# 5.PyTorch可视化

## 5.1.TensorBoard安装

TensorBoard是一个可视化工具，它可以用来展示网络图、张量的指标变化、张量的分布情况等。特别是在训练网络的时候，我们可以设置不同的参数（比如：权重W、偏置B、卷积层数、全连接层数等），使用TensorBoader可以很直观的帮我们进行参数的选择。

TensorBoard的执行流程为：

1. 通过python脚本，记录可视化的数据
2. 运行代码，可以生成了一个或多个事件文件(event files)
3. 启动TensorBoard的Web服务器展示数据

TensorBoard安装过程：

pip install tensorboard

pip install future

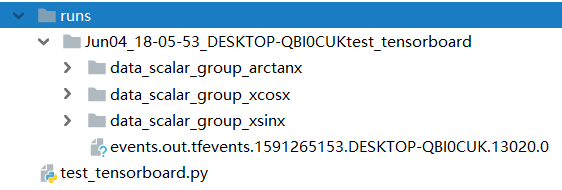
安装完成后，测试TensorBoard

代码实现：

*# -\*- coding:utf-8 -\*-***import** numpy **as** np  
*# 导入SummaryWriter，用来记录需要可视化的数据***from** torch.utils.tensorboard **import** SummaryWriter

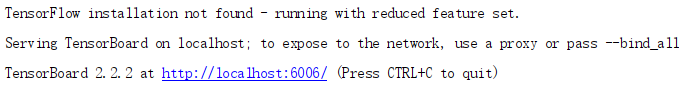
*# 创建SummaryWriter，用于将数据存入磁盘，参数为存入磁盘文件目录的后缀名*  
writer = SummaryWriter(comment=**'test\_tensorboard'**)  
  
**for** x **in** range(100):  
  
 *# 记录标量，三个参数分别为曲线的名称、曲线的y值、曲线的x值* writer.add\_scalar(**'y=2x'**, x \* 2, x)  
 writer.add\_scalar(**'y=pow(2, x)'**, 2 \*\* x, x)  
  
 *# 一张图中记录多条曲线* writer.add\_scalars(**'data/scalar\_group'**, {**"xsinx"**: x \* np.sin(x),  
 **"xcosx"**: x \* np.cos(x),  
 **"arctanx"**: np.arctan(x)}, x)  
writer.close()

执行完成后，会在当前文件所在目录下生成runs目录已经对应的文件：



接着，需要在终端读取保存的数据进行展示

首先，cd到runs所在的目录，然后执行tensorboard --logdir=./runs（也可以使用绝对路径）



通过http://localhost:6006访问页面



## 5.2.TensorBoard使用

在TensorBoard中，创建图像主要用到的类为SummaryWriter，接下来学习SummaryWriter的创建：

SummaryWriter  
 功能：提供创建event file的高级接口

主要属性：  
 • log\_dir： 指定event file输出目录  
 • comment：不指定log\_dir时，目录名后缀  
 • filename\_suffix： event file文件名后缀

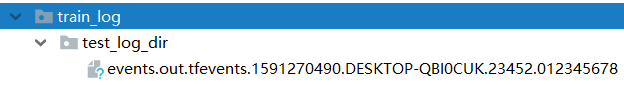
代码实现：

*# -\*- coding:utf-8 -\*-***import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**from** torch.utils.tensorboard **import** SummaryWriter  
**import** random  
**import** torch  
  
**def** set\_seed(seed=1):  
 random.seed(seed)  
 np.random.seed(seed)  
 torch.manual\_seed(seed)  
 torch.cuda.manual\_seed(seed)  
  
set\_seed(1) *# 设置随机种子  
  
# flag = 0*flag = 1  
**if** flag:

*# SummaryWriter相关属性*  
 *# 输出路径*  
 log\_dir = **"./train\_log/test\_log\_dir"**

*# 创建writer，指定路径、目录后缀（不指定路径时生效）、文件后缀*writer = SummaryWriter(log\_dir=log\_dir, comment=**'\_scalars'**, filename\_suffix=**"12345678"**)

*# 如果不设置log\_dir，默认是在当前文件所在目录创建*  
 *# writer = SummaryWriter(comment='\_scalars', filename\_suffix="12345678")* **for** x **in** range(100):  
 writer.add\_scalar(**'y=pow\_2\_x'**, 2 \*\* x, x)  
  
 writer.close()



SummaryWriter提供的方法：

1. add\_scalar()  
 功能：记录标量，画一条曲线图  
 • tag：图像的标签名，图的唯一标识  
 • scalar\_value：要记录的标量  
 • global\_step： x轴

add\_scalar(tag, scalar\_value, global\_step=None,  
 walltime=None)

2. add\_scalars()

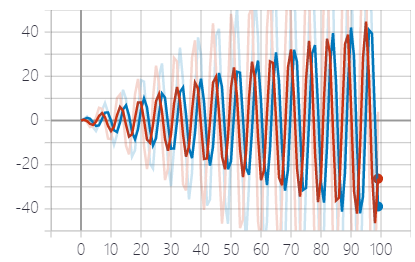
功能：记录标量，画多条曲线图  
 • main\_tag：该图的标签  
 • tag\_scalar\_dict： key是变量的tag， value是变量的值

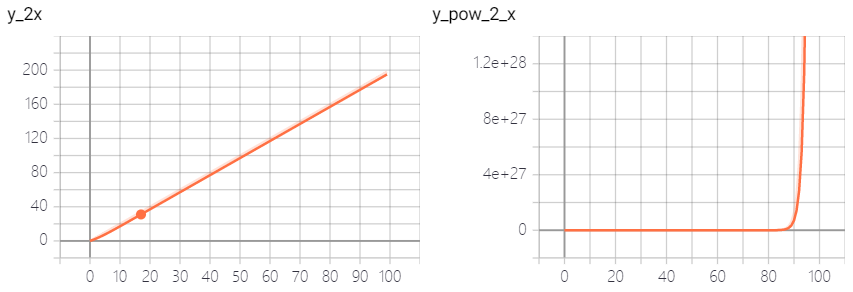
add\_scalars(main\_tag, tag\_scalar\_dict,  
 global\_step=None, walltime=None)

代码实现：

*# flag = 0*flag = 1  
**if** flag:  
 *# 设置迭代周期* max\_epoch = 100  
  
 writer = SummaryWriter(comment=**'test\_comment'**, filename\_suffix=**"test\_suffix"**)  
  
 **for** x **in** range(max\_epoch):  
 *# 绘制一条曲线* writer.add\_scalar(**'y=2x'**, x \* 2, x) *# y=2x* writer.add\_scalar(**'y=pow\_2\_x'**, 2 \*\* x, x) *# y=2\*\*x  
  
 # 一张图绘制多条曲线* writer.add\_scalars(**'data/scalar\_group'**, {**"xsinx"**: x \* np.sin(x),  
 **"xcosx"**: x \* np.cos(x)}, x)  
  
 writer.close()

启动客户端：tensorboard --logdir=./





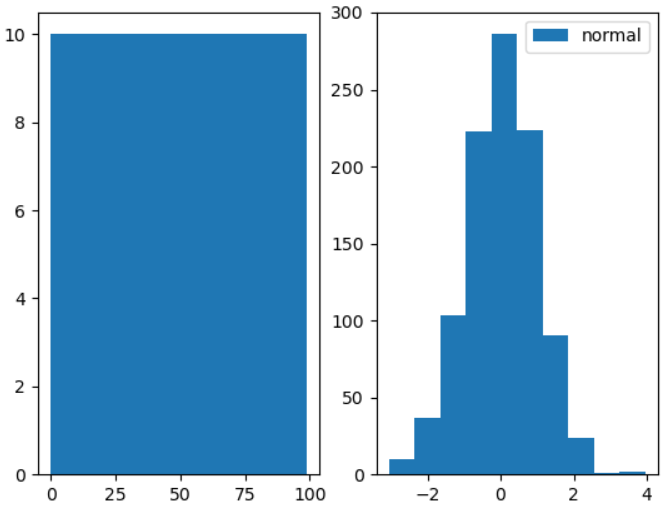
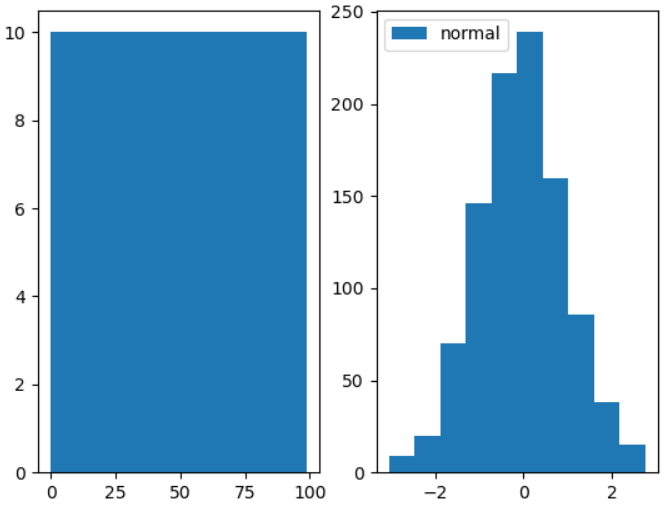
3. add\_histogram()  
 功能：统计直方图与多分位数折线图  
 • tag：图像的标签名，图的唯一标识  
 • values：要统计的参数  
 • global\_step： y轴  
 • bins：取直方图的数量bins

add\_histogram(tag, values, global\_step=None,  
 bins='tensorflow', walltime=None)

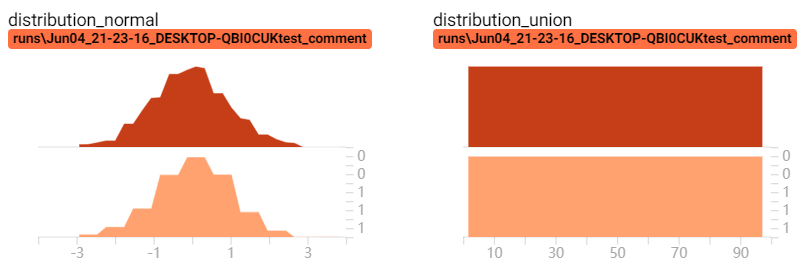
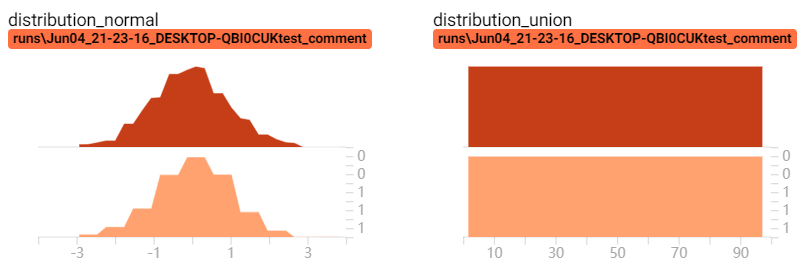
代码实现：

*# flag = 0*flag = 1  
**if** flag:  
  
 writer = SummaryWriter(comment=**'test\_comment'**, filename\_suffix=**"test\_suffix"**)  
  
 *# 绘制两次图像* **for** x **in** range(2):  
 *# 两次的随机种子不同* np.random.seed(x)  
  
 *# 生成0-99的数* data\_union = np.arange(100)  
 *# 生成1000个正太分布的数* data\_normal = np.random.normal(size=1000)  
  
 *# 绘制直方图* writer.add\_histogram(**'distribution union'**, data\_union, x, 10)  
 writer.add\_histogram(**'distribution normal'**, data\_normal, x, 10)  
  
 *# 通过plt绘制* plt.subplot(121).hist(data\_union, label=**"union"**)  
 plt.subplot(122).hist(data\_normal, label=**"normal"**)  
 plt.legend()  
 plt.show()  
  
 writer.close()

plt绘制的图像



tensorboard显示结果：左边为正太分布图像，右边为均匀分布图像

## 5.3.图像可视化

1. add\_image()  
 功能：记录图像  
 • tag：图像的标签名，图的唯一标识  
 • img\_tensor：图像数据，注意尺度，如果不超过1，会被放大到0-255  
 • global\_step： x轴  
 • dataformats：数据形式， CHW， HWC， HW

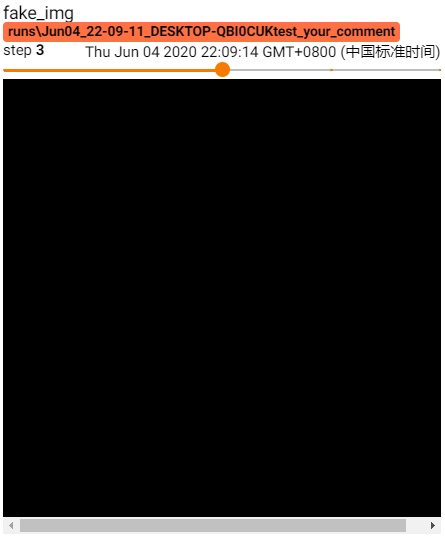
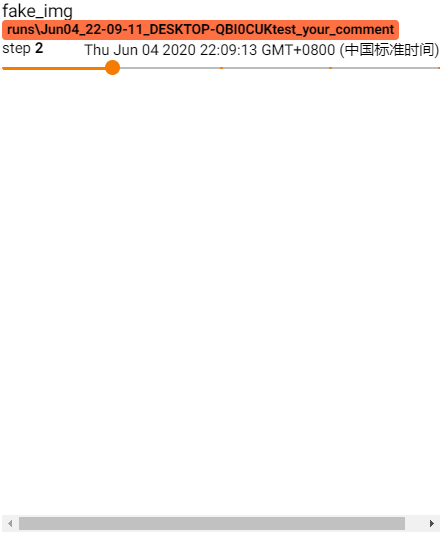
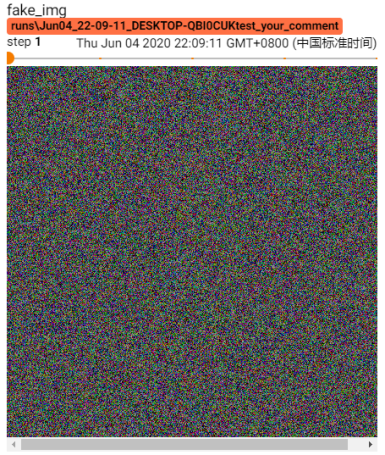
add\_image(tag, img\_tensor, global\_step=None,  
 walltime=None, dataformats='CHW')

代码实现：

*# -\*- coding:utf-8 -\*-***import** os  
**import** torch  
**import** time  
**import** random  
**import** numpy **as** np  
**import** torchvision.transforms **as** transforms  
**import** torchvision.utils **as** vutils

**import** torchvision.models **as** models  
**import** torch.nn **as** nn  
**from** tools.my\_dataset **import** RMBDataset  
**from** torch.utils.tensorboard **import** SummaryWriter  
**from** torch.utils.data **import** DataLoader  
**from** model.lenet **import** LeNet  
**from** PIL **import** Image  
  
**def** set\_seed(seed=1):  
 random.seed(seed)  
 np.random.seed(seed)  
 torch.manual\_seed(seed)  
 torch.cuda.manual\_seed(seed)  
  
set\_seed(1) *# 设置随机种子*flag = 0  
*# flag = 1***if** flag:  
  
 writer = SummaryWriter(comment=**'test\_your\_comment'**, filename\_suffix=**"\_test\_your\_filename\_suffix"**)  
  
 *# 图像一：构建标准正太分布3×512×512的图像* fake\_img = torch.randn(3, 512, 512)  
 writer.add\_image(**"fake\_img"**, fake\_img, 1)  
 time.sleep(1)  
  
 *# 图像二：构建全是1的3×512×512的图像，由于没有超过1，尺度放大到255* fake\_img = torch.ones(3, 512, 512)  
 time.sleep(1)  
 writer.add\_image(**"fake\_img"**, fake\_img, 2)  
  
 *# 图像三：构建全是1.1的3×512×512的图像，尺度超过1，所以默认还是1.1* fake\_img = torch.ones(3, 512, 512) \* 1.1  
 time.sleep(1)  
 writer.add\_image(**"fake\_img"**, fake\_img, 3)  
  
 *# 图像四：构建[0, 1)之间均匀分布的512×512的图像，指定数据形式为灰度图* fake\_img = torch.rand(512, 512)  
 writer.add\_image(**"fake\_img"**, fake\_img, 4, dataformats=**"HW"**)  
  
 *# 图像五：构建[0, 1)之间均匀分布的512×512×3的图像，指定图像形式为通道在最后* fake\_img = torch.rand(512, 512, 3)  
 writer.add\_image(**"fake\_img"**, fake\_img, 5, dataformats=**"HWC"**)  
  
 writer.close()

执行完成后，启动客户端，通过拖动图像上面的图像条（即global\_step）查看图像





2.torchvision.utils.make\_grid  
 功能：制作网格图像，即类似plt中的子图  
 • tensor：图像数据, B\*C\*H\*W形式  
 • nrow：每行图像的数量  
 • padding：图像间距（像素单位）  
 • normalize：是否将像素值标准化，如果不是0-255则需要标准化为0-255  
 • range：标准化范围，将这个范围缩放到0-255  
 • scale\_each：是否单张图维度标准化  
 • pad\_value： padding的像素值

make\_grid(tensor, nrow=8, padding=2,  
 normalize=False, range=None, scale\_each=False,  
 pad\_value=0)

代码实现：

*# flag = 0*flag = 1  
**if** flag:  
 writer = SummaryWriter(comment=**'test\_your\_comment'**, filename\_suffix=**"\_test\_your\_filename\_suffix"**)  
  
 *# 使用之前的rmb数据集* split\_dir = os.path.join(**".."**, **".."**, **"data"**, **"rmb\_split"**)  
 train\_dir = os.path.join(split\_dir, **"train"**)  
  
 *# 对图像预处理，大小进行resize，并且ToTensor将尺度缩放为0-1* transform\_compose = transforms.Compose([transforms.Resize((32, 64)), transforms.ToTensor()])  
 *# 构建dataset，使用之前构建好的* train\_data = RMBDataset(data\_dir=train\_dir, transform=transform\_compose)  
 *# 加载数据，并设置批大小和打乱顺序* train\_loader = DataLoader(dataset=train\_data, batch\_size=16, shuffle=**True**)  
 *# 通过iter()方法将train\_loader转换为可迭代对象，通过next获取一条数据* data\_batch, label\_batch = next(iter(train\_loader))  
  
 *# 通过make\_grid()方法创建图像网格* img\_grid = vutils.make\_grid(data\_batch, nrow=4)  
 writer.add\_image(**"input img"**, img\_grid, 0)  
  
 writer.close()

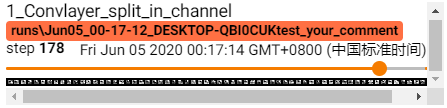
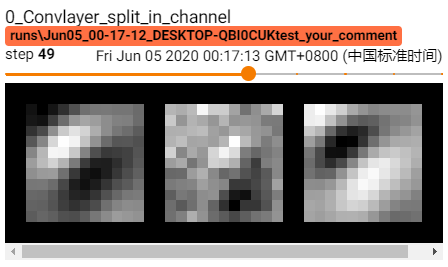


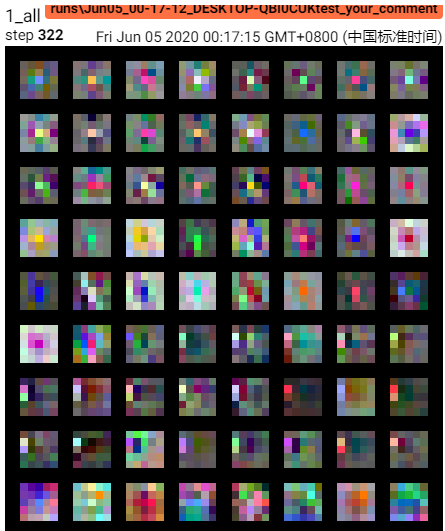
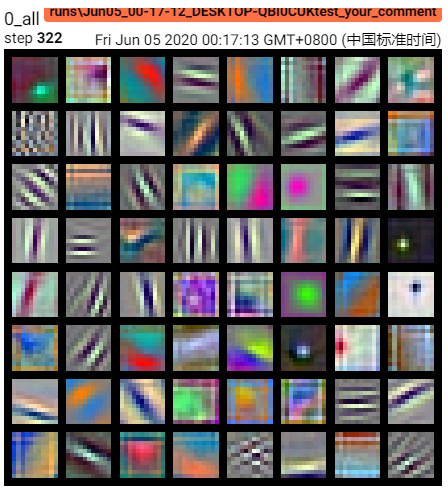
利用make\_grid绘制AlexNet模型的卷积核图像

代码实现：

*# flag = 0*flag = 1  
**if** flag:  
 *# 对AlexNet模型第一个卷积层的卷积核进行可视化* writer = SummaryWriter(comment=**'test\_your\_comment'**, filename\_suffix=**"\_test\_your\_filename\_suffix"**)  
  
 *# 创建AlexNet模型，pretrained=True表示使用已经训练好的AlexNet模型  
 # 这个模型需要在线下载，可以使用已经下载好的，放入对应的目录即可，如C:\Users\用户名\.cache\torch\checkpoints* alexnet = models.alexnet(pretrained=**True**)  
  
 *# 用来指示当前是第几个卷积层* kernel\_num = -1  
 *# 最大可视化卷积层，这里只展示第0和1两个卷积层* vis\_max = 1  
  
 *# 遍历AlexNet模型中所有的子模型* **for** sub\_module **in** alexnet.modules():  
 *# 判断当前的子模型是否为卷积层* **if** isinstance(sub\_module, nn.Conv2d):  
 *# 如果是卷积层，则kernel\_num+1* kernel\_num += 1  
 *# 判断则kernel\_num是否大于vis\_max，如果已经大于，则跳出循环* **if** kernel\_num > vis\_max:  
 **break** *# 获取当前卷积层的权重参数* kernels = sub\_module.weight  
 *# 卷积核的shape，4维，分别表示卷积核数量，通道数，高、宽* c\_out, c\_int, k\_w, k\_h = tuple(kernels.shape)  
  
 *# 循环次数为卷积核数量，有64个核函数* **for** o\_idx **in** range(c\_out):  
 *# 遍历每个卷积核，默认的shape为CHW，其中C=3，这里拓展一个维度，变为BCHW，其中B为之前的C，新的C为1* kernel\_idx = kernels[o\_idx, :, :, :].unsqueeze(1) *# 相当于三通道每个通道为一个batch  
 # 创建卷积核的网格图像，进行标准化，一行图像数为卷积核通道数，即一行三个图像* kernel\_grid = vutils.make\_grid(kernel\_idx, normalize=**True**, scale\_each=**True**, nrow=c\_int)  
 *# 输出图像，x轴为卷积核数量* writer.add\_image(**'{}\_Convlayer\_split\_in\_channel'**.format(kernel\_num), kernel\_grid, global\_step=o\_idx)  
  
 *# view()函数返回和原tensor数据个数相同，但size不同的tensor，这里如果数据不符合BCHW，则做变换* kernel\_all = kernels.view(-1, 3, k\_h, k\_w) *# 3, h, w  
 # 总共有64个核函数，所以设置每行8个* kernel\_grid = vutils.make\_grid(kernel\_all, normalize=**True**, scale\_each=**True**, nrow=8)  
 writer.add\_image(**'{}\_all'**.format(kernel\_num), kernel\_grid, global\_step=322)  
  
 print(**"{}\_convlayer shape:{}"**.format(kernel\_num, tuple(kernels.shape)))  
  
 writer.close()

|  |
| --- |
| 0\_convlayer shape:(64, 3, 11, 11)  1\_convlayer shape:(192, 64, 5, 5) |



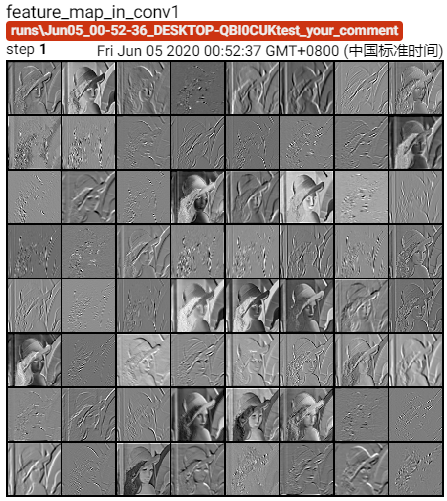


利用AlexNet模型的第一层卷积核对已有图像进行卷积，输出卷积后的特征图

代码实现：

*# flag = 0*flag = 1  
**if** flag:  
 writer = SummaryWriter(comment=**'test\_your\_comment'**, filename\_suffix=**"\_test\_your\_filename\_suffix"**)  
  
 *# 图片路径* path\_img = **"./lena.png"** *# your path to image  
  
 # 图片标准化时的均值与方差* normMean = [0.49139968, 0.48215827, 0.44653124]  
 normStd = [0.24703233, 0.24348505, 0.26158768]  
  
 *# 图片预处理，分别resize转换大小，ToTensor变张量并归一，Normalize标准化* img\_transforms = transforms.Compose([  
 transforms.Resize((224, 224)),  
 transforms.ToTensor(),  
 transforms.Normalize(normMean, normStd)  
 ])  
  
 *# 以RGB形式读取图片* img\_pil = Image.open(path\_img).convert(**'RGB'**)  
 **if** img\_transforms **is not None**:  
 img\_tensor = img\_transforms(img\_pil)  
 *# 扩展第一维度，由chw转换为bchw* img\_tensor.unsqueeze\_(0) *# chw --> bchw  
  
 # 模型* alexnet = models.alexnet(pretrained=**True**)  
  
 *# 获取alexnet模型的第一层，为Conv2d卷积层模型* convlayer1 = alexnet.features[0]  
  
 *# 利用第一层的卷积层对图像进行卷积处理* fmap\_1 = convlayer1(img\_tensor)  
  
 *# 预处理* fmap\_1.transpose\_(0, 1) *# bchw=(1, 64, 55, 55) --> (64, 1, 55, 55)* fmap\_1\_grid = vutils.make\_grid(fmap\_1, normalize=**True**, scale\_each=**True**, nrow=8)  
  
 writer.add\_image(**'feature map in conv1'**, fmap\_1\_grid, global\_step=1)  
 writer.close()

原始图像与卷积后的特征图像比较



## 5.4.Hook函数

Pytorch在进行完一次反向传播后，出于节省内存的考虑，只会存储叶子节点的梯度信息，并不会存储中间变量的梯度信息。然而有些时候我们又不得不使用中间变量的梯度信息完成某些工作（如获取中间层的梯度，获取中间层的特征图），这时候hook()函数就可以派上用场了。

Hook函数机制：不改变主体，实现额外功能，hook()函数翻译成中文叫做钩子函数，这非常形象：我们的主任务是反向传播更新梯度，而钩子函数就是挂在主任务上的辅任务

主要有四种钩子函数：

1. torch.Tensor.register\_hook  
 2. torch.nn.Module.register\_forward\_hook  
 3. torch.nn.Module.register\_forward\_pre\_hook  
 4. torch.nn.Module.register\_backward\_hook

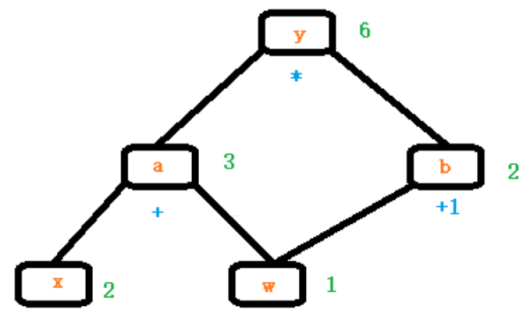
接下来分别对他们进行介绍：

1. Tensor.register\_hook  
 功能：注册一个反向传播hook函数

Hook函数仅一个输入参数，为张量的梯度

hook(grad)

例如，在之前的计算梯度时：



当进行反向传播时，a和b的梯度将会释放，如何保存a和b的梯度呢？可以利用hook函数进行保存。同时，hook函数也可以修改梯度。

代码实现：

*# -\*- coding:utf-8 -\*-***import** torch  
**import** torch.nn **as** nn  
**import** random  
**import** numpy **as** np  
  
**def** set\_seed(seed=1):  
 random.seed(seed)  
 np.random.seed(seed)  
 torch.manual\_seed(seed)  
 torch.cuda.manual\_seed(seed)  
set\_seed(1) *# 设置随机种子  
  
# flag = 0*flag = 1  
**if** flag:  
  
 *# 构建计算图* w = torch.tensor([1.], requires\_grad=**True**)  
 x = torch.tensor([2.], requires\_grad=**True**)  
 a = torch.add(w, x)  
 b = torch.add(w, 1)  
 y = torch.mul(a, b)  
  
 *# 存放a的梯度的list* a\_grad = list()  
  
 *# 定义一个hook函数，将参数的梯度加入到a的梯度的list中* **def** grad\_hook(grad):  
 a\_grad.append(grad)

*# 也可以对梯度进行修改   
 # grad \*= 10*

*# 将hook函数注册到张量a上，返回一个handle，它有一个方法handle.remove()，用来移除hook* handle = a.register\_hook(grad\_hook)  
  
 *# 进行反向传播* y.backward()  
  
 *# 打印梯度* print(**"gradient:"**, w.grad, x.grad, a.grad, b.grad, y.grad)  
 *# 打印a\_grad list中是否存放了梯度* print(**"a\_grad: "**, a\_grad)  
 *# 移除hook* handle.remove()

|  |
| --- |
| gradient: tensor([5.]) tensor([2.]) None None None  a\_grad: [tensor([2.])] |

2. Module.register\_forward\_hook  
 功能：注册module的前向传播hook函数，由于前向传播中特征图会释放，可以在hook函数中获取  
 • module: 当前网络层  
 • input：当前网络层输入数据  
 • output：当前网络层输出数据

hook(module, input, output)

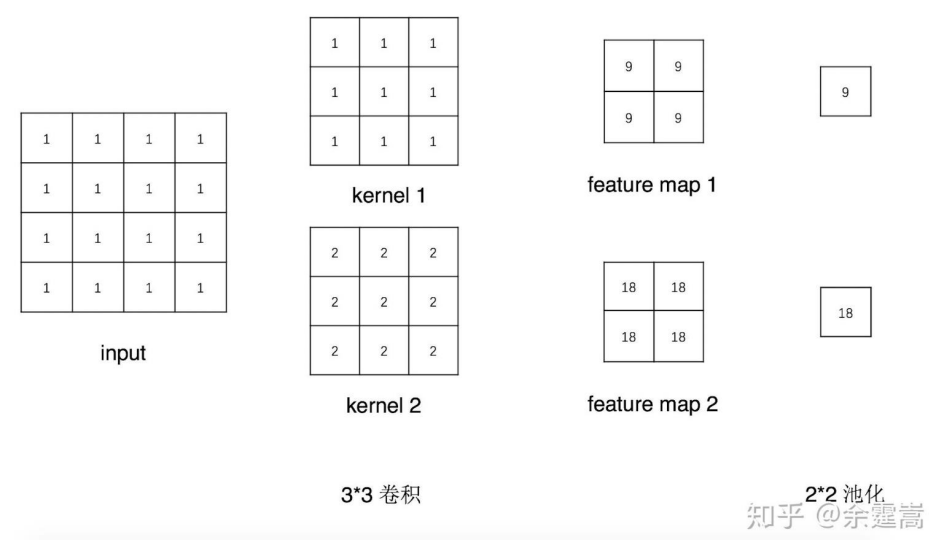
3. Module.register\_forward\_pre\_hook  
 功能：注册module前向传播前的hook函数，此时模型还没有计算数据，所以无法获取计算后的输出，只能获取模型和输入数据相关信息  
 • module: 当前网络层  
 • input：当前网络层输入数据

hook(module, input)

4. Module.register\_backward\_hook  
 功能：注册module反向传播的hook函数  
 • module: 当前网络层  
 • grad\_input：当前网络层输入梯度数据  
 • grad\_output：当前网络层输出梯度数据

hook(module, grad\_input, grad\_output)

案例：对于下面4×4全是1的数据，通过两个3×3的卷积核进行卷积，得到卷积后的特征图，然后再进行池化，最终得到输出结果，分别用上面的三个hook函数记录相关的数据信息



代码实现：

*# flag = 0*flag = 1  
**if** flag:  
 *# 定义模型* **class** Net(nn.Module):  
 **def** \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 *# 定义卷积层，输入通道为1，输出通道为2，卷积核尺寸为3* self.conv1 = nn.Conv2d(1, 2, 3)  
 *# 定义池化层，池化核尺寸为2，步长为2* self.pool1 = nn.MaxPool2d(2, 2)  
  
 *# 组装网络结构* **def** forward(self, x):  
 x = self.conv1(x)  
 x = self.pool1(x)  
 **return** x  
  
 *# 定义前向传播hook函数* **def** forward\_hook(module, data\_input, data\_output):  
 *# 将输出数据加入到fmap\_block的list中* fmap\_block.append(data\_output)  
 *# 将输入数据加入到input\_block的list中* input\_block.append(data\_input)  
  
 *# 定义前向传播前的hook函数* **def** forward\_pre\_hook(module, data\_input):  
 *# 打印前向传播前的输入数据* print(**"forward\_pre\_hook input:{}"**.format(data\_input))  
  
 *# 定义反向传播hook函数* **def** backward\_hook(module, grad\_input, grad\_output):  
 *# 分别打印反向传播的输入梯度和输出梯度* print(**"backward hook input:{}"**.format(grad\_input))  
 print(**"backward hook output:{}"**.format(grad\_output))  
  
 *# 初始化网络* net = Net()  
 *# 初始化权重，第一个卷积核全是1，第二个卷积核全是2，bias为0* net.conv1.weight[0].detach().fill\_(1)  
 net.conv1.weight[1].detach().fill\_(2)  
 net.conv1.bias.data.detach().zero\_()  
  
 *# 定义两个list* fmap\_block = list()  
 input\_block = list()  
 *# 将hook注册到第一个卷积层* net.conv1.register\_forward\_hook(forward\_hook)  
 net.conv1.register\_forward\_pre\_hook(forward\_pre\_hook)  
 *# 将反向传播的hook注册到池化层* net.pool1.register\_backward\_hook(backward\_hook)  
  
  
 *# 构建数据了，4×4的全是1的数据* fake\_img = torch.ones((1, 1, 4, 4)) *# batch size \* channel \* H \* W  
 # 将数据交给net模型得到输出结果* output = net(fake\_img)  
  
 *# 构建loss，并对loss进行反向传播* loss\_fnc = nn.L1Loss()  
 *# 生成目标值，与输出同shape的标准正太分布数据* target = torch.randn\_like(output)  
 *# 计算loss* loss = loss\_fnc(target, output)  
 *# 对损失函数进行反向传播* loss.backward()  
  
 *# 打印输出的shape和输出的value值* print(**"output shape: {}\noutput value: {}\n"**.format(output.shape, output))  
 *# 打印卷积后输出的特征的shape和value值* print(**"feature maps shape: {}\noutput value: {}\n"**.format(fmap\_block[0].shape, fmap\_block[0]))  
 *# 打印卷积前输入数据的shape和value值* print(**"input shape: {}\ninput value: {}"**.format(input\_block[0][0].shape, input\_block[0]))

|  |
| --- |
| forward\_pre\_hook input:(tensor([[[[1., 1., 1., 1.],  [1., 1., 1., 1.],  [1., 1., 1., 1.],  [1., 1., 1., 1.]]]]),)  backward hook input:(tensor([[[[0.5000, 0.0000],  [0.0000, 0.0000]],  [[0.5000, 0.0000],  [0.0000, 0.0000]]]]),)  backward hook output:(tensor([[[[0.5000]],  [[0.5000]]]]),)  output shape: torch.Size([1, 2, 1, 1])  output value: tensor([[[[ 9.]],  [[18.]]]], grad\_fn=<MaxPool2DWithIndicesBackward>)  feature maps shape: torch.Size([1, 2, 2, 2])  output value: tensor([[[[ 9., 9.],  [ 9., 9.]],  [[18., 18.],  [18., 18.]]]], grad\_fn=<ThnnConv2DBackward>)  input shape: torch.Size([1, 1, 4, 4])  input value: (tensor([[[[1., 1., 1., 1.],  [1., 1., 1., 1.],  [1., 1., 1., 1.],  [1., 1., 1., 1.]]]]),) |

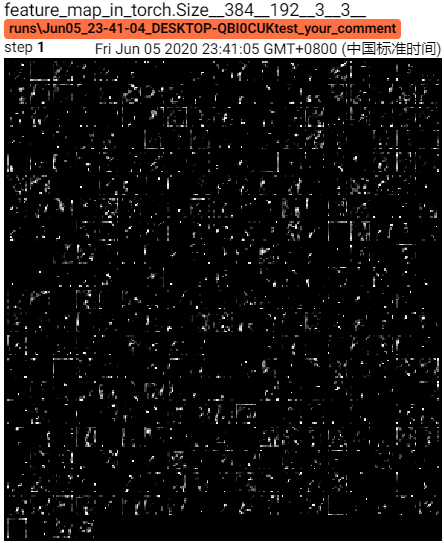
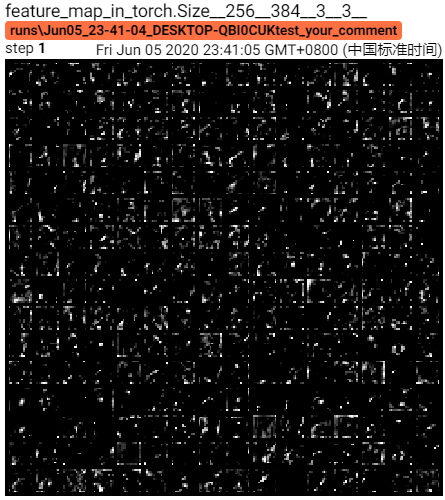
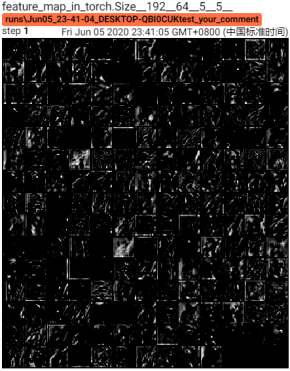
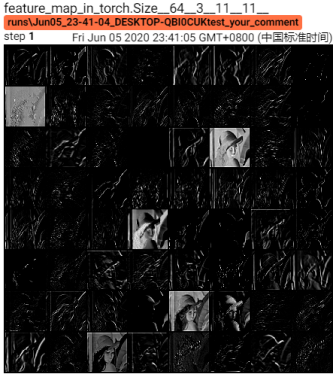
接下来，通过hook实现对alexnet模型所有卷积层的可视化

代码实现：

*# -\*- coding:utf-8 -\*-***import** torch  
**import** torch.nn **as** nn  
**import** numpy **as** np  
**from** PIL **import** Image  
**import** random  
**import** torchvision.transforms **as** transforms  
**import** torchvision.utils **as** vutils  
**from** torch.utils.tensorboard **import** SummaryWriter  
**import** torchvision.models **as** models  
  
**def** set\_seed(seed=1):  
 random.seed(seed)  
 np.random.seed(seed)  
 torch.manual\_seed(seed)  
 torch.cuda.manual\_seed(seed)  
  
set\_seed(1) *# 设置随机种子*writer = SummaryWriter(comment=**'test\_your\_comment'**, filename\_suffix=**"\_test\_your\_filename\_suffix"**)  
  
*# 设置图片路径*path\_img = **"./lena.png"***# 设置图片标准化参数*normMean = [0.49139968, 0.48215827, 0.44653124]  
normStd = [0.24703233, 0.24348505, 0.26158768]  
  
*# 对图片进行预处理*img\_transforms = transforms.Compose([  
 transforms.Resize((224, 224)),  
 transforms.ToTensor(),  
 transforms.Normalize(normMean, normStd)  
])  
  
*# 加载图片并预处理*img\_pil = Image.open(path\_img).convert(**'RGB'**)  
**if** img\_transforms **is not None**:  
 img\_tensor = img\_transforms(img\_pil)  
*# 扩展维度，chw --> bchw*img\_tensor.unsqueeze\_(0)  
  
*# 创建alexnet模型*alexnet = models.alexnet(pretrained=**True**)  
  
*# 创建一个字典，用于存储所有卷积层的特征图，key为每个卷积层*fmap\_dict = dict()  
  
*# alexnet.named\_modules()方法返回模型所有的子网络层的名称和模型***for** name, sub\_module **in** alexnet.named\_modules():  
 *# 判断是否为卷积层* **if** isinstance(sub\_module, nn.Conv2d):  
 *# 将卷积层的权重的shape作为字典的key* key\_name = str(sub\_module.weight.shape)  
 *# 利用setdefault方法进行查找，如果键不存在，则设置值为list()* fmap\_dict.setdefault(key\_name, list())  
  
 *# name为features.0、features.1、...、classifier.0、...的形式* n1, n2 = name.split(**"."**)  
  
 *# 定义hook函数，参数为：模型，模型的输入，模型的输出* **def** hook\_func(m, i, o):  
 *# 获取模型权重shape，即字典的key* key\_name = str(m.weight.shape)  
 *# 将模型的输出（特征）添加到字典中* fmap\_dict[key\_name].append(o)  
  
 *# \_modules[n1]表示获取模型容器，\_modules[n2]表示获取容器中的第几个模型，并为此模型注册hook* alexnet.\_modules[n1].\_modules[n2].register\_forward\_hook(hook\_func)  
  
*# 将图片交给alexnet模型并得到输出*output = alexnet(img\_tensor)  
  
*# 遍历字典并拿到key和value***for** layer\_name, fmap\_list **in** fmap\_dict.items():  
 *# 获取特征* fmap = fmap\_list[0]  
 *# 调换维度为0和1的位置，如bchw=(1, 64, 55, 55) --> (64, 1, 55, 55)* fmap.transpose\_(0, 1)  
  
 *# 设置每行图片数量为批大小的开方个* nrow = int(np.sqrt(fmap.shape[0]))  
 *# 创建网格图* fmap\_grid = vutils.make\_grid(fmap, normalize=**True**, scale\_each=**True**, nrow=nrow)  
 writer.add\_image(**'feature map in {}'**.format(layer\_name), fmap\_grid, global\_step=1)

执行完成后，启动客户端：tensorboard --logdir=./

得到特征图如下：



发现其特征图与之前只用一层时有所不同，原因是当时并没有使用激活函数，而这里默认有激活函数，所以图像有区别。