目　录

引入

0.1 版本需求

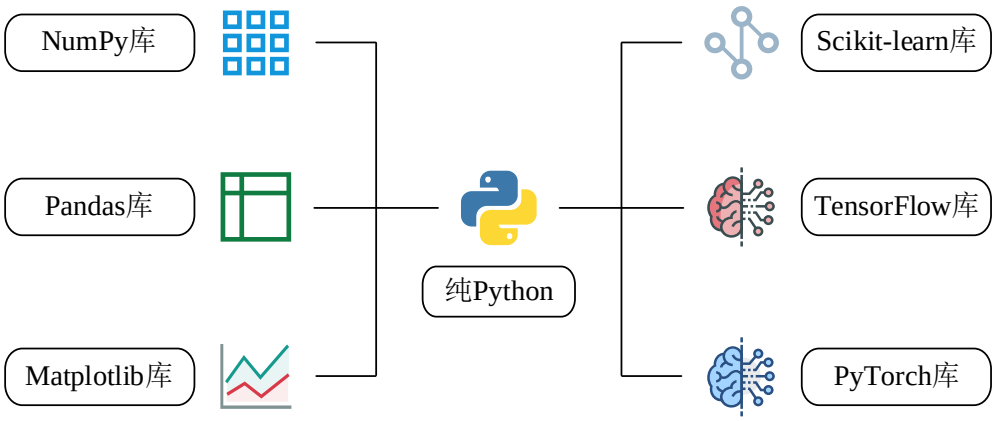
本视频中，使用的 Python 解释器与第三方库的版本如下。

- Python 为 3.9 版本，自 3.4 以来改动的语法可忽略不计；
- NumPy 为 1.21 版本，Pandas 为 1.2.4 版本，Matplotlib 为 3.5.1 版本；
- PyTorch 为 1.12.0 版本，此版本相对较新，更高的只有 1.13.0 和 2.0.0。据悉，PyTorch 2.0.0 对性能有极大的提升，语法规则的变动较小。

0.2 视频特点

- **清晰度**：本视频分辨率为 1080P，请调高分辨率；
- **交流群**：关注【布尔艺数】公众号回复**“杰哥”**，弹出助理的二维码；
- **讲义链接**：微信交流群的群公告中。
- **本课基础**：环境配置、Python 基础、NumPy 数组库、深度神经网络 DNN。

0.3 深度学习的相关库



① NumPy 包为 Python 加上了关键的数组变量类型，弥补了 Python 的不足；

② Pandas 包在 NumPy 数组的基础上添加了与 Excel 类似的行列标签；

③ Matplotlib 库借鉴 Matlab，帮 Python 具备了绘图能力，使其如虎添翼；

④ Scikit-learn 库是机器学习库，内含分类、回归、聚类、降维等多种算法；

⑤ TensorFlow 库是 Google 公司开发的深度学习框架，于 2015 年问世；

⑥ PyTorch 库是 Facebook 公司开发的深度学习框架，于 2017 年问世。

0.4 深度学习的基本常识

- **人工智能**是一个很大的概念，其中一个最重要的分支就是机器学习；
- **机器学习**的算法多种多样，其中最核心的就是神经网络；
- **神经网络**的隐藏层若足够深，就被称为深层神经网络，也即深度学习；
- **深度学习**包含深度神经网络、卷积神经网络、循环神经网络等。

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。

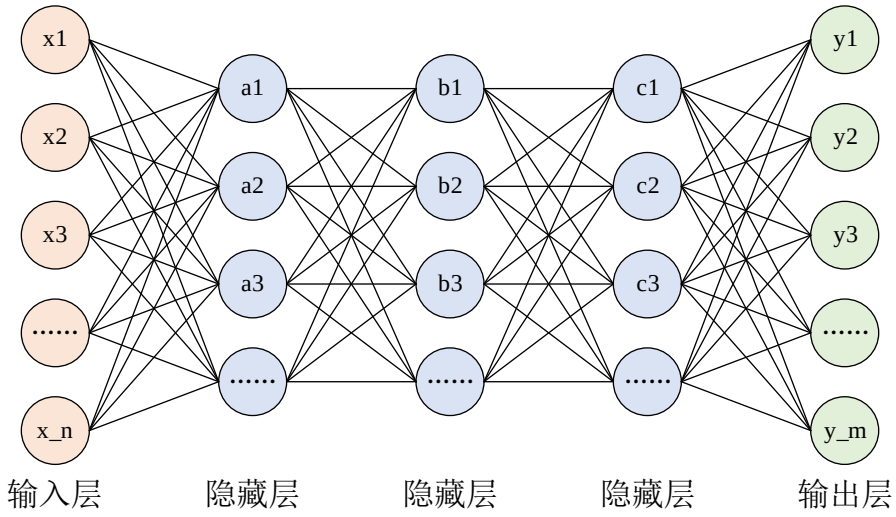


图 1-1　DNN 的结构

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。

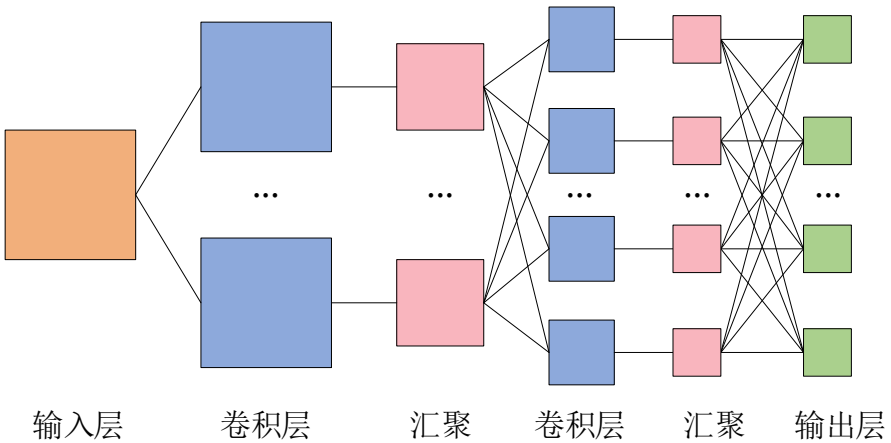


图 1-2　CNN 的结构

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。

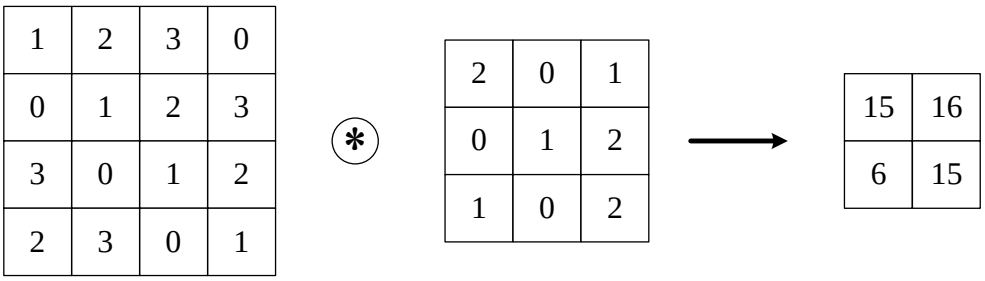


图 1-3 卷积核的运算

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。

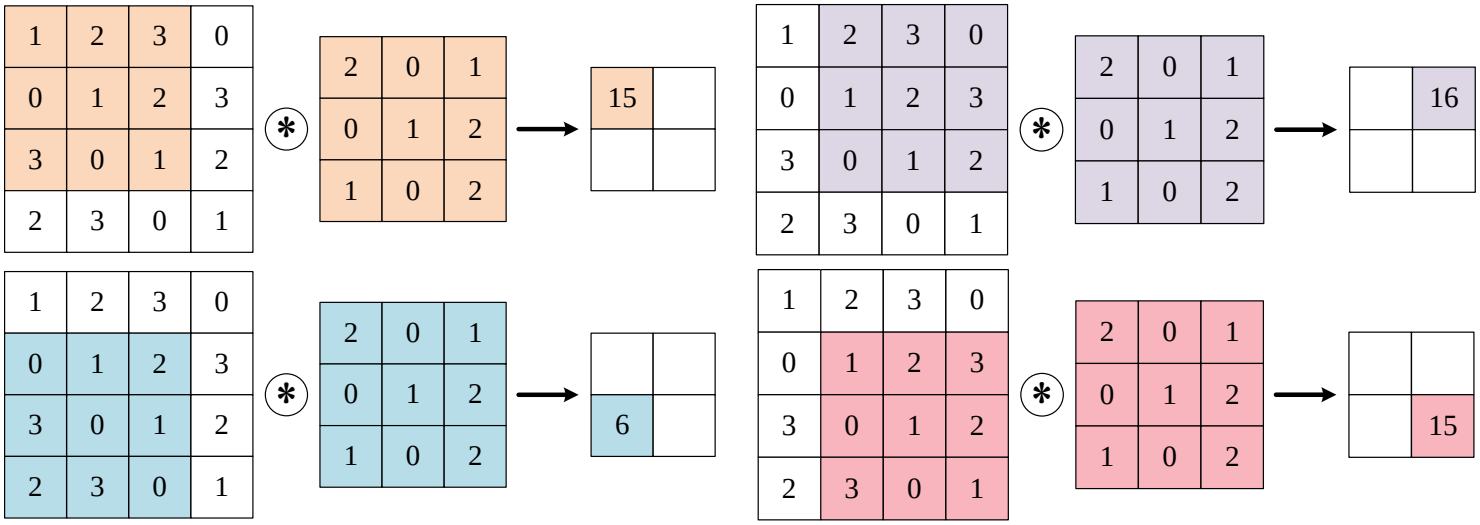


图 1-4 卷积运算的具体步骤

（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。

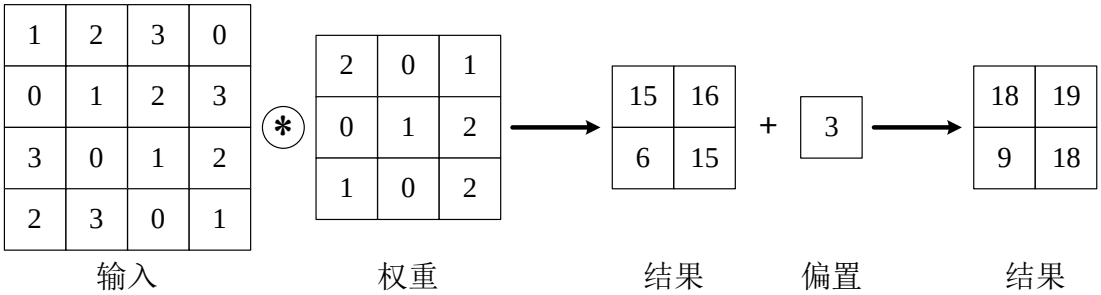


图 1-5 权重与偏置共同作用

（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）。

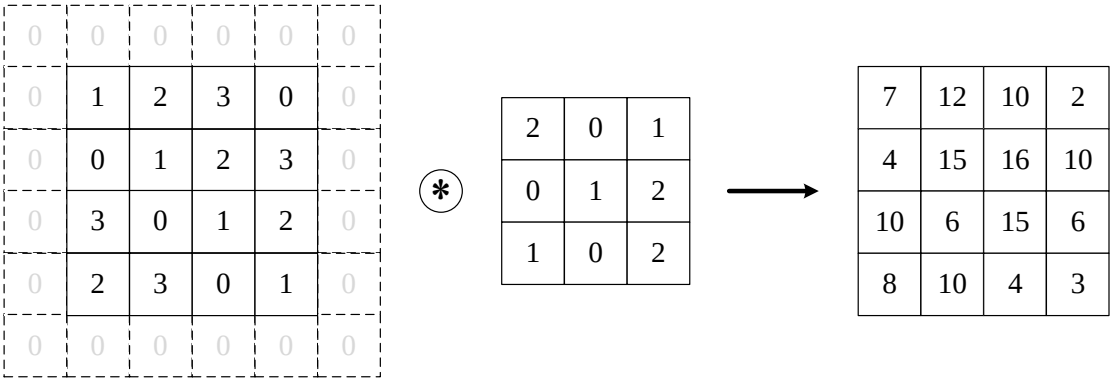


图 1-6 填充

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。

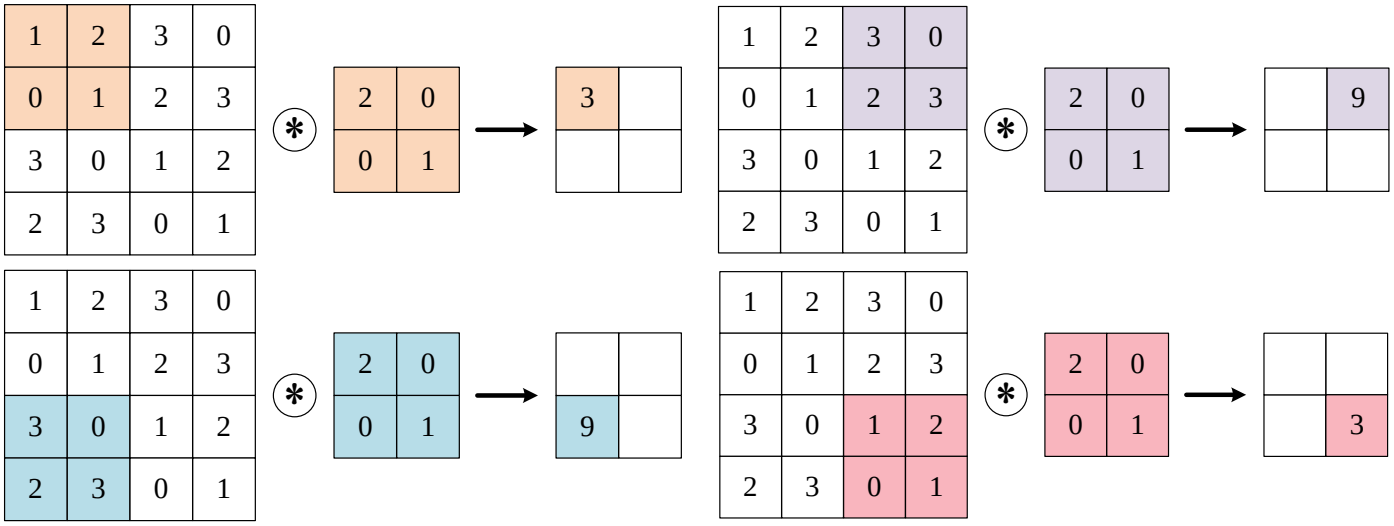


图 1-7 步幅为 2 的计算方式

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。

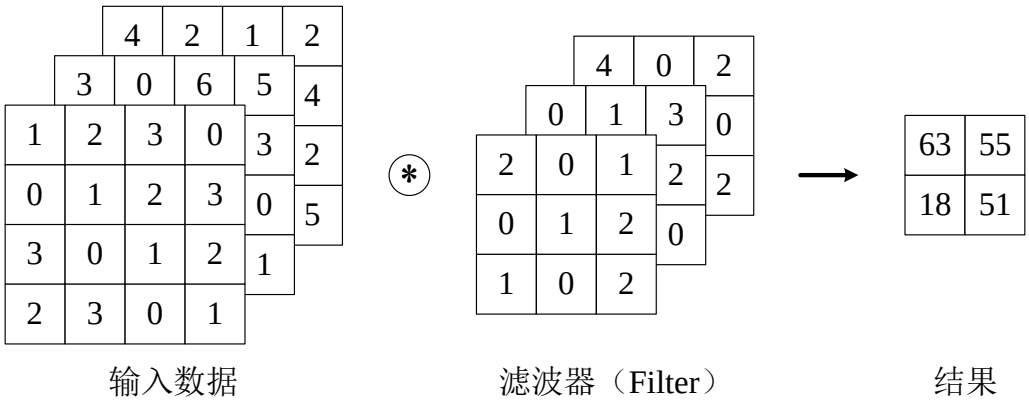


图 1-8 多通道下滤波器的运算

将数据和滤波器看作长方体，如图 1-9 所示。

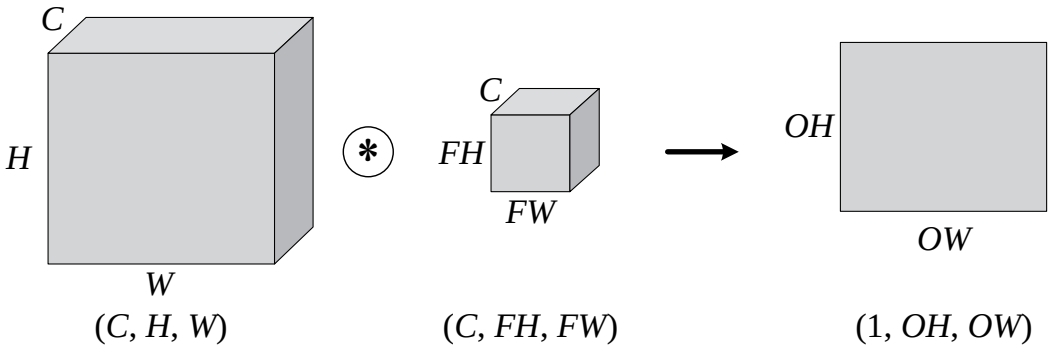


图 1-9 多通道下的卷积运算

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。

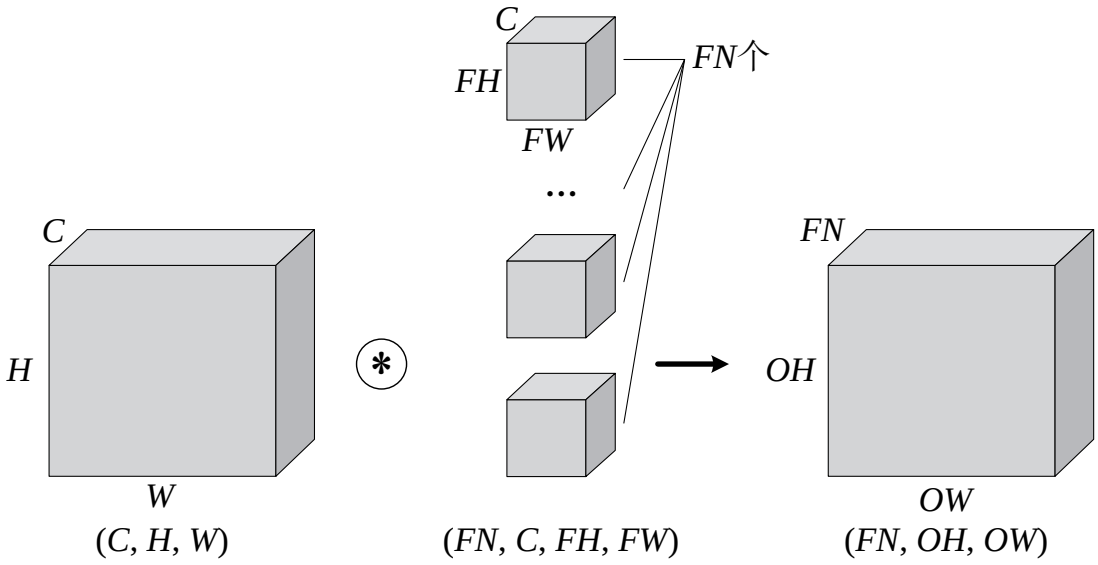


图 1-10 多通道输出

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。

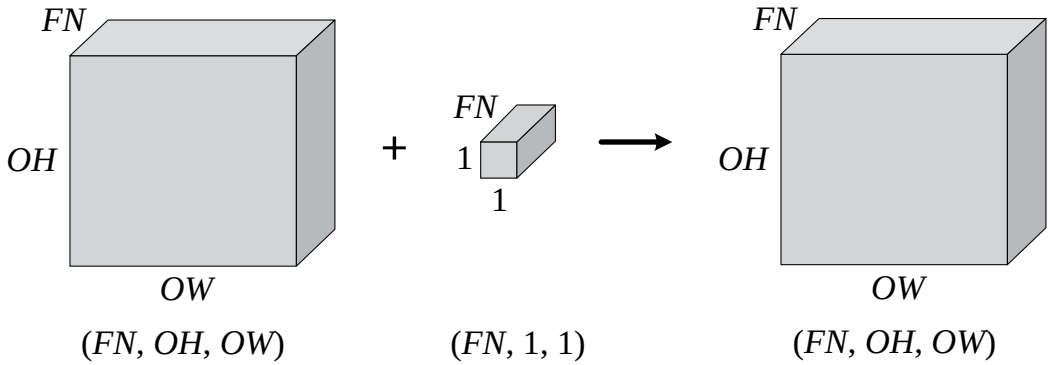


图 1-11 多通道下的偏置

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。

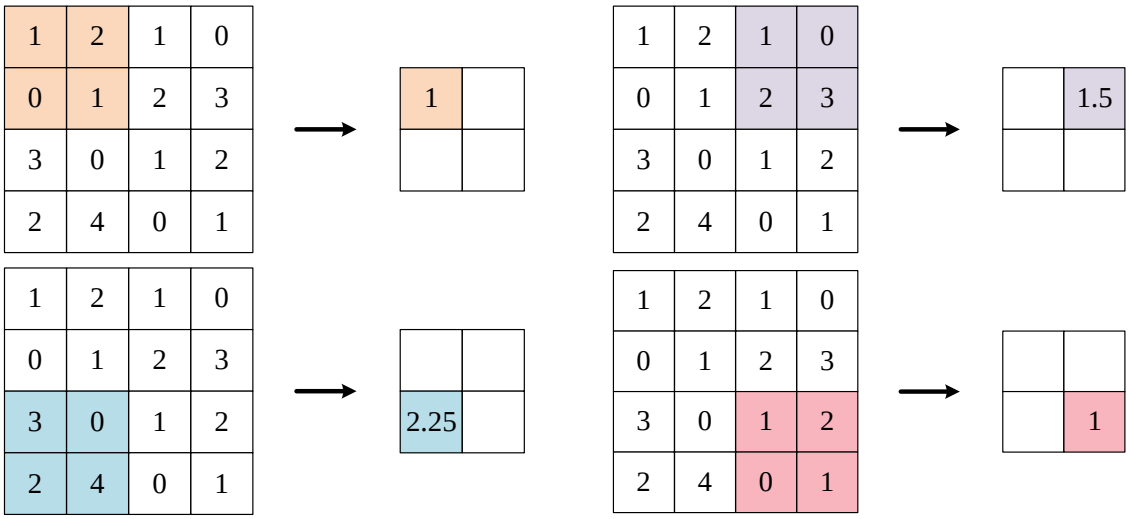


图 1-12 平均汇聚

（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。

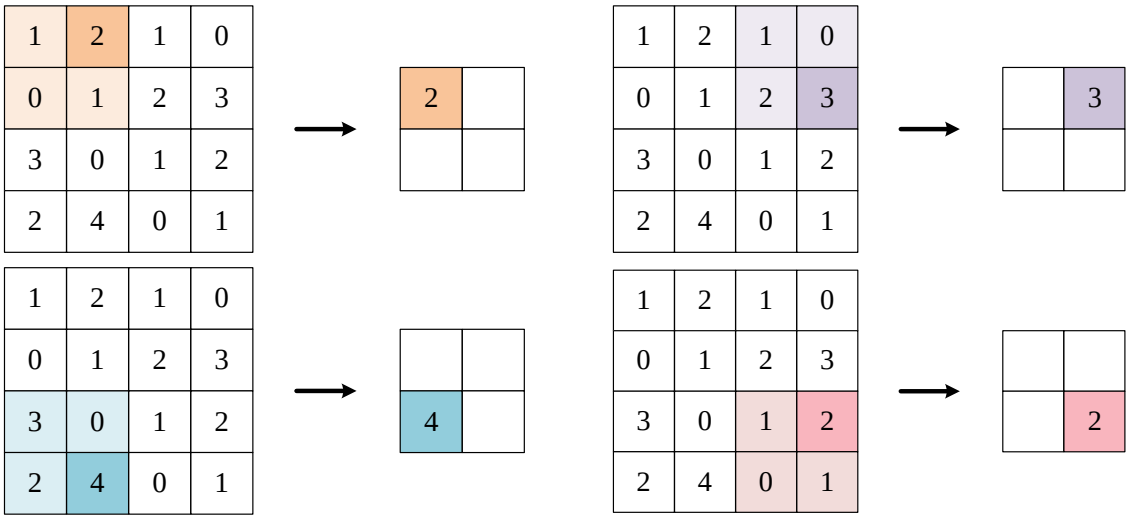


图 1-13 最大值汇聚

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

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

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

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

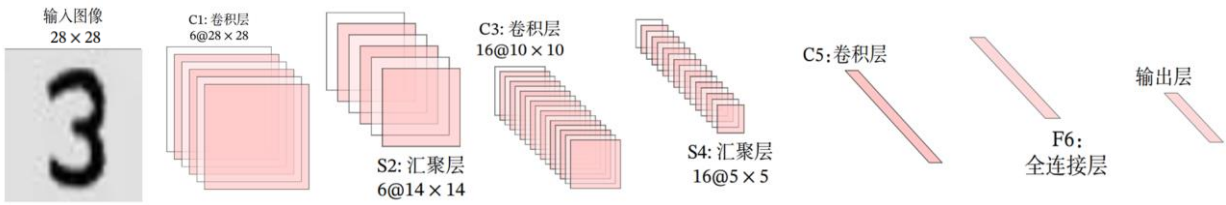


图 2-1 网络结构

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
```

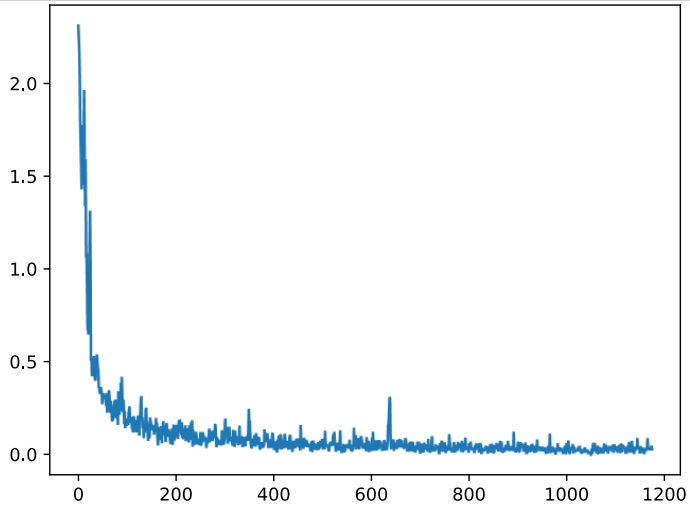
In [9] :
```
# 优化算法的选择
learning_rate = 0.9      # 设置学习率
optimizer = torch.optim.SGD(
        model.parameters(),
        lr = learning_rate,
)
```

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 %

三、AlexNet

3.1 网络结构

AlexNet 是第一个现代深度卷积网络模型，其首次使用了很多现代网络的技术方法，作为 2012 年 ImageNet 图像分类竞赛冠军，输入为 3×224×224 的图像，输出为 1000 个类别的条件概率。

考虑到如果使用 ImageNet 训练集会导致训练时间过长，这里使用稍低一档的 1×28×28 的 MNIST 数据集，并手动将其分辨率从 1×28×28 提到 1×224×224，同时输出从 1000 个类别降到 10 个，修改后的网络结构见表 3-1。

表 3-1 网络结构

层	In	C1	S2	C3	S4	C5
类型	输入层	卷积层	最大汇聚	卷积层	最大汇聚	卷积层
滤波器		96×1×11×11	3×3	256×96×5×5	3×3	384×256×3×3
步幅		4	2	1	2	1
填充		1		2		1
激活函数		ReLU		ReLU		ReLU
尺寸	1×224×224	96×54×54	96×26×26	256×26×26	256×12×12	384×12×12

层	C6	C7	S8	F9	F10	Out
类型	卷积层	卷积层	最大汇聚	全连接层	全连接层	全连接层
滤波器	384×384×3×3	256×384×3×3	3×3			
步幅	1	1	2			
填充	1	1				
激活函数	ReLU	ReLU		ReLU	ReLU	Softmax
尺寸	384×12×12	256×12×12	256×5×5	4096	4096	10

根据网络结构，在 PyTorch 的 nn.Sequential 中编写为

```
self.net = nn.Sequential(
        nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=1), nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.ReLU(),
        nn.Conv2d(384, 384, kernel_size=3, padding=1), nn.ReLU(),
        nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Flatten(),
        nn.Linear(6400, 4096), nn.ReLU(),
        nn.Dropout(p=0.5),                    # Dropout——随机丢弃权重
        nn.Linear(4096, 4096), nn.ReLU(),
        nn.Dropout(p=0.5),                    #  按概率 p 随机丢弃突触
        nn.Linear(4096, 10)
)
```

3.2 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.Resize(224),
        transforms.Normalize(0.1307, 0.3081)
])

# 下载训练集与测试集
train_Data = datasets.FashionMNIST(
        root = 'D:/Jupyter/dataset/mnist/',
        train = True,
        download = True,
        transform = transform
)
test_Data = datasets.FashionMNIST(
        root = 'D:/Jupyter/dataset/mnist/',
        train = False,
        download = True,
        transform = transform
)
```

In [4] :
```
# 批次加载器
train_loader = DataLoader(train_Data, shuffle=True, batch_size=128)
test_loader = DataLoader(test_Data, shuffle=False, batch_size=128)
```

3.3 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, 96, kernel_size=11, stride=4, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(96, 256, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(256, 384, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(384, 384, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Flatten(),
            nn.Linear(6400, 4096), nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096), nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 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, 224, 224))
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, 96, 54, 54])
ReLU output shape:          torch.Size([1, 96, 54, 54])
MaxPool2d output shape:     torch.Size([1, 96, 26, 26])
Conv2d output shape:        torch.Size([1, 256, 26, 26])
ReLU output shape:          torch.Size([1, 256, 26, 26])
MaxPool2d output shape:     torch.Size([1, 256, 12, 12])
Conv2d output shape:        torch.Size([1, 384, 12, 12])
ReLU output shape:          torch.Size([1, 384, 12, 12])
Conv2d output shape:        torch.Size([1, 384, 12, 12])
ReLU output shape:          torch.Size([1, 384, 12, 12])
```

Conv2d output shape:	torch.Size([1, 256, 12, 12])
ReLU output shape:	torch.Size([1, 256, 12, 12])
MaxPool2d output shape:	torch.Size([1, 256, 5, 5])
Flatten output shape:	torch.Size([1, 6400])
Linear output shape:	torch.Size([1, 4096])
ReLU output shape:	torch.Size([1, 4096])
Dropout output shape:	torch.Size([1, 4096])
Linear output shape:	torch.Size([1, 4096])
ReLU output shape:	torch.Size([1, 4096])
Dropout output shape:	torch.Size([1, 4096])
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')
```

3.4 训练网络

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

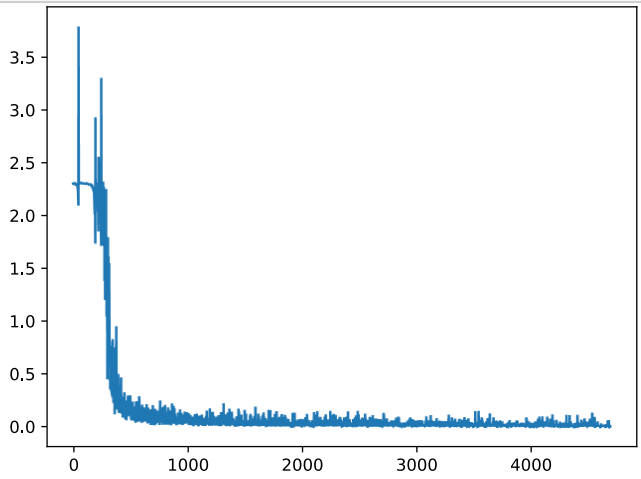
In [9] :
```
# 优化算法的选择
learning_rate = 0.1      # 设置学习率
optimizer = torch.optim.SGD(
        model.parameters(),
        lr = learning_rate,
)
```

In [10] :
```
# 训练网络
epochs = 10
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] :



3.5 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: 99.29999542236328 %
```

四、GoogLeNet

4.1 网络结构

2014 年，获得 ImageNet 图像分类竞赛的冠军是 GoogLeNet，其解决了一个重要问题：滤波器超参数选择困难，如何能够自动找到最佳的情况。

其在网络中引入了一个小网络——Inception 块，由 4 条并行路径组成，4 条路径互不干扰。这样一来，超参数最好的分支的那条分支，其权重会在训练过程中不断增加，这就类似于帮我们挑选最佳的超参数，如示例所示。

```
# 一个 Inception 块
class Inception(nn.Module):
    def __init__(self, in_channels):
        super(Inception, self).__init__()
        self.branch1 = nn.Conv2d(in_channels, 16, kernel_size=1)
        self.branch2 = nn.Sequential(
            nn.Conv2d(in_channels, 16, kernel_size=1),
            nn.Conv2d(16, 24, kernel_size=3, padding=1),
            nn.Conv2d(24, 24, kernel_size=3, padding=1)
        )
        self.branch3 = nn.Sequential(
            nn.Conv2d(in_channels, 16, kernel_size=1),
            nn.Conv2d(16, 24, kernel_size=5, padding=2)
        )
        self.branch4 = nn.Conv2d(in_channels, 24, kernel_size=1)

    def forward(self, x):
        branch1 = self.branch1(x)
        branch2 = self.branch2(x)
        branch3 = self.branch3(x)
        branch4 = self.branch4(x)
        outputs = [branch1, branch2, branch3, branch4]
        return torch.cat(outputs, 1)
```

此外，分支 2 和分支 3 上增加了额外 1×1 的滤波器，这是为了减少通道数，降低模型复杂度。

GoogLeNet 之所以叫 GoogLeNet，是为了向 LeNet 致敬，其网络结构为

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 10, kernel_size=5), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            Inception(in_channels=10),
            nn.Conv2d(88, 20, kernel_size=5), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            Inception(in_channels=20),
```

```
            nn.Flatten(),
            nn.Linear(1408, 10)
        )

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

4.2 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.FashionMNIST(
        root = 'D:/Jupyter/dataset/mnist/',
        train = True,
        download = True,
        transform = transform
)
test_Data = datasets.FashionMNIST(
        root = 'D:/Jupyter/dataset/mnist/',
        train = False,
        download = True,
        transform = transform
)
```

In [4]:
```
# 批次加载器
train_loader = DataLoader(train_Data, shuffle=True, batch_size=128)
test_loader = DataLoader(test_Data, shuffle=False, batch_size=128)
```

4.3 搭建神经网络

In [5] :
```
# 一个 Inception 块
class Inception(nn.Module):
    def __init__(self, in_channels):
        super(Inception, self).__init__()
        self.branch1 = nn.Conv2d(in_channels, 16, kernel_size=1)
        self.branch2 = nn.Sequential(
            nn.Conv2d(in_channels, 16, kernel_size=1),
            nn.Conv2d(16, 24, kernel_size=3, padding=1),
            nn.Conv2d(24, 24, kernel_size=3, padding=1)
        )
        self.branch3 = nn.Sequential(
            nn.Conv2d(in_channels, 16, kernel_size=1),
            nn.Conv2d(16, 24, kernel_size=5, padding=2)
        )
        self.branch4 = nn.Conv2d(in_channels, 24, kernel_size=1)

    def forward(self, x):
        branch1 = self.branch1(x)
        branch2 = self.branch2(x)
        branch3 = self.branch3(x)
        branch4 = self.branch4(x)
        outputs = [branch1, branch2, branch3, branch4]
        return torch.cat(outputs, 1)
```

In [6] :
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 10, kernel_size=5), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            Inception(in_channels=10),
            nn.Conv2d(88, 20, kernel_size=5), nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            Inception(in_channels=20),
            nn.Flatten(),
            nn.Linear(1408, 10)
        )

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

In [7] :
```
# 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 [7] :
```
Conv2d output shape:          torch.Size([1, 10, 24, 24])
ReLU output shape:            torch.Size([1, 10, 24, 24])
MaxPool2d output shape:       torch.Size([1, 10, 12, 12])
Inception output shape:       torch.Size([1, 88, 12, 12])
Conv2d output shape:          torch.Size([1, 20, 8, 8])
ReLU output shape:            torch.Size([1, 20, 8, 8])
MaxPool2d output shape:       torch.Size([1, 20, 4, 4])
Inception output shape:       torch.Size([1, 88, 4, 4])
Flatten output shape:         torch.Size([1, 1408])
Linear output shape:          torch.Size([1, 10])
```

In [8] :
```
# 创建子类的实例，并搬到 GPU 上
model = CNN().to('cuda:0')
```

4.4 训练网络

In [9] :
```
# 损失函数的选择
loss_fn = nn.CrossEntropyLoss()
```

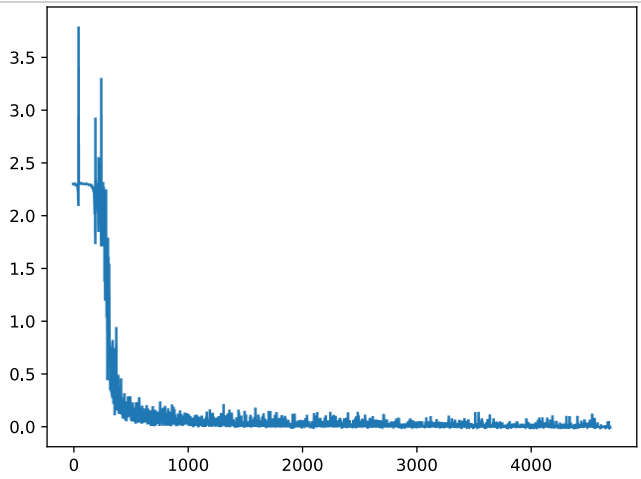
In [10] :
```
# 优化算法的选择
learning_rate = 0.1      # 设置学习率
optimizer = torch.optim.SGD(
        model.parameters(),
        lr = learning_rate,
)
```

In [11] :
```
# 训练网络
epochs = 10
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 [11] :



4.5 测试网络

In [12] :

```
# 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 [12] :

```
测试集精准度: 98.91999816894531 %
```

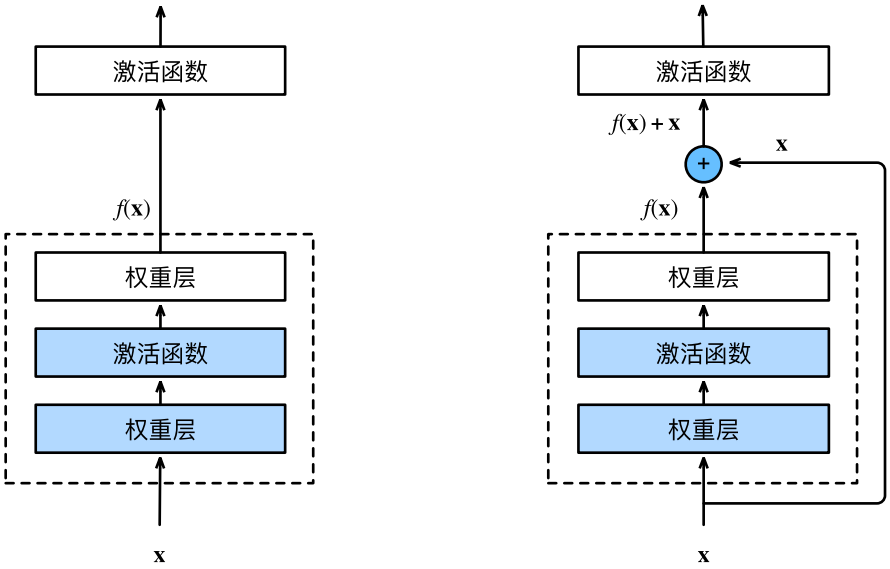
五、ResNet

5.1 网络结构

残差网络（Residual Network，ResNet）荣获 2015 年的 ImageNet 图像分类竞赛冠军，其可以缓解深度神经网络中增加深度带来的“梯度消失”问题。

在反向传播计算梯度时，梯度是不断相乘的，假如训练到后期，各层的梯度均小于 1，则其相乘起来就会不断趋于 0。因此，深度学习的隐藏层并非越多越好，隐藏层越深，梯度越趋于 0，此之谓“梯度消失”。

而残差块将某模块的输入 x 引到输出 y 处，使原本的梯度 dy/dx 变成了 $(dy+dx)/dx$，也即 $(dy/dx+1)$，这样一来梯度就不会消失了。



（a）普通块　　（b）残差块

图 6-1 残差块的结构

5.2 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.FashionMNIST(
        root = 'D:/Jupyter/dataset/mnist/',
        train = True,
        download = True,
        transform = transform
)
test_Data = datasets.FashionMNIST(
        root = 'D:/Jupyter/dataset/mnist/',
        train = False,
        download = True,
        transform = transform
)
```

In [4] :

```
# 批次加载器
train_loader = DataLoader(train_Data, shuffle=True, batch_size=128)
test_loader = DataLoader(test_Data, shuffle=False, batch_size=128)
```

5.3 搭建神经网络

In [5] :

```
# 残差块
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(channels, channels, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(channels, channels, kernel_size=3, padding=1),
        )

    def forward(self, x):
        y = self.net(x)
        return nn.functional.relu(x+y)
```

In [6] :

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5), nn.ReLU(),
            nn.MaxPool2d(2), ResidualBlock(16),
            nn.Conv2d(16, 32, kernel_size=5), nn.ReLU(),
            nn.MaxPool2d(2), ResidualBlock(32),
            nn.Flatten(),
            nn.Linear(512, 10)
        )
```

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

In [7] :
```
# 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 [7] :
```
Conv2d output shape:          torch.Size([1, 16, 24, 24])
ReLU output shape:            torch.Size([1, 16, 24, 24])
MaxPool2d output shape:       torch.Size([1, 16, 12, 12])
ResidualBlock output shape:   torch.Size([1, 16, 12, 12])
Conv2d output shape:          torch.Size([1, 32, 8, 8])
ReLU output shape:            torch.Size([1, 32, 8, 8])
MaxPool2d output shape:       torch.Size([1, 32, 4, 4])
ResidualBlock output shape:   torch.Size([1, 32, 4, 4])
Flatten output shape:         torch.Size([1, 512])
Linear output shape:          torch.Size([1, 10])
```

In [8] :
```
# 创建子类的实例，并搬到 GPU 上
model = CNN().to('cuda:0')
```

5.4 训练网络

In [9] :
```
# 损失函数的选择
loss_fn = nn.CrossEntropyLoss()
```

In [10] :
```
# 优化算法的选择
learning_rate = 0.1      # 设置学习率
optimizer = torch.optim.SGD(
    model.parameters(),
    lr = learning_rate,
)
```

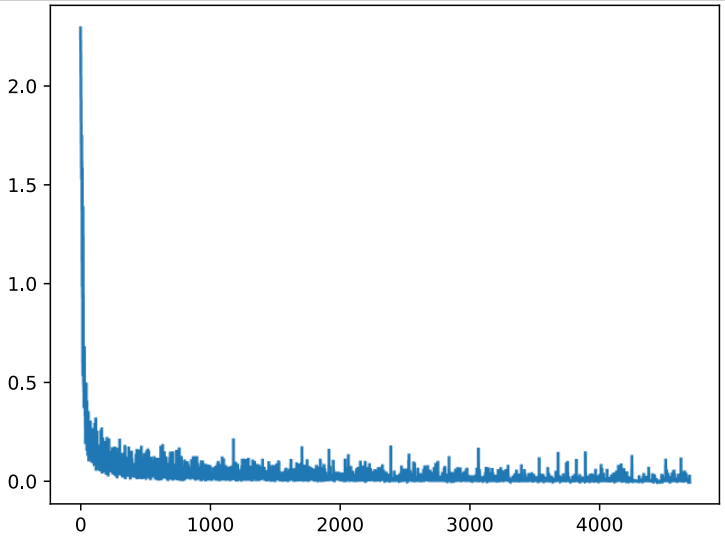
In [11] :
```
# 训练网络
epochs = 10
losses = []          # 记录损失函数变化的列表

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 [11] :



5.5 测试网络

In [12] :

```
# 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 [12] : 测试集精准度: 98.91999816894531 %