

E7_22336216



中山大學
SUN YAT-SEN UNIVERSITY

人工智能实验

中山大学计算机学院

人工智能

本科生实验报告

(2023学年春季学期)

课程名称：Artificial Intelligence

教学班级	DCS315	专业（方向）	计算机科学与技术（系统结构）
学号	22336216	姓名	陶宇卓

1 实验题目

实验七：CNN 中草药图像识别

一、算法原理

- 卷积神经网络（CNN）

卷积神经网络（CNN）是一种专门用于处理数据具有格栅结构（例如图像）的深度学习模型。CNN的主要构件包括卷积层、激活函数层、池化层和全连接层。

1. **卷积层（Convolutional Layer）**：卷积层是CNN的核心组件，通过对输入图像应用卷积核（也称为过滤器）来提取特征。卷积核是一个小矩阵，通常大小为3x3或5x5，在输入图像上滑动（卷积操作），与图像的每个区域进行点积运算，从而产生特征图（feature map）。

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
```

- `in_channels`：输入图像的通道数（例如RGB图像有3个通道）。
 - `out_channels`：输出特征图的通道数，即卷积核的数量。
 - `kernel_size`：卷积核的大小。
 - `stride`：卷积核的步长。
 - `padding`：在输入图像边界添加的填充值，通常用来保持特征图的尺寸。
- 输出特征图的尺寸计算

$$outputsize = \lfloor \frac{inputsize + 2 \times padding - kernelsize}{stride} \rfloor + 1$$

- 工作流程

- 卷积核初始化：卷积核的参数在模型初始化时随机设置。
- 滑动卷积核：卷积核在输入图像上滑动，每次移动一个步长（stride）。
- 点积运算：卷积核的值与覆盖的输入图像区域进行元素级的乘法运算，然后求和。
- 生成特征图：将点积运算的结果存储在特征图中对应的位置。
- 多卷积核应用：通常会使用多个卷积核，以产生多个特征图，每个特征图捕捉不同的特征。

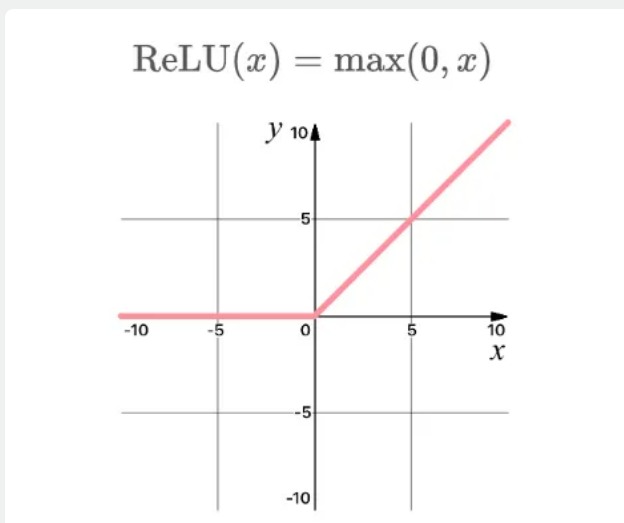
2. **批量归一化层（Batch Normalization Layer）**：批量归一化层在每个小批次上应用归一化，减小内部协变量偏移，提高模型的训练速度和稳定性。这一层通过对每个小批次的激活值进行归一化，然后应用缩放和平移来恢复数据分布。

```
torch.nn.BatchNorm2d(num_features)
```

- `num_features`：输入的特征图的通道数。
- 工作流程
 - 计算均值和方差：对每个小批次的激活值计算均值和方差。
 - 归一化：将激活值减去均值，除以标准差。
 - 缩放和平移：应用可学习的缩放参数和偏移参数，将归一化后的值重新调整。

3. **激活函数层 (ReLU)**：ReLU (Rectified Linear Unit) 是最常用的激活函数，它将所有负值变为零，保持正值不变。

```
torch.nn.ReLU()
```



- **工作流程**

- 应用ReLU函数：将输入中的负值置为零，正值保持不变。
- 输出结果：输出经过ReLU激活的特征图。

4. **池化层 (Max Pooling Layer)**：池化层通过减少特征图的尺寸，降低计算复杂度，同时保留重要特征。最大池化 (Max Pooling) 是最常用的池化方法，它取池化窗口中的最大值作为输出。

```
torch.nn.MaxPool2d(kernel_size, stride)
```

- `kernel_size`：池化窗口的大小。
- `stride`：池化窗口的步长。

- **工作流程**

- 选择池化窗口大小和步长：通常选择2x2窗口和步长为2。
- 滑动池化窗口：池化窗口在特征图上滑动，每次移动一个步长。
- 最大值计算：在每个池化窗口中选择最大值作为输出。
- 生成池化后的特征图：将最大值填入池化后的特征图中。

5. **全连接层 (Fully Connected Layer)**：全连接层将前面提取的特征整合，用于分类或回归任务。它将输入特征展平并传递给全连接的神经元，每个神经元与前一层的所有节点相连。

```
torch.nn.Linear(in_features, out_features)
```

- `in_features` : 输入特征的大小。
- `out_features` : 输出特征的大小, 即分类的类别数。

- 工作流程

- 展平输入特征: 将卷积层输出的多维特征展平成一维向量。

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

-
- 矩阵乘法: 输入向量与权重矩阵相乘, 加上偏置项, 生成输出。
- 激活函数应用: 对输出应用激活函数 (如ReLU), 引入非线性。
- 输出分类: 最后一层全连接层的输出用于分类任务, 通常通过Softmax函数转换为概率分布。

6. **Dropout层**: 在训练过程中, Dropout会以一定的概率 (称为Dropout概率或保持率, 通常为0.5) 随机丢弃 (设置为0) 一些神经元的输出。这种操作在每个训练步骤都会有所不同, 形成一个“随机子网络”。在测试过程中, 为了使输出结果稳定, 通常会将所有神经元的输出按训练时的Dropout概率进行缩放。

```
nn.Dropout(p=0.5)
```

- 损失函数和优化器

- 交叉熵损失函数 (Cross-Entropy Loss) :

损失函数用于衡量模型预测值与真实值之间的差异，反映模型的性能。通过最小化损失函数的值，模型可以提高预测的准确性。在分类任务中，交叉熵损失函数 (Cross-Entropy Loss) 是最常用的损失

函数，定义如下：

$$\text{Loss} = - \sum_{i=1}^C y_i \log(p_i)$$

C 是类别的数量。

y_i 是真实标签，如果样本属于第 i 类则为 1，否则为 0。

p_i 是模型预测样本属于第 i 类的概率。

交叉熵

首先将KL散度公式拆开：

$$\begin{aligned} D_{\text{KL}}(p||q) &= \sum_{i=1}^n p(x_i) \log\left(\frac{p(x_i)}{q(x_i)}\right) \\ &= \sum_{i=1}^n p(x_i) \log(p(x_i)) - \sum_{i=1}^n p(x_i) \log(q(x_i)) \\ &= -H(p(x)) + \left[- \sum_{i=1}^n p(x_i) \log(q(x_i)) \right] \end{aligned}$$

前者 $H(p(x))$ 表示信息熵，后者即为交叉熵，**KL散度 = 交叉熵 - 信息熵**

交叉熵公式表示为：

$$H(p, q) = - \sum_{i=1}^n p(x_i) \log(q(x_i))$$

- 优化器 (Optimizer) :

Adam优化器通过计算梯度的一阶矩估计和二阶矩估计，自适应地调整每个参数的学习率。

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize timestep)

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)

end while

return θ_t (Resulting parameters)

知乎 @Ultimate

资料显示，众多优化器中效果最好的是Adam。Adam(Adaptive Moment Estimation)本质上是带有动量项的RMSprop，它利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率。Adam的优点主要在于经过偏置校正后，每一次迭代学习率都有个确定范围，使得参数比较平稳。

规则如下：

1. 一阶矩估计（动量项）：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

2. 二阶矩估计：

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

3. 偏差校正：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

4. 参数更新：

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

• 工作流程

- 计算梯度：根据当前批次的损失，计算每个参数的梯度。
- 更新一阶矩估计：使用当前梯度更新动量项。
- 更新二阶矩估计：使用当前梯度的平方更新二阶矩估计。
- 应用偏差校正：对一阶矩估计和二阶矩估计进行偏差校正。

- 更新参数：使用校正后的梯度更新模型参数（反向传播）。

• 具体实现流程

1. **输入数据**：输入的中草药图像尺寸被统一重置为 3×128×128（3个颜色通道，128×128的图像）。

2. 卷积层1：

- 使用 `torch.nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)`，将输入图像通道数从3变为16。
- 经过卷积操作后，图像尺寸仍为 16×128×128。
- 通过批量归一化层 `torch.nn.BatchNorm2d(16)`，标准化数据。
- 经过 ReLU 激活函数 `torch.nn.ReLU()` 引入非线性。
- 最大池化层 `torch.nn.MaxPool2d(kernel_size=2, stride=2)` 将图像尺寸减半为 16×64×64。

3. 卷积层2：

- 使用 `torch.nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)`，将通道数从16变为32。
- 经过卷积操作后，图像尺寸为 32×64×64。
- 通过批量归一化层 `torch.nn.BatchNorm2d(32)`，标准化数据。
- 经过 ReLU 激活函数 `torch.nn.ReLU()` 引入非线性。
- 最大池化层 `torch.nn.MaxPool2d(kernel_size=2, stride=2)` 将图像尺寸减半为 32×32×32。

4. 全连接层：

- 将卷积层输出的特征图展平成一维向量，大小为 $32 \times 32 \times 32 = 32768$ 。
- 使用全连接层 `torch.nn.Linear(32 * 32 * 32, 5)`，输出5个类别的预测概率。

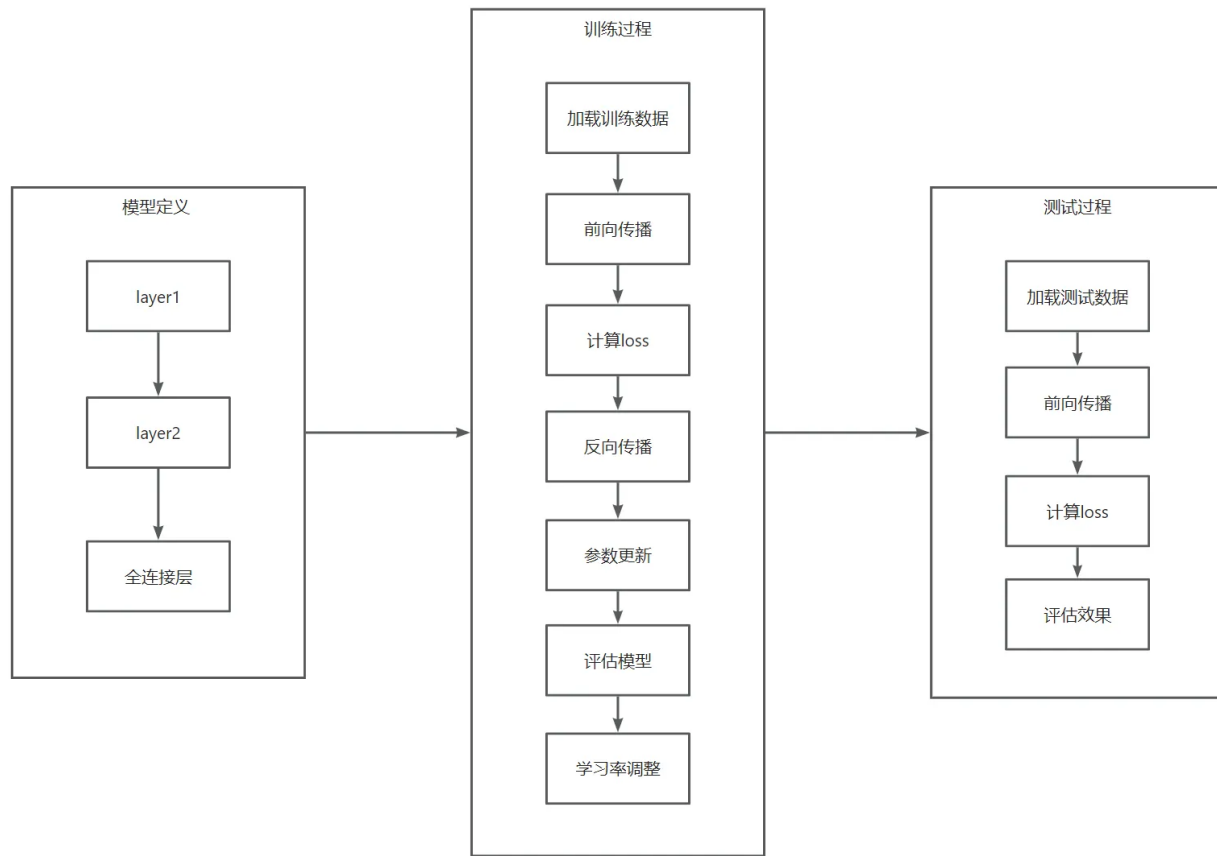
5. 损失函数和优化器：

- 使用交叉熵损失函数 `torch.nn.CrossEntropyLoss()` 计算模型预测与真实标签之间的差异。
- 使用 Adam 优化器 `torch.optim.Adam(model.parameters(), lr=0.001)` 根据损失值调整模型参数。

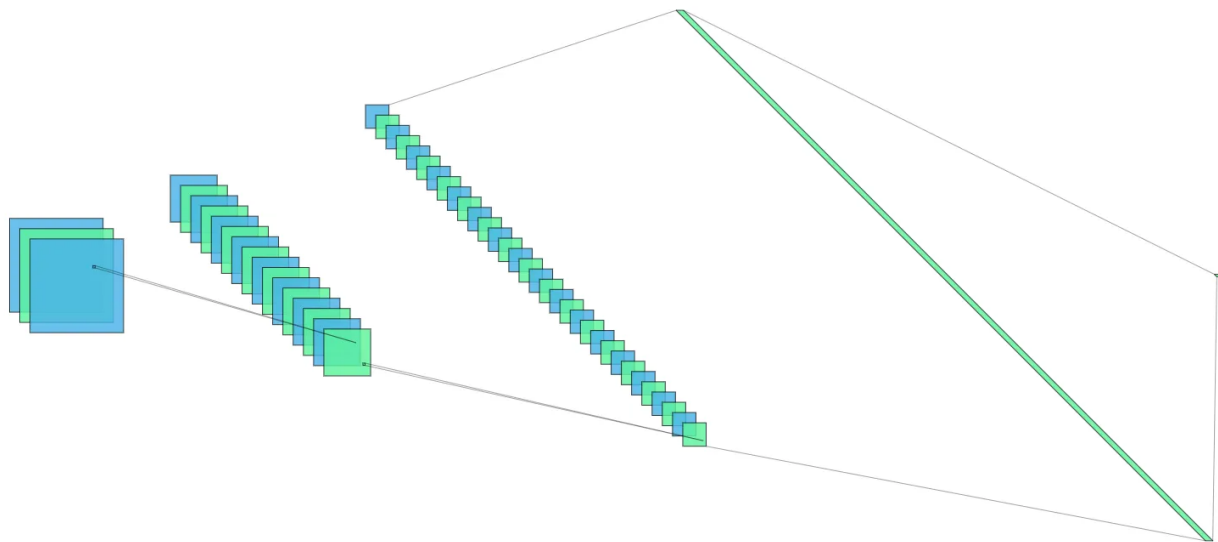
6. 训练和测试：

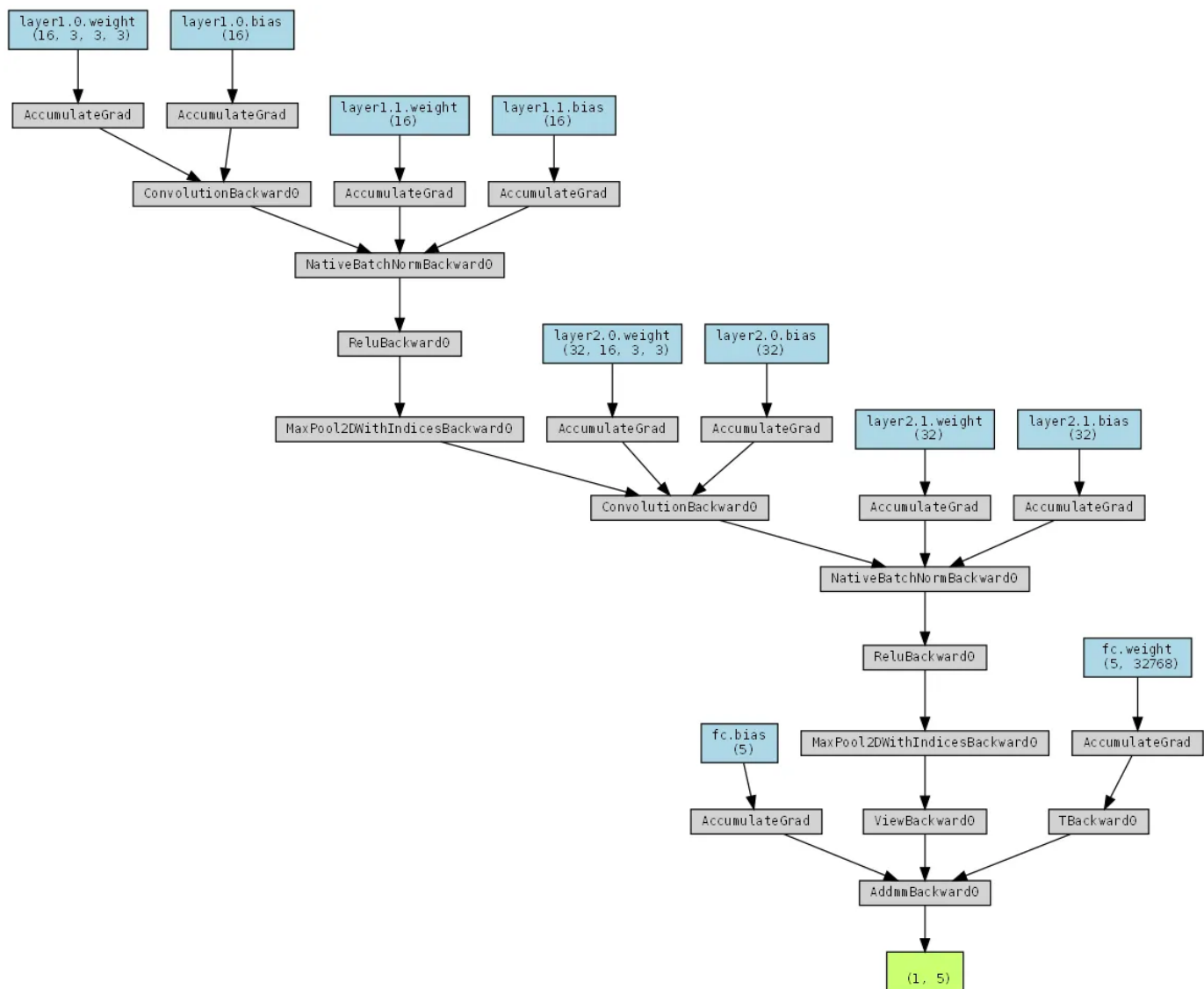
- 在训练过程中，输入图像通过上述网络结构进行前向传播，计算损失并进行反向传播以更新模型参数。
- 在每个 epoch 结束后，调整学习率，并在测试数据集上评估模型性能。

- 流程图



- 可视化神经网络（仅供参考）





二、伪代码

```
1  # 定义卷积神经网络 (CNN) 类
2  类 卷积神经网络:
3      初始化函数:
4          定义 卷积层1:
5              输入通道数 = 3
6              输出通道数 = 16
7              卷积核大小 = 3x3
8              步长 = 1
9              填充 = 1
10             批量归一化层
11             ReLU激活函数
12             最大池化层 (2x2)
13
14         定义 卷积层2:
15             输入通道数 = 16
16             输出通道数 = 32
17             卷积核大小 = 3x3
18             步长 = 1
19             填充 = 1
20             批量归一化层
21             ReLU激活函数
22             最大池化层 (2x2)
23             Dropout随机丢弃神经元
24
25         定义 全连接层:
26             输入特征大小 = 32 * 32 * 32
27             输出类别数 = 5
28
29         前向传播函数:
30             输入 x:
31             经过卷积层1处理 x
32             经过卷积层2处理 x
33             将x展平成一维向量
34             经过全连接层输出预测结果
35             返回 预测结果
36
37     # 定义训练函数
38     函数 训练():
39         实例化 卷积神经网络模型
40         定义 交叉熵损失函数
41         定义 Adam优化器
```

```

42     定义 学习率调度器（每20个epoch后，学习率衰减0.1）
43
44     初始化 训练和测试的损失和准确率列表
45
46     对于每个训练周期（epoch）在范围内：
47         初始化 当前周期的训练损失和准确率
48
49         对于每个批次（batch）在训练数据集中：
50             获取 输入图像 和 标签
51             前向传播 获取预测结果
52             计算损失
53             梯度清零
54             反向传播计算梯度
55             优化器更新模型参数
56             累加 当前批次的损失和准确率
57
58         计算并保存 当前周期的平均训练损失和准确率
59         在测试数据集上评估模型：
60             计算并保存 测试损失和准确率
61         学习率调度器步进
62
63     保存 训练好的模型参数
64     返回 训练好的模型，训练损失列表，训练准确率列表，测试损失列表，测试准确率列
    表
65
66     # 定义测试函数
67     函数 模型测试(模型):
68         初始化 测试损失和准确率
69         禁用梯度计算
70
71         对于每个批次（batch）在测试数据集中：
72             获取 输入图像 和 标签
73             前向传播 获取预测结果
74             计算损失
75             累加 当前批次的损失和准确率
76
77         计算并返回 平均测试损失和准确率
78
79     # 主程序入口
80     如果 __name__ == "__main__":
81         调用 训练函数进行训练
82

```

三、关键代码展示

完整代码见../code

- 读取数据

```
1 transform = transforms.Compose([
2     transforms.Resize((128, 128)), # 调整图片大小
3     transforms.ToTensor()
4 ])
5
6 # 导入图片数据集
7 train_data = datasets.ImageFolder(
8     root="../cnn_data/train",
9     transform=transform # 使用定义的transform变量
10 )
11
12 test_data = datasets.ImageFolder(
13     root="../cnn_data/test",
14     transform=transform # 使用定义的transform变量
15 )
16
17 train_data_loader = torch.utils.data.DataLoader(
18     dataset=train_data,
19     batch_size=50,
20     shuffle=True, # 打乱数据
21     drop_last=True) # 丢弃最后一个不完整的batch
22
23 test_data_loader = torch.utils.data.DataLoader(
24     dataset=test_data,
25     batch_size=1,
26     shuffle=False, # 不打乱数据
27     drop_last=False) # 不丢弃最后一个不完整的batch
```

- CNN模型

```

1 class CNN(torch.nn.Module):
2     def __init__(self):
3         super(CNN, self).__init__()
4         self.layer1 = torch.nn.Sequential(
5             torch.nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=
1), # 卷积层, 3个输入通道, 16个输出通道
6             torch.nn.BatchNorm2d(16), # 批标准化
7             torch.nn.ReLU(), # 激活函数
8             torch.nn.MaxPool2d(kernel_size=2, stride=2) # 池化层, 缩小
            图片尺寸
9         )
10        self.layer2 = torch.nn.Sequential(
11            torch.nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=
1),
12            torch.nn.BatchNorm2d(32),
13            torch.nn.ReLU(),
14            torch.nn.MaxPool2d(kernel_size=2, stride=2),
15            torch.nn.Dropout(0.5) # Dropout层
16        )
17        # 根据中草药类型的数量更改输出层的节点数
18        self.fc = torch.nn.Linear(32 * 32 * 32, 5) # 全连接层, 5个神经
            元
19
20    def forward(self, x):
21        x = self.layer1(x) # 通过第一个卷积层
22        x = self.layer2(x) # 通过第二个卷积层
23        x = x.view(x.size(0), -1) # 将图片数据展平
24        x = self.fc(x) # 通过全连接层
25        return x
26

```

- 训练模型

```

1  def train():
2      model = CNN().to(DEVICE)
3      criterion = torch.nn.CrossEntropyLoss()
4      optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
5
6      # 定义学习率调度器
7      scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.
1)
8
9      train_loss_list = [] # 记录每个epoch的损失
10     train_acc_list = [] # 记录每个epoch的准确率
11     test_loss_list = [] # 记录每个epoch的测试损失
12     test_acc_list = [] # 记录每个epoch的测试准确率
13
14     for epoch in range(EPOCHS):
15         train_loss = 0
16         train_acc = 0
17         # 在每个epoch开始时调用scheduler.step()
18         scheduler.step()
19
20         progress_bar = tqdm(enumerate(train_data_loader), total=len(t
rain_data_loader), desc="Epoch {}".format(epoch + 1))
21         for i, (images, labels) in progress_bar:
22             images, labels = images.to(DEVICE), labels.to(DEVICE)
23
24             outputs = model(images)
25             loss = criterion(outputs, labels)
26
27             optimizer.zero_grad()
28             loss.backward()
29             optimizer.step()
30
31             train_loss += loss.item()
32             _, predicted = torch.max(outputs.data, 1)
33             train_acc += (predicted == labels).sum().item()
34             progress_bar.set_postfix(
35                 {'Train Loss': '{:.4f}'.format(train_loss/len(train_d
ata_loader)), 'Train Accuracy': '{:.4f}'.format(train_acc/len(train_d
ata_loader.dataset))})
36
37         train_loss /= len(train_data_loader)

```

```

38         train_acc /= len(train_data_loader.dataset)
39         train_loss_list.append(train_loss)
40         train_acc_list.append(train_acc)
41
42         test_loss, test_acc = model_test(model)
43         test_loss_list.append(test_loss)
44         test_acc_list.append(test_acc)
45
46         # print(f"\nEpoch [{epoch + 1}/{EPOCHS}], Train Loss: {train_
loss:.4f}, Train Accuracy: {train_acc:.4f}, Test Loss: {test_loss:.4
f}, Test Accuracy: {test_acc:.4f}")
47         print("Progress Finished")
48         # 保存模型参数
49         torch.save(model.state_dict(), 'model.pth')
50         return model, train_loss_list, train_acc_list, test_loss_list, te
st_acc_list

```

- 测试模型


```

1 def model_test(model, epoch):
2     model.eval()
3     criterion = torch.nn.CrossEntropyLoss()
4
5     test_loss = 0
6     test_acc = 0
7     progress_bar = tqdm(enumerate(test_data_loader), total=len(test_data_loader), desc="Testing")
8     with torch.no_grad():
9         for i, (images, labels) in progress_bar:
10             images, labels = images.to(DEVICE), labels.to(DEVICE)
11
12             # 获取每一层的输出
13             x = model.layer1(images)
14             x1 = model.layer2(x)
15             outputs = model(images)
16
17             if i == 1 and epoch == 0:
18                 plot_images_and_layers(images, x, x1)
19
20             loss = criterion(outputs, labels)
21             test_loss += loss.item()
22             _, predicted = torch.max(outputs.data, 1)
23             test_acc += (predicted == labels).sum().item()
24
25             progress_bar.set_postfix(
26                 {'Test Loss': '{:.4f}'.format(test_loss / len(test_data_loader)),
27                  'Test Accuracy': '{:.4f}'.format(test_acc / len(test_data_loader.dataset))})
28
29     test_loss /= len(test_data_loader)
30     test_acc /= len(test_data_loader.dataset)
31     return test_loss, test_acc

```

四、创新点&优化

- 可视化神经网络

在graph.py里调用了torchviz库实现了网络结构的可视化，另外  NN SVG 提供了CNN可视化的工具。

- 数据预处理与加载

对不同大小的图片进行格式化处理，提高了训练效果和速度

- **使用批标准化层 (BatchNorm2d)**

1. 加速收敛速度：批标准化可以使每一层的输入分布稳定，减少了训练过程中参数的变化范围，有助于加速收敛速度。这意味着网络可以更快地学习到数据的表示。
2. 减少梯度消失或爆炸：在深度神经网络中，梯度消失或爆炸是常见的问题，尤其是在网络层数较多时。批标准化通过规范化每一层的输入，可以有效地减少这些问题的发生，使得梯度更加稳定，从而有助于更深层次的网络训练。
3. 降低对初始化的敏感性：批标准化使得网络对初始权重的选择不再那么敏感。即使使用较大的学习率进行训练，也不太容易出现训练不稳定的情况，这为神经网络的训练提供了更多的灵活性。
4. 正则化作用：批标准化在一定程度上具有正则化的效果，因为它在每个小批量数据上计算均值和方差，并将其作为额外的参数进行训练。这种额外的噪声有助于降低模型的过拟合风险。
5. 使得激活函数的输入更加均匀：激活函数的输入如果分布不均匀，可能会导致梯度消失或爆炸，批标准化可以使得每个神经元的输入更加均匀，有助于提高模型的稳定性和泛化能力。

- **使用Dropout层防止过拟合**

1. 减少过拟合：Dropout通过随机丢弃神经元，使得每个神经元不能过于依赖其他特定的神经元，从而强制网络学习更通用的特征。模型在面对不同的输入时表现得更为稳定和鲁棒，能够更好地应对噪声和不确定性。
2. 增强泛化能力：通过在训练过程中引入随机性，Dropout可以有效地减少过拟合，从而提升模型在未见数据上的泛化能力。

- **使用学习率调度器 (StepLR)**

```
scheduler = lr_scheduler.StepLR(optimizer, step_size=15, gamma=0.1)
```

1. 提高训练效率：使用学习率调度器可以显著加速模型的收敛速度。高学习率有助于快速跳出局部最小值，而低学习率则帮助模型在最优解附近进行精细调整。在训练的不同阶段调整学习率，可以使模型更快地收敛到最优解。初始阶段使用较高的学习率有助于快速接近最优解，而后期使用较低的学习率则有助于精细调整参数，避免震荡。
2. 防止过拟合：在训练的后期，通过降低学习率，可以减少参数更新的幅度，从而使模型在训练集上的性能趋于稳定，防止模型过拟合到训练数据。在训练过程中，动态调整学习率可以防止学习率过大导致的训练不稳定现象，以及学习率过小导致的收敛速度慢的问题。通过调度器的调整，模型训练可以更加平稳和高效。

- **进度条显示**

1. 了解训练进度：进度条可以实时显示当前训练的进度，包括已经完成的epoch和batch。这样可以让你随时知道训练过程所处的阶段。

2. 时间预估：进度条通常会显示预计的剩余时间，这有助于规划和管理时间，尤其是对于训练时间较长的任务。
 3. 监控指标：在进度条中可以显示当前的训练损失、准确率等指标，这有助于实时监控模型的性能，及时发现和调整问题。
 4. 用户友好：进度条可以提供直观的视觉反馈，使用户能够更容易地理解和跟踪训练过程，而不是面对一长串的日志信息。
- 可视化每层输出结果

Python |

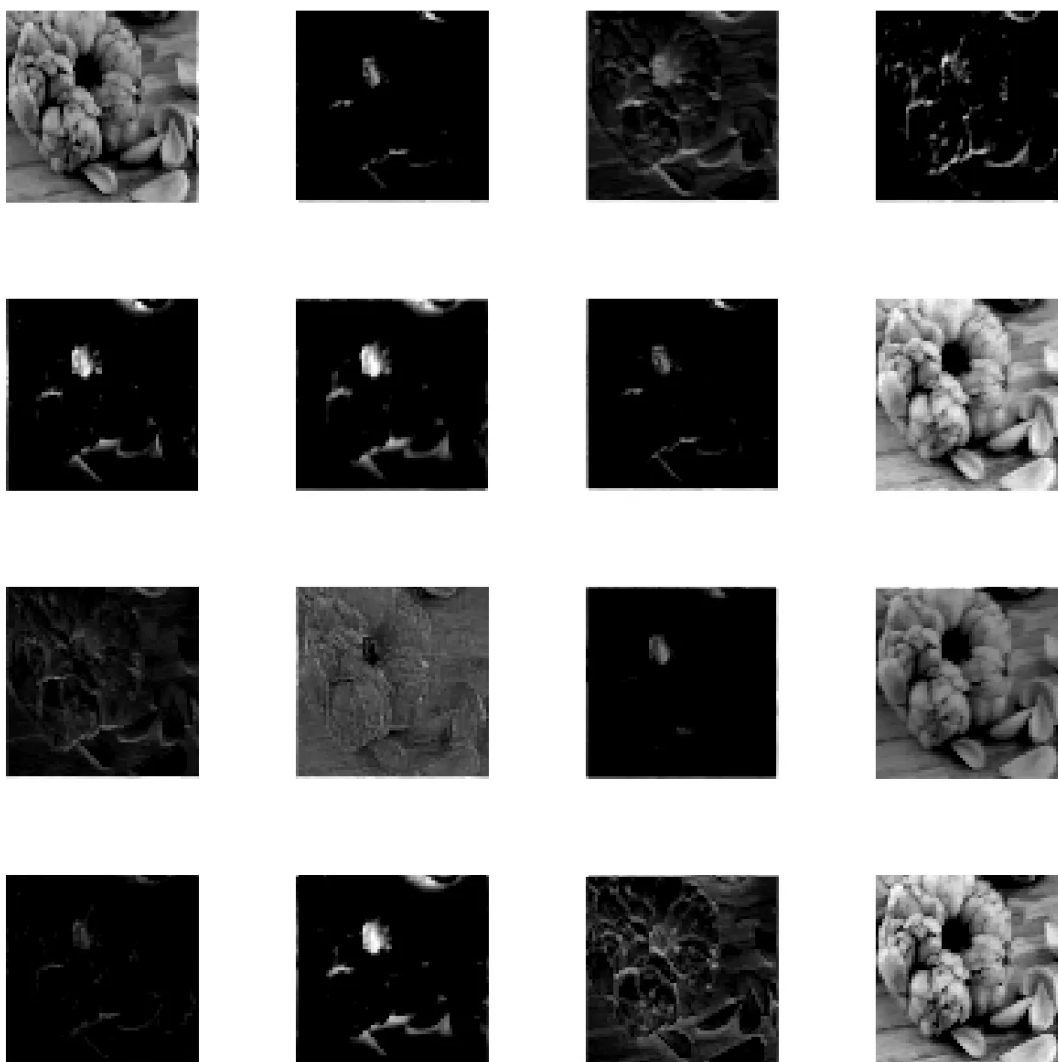
```
1 def plot_images_and_layers(images, x, x1):
2     fig, axs = plt.subplots(1, 3, figsize=(15, 15))
3     for k in range(3):
4         axs[k].imshow(images[0, k, :, :].cpu().numpy(), cmap='gray')
5         axs[k].axis('off')
6     plt.suptitle('Input Image')
7     plt.savefig('../pics/input_image.png')
8
9     for j, img in enumerate([x, x1]):
10        img = img.detach().cpu().numpy()
11        img = np.transpose(img, (
12            0, 2, 3, 1)) # (batch_size, channels, height, width) -> (batch_size, height, width, channels)
13        # img = (img - np.min(img)) / (np.max(img) - np.min(img))
14        num_channels = img.shape[3] # 获取通道数
15        fig, axs = plt.subplots(4, num_channels // 4, figsize=(15, 15)) # 创建一个1行, num_channels列的子图
16
17        for m in range(4):
18            for n in range(num_channels // 4):
19                channel_idx = m * 4 + n
20                axs[m, n].imshow(img[0, :, :, channel_idx], cmap='gray') # 在第i个子图中显示第i个通道的图像
21                axs[m, n].axis('off') # 关闭坐标轴
22                plt.subplots_adjust(wspace=0.5, hspace=0.5) # 设置子图之间的间隔
23
24        plt.suptitle(f'Layer{j + 1} Output') # 设置标题
25        plt.savefig(f'../pics/layer{j + 1}_output.png')
```

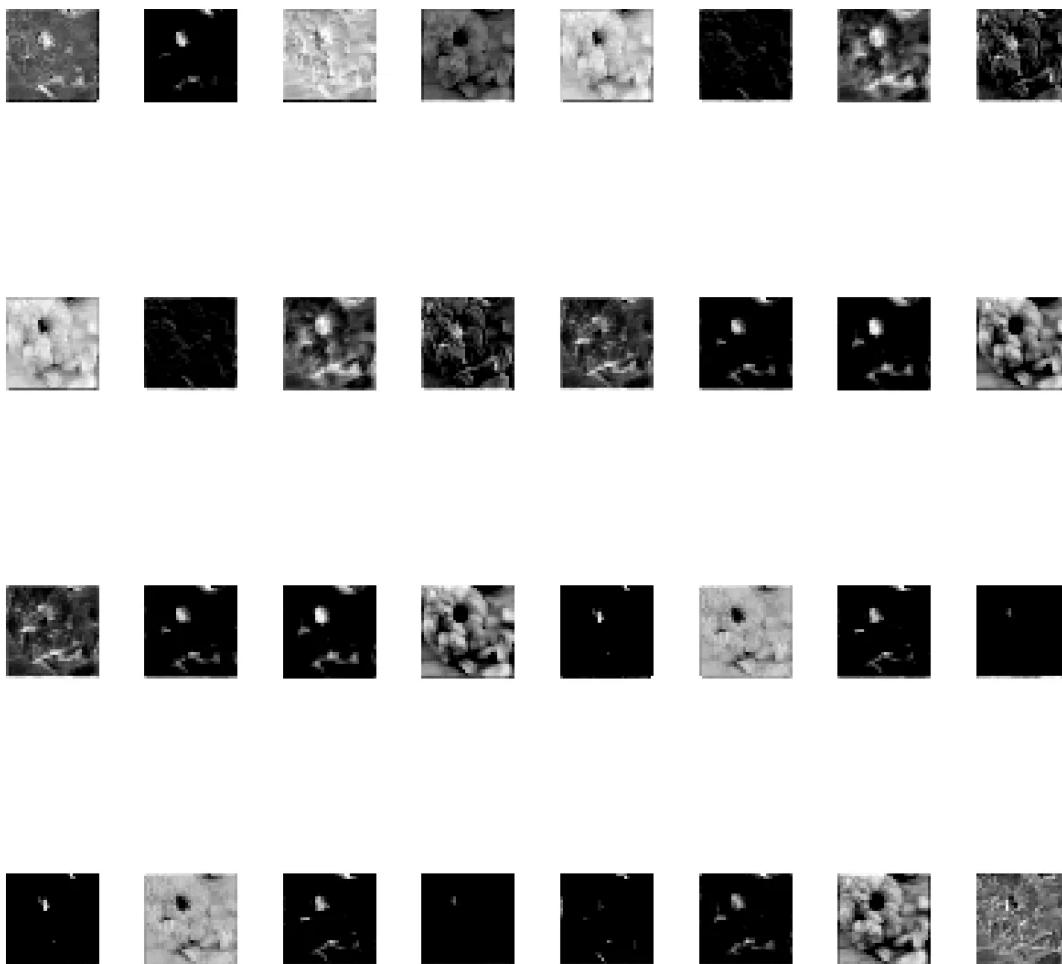
通过可视化，可以看到每一层的卷积核在输入图像上提取了哪些特征，如边缘、纹理、颜色等。这有助于理解模型是如何逐步从低级特征构建到高级特征的，输出如下：

Input Image



Layer1 Output



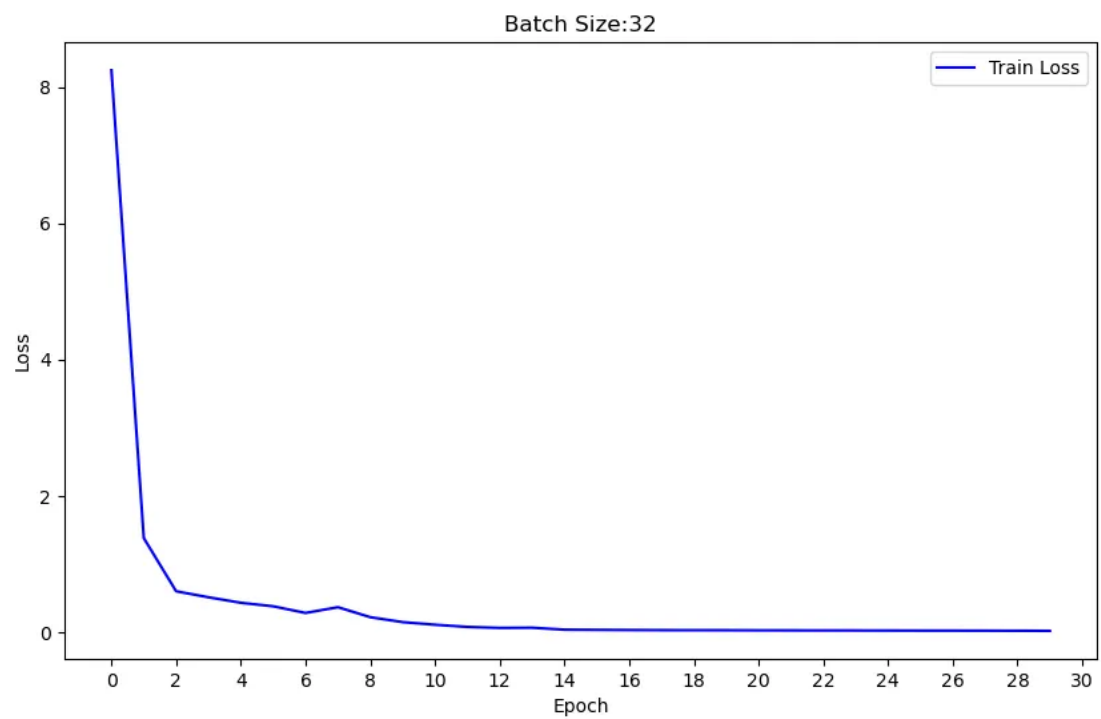
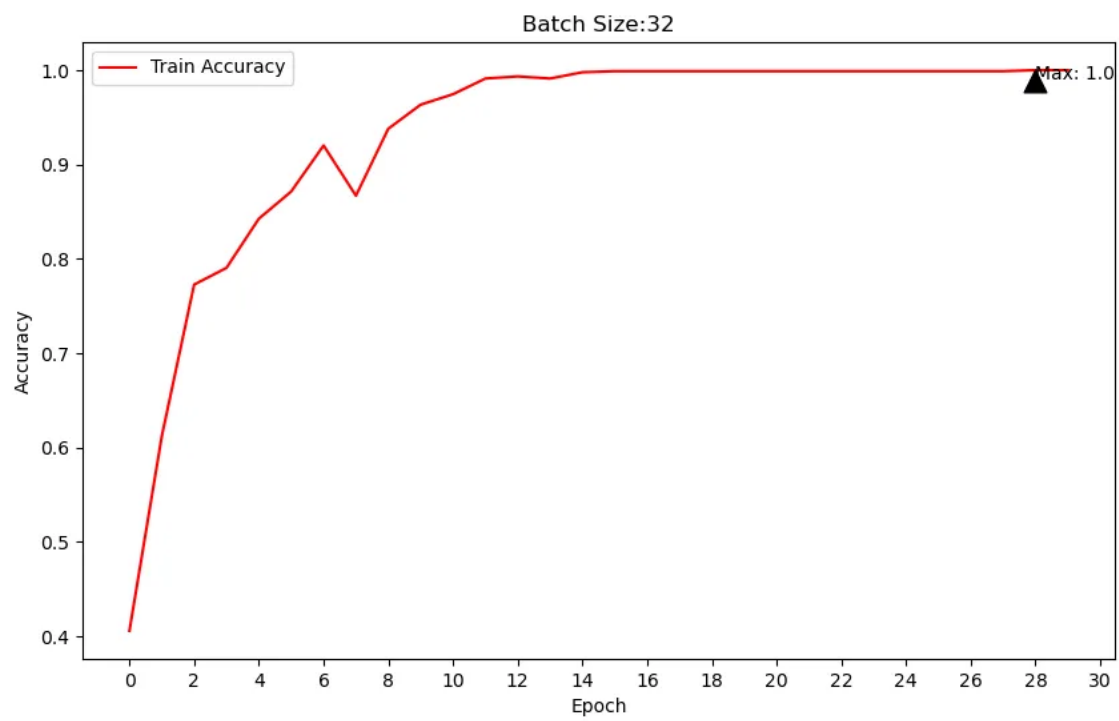


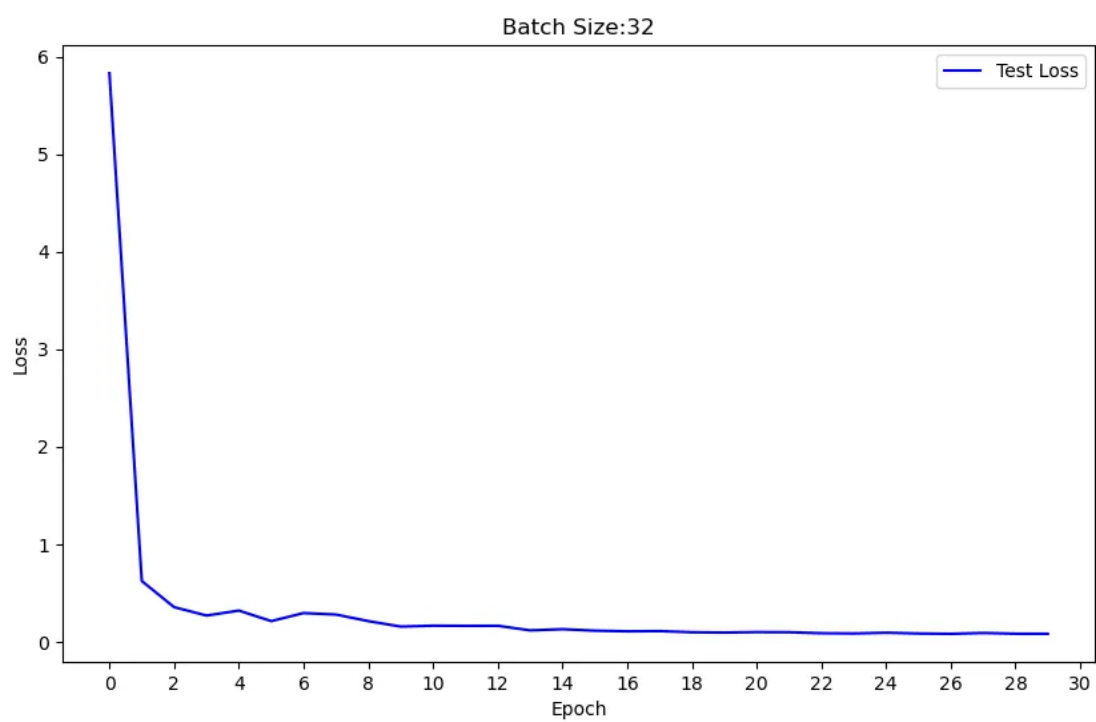
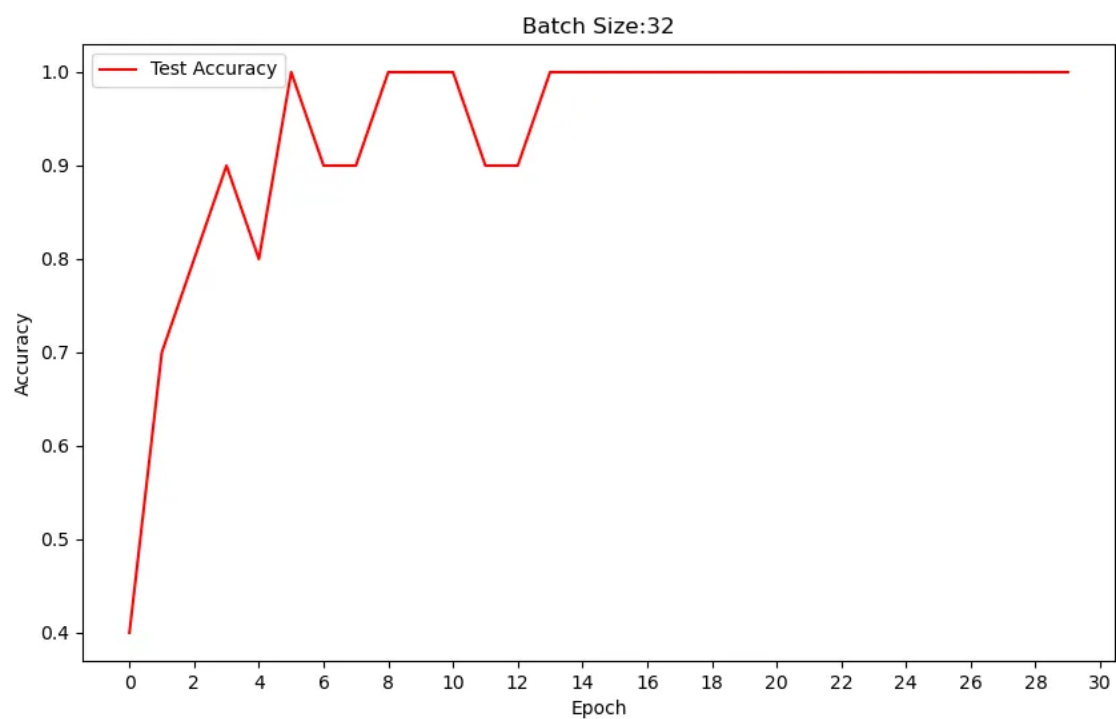
3 实验结果及分析

1. 实验结果展示，评测指标分析

选取 `Batchsize` =32, `epoch` =30, `lr` =0.001, 衰减代数15, 衰减率 `gamma` =0.1

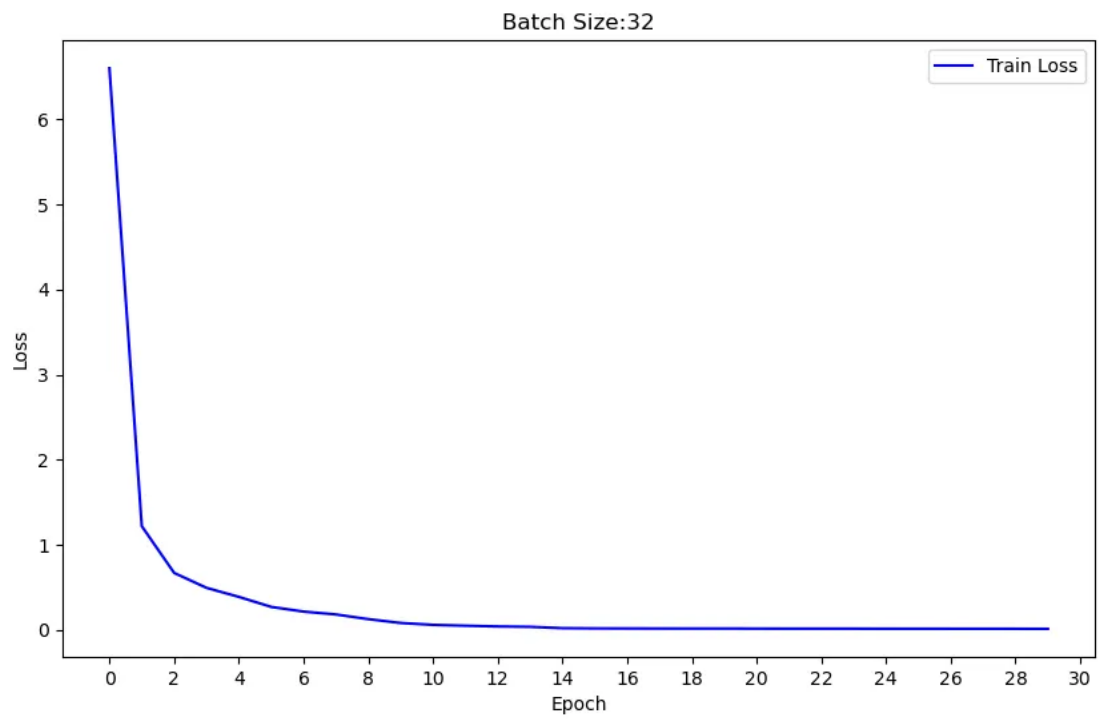
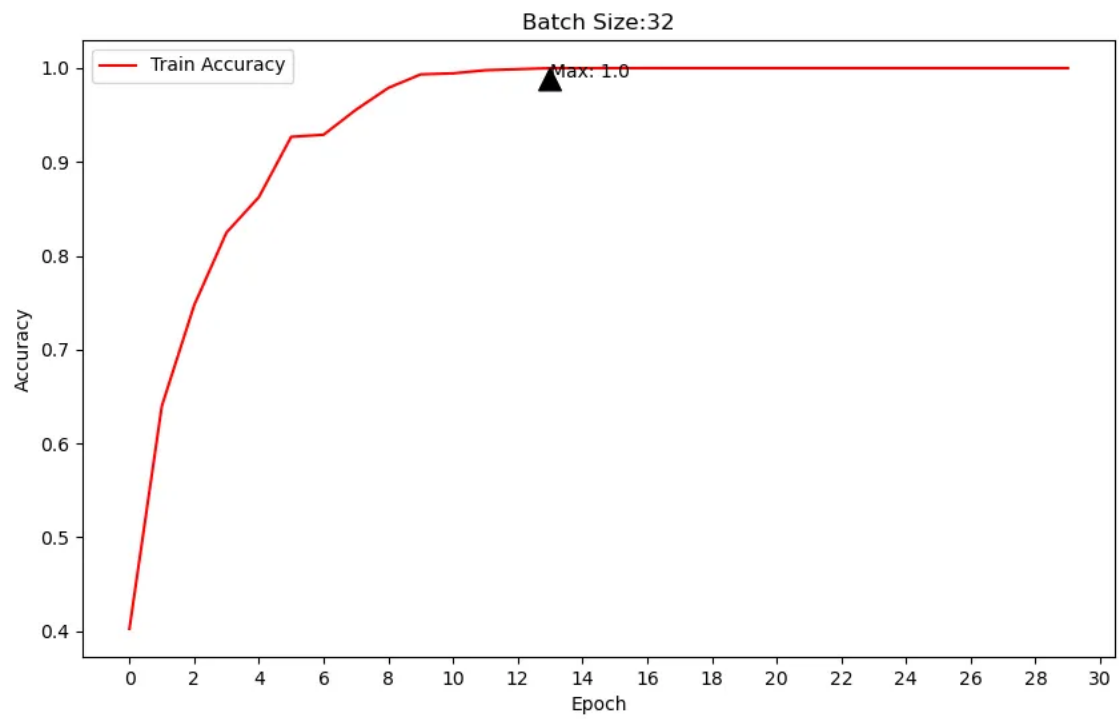
未采用Dropout时，训练过程中的指标如下，最高准确率100%，最小损失值0.0219。相对应的，测试过程中的相应指标如下，最高准确率100%，最小损失值0.0851，如下图所示。

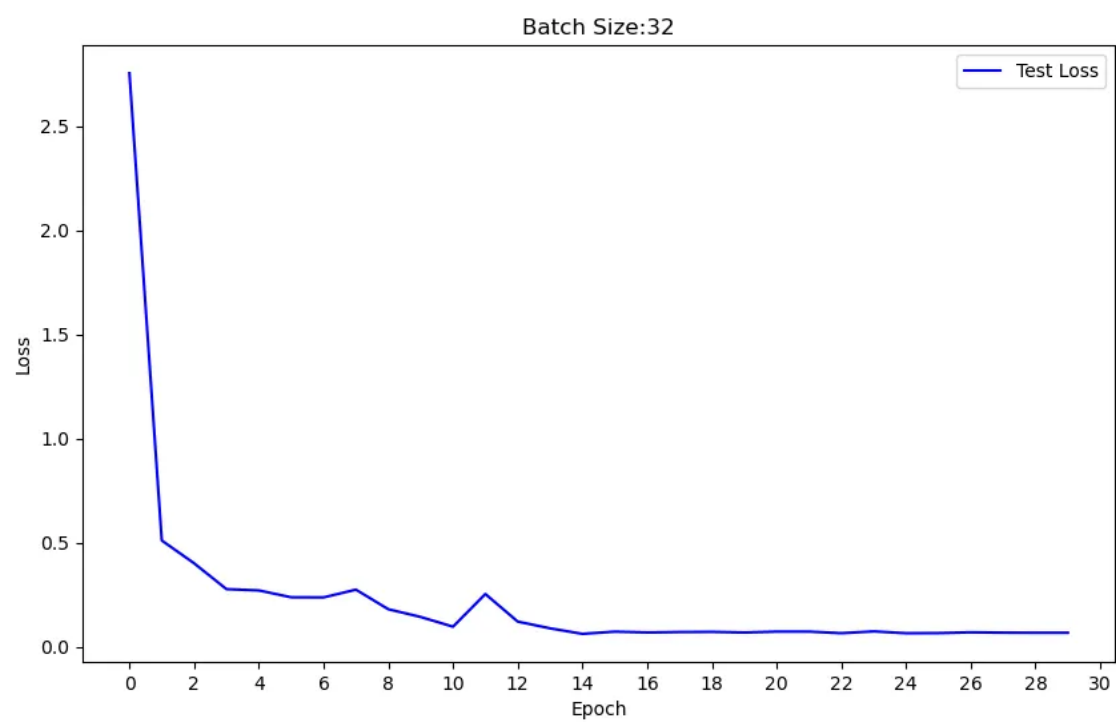
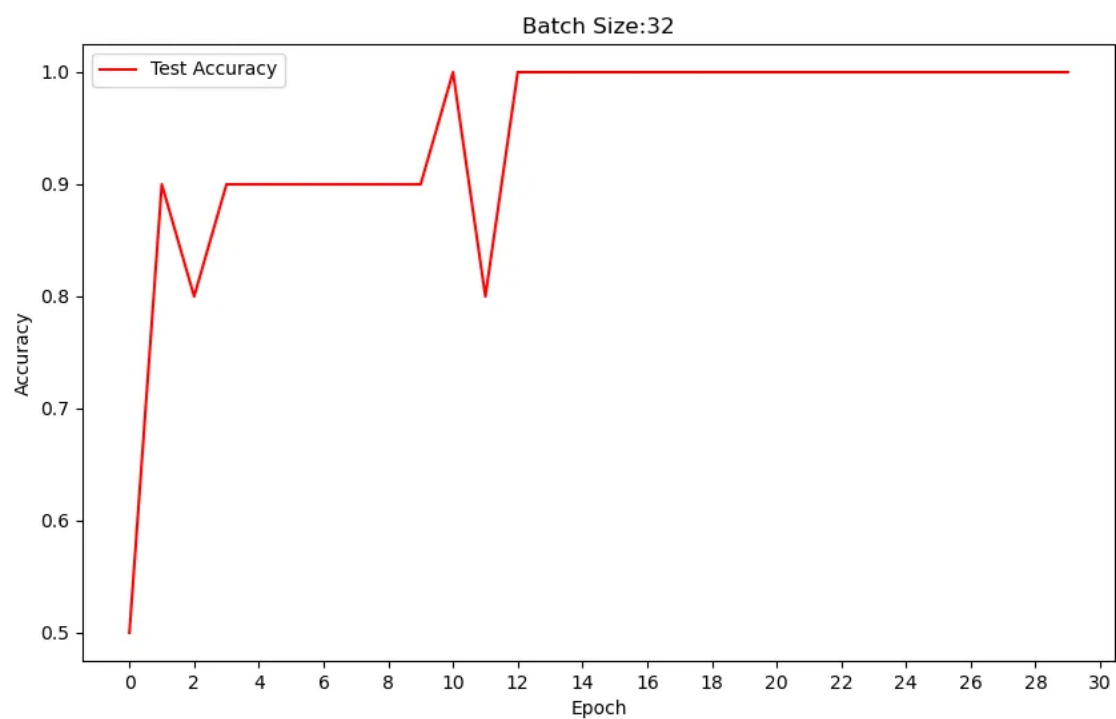




采用Dropout时，训练过程中的指标如下，最高准确率100%，最小损失值0.0112。相对应的，测试过程中的相应指标如下，最高准确率100%，最小损失值0.0687，如下图所示。

可见效果变好了，防止了过拟合。





在相等的lr和epoch条件下，将Adam与SGD等优化器比较训练集上的结果如下，可见Adam效果最好，原因可见算法原理部分。

type	min loss	max accuracy/所需最小代数
Adam	0.0112	100%/13

SGD	0.3436	92.68%/30
SGD+Momentum	0.0357	100%/30
RMSprop	0.0535	99.66%/30

时空复杂度分析：

- **时间复杂度：** $O(E \times (N + B \times H \times W \times C \times K^2 \times F))$ 。其中 E 是总的 epoch 数，N 是数据集大小，B 是 batch 大小，H 和 W 是图像尺寸，C 是通道数，K 是卷积核尺寸，F 是输出通道数。
- **空间复杂度：** $O(K^2 \times C \times F) + O(D \times O) + O(B \times H \times W \times F)$ 。其中 D 是展平后的特征向量维度，O 是输出类别数。

4 思考题

无

5 参考资料

实验课PPT

 [NN SVG](#)

 [CNN Explainer](#)