应用实例

中华文明具有突出的连续性，这种一以贯之的延续和发展成就了中华文明独一无二的特色和厚重深邃的内涵。中医药是中国古老而独特的医药体系，历经几千年的实践和发展，形成了独特的理论体系、诊断方法和治疗手段，是中华传统优秀文化的一个重要组成部分，为中华民族的繁衍生息做出了巨大贡献。它不仅被用于治疗各种疾病，还被应用于保健、养生等方面，其疗效和独特性受到了许多人的认可和追捧，在中国乃至世界范围内都有着广泛的应用和影响。

习近平总书记强调“中医药学包含着中华民族几千年的健康养生理念及其实践经验，是中华文明的一个瑰宝，凝聚着中国人民和中华民族的博大智慧。”包括《“十四五”中医药发展规划》在内的多项推动中医药发展的相关文件和政策陆续出台，充分彰显了党的十八大以来，以习近平同志为核心的党中央把中医药工作摆在更加重要的位置。党的二十大报告中也明确提出要“促进中医药传承创新发展。”

中草药是中医药学的核心组成部分。中草药以其天然、综合的药理作用，广泛应用于治疗各种疾病。然而，中药材种类纷繁复杂，传统中草药的识别通常需要依赖专业的中医药师，他们凭借多年的经验和知识来判断草药的种类和功效，普通人对中药材的辨识知识比较匮乏，可能导致误用等不良后果。

本实践案例采用卷积神经网络，实现了快速准确地对中草药进行自动识别，大大提高了识别的效率。而且自动识别技术基于大量的数据和模型训练，提供客观准确的识别结果，减少了主观误差的影响，提高了识别的准确性。

本案例的数据集来源于互联网，包括百合（baihe）、白芷（baizhi）、枸杞（gouqi）、天麻（tianma）和薏仁（yiren）五类，每类包含 200 张图片左右，分别存储在 Chinese_Medicine 文件夹中与其名称对应的子文件夹内，如图 8-18 所示。

图8-1　图片数据集

首先，我们加载图像数据集，并对图片进行预处理，代码如下。

代码8-1　加载数据集并进行预处理

```
import torch
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder

# 设置随机种子
torch.manual_seed(42)

# 检查 GPU 是否可用
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

data_dir = "Chinese_Medicine"  # Chinese_Medicine 文件夹路径
image_size = (224, 224)  # 图像大小
batch_size = 32  # 批大小

# 定义数据预处理的转换操作
transform = transforms.Compose([
    transforms.Resize(image_size),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),  # 标准化操作
])

# 加载图像数据集
dataset = ImageFolder(data_dir, transform=transform)

# 获取数据集的类别数量
num_classes = len(dataset.classes)
```

```
# 打印数据集的大小和类别数量
print("数据集大小:", len(dataset))
print("类别数量:", num_classes)
print("类别:", dataset.classes)
```

以上代码中，torchvision 是 PyTorch 中提供的一个用于计算机视觉任务的软件包。它为图像处理、数据加载和预处理、模型架构和训练等方面提供了一系列工具和功能。

device = torch.device("cuda" if torch.cuda.is_available() else "cpu") 根据 torch.cuda.is_available()的结果选择要使用的设备。如果 CUDA 设备可用，则选择使用 GPU 设备（"cuda"），否则选择使用 CPU 设备（"cpu"）。通过这样的方式，我们可以根据系统环境动态选择 GPU 或 CPU 作为训练模型的设备，并相应地进行模型的移动、计算和优化操作。这样可以确保代码的通用性，并在有 GPU 时充分利用 GPU 的计算能力，而在没有 GPU 时仍然能够在 CPU 上运行代码。深度学习模型通常具有复杂的结构和大量的参数，这些模型对计算资源的要求很高，使用 GPU 可以更好地满足其计算需求，使得训练过程更稳定、更快速。

torchvision.transforms 模块中的函数 Compose 用于将多个图像预处理操作组合成一个串联的操作，将 Resize、ToTensor 和 Normalize 操作按顺序组合在一起，形成一个预处理的流水线。输入的图像首先被缩放为统一的 224×224 像素大小，因为卷积神经网络通常要求输入图像的尺寸是固定的。选择 224×224 大小作为常用的图像尺寸是因为它在很多经典的卷积神经网络中被广泛使用，如 AlexNet、VGGNet 和 ResNet 等。ToTensor 将图像数据从 PIL 图像对象转换为 PyTorch 张量。Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) 则是对图像进行标准化处理，以便使图像的像素值在一定范围内。在这里，我们使用了预定义的均值和标准差值，这些值是在大规模图像数据集上进行计算得到的。标准化可以使模型更容易学习和收敛，并提高模型的性能。

数据标注

接下来，使用 ImageFolder 类来加载图像数据集。我们将数据集路径和预处理的转换操作传递给 ImageFolder 类来创建数据集对象。ImageFolder 是一个可以从文件夹结构中自动推断类别标签的数据集类。此案例中是将与图片对应的子文件夹名作为分类标签，相当于自动实现了数据标注。所谓数据标注是指为机器学习和深度学习任务准备训练数据时，为数据样本提供准确的标签或注释的过程。

代码 8-1 的输出结果如图 8-19 所示。

```
数据集大小: 1047
类别数量: 5
类别: ['baihe', 'baizhi', 'gouqi', 'tianma', 'yiren']
```

图8-2　加载数据集信息

接下来，拆分训练集和测试集，并创建数据加载器，代码如下。

代码8-2　拆分数据集并创建数据加载器

```
#计算测试集的样本数量
num_test_samples = int(len(dataset) * 0.3)

#计算训练集的样本数量
num_train_samples = len(dataset) - num_test_samples

#使用随机分割函数将数据集拆分为训练集和测试集
train_dataset, test_dataset = torch.utils.data.random_split(
   dataset, [num_train_samples, num_test_samples])

#创建数据加载器
train_loader = torch.utils.data.DataLoader(
   train_dataset,
   batch_size=batch_size,
   shuffle=True)

test_loader = torch.utils.data.DataLoader(
   test_dataset,
   batch_size=batch_size,
   shuffle=False)
```

在以上代码中，使用 torch.utils.data.random_split 函数将原始数据集 dataset 按照指定的划分比例随机分割成训练集和测试集，将数据集的 70%作为训练集，余下的 30%作为测试集。

torch.utils.data.DataLoader 用于将数据集封装成可迭代的数据加载器，方便对数据进行批处理和迭代。其中，第一个参数为数据集对象，第二个参数为批大小 batch_size，指定每次迭代的样本数量，第三个参数 shuffle 表示是否在每个 epoch 中对数据进行洗牌，这里训练集设置为 shuffle=True，测试集设置为 shuffle=False。

处理好数据后，我们定义卷积神经网络模型，代码如下。

代码8-3　定义卷积神经网络模型

```
import torch.nn as nn

# 定义卷积神经网络模型
class ConvNet_1(nn.Module):
   def __init__(self, num_classes):
      super(ConvNet_1, self).__init__()
      self.model = nn.Sequential(
```

```
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.Linear(32 * 56 * 56, 256),
            nn.ReLU(),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        return self.model(x)

# 实例化卷积神经网络模型
model = ConvNet_1(num_classes)

# 模型移动到指定的设备GPU/CPU
model.to(device)
```

在以上代码中，首先定义了一个继承自 nn.Module 的卷积神经网络模型 ConvNet。在__init__方法中，调用父类的初始化方法，确保模型正确地继承了 nn.Module 的属性和方法。

在模型的 self.model 中，使用 nn.Sequential 按顺序定义了一系列网络层。其中，包括了卷积层、激活函数（ReLU）、最大池化层（MaxPool2d）、Flatten 层（将多维的输入展平为一维）、全连接层（Linear）等。这些层的组合形成了卷积神经网络的结构。nn.Conv2d 是 PyTorch 中的一个卷积层的类，用于定义二维卷积操作，nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)中的每个参数的含义如下：

- 3：输入通道数（input_channels），表示输入数据的通道数。在这里，输入数据是 RGB 彩色图像，所以有 3 个通道。
- 16：输出通道数（output_channels），表示卷积操作后输出的通道数。这个数字可以理解为卷积核的数量，也是卷积层输出的特征图数量。在卷积神经网络中，每一个卷积层的输入和输出通道数都需要经过计算得到。
- kernel_size=3：卷积核的尺寸，表示卷积核的宽度和高度。在这里，卷积核的尺寸是 3x3。
- stride=1：卷积操作的步长，默认为 1。
- padding=1：输入数据的边界填充数，用于控制卷积操作的边界处理。这里的 1 表示在输入数据的周围添加一圈宽度为 1 的填充像素。

在 forward 方法中，定义了模型的前向传播过程，即输入数据通过各层的计算得到输出。在这里，通过调用 self.model(x)实现了模型的前向计算。

各层参数计算

接下来，通过 model = ConvNet(num_classes)实例化了一个卷积神经网络模型对象，传入 num_classes 作为输出类别的数量，用于模型的最后一层全连接层的输出维度。

最后，通过 model.to(device)将模型移动到指定的设备。代码 8-3 的输出结果如图 8-20 所示。

```
ConvNet_1(
  (model): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Flatten(start_dim=1, end_dim=-1)
    (7): Linear(in_features=100352, out_features=256, bias=True)
    (8): ReLU()
    (9): Linear(in_features=256, out_features=5, bias=True)
  )
)
```

图8-3　卷积神经网络模型结构

定义好网络后，设置损失函数、优化器等超参数，开始训练模型并输出训练结果，代码如下。

代码8-4　训练模型

```
import torch.optim as optim

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

# 训练模型
num_epochs = 20
train_losses = []
train_accuracies = []
test_accuracies = []

for epoch in range(num_epochs):
    running_loss = 0.0
    correct_predictions = 0
    total_predictions = 0
    for images, labels in train_loader:
        # 将数据移动到设备上
        images = images.to(device)
        labels = labels.to(device)

        # 前向传播
```

```
        outputs = model(images)
        loss = criterion(outputs, labels)

        # 反向传播和优化
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # 统计训练过程中的损失值和准确率
        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total_predictions += labels.size(0)
        correct_predictions += (predicted == labels).sum().item()

    # 计算训练集的平均损失值和准确率
    train_loss = running_loss / len(train_loader)
    train_accuracy = correct_predictions / total_predictions
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)

    # 在每个epoch结束后，在测试集上评估模型
    model.eval()

    test_correct_predictions = 0
    test_total_predictions = 0

    for images, labels in test_loader:
        # 将数据移动到设备上
        images = images.to(device)
        labels = labels.to(device)

        # 前向传播
        outputs = model(images)

        # 统计测试集上的准确率
        _, predicted = torch.max(outputs.data, 1)
        test_total_predictions += labels.size(0)
        test_correct_predictions += (predicted == labels).sum().item()

    # 计算测试集的准确率
    test_accuracy = test_correct_predictions / test_total_predictions
    test_accuracies.append(test_accuracy)

    # 打印训练过程中的损失值和准确率
    print(f"Epoch [{epoch+1}/{num_epochs}], Train Loss:
{train_loss:.4f}, Train Accuracy: {train_accuracy:.4f}, Test Accuracy:
{test_accuracy:.4f}")
```

以上代码的输出结果如图 8-21 所示。从运行结果可见，模型在训练集上收敛的比较快，第 9 次迭代时，准确率即达到了 1（100%），而在测试集上的准确率最高只达到了 60%左右，

说明模型存在过拟合问题。

```
Epoch [1/20], Train Loss: 4.8114, Train Accuracy: 0.2838, Test Accuracy: 0.3535
Epoch [2/20], Train Loss: 1.3457, Train Accuracy: 0.3724, Test Accuracy: 0.3854
Epoch [3/20], Train Loss: 0.9455, Train Accuracy: 0.6112, Test Accuracy: 0.5318
Epoch [4/20], Train Loss: 0.6007, Train Accuracy: 0.7735, Test Accuracy: 0.5223
Epoch [5/20], Train Loss: 0.2920, Train Accuracy: 0.9236, Test Accuracy: 0.5064
Epoch [6/20], Train Loss: 0.1112, Train Accuracy: 0.9782, Test Accuracy: 0.5828
Epoch [7/20], Train Loss: 0.0358, Train Accuracy: 0.9959, Test Accuracy: 0.5860
Epoch [8/20], Train Loss: 0.0160, Train Accuracy: 0.9973, Test Accuracy: 0.6146
Epoch [9/20], Train Loss: 0.0059, Train Accuracy: 1.0000, Test Accuracy: 0.5796
Epoch [10/20], Train Loss: 0.0021, Train Accuracy: 1.0000, Test Accuracy: 0.6146
Epoch [11/20], Train Loss: 0.0010, Train Accuracy: 1.0000, Test Accuracy: 0.6146
Epoch [12/20], Train Loss: 0.0007, Train Accuracy: 1.0000, Test Accuracy: 0.6146
Epoch [13/20], Train Loss: 0.0006, Train Accuracy: 1.0000, Test Accuracy: 0.6051
Epoch [14/20], Train Loss: 0.0005, Train Accuracy: 1.0000, Test Accuracy: 0.6083
Epoch [15/20], Train Loss: 0.0004, Train Accuracy: 1.0000, Test Accuracy: 0.6115
Epoch [16/20], Train Loss: 0.0004, Train Accuracy: 1.0000, Test Accuracy: 0.6083
Epoch [17/20], Train Loss: 0.0003, Train Accuracy: 1.0000, Test Accuracy: 0.6083
Epoch [18/20], Train Loss: 0.0003, Train Accuracy: 1.0000, Test Accuracy: 0.6051
Epoch [19/20], Train Loss: 0.0003, Train Accuracy: 1.0000, Test Accuracy: 0.6051
Epoch [20/20], Train Loss: 0.0002, Train Accuracy: 1.0000, Test Accuracy: 0.6051
```

图8-4　ConvNet_1 模型训练结果

为了改善模型的训练和泛化能力，我们在网络中加入批归一化操作，代码如下。

代码8-5　加入批归一化的卷积神经网络模型

```
import torch.nn as nn

# 定义卷积神经网络模型
class ConvNet_2(nn.Module):
    def __init__(self, num_classes):
        super(ConvNet_2, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.BatchNorm2d(16),  # 添加批归一化层
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.BatchNorm2d(32),  # 添加批归一化层
            nn.Flatten(),
            nn.Linear(32 * 56 * 56, 256),
            nn.ReLU(),
            nn.BatchNorm1d(256),  # 添加批归一化层
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        return self.model(x)

# 实例化卷积神经网络模型
model = ConvNet_2(num_classes)

# 模型移动到指定的设备GPU/CPU
model.to(device)
```

使用以上模型进行训练的结果如图 8-22 所示，在测试集上的准确率提高到了最高 69%左右，可见，增加批归一化后，模型的泛化能力有所提高，但效果依然不太理想。

```
Epoch [1/20], Train Loss: 1.3990, Train Accuracy: 0.4966, Test Accuracy: 0.4809
Epoch [2/20], Train Loss: 0.8395, Train Accuracy: 0.6930, Test Accuracy: 0.6688
Epoch [3/20], Train Loss: 0.5746, Train Accuracy: 0.8240, Test Accuracy: 0.6911
Epoch [4/20], Train Loss: 0.3760, Train Accuracy: 0.8854, Test Accuracy: 0.6529
Epoch [5/20], Train Loss: 0.2084, Train Accuracy: 0.9468, Test Accuracy: 0.6783
Epoch [6/20], Train Loss: 0.1070, Train Accuracy: 0.9823, Test Accuracy: 0.6752
Epoch [7/20], Train Loss: 0.0486, Train Accuracy: 0.9932, Test Accuracy: 0.6465
Epoch [8/20], Train Loss: 0.0246, Train Accuracy: 0.9986, Test Accuracy: 0.6943
Epoch [9/20], Train Loss: 0.0091, Train Accuracy: 1.0000, Test Accuracy: 0.6815
Epoch [10/20], Train Loss: 0.0057, Train Accuracy: 1.0000, Test Accuracy: 0.6911
Epoch [11/20], Train Loss: 0.0041, Train Accuracy: 1.0000, Test Accuracy: 0.6783
Epoch [12/20], Train Loss: 0.0031, Train Accuracy: 1.0000, Test Accuracy: 0.6815
Epoch [13/20], Train Loss: 0.0025, Train Accuracy: 1.0000, Test Accuracy: 0.6752
Epoch [14/20], Train Loss: 0.0021, Train Accuracy: 1.0000, Test Accuracy: 0.6783
Epoch [15/20], Train Loss: 0.0017, Train Accuracy: 1.0000, Test Accuracy: 0.6783
Epoch [16/20], Train Loss: 0.0015, Train Accuracy: 1.0000, Test Accuracy: 0.6752
Epoch [17/20], Train Loss: 0.0013, Train Accuracy: 1.0000, Test Accuracy: 0.6720
Epoch [18/20], Train Loss: 0.0011, Train Accuracy: 1.0000, Test Accuracy: 0.6752
Epoch [19/20], Train Loss: 0.0010, Train Accuracy: 1.0000, Test Accuracy: 0.6720
Epoch [20/20], Train Loss: 0.0009, Train Accuracy: 1.0000, Test Accuracy: 0.6720
```

图8-5　加入批归一化后的 ConvNet_2 模型训练结果

为了进一步提高性能，我们将模型更改为 VGGNet 网络中的 VGG-16 模型，代码如下。

代码8-6　构建 VGGNet-16

```
import torch.nn as nn
from torchvision.models import vgg16

# 加载预训练的 VGG16 模型
model = vgg16(pretrained=True)

# 修改分类器以匹配类的数量
model.classifier[6] = nn.Linear(4096, num_classes)
model.classifier.add_module("7", nn.Softmax(dim=1))

# 模型移动到指定的设备 GPU/CPU
model.to(device)
```

预训练模型

在以上代码中，使用了 torchvision.models 模块中的 VGG-16 预训练模型。model = vgg16(pretrained=True)加载了预训练的 VGG-16 模型，pretrained=True 表示加载在 ImageNet 数据集上预训练的权重。

model.classifier[6] = nn.Linear(4096, num_classes)修改了 VGG-16 模型的分类器部分。VGG-16 模型的分类器是一个全连接层序列，包含多个线性层和激活函数。通过修改第 6 个线性层（索引从 0 开始），将其输出大小改为 num_classes，即所需的类别数量。

model.classifier.add_module("7", nn.Softmax(dim=1))添加了一个新的模块到 VGG-16 模型的分类器部分。该模块是一个 softmax 层，用于对模型的输出进行概率归一化，使其表示各个类别的概率分布。

softmax 函数

此外，还要将优化器改为随机梯度下降，即：

optimizer = optim.SGD(model.parameters(), lr=0.001)

使用 VGG-16 训练的结果如图 8-23 所示。从结果可见，同样迭代 20 次，其在测试集上的准确率可以达到 94%，相比之前的网络，有大幅提升。并且，模型在训练集上的准确率还没有收敛，读者可以尝试增加迭代次数，观察是否还可以继续提升模型的性能。

```
Epoch [1/20], Train Loss: 1.5989, Train Accuracy: 0.2538, Test Accuracy: 0.3535
Epoch [2/20], Train Loss: 1.5464, Train Accuracy: 0.4980, Test Accuracy: 0.5000
Epoch [3/20], Train Loss: 1.4625, Train Accuracy: 0.6166, Test Accuracy: 0.6210
Epoch [4/20], Train Loss: 1.3439, Train Accuracy: 0.7231, Test Accuracy: 0.7643
Epoch [5/20], Train Loss: 1.2195, Train Accuracy: 0.8158, Test Accuracy: 0.8535
Epoch [6/20], Train Loss: 1.1202, Train Accuracy: 0.8745, Test Accuracy: 0.8885
Epoch [7/20], Train Loss: 1.0609, Train Accuracy: 0.9154, Test Accuracy: 0.9076
Epoch [8/20], Train Loss: 1.0287, Train Accuracy: 0.9222, Test Accuracy: 0.9140
Epoch [9/20], Train Loss: 1.0109, Train Accuracy: 0.9263, Test Accuracy: 0.9140
Epoch [10/20], Train Loss: 0.9983, Train Accuracy: 0.9291, Test Accuracy: 0.9204
Epoch [11/20], Train Loss: 0.9898, Train Accuracy: 0.9345, Test Accuracy: 0.9268
Epoch [12/20], Train Loss: 0.9833, Train Accuracy: 0.9386, Test Accuracy: 0.9236
Epoch [13/20], Train Loss: 0.9776, Train Accuracy: 0.9441, Test Accuracy: 0.9299
Epoch [14/20], Train Loss: 0.9715, Train Accuracy: 0.9454, Test Accuracy: 0.9268
Epoch [15/20], Train Loss: 0.9671, Train Accuracy: 0.9550, Test Accuracy: 0.9236
Epoch [16/20], Train Loss: 0.9621, Train Accuracy: 0.9550, Test Accuracy: 0.9268
Epoch [17/20], Train Loss: 0.9579, Train Accuracy: 0.9618, Test Accuracy: 0.9299
Epoch [18/20], Train Loss: 0.9544, Train Accuracy: 0.9645, Test Accuracy: 0.9268
Epoch [19/20], Train Loss: 0.9513, Train Accuracy: 0.9659, Test Accuracy: 0.9299
Epoch [20/20], Train Loss: 0.9489, Train Accuracy: 0.9673, Test Accuracy: 0.9395
```

图8-6　使用 VGG-16 网络训练结果

最后，对于如图 8-24 所示的文件名为“gouqi.jpg”的枸杞图片，使用训练好的模型进行预测。

图8-7　用于测试的图片

进行预测的代码如下。

代码8-7　使用模型进行预测

```
from PIL import Image

#加载预测图片并进行预处理
test_image_path = 'gouqi.jpg'
```

```
test_image = Image.open(test_image_path).convert('RGB')
test_image = transform(test_image).unsqueeze(0).to(device)

#将图片输入模型进行预测
model.eval()
with torch.no_grad():
   output = model(test_image)
   _, predicted_idx = torch.max(output, 1)

#获取预测类别名
predicted_label = dataset.classes[predicted_idx.item()]
print(f"Predicted Label: {predicted_label}")
```

以上代码的预测结果如图 8-25 所示，说明预测是正确的。

```
Predicted Label: gouqi
```

图8-8　VGG-16 模型预测结果

读者可以以本案例为基础，增加新的中药材的图片集，扩充数据集，经过训练后就可以识别更多种类的中药材。

为了更加便捷地应用模型，我们可以将模型部署在网站或手机上，这就需要模型的保存和部署，并且涉及到前、后端开发和模型集成等方面的技术。

模型部署