深度学习

1.CNN:卷积神经网络

输入层(inputs layer)：即构建模型

卷积层： 特征提取 卷积核/滤波器：权值函数，为了和所给图像的矩阵表示做卷积 步长:每次卷积移动的位数

池化层（pool)：特征提取

最大池化（max-pooling)：选取最大值

平均池化(mean-pooling)：选取平均值

随机池化(stochastic-polling)：

池化具有平移不变性：就是说图片特征不论在哪，都可以识别到

padding:

same padding（相同补充）:在原图四周补0，卷积窗口采样后得到一个跟原来大小一样的平面

valid padding（无补充）:不会超过平面外部，卷积窗口采样后得到一个比原来平面小的平面

loss函数：

在机器学习和深度学习中，损失函数（或成本函数）是用来衡量模型预测值与真实值之间差距的指标。选择合适的损失函数对于模型的训练和性能至关重要。以下是一些常见的损失函数，按任务类型分类：

1. 回归任务

- **均方误差（Mean Squared Error, MSE）**：

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

适合于回归问题，衡量预测值与真实值之间的平均平方差。

- **平均绝对误差（Mean Absolute Error, MAE）**：

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

计算预测值与真实值之间的平均绝对差，较不敏感于异常值。

2. 分类任务

- **二元交叉熵（Binary Cross-Entropy）**：

$$\text{BCE} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

适用于二分类问题，衡量模型预测概率与真实标签之间的差距。

- **多类交叉熵（Categorical Cross-Entropy）**：

$$\text{CCE} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

适用于多分类问题，其中 (C) 是类别总数，(y_i) 是真实分布，(\hat{y}_i) 是预测分布。

3. 特殊任务

- **对比损失（Contrastive Loss）**：
  用于度量学习，特别是 Siamese 网络中，旨在拉近相似样本的距离并拉远不相似样本的距离。
- **损失函数的正则化**：
  在损失函数中加入正则化项（如 L1 或 L2 正则化），用于防止过拟合：

$$\text{Loss} = \text{Original Loss} + \lambda \cdot \text{Regularization Term}$$

4. 选择损失函数的考虑因素

- **任务类型**：选择适合回归或分类的损失函数。
- **数据分布**：考虑数据是否存在异常值，选择对异常值敏感或不敏感的损失函数。
- **模型复杂性**：更复杂的模型可能需要更复杂的损失函数来捕捉数据的特征。

激活函数：

让多层感知机成为真正的多层 引入非线性，使网络可以逼近非线性函数。

常见激活函数：

sigmoid:输出范围在 (0, 1) 之间，常用于二分类问题的输出层。

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

tanh:输出范围在 (-1, 1) 之间，适合需要中心化的情况。

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

ReLU：输出范围在 [0, ∞) 之间，常用于隐藏层。

max(0,x)

激活函数的导数影响反向传播中的梯度计算。例如，像 ReLU 这样的激活函数在正区间内导数为常数，这有助于加速训练；而像 sigmoid 和 tanh 函数在极端值附近的梯度较小，可能导致梯度消失问题。

全连接层：

局部感受野：和权值共享减少了神经网络需要训练的参数个数

独热编码的缺点：稀疏，高数量，语意相关

LeNet 架构

LeNet 是一种经典的卷积神经网络（CNN）模型，最初由 Yann LeCun 等人在 1989 年提出。它主要用于手写数字识别，尤其是在 MNIST 数据集上表现出色。以下是 LeNet 模型的基本结构和组成部分。

LeNet 主要由以下几个层组成：

1. **输入层**：输入为 32x32 像素的灰度图像（例如 MNIST 数据集中的手写数字）。
2. **卷积层 1 (C1)**：
   - 过滤器数量：6
   - 过滤器大小：5x5

- 激活函数：Sigmoid 或 Tanh
- 输出大小：28x28x6（经过卷积后，边长减少）

3. **池化层 1 (S2)**：
   - 池化类型：平均池化
   - 池化窗口：2x2
   - 步长：2
   - 输出大小：14x14x6（尺寸减半）
4. **卷积层 2 (C3)**：
   - 过滤器数量：16
   - 过滤器大小：5x5
   - 输出大小：10x10x16
5. **池化层 2 (S4)**：
   - 池化类型：平均池化
   - 输出大小：5x5x16
6. **全连接层 1 (C5)**：
   - 输入：5x5x16 = 400
   - 输出神经元：120
7. **全连接层 2 (C6)**：
   - 输入：120
   - 输出神经元：84
8. **输出层 (Output)**：
   - 输出神经元：10（对应数字 0-9）
   - 激活函数：softmax（用于多分类）

LeNet 的特点

- **简单性**：LeNet 是 CNN 领域的开创性工作，结构相对简单，便于理解和实现。
- **应用广泛**：虽然最初用于手写数字识别，但其基本架构被广泛应用于其他计算机视觉任务中。
- **激活函数**：早期版本使用 Sigmoid 或 Tanh，但现代应用通常使用 ReLU 等激活函数。

实现示例（使用 PyTorch）

以下是使用 PyTorch 实现 LeNet 的示例代码：

```
import numpy as np
from torch import nn,optim
from torch.autograd import Variable
from torchvision import datasets,transforms
from torch.utils.data import DataLoader
# 训练集
train_dataset = datasets.MNIST(
    root='./',  # 指定数据集存储的根目录。如果数据集已经存在，将从这里加载；如果不存在，将下载数据。
    train=True,  # 设置为 True 表示加载训练集；如果为 False，则加载测试集。
    transform=transforms.ToTensor(),  # 将图像数据转换为 PyTorch 张量格式，以便于后续处理和模型训练。
```

```
    download=True  # 如果指定的 MNIST 数据集不存在，则下载数据集。
)

# 测试集
test_dataset = datasets.MNIST(
    root='./',  # 同样指定数据集存储的根目录。
    train=False,  # 设置为 False 表示加载测试集；如果为 True，则加载训练集。
    transform=transforms.ToTensor(),  # 将图像数据转换为 PyTorch 张量格式。
    download=True  # 如果指定的 MNIST 数据集不存在，则下载数据集。
)
#批次大小
batch_size=64

#训练加载器
train_loader=DataLoader(dataset=train_dataset,  # 使用之前定义的训练数据集
                        batch_size=batch_size,
                        shuffle=True)   #打乱数据集

#测试加载器
test_loader=DataLoader(dataset=test_dataset,   # 使用之前定义的测试数据集
                        batch_size=batch_size,
                        shuffle=True)
for i,data in enumerate(train_loader):   # 循环遍历训练数据加载器中的数据，i 是当前批次的索引，data 是当前批次的数据
    inputs,labels=data     #解包当前批次的数据，将输入图像和标签分别赋值给 inputs 和 labels
    print(inputs.shape)    #训练集的形状 64张图，一个通道，28*28的像素大小
    print(labels.shape)    # 打印标签的形状，期望为 (64,) 或 (64, 1)，表示每个样本对应的标签
    break
import torch
import torch.nn as nn
#定义网络结构
class Net(nn.Module):  # 定义一个名为 Net 的神经网络类，继承自 nn.Module
    def __init__(self):  # 构造函数，初始化网络结构
        super(Net, self).__init__()  # 调用父类 nn.Module 的构造函数，确保正确初始化

 self.conv1=nn.Sequential(nn.Conv2d(1,32,5,1,2),nn.ReLU(),nn.MaxPool2d(2,2))
 #Conv2d (in_channels,out_channels,kernel_size,stride,padding,dilation)

 self.conv2=nn.Sequential(nn.Conv2d(32,64,5,1,2),nn.ReLU(),nn.MaxPool2d(2,2))

 self.fc1=nn.Sequential(nn.Linear(64*7*7,1000),nn.Dropout(p=0.5),nn.ReLU())
        self.fc2=nn.Sequential(nn.Linear(1000,10),nn.Softmax(dim=1))
    def forward(self, x):  # 定义前向传播方法，接收输入 x
        # (64,1,28,28)
        x=self.conv1(x)
        x=self.conv2(x)
        #(64,64,7,7)
        x=x.view(x.size()[0],-1)
        x=self.fc1(x)
        x=self.fc2(x)
        return x
LR=0.001
#定义模型
model=Net()
```

```
#定义代价函数
mse_loss=nn.CrossEntropyLoss()
#定义优化器
optimizer=optim.Adam(model.parameters(),LR)
def train():
    model.train() #模型的训练状态
    for i,data in enumerate(train_loader):
        #获得一个批次的数据和标签
        inputs,labels=data
        #获得模型的数据和标签
        out=model(inputs)
        #交叉熵代价函数out(batch,C),labels(batch)
        loss=mse_loss(out,labels)
        #梯度清零
        optimizer.zero_grad()
        #计算梯度
        loss.backward()
        #修改权值
        optimizer.step()

def test():
    model.eval()  #模型的测试状态
    correct=0  #正确数量
    for i,data in enumerate(test_loader):
        inputs,labels=data
        out=model(inputs)
        #获得最大值，以及最大值所在的位置
        _,predicted=torch.max(out,1)  #这里会舍弃其最大值，然后要其最大值所在位置
        #预测正确的数量
        correct+=(predicted==labels).sum()

    print("Test acc:{0}".format(correct.item()/len(test_dataset))) #这里用item是用
于将correct张量转换为python标量


    correct=0  #正确数量
    for i,data in enumerate(train_loader):
        inputs,labels=data
        out=model(inputs)
        #获得最大值，以及最大值所在的位置
        _,predicted=torch.max(out,1)
        #预测正确的数量
        correct+=(predicted==labels).sum()

    print("Train acc:{0}".format(correct.item()/len(train_dataset)))
for epoch in range(20):
    print('epoch:',epoch)
    train()
    test()


#保存模型
torch.save(model,'m.pth')
#参数保存
torch.save(modle.state_dict(),'m.pth')
```

```
model.load_state_dict(torch.load(model.pth))
```

总结

LeNet 是卷积神经网络的基础模型，具有重要的历史意义。虽然现代深度学习模型更加复杂，但 LeNet 为后续的 CNN 发展奠定了基础。

论文出处："C:\Users\19338\Desktop"

AlexNet 架构

AlexNet 是一种重要的卷积神经网络（CNN）架构，由 Alex Krizhevsky 等人在 2012 年提出，标志着深度学习在计算机视觉领域的突破。它在 ImageNet 图像分类挑战赛中取得了显著成功，推动了深度学习的广泛应用。

AlexNet 的主要特点和组成部分包括：

1. **输入层**：
   - 输入图像大小为 224x224x3（RGB 图像）。
2. **卷积层**：
   - **Conv1**：使用 96 个 11x11 的卷积核，步幅为 4，输出大小为 54x54x96。
   - **Conv2**：使用 256 个 5x5 的卷积核，步幅为 1，输出大小为 27x27x256。
   - **Conv3**：使用 384 个 3x3 的卷积核，步幅为 1，输出大小为 13x13x384。
   - **Conv4**：使用 384 个 3x3 的卷积核，步幅为 1，输出大小为 13x13x384。
   - **Conv5**：使用 256 个 3x3 的卷积核，步幅为 1，输出大小为 13x13x256。
3. **池化层**：
   - 在 Conv1 和 Conv2 之间使用最大池化（max pooling），池化窗口为 3x3，步幅为 2。
   - 在 Conv3、Conv4 和 Conv5 之间没有池化层，但 Conv5 之后有一个最大池化层，池化窗口为 3x3，步幅为 2。
4. **全连接层**：
   - **FC1**：输入为 6, 000 个神经元，输出为 4096 个神经元。
   - **FC2**：输入为 4096 个神经元，输出为 4096 个神经元。
   - **FC3**：输入为 4096 个神经元，输出为 1000 个神经元（对应 1000 个类别）。
5. **激活函数**：
   - 使用 ReLU（Rectified Linear Unit）作为激活函数，代替传统的 Sigmoid 或 Tanh 激活函数，这加速了训练过程。
6. **Dropout**：
   - 在全连接层中使用 Dropout 技术，以减少过拟合。
7. **数据增强**：
   - 在训练过程中应用数据增强技术（如随机裁剪、翻转等），以提高模型的泛化能力。

总结

AlexNet 是一种深度卷积神经网络，其架构在当时是非常创新的，通过使用更深的网络结构、ReLU 激活函数和 Dropout 等技术，显著提高了图像分类的准确率。它为后续的深度学习模型（如 VGG、GoogLeNet、ResNet 等）的发展奠定了基础。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchsummary import summary
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from torch import optim

# 定义 AlexNet 模型
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.relu = nn.ReLU(inplace=True)
        self.c1 = nn.Conv2d(1, 96, kernel_size=11, stride=4)
        self.p1 = nn.MaxPool2d(kernel_size=3, stride=2)
        self.c2 = nn.Conv2d(96, 256, kernel_size=5, padding=2)
        self.p2 = nn.MaxPool2d(kernel_size=3, stride=2)
        self.c3 = nn.Conv2d(256, 384, kernel_size=3, padding=1)
        self.c4 = nn.Conv2d(384, 384, kernel_size=3, padding=1)
        self.c5 = nn.Conv2d(384, 256, kernel_size=3, padding=1)
        self.p3 = nn.MaxPool2d(kernel_size=3, stride=2)

        self.flatten = nn.Flatten()
        self.f1 = nn.Linear(256 * 6 * 6, 4096)
        self.f2 = nn.Linear(4096, 4096)
        self.f3 = nn.Linear(4096, 10)

    def forward(self, x):
        x = self.relu(self.c1(x))
        x = self.p1(x)
        x = self.relu(self.c2(x))
        x = self.p2(x)
        x = self.relu(self.c3(x))
        x = self.relu(self.c4(x))
        x = self.relu(self.c5(x))
        x = self.p3(x)

        x = self.flatten(x)
        x = self.relu(self.f1(x))
        x = F.dropout(x, p=0.5)
        x = self.relu(self.f2(x))
        x = F.dropout(x, p=0.5)
        x = self.f3(x)
        return x

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
transform = transforms.Compose([
    transforms.Resize((227, 227)),
    transforms.ToTensor(),
])

# 加载数据集
train_dataset = datasets.MNIST(
    root='./data',
```

```
    train=True,
    transform=transform,
    download=True
)

test_dataset = datasets.MNIST(
    root='./data',
    train=False,
    transform=transform,
    download=True
)

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)  # 测试集不打乱

# 初始化模型并移至设备
model = AlexNet().to(device)
summary(model, (1, 227, 227))

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 训练函数
def train():
    model.train()
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)  # 移至相同设备

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

# 测试函数 - 修正：移除未使用的dataset参数
def test(loader, name):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    accuracy = 100 * correct / total
    print(f"{name} Accuracy: {accuracy:.2f}%")
    return accuracy

# 训练循环
for epoch in range(10):
```

```
    print(f'Epoch {epoch+1}/10')
    train()
    test(train_loader, "Train")
    test_acc = test(test_loader, "Test")
    print('-' * 50)

print("Training completed!")
```

VGG 架构

VGG 是一种深度卷积神经网络架构，由牛津大学的 Visual Geometry Group（VGG）在 2014 年提出。VGG 以其简单而高效的设计以及出色的性能而闻名，特别是在图像分类任务中。以下是对 VGG 框架的详细介绍。

1. **基本结构**：
    - VGG 的核心思想是使用连续的小卷积核（3x3）来代替大卷积核，以增加网络的深度。
    - 通常，VGG 模型包含多个卷积层，后接最大池化层，然后是全连接层。
2. **网络层次**：
    - VGG 的不同版本（如 VGG16、VGG19）表示网络的深度，数字表示网络中卷积层的总数。
        - **VGG16**：包含 16 层可训练的权重（13 个卷积层和 3 个全连接层）。
        - **VGG19**：包含 19 层可训练的权重（16 个卷积层和 3 个全连接层）。
3. **卷积层**：
    - 每个卷积层后面通常跟着一个 ReLU 激活函数。
    - 使用多个连续的 3x3 卷积核有助于捕捉更复杂的特征，同时保持感受野的大小。
4. **池化层**：
    - 在卷积层之后，使用 2x2 的最大池化层，步幅为 2，以减少特征图的尺寸。
5. **全连接层**：
    - 在卷积和池化层之后，VGG 通常包含 3 个全连接层，最后一层的输出节点数等于类别数（例如，对于 ImageNet，输出为 1000）。

VGG 结构示例

以下是 VGG16 的具体结构示例：

- **输入层**：图像大小为 224x224x3（RGB）。
- **卷积层**：
    - Conv3-64 (3x3, 64 filters)
    - Conv3-64 (3x3, 64 filters)
    - MaxPool (2x2)
    - Conv3-128 (3x3, 128 filters)
    - Conv3-128 (3x3, 128 filters)
    - MaxPool (2x2)
    - Conv3-256 (3x3, 256 filters)
    - Conv3-256 (3x3, 256 filters)

    - Conv3-256 (3x3, 256 filters)
    - MaxPool (2x2)
    - Conv3-512 (3x3, 512 filters)
    - Conv3-512 (3x3, 512 filters)
    - Conv3-512 (3x3, 512 filters)
    - MaxPool (2x2)
    - Conv3-512 (3x3, 512 filters)
    - Conv3-512 (3x3, 512 filters)
    - Conv3-512 (3x3, 512 filters)
    - MaxPool (2x2)
- **全连接层**:
    - FC (4096)
    - FC (4096)
    - FC (1000) # 输出层，对应 1000 个类别

VGG 的优缺点

优点

- **简单性**：结构简单，易于理解和实现。
- **良好的性能**：在多个图像分类任务中取得了优异的性能，尤其是在 ImageNet 挑战赛中。
- **可迁移性**：VGG 模型可以作为预训练模型，迁移到其他计算机视觉任务中。

缺点

- **计算资源**：VGG 模型比较深，计算量大，需要更多的计算资源和内存。
- **模型大小**：由于全连接层的数量，模型参数较多，导致模型体积庞大。

下面是一个使用 VGG11 架构实现 Fashion-MNIST 手写服装识别的完整 PyTorch 示例代码。我们将使用 PyTorch 的 `torchvision` 库来构建 VGG11 模型，并适配 Fashion-MNIST 数据集。

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# 定义 VGG11 模型
class VGG11(nn.Module):
    def __init__(self):
        super(VGG11, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, padding=1),  # 输入通道为 1 (灰度图像)
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
```

```
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 10),  # 输出层，10 个类别
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # 展平
        x = self.classifier(x)
        return x

# 数据预处理
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # 调整图像大小以适应 VGG 输入
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])  # 标准化
])

# 加载 Fashion-MNIST 数据集
train_dataset = datasets.FashionMNIST(root='./data', train=True, download=True,
transform=transform)
test_dataset = datasets.FashionMNIST(root='./data', train=False, download=True,
transform=transform)
```

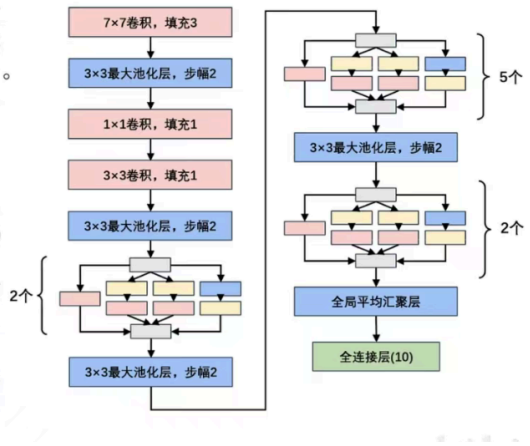
```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 初始化模型、损失函数和优化器
model = VGG11().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 训练模型
def train(model, train_loader, criterion, optimizer, epochs=5):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
        print(f'Epoch [{epoch+1}/{epochs}], Loss:
{running_loss/len(train_loader):.4f}')

# 测试模型
def test(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    print(f'Accuracy of the model on the test images: {accuracy:.2f}%')

# 执行训练和测试
train(model, train_loader, criterion, optimizer, epochs=5)
test(model, test_loader)
```

总结

VGG 是一种经典的深度学习架构，通过使用小卷积核和深层网络结构，成功地在图像分类和其他计算机视觉任务中取得了显著成果。尽管计算开销较大，但其简单而有效的设计理念对后续的深度学习模型（如 ResNet）产生了深远影响。

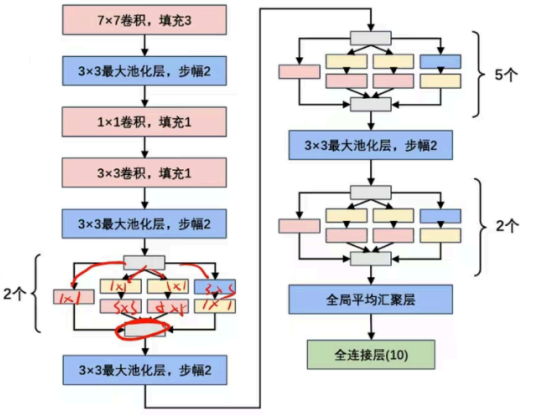
GoogLeNet架构

第1层输入层： 输入为224×224×3 三通道的图像。
第2块：
（1）卷积层；输入为224×224×3，卷积核数量为64个；卷积核的尺寸大小为7×7×3；步幅为2（stride = 2），填充为3（padding=3）；卷积后得到shape为112×112×64的特征图输出。
（2）最大池化层；输入为112×112×64，池化核为3×3，步幅为2（stride = 2），填充为1（padding=1）；后得到尺寸为56×56×64的池化层的特征图输出。
第3块：
（1）卷积层；输入为56×56×64，卷积核数量为64个；卷积核的尺寸大小为1×1×64；步幅为1（stride =1），填充为0（padding=0）；卷积后得到shape为56×56×64的特征图输出。
（2）卷积层；输入为56×56×64，卷积核数量为192个；卷积核的尺寸大小为3×3×64；步幅为1（stride =1），填充为1（padding=1）；卷积后得到shape为56×56×192的特征图输出。
（3）最大池化层；输入为56×56×192，池化核为3×3，步幅为2（stride = 2），填充为1（padding=1）；后得到尺寸为28×28×192的池化层的特征图输出。



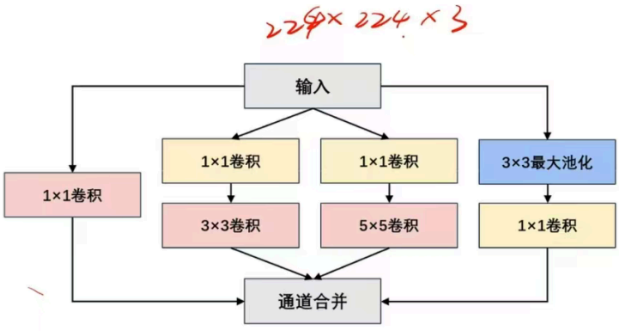
接下来：

第1个Inception块：
输入为28×28×192的特征图。
路径1：
（1）输入为28×28×192，卷积核数量为64个；卷积核的尺寸大小为1×1×192；步幅为1（stride = 1），填充为0（padding=0）；卷积后得到shape为28×28×64的特征图输出。
路径2：
（1）输入为28×28×192，卷积核数量为96个；卷积核的尺寸大小为1×1×192；步幅为1（stride = 1），填充为0（padding=0）；卷积后得到shape为28×28×96的特征图输出。
（2）输入为28×28×96，卷积核数量为128个；卷积核的尺寸大小为3×3×96；步幅为1（stride = 1），填充为1（padding=1）；卷积后得到shape为28×28×128的特征图输出。
路径3：
（1）输入为28×28×192，卷积核数量为16个；卷积核的尺寸大小为1×1×192；步幅为1（stride = 1），填充为0（padding=0）；卷积后得到shape为28×28×16的特征图输出。
（2）输入为28×28×16，卷积核数量为32个；卷积核的尺寸大小为5×5×16；步幅为1（stride = 1），填充为2（padding=2）；卷积后得到shape为28×28×32的特征图输出。
路径4：
（1）输入为28×28×192，池化核的尺寸大小为3×3；步幅为1（stride = 1），填充为1（padding=1）；池化后得到shape为28×28×192的特征图输出。
（2）输入为28×28×192，卷积核数量为32个；卷积核的尺寸大小为1×1×3；步幅为1（stride = 1），填充为0（padding=0）；卷积后得到shape为28×28×32的特征图输出。



具体一个inception:

输入为224×224×3三通道的图像。
路径1：
（1）输入为224×224×3，卷积核数量为64个；卷积核的尺寸大小为1×1×3；步幅为1（stride=1），填充为0（padding=0）；卷积后得到shape为224×224×64的特征图输出。
路径2：
（1）输入为224×224×3，卷积核数量为96个；卷积核的尺寸大小为1×1×3；步幅为1（stride = 1），填充为0（padding=0）；卷积后得到shape为224×224×64的特征图输出。
（2）输入为224×224×64，卷积核数量为128个；卷积核的尺寸大小为3×3×64；步幅为1（stride = 1），填充为1（padding=1）；卷积后得到shape为224×224×128的特征图输出。



路径3：

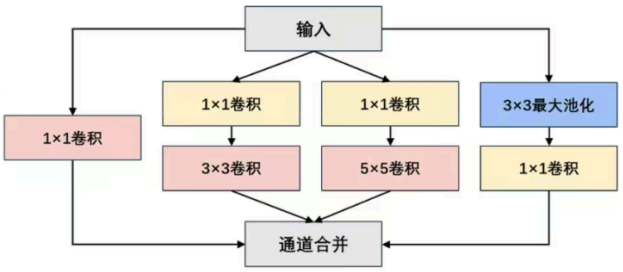
（1）输入为224×224×3，卷积核数量为16个；卷积核的尺寸大小为1×1×3；步幅为1（stride = 1），填充为0（padding=0）；卷积后得到shape为224×224×16的特征图输出。

（2）输入为224×224×16，卷积核数量为32个；卷积核的尺寸大小为5×5×16；步幅为1（stride = 1），填充为2（padding=2）；卷积后得到shape为224×224×32的特征图输出。

路径4：

（1）输入为224×224×3，池化核的尺寸大小为3×3；步幅为1（stride = 1），填充为1（padding=1）；池化后得到shape为224×224×3的特征图输出。

（2）输入为224×224×3，卷积核数量为32个；卷积核的尺寸大小为1×1×3；步幅为1（stride = 1），填充为0（padding=0）；卷积后得到shape为224×224×32的特征图输出。



通道合并：

路径1的到输出为：224×224×64

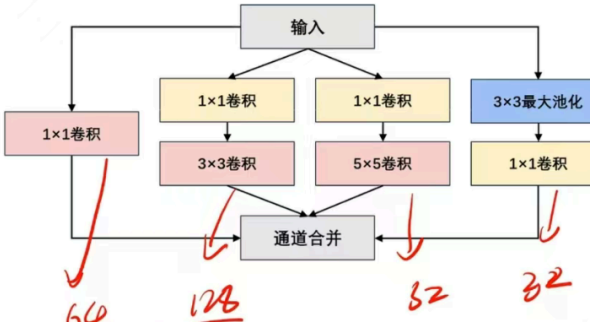
路径2的到输出为：224×224×128

路径3的到输出为：224×224×32

路径4的到输出为：224×224×32

最终通道合并为64+128+32+32=256，最终的输出为：224×224×256。

那么为什么GoogLeNet这个网络如此有效呢？首先我们考虑一下滤波器（filter）的组合，它们可以用各种滤波器尺寸探索图像，这意味着不同大小的滤波器可以有效地识别不同范围的图像细节。同时，我们可以为不同的滤波器分配不同数量的参数。




```
import torch
from torch import nn
from torchsummary import summary

class Inception(nn.Module):
    def __init__(self, in_channels, c1, c2, c3, c4):
        super(Inception, self).__init__()
        self.relu = nn.ReLU()

        # Branch 1: 1x1 convolution
        self.p1_1 = nn.Conv2d(in_channels, c1, kernel_size=1)

        # Branch 2: 1x1 conv -> 3x3 conv
        self.p2_1 = nn.Conv2d(in_channels, c2[0], kernel_size=1)
        self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)  # Fixed
padding

        # Branch 3: 1x1 conv -> 5x5 conv
        self.p3_1 = nn.Conv2d(in_channels, c3[0], kernel_size=1)
        self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)  # Fixed
padding (2 for 5x5)

        # Branch 4: MaxPool -> 1x1 conv
        self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.p4_2 = nn.Conv2d(in_channels, c4, kernel_size=1)  # Correct input
channels

    def forward(self, x):
        p1 = self.relu(self.p1_1(x))
        p2 = self.relu(self.p2_2(self.relu(self.p2_1(x))))
```

```
        p3 = self.relu(self.p3_2(self.relu(self.p3_1(x))))
        p4 = self.relu(self.p4_2(self.p4_1(x)))  # Removed extra ReLU
        return torch.cat((p1, p2, p3, p4), dim=1)  # Concatenate along channels

class GoogleNet(nn.Module):
    def __init__(self):
        super(GoogleNet, self).__init__()
        self.b1 = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.b2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=1),
            nn.ReLU(),
            nn.Conv2d(64, 192, kernel_size=3, padding=1),  # Fixed padding (1 for
3x3)
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.b3 = nn.Sequential(
            Inception(192, 64, (96, 128), (16, 32), 32),  # c3: (16,32)
            Inception(256, 128, (128, 192), (32, 96), 64), # Fixed c3: (32,96)
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.b4 = nn.Sequential(
            Inception(480, 192, (96, 208), (16, 48), 64),  # Fixed channels
            Inception(512, 160, (112, 224), (24, 64), 64),
            Inception(512, 128, (128, 256), (24, 64), 64),
            Inception(512, 112, (128, 288), (32, 64), 64),
            Inception(528, 256, (160, 320), (32, 128), 128),  # Fixed c1: 256
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.b5 = nn.Sequential(
            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)  # 1024 channels from last Inception
        )

        # Initialize weights
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear) and m.bias is not None:
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        x = self.b1(x)
        x = self.b2(x)
        x = self.b3(x)
        x = self.b4(x)
```

```
        x = self.b5(x)
        return x

if __name__ == '__main__':
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model = GoogleNet().to(device)
    print(summary(model, (1, 224, 224)))
```

NIN 架构

Network in Network (NIN) 是一种卷积神经网络架构，旨在通过引入更深的网络结构和更复杂的特征提取方式来提高模型的表现。NIN 由 Lin 等人在 2013 年提出，主要通过将传统的卷积层替换为更小的多层感知器（MLP）来实现。

1. **基本思想**：
    - NIN 用小的卷积核（通常是 1x1 的卷积）代替传统的卷积层，以实现更复杂的特征提取。1x1 卷积可以看作是对每个像素的特征进行线性组合。
    - 通过将多个 1x1 卷积层堆叠，NIN 能够在保持高效性的同时增加模型的非线性表达能力。
2. **结构组成**：
    - NIN 由多个卷积层组成，通常采用小卷积核（如 3x3 和 1x1），并结合全局平均池化和全连接层。
    - 网络结构通常包括如下几个部分：
        - **卷积层**：使用多个 3x3 卷积层提取特征。
        - **MLP 卷积层**：使用 1x1 卷积来进行特征的组合和映射。
        - **池化层**：使用最大池化或平均池化来降低特征图的尺寸。
        - **全局平均池化**：在最后的特征图上应用全局平均池化，减少参数数量。
3. **模型结构示例**：
   NIN 的具体结构如下：
    - **输入层**：通常为 RGB 图像。
    - **卷积层**：多个 3x3 卷积层。
    - **1x1 卷积层（MLP 卷积）**：用于特征组合。
    - **池化层**：如最大池化层。
    - **全局平均池化**：在最后的特征图上应用，输出为每个类别的概率。

NIN 的优缺点

优点

- **更深的网络**：NIN 通过引入更深的结构和复杂的特征提取方式，能够更好地捕捉数据中的复杂模式。
- **减少参数量**：使用 1x1 卷积可以显著减少参数数量，从而降低过拟合的风险。
- **高效性**：NIN 在计算上比传统的全连接层更高效。

缺点

- **计算资源**：尽管使用了 1x1 卷积，但整个网络仍然较深，对计算资源的需求较高。

- **模型复杂性**：网络结构的复杂性可能使得调参和训练更加困难。

总结

NIN 是一种创新的卷积神经网络架构，通过引入 1x1 卷积来增强特征提取能力，显著提高了图像分类等任务的性能。其设计理念对后续的深度学习模型（如 GoogLeNet 和 ResNet）产生了影响，推动了深度学习在计算机视觉领域的发展。

残差网络（ResNet)

让我们先思考一个问题：对神经网络模型添加新的层，充分训练后的模型是否只可能更有效地降低训练误差？理论上，原模型解的空间只是新模型解的空间的子空间。也就是说，如果我们能将新添加的层训练成恒等映射f(x)=x，新模型和原模型将同样有效。由于新模型可能得出更优的解来拟合训练数据集，因此添加层似乎更容易降低训练误差。然而在实践中，添加过多的层后训练误差往往不降反升。即使利用批量归一化带来的数值稳定性使训练深层模型更加容易，该问题仍然存在。针对这一问题，何恺明等人提出了残差网络（ResNet)。

稠密连接网络（DenseNet)

图5.10中将部分前后相邻的运算抽象为模块A*A*和模块B*B*。与ResNet的主要区别在于，DenseNet里模块B*B*的输出不是像ResNet那样和模块A*A*的输出相加，而是在通道维上连结。这样模块A*A*的输出可以直接传入模块B*B*后面的层。在这个设计里，模块A*A*直接跟模块B*B*后面的所有层连接在了一起。这也是它被称为“稠密连接”的原因。

2.RNN:循环神经网络

适用与序列模型，比如语音，歌曲，文字等,适用于小数据集，低算力的问题有效。

类型：

one to one:固定得到输入到输出，如图像分类

one to many:固定的输入到序列输出，如图像的文字描述

many to one:序列输入到输出，如情感分析，分类正面负面情绪

many to many:序列输入到序列的输出，如机器翻译，称之为编解码网络

同步多对多：同步序列输入到同步输出，如文本生成，视频每一帧的分类，也称之为序列生成。

输入：词的表示：用one-hot编码（就是会建立所有词的词库）

输出：用softmax激活函数表示概率（输出词的概率）

缺点：有限记忆

时间序列表示方法：sequence representation:[个数，维度]

LSTM: Long short term memory（长短时记忆函数）

增加记忆细胞，可以传递前部远处部位信息

BRNN:双向循环神经网络