



南開大學
Nankai University

深度学习实验报告

实验名称：卷积神经网络

姓名：张恒硕

学号：2212266

专业：智能科学与技术

目录

一、 实验目的	3
二、 实验原理	3
1. 卷积层 (Convolutional Layer)	3
2. 激活函数 (Activation Function)	3
3. 池化层 (Pooling Layer)	5
4. 全连接层 (Fully Connected Layer)	5
5. LeNet	6
三、 实验步骤	6
1. 数据集导入与划分	6
2. 神经网络搭建	6
3. 模型	6
4. 对于代码的说明	7
四、 基础代码	7
1. 网络部分	7
2. 训练、评估部分	8
3. 主函数部分	10
五、 调试训练与分析	11
1. 结果展示 (LeNet.py)	11
2. sigmoid->relu (LeNet.py)	11
3. sigmoid+batch normalization (batch_normalization.py)	12
4. 调整卷积核大小 (LeNet.py)	13
5. 调整输出通道数量 (LeNet.py)	14
6. average polling->max polling (LeNet.py)	16
六、 附加题	17
1. VGGNet (VGGNet.py)	17
2. NIN (NIN.py)	20
3. GoogleNet (GoogleNet.py)	22
4. ResNet (ResNet.py)	26
5. 特征可视化 (feature.py)	30

一、实验目的

实现一个卷积神经网络 LeNet，对 fashion_mnist 数据集进行图像分类，并对训练情况进行分析，尝试各种改进手段并分析。另外测试其他卷积神经网络的效果，并尝试对卷积特征结果进行可视化。

二、实验原理

卷积神经网络 (Convolutional Neural Network, CNN) 是一种在计算机视觉领域有重要作用的深度学习模型，常用于图像识别、物体检测、语义分割等任务。

1. 卷积层 (Convolutional Layer)

卷积层通过一组可学习的小型滤波器 (也称为内核或核，下文称作卷积核) 与输入数据进行局部连接，用于检测输入中的特定模式或特征。每个卷积核在输入数据上滑动，计算点乘并求和来生成特征映射。而多个卷积核的组合可以产生多通道特征映射。以下公式和图 2.1 给出了计算公式和具体的例子。

原图像: $H_0 = N_0 \times M_0 \times A_0$

卷积核: B 个, $F \times F \times A_0$

Padding: P

Stride: S

新图像: $N_1 = \frac{N_0 + 2P - F}{S} + 1$, $A_1 = B$

参数量: $F \times F \times A_0 \times B$

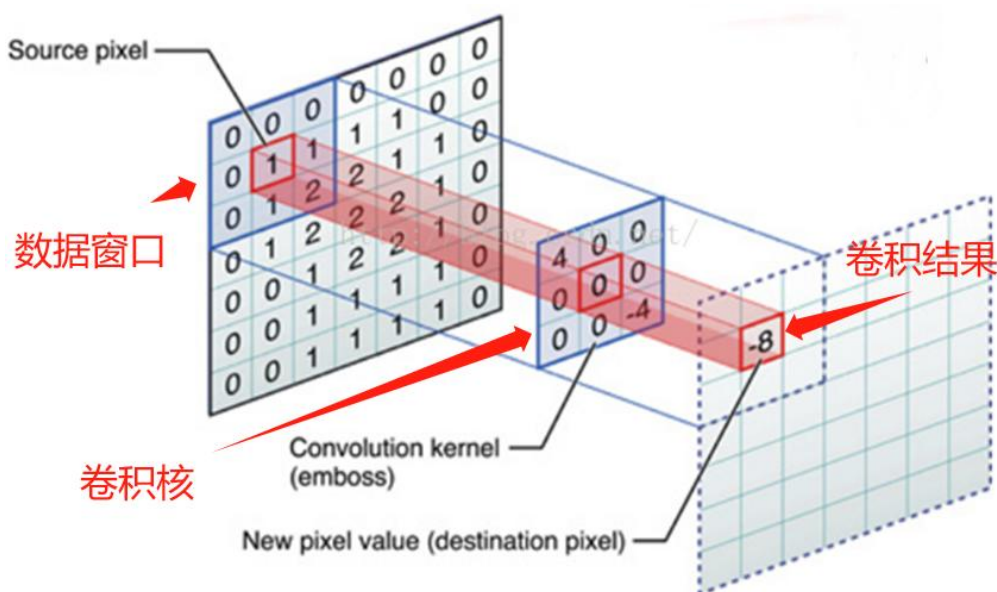
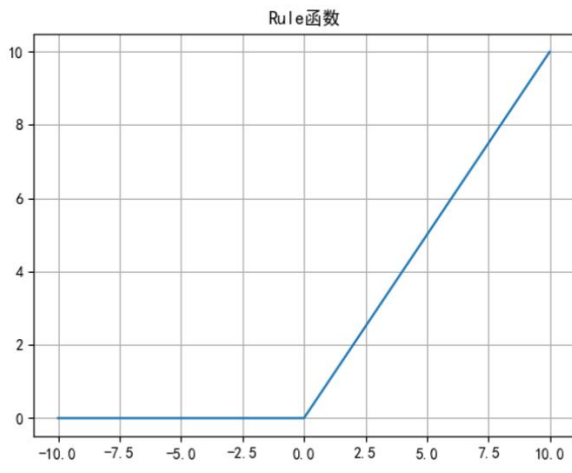
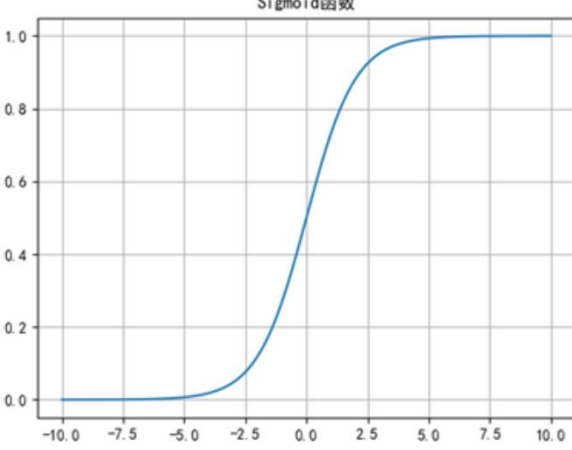
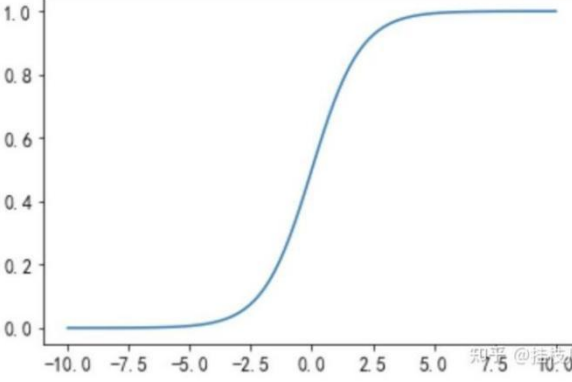
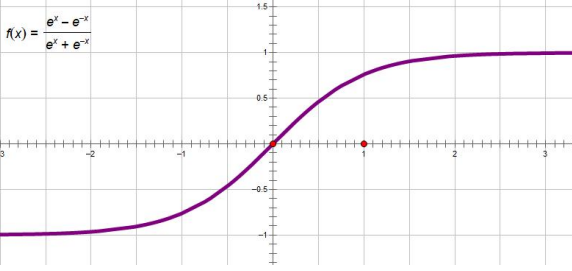


图 2.1 卷积过程示意图

2. 激活函数 (Activation Function)

对卷积结果应用激活函数，可以引入非线性特性，从而使模型能够拟合复杂的模式。上一个报告已经对激活函数有了具体的讲解，这里仅截取部分进行简单展示，如下表 1。

表 1 常用激活函数

激活函数	函数	函数图像
ReLU	$y = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$	 <p>The graph of the ReLU function, titled "Rule函数", shows a blue line that is zero for all negative values of x and increases linearly with a slope of 1 for all positive values of x. The x-axis ranges from -10.0 to 10.0, and the y-axis ranges from 0 to 10.</p>
Sigmoid	$y = \frac{1}{1 + e^{-z}}$	 <p>The graph of the Sigmoid function, titled "Sigmoid函数", shows a blue S-shaped curve that maps any real-valued number into the range (0, 1). The curve passes through (0, 0.5) and approaches 0 as x goes to negative infinity and 1 as x goes to positive infinity. The x-axis ranges from -10.0 to 10.0, and the y-axis ranges from 0.0 to 1.0.</p>
softmax	$y(x_i) = \frac{e^{x_i}}{\sum_{i=1}^n e^{x_i}}$	 <p>The graph of the Softmax function shows a blue S-shaped curve, similar to the Sigmoid function, mapping values into the range (0, 1). The x-axis ranges from -10.0 to 10.0, and the y-axis ranges from 0.0 to 1.0. A watermark "知至 @ 拾玖" is visible in the bottom right corner of the plot area.</p>
tanh	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	 <p>The graph of the Tanh function, titled "f(x) = (e^x - e^-x) / (e^x + e^-x)", shows a purple S-shaped curve that maps any real-valued number into the range (-1, 1). The curve passes through (0, 0) and approaches -1 as x goes to negative infinity and 1 as x goes to positive infinity. The x-axis ranges from -3 to 3, and the y-axis ranges from -1 to 1.5.</p>

3. 池化层 (Pooling Layer)

池化层通常位于卷积层之后，用于减少特征映射的空间维度，从而降低计算复杂度，并有助于提取平移不变性。常用的池化方法有如下两种：

- 最大池化 (Max Pooling)：选取区域的最大值作为代表性的特征值，如下图 2.3-1。

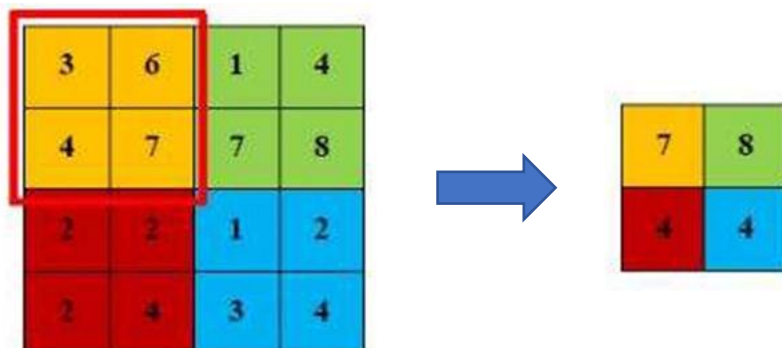


图 2.3-1 最大池化示意图

- 平均池化 (Average Pooling)：计算区域的平均值作为代表性的特征值，如下图 2.3-2。

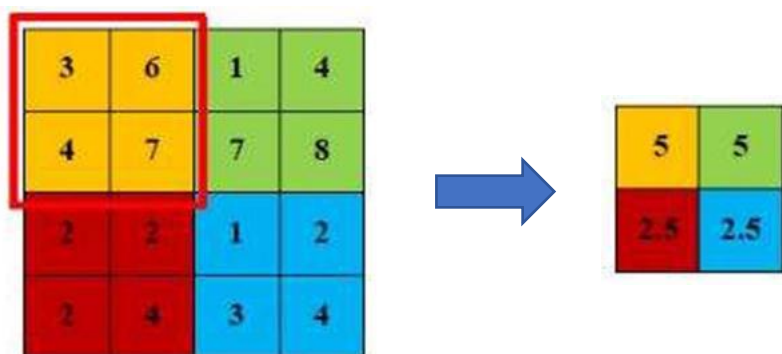


图 2.3-2 平均池化示意图

4. 全连接层 (Fully Connected Layer)

在传统的 CNN 架构中，卷积层和池化层之后通常是全连接层，其用于将特征映射展平成一维向量，并映射到分类标签空间。然而其在现代网络中的使用越来越少，取而代之的是全局池化或更复杂的结构。以下图 2.4 给出了相应的示意图。

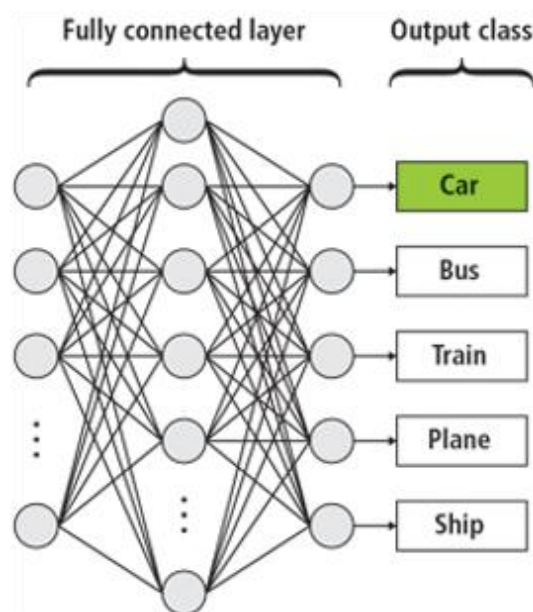


图 2.4 全连接层与输出层示意图

5. LeNet

LeNet (LeNet-5) 由 Yann Lecun 提出，作为经典的卷积神经网络，是现代卷积神经网络的起源之一，最初用于手写字符识别。其包含一个输入层、两个卷积层、两个池化层、三个全连接层，其中最后一个全连接层为输出层，以下图 2.5 给出了相应结构。

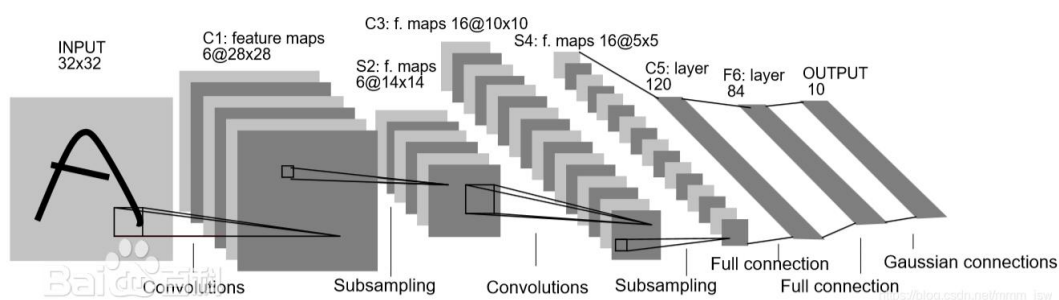


图 2.5 LeNet 结构

三、实验步骤

1. 数据集导入与划分

从包中导入 `fashion_mnist` 数据集，其是衣服的分类数据集（见五、1.），之后划分训练集、验证集、测试集，并设定每次训练的规模。

2. 神经网络搭建

(1) 激活函数：原网络使用 `sigmoid` 激活函数，另尝试了 `relu` 激活函数（见五、2.）。另测试了 `sigmoid` 与 `batch normalization` 协作的性能（见五、3.）。

(2) 池化层：原网络使用平均池化，另尝试了最大池化（见五、6.）。

(3) 网络与初始化：设计各层网络，并初始化网络参数。改变卷积核大小（见五、4.）、通道数量进行调试（见五、5.）。

3. 模型

(1) 优化器：随机梯度下降优化器。

(2) 损失函数：交叉熵损失函数。

4. 对于代码的说明

代码的训练、评估部分为简化版的书本代码，各种卷积网络的搭建部分选取或简化了相应的原始网络。代码省略了测试部分，以训练正确率、验证正确率和损失值来衡量效果。

四、基础代码

1. 网络部分

```
# LeNet

net = nn.Sequential(

    nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),

    nn.AvgPool2d(kernel_size=2, stride=2),

    # nn.MaxPool2d(kernel_size=2, stride=2),

    nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),

    nn.AvgPool2d(kernel_size=2, stride=2),

    # nn.MaxPool2d(kernel_size=2, stride=2),

    nn.Flatten(),

    nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),

    nn.Linear(120, 84), nn.Sigmoid(),

    nn.Linear(84, 10))

X = torch.rand(size=(1, 1, 28, 28), dtype=torch.float32)

for layer in net:

    X = layer(X)

    print(layer.__class__.__name__, 'output shape: \t', X.shape)
```

网络包含一个输入层、两个卷积层、两个平均池化层、三个全连接层，具体如下：

Conv2d 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])
AvgPool2d output shape:   torch.Size([1, 16, 5, 5])
Flatten output shape:     torch.Size([1, 400])
Linear output shape:      torch.Size([1, 120])
Linear output shape:      torch.Size([1, 84])
Linear output shape:      torch.Size([1, 10])
```

其在经历两次卷积、池化后，展平进行三次全连接层，最终输出结果。

2. 训练、评估部分

评估

```
def evaluate(net, data_iter, device):

    net.eval()

    metric = d2l.Accumulator(2)  # 正确预测的数量，总预测的数量

    with torch.no_grad():

        for X, y in data_iter:

            if isinstance(X, list):

                X = [x.to(device) for x in X]

            else:

                X = X.to(device)

            y = y.to(device)

            metric.add(d2l.accuracy(net(X), y), y.numel())

    return metric[0] / metric[1]
```

训练

```
def train(net, train_iter, test_iter, num_epochs, lr, device):

    start_time = time.time()

    # 初始化权重
```



```

def init_weights(m):

    if type(m) == nn.Linear or type(m) == nn.Conv2d:

        nn.init.xavier_uniform_(m.weight)

# 训练初始化

net.apply(init_weights)

net.to(device)

optimizer = torch.optim.SGD(net.parameters(), lr=lr) # 优化器：随机梯度下降

loss = nn.CrossEntropyLoss() # 损失函数：交叉熵损失

animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], legend=['train loss',
'train acc', 'test acc'])

timer, num_batches = d2l.Timer(), len(train_iter)

# 训练

for epoch in range(num_epochs):

    metric = d2l.Accumulator(3) # 训练损失之和，训练准确率之和，样本数

    net.train()

    for i, (X, y) in enumerate(train_iter):

        timer.start()

        optimizer.zero_grad()

        X, y = X.to(device), y.to(device)

        y_hat = net(X)

        l = loss(y_hat, y)

        l.backward()

```

```

optimizer.step()

with torch.no_grad():

    metric.add(1 * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])

timer.stop()

train_l = metric[0] / metric[2]

train_acc = metric[1] / metric[2]

if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:

    animator.add(epoch + (i + 1) / num_batches, (train_l, train_acc, None))

test_acc = evaluate(net, test_iter, device)

animator.add(epoch + 1, (None, None, test_acc))

print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, test acc {test_acc:.3f}')

print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec on {str(device)}')

end_time = time.time()

plt.show()

return end_time - start_time

```

- 保证模型在 GPU 下高速训练。
- 在训练模式下，以随机梯度下降为优化器、交叉熵损失为损失函数。
- 前向传播计算预测值，清除梯度，反向传播计算损失相对于模型参数的梯度，更新参数。
- 保留训练过程中的训练正确率、验证正确率、损失值，最后输出图像。

3. 主函数部分

```

# 主函数

batch_size = 256

train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

lr, num_epochs = 0.9, 20

```

```
time = train(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())

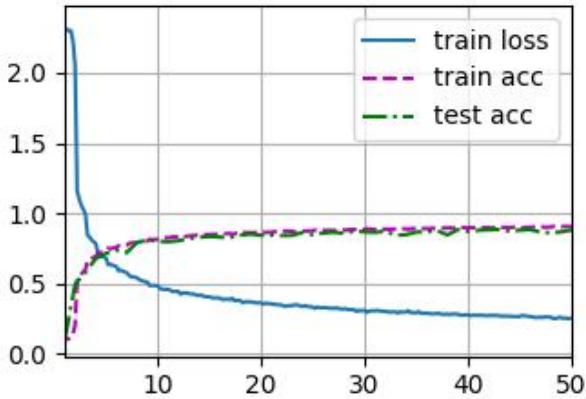
print(f"代码运行时间:{time}秒")
```

五、调试训练与分析

(注：括号中对应相应的代码)

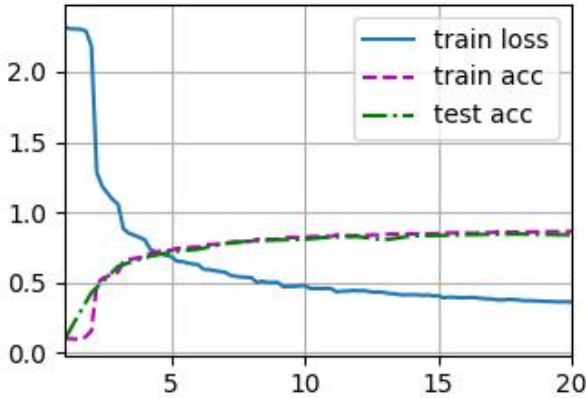
1. 结果展示 (LeNet. py)

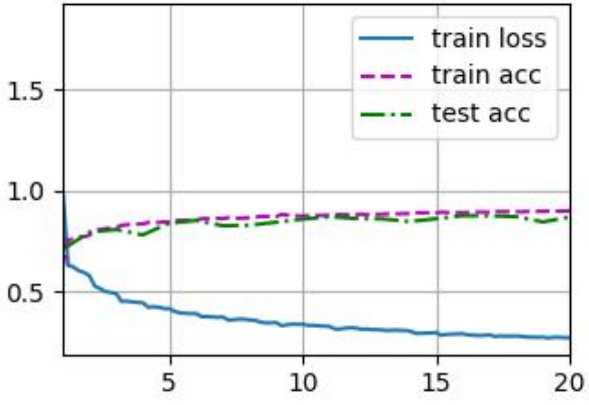
以下首先对 LeNet 的性能进行测试。

LeNet	
损失	0.254
训练正确率	0.905
验证正确率	0.882
代码运行效率 (例/每秒)	59383.5
代码运行时间 (秒)	325.3325755596161
分析：在 fashion-mnist 数据集的测试任务中，LeNet 表现出较好性能。其运行效率较快，结果正确率较高，在 20 代时便基本收敛。	

2. sigmoid->relu (LeNet. py)

作为常用的激活函数，sigmoid 激活函数和 relu 激活函数都可以给网络带来非线性，改善网络的性能，但是二者对本任务的功效可能有一定差距，本部分对这种差异展开研究。

LeNet (sigmoid, lr=0.9)	
-------------------------	--

损失	0.362
训练正确率	0.865
验证正确率	0.840
代码运行效率 (例/每秒)	53194.1
代码运行时间 (秒)	132.89130353927612
LeNet (relu, lr=0.1)	
损失	0.270
训练正确率	0.900
验证正确率	0.870
代码运行效率 (例/每秒)	53283.9
代码运行时间 (秒)	132.34272050857544
<p>分析: relu 激活函数可以解决梯度消失问题, 加速收敛, 产生稀疏激活以降低复杂度, 并提供比 sigmoid 激活函数更强的非线性。以上两组实验结果都是运行 20 代的结果, 可以发现 relu 激活函数的正确率已经逼近 sigmoid 激活函数 50 代的结果, 这验证了 relu 函数的优点。由于二者对学习率有不同的要求, 这里的横向比较并不完善, relu 激活函数也可能因为使部分神经元永远无法激活而形成死区造成误差。</p>	

3. sigmoid+batch normalization (batch_normalization.py)

Batch Normalization 通过对每个小批量样本进行归一化操作, 使得输入的特征具有零均值和单位方差, 这有助于减少梯度消失和梯度爆炸问题, 从而加速神经网络的收敛。其可以提高模型稳定性、泛化能力, 并减少对其他正则化技术的依赖。

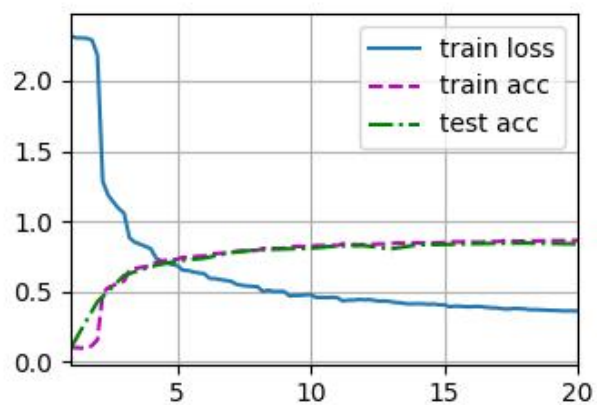
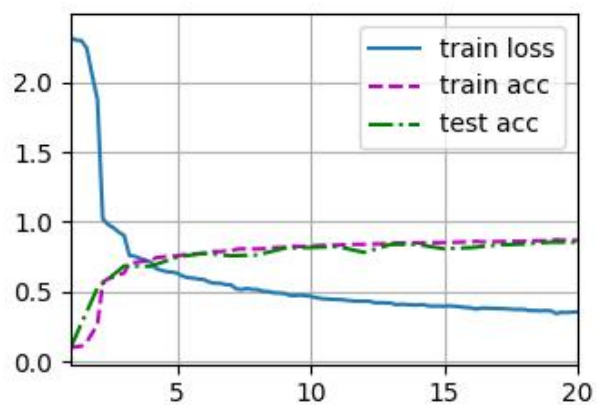
在每一个卷积层、非输出层全连接层, 调用以下函数进行归一化, 其参数是该层网络的输出通道数。

```
nn.BatchNorm1d()
```

LeNet (无 batch normalization)	
损失	0.362
训练正确率	0.865
验证正确率	0.840
代码运行效率 (例/每秒)	53194.1
代码运行时间 (秒)	132.89130353927612
LeNet (有 batch normalization)	
损失	0.185
训练正确率	0.931
验证正确率	0.894
代码运行效率 (例/每秒)	44522.8
代码运行时间 (秒)	138.26381587982178
分析：观察有 batch normalization 组的实验结果，可以发现，训练正确率和验证正确率存在较大偏差，表明模型出现过拟合现象。这是因为本图像分类任务较为简单，使用 batch normalization 就会过分关注训练数据，导致鲁棒性和随机性差。	

4. 调整卷积核大小 (LeNet.py)

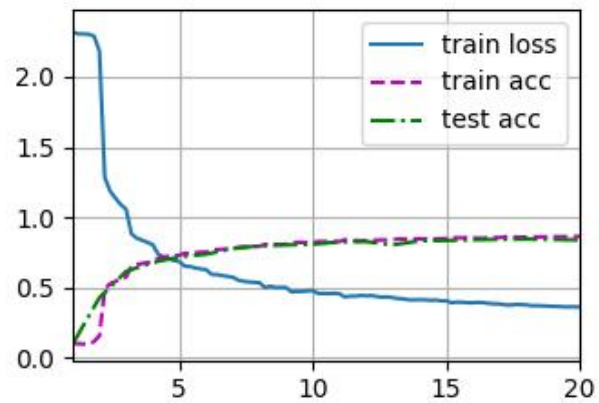
较大的卷积核可以捕捉更大的区域信息，其参数较多，较为复杂，对小数据集容易过拟合；而较小的卷积核可以捕捉细节特征，其参数较小，更轻量级。以下对不同卷积核的性能进行比较。

LeNet (k=5)	
损失	0.362
训练正确率	0.865
验证正确率	0.840
代码运行效率 (例/每秒)	53194.1
代码运行时间 (秒)	132.89130353927612
LeNet (k=3)	
损失	0.354
训练正确率	0.867
验证正确率	0.853
代码运行效率 (例/每秒)	65027.0
代码运行时间 (秒)	97.16569018363953
分析: 两个卷积核的结果差不多, 稍小卷积核的效果略好一点。因本数据集的图像过小, 不适宜尝试更大的卷积核。	

5. 调整输出通道数量 (LeNet.py)

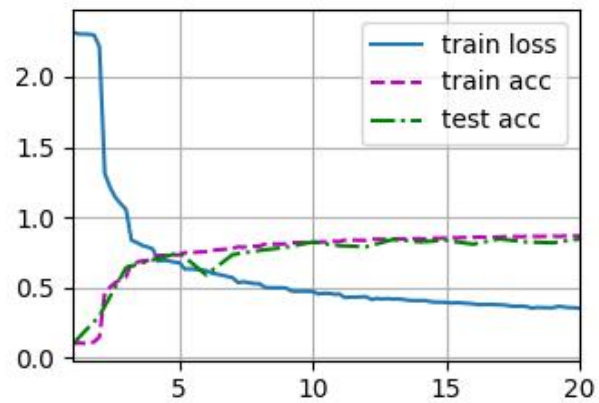
通道数的增加, 会带来更多的参数, 计算更耗时, 但同时, 也可以使模型的学习能力更强, 提高表达能力, 甚至会带来过拟合。对于简单的图像分类任务, 少量通道即可。以下增加通道数来进行调试。

LeNet
(1-6-16)



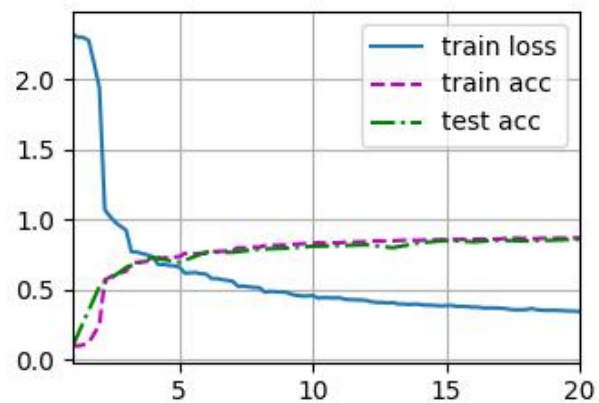
损失	0.362
训练正确率	0.865
验证正确率	0.840
代码运行效率 (例/每秒)	53194.1
代码运行时间 (秒)	132.89130353927612

LeNet
(1-10-20)



损失	0.354
训练正确率	0.870
验证正确率	0.847
代码运行效率 (例/每秒)	50520.1
代码运行时间 (秒)	102.89937853813171

LeNet
(1-20-50)



损失	0.345
----	-------

训练正确率	0.871
验证正确率	0.861
代码运行效率 (例/每秒)	45935.4
代码运行时间 (秒)	104.79499673843384
分析：以上测试中并没有出现过拟合现象，而结果也与通道数基本成正比。	

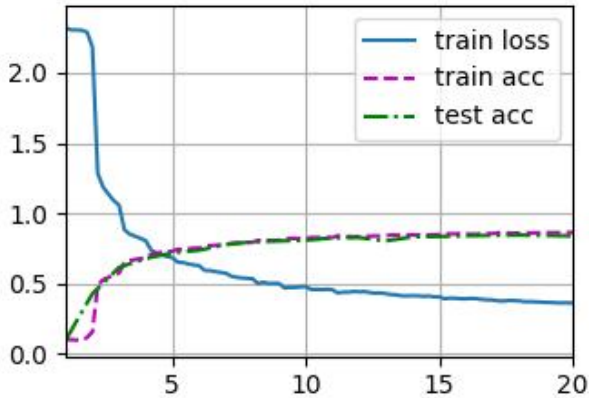
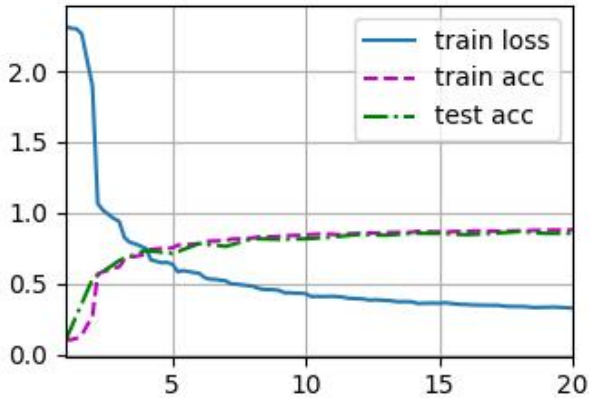
6. average pooling->max pooling (LeNet.py)

平均池化和最大池化是常用的两种简单池化手段，以下比较二者的性能差异。
将

```
nn.AvgPool2d(kernel_size=2, stride=2),
```

替换为

```
nn.MaxPool2d(kernel_size=2, stride=2),
```

LeNet (平均池化)	
损失	0.362
训练正确率	0.865
验证正确率	0.840
代码运行效率 (例/每秒)	53194.1
代码运行时间 (秒)	132.89130353927612
LeNet (最大池化)	
损失	0.326
训练正确率	0.879
验证正确率	0.855


```

        layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1))

        layers.append(nn.ReLU())

        in_channels = out_channels

    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))

    return nn.Sequential(*layers)

# VGGNet

# conv_arch: ((1, 1, 64), (1, 64, 128), (2, 128, 256), (2, 256, 512), (2, 512,
512)) -> ((1, 1, 32), (1, 32, 64))

class VGGNet(nn.Module):

    def __init__(self, conv_arch=((1, 1, 32), (1, 32, 64)), num_classes=10):

        super(VGGNet, self).__init__()

        self.conv_arch = conv_arch

        self.features = self.make_layers(conv_arch)

        self.classifier = nn.Sequential(

            nn.Flatten(),

            nn.Linear(64 * 7 * 7, 1024),

            nn.ReLU(),

            nn.Dropout(0.5),

            nn.Linear(1024, 512),

            nn.ReLU(),

            nn.Dropout(0.5),

```

```

nn.Linear(512, num_classes)

)

def make_layers(self, conv_arch):

    layers = []

    in_channels = 1

    for (num_convs, in_channel, out_channel) in conv_arch:

        layers.append(vgg_block(num_convs, in_channel, out_channel))

        in_channels = out_channel

    return nn.Sequential(*layers)

def forward(self, x):

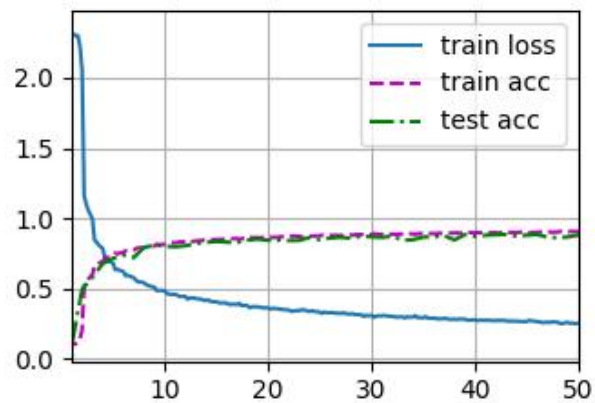
    x = self.features(x)

    x = self.classifier(x)

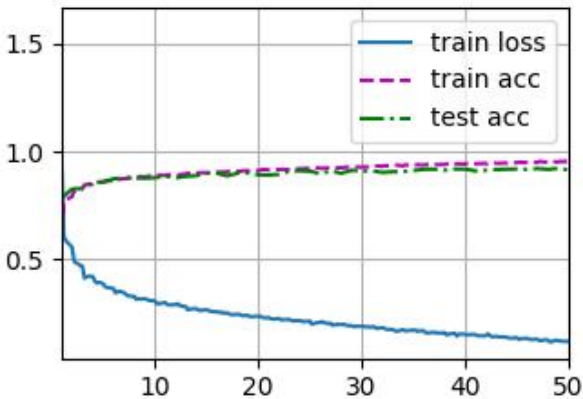
    return x

```

LeNet



损失	0.254
训练正确率	0.905
验证正确率	0.882
代码运行效率 (例/每秒)	59383.5
代码运行时间 (秒)	325.3325755596161

VGGNet	
损失	0.120
训练正确率	0.955
验证正确率	0.915
代码运行效率 (例/每秒)	38496.9
代码运行时间 (秒)	265.7087256908417
分析：对比发现，简化版的 VGGNet 的性能仍十分优越。其损失和正确率都优于 LeNet，但进行同样轮次却更快，这是因为每一轮次的参数量不同导致的。另外，并没有出现过拟合现象。	

2. NIN (NIN.py)

NiN (Network in Network)，是 Min Lin, Qiang Chen 和 Shuicheng Yan 在 2013 年提出的一种深度卷积神经网络。其采用 1×1 卷积，大幅度减少网络参数，提高模型的效率和泛化能力。下图 6.2 给出了它的具体结构。

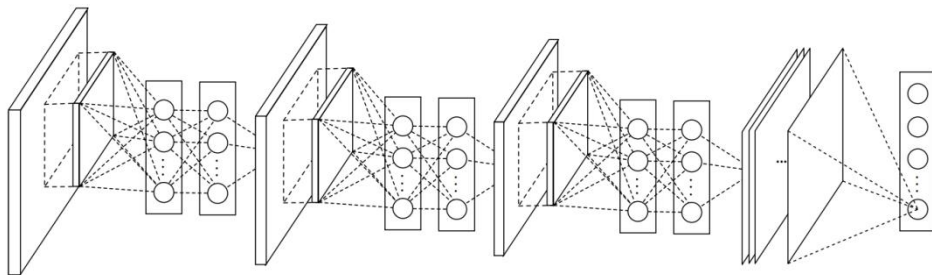


图 6.1 NiN 结构

在本次实验中，由于待分类图像较小，原网络会将图像卷积没，因此对卷积核大小进行了修改。以下是网络部分的代码。

```
# 定义基本卷积块

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(),
```

```

        nn.Conv2d(out_channels, out_channels, kernel_size=1), # 1x1 卷积

        nn.ReLU(),

        nn.Conv2d(out_channels, out_channels, kernel_size=1), # 1x1 卷积

        nn.ReLU()

    )

    return blk

# 定义 NIN 模型

# nn.MaxPool2d(kernel_size=3, stride=2)*3->nn.MaxPool2d(kernel_size=2, stride=2)*2

class NIN(nn.Module):

    def __init__(self, num_classes=10):

        super(NIN, self).__init__()

        self.net = nn.Sequential(

            nin_block(1, 96, kernel_size=5, stride=1, padding=2), # 输出: 28x28

            nn.MaxPool2d(kernel_size=2, stride=2), # 输出: 14x14

            nin_block(96, 256, kernel_size=3, stride=1, padding=1), # 输出: 14x14

            nn.MaxPool2d(kernel_size=2, stride=2), # 输出: 7x7

            nin_block(256, 384, kernel_size=3, stride=1, padding=1), # 输出: 7x7

            nn.Dropout(0.5),

            nn.Conv2d(384, num_classes, kernel_size=3, stride=1, padding=1),

            nn.AdaptiveAvgPool2d((1, 1)),

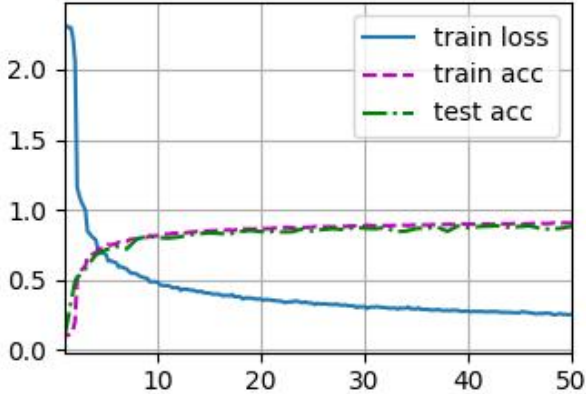
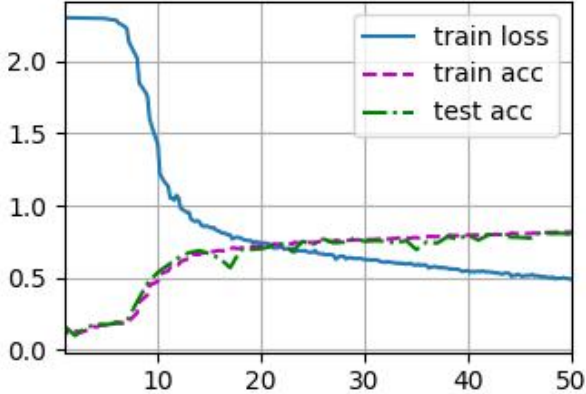
            nn.Flatten()

```

```
)

def forward(self, X):

    return self.net(X)
```

LeNet	
损失	0.254
训练正确率	0.905
验证正确率	0.882
代码运行效率 (例/每秒)	59383.5
代码运行时间 (秒)	325.3325755596161
NIN	
损失	0.491
训练正确率	0.819
验证正确率	0.801
代码运行效率 (例/每秒)	5194.7
代码运行时间 (秒)	814.3975534439087
分析: 变更的 NIN 在本任务中的表现并不好, 这可能是相关参数设置不当的原因。但是, 作为特点的 1x1 卷积在本任务中并没有针对性, 也可能导致效果不佳。	

3. GoogLeNet (GoogLeNet.py)

GoogLeNet (Inception-v1), 是 Google 设计的一种深度卷积神经网络, 在 2014 年 ImageNet 中获得第一名。其主要目标是在不显著增加计算成本的前提下,

提高网络的深度和宽度,从而提高模型的准确率。下图 6.3 给出了它的具体结构。

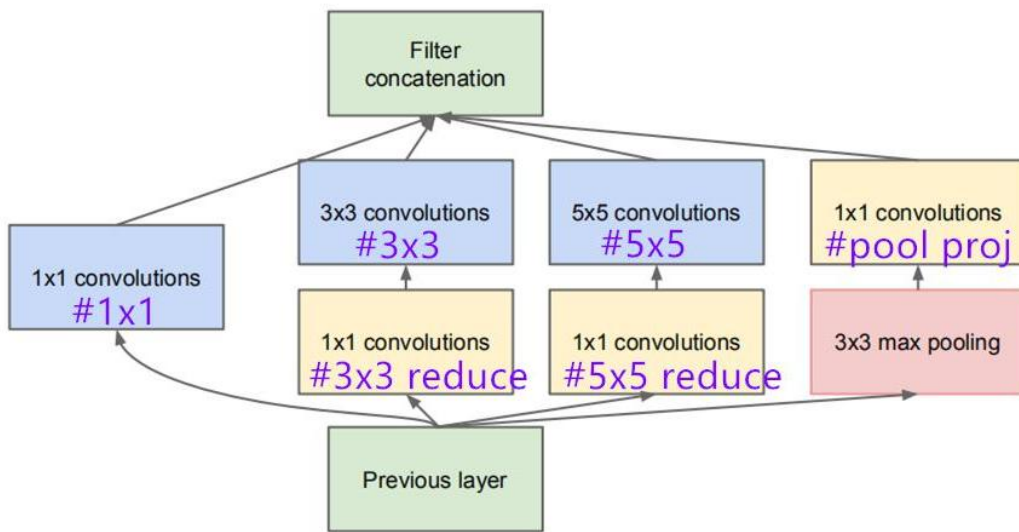


图 6.3 GoogLeNet 结构

本次实现的 GoogLeNet 并没有使用辅助分类器, 以下是网络部分的代码。

```
# Inception 块

class Inception(nn.Module):

    def __init__(self, in_channels, ch1, ch2_in, ch2_out, ch3_in, ch3_out, ch4_out,
**kwargs):

        super(Inception, self).__init__(**kwargs)

        # 1x1 卷积层

        self.p1_1 = nn.Conv2d(in_channels, ch1, kernel_size=1)

        # 1x1 卷积层+3x3 卷积层

        self.p2_1 = nn.Conv2d(in_channels, ch2_in, kernel_size=1)

        self.p2_2 = nn.Conv2d(ch2_in, ch2_out, kernel_size=3, padding=1)

        # 1x1 卷积层+5x5 卷积层

        self.p3_1 = nn.Conv2d(in_channels, ch3_in, kernel_size=1)

        self.p3_2 = nn.Conv2d(ch3_in, ch3_out, kernel_size=5, padding=2)

        # 3x3 最大池化层+1x1 卷积层
```

```

        self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)

        self.p4_2 = nn.Conv2d(in_channels, ch4_out, kernel_size=1)

    def forward(self, x):

        p1 = self.p1_1(x)

        p2 = self.p2_2(F.relu(self.p2_1(x)))

        p3 = self.p3_2(F.relu(self.p3_1(x)))

        p4 = self.p4_2(self.p4_1(x))

        return torch.cat((p1, p2, p3, p4), dim=1)

```

GoogleNet

```

net = nn.Sequential(

    nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=1),

    nn.ReLU(),

    nn.MaxPool2d(kernel_size=3, stride=2, padding=1),

    nn.Conv2d(64, 64, kernel_size=1),

    nn.Conv2d(64, 192, kernel_size=3, padding=1),

    nn.MaxPool2d(kernel_size=3, stride=2, padding=1),

    Inception(192, 64, 96, 128, 16, 32, 32),

    Inception(256, 128, 128, 192, 32, 96, 64),

    nn.MaxPool2d(kernel_size=3, stride=2, padding=1),

    Inception(480, 192, 96, 208, 16, 48, 64),

    Inception(512, 160, 112, 224, 24, 64, 64),

```



```

Inception(512, 128, 128, 256, 24, 64, 64),

Inception(512, 112, 144, 288, 32, 64, 64),

Inception(528, 256, 160, 320, 32, 128, 128),

nn.MaxPool2d(kernel_size=3, stride=2, padding=1),

Inception(832, 256, 160, 320, 32, 128, 128),

Inception(832, 384, 192, 384, 48, 128, 128),

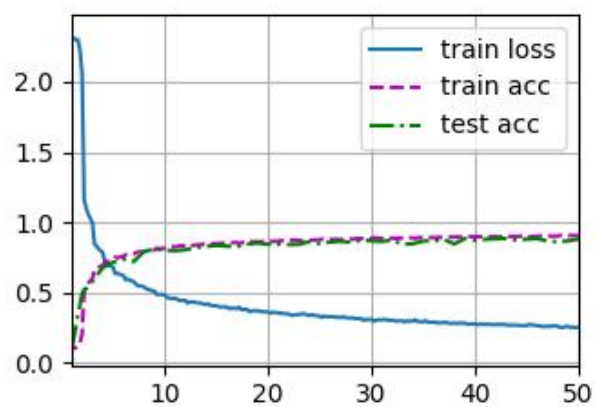
nn.AdaptiveAvgPool2d((1, 1)),

nn.Flatten(),

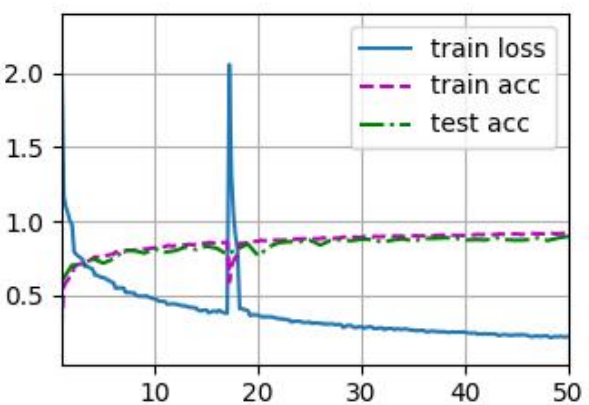
nn.Linear(1024, 10))

```

LeNet



损失	0.254
训练正确率	0.905
验证正确率	0.882
代码运行效率（例/每秒）	59383.5
代码运行时间（秒）	325.3325755596161

GoogLeNet	
损失	0.220
训练正确率	0.918
验证正确率	0.899
代码运行效率 (例/每秒)	5698.2
代码运行时间 (秒)	735.6776475906372
分析: GoogLeNet 的结果并没有显著优于 LeNet, 且更为耗时。本任务过于简单, 网络宽度上的增加并没有效果。	

4. ResNet (ResNet.py)

ResNet 是微软研究院的何恺明等人提出的卷积神经网络, 斩获 2015 年 ImageNet 竞赛中分类任务和目标检测第一名。残差引出了“退化现象 (Degradation)”, 针对此发明的“直连边/短连接 (Shortcut connection)”, 极大消除了深度过大的神经网络训练困难的问题。下图 6.4 给出了它的具体结构。

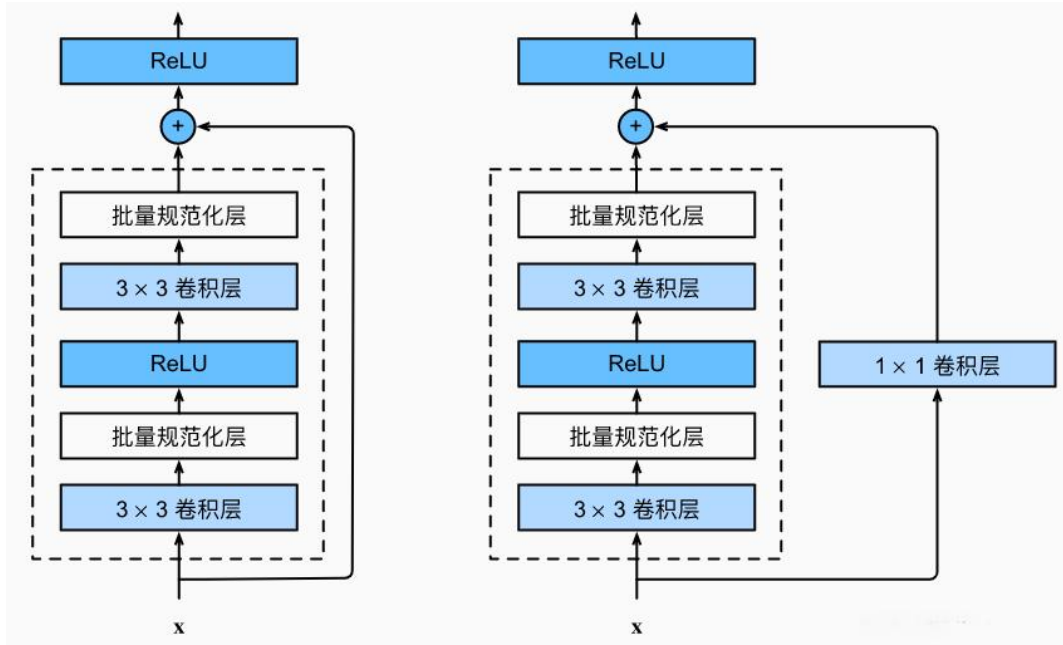


图 6.4 ResNet 结构

以下是网络部分的代码。

残差块

```
class Residual(nn.Module):
```

```
    def __init__(self, in_channels, out_channels, use_1x1conv=False, strides=1):
```

```
        super().__init__()
```

```
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1,
stride=strides)
```

```
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
```

```
        if use_1x1conv:
```

```
            self.conv3 = nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=strides)
```

```
        else:
```

```
            self.conv3 = None
```

```
        self.bn1 = nn.BatchNorm2d(out_channels)
```

```
        self.bn2 = nn.BatchNorm2d(out_channels)
```

```
    def forward(self, X):
```

```
        Y = F.relu(self.bn1(self.conv1(X)))
```

```
        Y = self.bn2(self.conv2(Y))
```

```
        if self.conv3:
```

```
            X = self.conv3(X)
```

```
        Y += X
```

```
        return F.relu(Y)
```

```
# 残差块组
```

```
def resnet_block(in_channels, out_channels, num_residuals, first_block=False):

    blk = []

    for i in range(num_residuals):

        if i == 0 and not first_block:

            blk.append(Residual(in_channels, out_channels, use_1x1conv=True,
strides=2))

        else:

            blk.append(Residual(out_channels, out_channels))

    return nn.Sequential(*blk)
```

```
# ResNet
```

```
class ResNet(nn.Module):

    def __init__(self, arch=((1, 64), (2, 128), (2, 256), (2, 512)), num_classes=10):

        super(ResNet, self).__init__()

        self.net = nn.Sequential(

            nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),

            nn.BatchNorm2d(64),

            nn.ReLU(),

            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        )

        self.resnet_blocks = nn.Sequential()
```

```

in_channels = 64

for i, (num_residuals, out_channels) in enumerate(arch):

    self.resnet_blocks.add_module(f"block{i+1}", resnet_block(in_channels,
out_channels, num_residuals, i == 0))

    in_channels = out_channels

self.net.add_module("resnet_blocks", self.resnet_blocks)

self.net.add_module("global_avg_pool", nn.AdaptiveAvgPool2d((1, 1)))

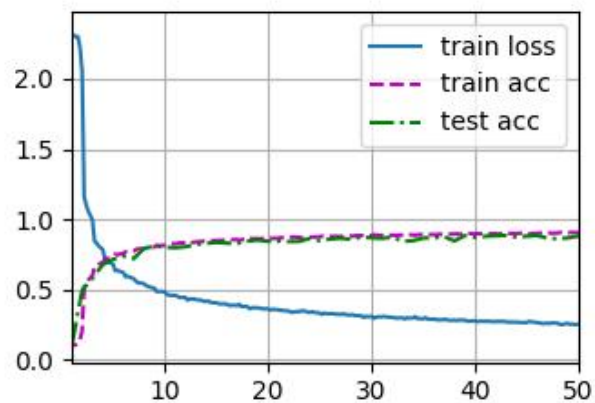
self.net.add_module("fc", nn.Sequential(nn.Flatten(), nn.Linear(out_channels,
num_classes)))

def forward(self, X):

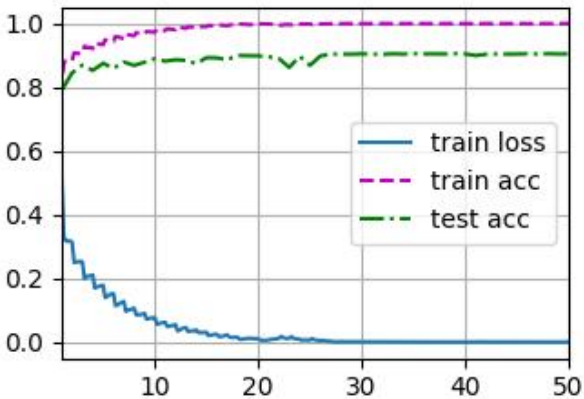
    return self.net(X)

```

LeNet



损失	0.254
训练正确率	0.905
验证正确率	0.882
代码运行效率（例/每秒）	59383.5
代码运行时间（秒）	325.3325755596161

ResNet	
损失	0.000
训练正确率	1.000
验证正确率	0.905
代码运行效率 (例/每秒)	15623.7
代码运行时间 (秒)	391.4592020511627
分析：可以发现，对于训练集来说，ResNet 可以很快达到 100% 正确率，然而对于验证集来说，并没有达到相应的准确率，这说明网络对于简单的小图像分类任务存在过拟合现象。ResNet 常用于深度大的神经网络，但是本任务不需要大深度的神经网络，所以出现了对于训练集的过拟合。	

5. 特征可视化 (feature.py)

参照提供的论文，本部分尝试了对卷积过程中的特征进行可视化。

基于 Lenet 代码添加了以下函数：

```
# 提取指定层输出

def get_output(net, layer_index, input):

    for i, layer in enumerate(net):

        input = layer(input)

        if i == layer_index:

            return input

    return input

# 特征可视化

def visualize_features(net, layer_index, input):
```

```

# 获取指定层输出

features = get_output(net, layer_index, input)

# 转换为 numpy 数组

features = features.detach().cpu().numpy()[0]

# 确定要显示的特征图数量

num_kernels = features.shape[0]

rows = int(num_kernels ** 0.5)

cols = (num_kernels + rows - 1) // rows

# 绘制特征图

fig, axes = plt.subplots(rows, cols, figsize=(10, 10))

for i in range(num_kernels):

    ax = axes[i // cols, i % cols]

    ax.imshow(features[i], cmap='gray')

    ax.axis('off')

plt.tight_layout()

plt.show()

```

在主函数部分添加相应的使用：

```

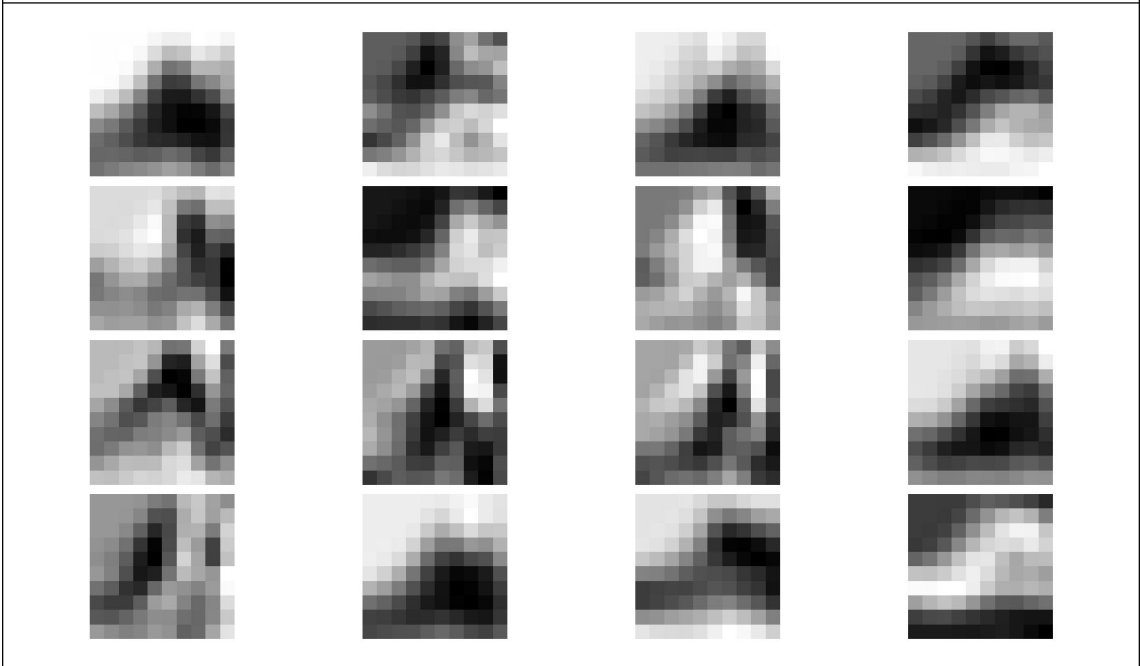
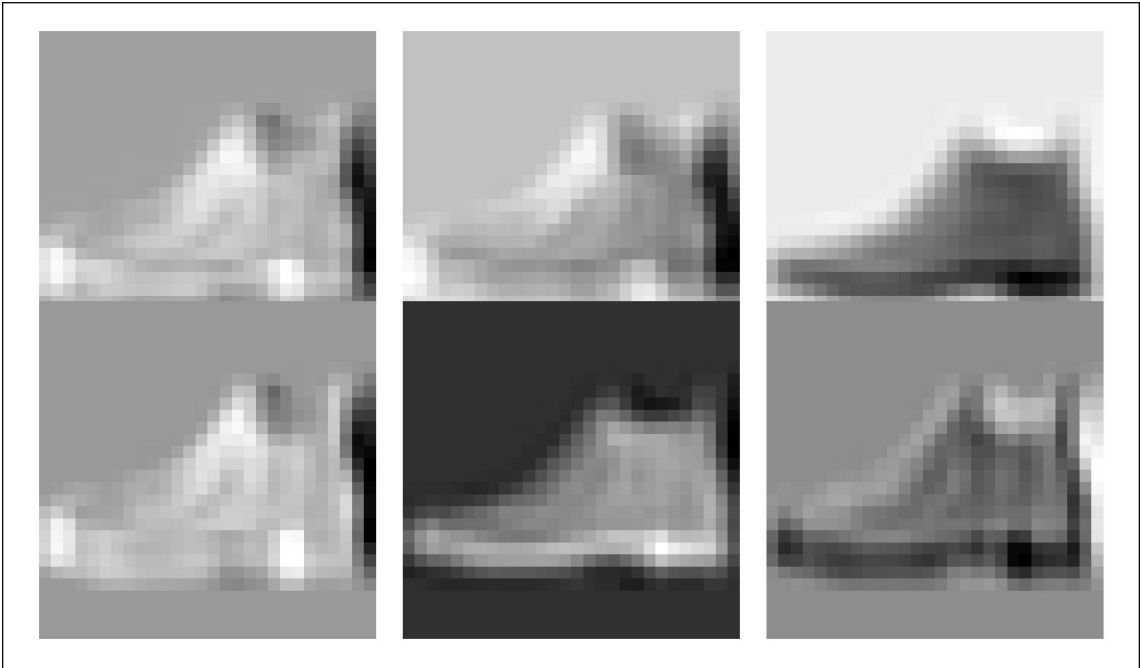
input_data = next(iter(test_iter))[0].to(d2l.try_gpu())

visualize_features(net, 0, input_data)

visualize_features(net, 3, input_data)

```

得到如下结果，依次是前后两次卷积的可视化结果：



分析：这里仅选取了鞋子的一个批次，但是也很好地展现了卷积过程中，网络中所传递的信息的变化。可以发现，第一次卷积的结果完整体现了鞋子的轮廓，但是第二次卷积的结果的可理解性就比较差了。