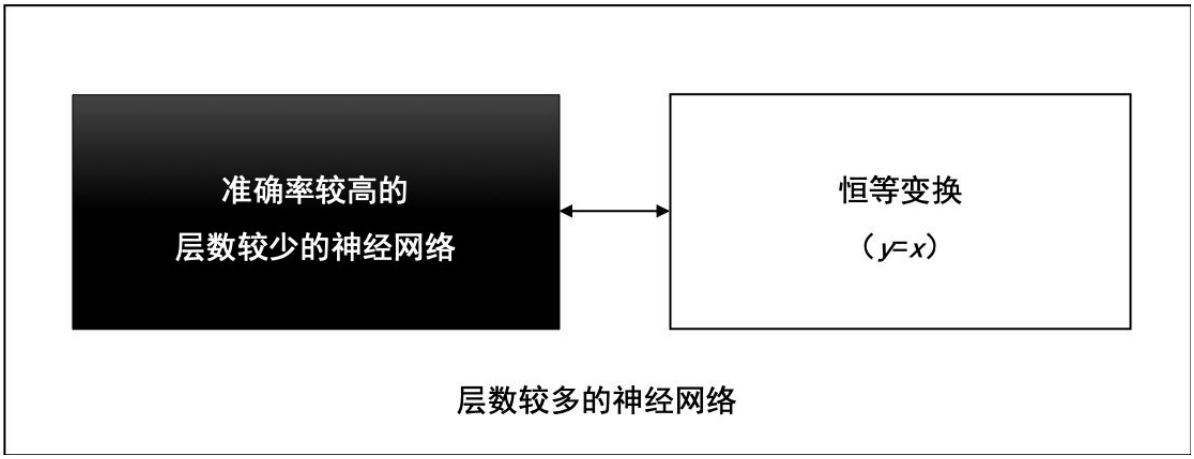
Residual Network（ResNet 残差神经网络）

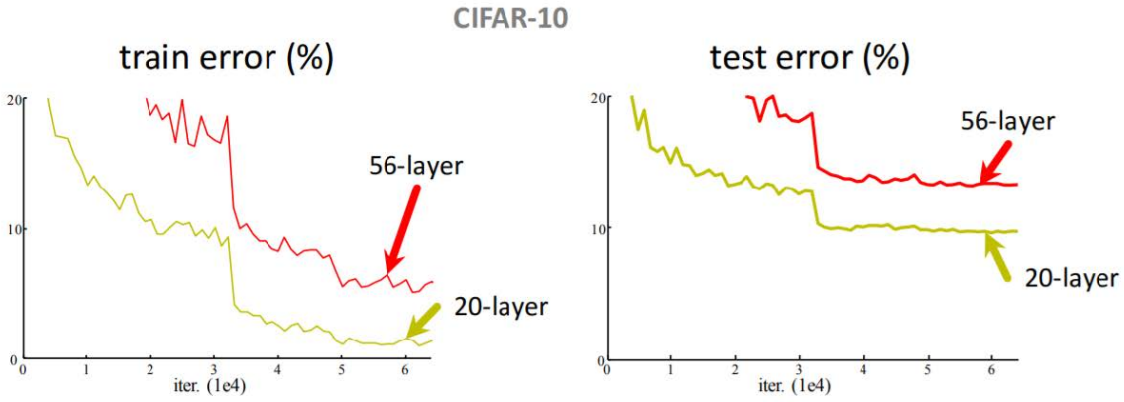
1. 提出问题：神经网络是不是越深越好?

这个问题在理论上和大家的认知上是正确的，如下图：



假设一个层数较少的神经网络已经达到了较高准确率，我们可以在这个神经网络之后，**拼接一段恒等变换的网络层**，这些恒等变换的网络层对输入数据不做任何转换，直接返回（$y=x$），就能得到一个深度较大的神经网络，这个深度较大的神经网络的准确率等于拼接之前的神经网络准确率，**准确率没有理由降低。**

通过实验，ResNet随着网络层不断的加深，模型的准确率先是不断的提高，达到最大值（准确率饱和），然后随着网络深度的继续增加，模型准确率毫无征兆的出现大幅度的降低。

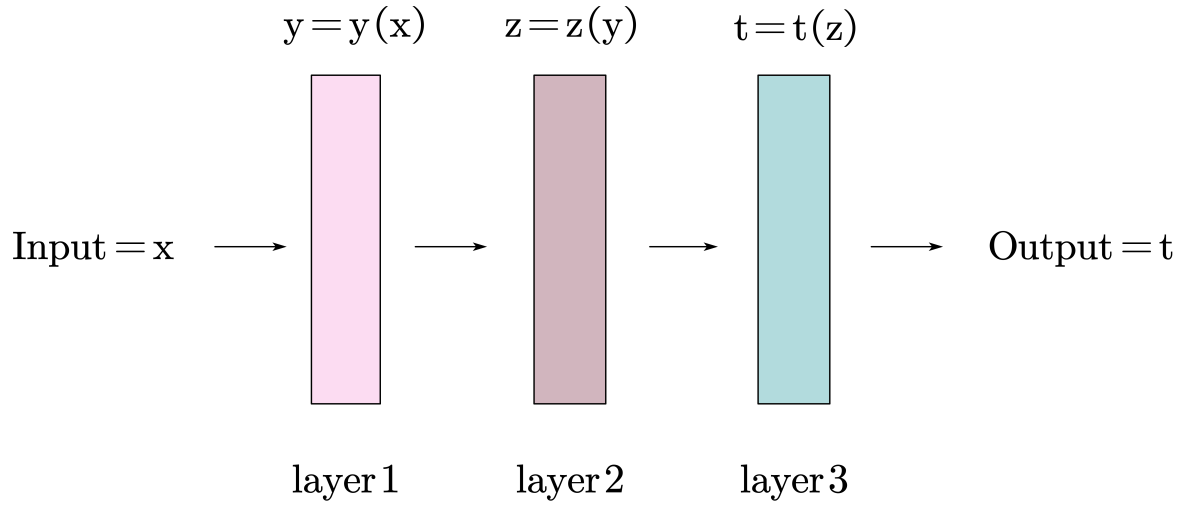


把3x3的卷积层不断堆积，会发现20层的效果是比56层的效果好？！

注意：纵坐标是loss

这个现象与"越深的网络准确率越高"的信念显然是矛盾的、冲突的。ResNet团队把这一现象称为"退化（Degradation）"。

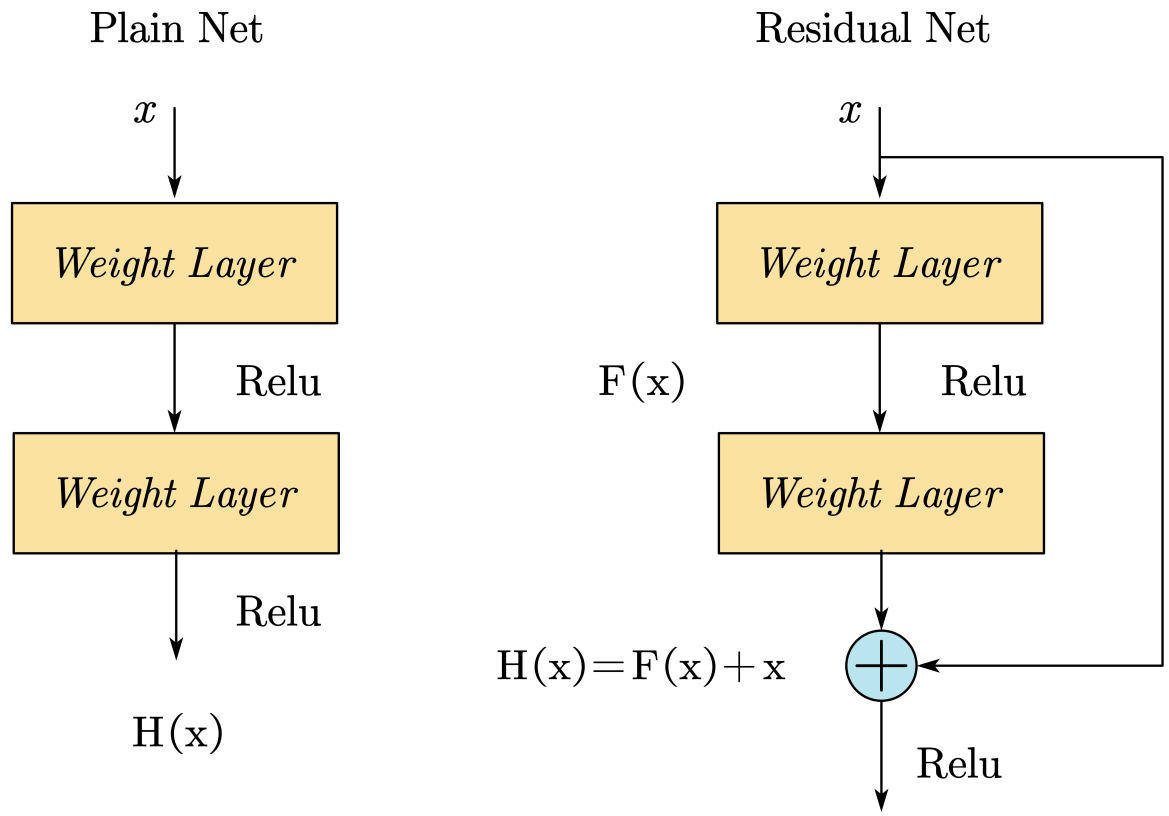
可以用**梯度消失**来理解，离输入比较近的部分实际上无法得到充分的训练

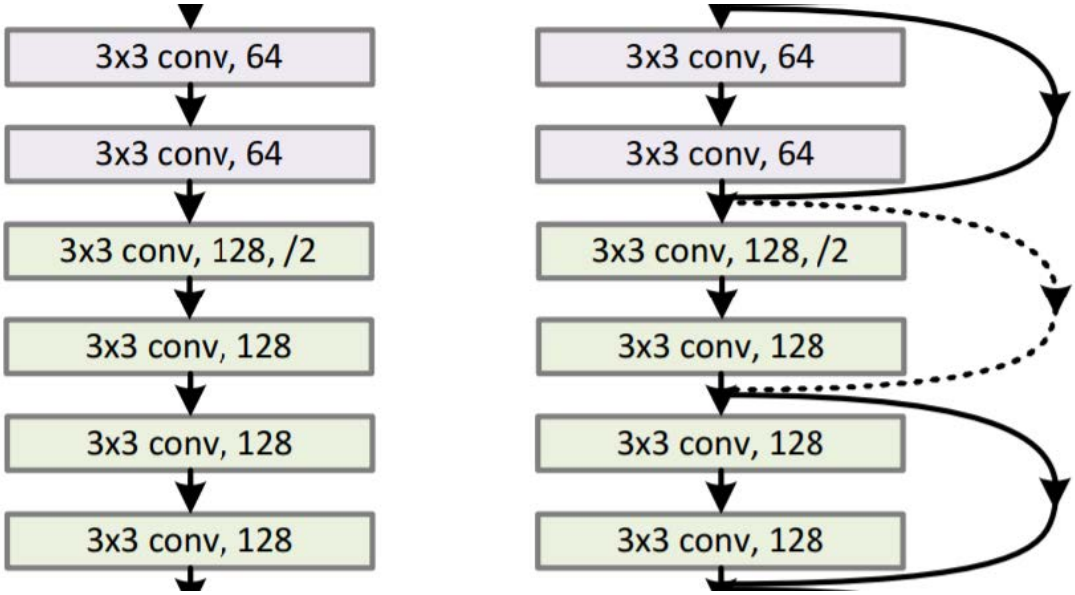


训练layer 1： $\nabla_x \text{Output} = \nabla_x y \cdot \nabla_y z \cdot \nabla_z t$

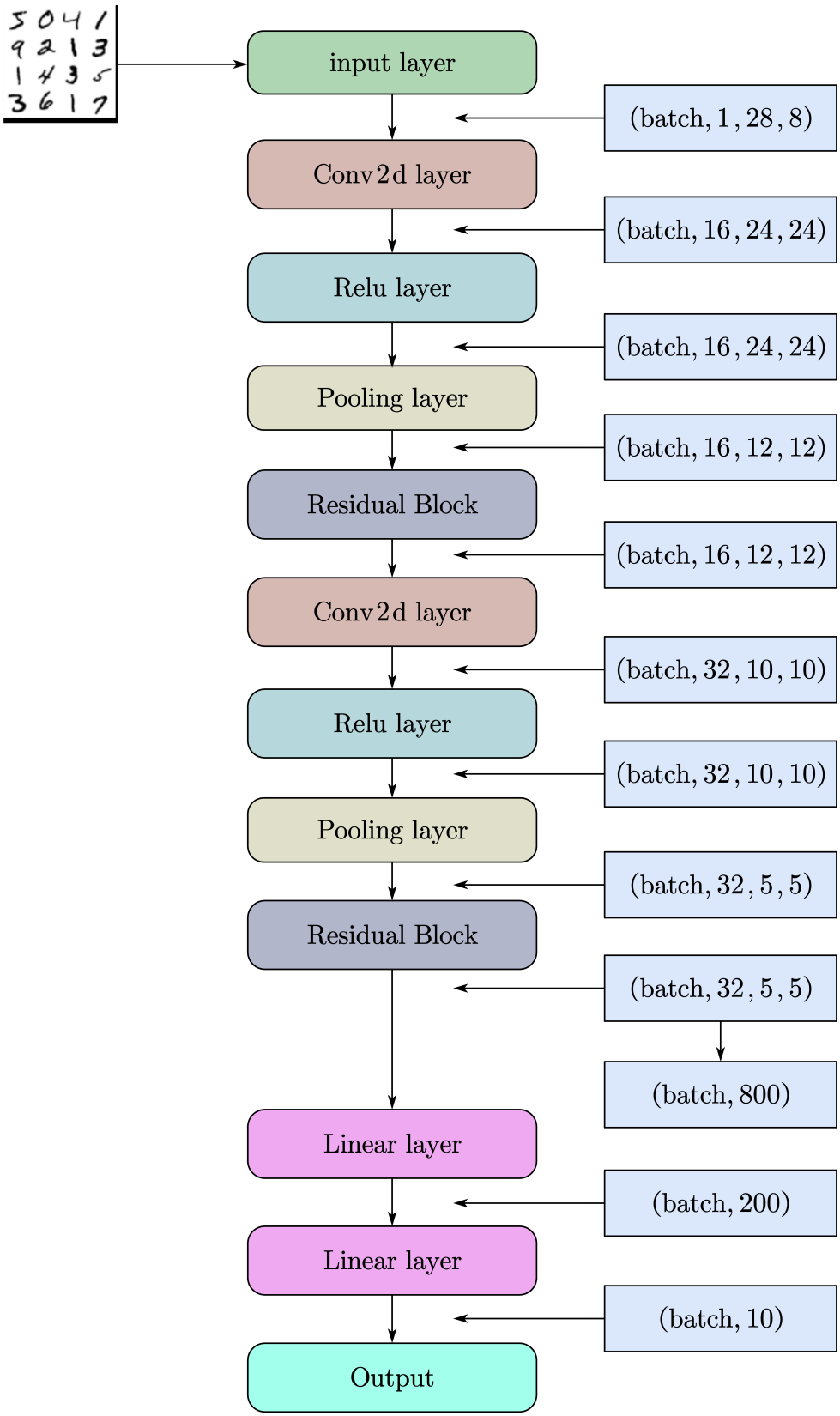
训练layer 2： $\nabla_y \text{Output} = \nabla_y z \cdot \nabla_z t$

2. ResNet的构造





3. 代码实践（数据集为MNIST）




```
import torch
from torchvision import transforms
from torchvision import datasets
from torch.utils.data import DataLoader
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
```

```
# prepare dataset
# ToTensor()将shape为(H, W, C)的np.ndarray或img转为shape为(C, H, W)的tensor,并将每一个数值归一化到[0,1]
# Normalize()使用公式"(x-mean)/std"，将每个元素分布到(-1,1)
tran = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))])

train_dataset = datasets.MNIST(root='../MNIST-Classification/data/MNIST-dataset/mnist/', train=True, download=True,
                               transform=tran)
train_loader = DataLoader(train_dataset, shuffle=True, batch_size=30)
test_dataset = datasets.MNIST(root='../MNIST-Classification/data/MNIST-dataset/mnist/', train=False, download=True,
                              transform=tran)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=30)


# Use GPU or CPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


# Design model
class ResBlock(torch.nn.Module):
    def __init__(self, channel):
        super(ResBlock, self).__init__()
        self.channel = channel
        self.conv1 = torch.nn.Conv2d(channel, channel, kernel_size=3, padding=1)
        self.conv2 = torch.nn.Conv2d(channel, channel, kernel_size=5, padding=2)
        # self.conv3 = torch.nn.Conv2d(channel, channel, kernel_size=1)
        # self.sig = torch.nn.Sigmoid()

    def forward(self, x):
        y = F.relu(self.conv1(x))
        y = self.conv2(y)
        return F.relu(x + y)
        # g = F.sigmoid(self.conv3(x))
        # return F.relu((1-g)*x + g*y)


class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        # Convolution layer
        self.conv1 = torch.nn.Conv2d(1, 16, kernel_size=5)
        self.conv2 = torch.nn.Conv2d(16, 32, kernel_size=3)
        # MaxPooling layer
        self.MP = torch.nn.MaxPool2d(2)
        # Residual Block
        self.RB1 = ResBlock(16)
        self.RB2 = ResBlock(32)
        # Linear layer
        self.linear1 = torch.nn.Linear(800, 200)
        self.linear2 = torch.nn.Linear(200, 10)
```

```
    def forward(self, x):
        batch = x.size(0)  # get the batch_size of x
        y = self.MP(F.relu(self.conv1(x)))
        y = self.RB1(y)
        y = self.MP(F.relu(self.conv2(y)))
        y = self.RB2(y)
        y = y.view(batch, -1)
        y = F.relu(self.linear1(y))
        y = self.linear2(y)
        return y


# construct loss and optimizer
model = Model()
model.to(device)
loss = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.005, momentum=0.5)  # lr:学习率


# train
def train(epoch):
    num = 0
    # enumerate have two returned value, i.e. serial number and data(including
train_x and label)
    for i, data1 in enumerate(train_loader, 0):
        (x_1, y_1) = data1
        (x_1, y_1) = (x_1.to(device), y_1.to(device))

        # forward
        y_hat = model.forward(x_1)
        l = loss(y_hat, y_1)
        num += 1
        if num % 300 == 0:
            print('epoch:', epoch, 'num:', num, 'Loss:', l.item())

        # backward
        optimizer.zero_grad()
        l.backward()

        # update
        optimizer.step()


# test
def test():
    correct = 0
    total = 0
    with torch.no_grad():
        for data2 in test_loader:
            (x_test, labels) = data2
            (x_test, labels) = (x_test.to(device), labels.to(device))
            y_test = model.forward(x_test)
            _, predict = torch.max(y_test.data, dim=1)
            total += labels.size(0)
            correct += (predict == labels).sum().item()
```

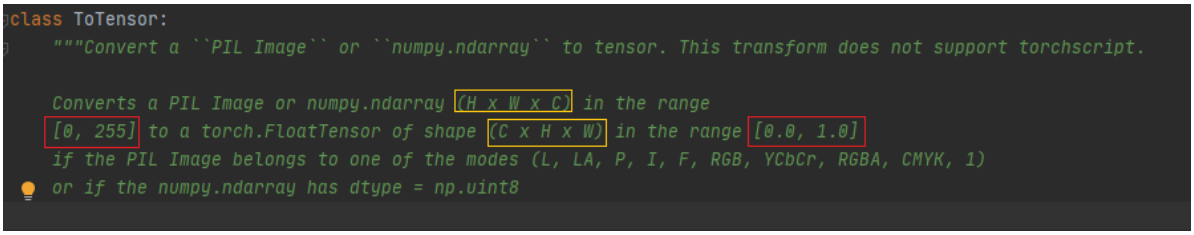
```
    # a.append(correct/total)
    print('准确率: ', (100 * correct / total), '%')
    return correct / total


if __name__ == "__main__":
    num_epo = 50
    acc_list = []
    acc_max = 0.992

    for epoch in range(1, num_epo):
        train(epoch)
        acc = test()
        # use to record acc
        acc_list.append(acc)

        if acc > acc_max:
            # save model at present
            torch.save(obj=model.state_dict(), f="./models/Res.pth")
            acc_max = acc

    # plot
    Epoch = list(range(1, num_epo))
    plt.plot(Epoch, acc_list, 'o-b')
    plt.xlabel("epoch")
    plt.ylabel("Accuracy")
    for i in range(0, num_epo - 1):
        plt.text(Epoch[i], acc_list[i], round(acc_list[i], 4), fontsize=10,
verticalalignment="bottom",
                 horizontalalignment="center")
    plt.show()
```

部分代码详解：

```
tran = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))])
```

`transforms.ToTensor()`：

可以将PIL和numpy格式的数据从[0,255]范围转换到[0,1]，具体做法其实就是将原始数据除以255。另外原始数据的shape是（H x W x C），通过`transforms.ToTensor()`后shape会变为（C x H x W）。

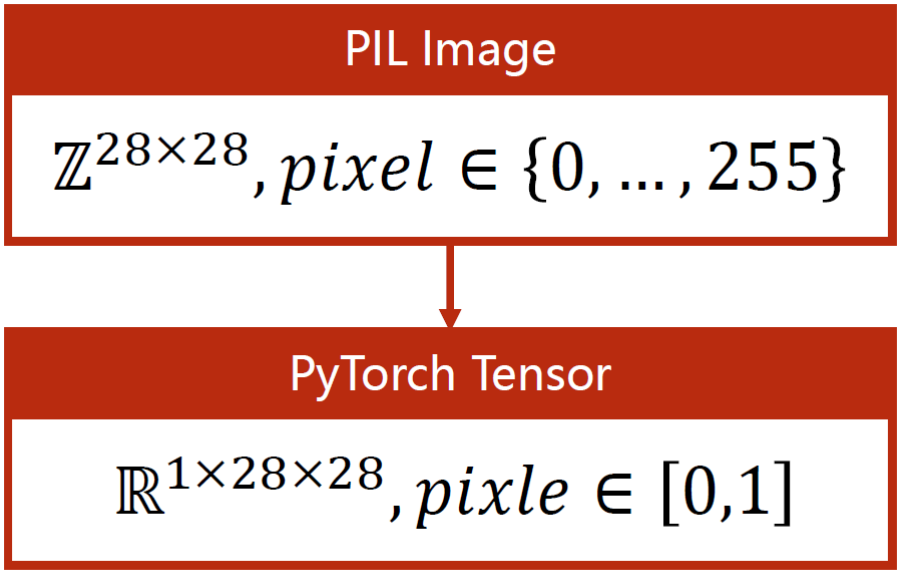


`transforms.Normalize`：

对每个通道执行output[channel] = (input[channel] - mean[channel]) / std[channel]

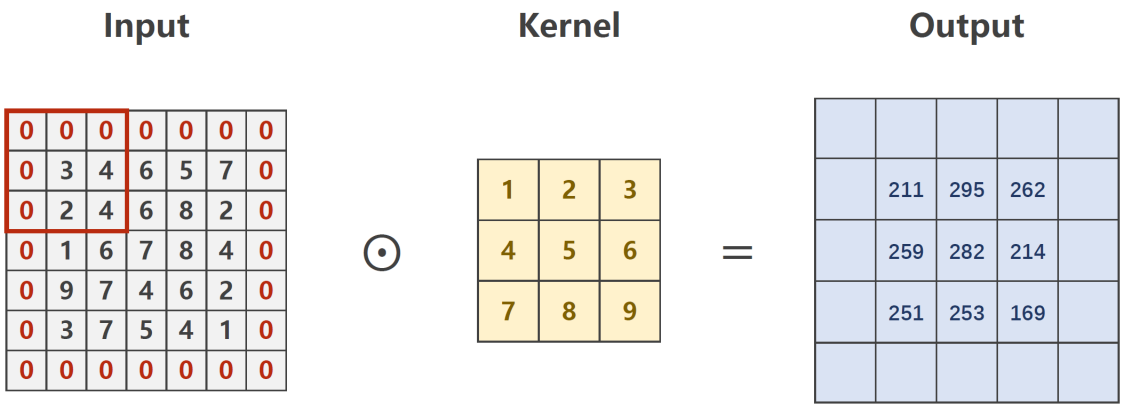
```
class Normalize(torch.nn.Module):
    """Normalize a tensor image with mean and standard deviation.
    This transform does not support PIL Image.
    Given mean: ``(mean[1],...,mean[n])`` and std: ``(std[1],..,std[n])`` for ``n``
    channels, this transform will normalize each channel of the input
    ``torch.*Tensor`` i.e.,
    ``output[channel] = (input[channel] - mean[channel]) / std[channel]``
```

最终效果：



我们完成了这么一件事

```
padding=1 & 2
```



要让W,H不变