二、DNN 的原理

神经网络通过学习大量样本的输入与输出特征之间的关系，以拟合出输入与输出之间的函数，学习完成后，只给它输入特征，它便会可以给出输出特征。神经网络可以分为这么几步：划分数据集、搭建神经网络、训练网络、测试网络、使用网络。

2.1 划分数据集

数据集里每个样本必须包含输入与输出，将数据集按一定的比例划分为训练集与测试集，分别用于训练网络与测试网络，如表 2-1 所示。

表 2-1 数据集的划分

	样本	输入特征			输出特征		
		In1	In2	In3	Out1	Out2	Out3
训练集	1	*	*	*	*	*	*
	2	*	*	*	*	*	*
	3	*	*	*	*	*	*
	4	*	*	*	*	*	*
	5	*	*	*	*	*	*
	…	*	*	*	*	*	*
	800	*	*	*	*	*	*
测试集	801	*	*	*	*	*	*
	802	*	*	*	*	*	*
	…	*	*	*	*	*	*
	1000	*	*	*	*	*	*

考虑到数据集的输入特征与输出特征都是 3 列，因此神经网络的输入层与输出层也必须都是 3 个神经元，隐藏层可以自行设计，如图 2-1 所示。

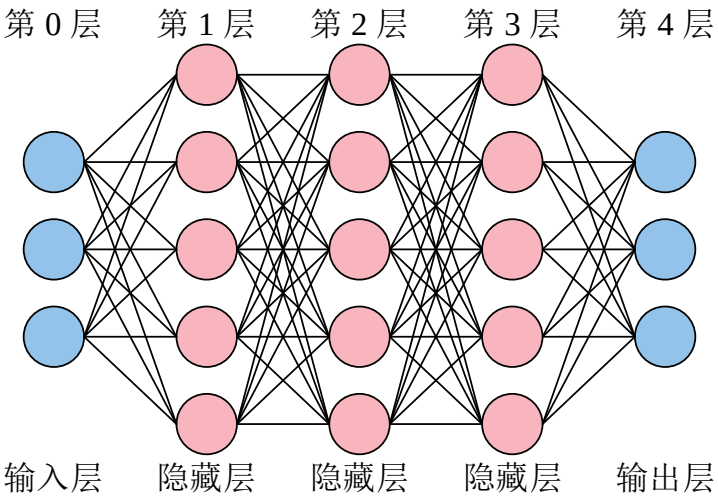


图 2-1 神经网络的结构

考虑到 Python 列表、NumPy 数组以及 PyTorch 张量都是从索引[0]开始，再加之输入层没有内部参数（权重 ω 与偏置 b），所以习惯将输入层称之为第 0 层。

2.2 训练网络

神经网络的训练过程，就是经过很多次前向传播与反向传播的轮回，最终不断调整其内部参数（权重 ω 与偏置 b），以拟合任意复杂函数的过程。内部参数一开始是随机的（如 Xavier 初始值、He 初始值），最终会不断优化到最佳。

还有一些训练网络前就要设好的外部参数：网络的层数、每个隐藏层的节点数、每个节点的激活函数类型、学习率、轮回次数、每次轮回的样本数等等。

业界习惯把内部参数称为参数，外部参数称为超参数。

（1）前向传播

将单个样本的 3 个输入特征送入神经网络的输入层后，神经网络会逐层计算到输出层，最终得到神经网络预测的 3 个输出特征。计算过程中所使用的参数就是内部参数，所有的隐藏层与输出层的神经元都有内部参数，以第 1 层的第 1 个神经元，如图 2-2 所示。

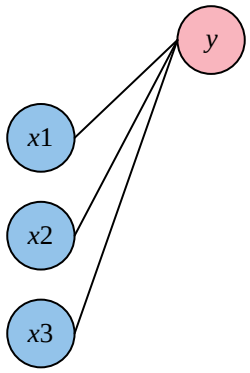


图 2-2 每个神经元节点的计算

该神经元节点的计算过程为 $y=\omega_1 x_1+\omega_2 x_2+\omega_3 x_3+b$。你可以理解为，每一根线就是一个权重 ω，每一个神经元节点也都有它自己的偏置 b。

当然，每个神经元节点在计算完后，由于这个方程是线性的，因此必须在外面套一个非线性的函数：$y=\sigma\left(\omega_1 x_1+\omega_2 x_2+\omega_3 x_3+b\right)$，$\sigma$ 被称为激活函数。如果你不套非线性函数，那么即使 10 层的网络，也可以用 1 层就拟合出同样的方程。

（2）反向传播

经过前向传播，网络会根据当前的内部参数计算出输出特征的预测值。但是这个预测值与真实值直接肯定有差距，因此需要一个损失函数来计算这个差距。例如，求预测值与真实值之间差的绝对值，就是一个典型的损失函数。

损失函数计算好后，逐层退回求梯度，这个过程很复杂，原理不必掌握，大致意思就是，看每一个内部参数是变大还是变小，才会使得损失函数变小。这样就达到了优化内部参数的目的。

在这个过程中，有一个外部参数叫学习率。学习率越大，内部参数的优化越快，但过大的学习率可能会使损失函数越过最低点，并在谷底反复横跳。因此，在网络的训练开始之前，选择一个合适的学习率很重要。

（3）batch_size

前向传播与反向传播一次时，有三种情况：

- 批量梯度下降（Batch Gradient Descent，BGD），把所有样本一次性输入进网络，这种方式计算量开销很大，速度也很慢。
- 随机梯度下降（Stochastic Gradient Descent，SGD），每次只把一个样本输入进网络，每计算一个样本就更新参数。这种方式虽然速度比较快，但是收敛性能差，可能会在最优点附近震荡，两次参数的更新也有可能抵消。
- 小批量梯度下降（Mini-Batch Gradient Decent，MBGD）是为了中和上面二者而生，这种办法把样本划分为若干个批，按批来更新参数。

所以，batch_size 即一批中的样本数，也是一次喂进网络的样本数。此外，由于 Batch Normalization 层（用于将每次产生的小批量样本进行标准化）的存在，batch_size 一般设置为 2 的幂次方，并且不能为 1。

注：PyTorch 实现时只支持批量与小批量，不支持单个样本的输入方式。PyTorch 里的 torch.optim.SGD 只表示梯度下降，批量与小批量见第四、五章。

（4）epochs

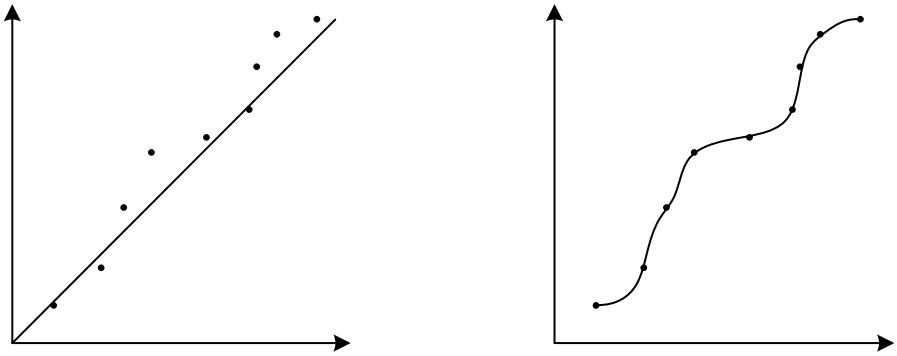
1 个 epoch 就是指全部样本进行 1 次前向传播与反向传播。

假设有 10240 个训练样本，batch_size 是 1024，epochs 是 5。那么：

- 全部样本将进行 5 次前向传播与反向传播；
- 1 个 epoch，将发生 10 次（10240÷1024）前向传播与反向传播；
- 一共发生 50 次（10×5）前向传播和反向传播。

2.3 测试网络

为了防止训练的网络过拟合，因此需要拿出少量的样本进行测试。过拟合的意思是：网络优化好的内部参数只能对训练样本有效，换成其它就寄。以线性回归为例，过拟合如图 2-3（b）所示。



（a）正确的拟合　　（b）过拟合

图 2-3 过拟合的危害

当网络训练好后，拿出测试集的输入，进行 1 次前向传播后，将预测的输出与测试集的真实输出进行对比，查看准确率。

2.4 使用网络

真正使用网络进行预测时，样本只知输入，不知输出。直接将样本的输入进行 1 次前向传播，即可得到预测的输出。

三、DNN 的实现

torch.nn 提供了搭建网络所需的所有组件，nn 即 Neural Network 神经网络。因此，可以单独给 torch.nn 一个别名，即 **import** torch.nn **as** nn。

In [1] :
```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2] :
```
# 展示高清图
from matplotlib_inline import backend_inline
backend_inline.set_matplotlib_formats('svg')
```

3.1 制作数据集

在训练之前，要准备好训练集的样本。

这里生成 10000 个样本，设定 3 个输入特征与 3 个输出特征，其中

- 每个输入特征相互独立，均服从均匀分布；
- 当(X1+X2+X3)< 1 时，Y1 为 1，否则 Y1 为 0；
- 当 1<(X1+X2+X3)<2 时，Y2 为 1，否则 Y2 为 0；
- 当(X1+X2+X3)>2 时，Y3 为 1，否则 Y3 为 0；
- .float()将布尔型张量转化为浮点型张量。

In [3] :
```
# 生成数据集
X1 = torch.rand(10000,1)      # 输入特征 1
X2 = torch.rand(10000,1)      # 输入特征 2
X3 = torch.rand(10000,1)      # 输入特征 3
Y1 = ( (X1+X2+X3)<1 ).float()                          # 输出特征 1
Y2 = ( (1<(X1+X2+X3)) & ((X1+X2+X3)<2) ).float()       # 输出特征 2
Y3 = ( (X1+X2+X3)>2 ).float()                          # 输出特征 3
Data = torch.cat([X1,X2,X3,Y1,Y2,Y3],axis=1)   # 整合数据集
Data = Data.to('cuda:0')                       # 把数据集搬到 GPU 上
Data.shape
```

Out [3] : torch.Size([10000, 6])

事实上，数据的 3 个输出特征组合起来是一个 One-Hot 编码（独热编码）。

In [4] :
```
# 划分训练集与测试集
train_size = int(len(Data) * 0.7)       # 训练集的样本数量
test_size = len(Data) - train_size      # 测试集的样本数量
Data = Data[torch.randperm( Data.size(0)) , : ]      # 打乱样本的顺序
train_Data = Data[ : train_size , : ]      # 训练集样本
test_Data = Data[ train_size : , : ]       # 测试集样本
train_Data.shape, test_Data.shape
```

Out [4] : (torch.Size([7000, 6]), torch.Size([3000, 6]))

In [4]的代码属于通用型代码，便于我们手动分割训练集与测试集。

3.2 搭建神经网络

搭建神经网络时，以 nn.Module 作为父类，我们自己的神经网络可直接继承父类的方法与属性，nn.Module 中包含网络各个层的定义。

在定义的神经网络子类中，通常包含__init__特殊方法与 forward 方法。__init__特殊方法用于构造自己的神经网络结构，forward 方法用于将输入数据进行前向传播。由于张量可以自动计算梯度，所以不需要出现反向传播方法。

In [5] :
```
class DNN(nn.Module):

    def __init__(self):
        ''' 搭建神经网络各层 '''
        super(DNN,self).__init__()
        self.net = nn.Sequential(              # 按顺序搭建各层
            nn.Linear(3, 5), nn.ReLU(),        # 第 1 层：全连接层
            nn.Linear(5, 5), nn.ReLU(),        # 第 2 层：全连接层
            nn.Linear(5, 5), nn.ReLU(),        # 第 3 层：全连接层
            nn.Linear(5, 3)                    # 第 4 层：全连接层
        )

    def forward(self, x):
        ''' 前向传播 '''
        y = self.net(x)       # x 即输入数据
        return y              # y 即输出数据
```

In [6] :
```
model = DNN().to('cuda:0')      # 创建子类的实例，并搬到 GPU 上
model                           # 查看该实例的各层
```

Out [6] :
```
DNN(
  (net): Sequential(
    (0): Linear(in_features=3, out_features=5, bias=True)
    (1): ReLU()
    (2): Linear(in_features=5, out_features=5, bias=True)
    (3): ReLU()
    (4): Linear(in_features=5, out_features=5, bias=True)
    (5): ReLU()
    (6): Linear(in_features=5, out_features=3, bias=True)
  )
)
```

在上面的 nn.Sequential()函数中，每一个隐藏层后都使用了 RuLU 激活函数，各层的神经元节点个数分别是：3、5、5、5、3。

注意，输入层有 3 个神经元、输出层有 3 个神经元，这不是巧合，是有意而为之。输入层的神经元数量必须与每个样本的输入特征数量一致，输出层的神经元数量必须与每个样本的输出特征数量一致。

3.3 网络的内部参数

神经网络的内部参数是权重与偏置，内部参数在神经网络训练之前会被赋予随机数，随着训练的进行，内部参数会逐渐迭代至最佳值，现对参数进行查看。

In [7] :
```
# 查看内部参数（非必要）
for name, param in model.named_parameters():
    print(f"参数:{name}\n 形状:{param.shape}\n 数值:{param}\n")
```

Out [7] :
```
参数:net.0.weight
形状:torch.Size([5, 3])
数值:Parameter containing:
tensor([ [ 0.0526,   -0.3374,   -0.0227],
         [ 0.1673,   0.4338,   0.3040],
         [ 0.5739,   -0.4609,   0.3183],
         [-0.1983,   -0.3941,   0.2630],
         [-0.5472,   0.4121,   -0.2182]],
         device='cuda:0', requires_grad=True)

参数:net.0.bias
形状:torch.Size([5])
数值:Parameter containing:
tensor([ 0.5564,   -0.0882,   -0.4600,   -0.2319,   0.2650],
         device='cuda:0', requires_grad=True)

（页面有限，此处仅展示 net.0 的权重与偏置）
```

代码一共给了我们 8 个参数，其中参数与形状的结果如表 3-1 所示，考虑到其数值初始状态时是随机的（如 Xavier 初始值、He 初始值），此处不讨论。

表 3-1 网络的内部参数及其形状

参数	形状	参数	形状
net.0.weight	torch.Size([5, 3])	net.0.bias	torch.Size([5])
net.2.weight	torch.Size([5, 5])	net.2.bias	torch.Size([5])
net.4.weight	torch.Size([5, 5])	net.4.bias	torch.Size([5])
net.6.weight	torch.Size([3, 5])	net.6.bias	torch.Size([3])

可见，具有权重与偏置的地方只有 net.0、net.2、net.4、net.6，结合 Out [3]的结果，可知这几个地方其实就是所有的隐藏层与输出层，这符合理论。

- 首先，net.0.weight 的权重形状为[5, 3]，5 表示它自己的节点数是 5，3 表示与之连接的前一层的节点数为 3。
- 其次，由于 In [3]里进行了 model = DNN().to('cuda:0')操作，因此所有的内部参数都自带 device='cuda:0'。
- 最后，注意到 requires_grad=True，说明所有需要进行反向传播的内部参数（即权重与偏置）都打开了张量自带的梯度计算功能。

3.4 网络的外部参数

外部参数即超参数，这是调参师们关注的重点。搭建网络时的超参数有：网络的层数、各隐藏层节点数、各节点激活函数、内部参数的初始值等。训练网络的超参数有：如损失函数、学习率、优化算法、batch_size、epochs 等。

（1）激活函数

PyTorch 1.12.0 版本进入 https://pytorch.org/docs/1.12/nn.html 搜索 Non-linear Activations，即可查看 torch 内置的所有非线性激活函数（以及各种类型的层）。

（2）损失函数

进入 https://pytorch.org/docs/1.12/nn.html 搜索 Loss Functions，即可查看 torch 内置的所有损失函数。

In [8] :
```
# 损失函数的选择
loss_fn = nn.MSELoss()
```

（3）学习率与优化算法

进入 https://pytorch.org/docs/1.12/optim.html，可查看 torch 的所有优化算法。

In [9] :
```
# 优化算法的选择
learning_rate = 0.01      # 设置学习率
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

注：PyTorch 实现时只支持 BGD 或 MBGD，不支持单个样本的输入方式。这里的 torch.optim.SGD 只表示梯度下降，具体的批量与小批量见第四、五章。

3.5 训练网络

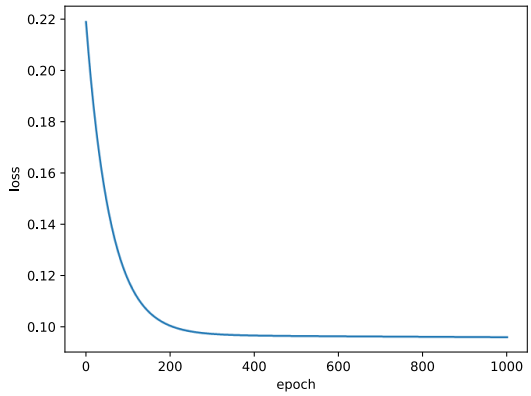
In [10] :
```
# 训练网络
epochs = 1000
losses = []          # 记录损失函数变化的列表

# 给训练集划分输入与输出
X = train_Data[:, :3]        # 前 3 列为输入特征
Y = train_Data[:, -3:]       # 后 3 列为输出特征

for epoch in range(epochs):
    Pred = model(X)                  # 一次前向传播（批量）
    loss = loss_fn(Pred, Y)          # 计算损失函数
    losses.append(loss.item())       # 记录损失函数的变化
    optimizer.zero_grad()            # 清理上一轮滞留的梯度
    loss.backward()                  # 一次反向传播
    optimizer.step()                 # 优化内部参数

Fig = plt.figure()
plt.plot(range(epochs), losses)
plt.ylabel('loss'), plt.xlabel('epoch')
plt.show()
```

Out [10] :



losses.append(loss.item())中，.append()是指在列表 losses 后再附加 1 个元素，而.item()方法可将 PyTorch 张量退化为普通元素。

3.6 测试网络

测试时，只需让测试集进行 1 次前向传播即可，这个过程不需要计算梯度，因此可以在该局部关闭梯度，该操作使用 **with** torch.no_grad():命令。

考虑到输出特征是独热编码，而预测的数据一般都是接近 0 或 1 的小数，为了能让预测数据与真实数据之间进行比较，因此要对预测数据进行规整。例如，使用 Pred[:,torch.argmax(Pred, axis=1)] = 1 命令将每行最大的数置 1，接着再使用 Pred[Pred!=1] = 0 将不是 1 的数字置 0，这就使预测数据与真实数据的格式相同。

In [11] :

```
# 测试网络

# 给测试集划分输入与输出
X = test_Data[:, :3]          # 前 3 列为输入特征
Y = test_Data[:, -3:]         # 后 3 列为输出特征

with torch.no_grad():         # 该局部关闭梯度计算功能
    Pred = model(X)           # 一次前向传播（批量）
    Pred[:,torch.argmax(Pred, axis=1)] = 1
    Pred[Pred!=1] = 0
    correct = torch.sum( (Pred == Y).all(1) )      # 预测正确的样本
    total = Y.size(0)                              # 全部的样本数量
    print(f'测试集精准度: {100*correct/total} %')
```

Out [11] : 测试集精准度: 67.16666412353516 %

在计算 correct 时需要动点脑筋。

首先，(Pred == Y)计算预测的输出与真实的输出的各个元素是否相等，返回一个 3000 行、3 列的布尔型张量。

其次，(Pred == Y).all(1)检验该布尔型张量每一行的 3 个数据是否都是 **True**，对于全是 **True** 的样本行，结果就是 **True**，否则是 **False**。all(1)中的 1 表示按“行”扫描，最终返回一个形状为 3000 的一阶张量。

最后，torch.sum((Pred == Y).all(1))的意思就是看这 3000 个向量相加，**True** 会被当作 1，**False** 会被当作 0，这样相加刚好就是预测正确的样本数。

3.7 保存与导入网络

现在我们要考虑一件大事，那就是有时候训练一个大网络需要几天，那么必须要把整个网络连同里面的优化好的内部参数给保存下来。

现以本章前面的代码为例，当网络训练好后，将网络以文件的形式保存下来，并通过文件导入给另一个新网络，让新网络去跑测试集，看看测试集的准确率是否也是 67%。

（1）保存网络

通过“torch.save(模型名, '文件名.pth')”命令，可将该模型完整的保存至 Jupyter 的工作路径下。

In [12]：
```
# 保存网络
torch.save(model, 'model.pth')
```

（2）导入网络

通过“新网络 = torch.load('文件名.pth ')”命令，可将该模型完整的导入给新网络。

In [13]：
```
# 把模型赋给新网络
new_model = torch.load('model.pth')
```

现在，new_model 就与 model 完全一致，可以直接去跑测试集。

（3）用新模型进行测试

In [14]：
```
# 测试网络

# 给测试集划分输入与输出
X = test_Data[:, :3]          # 前 3 列为输入特征
Y = test_Data[:, -3:]         # 后 3 列为输出特征

with torch.no_grad():                 # 该局部关闭梯度计算功能
    Pred = new_model(X)               # 用新模型进行一次前向传播
    Pred[:,torch.argmax(Pred, axis=1)] = 1
    Pred[Pred!=1] = 0
    correct = torch.sum( (Pred == Y).all(1) )       # 预测正确的样本
    total = Y.size(0)                               # 全部的样本数量
    print(f'测试集精准度: {100*correct/total} %')
```

Out [14]： 测试集精准度: 67.16666412353516 %

保存与加载成功，本视频仅演示这么 1 次，后面的章节不再保存网络。

六、手写数字识别

手写数字识别数据集（MNIST）是机器学习领域的标准数据集，它被称为机器学习领域的“Hello World”，只因任何 AI 算法都可以用此标准数据集进行检验。MNIST 内的每一个样本都是一副二维的灰度图像，如图 6-1 所示。

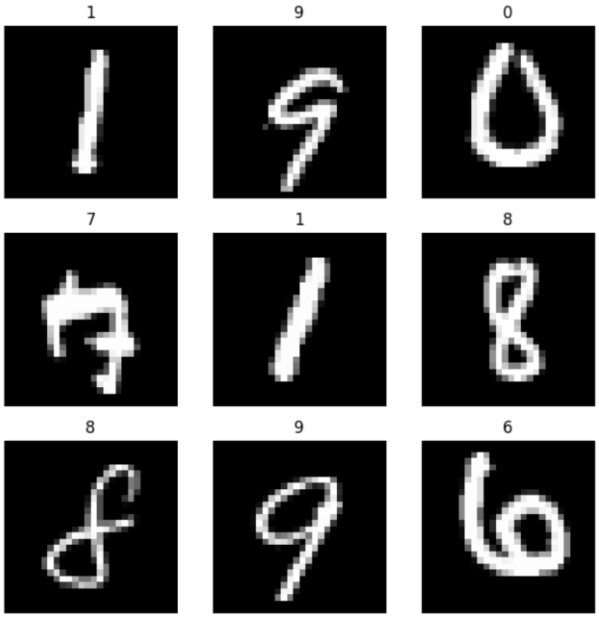
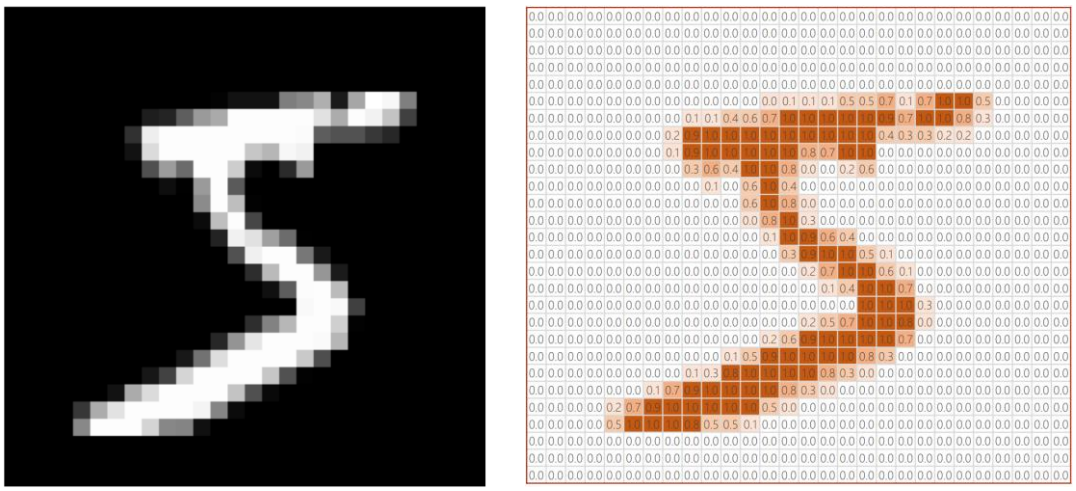


图 6-1 手写数字识别 MNIST 的数据集样本

在 MNIST 中，模型的输入是一副图像，模型的输出就是一个与图像中对应的数字（0 至 9 之间的一个整数，不是独热编码）。

我们不用手动将输出转换为独热编码，PyTorch 会在整个过程中自动将数据集的输出转换为独热编码.只有在最后测试网络时，我们对比测试集的预测输出与真实输出时，才需要注意一下。

某一个具体的样本如图 6-2 所示，每个图像都是形状为 28×28 的二维数组。



（a）样本外观　　（b）样本构成

图 6-2 某一个具体的样本

在这种多分类问题中，神经网络的输出层需要一个 softmax 激活函数，它可以把输出层的数据归一化到 0-1 上，且加起来为 1，这样就模拟出了概率的意思。


```
[0.1, 0.8, 0.05, 0.05, 0.005, 0.03, 0.03, 0.002, 0.015, 0.005]
```

6.1 制作数据集

这一章我们需要在 torchvision 库中分别下载训练集与测试集，因此需要从 torchvision 库中导入 datasets 以下载数据集，下载前需要借助 torchvision 库中的 transforms 进行图像转换，将数据集变为张量，并调整数据集的统计分布。

由于不需要手动构建数据集，因此不导入 utils 中的 Dataset；又由于训练集与测试集是分开下载的，因此不导入 utils 中的 random_split。

In [1] :
```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision import datasets
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2] :
```
# 展示高清图
from matplotlib_inline import backend_inline
backend_inline.set_matplotlib_formats('svg')
```

在下载数据集之前，要设定转换参数：transform，该参数里解决两个问题：

- **ToTensor**：将图像数据转为张量，且调整三个维度的顺序为 C*W*H；C 表示通道数，二维灰度图像的通道数为 1，三维 RGB 彩图的通道数为 3。
- **Normalize**：将神经网络的输入数据转化为标准正态分布，训练更好；根据统计计算，MNIST 训练集所有像素的均值是 0.1307、标准差是 0.3081。

In [3] :
```
# 制作数据集

# 数据集转换参数
transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(0.1307, 0.3081)
])

# 下载训练集与测试集
train_Data = datasets.MNIST(
        root = 'D:/Jupyter/dataset/mnist/',        # 下载路径
        train = True,                  # 是 train 集
        download = True,           # 如果该路径没有该数据集，就下载
        transform = transform       # 数据集转换参数
)
test_Data = datasets.MNIST(
        root = 'D:/Jupyter/dataset/mnist/',        # 下载路径
        train = False,                 # 是 test 集
        download = True,           # 如果该路径没有该数据集，就下载
        transform = transform       # 数据集转换参数
)
```

In [4]:
```
# 批次加载器
train_loader = DataLoader(train_Data, shuffle=True, batch_size=64)
test_loader = DataLoader(test_Data, shuffle=False, batch_size=64)
```

6.2 搭建神经网络

每个样本的输入都是形状为28×28的二维数组，那么对于 DNN 来说，输入层的神经元节点就要有 28×28＝784 个；输出层使用独热编码，需要 10 个节点。

In [5]:
```
class DNN(nn.Module):

    def __init__(self):
        ''' 搭建神经网络各层 '''
        super(DNN,self).__init__()
        self.net = nn.Sequential(                    # 按顺序搭建各层
            nn.Flatten(),                           # 把图像铺平成一维
            nn.Linear(784, 512), nn.ReLU(),   # 第 1 层：全连接层
            nn.Linear(512, 256), nn.ReLU(),   # 第 2 层：全连接层
            nn.Linear(256, 128), nn.ReLU(),   # 第 3 层：全连接层
            nn.Linear(128, 64), nn.ReLU(),    # 第 4 层：全连接层
            nn.Linear(64, 10)                  # 第 5 层：全连接层
        )

    def forward(self, x):
        ''' 前向传播 '''
        y = self.net(x)             # x 即输入数据
        return y                    # y 即输出数据
```

In [6]:
```
model = DNN().to('cuda:0')      # 创建子类的实例，并搬到 GPU 上
model                           # 查看该实例的各层
```

Out [6]:
```
DNN(
  (net): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=784, out_features=512, bias=True)
    (2): ReLU()
    (3): Linear(in_features=512, out_features=256, bias=True)
    (4): ReLU()
    (5): Linear(in_features=256, out_features=128, bias=True)
    (6): ReLU()
    (7): Linear(in_features=128, out_features=64, bias=True)
    (8): ReLU()
    (9): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

6.3 训练网络

In [7] :
```
# 损失函数的选择
loss_fn = nn.CrossEntropyLoss()      # 自带 softmax 激活函数
```

In [8] :
```
# 优化算法的选择
learning_rate = 0.01      # 设置学习率
optimizer = torch.optim.SGD(
        model.parameters(),
        lr = learning_rate,
        momentum = 0.5
)
```

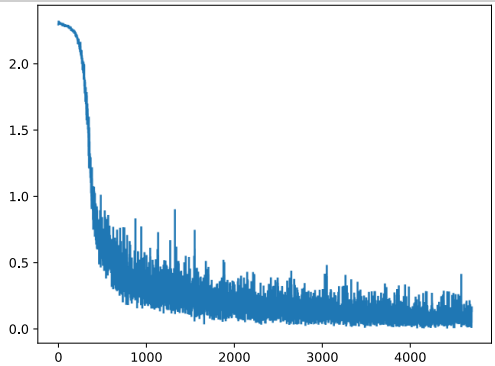
在 In [8]中，给优化器了一个新参数 momentum（动量），它使梯度下降算法有了力与惯性，该方法给人的感觉就像是小球在地面上滚动一样。

In [9] :
```
# 训练网络
epochs = 5
losses = []          # 记录损失函数变化的列表

for epoch in range(epochs):
        for (x, y) in train_loader:              # 获取小批次的 x 与 y
                x, y = x.to('cuda:0'), y.to('cuda:0')
                Pred = model(x)                  # 一次前向传播（小批量）
                loss = loss_fn(Pred, y)          # 计算损失函数
                losses.append(loss.item())       # 记录损失函数的变化
                optimizer.zero_grad()            # 清理上一轮滞留的梯度
                loss.backward()                  # 一次反向传播
                optimizer.step()                 # 优化内部参数

Fig = plt.figure()
plt.plot(range(len(losses)), losses)
plt.show()
```

Out [8] :



注意，由于数据集内部进不去，只能在循环的过程中取出一部分样本，就立即将之搬到 GPU 上。

6.4 测试网络

In [10] :
```
# 测试网络
correct = 0
total = 0

with torch.no_grad():                # 该局部关闭梯度计算功能
    for (x, y) in test_loader:       # 获取小批次的 x 与 y
        x, y = x.to('cuda:0'), y.to('cuda:0')
        Pred = model(x)              # 一次前向传播（小批量）
        _, predicted = torch.max(Pred.data, dim=1)
        correct += torch.sum( (predicted == y) )
        total += y.size(0)

print(f'测试集精准度: {100*correct/total} %')
```

Out [10] : 测试集精准度: 96.65999603271484 %

a, b = torch.max(Pred.data, dim=1)的意思是，找出 Pred 每一行里的最大值，数值赋给 a，所处位置赋给 b。因此上述代码里的 predicted 就相当于把独热编码转换回了普通的阿拉伯数字，这样一来可以直接与 y 进行比较。

由于此处 predicted 与 y 是一阶张量，因此 correct 行的结尾不能加.all(1)。

一、CNN 的原理

1.1 从 DNN 到 CNN

（1）卷积层与汇聚

- 深度神经网络 DNN 中，相邻层的所有神经元之间都有连接，这叫全连接；卷积神经网络 CNN 中，新增了卷积层（Convolution）与汇聚（Pooling）。
- DNN 的全连接层对应 CNN 的卷积层，汇聚是与激活函数类似的附件；单个卷积层的结构是：卷积层-激活函数-(汇聚)，其中汇聚可省略。

（2）CNN：专攻多维数据

在深度神经网络 DNN 课程的最后一章，使用 DNN 进行了手写数字的识别。但是，图像至少就有二维，向全连接层输入时，需要多维数据拉平为 1 维数据，这样一来，图像的形状就被忽视了，很多特征是隐藏在空间属性里的，如图 1-1。

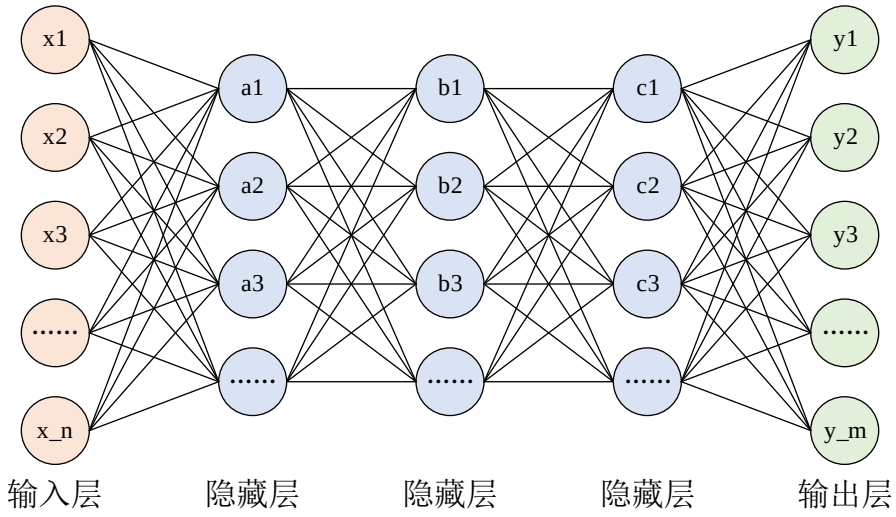


图 1-1　DNN 的结构

而卷积层可以保持输入数据的维数不变，当输入数据是二维图像时，卷积层会以多维数据的形式接收输入数据，并同样以多维数据的形式输出至下一层，如图 1-2 所示。

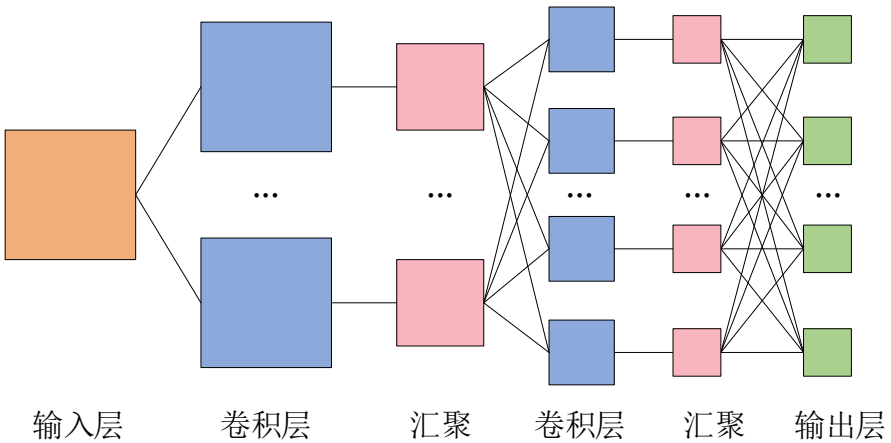


图 1-2　CNN 的结构

1.2 卷积层

CNN 中的卷积层与 DNN 中的全连接层是平级关系，全连接层中的权重与偏置即 $y=\omega_1 x_1+\omega_2 x_2+\omega_3 x_3+b$ 中的 ω 与 b，卷积层中的权重与偏置变得稍微复杂。

（1）内部参数：权重（卷积核）

当输入数据进入卷积层后，输入数据会与卷积核进行卷积运算，如图 1-3。

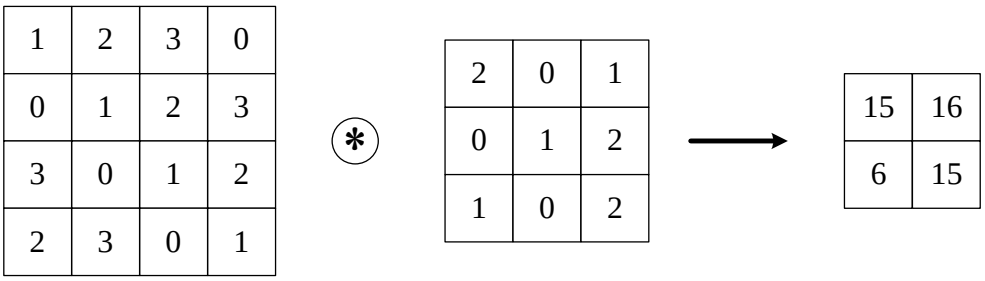


图 1-3 卷积核的运算

图 1-3 中，输入大小是(4, 4)，卷积核大小是(3, 3)，输出大小是(2, 2)。卷积运算的原理是逐元素乘积后再相加，如图 1-4 所示。

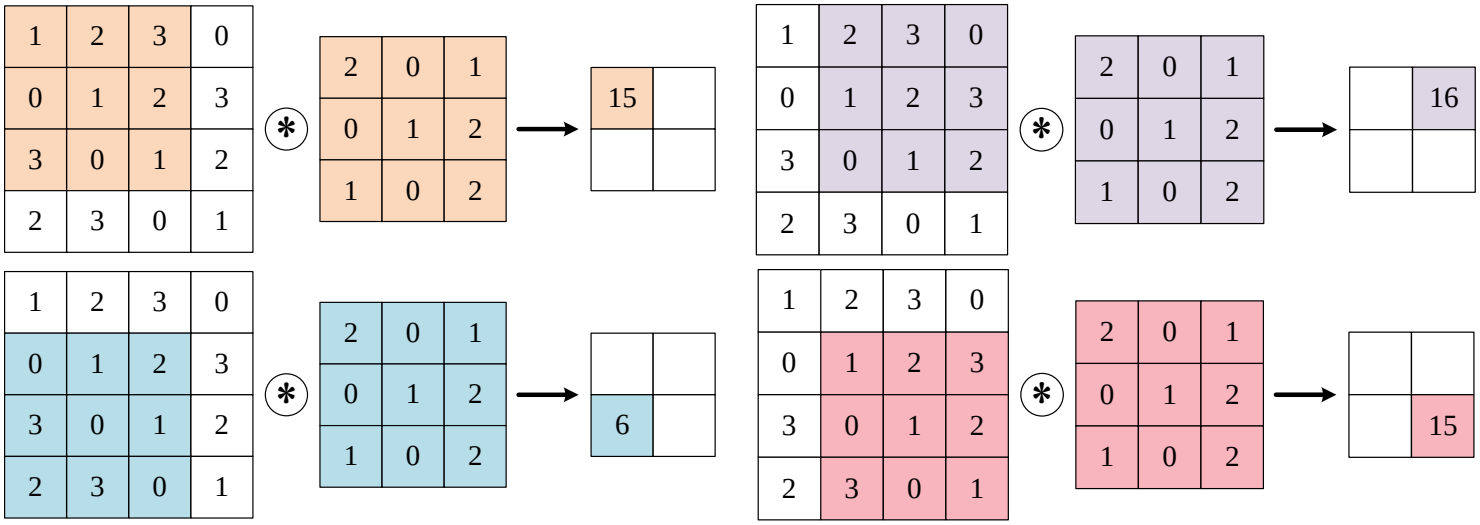


图 1-4 卷积运算的具体步骤

（2）内部参数：偏置

在卷积运算的过程中也存在偏置，如图 1-5 所示。

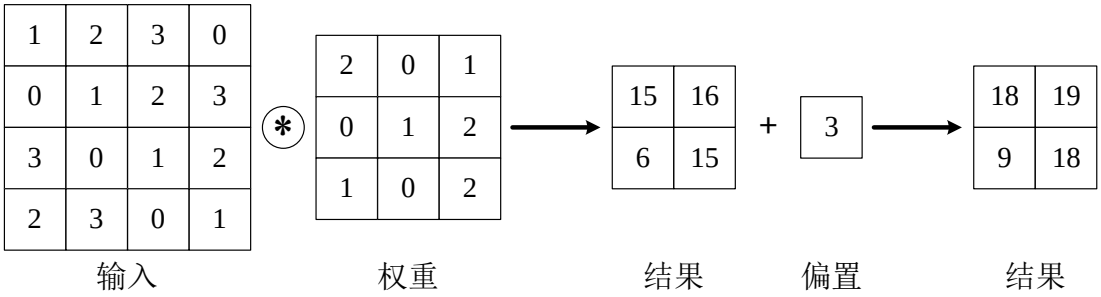


图 1-5 权重与偏置共同作用

（3）外部参数：填充

为了防止经过多个卷积层后图像越卷越小，可以在进行卷积层的处理之前，向输入数据的周围填入固定的数据（比如 0），这称为填充（padding）。

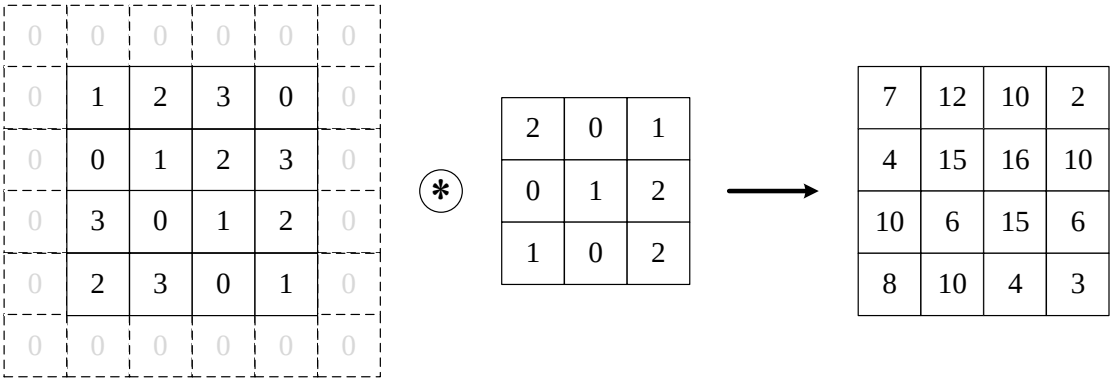


图 1-6 填充

图 1-6 中，对大小为(4, 4)的输入数据应用了幅度为 1 的填充，填充值为 0。

（4）外部参数：步幅

使用卷积核的位置间隔被称为步幅（stride），之前的例子中步幅都是 1，如果将步幅设为 2，则如图 1-7 所示，此时使用卷积核的窗口的间隔变为 2。

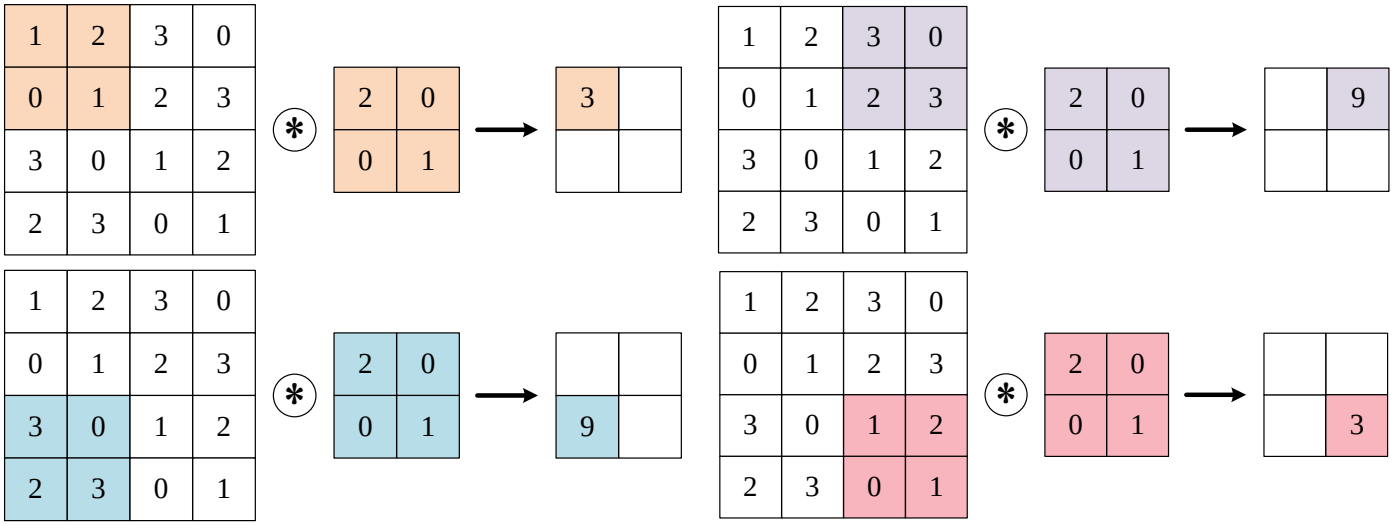


图 1-7 步幅为 2 的计算方式

综上，增大填充后，输出尺寸会变大；而增大步幅后，输出尺寸会变小。

（5）输入与输出尺寸的关系

假设输入尺寸为(H, W)，卷积核的尺寸为(FH, FW)，填充为 P，步幅为 S。则输出尺寸(OH, OW)的计算公式为

$$\begin{cases} OH = \dfrac{H + 2P - FH}{S} + 1 \\ OW = \dfrac{W + 2P - FW}{S} + 1 \end{cases}$$

1.3 多通道

在上一小节讲的卷积层，仅仅针对二维的输入与输出数据（一般是灰度图像），可称之为单通道。但是，彩色图像除了高、长两个维度之外，还有第三个维度：通道（channel）。例如，以 RGB 三原色为基础的彩色图像，其通道方向就有红、黄、蓝三部分，可视为 3 个单通道二维图像的混合叠加。

一般的，当输入数据是二维时，权重被称为卷积核（Kernel）；当输入数据是三维或更高时，权重被称为滤波器（Filter）。

（1）多通道输入

对三维数据的卷积操作如图 1-8 所示，输入数据与滤波器的通道数必须要设为相同的值，可以发现，这种情况下的输出结果降级为了二维。

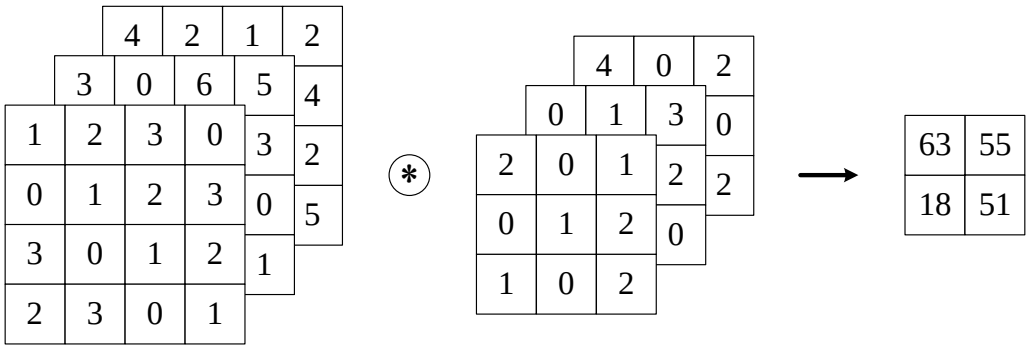


图 1-8 多通道下滤波器的运算

将数据和滤波器看作长方体，如图 1-9 所示。

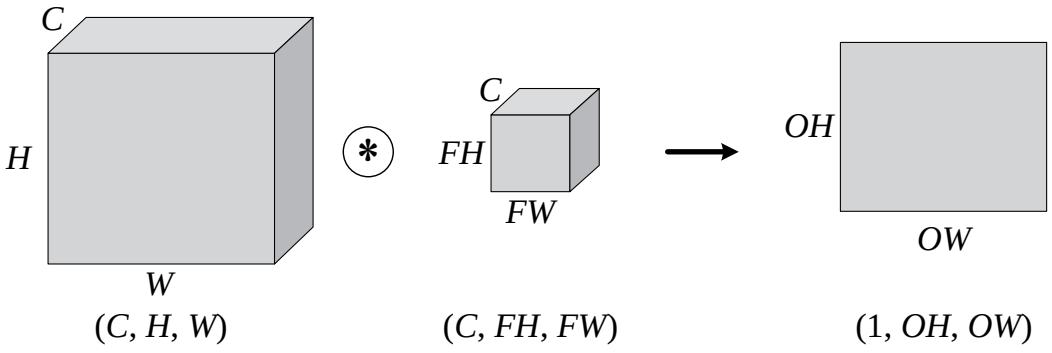


图 1-9 多通道下的卷积运算

C、H、W 是固定的顺序，通道数要写在高与宽的前面。

（2）多通道输出

图 1-9 可看出，仅通过一个卷积层，三维就被降成二维了。大多数时候我们想让三维的特征多经过几个卷积层，因此就需要多通道输出，如图 1-10 所示。

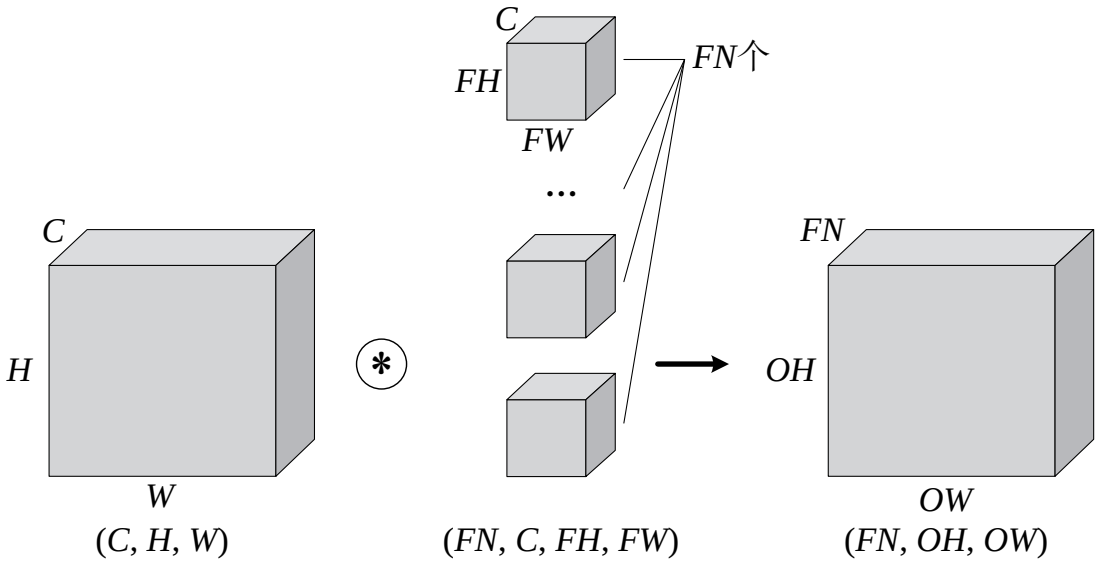


图 1-10 多通道输出

别忘了，卷积运算中存在偏置，如果进一步追加偏置的加法运算处理，则结果如图 1-11 所示，每个通道都有一个单独的偏置。

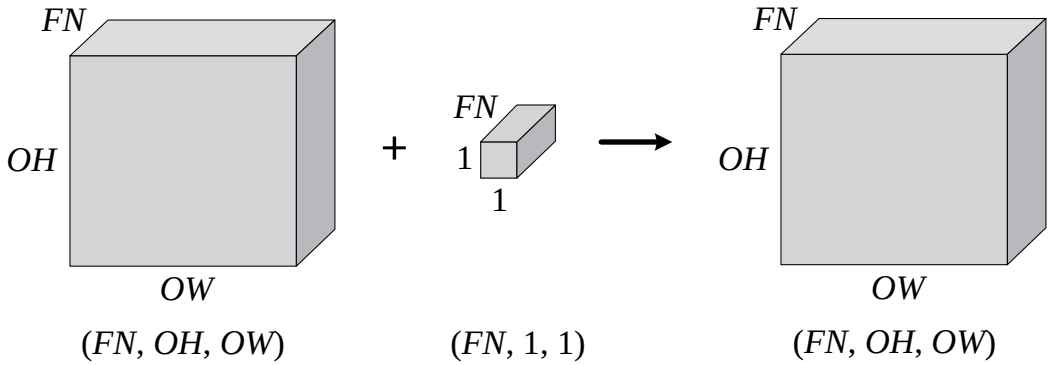


图 1-11 多通道下的偏置

1.4 汇聚

汇聚（Pooling）仅仅是从一定范围内提取一个特征值，所以不存在要学习的内部参数。一般有平均汇聚与最大值汇聚。

（1）平均汇聚

一个以步幅为 2 进行 2*2 窗口的平均汇聚，如图 1-12 所示。

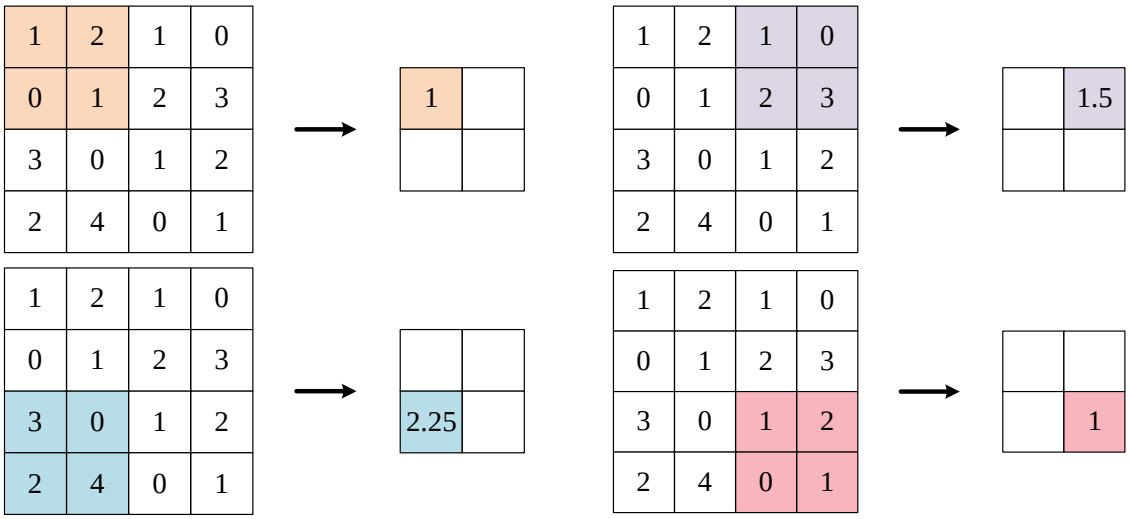


图 1-12 平均汇聚

（2）最大值汇聚

一个以步幅为 2 进行 2*2 窗口的最大值汇聚，如图 1-13 所示。

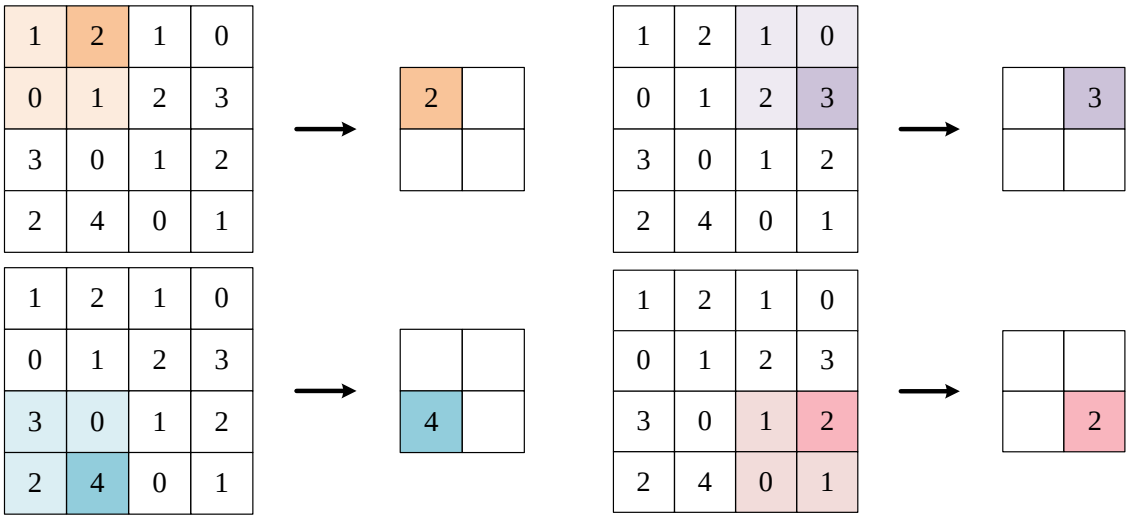


图 1-13 最大值汇聚

汇聚对图像的高 H 和宽 W 进行特征提取，不改变通道数 C。

1.5 尺寸变换总结

（1）卷积层

现假设卷积层的填充为 P，步幅为 S，由

- **输入数据**的尺寸是：(C,H,W)。
- **滤波器**的尺寸是：(FN,C,FH,FW)。
- **输出数据**的尺寸是：(FN,OH,OW)。

可得

$$(C,H,W) \circledast (FN,C,FH,FW) \longrightarrow (FN,OH,OW)$$

$$\begin{cases} OH = \dfrac{H+2P-FH}{S}+1 \\ OW = \dfrac{W+2P-FW}{S}+1 \end{cases}$$

（2）汇聚

现假设汇聚的步幅为 S，由

- **输入数据**的尺寸是：(C,H,W)。
- **输出数据**的尺寸是：(C,OH,OW)。

可得

$$(C,H,W) \longrightarrow (C,OH,OW)$$

$$\begin{cases} OH = H/S \\ OW = W/S \end{cases}$$

$$(C,H,W) \circledast (FN,C,FH,FW) \longrightarrow (FN,OH,OW)$$

$$\begin{cases} OH = \dfrac{H+2P-FH}{S}+1 \\ OW = \dfrac{W+2P-FW}{S}+1 \end{cases}$$

二、LeNet-5

2.1 网络结构

LeNet-5 虽诞生于 1998 年，但基于它的手写数字识别系统则非常成功。

该网络共 7 层，输入图像尺寸为 28×28，输出则是 10 个神经元，分别表示某手写数字是 0 至 9 的概率。

表 2-1 网络结构

层	In	C1	S2	C3	S4	C5	F6	Out
类型	输入层	卷积层	平均汇聚	卷积层	平均汇聚	卷积层	全连接层	全连接层
卷积核		6×1×5×5	2×2	16×6×5×5	2×2	120×16×5×5		
步幅		1	2	1	2	1		
填充		2						
激活函数		tanh		tanh		tanh	tanh	RBF
尺寸	1×28×28	6×28×28	6×14×14	16×10×10	16×5×5	120×1×1	84	10

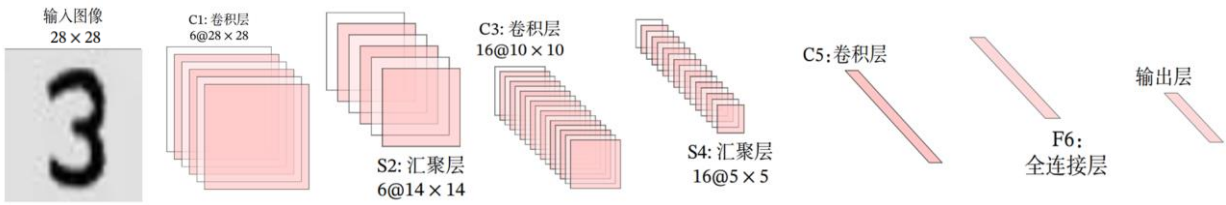


图 2-1 网络结构

注：输出层由 10 个径向基函数 RBF 组成，用于归一化最终的结果，目前 RBF 已被 Softmax 取代。

根据网络结构，在 PyTorch 的 nn.Sequential 中编写为

```
self.net = nn.Sequential(
        nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Tanh(),      # C1：卷积层
        nn.AvgPool2d(kernel_size=2, stride=2),                     # S2：平均汇聚
        nn.Conv2d(6, 16, kernel_size=5), nn.Tanh(),                # C3：卷积层
        nn.AvgPool2d(kernel_size=2, stride=2),                     # S4：平均汇聚
        nn.Conv2d(16, 120, kernel_size=5), nn.Tanh(),              # C5：卷积层
        nn.Flatten(),                                              # 把图像铺平成一维
        nn.Linear(120, 84), nn.Tanh(),                             # F5：全连接层
        nn.Linear(84, 10)                                          # F6：全连接层
)
```

其中，nn.Conv2d()需要四个参数，分别为

- in_channel：此层输入图像的通道数；
- out_channel：此层输出图像的通道数；
- kernel_size：卷积核尺寸；
- padding：填充；
- stride：步幅。

2.2 制作数据集

In [1] :
```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision import datasets
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2] :
```
# 展示高清图
from matplotlib_inline import backend_inline
backend_inline.set_matplotlib_formats('svg')
```

In [3] :
```
# 制作数据集

# 数据集转换参数
transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(0.1307, 0.3081)
])

# 下载训练集与测试集
train_Data = datasets.MNIST(
        root = 'D:/Jupyter/dataset/mnist/',        # 下载路径
        train = True,                  # 是 train 集
        download = True,           # 如果该路径没有该数据集，就下载
        transform = transform      # 数据集转换参数
)
test_Data = datasets.MNIST(
        root = 'D:/Jupyter/dataset/mnist/',        # 下载路径
        train = False,                 # 是 test 集
        download = True,           # 如果该路径没有该数据集，就下载
        transform = transform       # 数据集转换参数
)
```

In [4] :
```
# 批次加载器
train_loader = DataLoader(train_Data, shuffle=True, batch_size=256)
test_loader = DataLoader(test_Data, shuffle=False, batch_size=256)
```

2.3 搭建神经网络

In [5] :
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN,self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Tanh(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Conv2d(6, 16, kernel_size=5), nn.Tanh(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Conv2d(16, 120, kernel_size=5), nn.Tanh(),
            nn.Flatten(),
            nn.Linear(120, 84), nn.Tanh(),
            nn.Linear(84, 10)
        )

    def forward(self, x):
        y = self.net(x)
        return y
```

In [6] :
```
# 查看网络结构
X = torch.rand(size= (1, 1, 28, 28))
for layer in CNN().net:
    X = layer(X)
    print( layer.__class__.__name__, 'output shape: \t', X.shape )
```

Out [6] :
```
Conv2d output shape:        torch.Size([1, 6, 28, 28])
Tanh output shape:          torch.Size([1, 6, 28, 28])
AvgPool2d output shape:     torch.Size([1, 6, 14, 14])
Conv2d output shape:        torch.Size([1, 16, 10, 10])
Tanh output shape:          torch.Size([1, 16, 10, 10])
AvgPool2d output shape:     torch.Size([1, 16, 5, 5])
Conv2d output shape:        torch.Size([1, 120, 1, 1])
Tanh output shape:          torch.Size([1, 120, 1, 1])
Flatten output shape:       torch.Size([1, 120])
Linear output shape:        torch.Size([1, 84])
Tanh output shape:          torch.Size([1, 84])
Linear output shape:        torch.Size([1, 10])
```

In [7] :
```
# 创建子类的实例，并搬到 GPU 上
model = CNN().to('cuda:0')
```

2.4 训练网络

In [8] :
```
# 损失函数的选择
loss_fn = nn.CrossEntropyLoss()      # 自带 softmax 激活函数
```

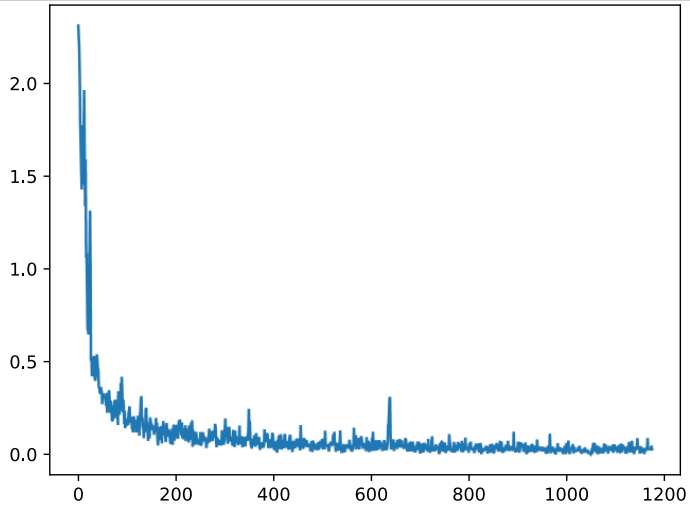
In [9] :
```
# 优化算法的选择
learning_rate = 0.9      # 设置学习率
optimizer = torch.optim.SGD(
        model.parameters(),
        lr = learning_rate,
)
```

In [10] :
```
# 训练网络
epochs = 5
losses = []          # 记录损失函数变化的列表

for epoch in range(epochs):
        for (x, y) in train_loader:                 # 获取小批次的 x 与 y
                x, y = x.to('cuda:0'), y.to('cuda:0')
                Pred = model(x)                     # 一次前向传播（小批量）
                loss = loss_fn(Pred, y)             # 计算损失函数
                losses.append(loss.item())          # 记录损失函数的变化
                optimizer.zero_grad()               # 清理上一轮滞留的梯度
                loss.backward()                     # 一次反向传播
                optimizer.step()                    # 优化内部参数

Fig = plt.figure()
plt.plot(range(len(losses)), losses)
plt.show()
```

Out [10] :



2.5 测试网络

In [11] :

```
# 测试网络
correct = 0
total = 0

with torch.no_grad():                  # 该局部关闭梯度计算功能
    for (x, y) in test_loader:        # 获取小批次的 x 与 y
        x, y = x.to('cuda:0'), y.to('cuda:0')
        Pred = model(x)                # 一次前向传播（小批量）
        _, predicted = torch.max(Pred.data, dim=1)
        correct += torch.sum( (predicted == y) )
        total += y.size(0)

print(f'测试集精准度: {100*correct/total} %')
```

Out [11] : 测试集精准度: 97.93999481201172 %