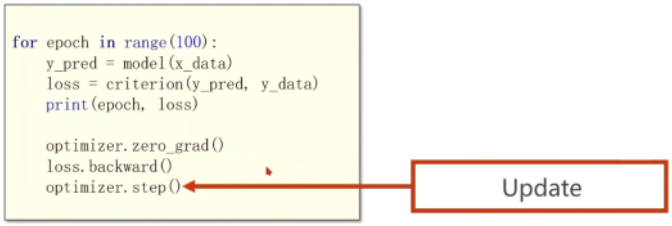




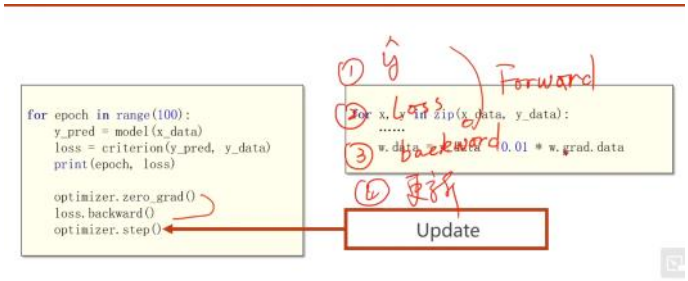
optimizer.zero_grad()：清零梯度，为下一次迭代做准备。

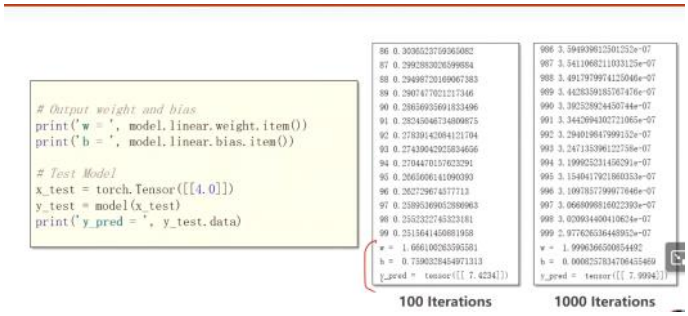


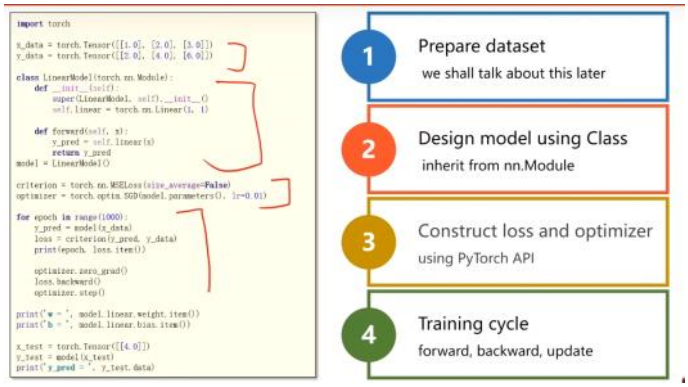
loss.backward()：反向传播，计算梯度。



optimizer.step()：根据计算得到的梯度更新权重。


for epoch in range(100):
y_pred = model(x_data)
loss = criterion(y_pred, y_data)
print(epoch, loss)
optimizer.zero_grad()
loss.backward()
optimizer.step()
Forward
backward
Update


# Output weight and bias
print('w = ', model.linear.weight.item())
print('b = ', model.linear.bias.item())
# Test Model
x_test = torch.Tensor([[4.0]])
y_test = model(x_test)
print('y_pred = ', y_test.data)
100 Iterations
1000 Iterations


import torch
class LinearModel(torch.nn.Module):
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
for epoch in range(1000):
1
Prepare dataset
we shall talk about this later
2
Design model using Class
inherit from nn.Module
3
Construct loss and optimizer
using PyTorch API
4
Training cycle
forward, backward, update

torch.nn.module

2024年4月24日 15:23

https://blog.csdn.net/qq_27825451/article/details/90550890?ops_request_misc=&request_id=&biz_id=102&spm=1018.2226.3001.4187

我们在定义自己的网络的时候，需要继承nn.Module类，并重新实现构造函数__init__构造函数和forward这两个方法。但有一些注意技巧：

（1）一般把网络中具有可学习参数的层（如全连接层、卷积层等）放在构造函数__init__()中，当然我也可以吧不具有参数的层也放在里面；

（2）一般把不具有可学习参数的层(如ReLU、dropout、BatchNormanation层)可放在构造函数中，也可不放在构造函数中，如果不放在构造函数__init__里面，则在forward方法里面可以使用nn.functional来代替

（3）forward方法是必须要重写的，它是实现模型的功能，实现各个层之间的连接关系的核心。

torch.nn.Linear

2024年4月24日 15:52

https://blog.csdn.net/sazass/article/details/123568203?ops_request_misc=%7B%22request%5Fid%22%3A%221713944881168001785971 93%22%2C%22scm%22%3A%2220140713.130102334.pc%5Fall.%22%7D&request_id=171394488116800178597193&biz_id=0&spm=1018.2226.3001.4187

1. nn.Linear的原理:

从名称就可以看出来，nn.Linear表示的是线性变换，原型就是初级数学里学到的线性函数：y=kx+b
不过在深度学习中，变量都是多维张量，乘法就是矩阵乘法，加法就是矩阵加法，因此nn.Linear()运行的真正的计算就是：

output = weight @ input + bias
@: 在python中代表矩阵乘法

input: 表示输入的Tensor，可以有多个维度

weights: 表示可学习的权重，shape=(output_feature,in_feature)

bias: 表示科学习的偏置，shape=(output_feature)

in_feature： nn.Linear 初始化的第一个参数，即输入Tensor最后一维的通道数

out_feature： nn.Linear 初始化的第二个参数，即返回Tensor最后一维的通道数

output: 表示输入的Tensor，可以有多个维度

2. nn.Linear的使用:

常用头文件：import torch.nn as nn
nn.Linear()的初始化：
nn.Linear(in_feature,out_feature,bias)
in_feature： int型, 在forward中输入Tensor最后一维的通道数

out_feature： int型, 在forward中输出Tensor最后一维的通道数

bias: bool型， Linear线性变换中是否添加bias偏置
nn.Linear()的执行： （即执行forward函数）

out=nn.Linear(input)

input: 表示输入的Tensor，可以有多个维度
output: 表示输入的Tensor，可以有多个维度

举例：

2维 Tensor：

```
m = nn.Linear(20, 40)
#规定这个线性层有20个输入特征和40个输出特征，讲一个形状为（N，20）的张量映射到（N,40）的张量上，N是批量大小
input = torch.randn(128, 20)
#创建一个形状为（128，20）的随机张量input，元素是标准正态分布中随机抽取的，即128行，20列（128个样本，每个样本20个特征）
output = m(input)
print(output.size())  # [(128，40])
#我们通过将 input 张量传递给线性层 m 来计算输出 output。由于 m 将每个输入样本映射到 40 个输出特征，所以 output 张量的形状将是 (128, 40)。
最后，我们打印 output 张量的大小，结果确实是 (128, 40)，这符合我们的预期。
```

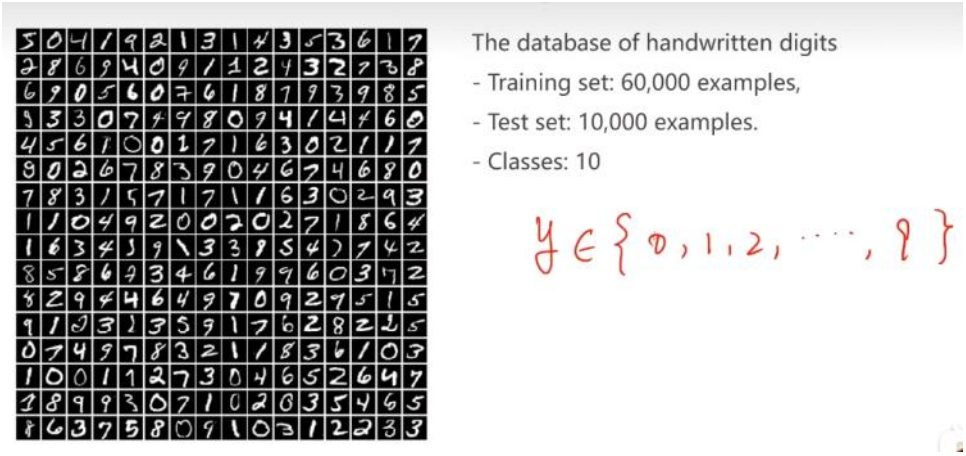
4维 Tensor:

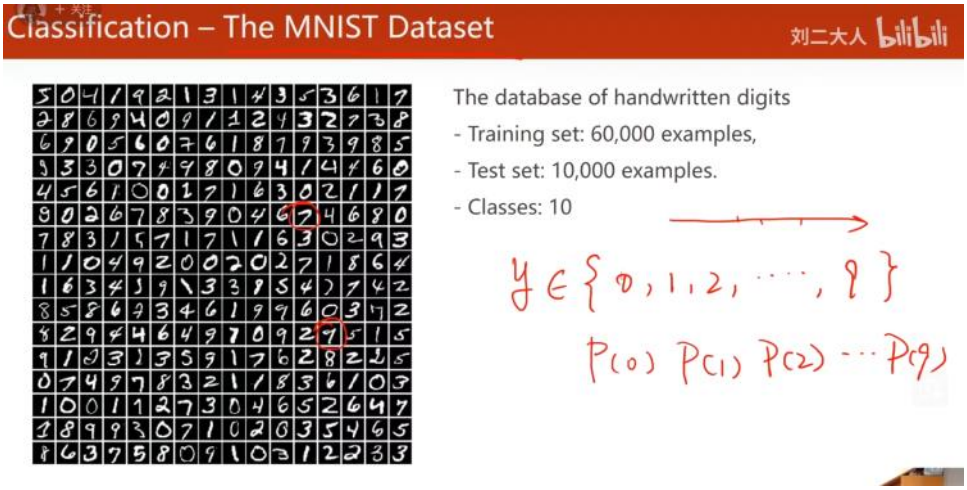
```
m = nn.Linear(128, 64)
input = torch.randn(512, 3,128,128)
output = m(input)
print(output.size())  # [(512, 3,128,64))
```

逻辑斯蒂回归

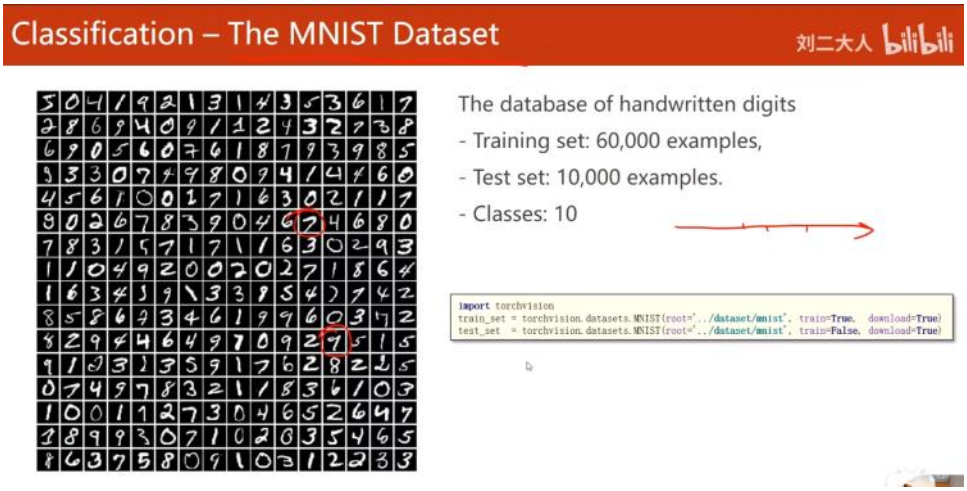
2024年4月11日 8:46

logistics regression实际是分类问题





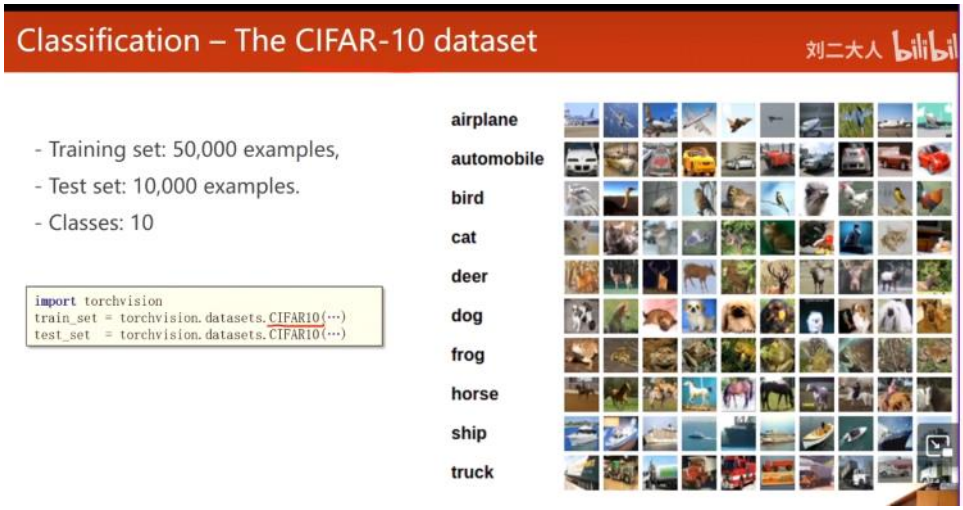
分类得到的是某个数字属于每个分类的概率值，最大的概率是预测的结果



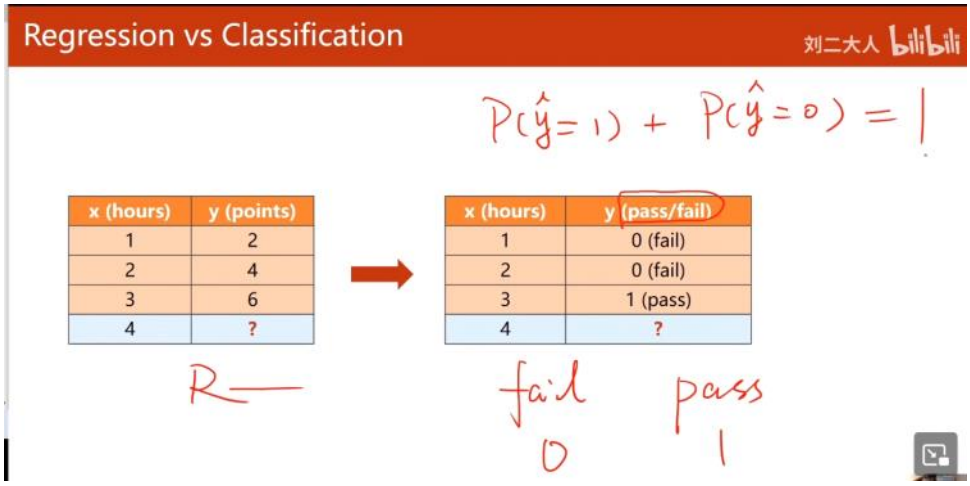
torchvision.datasets打包了很多数据集

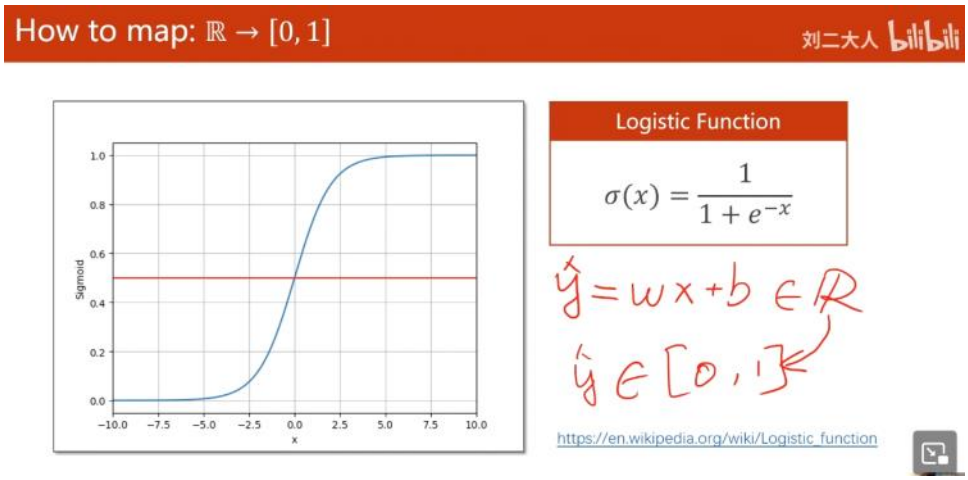
mnist是手写数字数据集

cifar是彩色图片数据集



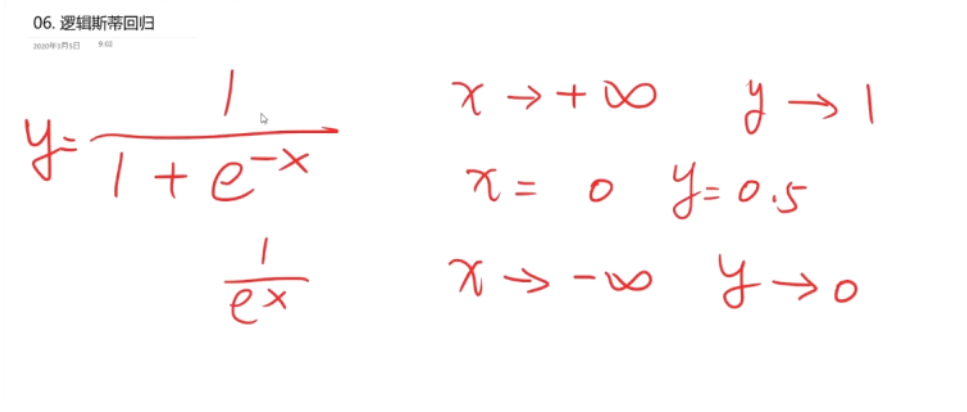
二分类 输出的是一个概率

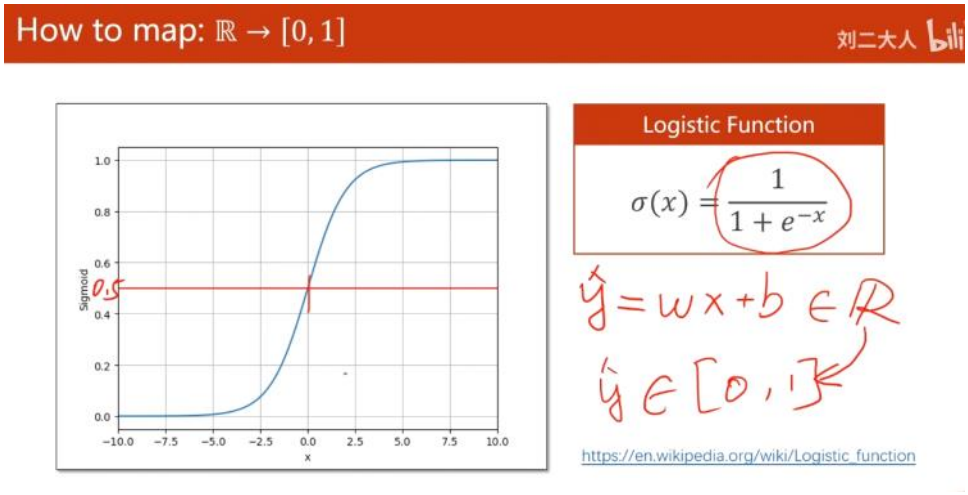


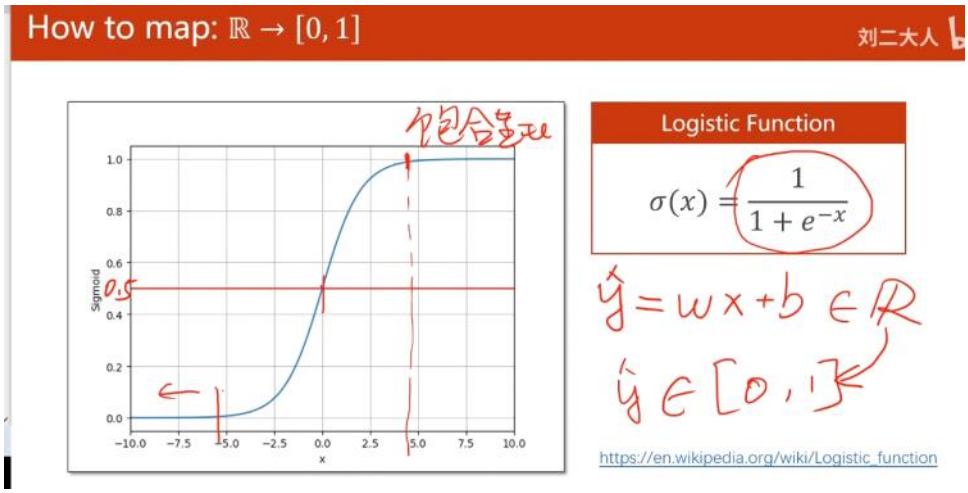


logistic Function（逻辑回归）用的是sigmoid（激活）函数

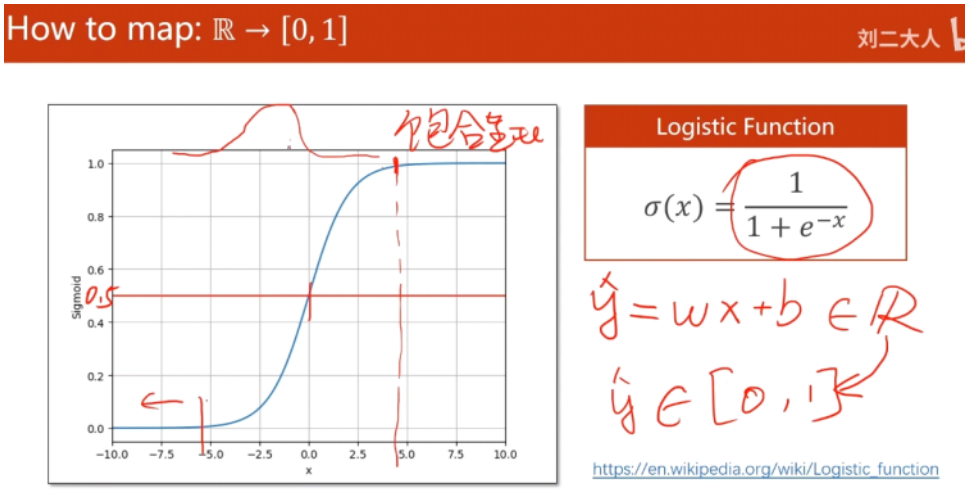
把实数集映射到[0,1]



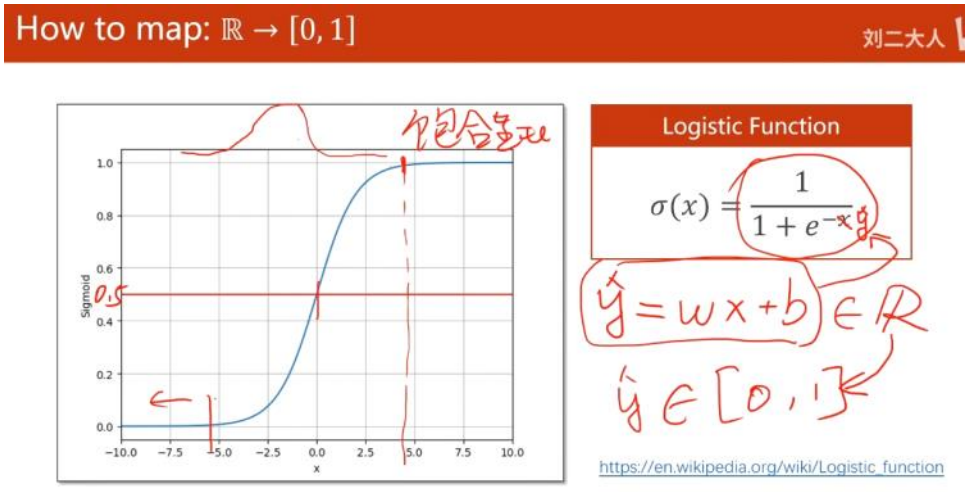




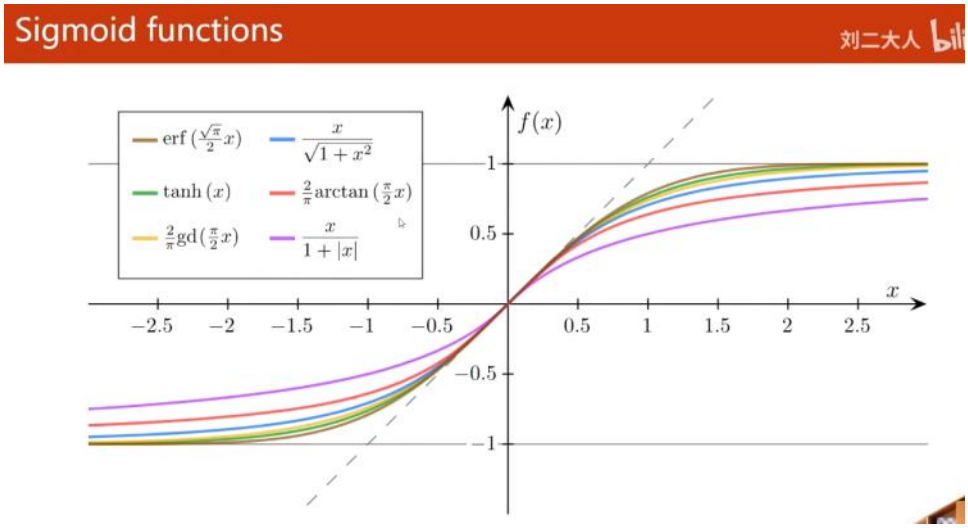
对应的导数图，像正态分布



将y_hat代入logistic function,就能保证得到【0，1】之间的数

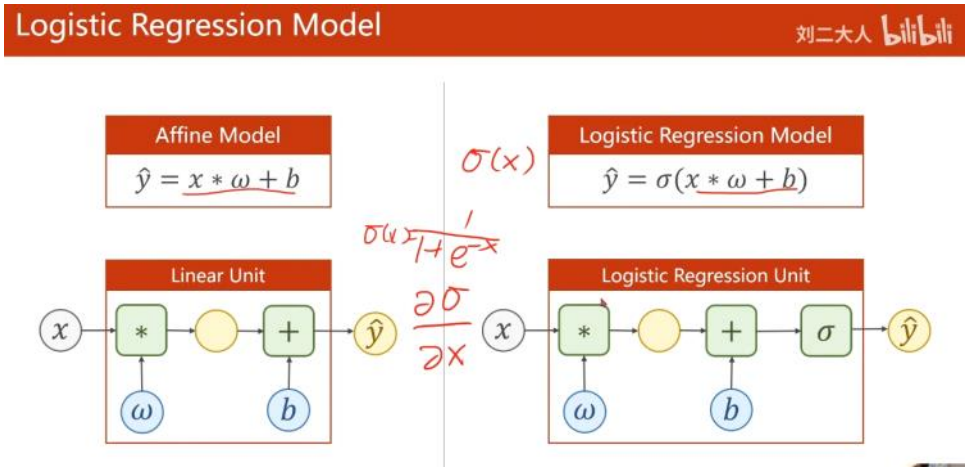


还有其他的sigmoid函数，最典型的上面的logistic函数



σ(西格玛)就是logistic函数

logistic回归单元



loss function for binary classification

二分法

BCE损失

Loss function for Binary Classification

刘二大人 bilibili

Loss Function for Linear Regression

$$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$$

$\hat{y} - y$

$\hat{y} = P(class = 1)$

$(1-\hat{y}) = P(class=0)$

$y>0$

$y = P(class=1)$

$=0$

$1-y = P(class=0)$

$=1$

Loss Function for Binary Classification

$$loss = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

要让loss尽量小，就得括号里面的尽量大

二分类 y=1 要让括号里面的尽量大,y_hat要尽量大

y=0要让括号里面的尽量大，y_hat尽量小

Mini-Batch Loss function for Binary Classification

刘二大人 bilib

Loss Function for Binary Classification

$$loss = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

Mini-Batch Loss Function for Binary Classification

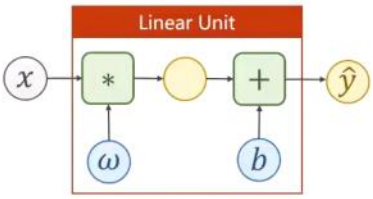
$$loss = -\frac{1}{N}\sum_{n=1}^{N} y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)$$

y	$\hat{y}$	BCE Loss
1	0.2	1.6094
1	0.8	0.2231
0	0.3	0.3567
0	0.7	1.2040
Mini-Batch Loss		0.8483

Implementation of Logistic Regression

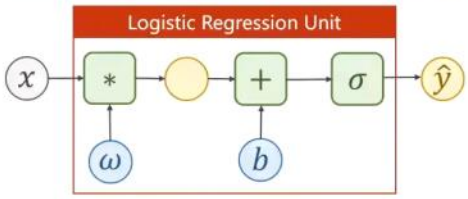
刘二大人 bilibili

Linear Unit




```
class LinearModel(torch.nn.Module):
    def __init__(self):
        super(LinearModel, self).__init__()
        self.linear = torch.nn.Linear(1, 1)

    def forward(self, x):
        y_pred = self.linear(x)
        return y_pred
```

Logistic Regression Unit




```
import torch.nn.functional as F

class LogisticRegressionModel(torch.nn.Module):
    def __init__(self):
        super(LogisticRegressionModel, self).__init__()
        self.linear = torch.nn.Linear(1, 1)

    def forward(self, x):
        y_pred = F.sigmoid(self.linear(x))
        return y_pred
```

logistic（sigmoid）函数在functional函数下

Implementation of Logistic Regression

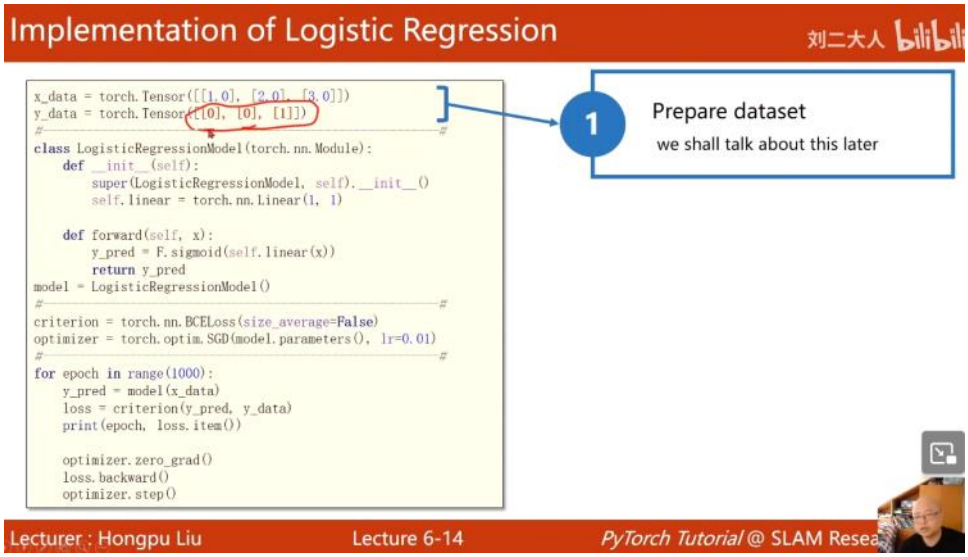
Mini-Batch Loss Function for Binary Classification

$$loss = -\frac{1}{N}\sum_{n=1}^{N} y_n \log \hat{y}_n + (1 - y_n)\log(1 - \hat{y}_n)$$

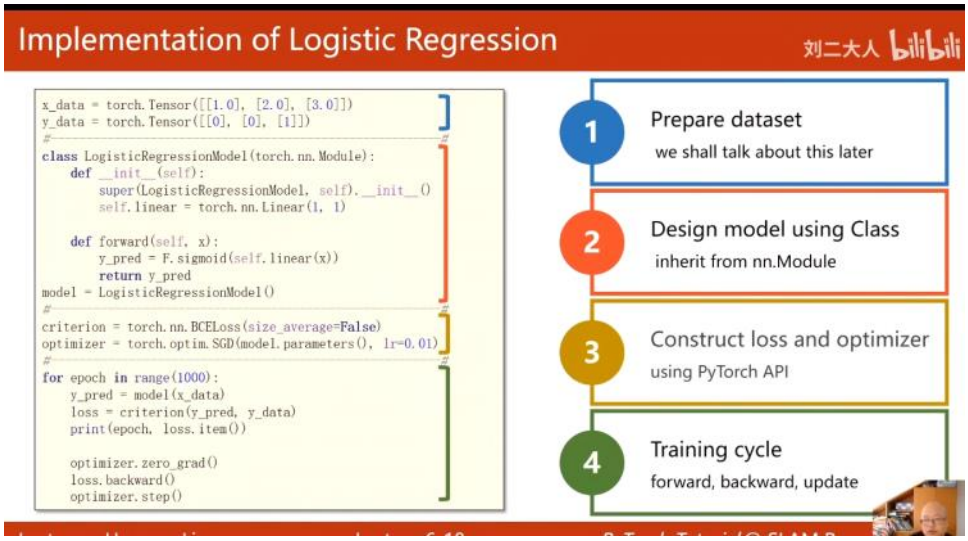
MSE

```
criterion = torch.nn.BCELoss(size_average=False)
```

原来算损失是MSE，现在用BCE 就是交叉熵



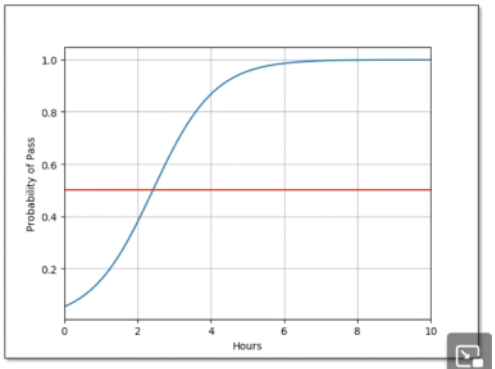
数据准备上的变化：y集变成分类



用model验证一个例子

Result of Logistic Regression


```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 200)
x_t = torch.Tensor(x).view((200, 1))
y_t = model(x_t)
y = y_t.data.numpy()
plt.plot(x, y)
plt.plot([0, 10], [0.5, 0.5], c='r')
plt.xlabel('Hours')
plt.ylabel('Probability of Pass')
plt.grid()
plt.show()
```



交叉熵

2024年4月11日 10:29

交叉熵损失函数(CrossEntropy Loss)
https://blog.csdn.net/tsyccnh/article/details/79163834?ops_request_misc=%7B%22request%5Fid%22%3A%22171280268816800226583691%22%2C%22scm%22%3A%2220140713.130102334.pc%5Fall.%22%7D&request_id=171280268816800226583691&biz_id=0&spm=1018.2226.3001.4187

信息论

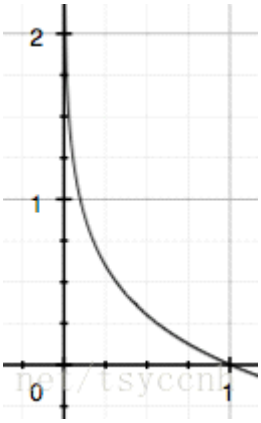
交叉熵是信息论中的一个概念，要想了解交叉熵的本质，需要先从最基本的概念讲起。

1 信息量

首先是信息量。假设我们听到了两件事，分别如下：
事件A：巴西队进入了2018世界杯决赛圈。
事件B：中国队进入了2018世界杯决赛圈。
仅凭直觉来说，显而易见事件B的信息量比事件A的信息量要大。究其原因，是因为事件A发生的概率很大，事件B发生的概率很小。所以当越不可能的事件发生了，我们获取到的信息量就越大。越可能发生的事件发生了，我们获取到的信息量就越小。那么信息量应该和事件发生的概率有关。
假设X是一个离散型随机变量，其取值集合为χ,概率分布函数$p(x)=Pr(X=x),x\in\chi$,则定义事件$X=x_0$的信息量为：

$$I(x_0) = -log(p(x_0))$$

由于是概率所以 p(x0)的取值范围是 [0,1],绘制为图形如下：



可见该函数符合我们对信息量的直觉

2 熵

考虑另一个问题，对于某个事件，有n种可能性，每一种可能性都有一个概率p(xi)
这样就可以计算出某一种可能性的信息量。举一个例子，假设你拿出了你的电脑，按下

开关，会有三种可能性，下表列出了每一种可能的概率及其对应的信息量

序号	事件	概率p	信息量I
A	电脑正常开机	0.7	-log(p(A))=0.36
B	电脑无法开机	0.2	-log(p(B))=1.61
C	电脑爆炸了	0.1	-log(p(C))=2.30

注：文中的对数均为自然对数

我们现在有了信息量的定义，而熵用来表示所有信息量的期望，即：

$$H(X) = -\sum_{i=1}^{n} p(x_i) log(p(x_i))$$

其中n代表所有的n种可能性，所以上面的问题结果就是

$$\begin{aligned} H(X) &= -[p(A)log(p(A)) + p(B)log(p(B)) + p(C))log(p(C))] \\ &= 0.7 \times 0.36 + 0.2 \times 1.61 + 0.1 \times 2.30 \\ &= 0.804 \end{aligned}$$

然而有一类比较特殊的问题，比如投掷硬币只有两种可能，字朝上或花朝上。买彩票只有两种可能，中奖或不中奖。我们称之为0-1分布问题（二项分布的特例），对于这类问题，熵的计算方法可以简化为如下算式：

$$\begin{aligned} H(X) &= -\sum_{i=1}^{n} p(x_i) log(p(x_i)) \\ &= -p(x) log(p(x)) - (1 - p(x)) log(1 - p(x)) \end{aligned}$$

3 相对熵（KL散度）

相对熵又称KL散度,如果我们对于同一个随机变量 x 有两个单独的概率分布 P(x) 和 Q(x)，我们可以使用 KL 散度（Kullback-Leibler (KL) divergence）来衡量这两个分布的差异

在机器学习中，P往往用来表示样本的真实分布，比如[1,0,0]表示当前样本属于第一类。Q用来表示模型所预测的分布，比如[0.7,0.2,0.1]

直观的理解就是如果用P来描述样本，那么就非常完美。而用Q来描述样本，虽然可以大致描述，但是不是那么的完美，信息量不足，需要额外的一些“信息增量”才能达到和P一样完美的描述。如果我们的Q通过反复训练，也能完美的描述样本，那么就不再需要额外的“信息增量”，Q等价于P。

KL散度的计算公式：

$$D_{KL}(p||q) = \sum_{i=1}^{n} p(x_i) log(\frac{p(x_i)}{q(x_i)})$$

n为事件的所有可能性。

D_{KL}的值越小，表示q分布和p分布越接近

4 交叉熵

对式3.1变形可以得到：

$$D_{KL}(p||q) = \sum_{i=1}^{n} p(x_i)log(p(x_i)) - \sum_{i=1}^{n} p(x_i)log(q(x_i))$$
$$= -H(p(x)) + [-\sum_{i=1}^{n} p(x_i)log(q(x_i))]$$

等式的前一部分恰巧就是p的熵，等式的后一部分，就是交叉熵：

$$H(p,q) = -\sum_{i=1}^{n} p(x_i)log(q(x_i))$$

在机器学习中，我们需要评估label和predicts之间的差距，使用KL散度刚刚好，即DKL(y||y_hat)，由于KL散度中的前一部分-H(y)不变，故在优化过程中，只需要关注交叉熵就可以了。所以一般在机器学习中直接用用交叉熵做loss，评估模型。

机器学习中交叉熵的应用

1 为什么要用交叉熵做loss函数?

在线性回归问题中，常常使用MSE（Mean Squared Error）作为loss函数

2 交叉熵在单分类问题中的使用

这里的单类别是指，每一张图像样本只能有一个类别，比如只能是狗或只能是猫。

交叉熵在单分类问题上基本是标配的方法

$$loss = -\sum_{i=1}^{n} y_i log(\hat{y}_i)$$

上式为一张样本的loss计算方法。式2.1中n代表着n种类别。

举例说明,比如有如下样本

对应的标签和预测值

*	猫	青蛙	老鼠
Label	0	1	0
Pred	0.3	0.6	0.1

那么

$$\begin{aligned} loss &= -(0 \times log(0.3) + 1 \times log(0.6) + 0 \times log(0.1) \\ &= -log(0.6) \end{aligned}$$

对应一个batch的loss就是

$$loss = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{n} y_{ji} log(\hat{y_{ji}})$$

m为当前batch的样本数

3 交叉熵在多分类问题中的使用

这里的多类别是指，每一张图像样本可以有多个类别，比如同时包含一只猫和一只狗

和单分类问题的标签不同，多分类的标签是n-hot。

比如下面这张样本图，即有青蛙，又有老鼠，所以是一个多分类问题

对应的标签和预测值

*	猫	青蛙	老鼠
Label	0	1	1
Pred	0.1	0.7	0.8

值得注意的是，这里的Pred不再是通过softmax计算的了，这里采用的是sigmoid。将每一个节点的输出归一化到[0,1]之间。所有Pred值的和也不再为1。换句话说，就是每一个Label都是独立分布的，相互之间没有影响。所以交叉熵在这里是单独对每一个节点进行计算，每一个节点只有两种可能值，所以是一个二项分布。前面说过对于二项分布这种特殊的分布，熵的计算可以进行简化。

同样的，交叉熵的计算也可以简化，即

$$loss = -ylog(\hat{y}) - (1-y)log(1-\hat{y})$$

注意，上式只是针对一个节点的计算公式。这一点一定要和单分类loss区分开来。
例子中可以计算为：

$$loss_{猫} = -0 \times log(0.1) - (1-0)log(1-0.1) = -log(0.9)$$
$$loss_{蛙} = -1 \times log(0.7) - (1-1)log(1-0.7) = -log(0.7)$$
$$loss_{鼠} = -1 \times log(0.8) - (1-1)log(1-0.8) = -log(0.8)$$

单张样本的loss即为loss=loss猫+loss蛙+loss鼠
每一个batch的loss就是：

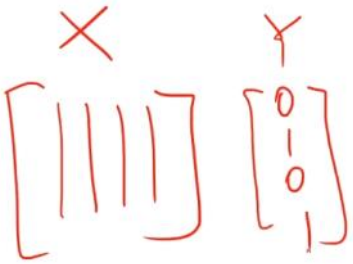
$$loss = \sum_{j=1}^{m}\sum_{i=1}^{n} -y_{ji}log(\hat{y_{ji}}) - (1-y_{ji})log(1-\hat{y_{ji}})$$

式中m为当前batch中的样本量，n为类别数。

处理多维特征的输入

2024年4月12日 10:28

之前的例子都是单维
multiple dimension input
现在求x是多维度的 求y的分类



之前的两个例子

x (hours)	y (points)
1	2
2	4
3	6
4	?

回归 $y \in \mathbb{R}$

x (hours)	y (pass/fail)
1	0 (fail)
2	0 (fail)
3	1 (pass)
4	?

分类 $y \in \{\ \}$

例子：
diabetes dataset，x1-8表示特征，Y表示病情是否会加重
每一行是一个样本
数据库里每行是record
每一列是一个特征
数据库的列是字段

Diabetes Dataset 刘二大人

X1	X2	X3	X4	X5	X6	X7	X8	Y
-0.29	0.49	0.18	-0.29	0.00	0.00	-0.53	-0.03	0
-0.88	-0.15	0.08	-0.41	0.00	-0.21	-0.77	-0.67	1
-0.06	0.84	0.05	0.00	0.00	-0.31	-0.49	-0.63	0
-0.88	-0.11	0.08	-0.54	-0.78	-0.16	-0.92	0.00	1
0.00	0.38	-0.34	-0.29	-0.60	0.28	0.89	-0.60	0
-0.41	0.17	0.21	0.00	0.00	-0.24	-0.89	-0.70	1
-0.65	-0.22	-0.18	-0.35	-0.79	-0.08	-0.85	-0.83	0
0.18	0.16	0.00	0.00	0.00	0.05	-0.95	-0.73	1
-0.76	0.98	0.15	-0.09	0.28	-0.09	-0.93	0.07	0
-0.06	0.26	0.57	0.00	0.00	0.00	-0.87	0.10	0

Sample

分类

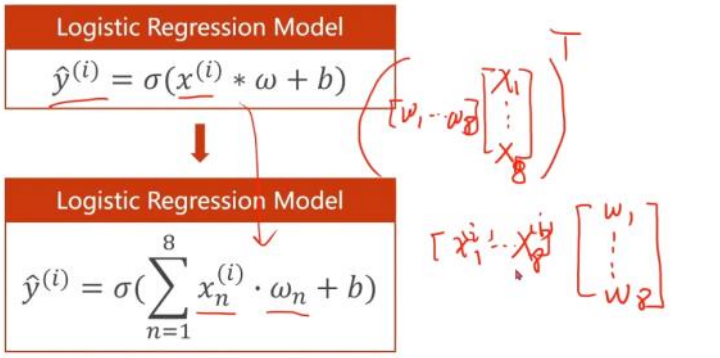
Diabetes Dataset

X1	X2	X3	X4	X5	X6	X7	X8	Y
-0.29	0.49	0.18	-0.29	0.00	0.00	-0.53	-0.03	0
-0.88	-0.15	0.08	-0.41	0.00	-0.21	-0.77	-0.67	1
-0.06	0.84	0.05	0.00	0.00	-0.31	-0.49	-0.63	0
-0.88	-0.11	0.08	-0.54	-0.78	-0.16	-0.92	0.00	1
0.00	0.38	-0.34	-0.29	-0.60	0.28	0.89	-0.60	0
-0.41	0.17	0.21	0.00	0.00	-0.24	-0.89	-0.70	1
-0.65	-0.22	-0.18	-0.35	-0.79	-0.08	-0.85	-0.83	0
0.18	0.16	0.00	0.00	0.00	0.05	-0.95	-0.73	1
-0.76	0.98	0.15	-0.09	0.28	-0.09	-0.93	0.07	0
-0.06	0.26	0.57	0.00	0.00	0.00	-0.87	0.10	0

Feature 特征/字段

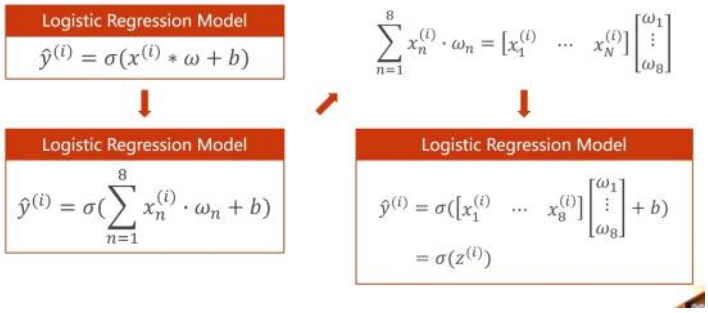
把x装到x 把y装到y

x从原来的单个维度变到8个维度



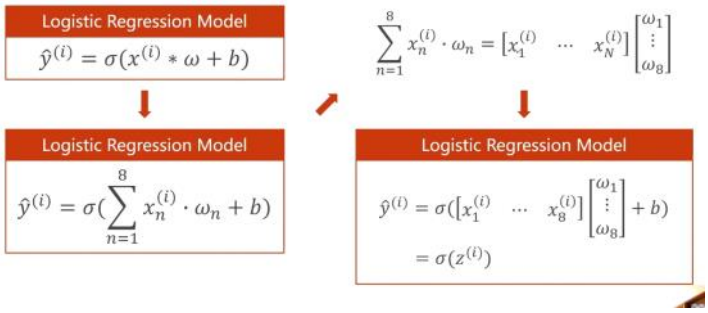
Multiple Dimension Logistic Regression Model

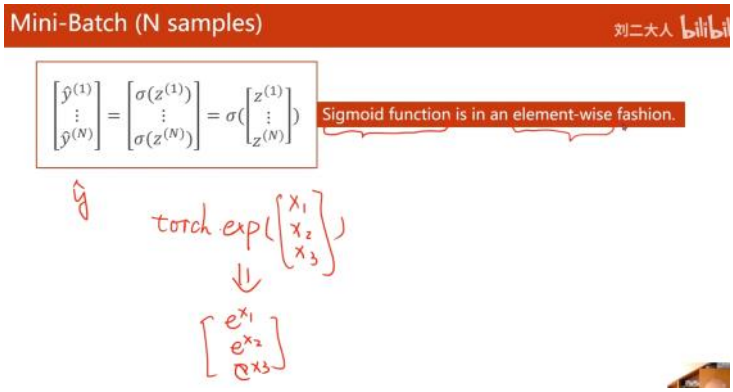
刘二大人

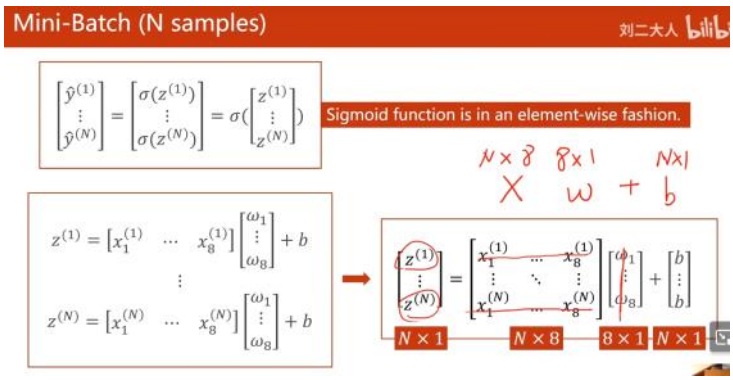


Multiple Dimension Logistic Regression Model

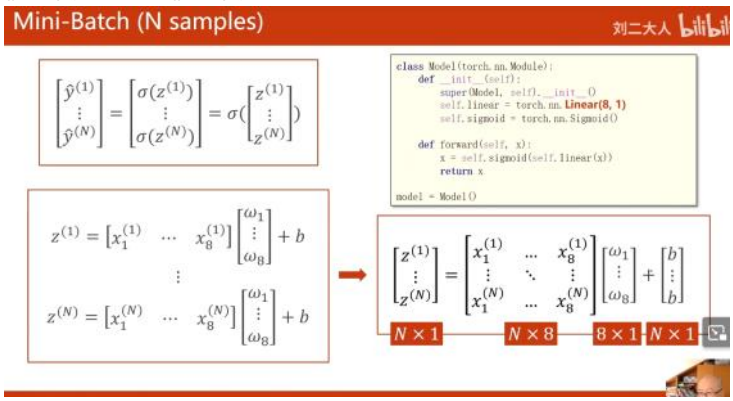
刘二大人

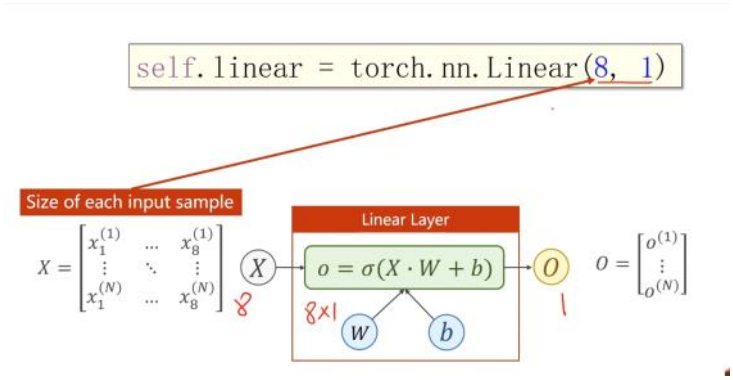






输入维度改成8，输出维度改成1

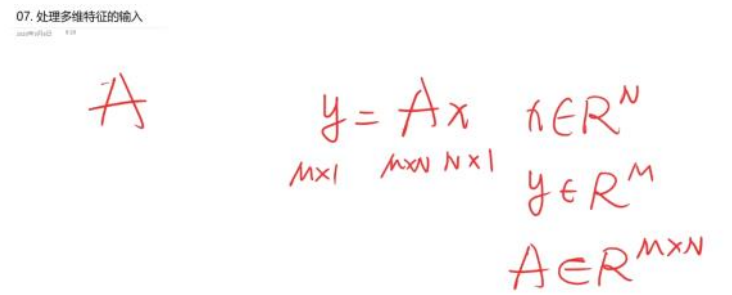




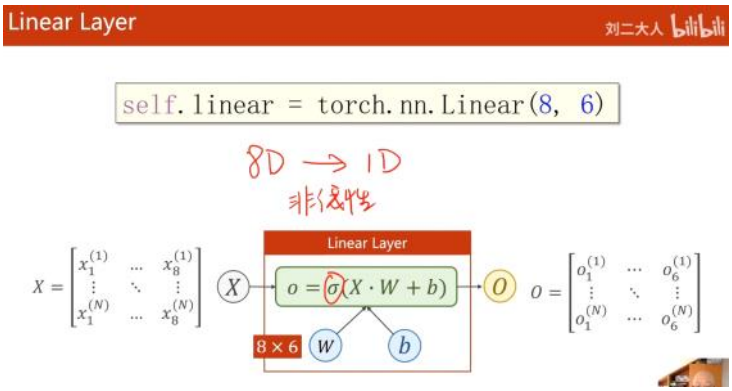
如果是多层神经网络呢?

处理多维特征的输入

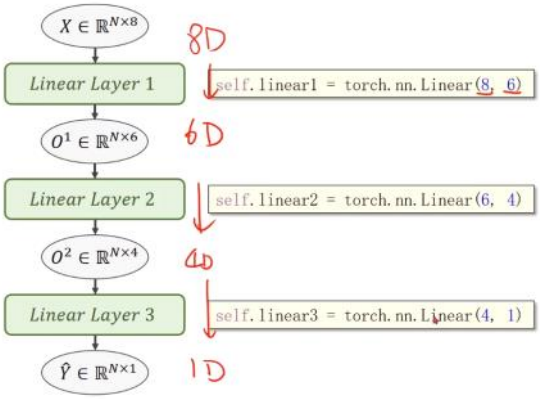
矩阵是空间变换的函数



Linear Layer



Example: Artificial Neural Network



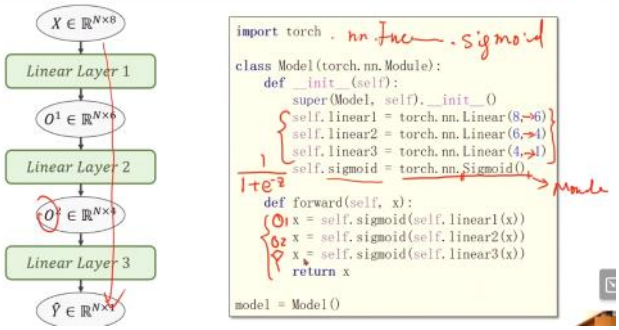
Example: 1. Prepare Dataset

```
import numpy as np
xy = np.loadtxt('diabetes.csv.gz', delimiter=',', dtype=np.float32)
x_data = torch.from_numpy(xy[:, :-1])
y_data = torch.from_numpy(xy[:, [-1]])
```

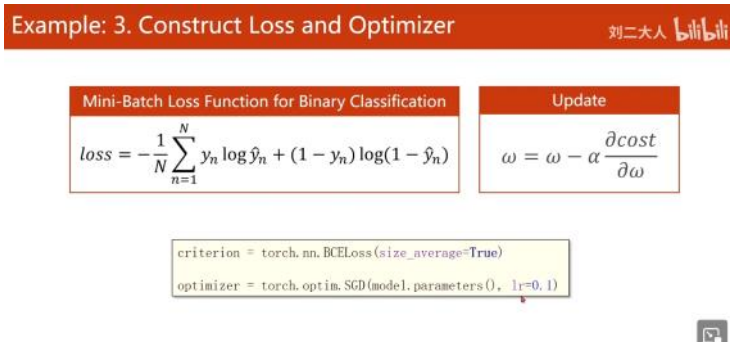
```
-0.294118,0.487437,0.180328,-0.292929,0,0.00149028,-0.53117,-0.0333333,0
-0.882353,-0.145729,0.0819672,-0.414141,0,-0.207153,-0.766866,-0.666667,1
-0.0588235,0.839196,0.0491803,0,0,-0.305514,-0.492741,-0.633333,0
-0.882353,-0.105528,0.0819672,-0.535354,-0.777778,-0.162444,-0.923997,0,1
0,0.376884,-0.344262,-0.292929,-0.602837,0.28465,0.887276,-0.6,0
-0.411765,0.165829,0.213115,0,0,-0.23696,-0.894962,-0.7,1
-0.647059,-0.21608,-0.180328,-0.353535,-0.791962,-0.0760059,-0.854825,-0.833333,0
0.176471,0.155779,0,0,0,0.052161,-0.952178,-0.733333,1
-0.764706,0.979899,0.147541,-0.0909091,0.283688,-0.0909091,-0.931682,0.0666667,0
-0.0588235,0.256281,0.57377,0,0,0,-0.868488,0.1,0
-0.529412,0.105528,0.508197,0,0,0.120715,-0.903501,-0.7,1
0.176471,0.688442,0.213115,0,0,0.132638,-0.608027,-0.566667,0
0.176471,0.396985,0.311475,0,0,-0.19225,0.163962,0.2,1
-0.882353,0.899497,-0.0163934,-0.535354,1,-0.102832,-0.726729,0.266667,0
```

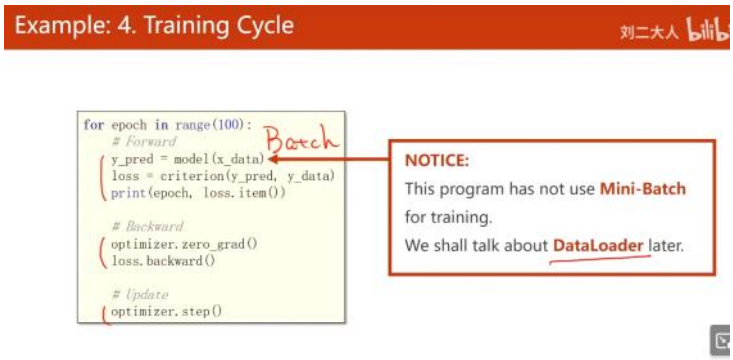
- delimiter表示分割符
- x_data = torch.from_numpy(xy[:, :-1])
  xy[:, :-1]使用NumPy的切片功能选择xy数组的所有行（:）和除最后一列外的所有列（:-1）
- torch.from_numpy()函数将NumPy数组转换为PyTorch张量。
- y_data = torch.from_numpy(xy[:, [-1]])
  这行代码将xy数组的最后一列转换为PyTorch张量。
  xy[:, [-1]]选择xy数组的所有行（:）和最后一列（[-1]）。注意这里使用了一个列表[-1]作为索引，这是因为当使用NumPy的切片来选择单列时，需要这样做来保持数据的二维形状（矩阵），不能是向量。

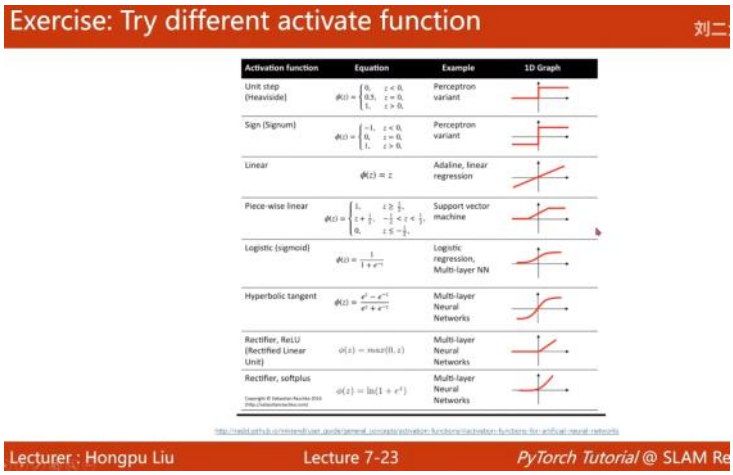
Example: 2. Define Model

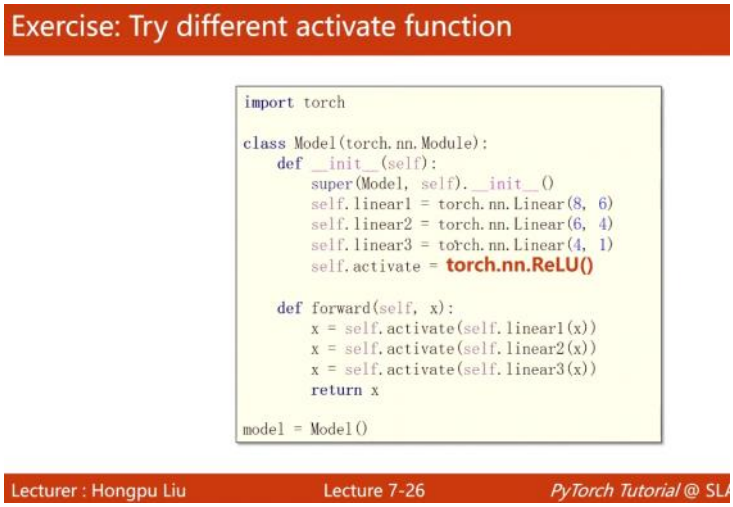



```
import torch . nn.functional.sigmoid

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear1 = torch.nn.Linear(8,6)
        self.linear2 = torch.nn.Linear(6,4)
        self.linear3 = torch.nn.Linear(4,1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.linear1(x))
        x = self.sigmoid(self.linear2(x))
        x = self.sigmoid(self.linear3(x))
        return x

model = Model()
```

- 

- 

- 

- 换一个激活函数 比如ReLU()
- 

加载数据集

2024年4月18日 19:59

dataset and dataloader

如何使用mini-batch

嵌套的两层循环

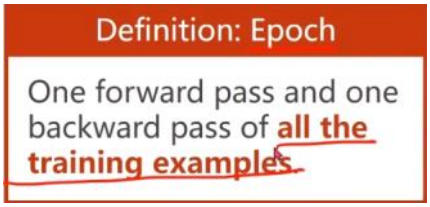
- 外层表示训练的周期，
- 内层表示训练的迭代，每一次迭代执行一个mini-batch

```
# Training cycle
for epoch in range(training_epochs):
    # Loop over all batches
    for i in range(total_batch):
```

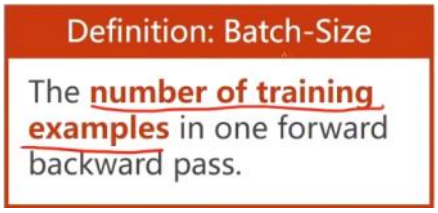
Mini-Batch

#mini-batch是每次取的样本个数，total_batch是取了多少次，两者的乘积才是总的样本个数

什么是epoch?

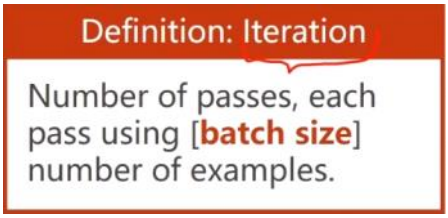


什么是batch-size?(总共10000个数据集，每次拿1000个训练，1000个就是batch-size)



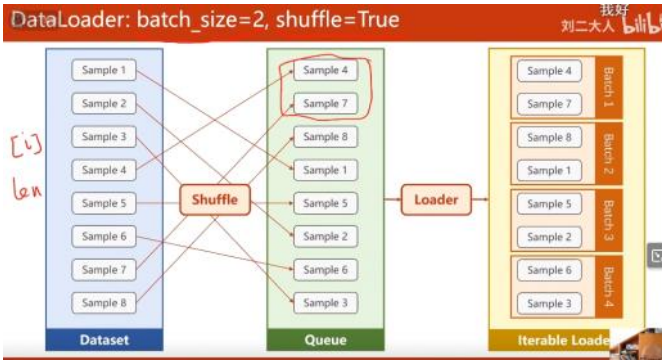
什么是iteration?内层的迭代执行了多少次

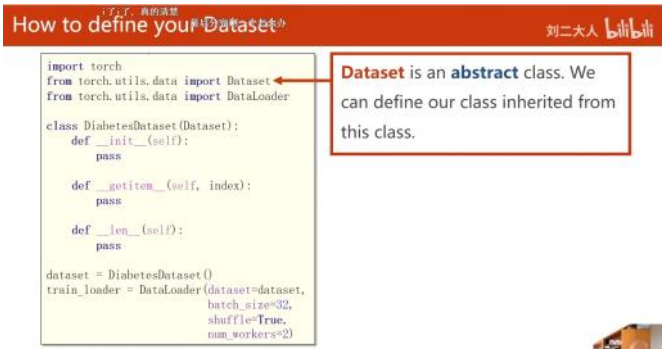
(总共10000个数据集，每次拿1000个训练，1000个就是batch-size,10就是iteration)



dataloader:batch_size=2,shuffle=True

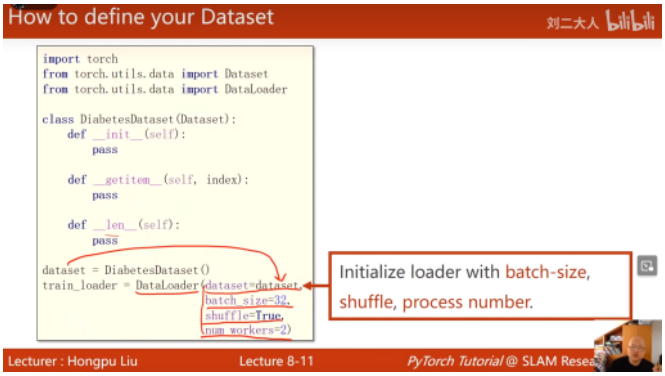
shuffle打乱顺序，要打乱顺序，说明要支持索引





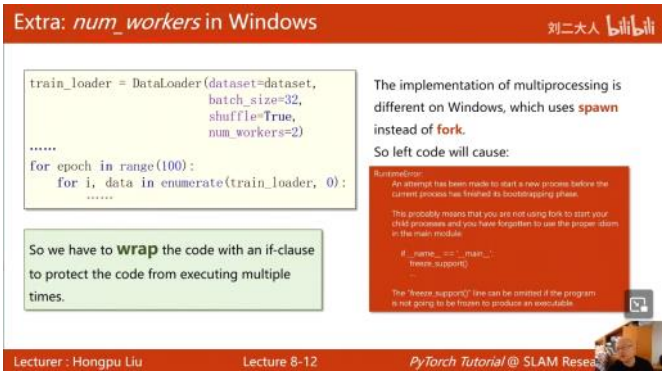
举例：糖尿病数据集，继承父类的一些功能、魔法方法

getitem对非结构数据图像、语音，难以一次性读完的大数据，可以先读检索

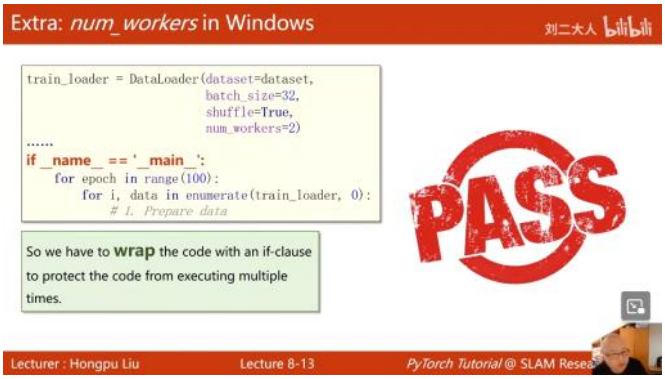


num workers有几个并行的多线程去读取数据，如果CPU高的话可以设多一点

但是这样windows会报错

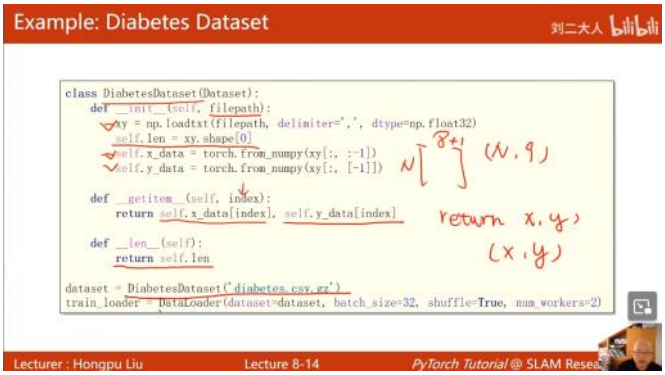


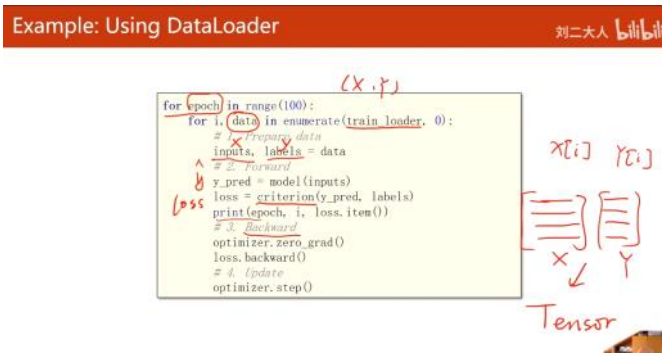
所以要封装代码或者封装函数，改写成if_name =='_main_'

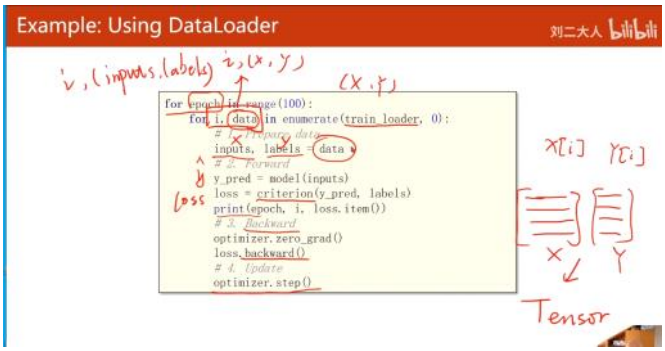


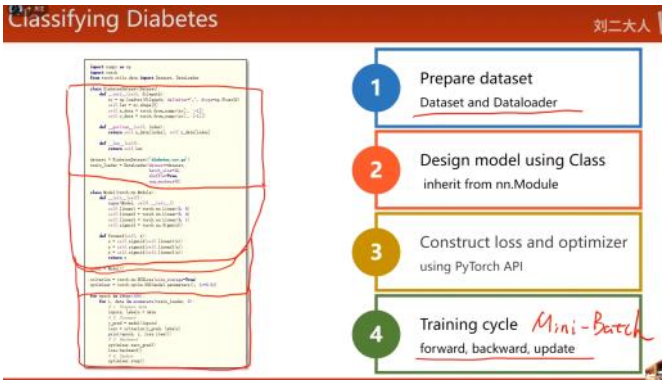
xy.shape[0]取行数

return返回的是（x,y）的元组









`enumerate` 是 Python 中的一个内置函数，用于将一个可遍历的数据对象（如列表、元组或字符串）组合为一个索引序列，同时列出数据和数据下标，一般用在 `for` 循环中。

```
enumerate(iterable, start=0)
```

- iterable：一个可遍历的对象，如列表、元组或字符串。
- start：可选参数，用于指定计数的起始值。默认为 0。

enumerate 函数返回一个枚举对象，该对象生成由数据对象元素及其索引（默认从 0 开始）组成的元组。你可以通过 for循环来遍历这个枚举对象。

```
# 列表
my_list = ['apple', 'banana', 'cherry']

# 使用 enumerate 遍历列表
for index, value in enumerate(my_list):
    print(f"Index: {index}, Value: {value}")
```

输出将会是：
Index: 0, Value: apple
Index: 1, Value: banana
Index: 2, Value: cherry

接下来test

2024年4月25日 19:11

以下是测试模型的基本步骤：

- 准备测试数据集：确保您有一个测试数据集，并且它与训练数据集分开。测试数据集应该遵循与训练数据集相同的格式。
- 加载测试数据集：将测试数据集加载到Dataset对象中，类似于您为训练数据集所做的那样。
- 创建测试数据加载器：使用DataLoader为测试数据集创建一个数据加载器。通常，**测试时不需要打乱数据（shuffle=False），并且可能不需要多个工作进程（num_workers=0）。**
- 设置模型为评估模式：在测试之前，使用model.eval()将模型设置为评估模式。这通常涉及禁用某些层（如Dropout或BatchNorm）的训练时行为。
- 遍历测试数据集：使用测试数据加载器遍历测试数据集，并对每个批次的数据进行预测。
- 计算测试损失和指标：使用模型的预测和测试数据集中的真实标签来计算测试损失。您还可以计算其他指标，如准确率。

dataset和dataloader

2024年4月24日 14:17

https://blog.csdn.net/jiebaoshayebuhui/article/details/130439027?ops_request_misc=&request_id=&biz_id=102&spm=1018.2226.3001.4187

Dataset和DataLoader都是用来帮助我们加载数据集的两个重要工具类。**Dataset用来构造支持索引的数据集**

在训练时需要在全部样本中拿出小批量数据参与每次的训练，因此我们需要使用DataLoader，即**DataLoader是用来在Dataset里取出一组数据(mini-batch)供训练时快速使用的**

一、Dataset简介及用法

Dataset本质上就是一个抽象类，可以把数据封装成Python可以识别的数据结构。
Dataset类不能实例化，所以在使用Dataset的时候，我们需要定义自己的数据集类，也是Dataset的子类，来继承Dataset类的属性和方法。
Dataset可作为DataLoader的参数传入DataLoader，实现基于张量的数据预处理。

Dataset主要有两种类型，分别为Map-style datasets和Iterable-style datasets

Map-style datasets类型：
实现了__getitem__()和__len__()方法，它代表数据的索引到真正数据样本的映射。
读取的数据并非直接把所有数据读取出来，而是读取的数据的索引或者键值,这种类型是使用最多的类型，采用这种访问数据的方式可以大大节约训练时需要的内存数量，提高模型的训练效率

Iterable-style datasets类型：
实现了__iter__()方法，与上述类型不同之处在于，他会将真实的数据全部载入，然后在整个数据集上进行迭代 这种读取数据的方式比较适合处理流数据

自己定义子类：

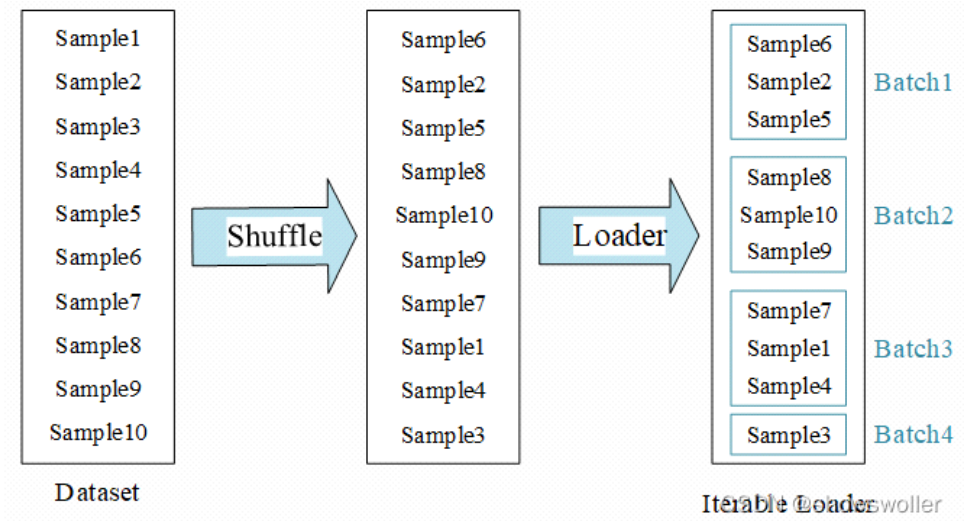
上面我们提到，Dataset作为一个抽象类，需要定义其子类来实例化。所以我们需要自己定义其子类或者使用已经定义好的子类

必须要继承已经内置的抽象类dataset 必须要重写其中的__init__()方法、__getitem__()方法和__len__()方法
其中__getitem__()方法实现通过给定的索引遍历数据样本，__len__()方法实现返回数据的条数

二、DataLoader简介及用法

Dataset和DataLoader是一起使用的，在模型训练的过程中不断为模型提供数据，同时，使用Dataset加载出来的数据集也是DataLoader的第一个参数。所以，DataLoader本质上就是用来将已经加载好的数据以模型能够接收的方式输入到即将训练的模型中去

数据的输入过程



Data_size=10，Batch_size=3，一次Epoch需要四次Iteration，第一列为所有样本，第二列为打乱之后的所有样本，由于Batch_size=3，所以通过DataLoader输入了4个batch，包括最后一个数量已经不够3个的Batch4，里边只包含sample3

DataLoader函数参数

Dataset：通过上一节Dataset加载出来的数据集

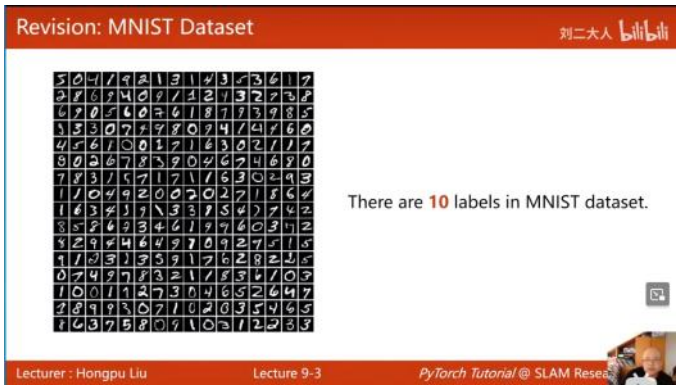
batch_size：每个batch加载多少个样本

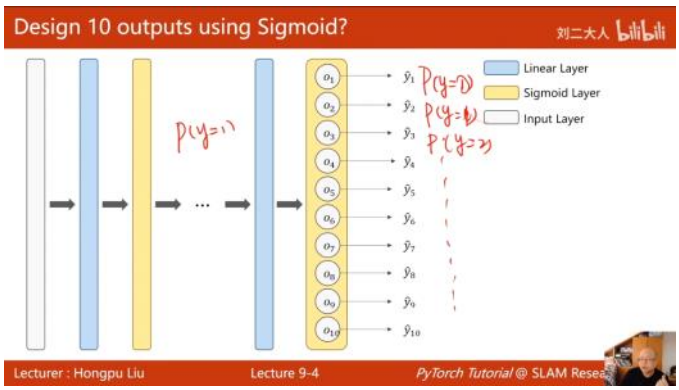
shuffle：是否打乱输入数据的顺序

多分类问题

2024年4月25日　20:24

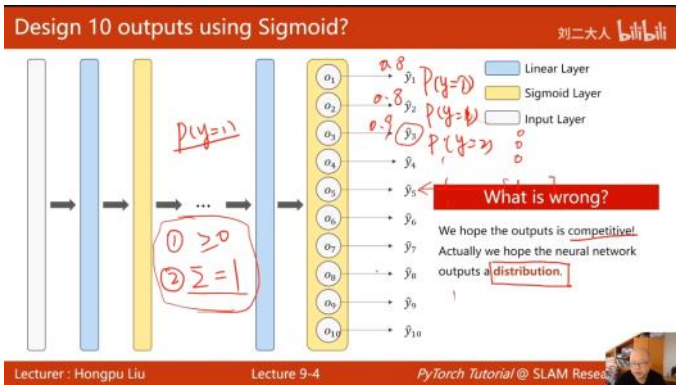
softmax解决多分类问题



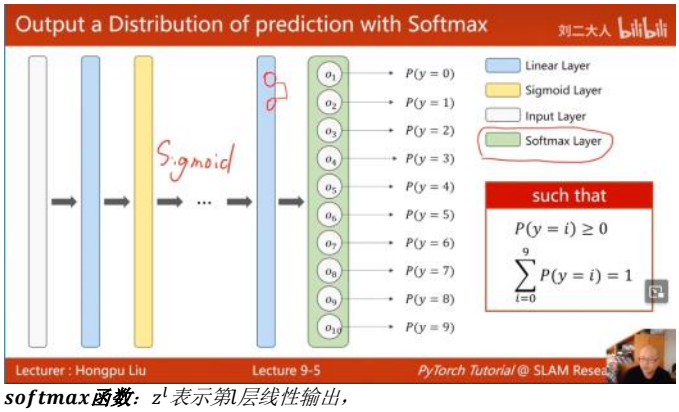


如果一个问题有10个分类?

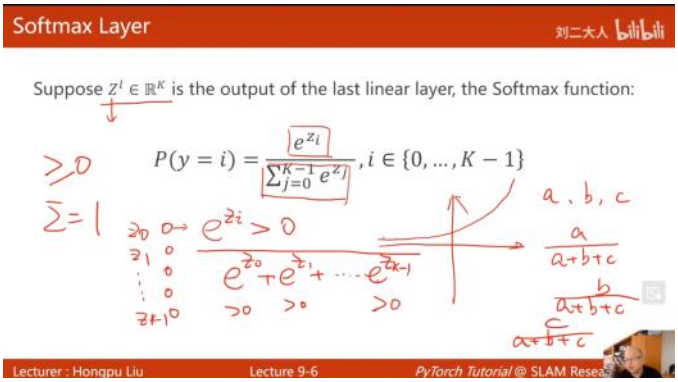
希望输出结果满足离散分布的要求，输出是分布，每项概率≥0，所有概率之和=1



这种分布输出sigmoid就不行，最终的输出必须softmax

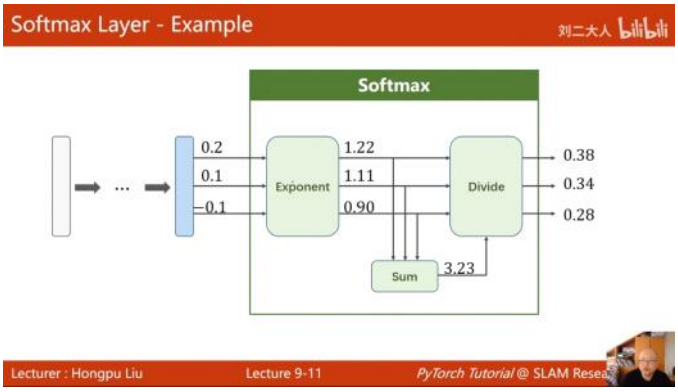


softmax函数：z^l *表示第l层线性输出，*
幂运算能保证大于0，无限接近于0的条件

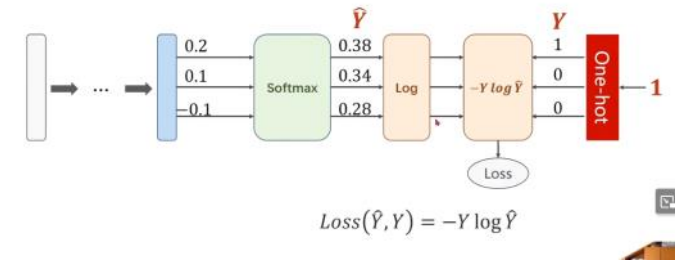


为什么是负数？神经网络计算出的结果有正有负

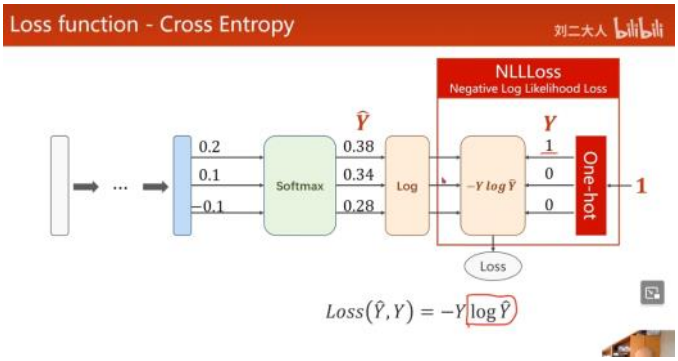
exponent是求e^{z_i}



多分类的交叉熵



直接忽略0项

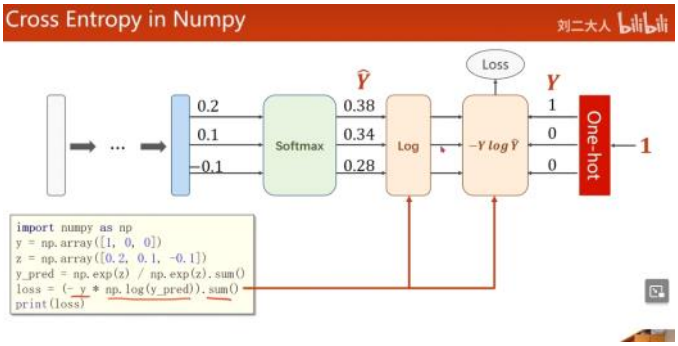


代码实现：

第一种：

1.手写

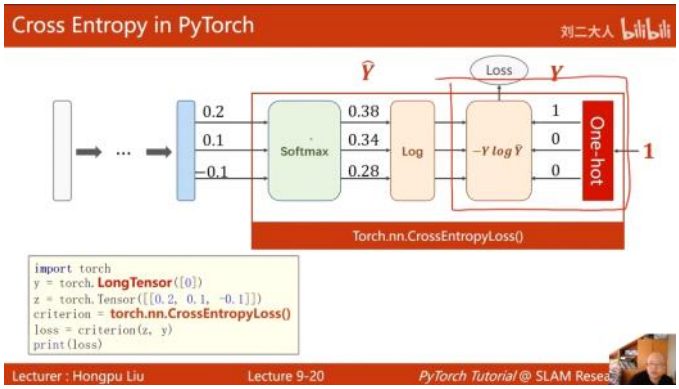
y[1,0,0]是制定第0类是正确的意思



- y长整形张量，选取第0个分类，就是告诉模型这个样本的真实类别是索引0，

这样损失函数就可以基于这个信息来计算预测误差。在多分类问题中，我们通常会有多个可能的类别（例如类别0、1、2等），而y中的整数值就是用来指定样本属于哪个类别的。

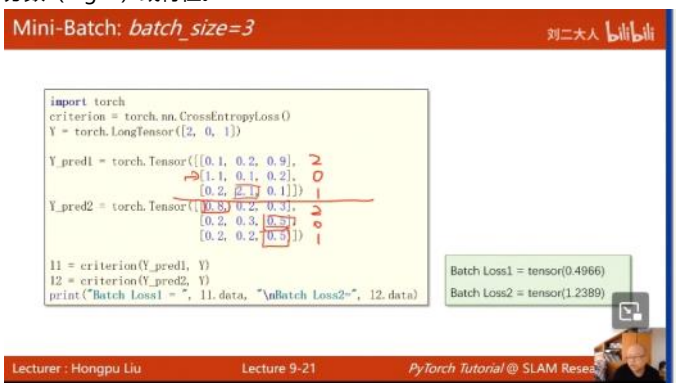
- y_pred会是一个数组，它包含了每个类别的softmax概率值。在这个例子中，y_pred可能会是类似于[0.5761, 0.2689, 0.1550]的数组（实际值取决于z数组的具体元素）。



- 在PyTorch中，LongTensor（或torch.LongTensor）和Tensor（或torch.Tensor，其默认的数据类型是torch.float32）的主要区别在于它们存储的数据类型不同。LongTensor用于存储长整型（通常是64位整数）数据，而Tensor用于存储浮点数。
- 索引表示：在多分类问题中，我们通常使用整数索引来表示每个样本的真实类别。这些索引对应于模型输出（logits）中每个类别的位置。由于整数索引天然就是长整型，所以使用LongTensor来存储这些索引是合适的。
- z=torch.Tensor([[0.2,0.1,-0.1]])是二维张量，形状（shape）是 (1, 3)

  外层的方括号 [] 定义了张量的第一维（行），在这个例子中只有一行。

  内层的方括号 [] 定义了张量的第二维（列在这个），例子中有三列，对应于三个分数（logits）或特征。

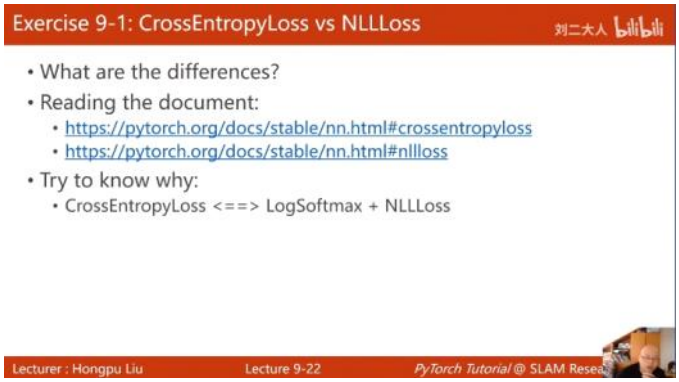


两组预测

2，0，1 分别表示正确答案是第3项最大，第1项最大，第2项最大

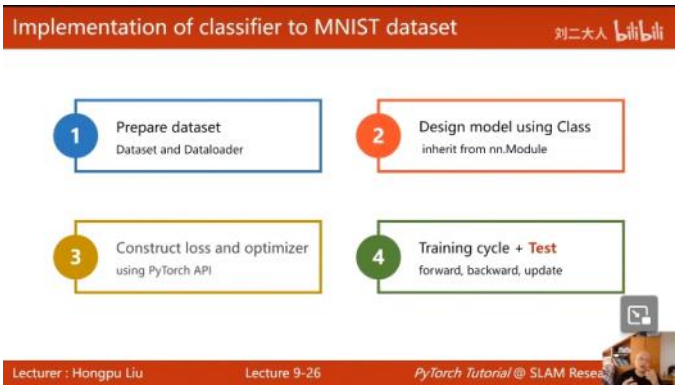
y_pred2准确度不高

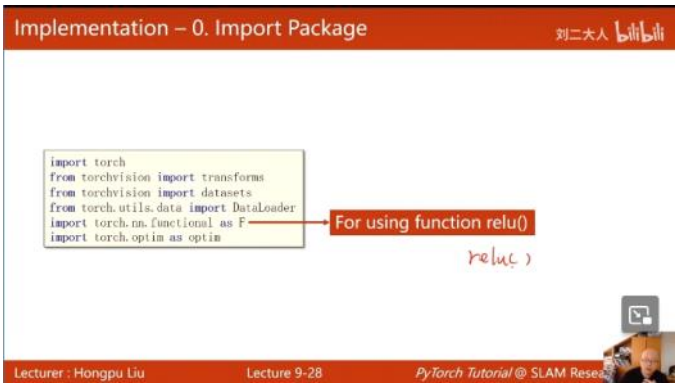
读文档

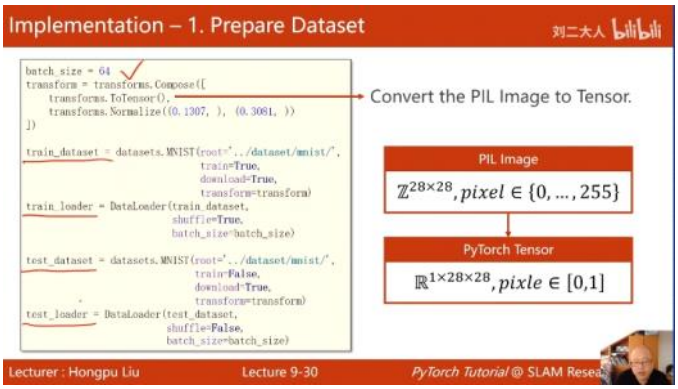


mnist数据集做多分类

每一小格映射到[0，1]



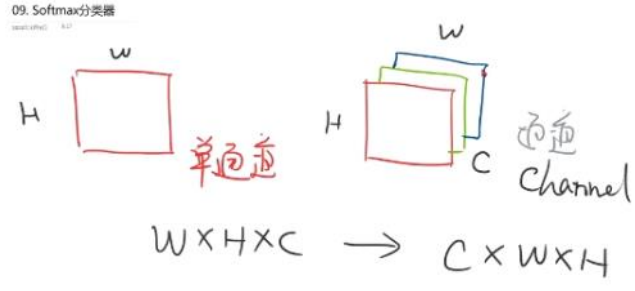


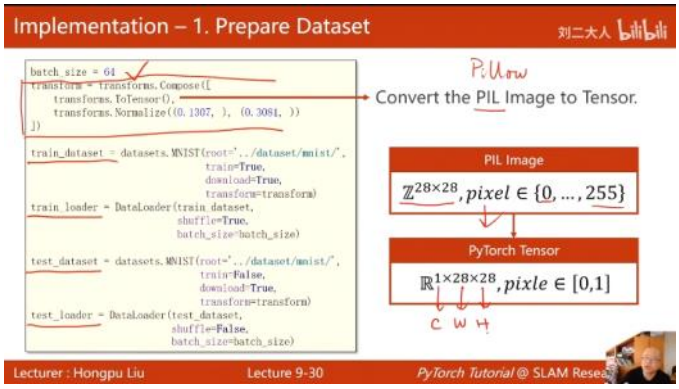


transform对图像做处理，图像变成图像张量

图像的每一小格，灰度是单通道，RGB是3个通道

pytorch把通道放最前面，以更好地进行图像处理

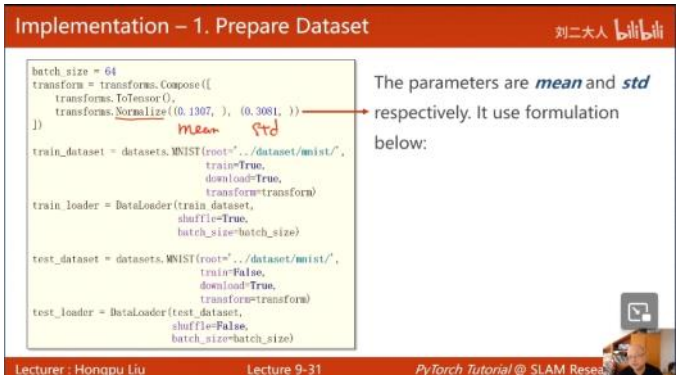


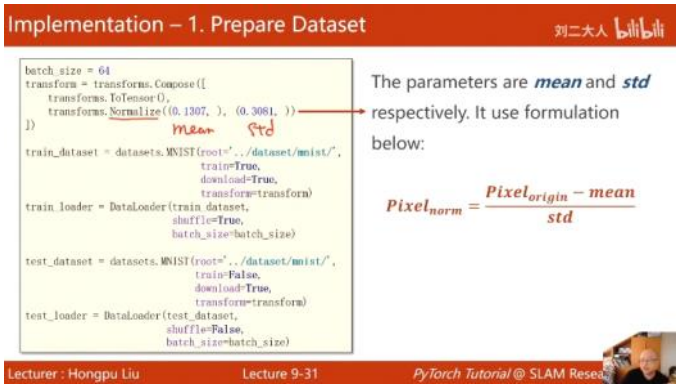


标准化：均值和标准差，mnist是大家已经算好的经验值

我们希望像素值也满足0,1分布

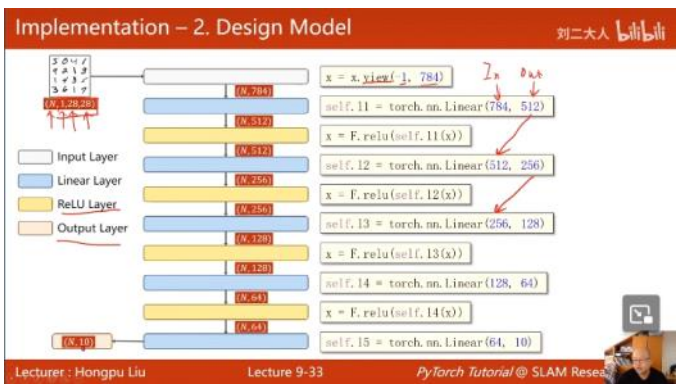
N（0,1）是平均值为0，标准差为1的正态分布





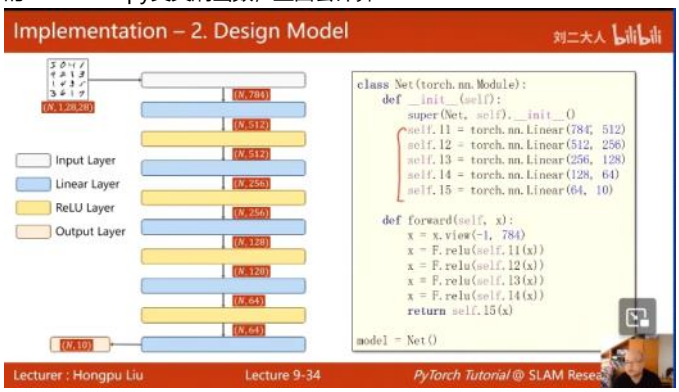
首先需要把（N,1,28,28）变成1阶张量，把图像的每一行拼起来，拼成1行，需要784个元素，view是改变张量的形状，变成2阶张量，即矩阵，有784列，-1是算数值的?

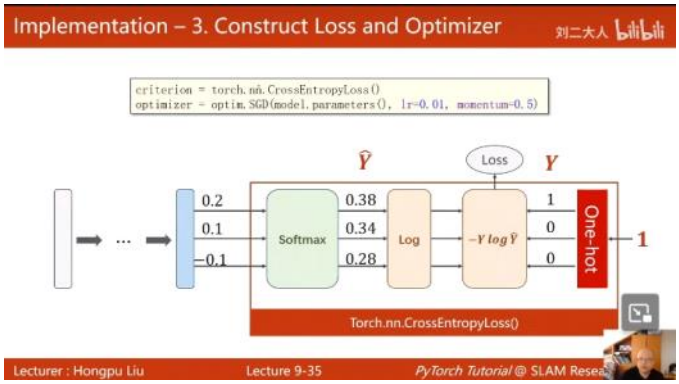
x=x.view(-1, 784)这行代码的作用就是将输入数据x重塑为一个二维张量。通过调用view(-1, 784)，告诉PyTorch希望得到的张量有两维，其中第二维的大小是784（因为MNIST图像的像素总数是28*28=784）。-1是一个占位符，它告诉PyTorch自动计算第一维的大小，以确保总元素数量不变。因此，第一维的大小实际上就是batch_size。



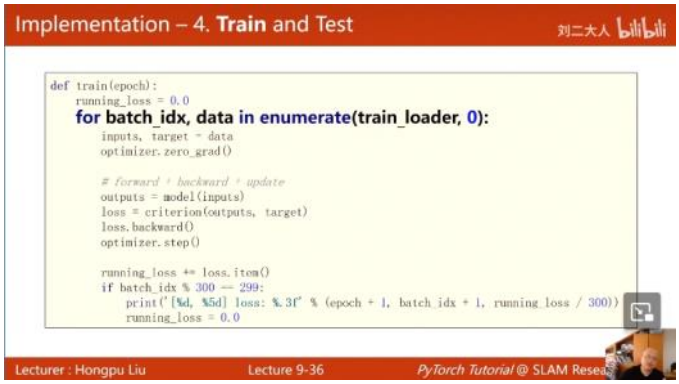
最后一层不做激活，softmax在后面使用的损失函数里面包含了，后面会定义自带

的crossEntropy交叉熵函数，里面会计算

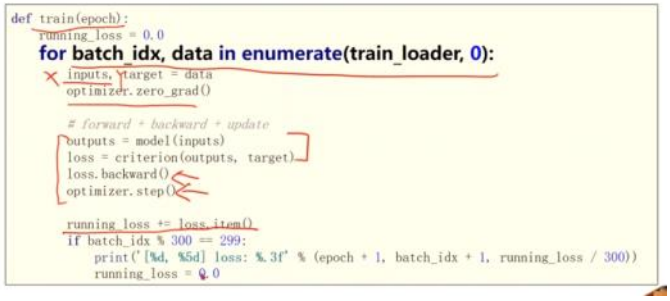




把训练封装到函数里面

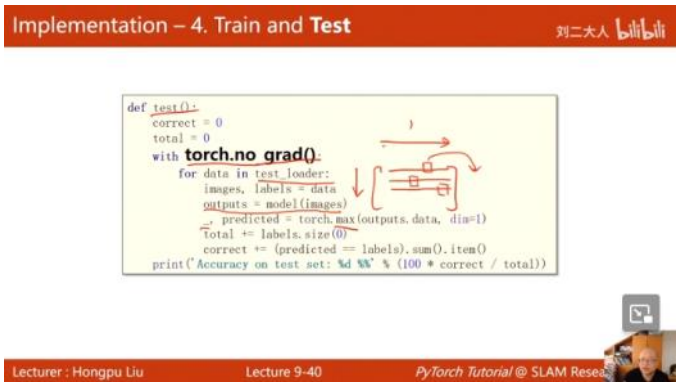


每训练300次输出一次



不用计算梯度

沿着维度1（行）找最大值下标



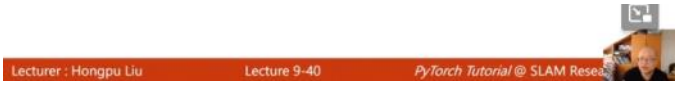
test集里是不需要计算梯度的

这里的 % 是取模运算符，它返回两个数相除的余数。如果当前批次索引是300的倍数（即每300个批次），当 batch_idx 到达300时，batch_idx % 300 的结果是0，这同样不满足 == 299 的条件。但是，当 batch_idx 是300的倍数减一（即299、599、899等）时，batch_idx % 300 的结果正好是299，这时 if batch_idx % 300 == 299 的条件就会成立。

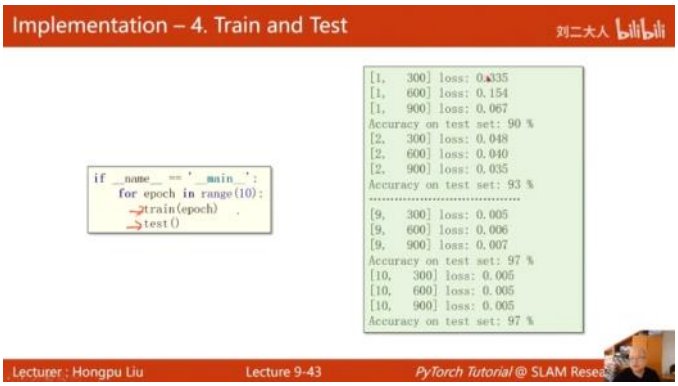
则打印当前epoch和批次索引，以及平均损失

```
    if batch_idx % 300 == 299:
        print('[%d,%5d] loss: %.3f' % (epoch + 1, batch_idx + 1, running_loss / 300))
 # 重置running_loss为0，准备计算下一个300批次周期的平均损失
        running_loss = 0.0
```

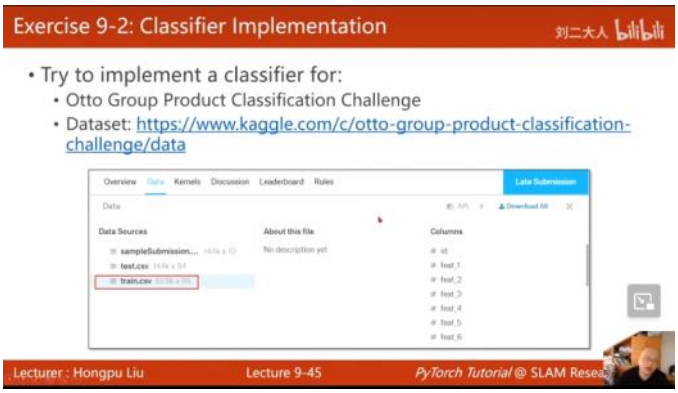
- `with` 关键字用于引入一个上下文管理器，这是 `Python` 的一个特性，允许你访问一个由上下文管理器所管理的对象。在这个特定的例子里，上下文管理器是 `torch.no_grad()`。`torch.no_grad()` 是 `PyTorch` 中的一个上下文管理器，用于临时禁用在代码块内部的所有计算图和梯度计算。这在模型推理（即模型评估或测试）阶段非常有用，因为在推理过程中不需要计算梯度。
- dim=1是dimension维度，第二个维度，dim=0 通常表示批次维度（即沿着所有图像的维度），而 dim=1 则表示每个图像内部的维度（即沿着每个图像的像素或特征的维度），即索引；使用 torch.max 函数找到输出 outputs 中每一行最大值的索引。这个索引即为模型预测的类别。torch.max返回两个值，一个是最大值，一个是最大值的索引。
- _ 是一个占位符，表示不打算使用第一个返回值（即最大值本身），而只关心第二个返回值（即预测的类别索引）。
- labels.size(0)：labels是包含真实标签的张量，size(0)返回这个张量在第一维度（通常是批次维度）的大小，即当前批次中的样本数。



test集里是不需要计算梯度的



练习：



- labels.size(0)：labels是包含真实标签的张量，size(0)返回这个张量在第一维度（通常是批次维度）的大小，即当前批次中的样本数。
- total +=：将当前批次的样本数累加到total变量中，最终得到测试集的总样本数。
- correct += (predicted == labels).sum().item()

torchvision

2024年4月28日 10:36

torchvision是pytorch的一个图形库，它服务于PyTorch深度学习框架的，主要用来构建计算机视觉模型。以下是torchvision的构成：

1. torchvision.datasets: 一些加载数据的函数及常用的数据集接口；
2. torchvision.models: 包含常用的模型结构（含预训练模型），例如AlexNet、VGG、ResNet等；
3. torchvision.transforms: 常用的图片变换，例如裁剪、旋转等；
4. torchvision.utils: 其他的一些有用的方法。

torchvision.transforms

torchvision.transforms.Compose()类。这个类的主要作用是串联多个图片变换的操作。

```
# 图像预处理步骤
transform = transforms.Compose([
    transforms.Resize(96), # 缩放到 96 * 96 大小
    transforms.ToTensor(), # 转化为Tensor
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # 归一化
])
```

transform.Normalize

2024年4月26日 16:36

transform.Normalize通常用于对数据进行标准化处理，使得数据的分布更加规范化，从而有助于模型的训练和收敛。标准化处理通常包括两个步骤：归一化和标准化。

1. 归一化（Normalization）：将数据缩放到一个小的、指定的范围，比如[0, 1]。
2. 标准化（Standardization）：将数据的分布转换为具有特定均值和标准差的形式，比如均值为0，标准差为1的正态分布。

在PyTorch的transform.Normalize中，你需要提供两个参数：mean和std。这两个参数通常是数据集中每个通道的均值和标准差。对于MNIST数据集，这些值是预先计算好的，并且经常在文献和实践中被使用。

对于MNIST数据集，0.1307和0.3081这两个数值是图像数据在灰度空间中每个像素值的均值和标准差。具体来说：

- 0.1307 是数据集图像的均值。MNIST数据集中的手写数字图像是灰度图像，每个像素的值范围从0到255。通过对所有图像的所有像素值求平均，可以得到整个数据集中所有像素值的平均灰度。
- 0.3081 是数据集图像的标准差。它衡量了数据集中像素值分布的离散程度。

这些统计数据通常是通过预先对整个MNIST数据集进行分析得到的。在实际应用中，使用这些预定义的均值和标准差可以帮助模型学习到更加鲁棒的特征。
对于 MNIST 数据集，由于图像是单通道（灰度图像），所以只需要一个均值和一个标准差。即使只有一个通道，也必须将它们放在元组中，因为 transforms.Normalize 方法的 API 设计要求使用元组来接受多个通道的均值和标准差。
这里的 (0.1307,) 和 (0.3081,) 都是单个元素的元组，逗号 , 是用来创建元组的。在 Python 中，即使元组只有一个元素，你也需要在元素后面加上逗号来表示它是一个元组，而不是一个括号表达式。
当然也可以这么写：transforms.Normalize(mean=[0.1307], std=[0.3081])

如果处理的是三通道的彩色图像，比如来自 CIFAR-10 或 ImageNet 数据集的图像，那么均值和标准差会是三个通道的均值和标准差，代码看起来会像这样：

transform.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
或者
transform.Normalize（(0.485, 0.456, 0.406)，（0.229, 0.224, 0.225））

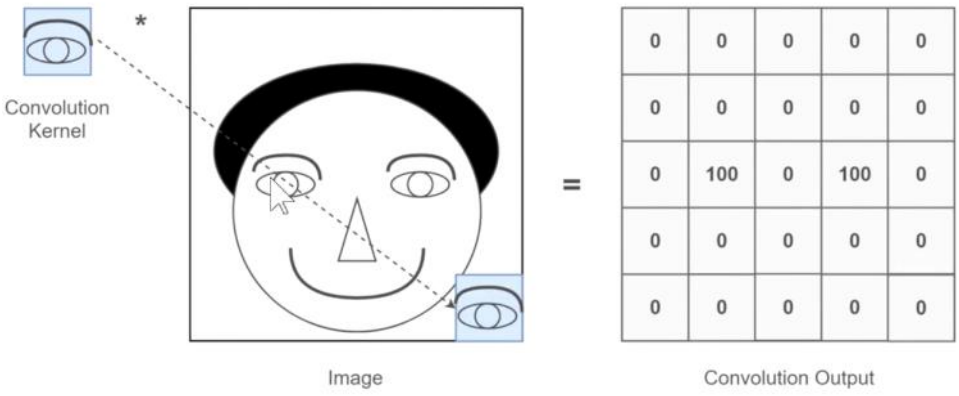
在这个例子中，mean 是一个包含三个元素的列表，表示每个颜色通道（红、绿、蓝）的均值；std 也是一个包含三个元素的列表，表示每个颜色通道的标准差。这样的设置允许对每个通道分别进行标准化处理，这在处理彩色图像时是很常见的做法。

卷积基础

2024年5月8日 10:23

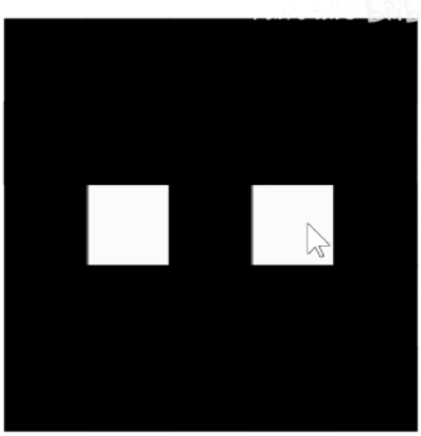
卷积

2024年5月8日 9:02

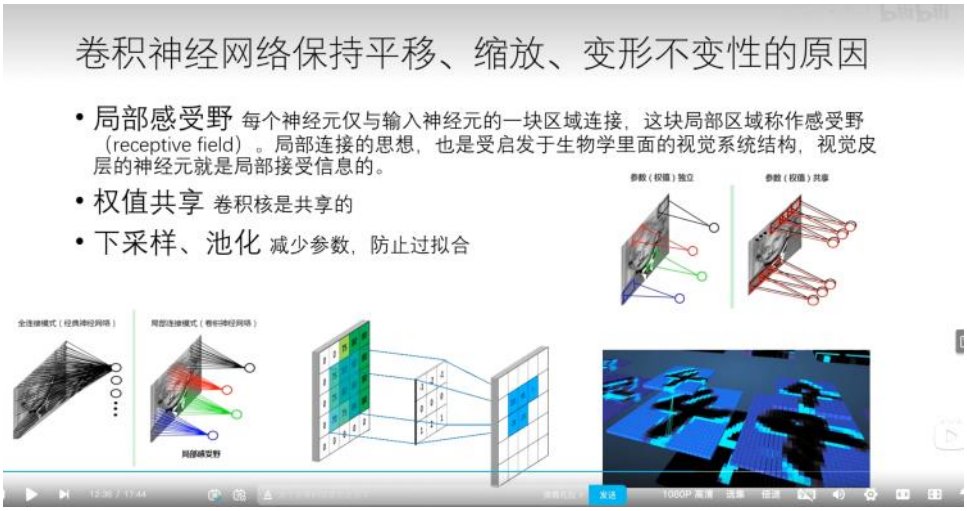


0	0	0	0	0
0	0	0	0	0
0	100	0	100	0
0	0	0	0	0
0	0	0	0	0

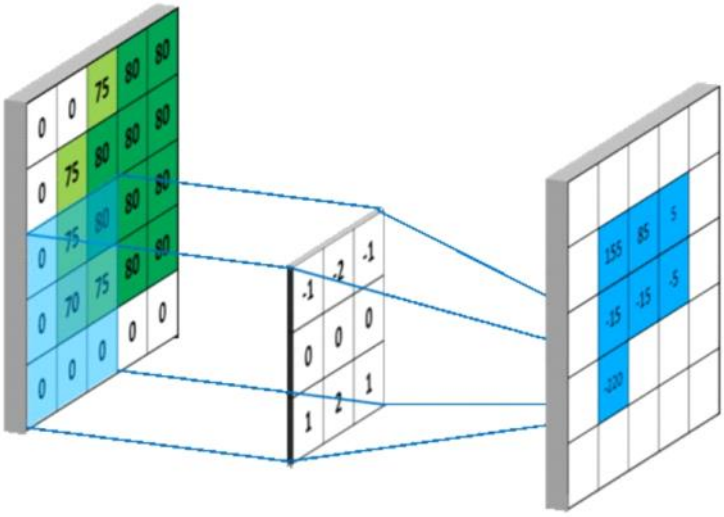
Convolution Output



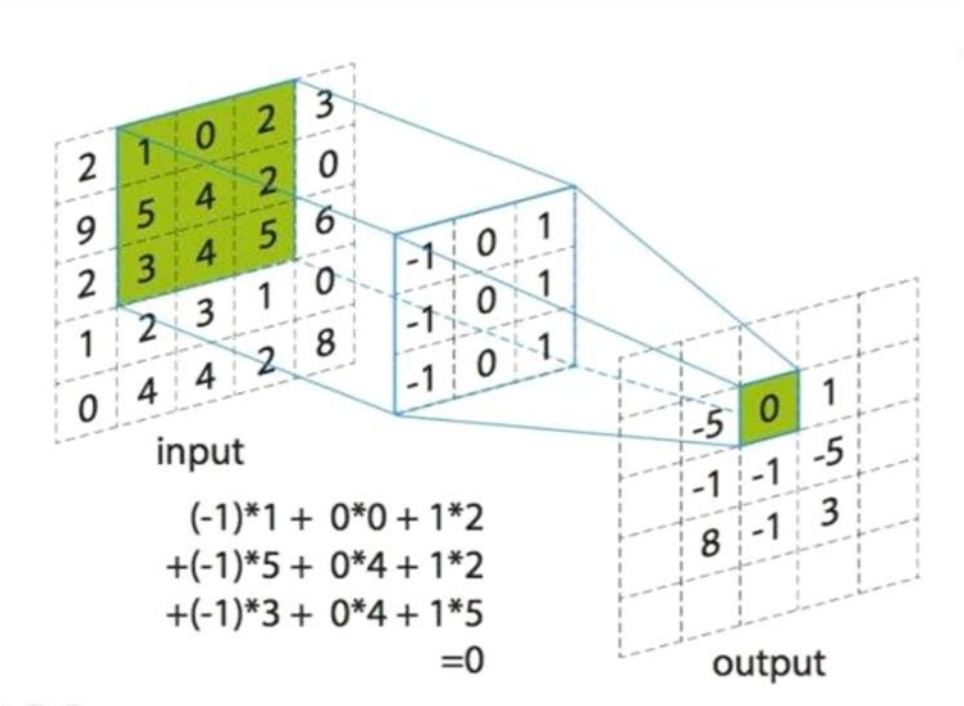
Convolution Output (as an Image)



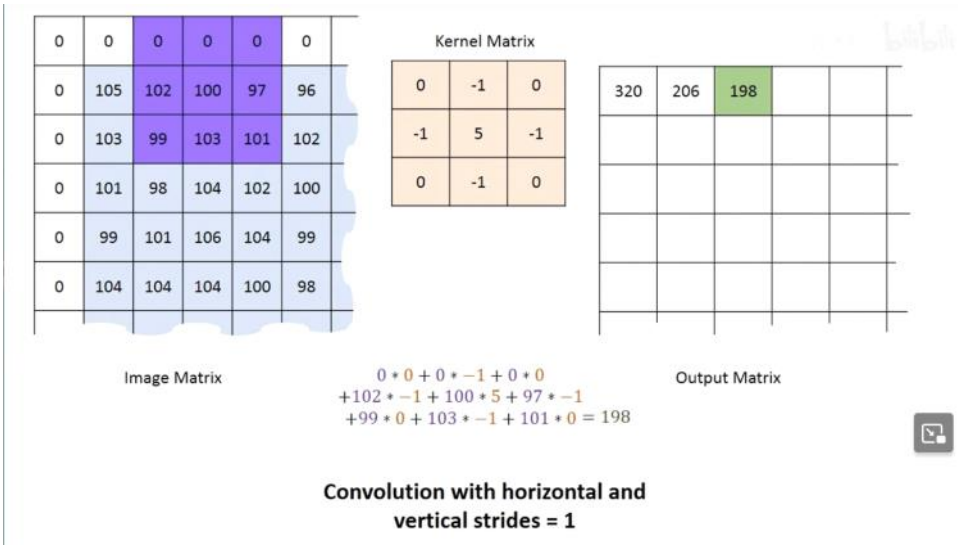
https://www.bilibili.com/video/BV1AJ411Q72b/?spm_id_from=333.1007.top_right_bar_window_custom_collection.content.click&vd_source=9b01f3d1e6addb97637b80b1bb9c008b

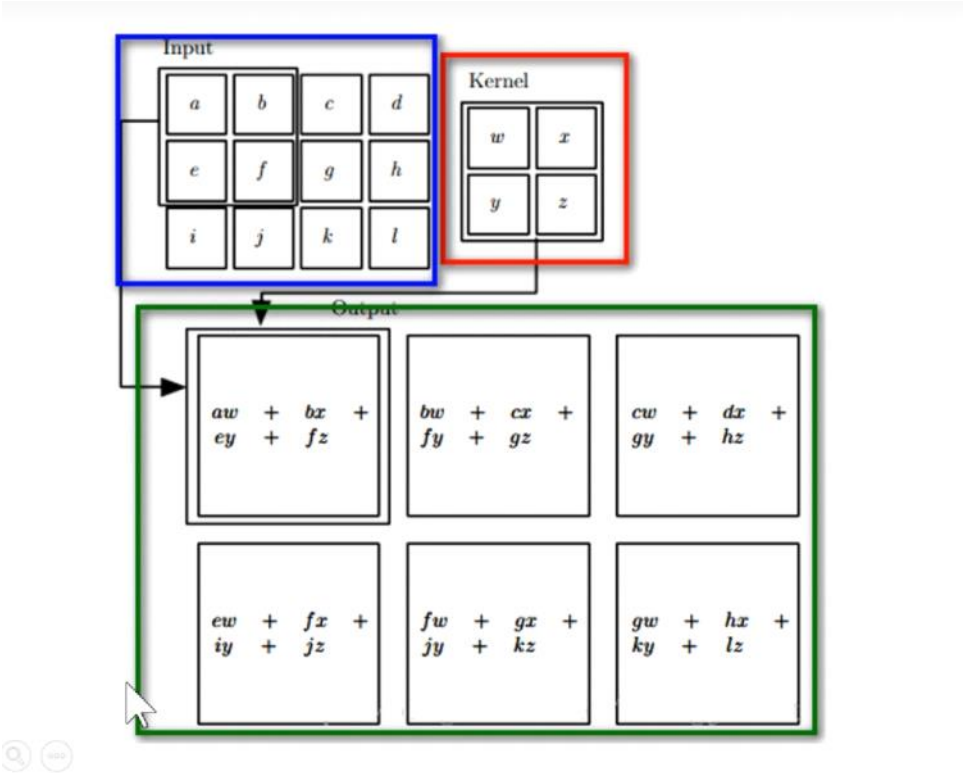


感受野：能看到的视野



padding补零：让周围有用的数字更多次地被扫描

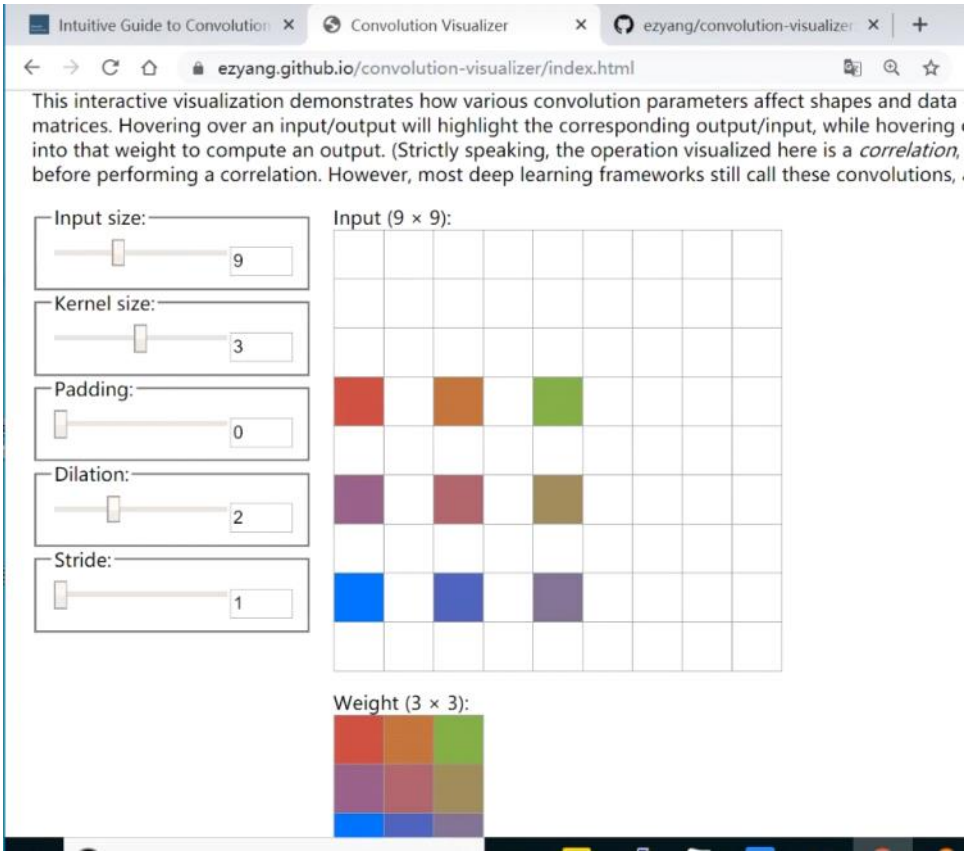




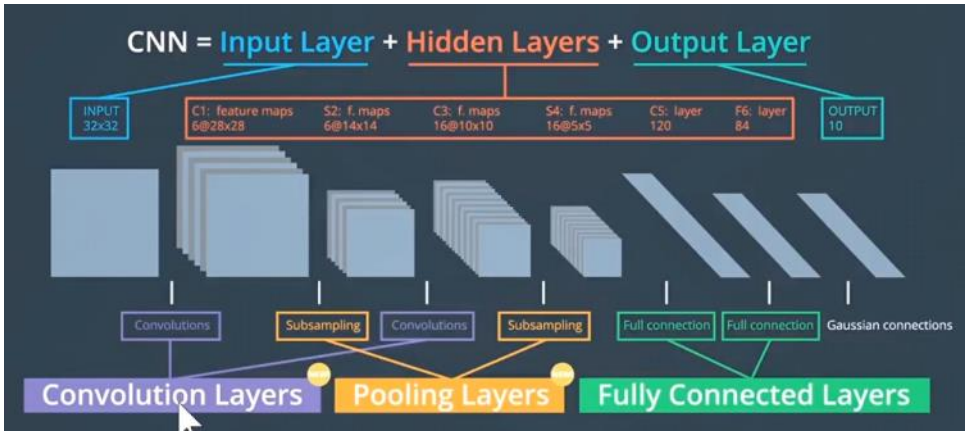
输出的图就叫特征图feature map

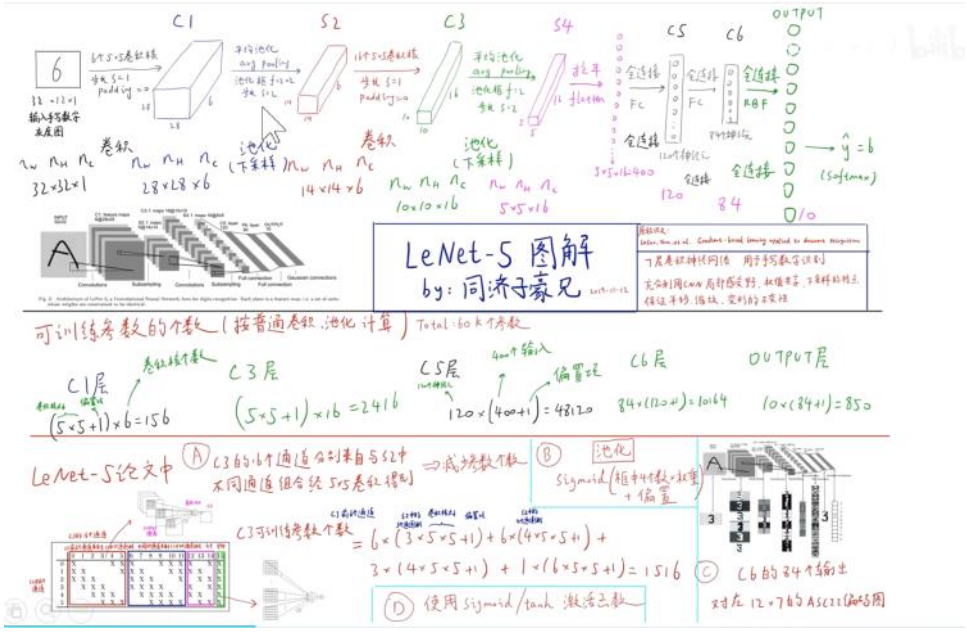
有多少个卷积核就有多少个feature map

单通道演示



dilation=2空洞卷积

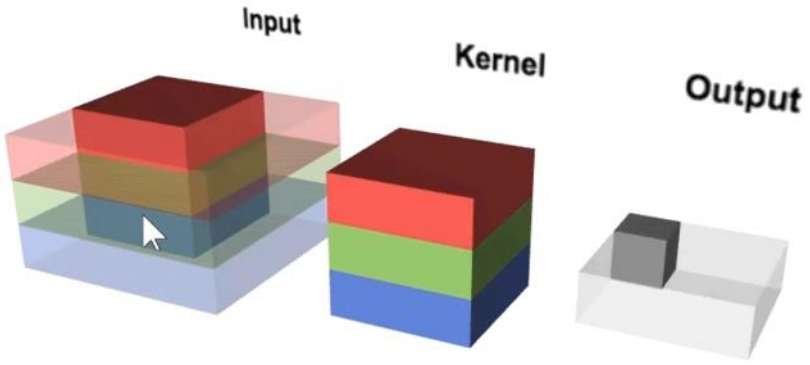

CNN = Input Layer + Hidden Layers + Output Layer
INPUT 32x32
C1: feature maps 6@28x28
S2: f. maps 6@14x14
C3: f. maps 16@10x10
S4: f. maps 16@5x5
C5: layer 120
F6: layer 84
OUTPUT 10
Convolutions
Subsampling
Convolutions
Subsampling
Full connection
Full connection
Gaussian connections
Convolution Layers
Pooling Layers
Fully Connected Layers

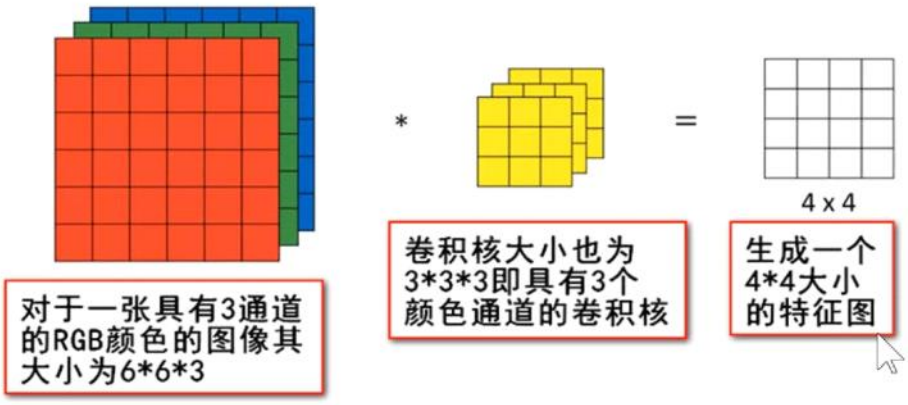

LeNet-5 图解
by: 同济子豪兄
可训练参数的个数（按普通卷积、池化 计算）
C1层
C3层
C5层
C6层
OUTPUT层
LeNet-5论文中

RGB卷积

2024年5月8日 9:24

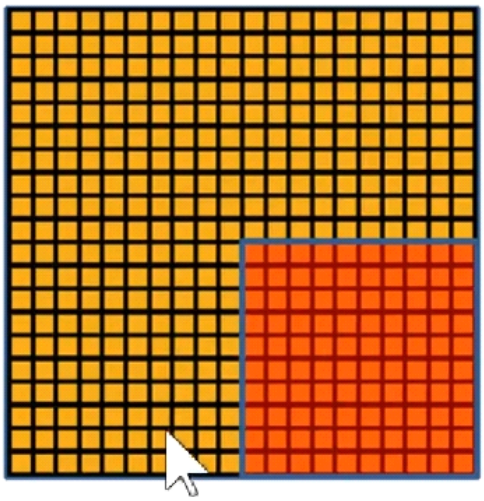
https://thomelane.github.io/convolutions/2DConvRGB.html





池化层和全链接层

2024年5月8日 10:23

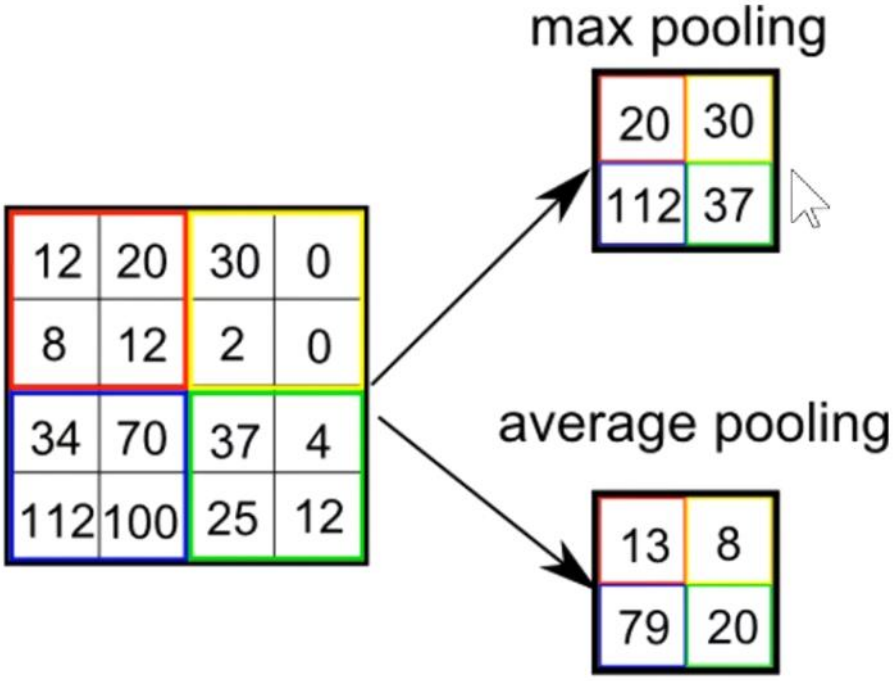


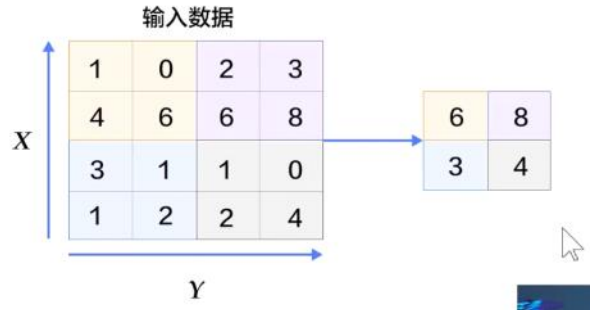
1	7
5	9

Convolved feature

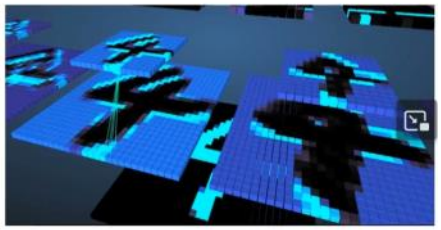
Pooled feature

池化又叫下采样 大框里选一个数来代表，两种方法：最大池化和平均池化

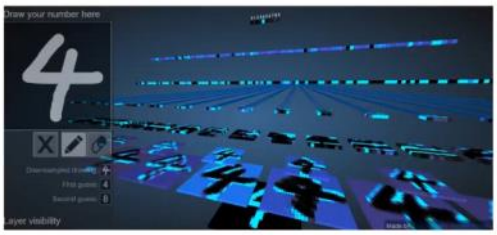


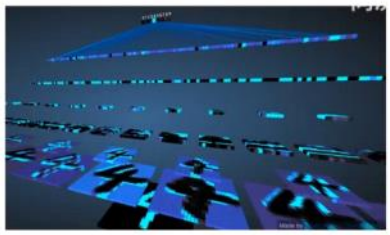


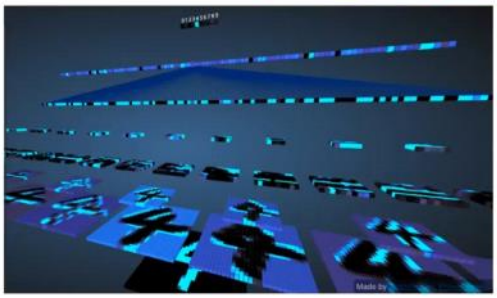
池化的作用
1.减少参数量
2.防止过拟合

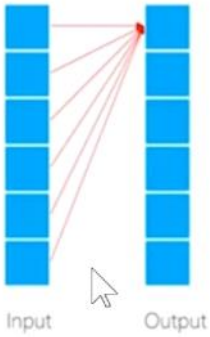


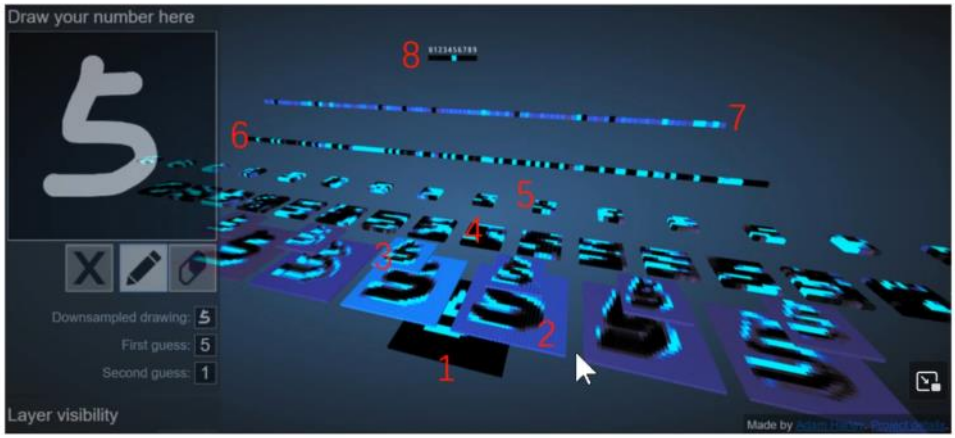
全连接层：将池化的所有结果拉成一个长向量
输出到最后的softmax层











1、把手写字体图片转换成像素矩阵

2、对像素矩阵进行第一层卷积运算，生成六个feature map

3、对每个feature map进行下采样（也叫做池化），在保留feature map特征的同时缩小数据量。生成六个小图，这六个小图和上一层各自的feature map长得很像，但尺寸缩小了。

4、对六个小图进行第二层卷积运算，生成更多feature map

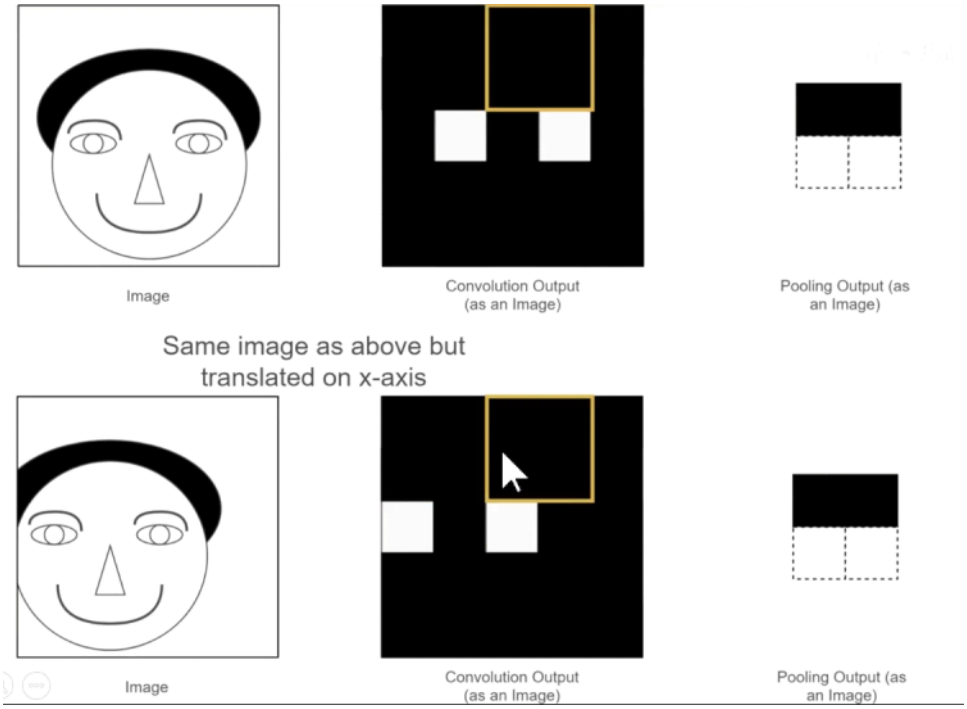
5、对第二次卷积生成的feature map进行下采样

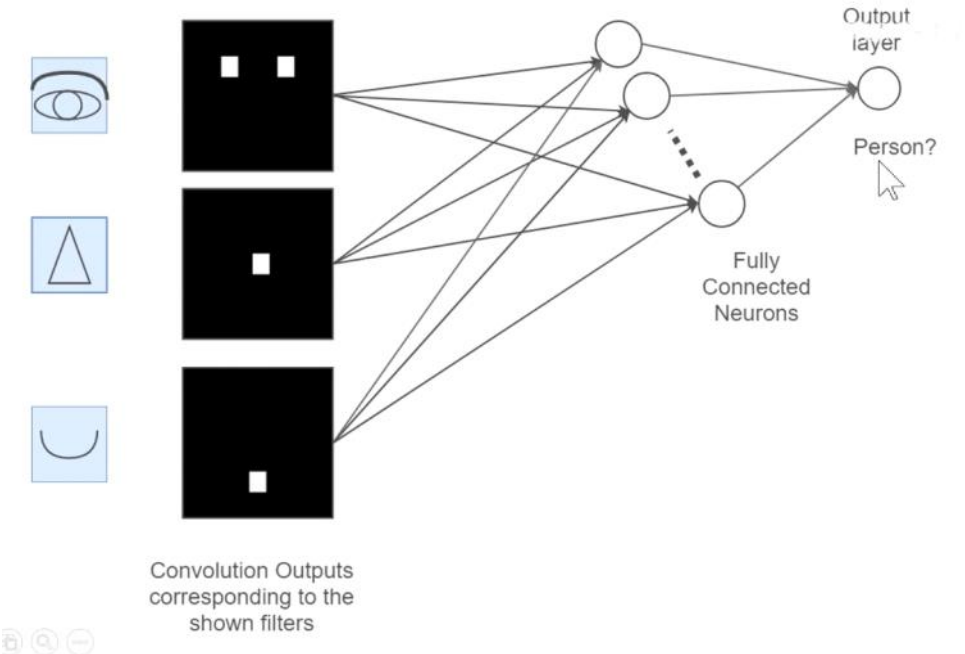
6、第一层全连接层

7、第二层全连接层

8、高斯连接层，输出分类结果

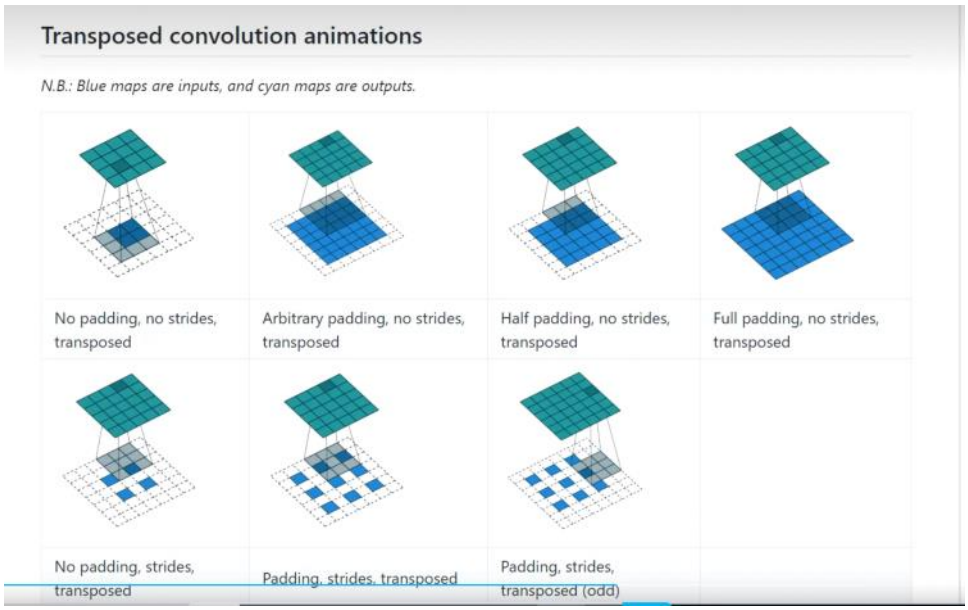
池化的平移不变性

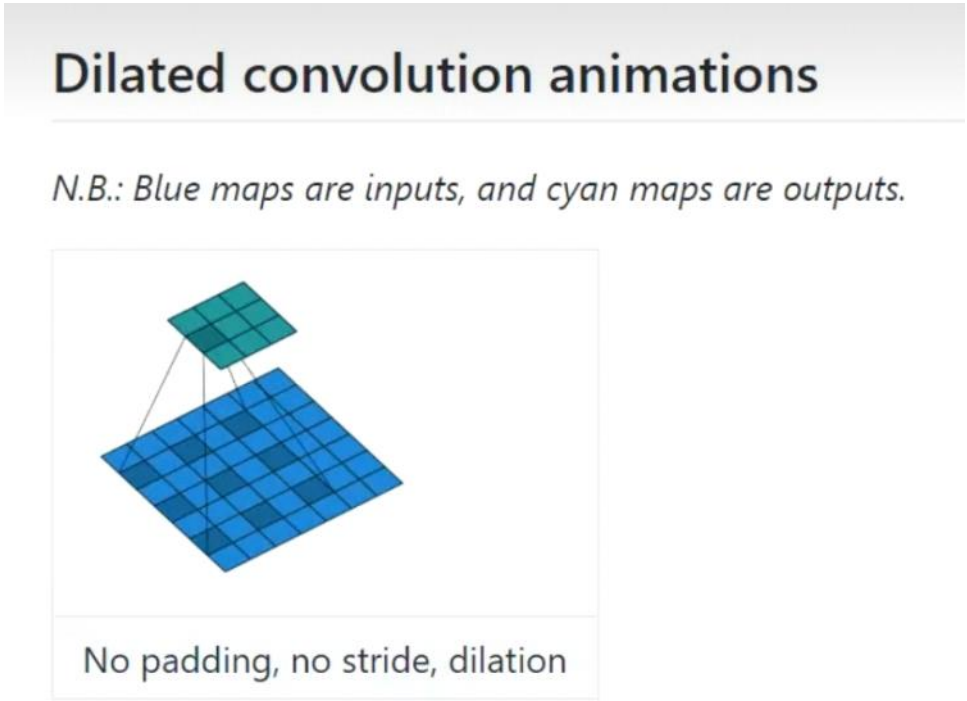




所有特征到全连接层 全连接层得到结果

上采样 转置卷积 反卷积 小的变成大的 升维

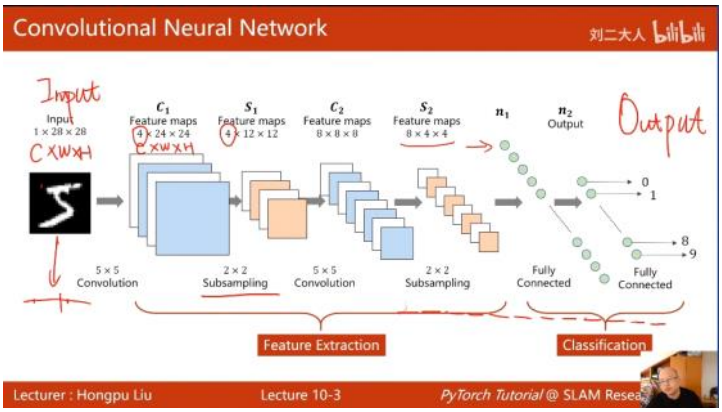




棋盘卷积

卷积神经网络（基础篇）

2024年4月28日　15:13



卷积：把图像数据按空间结构保存，保留空间特征

下采样：减少元素数量，降低运算需求

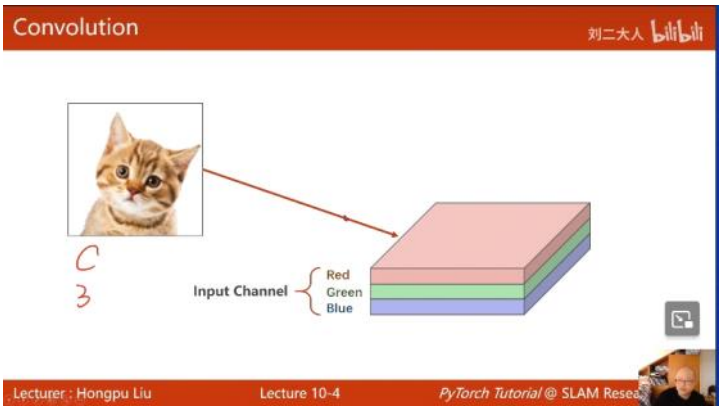
最终目标：不断卷积，做维度和大小的变化，输出10维向量

feature extraction:特征提取器，通过卷积找到特征

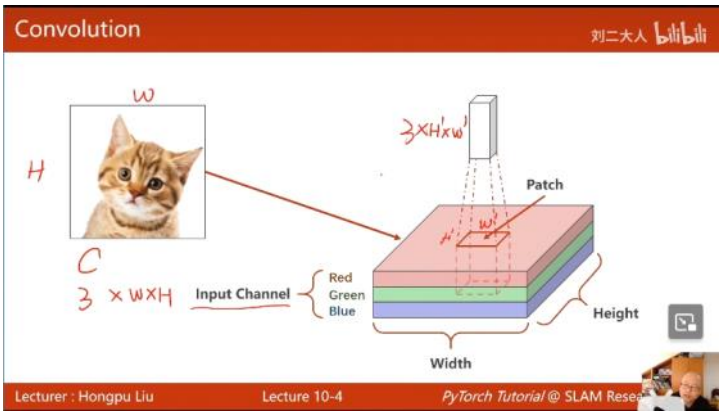
classification:分类器

需要思考的是：输入维度多少？输出维度多少？

RGB图像

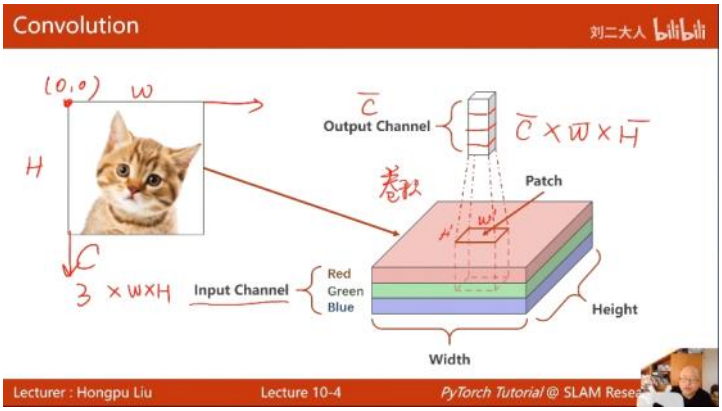


开始取一个图像块（取张量）



把图像块做卷积

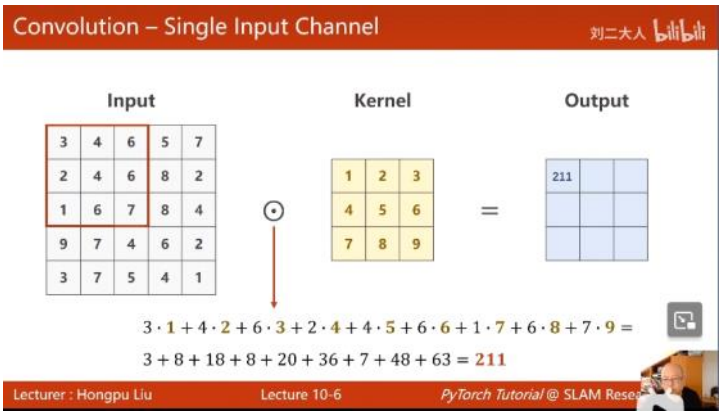
这个块遍历一遍，得到卷积结果



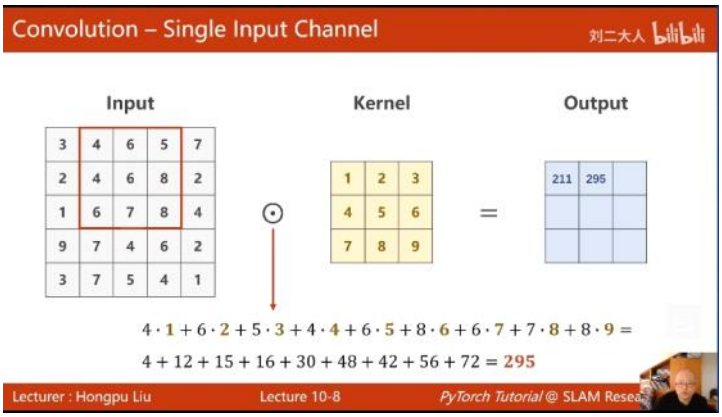
单通道的卷积过程：

卷积核和输入图像做数乘，得到输出结构第一格

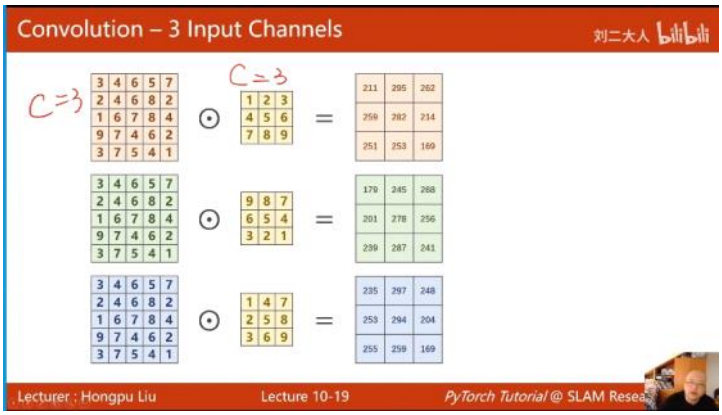
output.size=3 是5-3+1

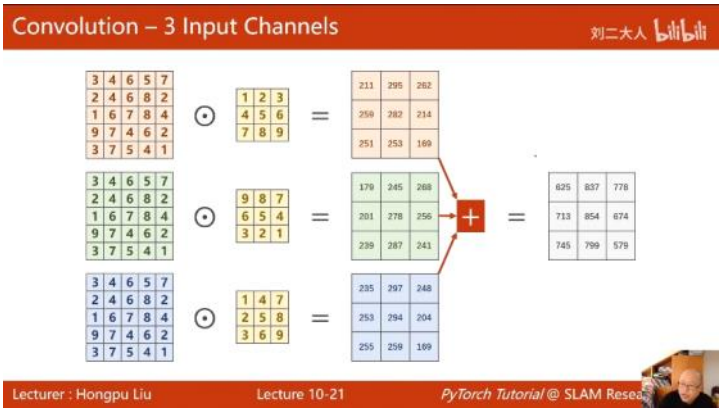


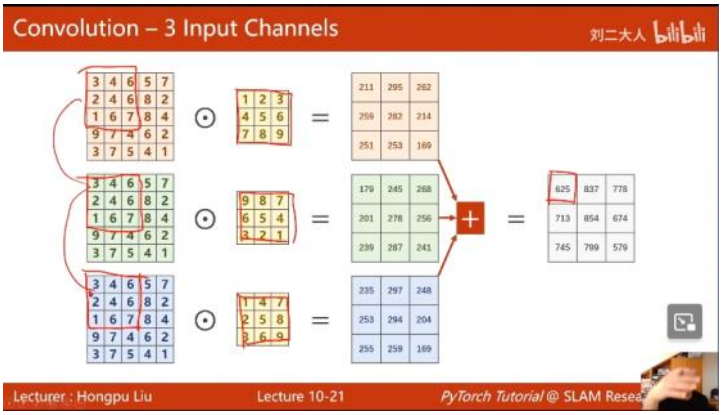
开始遍历



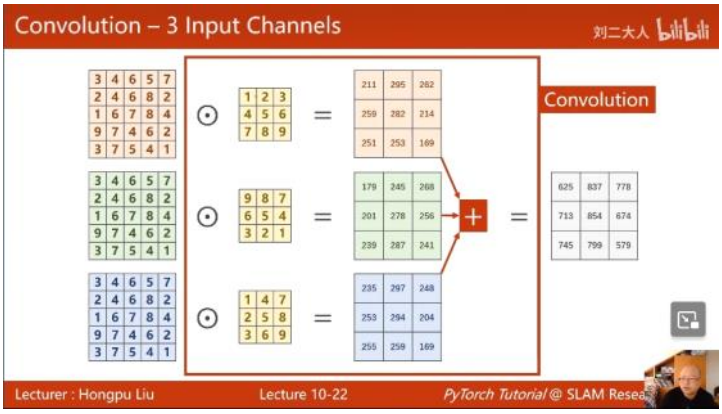
三通道的卷积过程：

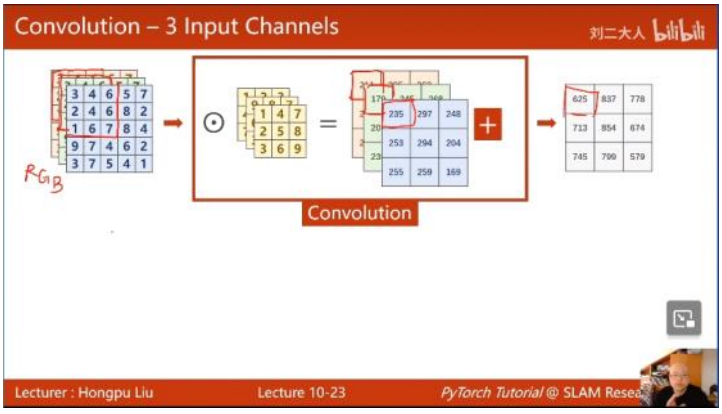


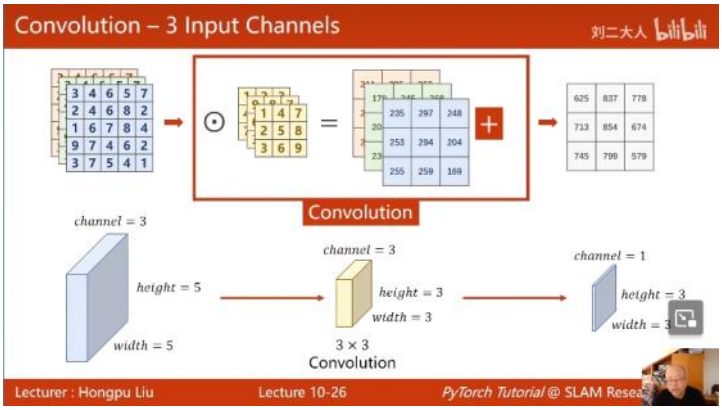

Convolution – 3 Input Channels
刘二大人 bilibili
Lecturer : Hongpu Liu
Lecture 10-21
PyTorch Tutorial @ SLAM Resea


Convolution – 3 Input Channels
刘二大人 bilibili
Lecturer : Hongpu Liu
Lecture 10-21
PyTorch Tutorial @ SLAM Resea

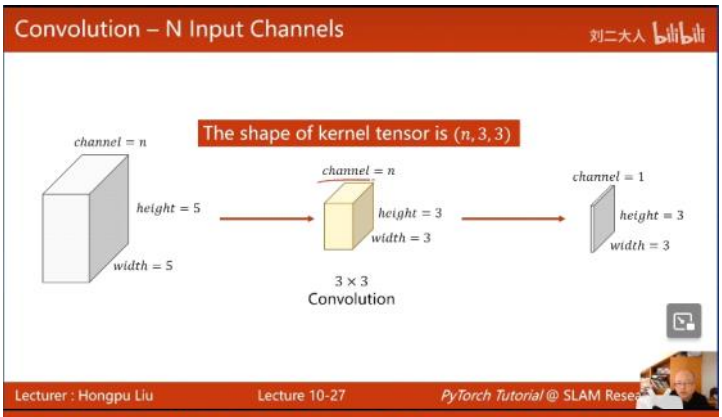
卷积运算

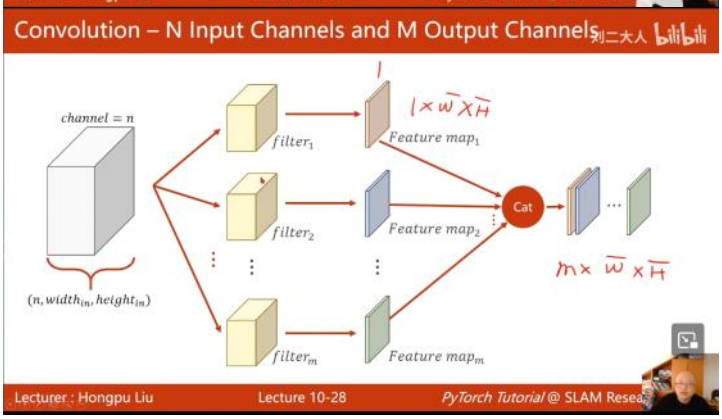

Convolution – 3 Input Channels
刘二大人 bilibili
Convolution
Lecturer : Hongpu Liu
Lecture 10-22
PyTorch Tutorial @ SLAM Resea


Convolution – 3 Input Channels
刘二大人 bilibili
RGB
Convolution
Lecturer : Hongpu Liu
Lecture 10-23
PyTorch Tutorial @ SLAM Resea



这样就得到一个通道





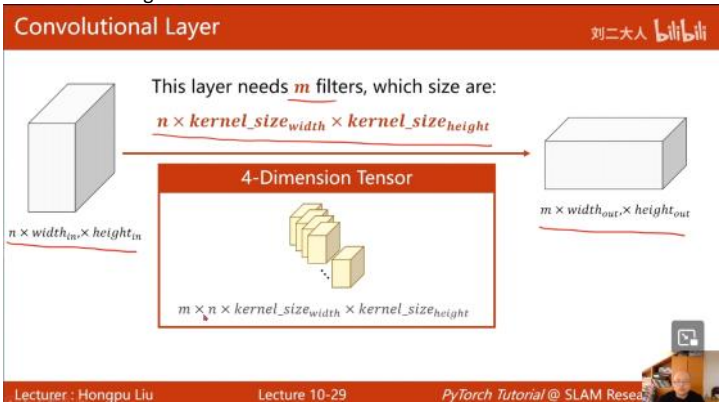
当我们拿到一个数据

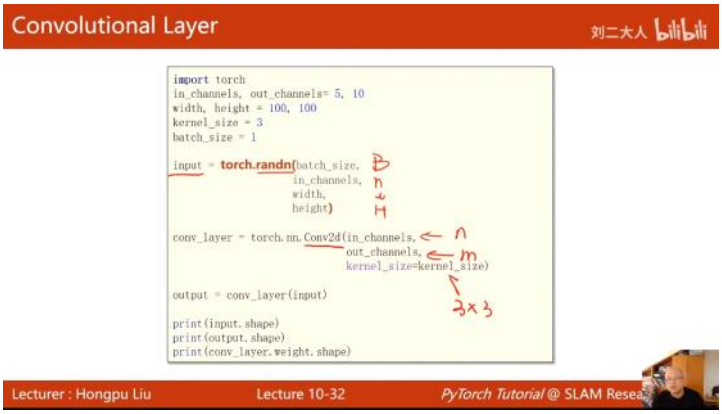
输入通道为n 输出通道为m

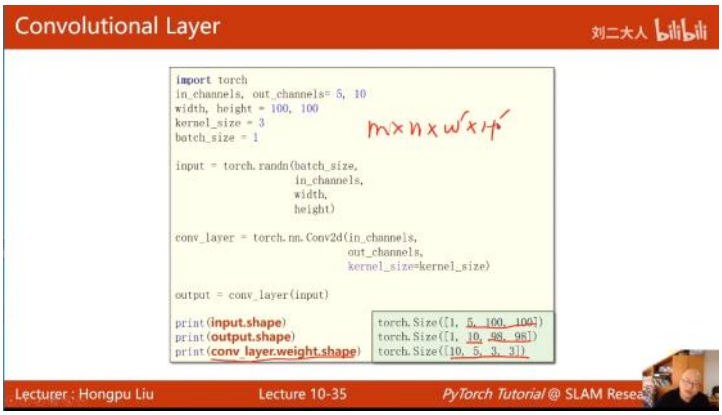
- 每一个卷积核的通道数与输入特征图的通道数一致，而卷积核的总数决定了输出特征图的通道数。
- **假设输入特征图有n个通道，那么每个卷积核也会有n个通道，这样才能与输入特征图的每个通道进行对应的卷积运算。而如果有m个卷积核，那么输出特征图就会有m个通道，每个通道都是由对应的卷积核与输入特征图进行卷积运算得到的。**

所以我们需要卷积层的维度是

m*n*width*height

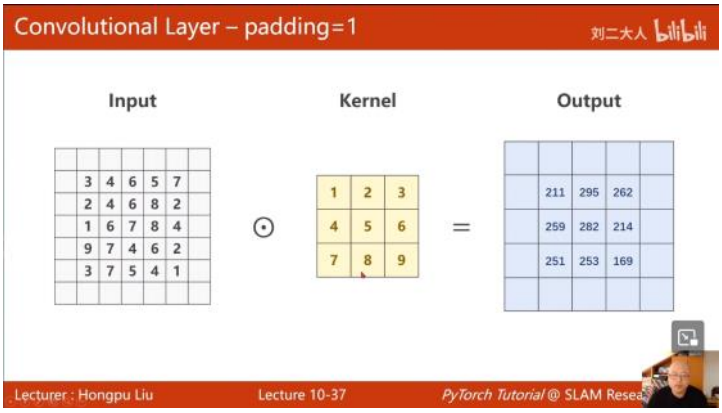




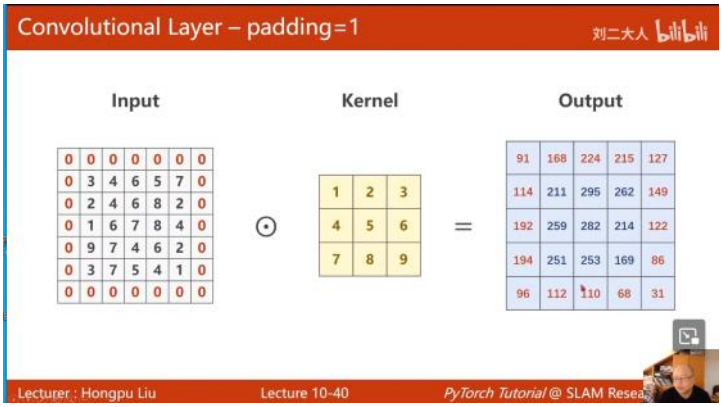


定义一个卷积层：

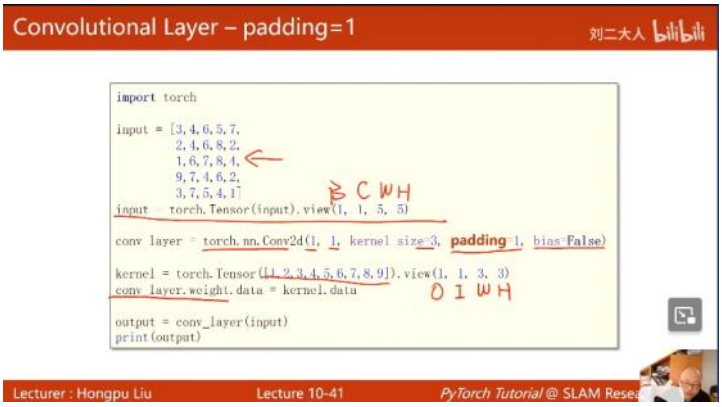
1. **卷积核的通道数必须和输入通道数一样**
2. **输出通道数决定卷积核总数**
3. **卷积核自身的长宽是给定的**



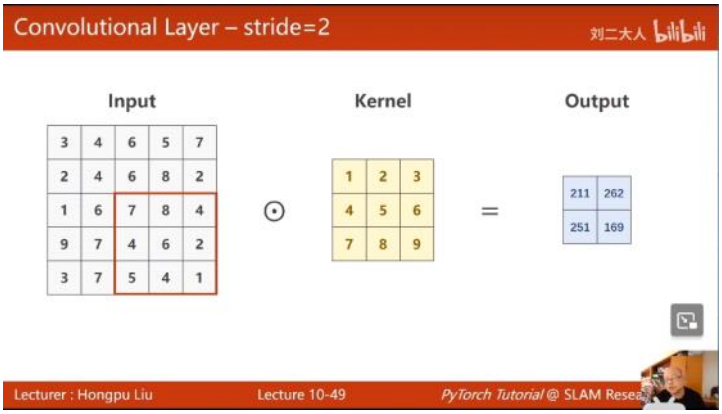
padding最常见的是填充0

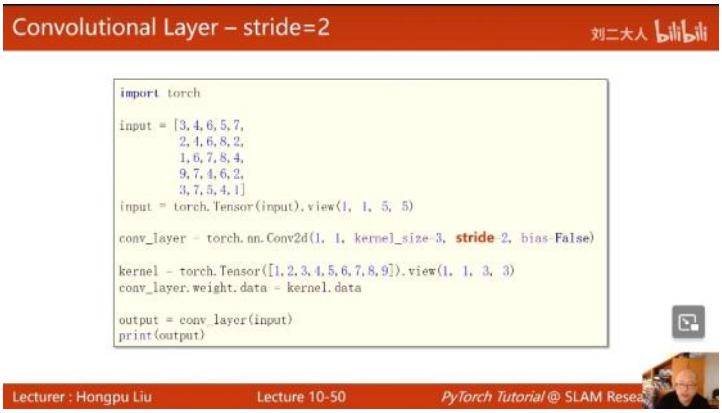


代码实现



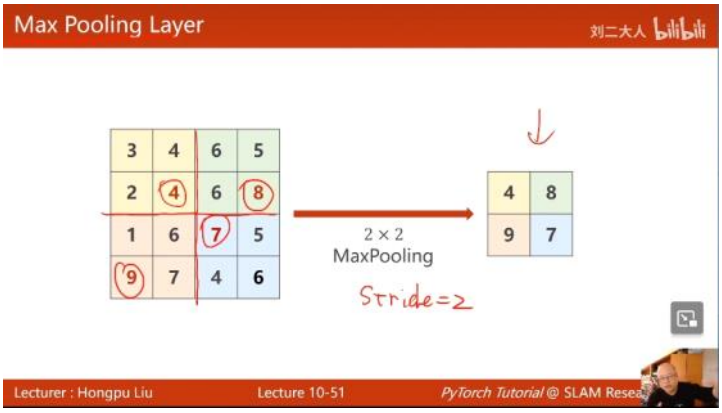
还有一种卷积 stride步长=2 中心点一次移2格

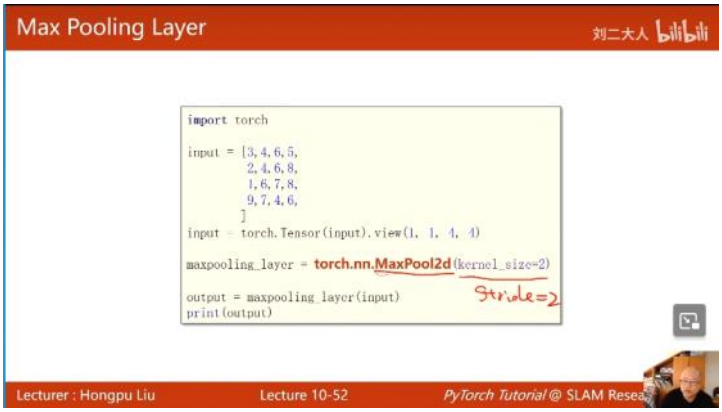


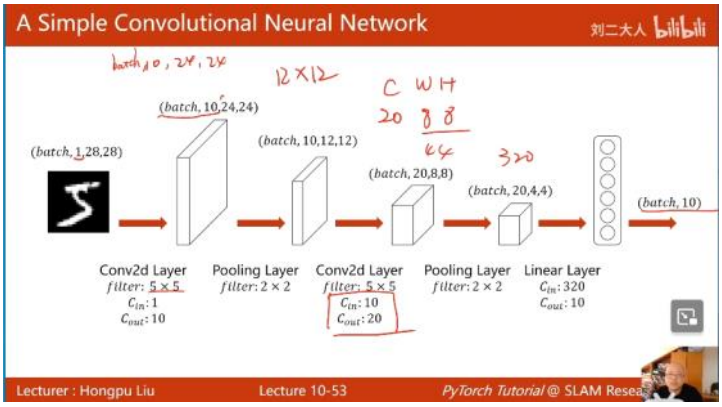


下采样——最大池化层Maxpooling Layer

默认stride=2,每一组找最大值

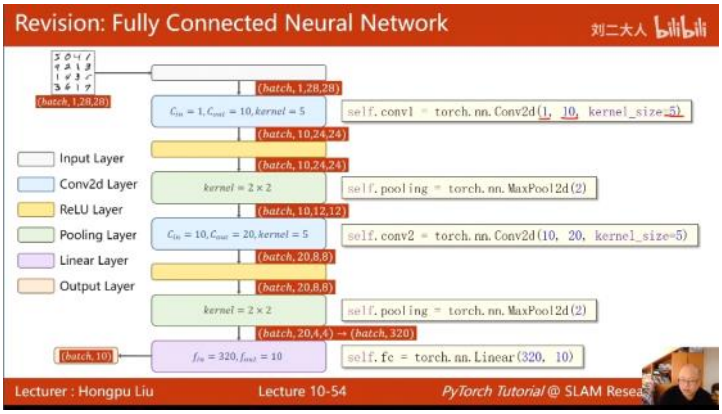


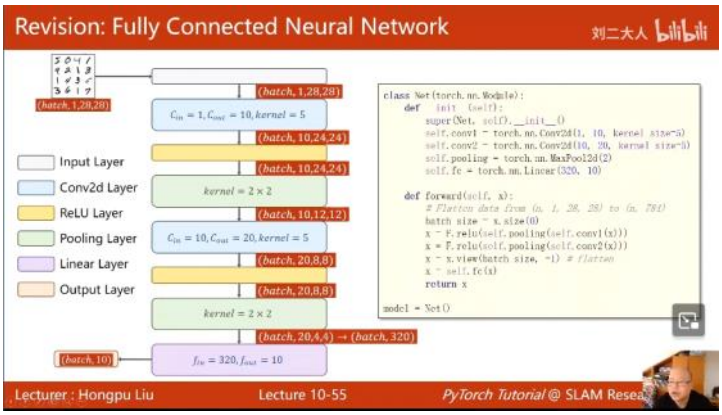




把之前的全链接网络改成卷积:

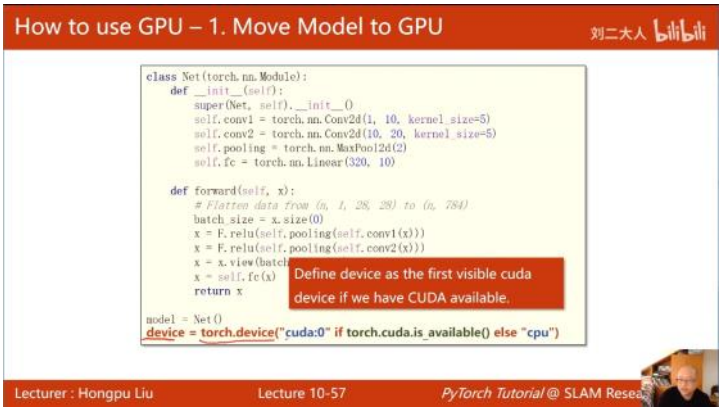
精髓在于设计维度要对上

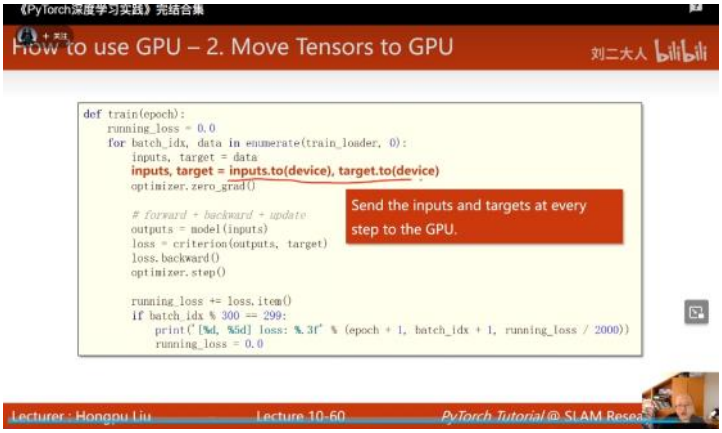


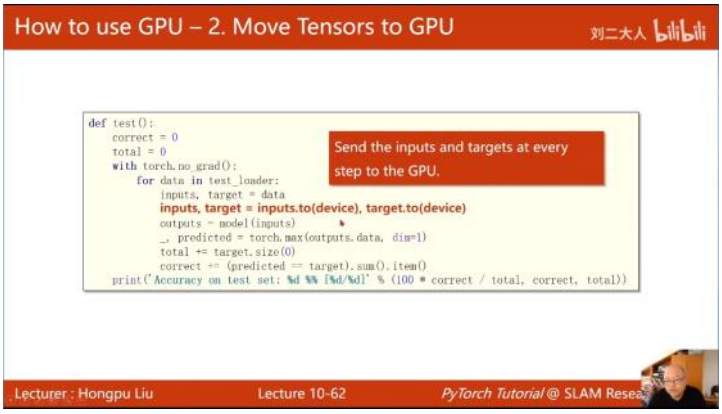


怎么用GPU跑

- #x.view(batch_size, -1)：将 x 重塑为一个二维张量，-1 表示 PyTorch 将自动计算第二个维度的大小，其第一维是批次大小，第二维是所有其他元素的平坦视图。这样做的目的是将多维的卷积层输出转换为一个维度，以便可以输入到全连接层。
- 举个例子，假设 x 的形状是 (batch_size, channels, height, width)，那么 x.view(batch_size, -1) 会将 x 展平为一个形状为 (batch_size, channels * height * width) 的二维张量。这样，每个样本就被表示为一个长度为 channels * height * width 的一维向量。
- x.view(batch_size, -1) 中的 -1 告诉 PyTorch 自动计算展平后的特征总数，以确保展平操作正确地将多维张量转换为具有 320 个特征的一维向量。如果卷积和池化层的输出尺寸是固定的，也可以显式地写出这个数字，例如 x.view(batch_size, 320)。


How to use GPU – 1. Move Model to GPU
刘二大人 bilibili
class Net(torch.nn.Module):
def __init__(self):
super(Net, self).__init__()
self.conv1 = torch.nn.Conv2d(1, 10, kernel_size=5)
self.conv2 = torch.nn.Conv2d(10, 20, kernel_size=5)
self.pooling = torch.nn.MaxPool2d(2)
self.fc = torch.nn.Linear(320, 10)
def forward(self, x):
# Flatten data from (n, 1, 28, 28) to (n, 784)
batch_size = x.size(0)
x = F.relu(self.pooling(self.conv1(x)))
x = F.relu(self.pooling(self.conv2(x)))
x = x.view(batch
x = self.fc(x)
return x
Define device as the first visible cuda device if we have CUDA available.
model = Net()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
Lecturer : Hongpu Liu
Lecture 10-57
PyTorch Tutorial @ SLAM Resea

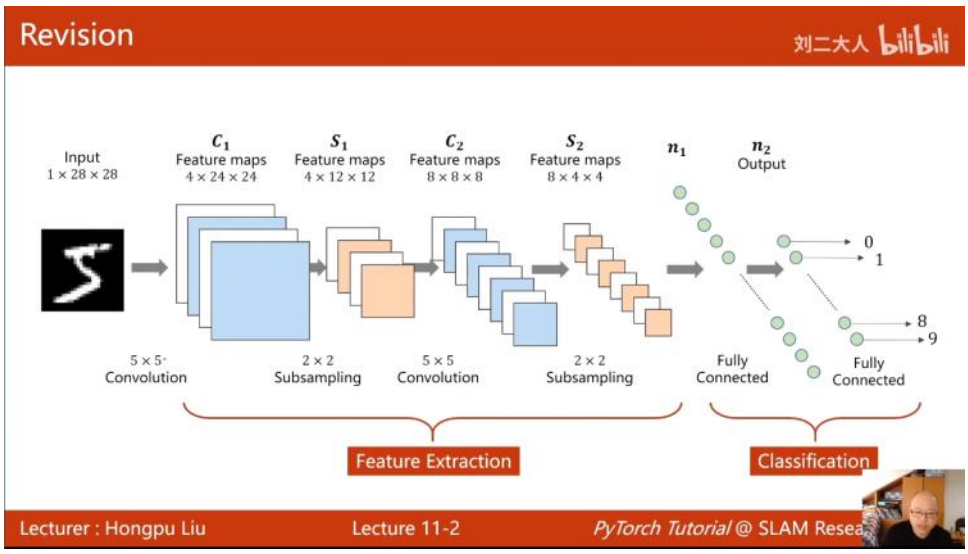

《PyTorch深度学习实践》完结合集
How to use GPU – 2. Move Tensors to GPU
刘二大人 bilibili
def train(epoch):
running_loss = 0.0
for batch_idx, data in enumerate(train_loader, 0):
inputs, target = data
inputs, target = inputs.to(device), target.to(device)
optimizer.zero_grad()
Send the inputs and targets at every step to the GPU.
# forward + backward + update
outputs = model(inputs)
loss = criterion(outputs, target)
loss.backward()
optimizer.step()
running_loss += loss.item()
if batch_idx % 300 == 299:
print('[%d, %5d] loss: %.3f' % (epoch + 1, batch_idx + 1, running_loss / 2000))
running_loss = 0.0
Lecturer : Hongpu Liu
Lecture 10-60
PyTorch Tutorial @ SLAM Resea


How to use GPU – 2. Move Tensors to GPU
刘二大人 bilibili
def test():
correct = 0
total = 0
with torch.no_grad():
for data in test_loader:
inputs, target = data
inputs, target = inputs.to(device), target.to(device)
Send the inputs and targets at every step to the GPU.
outputs = model(inputs)
_, predicted = torch.max(outputs.data, dim=1)
total += target.size(0)
correct += (predicted == target).sum().item()
print('Accuracy on test set: %d %% [%d/%d]' % (100 * correct / total, correct, total))
Lecturer : Hongpu Liu
Lecture 10-62
PyTorch Tutorial @ SLAM Resea

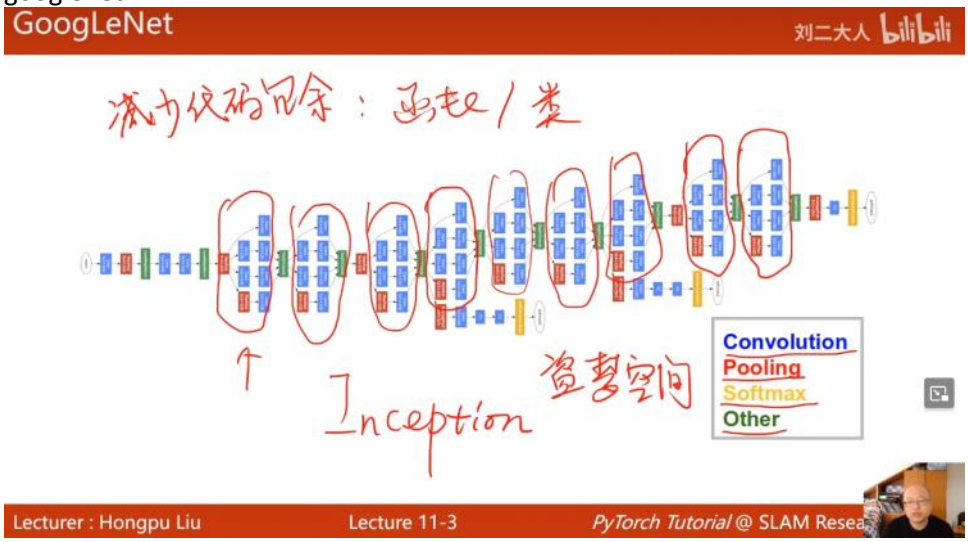
Z

卷积神经网络（高级篇）

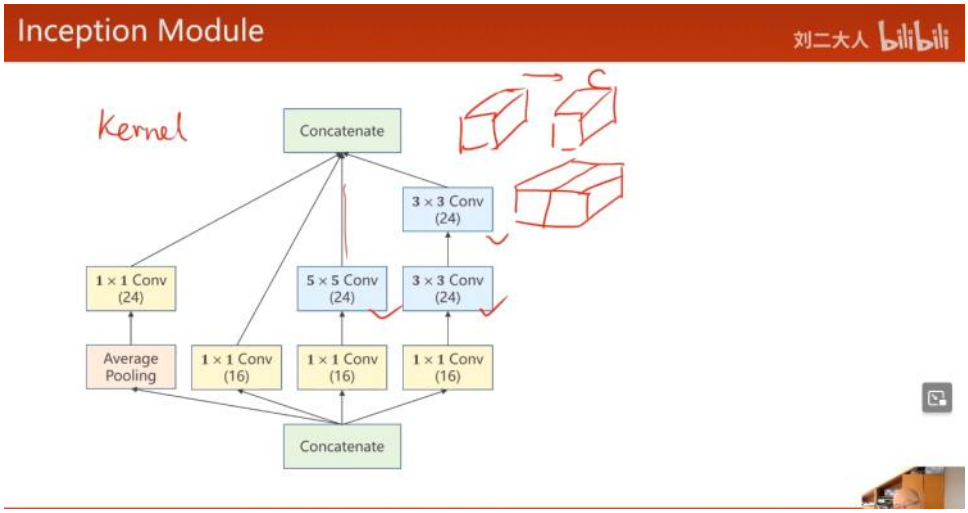
2024年5月6日　　19:33



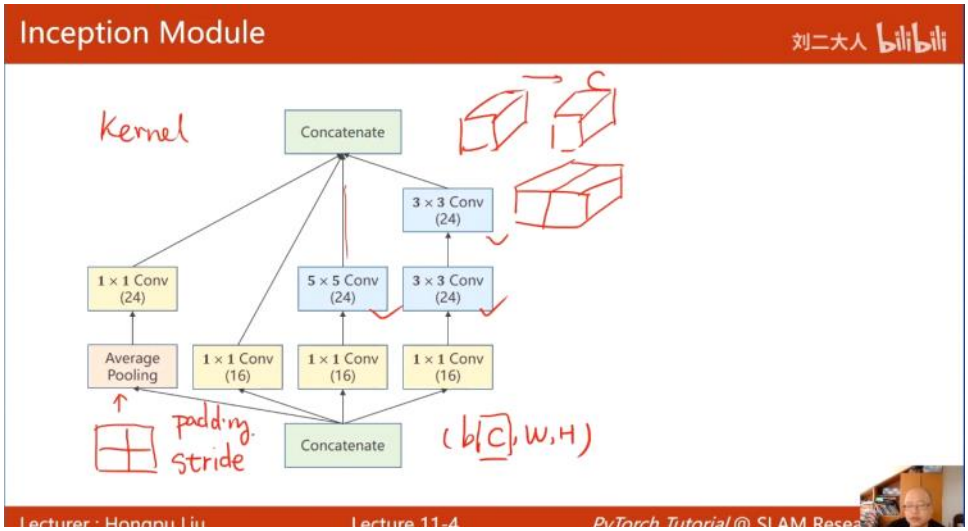
googlenet



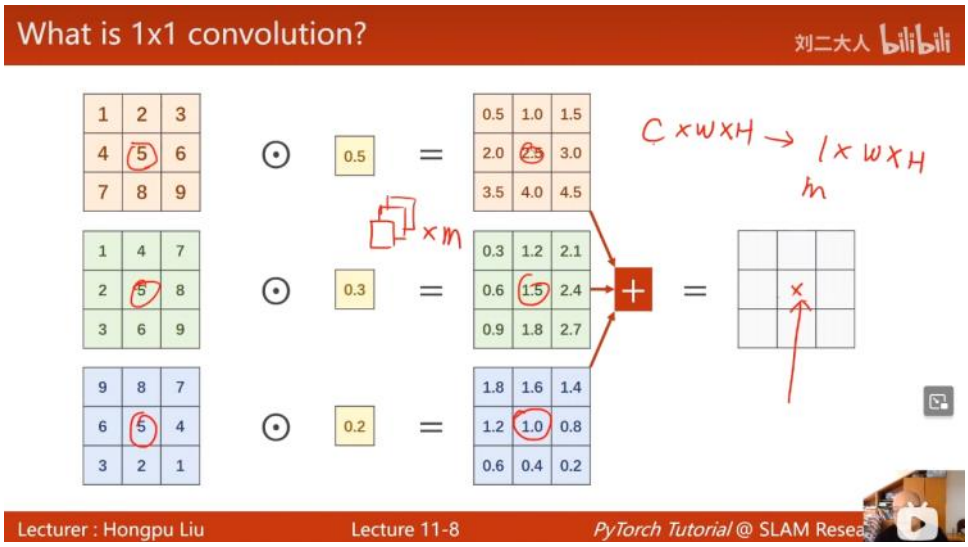
超参数：难选的参数 比如卷积核的尺寸，所以inception里选了很多个，通过训练，找到最优的组合



走不同路径唯一可以不同的只有channel,w和h必须一致

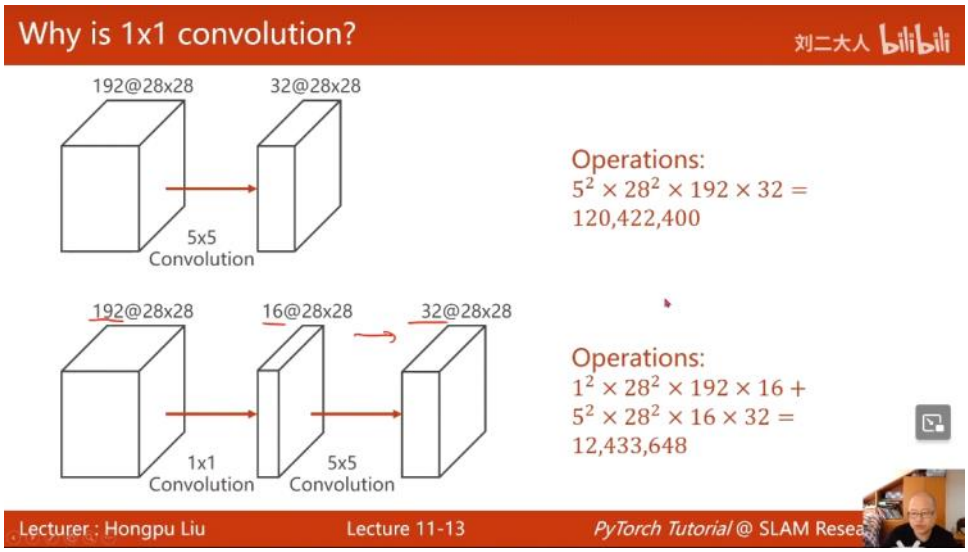


池化层通过padding保障宽高尺寸不变

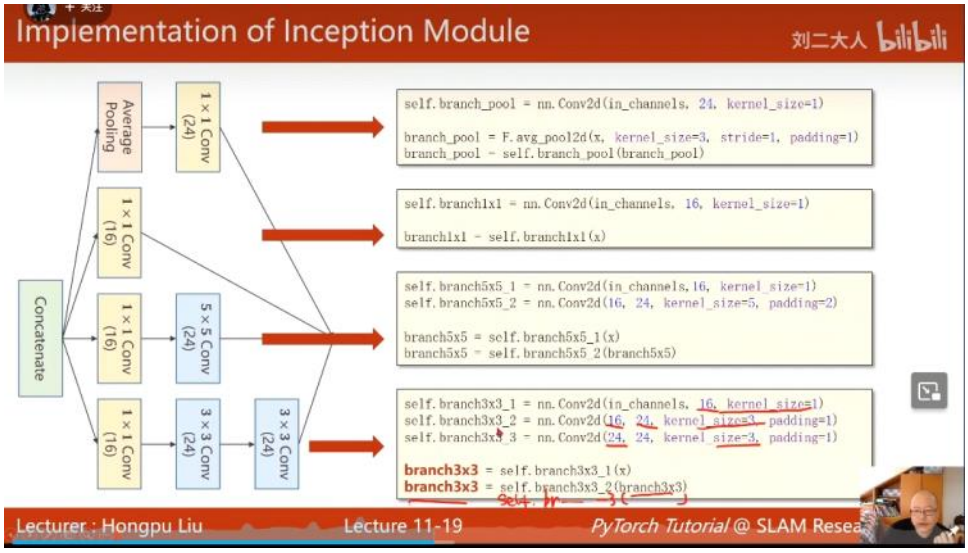


传说中的信息融合

1*1卷积核的意义就是改变通道数，见下图



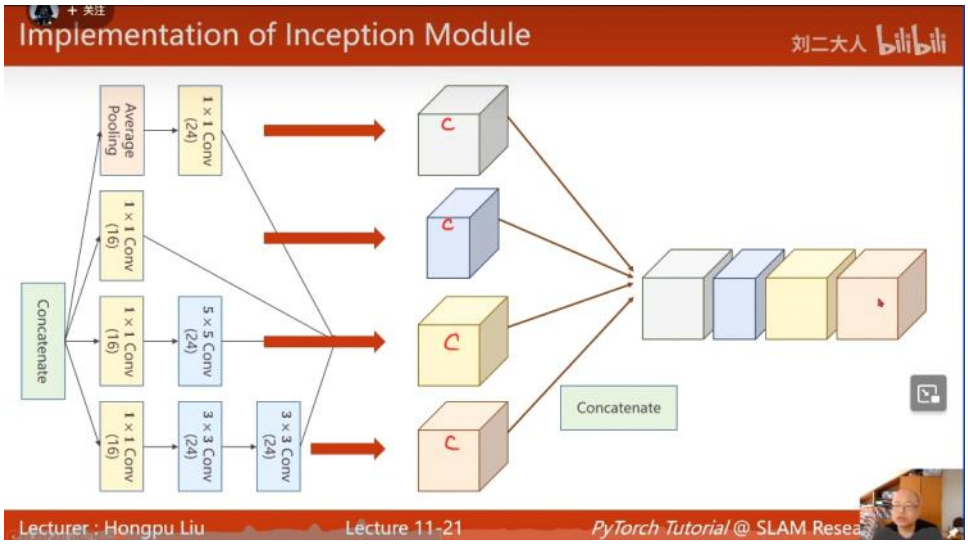
用了1*1的卷积核后，整体运算量只有原来的1/10

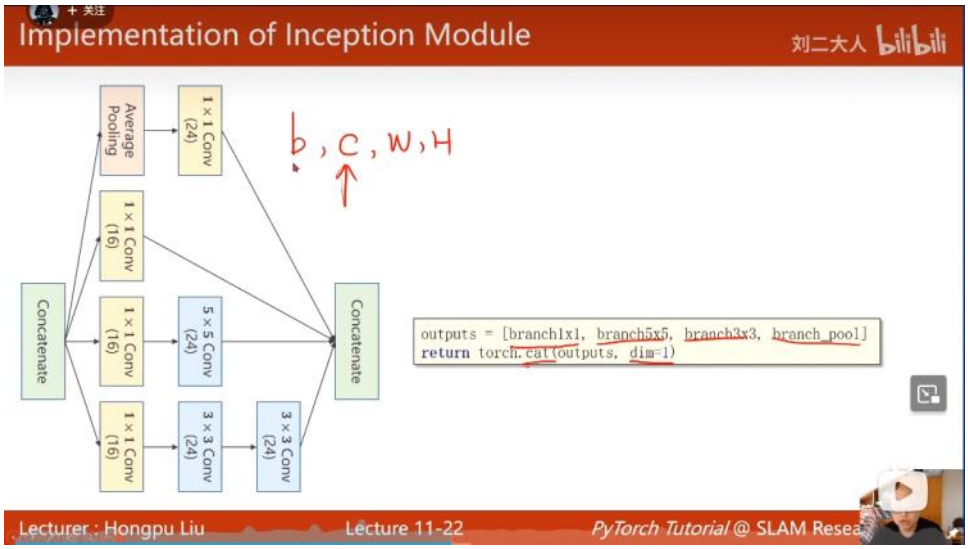


1.分支1 输出通道是24

2.分支2 输出通道是16

.....



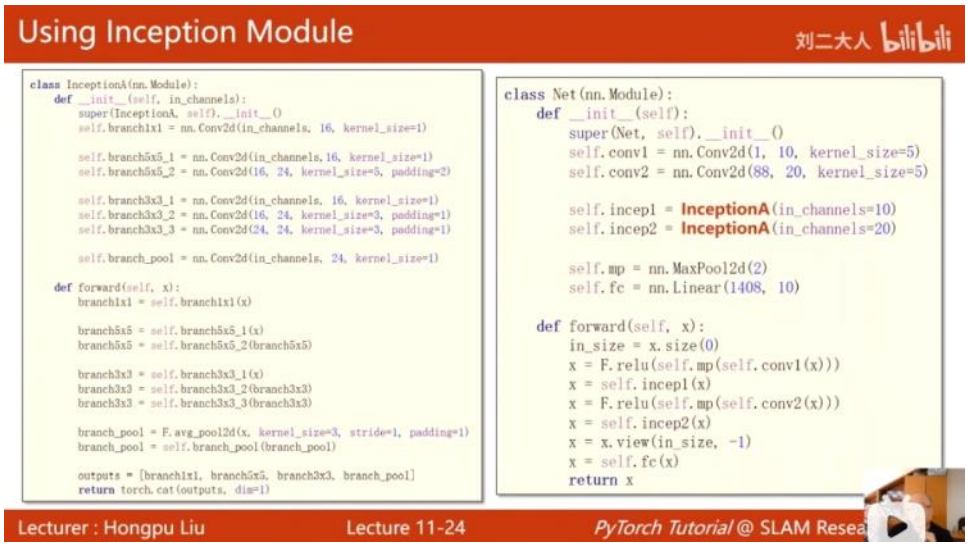


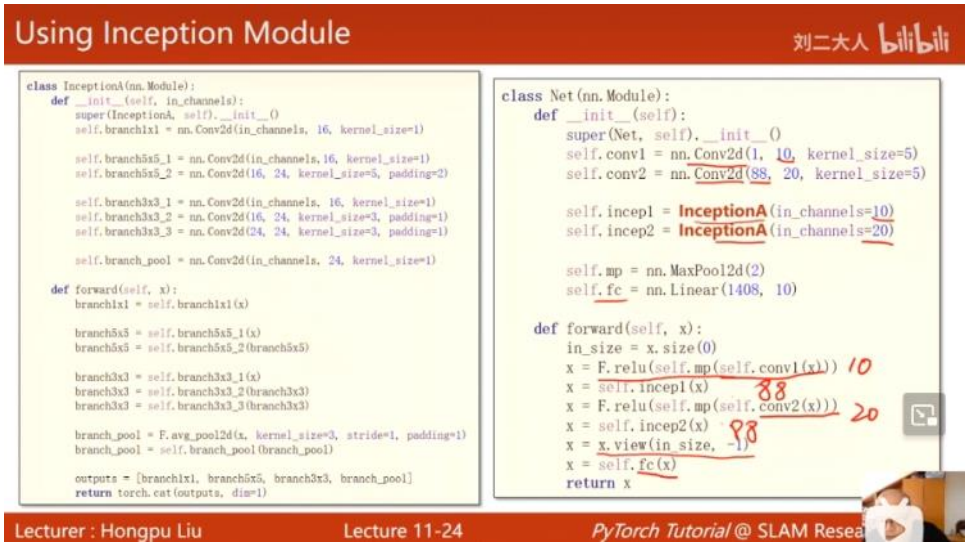
dim=1从channel维度拼起来

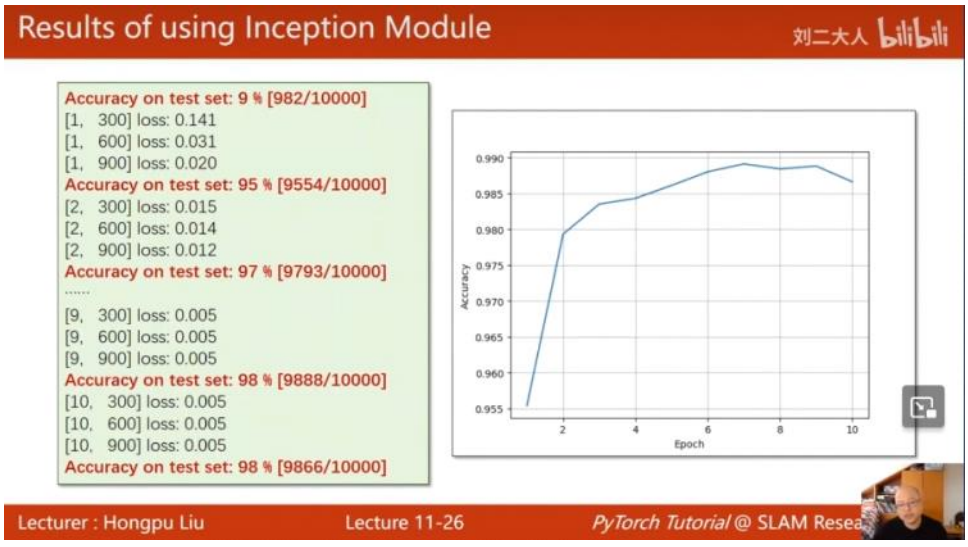
orch.cat(outputs, dim=1)是在通道的维度上将这四个分支的输出连接起来。所以，结果张量的通道数将是这四个分支的通道数之和：16 (branch1x1) + 24 (branch5x5) + 24

(branch3x3) + 24 (branch_pool) = 88。

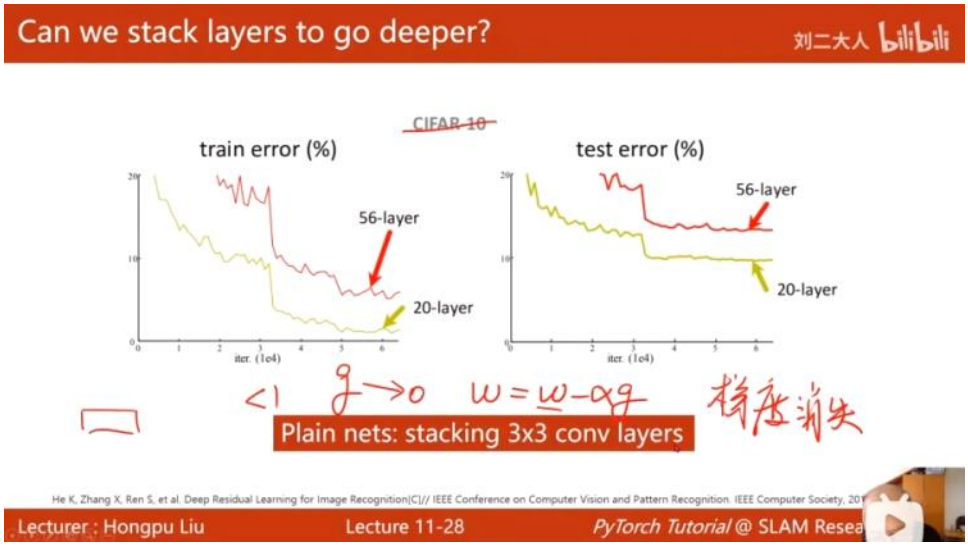
拼接要求宽度和高度一致，不要求通道数一致。







存在梯度消失的问题

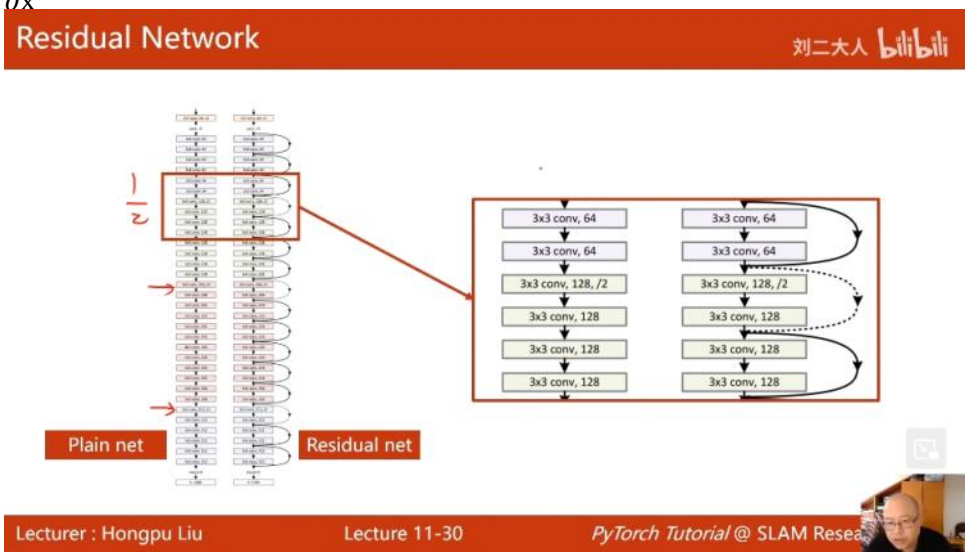


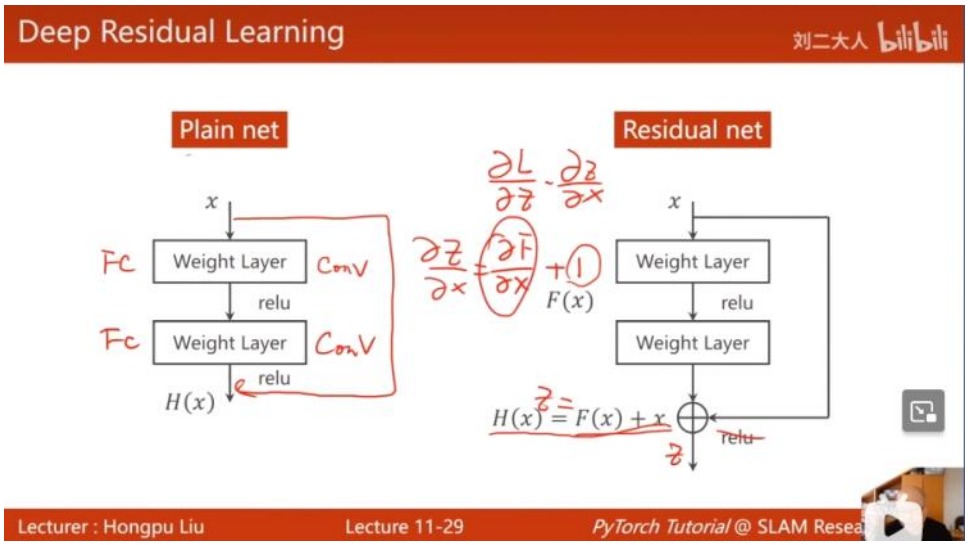
residual net

解决梯度消失问题

H(X)=F(X)+X

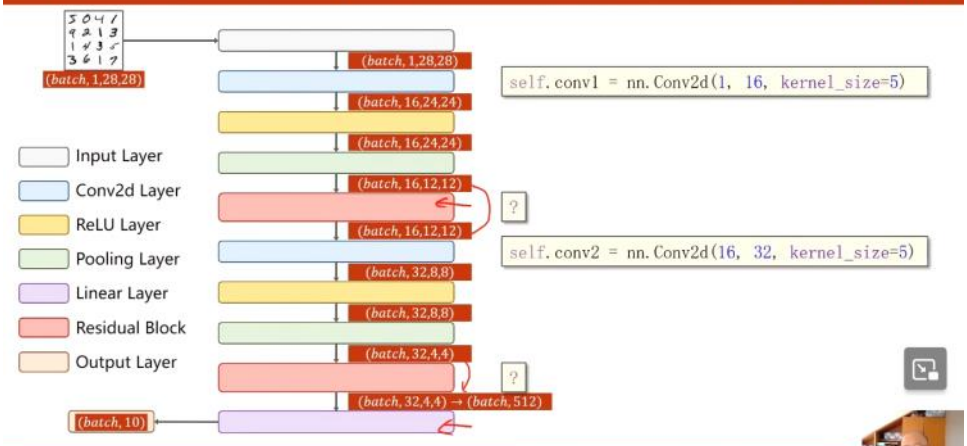
$\frac{\partial F}{\partial x}$趋于0的时候，加1





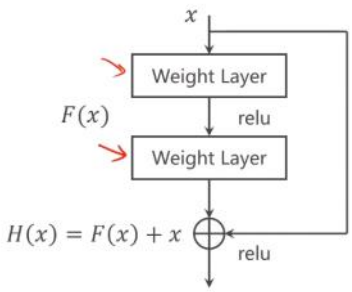
residual block

Implementation of Simple Residual Network



Lecturer : Hongpu Liu Lecture 11-31 *PyTorch Tutorial* @ SLAM Resea

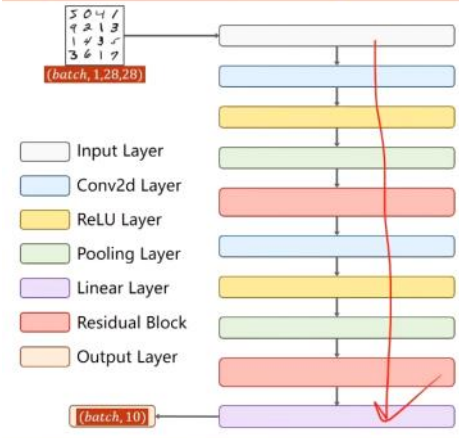
Implementation of Residual Block




```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.channels = channels
        self.conv1 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(channels, channels,
                               kernel_size=3, padding=1)

    def forward(self, x):
        y = F.relu(self.conv1(x))
        y = self.conv2(y)
        return F.relu(x + y)
```

Lecturer : Hongpu Liu Lecture 11-32 *PyTorch Tutorial* @ SLAM Resea

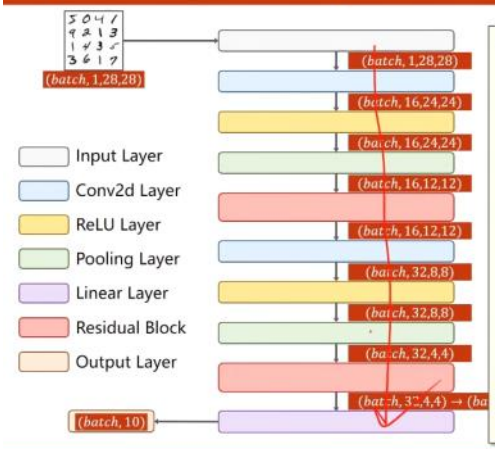
Implementation of Simple Residual Network




```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5)
        self.mp = nn.MaxPool2d(2)

        self.rblock1 = ResidualBlock(16)
        self.rblock2 = ResidualBlock(32)

        self.fc = nn.Linear(512, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = self.mp(F.relu(self.conv1(x)))
        x = self.rblock1(x)
        x = self.mp(F.relu(self.conv2(x)))
        x = self.rblock2(x)
        x = x.view(in_size, -1)
        x = self.fc(x)
        return x
```

Lecturer : Hongpu Liu Lecture 11-38 *PyTorch Tutorial* @ SLAM Resea

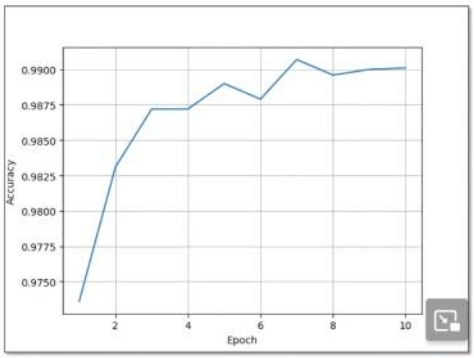
Implementation of Simple Residual Network



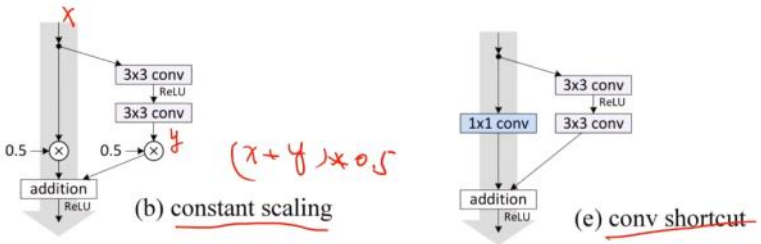

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5)
        self.mp = nn.MaxPool2d(2)

        self.rblock1 = ResidualBlock(16)
        self.rblock2 = ResidualBlock(32)

        self.fc = nn.Linear(512, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = self.mp(F.relu(self.conv1(x)))
        x = self.rblock1(x)
        x = self.mp(F.relu(self.conv2(x)))
        x = self.rblock2(x)
        x = x.view(in_size, -1)
        x = self.fc(x)
        return x
```



```
Accuracy on test set: 9 % [916/10000]
[1,   300] loss: 0.074
[1,   600] loss: 0.021
[1,   900] loss: 0.017
Accuracy on test set: 97 % [9736/10000]
[2,   300] loss: 0.013
[2,   600] loss: 0.011
[2,   900] loss: 0.011
Accuracy on test set: 98 % [9831/10000]
......
[9,   300] loss: 0.003
[9,   600] loss: 0.004
[9,   900] loss: 0.004
Accuracy on test set: 99 % [9900/10000]
[10,   300] loss: 0.003
[10,   600] loss: 0.003
[10,   900] loss: 0.004
Accuracy on test set: 99 % [9901/10000]
```



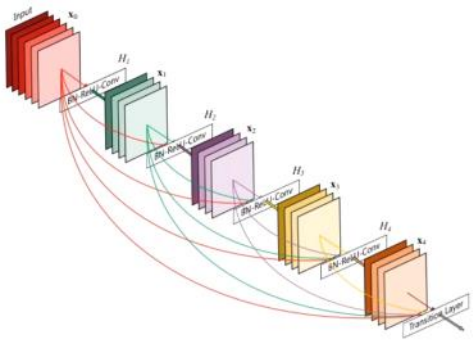
Exercise 11-1: Reading Paper and Implementing



He K, Zhang X, Ren S, et al. Identity Mappings in Deep Residual Networks[C]

Exercise 11-2: Reading and Implementing DenseNet

刘二大人 bilibili



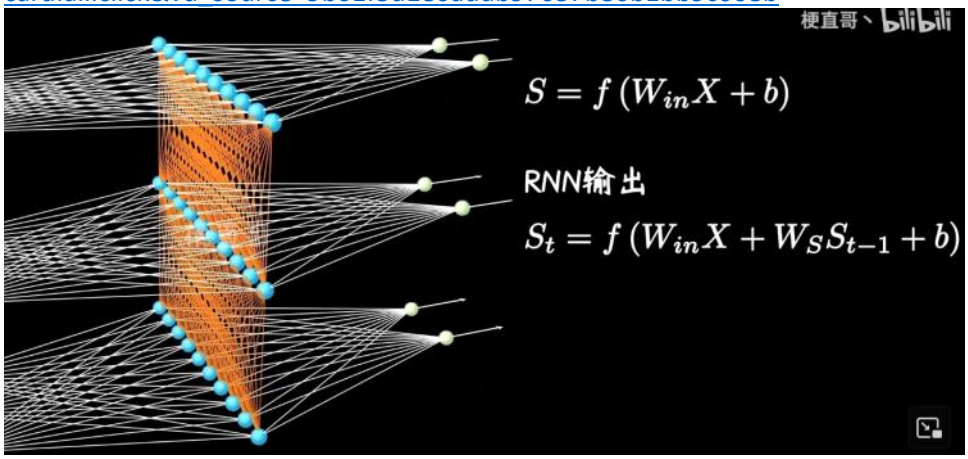
Huang G, Liu Z, Laurens V D M, et al. Densely Connected Convolutional Networks[J]. 2016:2261-2269.

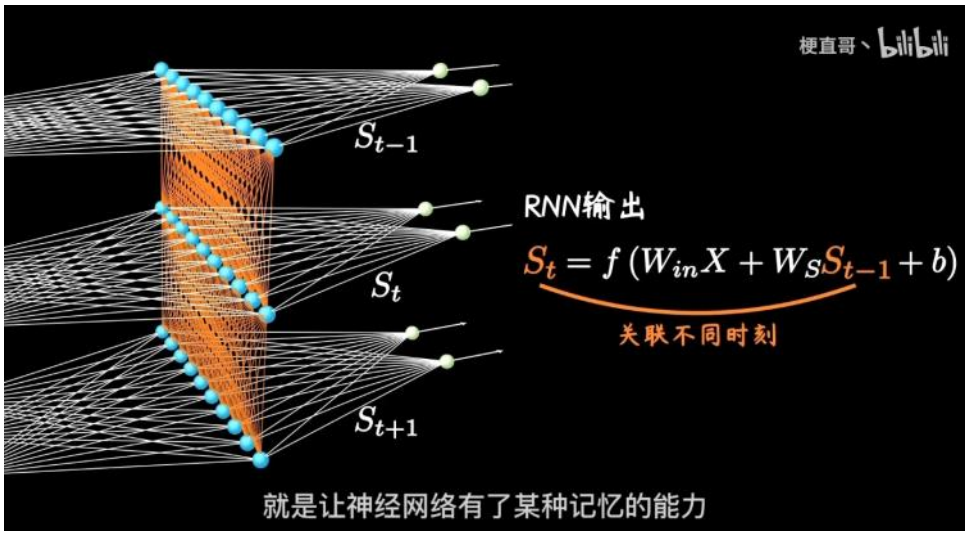
Lecturer : Hongpu Liu Lecture 11-41 *PyTorch Tutorial* @ SLAM Resea

循环基础

2024年5月9日 10:12

https://www.bilibili.com/video/BV1z5411f7Bm/?spm_id_from=333.337.search-card.all.click&vd_source=9b01f3d1e6addb97637b80b1bb9c008b





对于图片判定CNN就够了，因为没有时序

但是对于语言，涉及到时序，就需要RNN

循环神经网络（基础版）

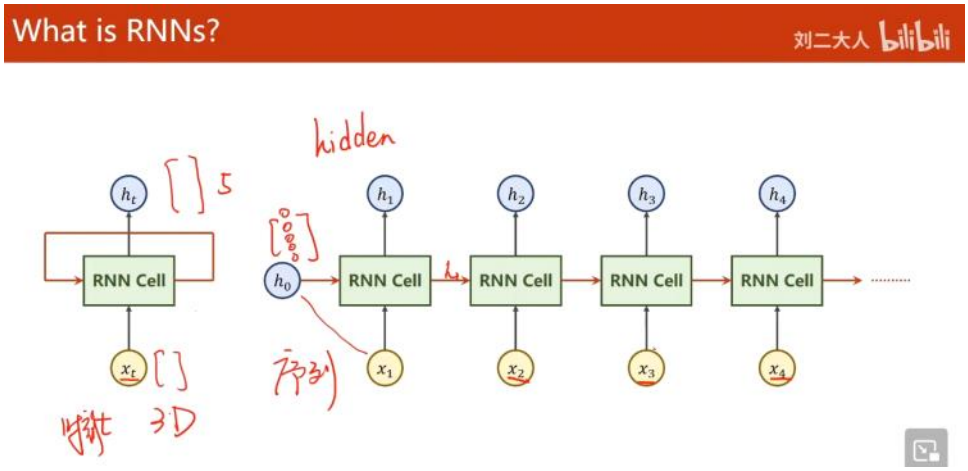
2024年5月8日　19:59

RNN处理具有序列连接的数据，比如天气、自然语言

RNNcell的本质是一个线性层

举例：

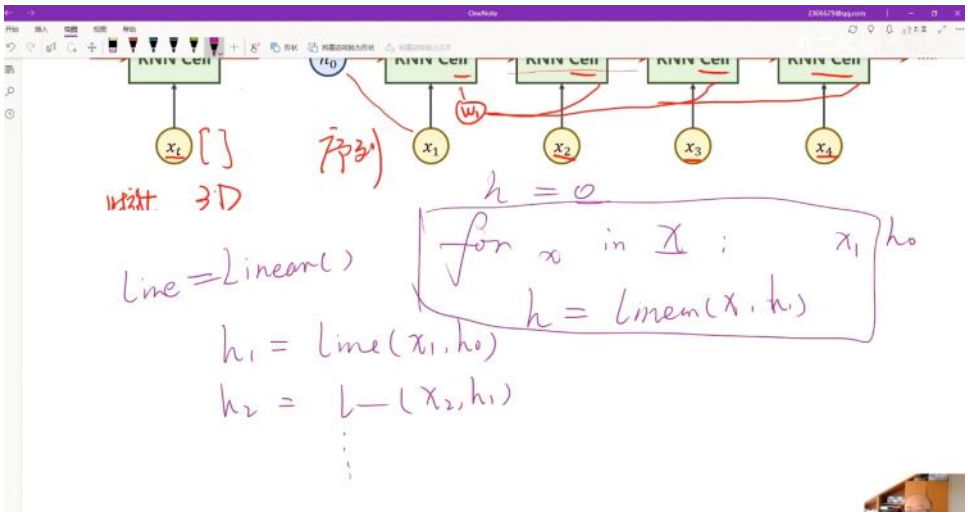
RNN Cell 将3维的数据升成5维（向量），说明其本质是线性层



x_2和h_1融合，送到$RNN\ \ Cell$

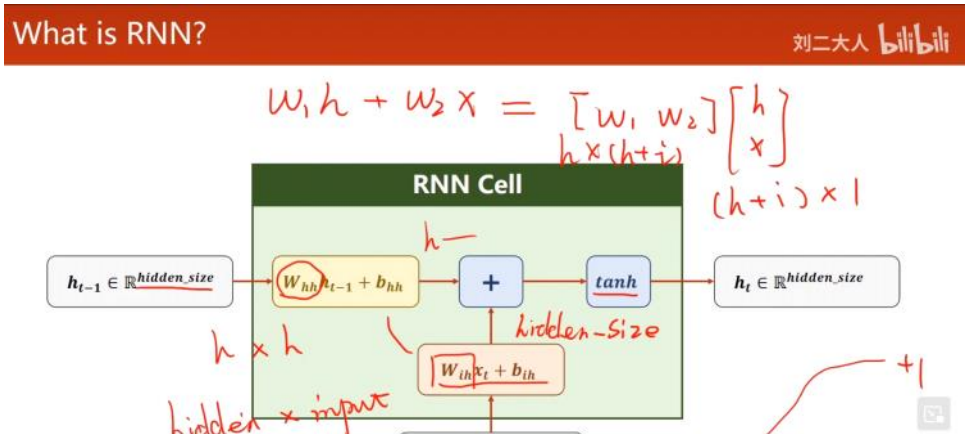
这些RNN Cell都是同一个线性层，循环使用

循环的计算过程

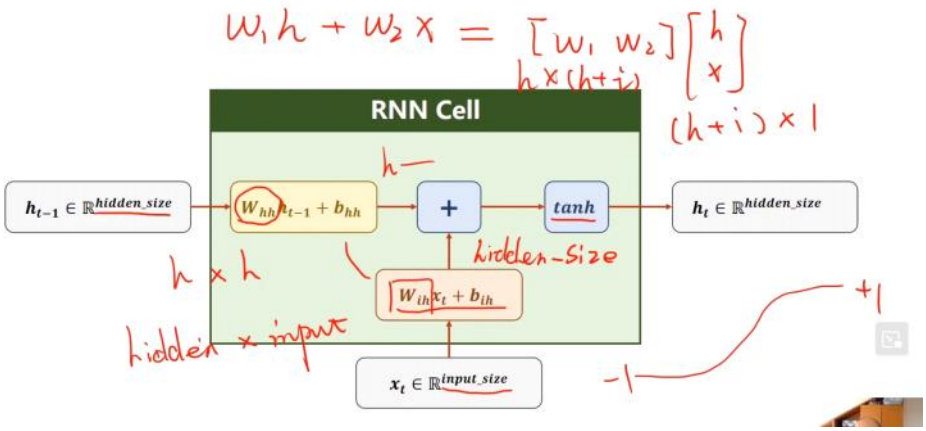


https://blog.csdn.net/tcn760/article/details/124010118?ops_request_misc=&request_id=&biz_id=102&spm=1018.2226.3001.4187

使用的是tanh激活函数

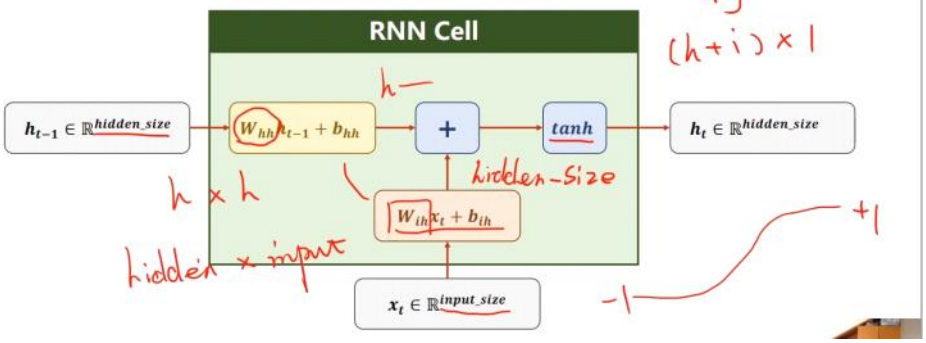


What is RNN?



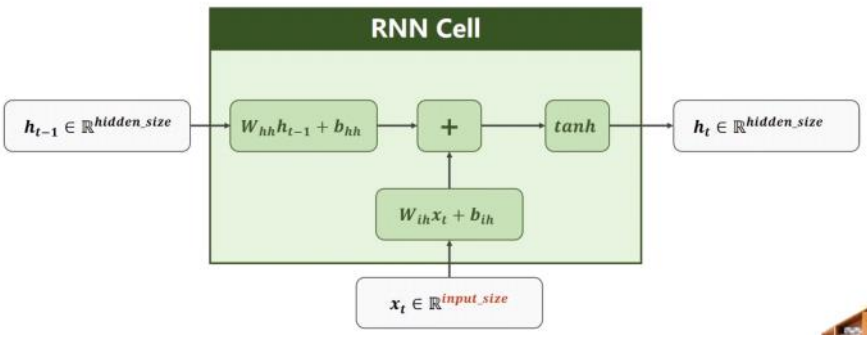
What is RNN?

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh})$$



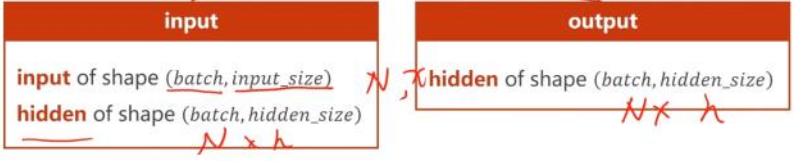
RNN Cell in PyTorch

```
cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)
```



RNN Cell in PyTorch

```
cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)

hidden = cell(input, hidden)
```

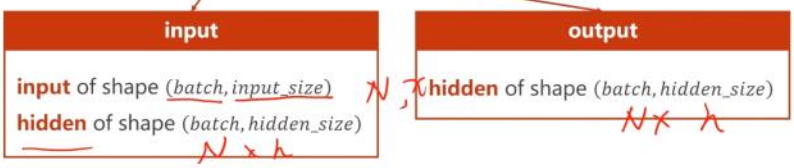


RNN Cell in PyTorch

刘二大人 bilibili

```
cell = torch.nn.RNNCell(input_size=input_size, hidden_size=hidden_size)
hidden = cell(input, hidden)
```

$h_1 = cell(x_1, h_0)$



How to use RNNCell

刘二大人 bilibili

- Suppose we have sequence with below properties:
  - $batchSize = 1$
  - $seqLen = 3$
  - $inputSize = 4$
  - $hiddenSize = 2$
- So the shape of inputs and outputs of RNNCell:
  - $input.shape = (batchSize, inputSize)$
  - $output.shape = (batchSize, hiddenSize)$
- The sequence can be warped in one Tensor with shape:
  - $dataset.shape = (seqLen, batchSize, inputSize)$