使用ResNet模型实现图片分类的大致步骤如下：
加载数据并对其进行预处理->调用模型->训练模型->模型评估

1. 加载数据并对其进行预处理

```
# 数据预处理
data_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean = [0.491, 0.482, 0.447],std = [0.247, 0.244, 0.262]) # 此处经由程序
])
```

数据集内的图片原尺寸为32x32，而ResNet对图片尺寸的要求是224x224，但如果直接放大到224x224，图片边缘的无关元素反而会影响模型评判数据，所以放大到一个稍大的尺寸256x256，再进行中心裁剪。
除此之外，神经网络所接受的图片也需要变为torch.Tensor形式。而归一化和规范化可以消除不同图像之间的量纲和量纲单位差异。

```
# 数据加载
train_dataset = datasets.ImageFolder(root='D:\\code\\eye\\dataset\\train',transform=data_transfo
train_loader = DataLoader(train_dataset,batch_size=50,shuffle=True)
test_dataset = datasets.ImageFolder(root='D:\\code\\eye\\dataset\\test',transform=data_transforr
test_loader = DataLoader(test_dataset,batch_size=50,shuffle=False) # 测试集没有必要打乱顺序
```

数据集可能是按照一定的规律存储数据的，比如按照图片的亮度排序，但是模型需要学习的不是这种顺序而是每个照片的特性，所以在训练的时候需要打乱加载图片的顺序。而测试集并不需要乱序。
2. 调用模型

```
# 调用模型并将其移动到GPU上
model = models.resnet18(pretrained=True)
change = model.fc.in_features
model.fc = torch.nn.Linear(change, 10) # 上述在调用resnet18模型时，选择默认分为1000类，需要更改参数
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")  # 如果没有可用的gpu则会采
model = model.to(device)
# 定义损失函数与优化器
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)# 设置动量为0.9以加速收敛
```

在调用模型时，对模型进行了预训练，提高了模型分类速度和准确度，但是预训练应用的ImageNet数据集，分为1000类，所以需要根据自己数据的类别数调整。
3. 训练模型

```
# 模型训练
model.train()
for inputs, labels in train_loader:
    # 前向传播
    inputs, labels = inputs.to(device), labels.to(device)
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    # 反向传播
    loss.backward()
    optimizer.step()
```

model.train()将模式设置为训练模式，某些特定的层（如Dropout层和BatchNormalization层）在训练和测试时的行为通常是不同的。虽然ResNet-18没有这样的层，但仍然规范写出。
optimizer.zero_grad()函数归零梯度，因为在backward()函数更新梯度时，实际上是在累积梯度而不是替代。所以需要每次迭代时先归零再累积以实现替代。
criterion(outputs, labels)第一个参数代表预测标签，第二个参数代表真实标签，criterion返回一批次的平均损失值，但返回类型是一个一维张量。
optimizer.step()用于根据计算出的梯度更新模型的参数。

1. 模型评估

```
# 模型评估
model.eval()
test_loss = 0.0
correct = 0
total = 0
with torch.no_grad():  # 不计算梯度，节省计算资源
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        test_loss += loss.item() * inputs.size(0)
        predicted = torch.max(outputs, 1)[1]
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

# 计算测试集上的损失和准确率
test_loss /= total
test_acc = 100 * correct / total
test_losses.append(test_loss)
test_accuracies.append(test_acc)
```

loss.item() * inputs.size(0)，因为criterion此处返回的loss是一个一维张量，需要用.item()转换为python中的浮点数进行运算，input.size(0)返回input一维的数据，而input是一个由输入数据转化而来的二维张量，它的一维大小表示batch_size也就是一批次的数据数目。
torch.max(outputs, 1) 函数的作用是找出 outputs 在维度 1（即每个样本的输出向量）上的最大值及其索引。它返回两个张量：第一个张量包含每个样本输出向量中的最大值，第二个张量包含这些最大值在输出向量中的索引。这些索引实际上就是模型对每个样本的预测类别。这里只需要用到第二个张量，所以为torch.max(outputs,1)[1]