NiN(network in network)

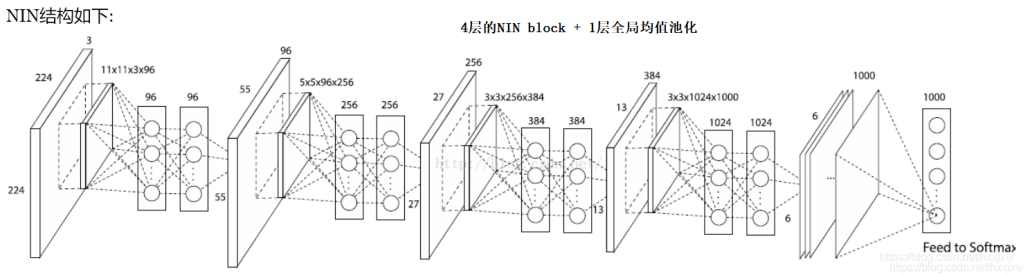
一、简介

> NIN(NetWork In NetWork) 是出自新加坡国立大写 2014 年的论文 *"Network In Network"* ,NIN 改进了传统的 CNN，采用了少量参数就取得了超过 AlexNet 的性能，AlexNet 网络参数大小是 230M，NIN 只需要 29M，此模型后来先后被 Inception 与 ResNet 等所借鉴。
>
> NIN 的第一个 N 指 mlpconv,第二个 N 指整个深度网络结构，即整个深度网络是由多个 mlpconv 构成的。NIN 的结构有两个创新点,都是比较有里程碑意义的,分别为 **MLPconv** 以及**全局平均池化。**

conventional 的卷积层 可以认为是 linear model , 为什么呢，因为 局部接收域上的每一个 tile 与 卷积核进行加权求和，然后接一个激活函数；它的 abstraction 的能力不够， 对处理线性可分的的 concept 也许是可以的，但是更复杂的 concepts 它有能力有点不够了，所以呢，需要引入 more potent 的非线性函数；

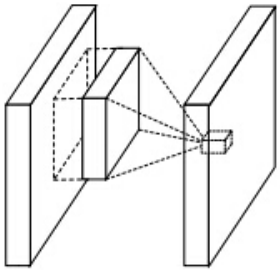
一般卷积操作可以看成特征的提取操作，而一般卷积一层只相当于一个线性操作，所以其只能提取出线性特征。所以该作者就想能否在卷积层后也加入一个 MLP 使得每层卷积操作能够提取非线性特征。

二、网络结构

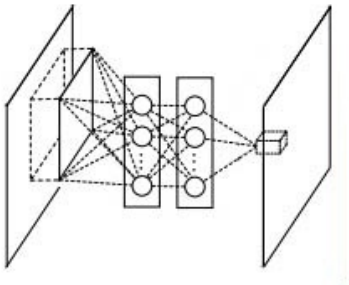


此网络结构总计 4 层：3mlpconv + 1global_average_pooling

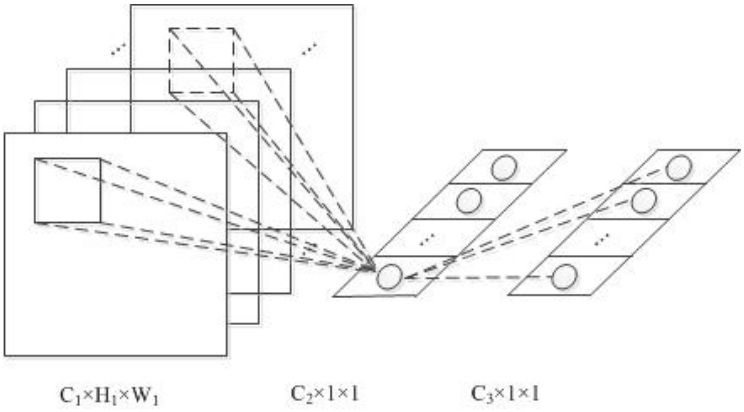
- **传统的 convolution 层**



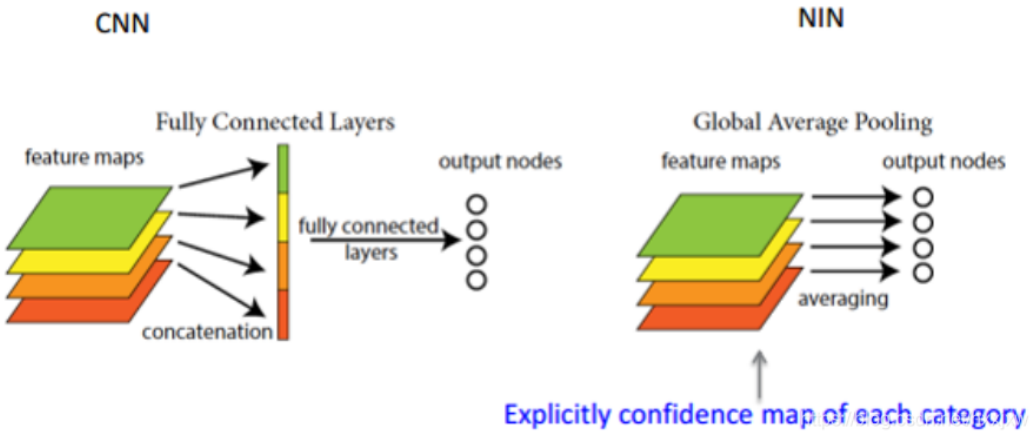
- **单通道 mlpconv 层**



- **跨通道 mlpconv 层（cccp 层）**



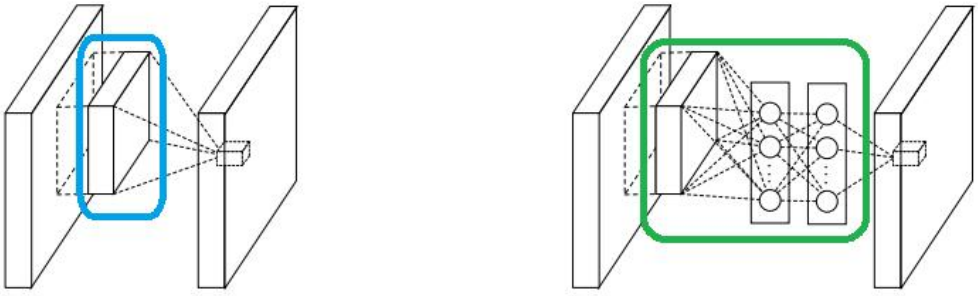
> 在跨通道（cross channel,cross feature map）情况下，mlpconv 等价于卷积层+1×1 卷积层，所以此时 mlpconv 层也叫 cccp 层（cascaded cross channel parametric pooling）。



NiN 块是 NiN 中的基础块。它由一个卷积层加两个充当全连接层的卷积层串联而而成。其中第一个卷积层的超参数可以自行设置，而而第二和第三个卷积层的超参数一般是固定的

1.MLP Convolution Layers

先前 CNN 中 **简单的 线性卷积层** [蓝框部分] 被替换为了 **多层感知机**（MLP，多层 全连接层 和非线性函数的组合） [绿框部分]



(a) Linear convolution layer (b) Mlpconv layer

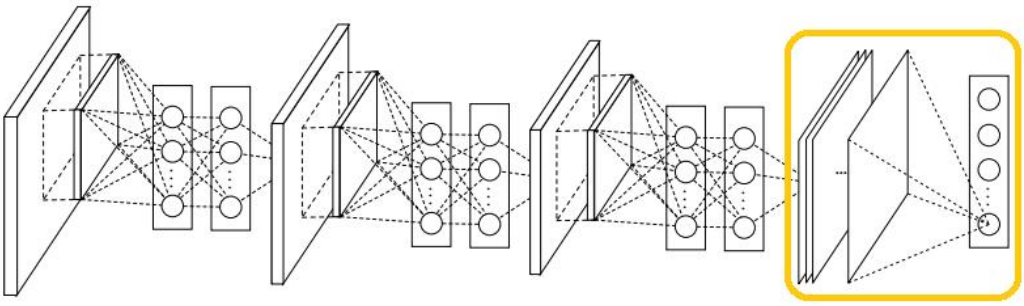
由上图看出，左图为**简单的卷积层网络**，右图为 **mlpconv（卷积层 + 1x1 卷积 + ReLu 函数组成）**

优点：

1. **提供了网络层间映射的一种新可能；**
2. **增加了网络卷积层的非线性能力。**

2.全局平均池化 GAP

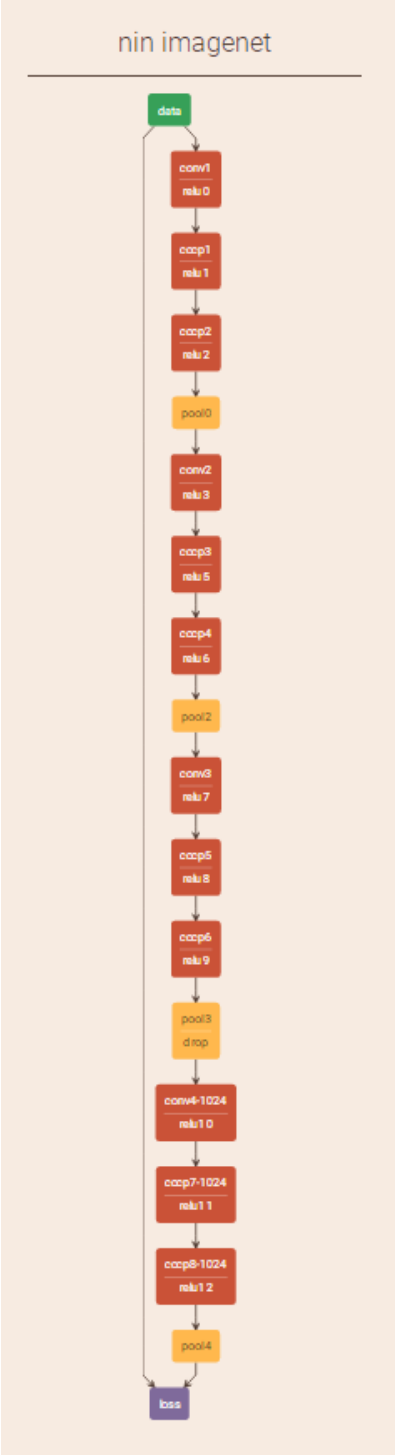
先前 CNN 中的 **全连接层** 被替换为 全局池化层（global average pooling）：



解释一: 最后改为全局池化层，含义为 1 张特征图转化为 1 个类别进行输出，如第一张图最后有 **1000 个特征图**,最后通过 softmax 对应 1000 个不同的类别输出。

解释二： 假设分类任务共有 C 个类别。先前 CNN 中最后一层为特征图层数共计 N 的全连接层，要映射到 C 个类别上；改为全局池化层后，最后一层为特征图层数共计 C 的全局池化层，恰好对应分类任务的 C 个类别，这样一来，就会有更好的可解释性了。

3.picture



三、两个创新点

1.MLP Convolution Layers（创新点 1）

> 在跨通道（cross channel,cross feature map）情况下，mlpconv 等价于卷积层+1×1 卷积层，所以此时 mlpconv 层也叫 cccp 层（cascaded cross channel parametric pooling）。

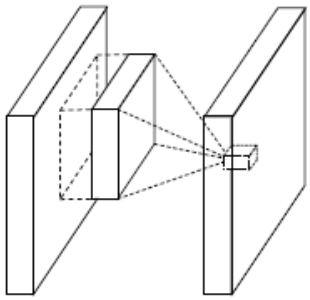
一言以蔽之，所谓 **MLPConv 其实就是在常规卷积（感受野大于 1 的）后接若干 1x1 卷积**，每个特征图视为一个神经元，特征图通过 1x1 卷积就类似多个神经元线性组合，这样就像是 MLP（多层感知机）了，这是文章最大的创新点，也就是 Network in Network（网络中内嵌微型网络）。

论文提到一开始并不知道潜在特征的分布，因此用一个通用的函数逼近器提取局部块特征，这样可以尽可能逼近潜在特征的抽象表示。

径向基（Radial basis network）和 从多层感知机（multilayer perceptron）是两种通用的函数逼近器，作者选择了多层感知机，因为**多层感知器与卷积神经网络的结构一样，都是通过反向传播训练。**其次多层感知器本身就是一个深度模型，符合特征再利用的原则。

普通卷积层 （感受野大于 1）及文中提到的 GLM(generalized linear model)相当于单层网络，抽象能力有限，其计算公式和示意图如下：

$$f_{i,j,k} = \max(w_k^T x_{i,j}, 0).$$

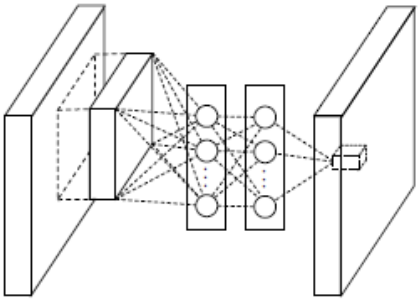


(a) Linear convolution layer

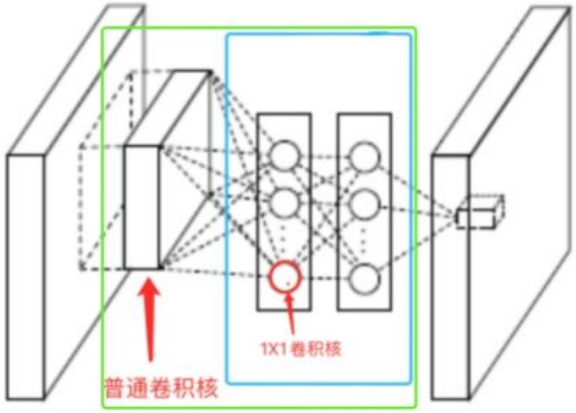
为了提高特征的抽象表达能力，作者用 MLPConv 代替了 GLM，计算公式为：

$$\begin{aligned} f_{i,j,k_1}^1 &= \max({w_{k_1}^1}^T x_{i,j} + b_{k_1}, 0). \\ &\vdots \\ f_{i,j,k_n}^n &= \max({w_{k_n}^n}^T f_{i,j}^{n-1} + b_{k_n}, 0). \end{aligned}$$

n 为网络层数，第一层为线性卷积层（卷积核尺寸大于 1），后面的为 1x1 卷积。MLPConv 其实就是在常规卷积（卷积核大小大于 1）后接若干层若干个 1x1 卷积，这些 **1x1 卷积如果看成神经元的话**,这样就像是 MLP（多层感知机）了，也就是说这里有一个微型的神经网络,而且还是全连接,用这个微型的神经网络也就是 MLP 来对特征进行提取, 这也是文章的最大创新点。

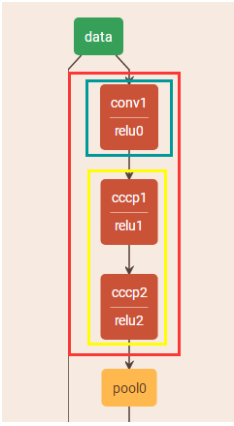


(b) Mlpconv layer



(b) Mlpconv layer

基于 AlexNet 修改的 NIN 中 MLPConv 为三层，conv1 常规卷积，occp1 和 occp2 为 1x1 卷积。



在当时作者应该是第一个使用 1x1 卷积的，具有划时代的意义，之后的 Googlenet 借鉴了 1*1 卷积，还专门致谢过这篇论文，现在很多优秀的网络结构都离不开 1x1 卷积，ResNet、ResNext、SqueezeNet、MobileNetv1-3、ShuffleNetv1-2 等等。

1x1 卷积作为 NIN 函数逼近器基本单元，除了增强了网络局部模块的抽象表达能力外，在现在看来还可以实现跨通道特征融合和通道升维降维。

2、Global Average Pooling（创新点 2）全局池化

> 传统卷积神经网络在网络的浅层进行卷积运算。对于分类任务，最后一个卷积层得到的特征图被向量化（flatten）然后送入全连接层，接一个 softmax 逻辑回归层。这种结构将卷积结构与传统神经网络分类器连接起来，卷积层作为特征提取器，得到的特征用传统神经网络进行分类。全连接层参数量是非常庞大的，模型通常会容

> 易过拟合，针对这个问题，Hinton 提出 Dropout 方法来提高泛化能力，但是全连接的计算量依旧很大（想想 4096+4096……）。
>
> 基于此，做法即移除全连接层，论文提出用全局平均池化代替全连接层，具体做法是对最后一层的特征图进行平均池化，得到的结果向量直接输入 softmax 层。这样做好处之一是使得特征图与分类任务直接关联，另一个优点是全局平均池化不需要优化额外的模型参数，因此模型大小和计算量较全连接大大减少，并且可以避免过拟合。

什么是全局池化?

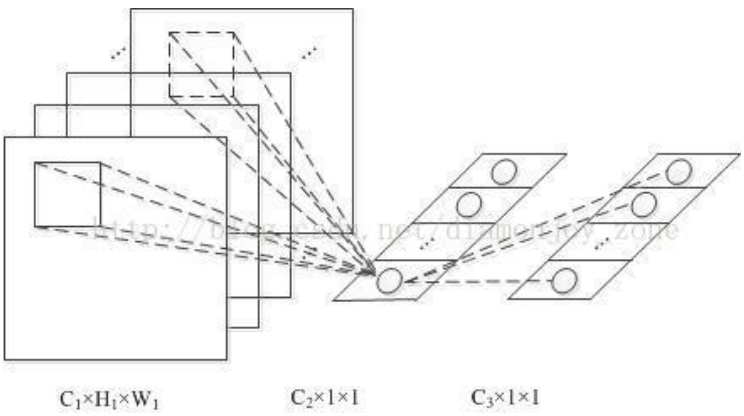
“global pooling”就是 pooling 的 **滑窗 size** 和整张 feature map 的 size 一样大。这样，每个 *W*×*H*×*C* 的 feature map 输入就会被转化为 1 × 1 × C 1×1×*C* 输出。因此，其实也等同于每个位置权重都为 1/(*W*×*H*) 的 FC 层操作。
等同于输入一个 tensor，输出一根 vector（向量）。
“global pooling”在滑窗内的具体 pooling 方法可以是任意的，所以就会被细分为“global avg pooling”、“global max pooling”等。
由于传统的 pooling 太过粗暴，操作复杂，目前业界已经逐渐放弃了对 pooling 的使用。**替代方案** 如下：

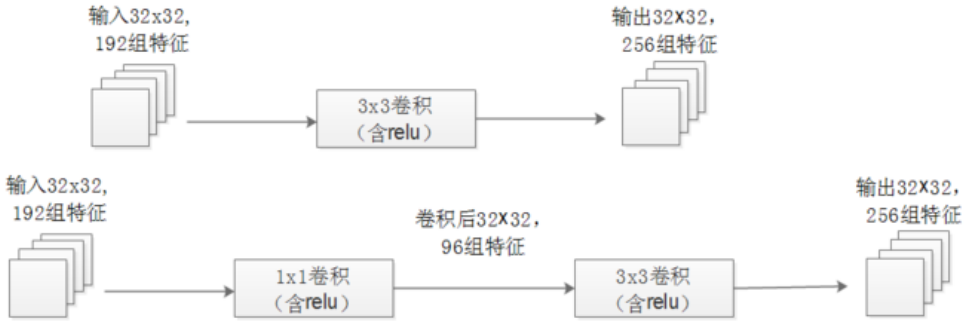
- 采用 **Global Pooling** 以简化计算；
- 增大 conv 的 **stride** 以免去附加的 pooling 操作。

四、两个重要特性

特性一 1*1 卷积核

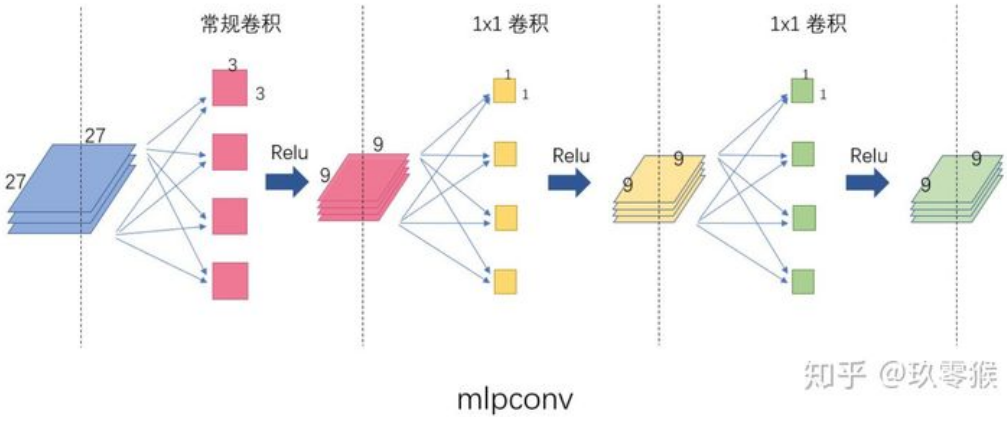
> 卷积核在 CNN 中经常被用到，一般常见的是 3×3 的或者 5×5 的。一般 1×1 的卷积核本质上并没有对图像做什么聚合操作，以为就是同一个 w 去乘以原图像上的每一个像素点，相当于做了一个 scaling。1×1 卷积核最初是在 Network in Network 这个网络结构中提出来的。它用了比 AlexNet 更少的参数，达到了跟其一样的效果。





例1： 使用1x1卷积进行降维，降低了计算复杂度。图2中间3x3卷积和5x5卷积前的1x1卷积都起到了这个作用。当某个卷积层输入的特征数较多，对这个输入进行卷积运算将产生巨大的计算量；如果对输入先进行降维，减少特征数后再做卷积计算量就会显著减少。下图是优化前后两种方案的乘法次数比较，同样是输入一组有192个特征、32x32大小，输出256组特征的数据，第一张图直接用3x3卷积实现，需要192x256x3x3x32x32=452984832次乘法；第二张图先用1x1的卷积降到96个特征，再用3x3卷积恢复出256组特征，需要192x96x1x1x32x32+96x256x3x3x32x32=245366784次乘法，使用1x1卷积降维的方法节省了一半的计算量。有人会问，用1x1卷积降到96个特征后特征数不就减少了么，会影响最后训练的效果么？答案是否定的，只要最后输出的特征数不变（256组），中间的降维类似于压缩的效果，并不影响最终训练的结果。

例二： 假设输入数据尺寸为 27X27,通道数为 3,然后经过一个常规卷积,即 4 个 3X3 的卷积核,输出特征图尺寸为 9X9,通道数为 4,然后经过 4 个 1X1 的卷积,输出特征图尺寸为 9X9,通道数为 4,最后再经过 4 个 1X1 的卷积,输出特征图结果依旧是尺寸为 9X9,通道数为 4(每一层 MLP 由 ReLU 激活)



这里的 1X1 卷积除了为了保持特征图的尺寸而使用外,个人觉得它也把**各通道的输入特征图进行线性加权**，起到综合各个通道特征图信息的作用,多几层这样的 1X1 卷积的话,那最终提取的特征会更加抽象。其实可以做个假设,如果不用 1X1 卷积而是用更大的卷

积核,那和连续用 GLM 有什么区别?就是因为用了 1X1 卷积,才增强了常规卷积的抽象表达能力。
MLPconv 其实就是在常规卷积后面加了 N 层 1X1 卷积,所以很容易理解,并且从代码的层次来讲,这种结构也比较的容易实现,以 Pytorch 为例

作用:

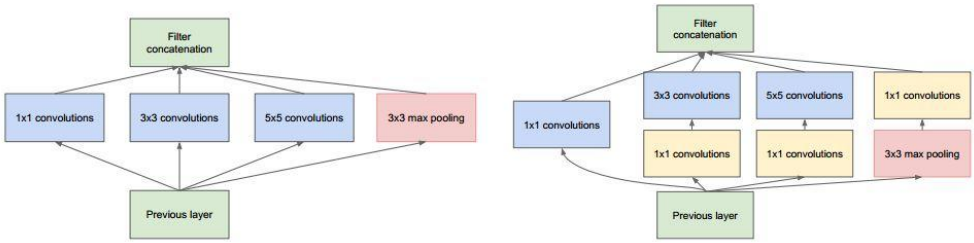
1. **加强了特征的重新组合;**
2. **降低了维度,减少计算量。**

那么 1×1 卷积核有什么作用呢,如果当前层和下一层都只有一个通道那么 1×1 卷积核确实没什么作用,但是如果它们分别为 m 层和 n 层的话,**1×1 卷积核可以起到一个跨通道聚合的作用,所以进一步可以起到降维(或者升维)的作用,起到减少参数的目的。**

比如当前层为 xxxxm 即图像大小为 xxx,特征层数为 m,然后如果将其通过 1×1 的卷积核,特征层数为 n,那么只要 n<m。这样就能起到降维的目的,减少之后步骤的运算量。如果使用 1x1 的卷积核,这个操作实现的就是多个 feature map 的线性组合,可以实现 feature map 在通道个数上的变化。 而因为卷积操作本身就可以做到各个通道的重新聚合的作用,所以 1×1 的卷积核也能达到这个效果。

如 5x5x126 ,通过 1x1x90 的卷积操作后,得到 90 个特征图,此为下次卷积减少乘积,也就减少了计算量。

以 GoogLeNet 的 3a 模块为例,输入的 feature map 是 28×28×192,3a 模块中 1×1 卷积通道为 64,3×3 卷积通道为 128,5×5 卷积通道为 32,如果是左图结构,那么卷积核参数为 1×1×192×64+3×3×192×128+5×5×192×32,而右图对 3×3 和 5×5 卷积层前分别加入了通道数为 96 和 16 的 1×1 卷积层,这样卷积核参数就变成了
1×1×192×64+(1×1×192×96+3×3×96×128)+(1×1×192×16+5×5×16×32),参数大约减少到原来的三分之一。



特性二 全局池化层 (如上图)

1. 大大的减少了模型的参数,由于该连接方式没有参数,从而没有计算,大大减少了计算量,提高了计算速度。2. 有效避免过拟合。3. 可解释性非常强,可以讲最终的 feature map 看作最终分类目标的 confidence map。

五、代码

```
def nin_block(in_channels,out_channels,kernel_size,strides,padding):
```

```
    blk = nn.Sequential(
        nn.Conv2d(in_channels,out_channels,kernel_size,strides,padding)
        nn.ReLU()
        nn.Conv2d(out_channels,out_channels,kernel_size=1)
        nn.ReLU()
        nn.Conv2d(out_channels,out_channels,kernel_size=1)
        nn.ReLU())
    return blk
```

NiN 实现手写数据集

```
import torch
from torch import nn, optim
import torchvision
from datetime import datetime
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
#NiN 块
def nin_block(in_channels, out_channels, kernel_size, stride, padding):
    blk = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding),
        nn.ReLU(),
        #1*1 卷积层
        nn.Conv2d(out_channels, out_channels, kernel_size=1),
        nn.ReLU(),
        #1*1 卷积层
        nn.Conv2d(out_channels, out_channels, kernel_size=1),
        nn.ReLU()
    )
    return blk
net = nn.Sequential(
    #输入 x 是[128, 1, 224, 224]
    #第一个卷积块
    nin_block(1, 96, kernel_size=11, stride=4, padding=0),
    #x 是[128, 96, 54, 54]
    nn.MaxPool2d(kernel_size=3, stride=2),
    #x 是[128, 96, 26, 26]
    #第二个卷积块
    nin_block(96, 256, kernel_size=5, stride=1, padding=2),
    #x 是[128, 256, 26, 26]
    nn.MaxPool2d(kernel_size=3, stride=2),
    #x 是[128, 256, 12, 12]
    #第三个卷积块
    nin_block(256, 384, kernel_size=3, stride=1, padding=1),
    #x 是[128,384,12,12]
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Dropout(0.5),
    #x 是[128, 384, 5, 5]
    #第四个卷积块
    # 标签类别数是 10
    nin_block(384, 10, kernel_size=3, stride=1, padding=1),
    #x 是[128, 10, 5, 5]
```

```
    #全局平均池化层
    #全局平均池化层可通过将窗口形状设置成输入的高和宽实现
    nn.AvgPool2d(kernel_size=5),
    #x 是[128, 10, 1, 1]
    # 将四维的输出转成二维的输出，其形状为(批量大小, 10)
    nn.Flatten(start_dim=1, end_dim=3)
    #x 是[128, 10]
)

def get_acc(output, label):
    total = output.shape[0]
    #output 是概率，每行概率最高的就是预测值
    _, pred_label = output.max(1)
    num_correct = (pred_label == label).sum().item()
    return num_correct / total

batch_size = 128

transform = torchvision.transforms.Compose([
    torchvision.transforms.Resize(size=224),
    torchvision.transforms.ToTensor()
])
train_set = torchvision.datasets.FashionMNIST(
    root='dataset/',
    train=True,
    download=True,
    transform=transform
)
#hand-out 留出法划分
train_set, val_set = torch.utils.data.random_split(train_set, [50000,10000])
test_set = torchvision.datasets.FashionMNIST(
    root='dataset/',
    train=False,
    download=True,
    transform=transform
)

train_loader = torch.utils.data.DataLoader(
    dataset=train_set,
    batch_size=batch_size,
    shuffle=True
)
val_loader = torch.utils.data.DataLoader(
    dataset=val_set,
    batch_size=batch_size,
    shuffle=True
)
test_loader = torch.utils.data.DataLoader(
    dataset=test_set,
    batch_size=batch_size,
    shuffle=False
```

```
)
lr = 2e-3
optimizer = optim.Adam(net.parameters(), lr=lr)
critetion = nn.CrossEntropyLoss()
net = net.to(device)
prev_time = datetime.now()
valid_data = val_loader
for epoch in range(3):
    train_loss = 0
    train_acc = 0
    net.train()
    for inputs,labels in train_loader:
        inputs = inputs.to(device)
        labels = labels.to(device)
        #forward
        outputs = net(inputs)
        loss = critetion(outputs,labels)
        #backward
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
        train_acc += get_acc(outputs, labels)
        #最后还要求平均的
    #显示时间
    cur_time = datetime.now()
    h,remainder = divmod((cur_time - prev_time).seconds, 3600)
    m,s = divmod(remainder,60)
    # time_str = 'Time %02d:%02d:%02d'%(h,m,s)
    time_str = 'Time %02d:%02d:%02d(from %02d/%02d/%02d %02d:%02d:%02d to %0
    #validation
    with torch.no_grad():
        net.eval()
        valid_loss = 0
        valid_acc = 0
        for inputs,labels in valid_data:
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = net(inputs)
            loss = critetion(outputs,labels)
            valid_loss += loss.item()
            valid_acc += get_acc(outputs,labels)
    print("Epoch %d. Train Loss: %f, Train Acc: %f, Valid Loss: %f, Valid Ac
          %(epoch, train_loss/len(train_loader), train_acc / len(train_loade
            valid_acc / len(valid_data))
          + time_str)
    torch.save(net.state_dict(),'params.pkl')

#测试
with torch.no_grad():
    net.eval()
```

```
    correct = 0
    total = 0
    for (images, labels) in test_loader:
        images, labels = images.to(device), labels.to(device)
        output = net(images)
        _, predicted = torch.max(output.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    print("The accuracy of total {} images: {}%".format(total, 100 * correct
```

六、总结

1. NIN 网络是 inception 的前身，提供了减少训练数量的思想，提高了运算的速度。
2. 有效避免了过拟合。

七、好文

- [ ] 1. http://chenjianqu.com/show-65.html
- [ ] 2. http://www.noobyard.com/article/p-prfuqnye-td.html
- [ ] 3. https://its401.com/article/AIHUBEI/120766854