



安徽大学
人工智能学院
School of Artificial Intelligence
Anhui University

《计算机视觉实验》报告

实验 1-6

学 院: 人工智能学院

专 业: 人工智能

学 生: 杨跃浙 WA2214014

指导老师: 张鹏

课程编号: ZH52407

课程学分: 1

提交日期: 25.04.20

目 录

1 实验一：图像去噪	6
1.1 实验目的	6
1.2 实验要求	6
1.3 实验原理	6
1.3.1 椒盐噪声 (Salt Pepper)	6
1.3.2 基于像素的均值滤波原理 (Mean)	6
1.3.3 基于像素的中值滤波原理 (Median)	7
1.3.4 基于卷积的均值滤波原理 (Convolution)	7
1.3.5 非局部均值去噪原理 (NLM)	7
1.4 延伸实验	8
1.4.1 高斯噪声 (Gaussian)	8
1.4.2 泊松噪声 (Poisson)	8
1.4.3 斑点噪声 (Speckle)	8
1.4.4 真实白噪声 (Real White Gaussian)	9
1.4.5 双边滤波	9
1.5 实验步骤	9
1.5.1 数据来源	9
1.5.2 代码编写	10
1.5.3 实验结果	11
1.5.4 实验分析	13
1.6 思考与改进方向	14
1.6.1 算法性能优化	14
1.6.2 参数自适应调整	14
1.6.3 计算效率提升	14
1.6.4 噪声混合场景处理	14
1.6.5 客观评价指标拓展	15
2 实验二：边缘检测	15
2.1 实验目的	15
2.2 实验要求	15
2.3 实验原理	15
2.3.1 Canny 边缘检测算子	15
2.3.2 人工选取阈值	16
2.3.3 迭代法自动阈值选取	16
2.3.4 OTSU 算法 (大津法)	16
2.4 延伸实验	17
2.4.1 形态学开闭运算	17
2.4.2 边缘平滑	17
2.5 实验步骤	18
2.5.1 数据来源	18

2.5.2 代码编写	18
2.5.3 实验结果	20
2.5.4 实验分析	22
2.6 思考与改进方向	22
2.6.1 Canny 边缘检测优化方向	22
2.6.2 阈值分割方法改进策略	23
2.6.3 延伸实验优化思路	23
2.6.4 代码性能与扩展性优化	23
3 实验三：实现一个简单的实用神经网络	23
3.1 实验目的	23
3.2 实验要求	23
3.3 实验原理	24
3.3.1 神经网络基础架构	24
3.3.2 卷积神经网络（CNN）原理	24
3.3.3 全连接层（Fully Connected Layer）	24
3.3.4 反向传播算法	25
3.3.5 损失函数	25
3.4 实验步骤	25
3.4.1 数据来源	25
3.4.2 代码编写	25
3.4.3 实验结果	32
3.4.4 实验分析	32
3.5 思考与改进方向	35
3.5.1 算法性能优化	35
3.5.2 数据增强与处理	36
3.5.3 超参数自适应调整	36
3.5.4 训练效率提升	36
3.5.5 模型评估与验证	36
3.5.6 对多类别分类任务的扩展	36
3.5.7 结论与展望	36
4 实验四：基于卷积神经网络的图像分类	37
4.1 实验目的	37
4.2 实验要求	37
4.3 实验原理	37
4.3.1 卷积神经网络（CNN）简介	37
4.3.2 残差网络（ResNet）原理	38
4.3.3 ResNet18 架构	38
4.3.4 CIFAR100 数据集	38
4.3.5 训练过程与评估	39
4.4 实验步骤	39
4.4.1 数据来源	39

4.4.2 代码编写	39
4.4.3 实验结果	47
4.5 思考与改进方向	51
4.5.1 改进数据增强	51
4.5.2 优化网络结构	51
4.5.3 超参数优化	51
4.5.4 更多评估指标	52
5 实验五：语义分割	52
5.1 实验目的	52
5.2 实验要求	52
5.3 实验原理	52
5.3.1 语义分割概述	52
5.3.2 UNet 模型结构	53
5.3.3 损失函数与优化	53
5.3.4 PASCAL VOC 数据集	54
5.3.5 模型评估	54
5.4 实验步骤	54
5.4.1 数据来源	54
5.4.2 代码编写	54
5.4.3 实验结果	59
5.4.4 实验分析	63
5.5 思考与改进方向	64
5.5.1 数据增强的改进	64
5.5.2 网络结构优化	64
5.5.3 训练时间和优化	64
5.5.4 更多评估指标	64
6 实验六：目标检测	64
6.1 实验目的	64
6.2 实验要求	65
6.3 实验原理	65
6.3.1 YOLOv3 模型概述	65
6.3.2 损失函数	65
6.3.3 PASCAL VOC 数据集	66
6.4 实验步骤	67
6.4.1 数据来源	67
6.4.2 代码编写	67
6.4.3 实验结果	81
6.4.4 实验分析	82
6.5 思考与改进方向	82
6.5.1 数据增强的进一步优化	82
6.5.2 模型架构的优化	83

6.5.3	超参数优化与训练技巧	83
6.5.4	更多评估指标的引入	83
6.5.5	模型训练时间的优化	83

1 实验一：图像去噪

1.1 实验目的

本次实验旨在通过 Python 编程实现彩色图像的椒盐噪声添加及多种去噪算法，深入理解不同去噪技术的原理与应用效果。具体包括掌握椒盐噪声生成方法及其对图像质量的影响，实现基于像素的均值滤波和中值滤波算法并对比两者在椒盐噪声处理中的差异，理解卷积操作在均值滤波中的应用并实现基于卷积的均值滤波，探究非局部均值去噪算法（参考教材 P37）的关键步骤如相似块搜索与权重计算及其在彩色图像去噪中的优势。通过实验对比不同算法在不同参数（如噪声密度、窗口大小）下的处理结果，分析各算法的优缺点，提升图像处理的实践能力与理论认知，为图像恢复技术的深入学习奠定基础。

1.2 实验要求

输入一张彩色图片，使用 Python 实现以下目标：

1. 添加椒盐噪声
2. 去噪：基于像素的均值、中值滤波
3. 去噪：基于卷积的均值滤波
4. 去噪：非局部均值去噪（P37）

1.3 实验原理

1.3.1 椒盐噪声 (Salt Pepper)

椒盐噪声是一种常见的图像噪声，也称为脉冲噪声。它主要由图像传感器的突发干扰或数据传输错误引起。在彩色图像中，通常包含红 (R)、绿 (G)、蓝 (B) 三个通道。添加椒盐噪声的过程可以描述如下：

设原始彩色图像为 $I(x, y)$ ，其中 x 和 y 表示图像的像素坐标， $I(x, y)$ 是一个三维向量，分别对应 R 、 G 、 B 通道的值。噪声密度为 p ，即每个像素有 p 的概率被噪声污染。

添加椒盐噪声后的图像 $I_{noisy}(x, y)$ 可通过以下方式生成：

$$I_{noisy}(x, y) = \begin{cases} (0, 0, 0) & \text{概率为 } \frac{p}{2} \\ (255, 255, 255) & \text{概率为 } \frac{p}{2} \\ I(x, y) & \text{概率为 } 1 - p \end{cases}$$

1.3.2 基于像素的均值滤波原理 (Mean)

均值滤波是一种简单的线性滤波方法，通过计算像素邻域内的均值来平滑图像并去除噪声。对于彩色图像，需要分别对 R 、 G 、 B 三个通道进行处理。

设 $I(x, y)$ 为原始图像， $I_{mean}(x, y)$ 为经过均值滤波后的图像，滤波窗口大小为 $m \times n$ （通常 $m = n$ ，如 3×3 、 5×5 等）。以像素 (x, y) 为中心的邻域内的像素值的均值计算如下：

$$I_{mean}(x, y)_c = \frac{1}{m \times n} \sum_{i=-\lfloor \frac{m}{2} \rfloor}^{\lfloor \frac{m}{2} \rfloor} \sum_{j=-\lfloor \frac{n}{2} \rfloor}^{\lfloor \frac{n}{2} \rfloor} I(x+i, y+j)_c$$

其中 $c = R, G, B$ 表示通道, $\lfloor \cdot \rfloor$ 表示向下取整操作。

1.3.3 基于像素的中值滤波原理 (Median)

中值滤波是一种非线性滤波方法, 它通过将像素邻域内的像素值排序, 取中间值作为该像素的新值, 对椒盐噪声等脉冲噪声有很好的抑制效果。

同样对于彩色图像的每个通道 $c = R, G, B$, 设 $I(x, y)$ 为原始图像, $I_{median}(x, y)$ 为经过中值滤波后的图像, 滤波窗口大小为 $m \times n$ 。以像素 (x, y) 为中心的邻域内的像素值按升序排列后, 取中间值作为新的像素值:

$$I_{median}(x, y)_c = \text{median}\{I(x+i, y+j)_c \mid i = -\lfloor \frac{m}{2} \rfloor, \dots, \lfloor \frac{m}{2} \rfloor; j = -\lfloor \frac{n}{2} \rfloor, \dots, \lfloor \frac{n}{2} \rfloor\}$$

其中 $\text{median}\{\cdot\}$ 表示取中值操作。

1.3.4 基于卷积的均值滤波原理 (Convolution)

基于卷积的均值滤波是利用卷积操作实现的一种均值滤波方法。首先定义一个均值卷积核 K , 其大小为 $m \times n$, 核内所有元素的值都相等, 且满足 $\sum_{i,j} K(i, j) = 1$ 。

对于彩色图像的每个通道 $c = R, G, B$, 设 $I(x, y)$ 为原始图像, $I_{conv_mean}(x, y)$ 为经过基于卷积的均值滤波后的图像, 则:

$$I_{conv_mean}(x, y)_c = \sum_{i=-\lfloor \frac{m}{2} \rfloor}^{\lfloor \frac{m}{2} \rfloor} \sum_{j=-\lfloor \frac{n}{2} \rfloor}^{\lfloor \frac{n}{2} \rfloor} K(i, j) \cdot I(x+i, y+j)_c$$

例如, 对于 3×3 的均值卷积核, K 为:

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

1.3.5 非局部均值去噪原理 (NLM)

非局部均值去噪 (Non-Local Means, NLM) 算法的基本思想是利用图像中存在的大量相似结构信息, 通过对相似图像块进行加权平均来去除噪声。

设 $I(x, y)$ 为原始噪声图像, $I_{NLM}(x, y)$ 为经过非局部均值去噪后的图像。对于图像中的每个像素 (x, y) , 在以其为中心的搜索窗口 $S(x, y)$ 内寻找与以 (x, y) 为中心的图像块相似的其他图像块。

相似性度量通常使用欧氏距离, 设以 (x, y) 为中心的图像块为 $B(x, y)$, 以 (u, v) 为中心的图像块为 $B(u, v)$, 则它们之间的相似性权重 $w(x, y, u, v)$ 计算如下:

$$w(x, y, u, v) = \frac{1}{Z(x, y)} \exp \left(-\frac{\|B(x, y) - B(u, v)\|_2^2}{h^2} \right)$$

其中 $Z(x, y)$ 是归一化因子, 确保所有权重之和为 1, 即 $Z(x, y) = \sum_{(u,v) \in S(x,y)} \exp\left(-\frac{\|B(x,y)-B(u,v)\|_2^2}{h^2}\right)$, h 是控制相似性权重衰减的参数, $\|\cdot\|_2$ 表示欧氏范数。

去噪后的像素值 $I_{NLM}(x, y)$ 为搜索窗口内所有像素值的加权平均:

$$I_{NLM}(x, y) = \sum_{(u,v) \in S(x,y)} w(x, y, u, v) \cdot I(u, v)$$

1.4 延伸实验

在本次实验中, 我还增加了不同的噪声, 并增加了适用于彩色图像的双边滤波, 以综合评估各个滤波器的性能。

1.4.1 高斯噪声 (Gaussian)

高斯噪声是一种具有正态分布概率密度函数的噪声, 其产生原因通常与图像传感器的电子元件、电路噪声等有关。在彩色图像中, 高斯噪声独立作用于红 (R)、绿 (G)、蓝 (B) 三个通道。其概率密度函数为:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

其中, μ 是均值, σ^2 是方差。对于图像 $I(x, y)$, 添加高斯噪声后的图像 $I_{gauss}(x, y)$ 可表示为:

$$I_{gauss}(x, y)_c = I(x, y)_c + N(x, y)_c, \quad c = R, G, B$$

其中 $N(x, y)_c$ 是服从高斯分布 $N(\mu, \sigma^2)$ 的随机噪声值。

1.4.2 泊松噪声 (Poisson)

泊松噪声通常出现在低光照条件下的图像采集过程中, 与光子的统计特性相关。Poisson 分布的概率质量函数为:

$$P(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

其中 k 是事件发生的次数, λ 是单位时间或单位面积内事件发生的平均次数。在图像中, 像素值 $I(x, y)$ 可视为光子到达探测器的平均数量, 添加 Poisson 噪声后的图像 $I_{poisson}(x, y)$ 的像素值 $I_{poisson}(x, y)_c$ 服从参数为 $I(x, y)_c$ 的 Poisson 分布, 即:

$$I_{poisson}(x, y)_c \sim Poisson(I(x, y)_c), \quad c = R, G, B$$

1.4.3 斑点噪声 (Speckle)

斑点噪声常见于合成孔径雷达 (SAR) 图像和超声图像中, 其本质是相干成像系统中大量微小散射体的回波信号相互干涉的结果。Speckle 噪声与图像信号是乘性的关系, 对于图像 $I(x, y)$, 添加 Speckle 噪声后的图像 $I_{speckle}(x, y)$ 可表示为:

$$I_{speckle}(x, y)_c = I(x, y)_c \cdot S(x, y)_c, \quad c = R, G, B$$

其中 $S(x, y)_c$ 是均值为 1 的随机噪声, 通常服从 Gamma 分布或对数正态分布。

1.4.4 真实白噪声 (Real White Gaussian)

白噪声是一种功率谱密度在整个频域内均匀分布的噪声，其自相关函数满足：

$$R_{xx}(\tau) = \sigma^2 \delta(\tau)$$

其中 σ^2 是噪声方差， $\delta(\tau)$ 是狄拉克函数。在实际图像中，白噪声的添加方式与高斯噪声类似，也是对图像的每个像素值叠加一个随机噪声值。假设 $N(x, y)$ 是白噪声，添加白噪声后的图像 $I_{white}(x, y)$ 为：

$$I_{white}(x, y)_c = I(x, y)_c + N(x, y)_c, \quad c = R, G, B$$

1.4.5 双边滤波

双边滤波是一种非线性的滤波方法，它结合了空间邻近度和像素值相似度的双重加权，在去除噪声的同时能够较好地保留图像边缘。对于图像 $I(x, y)$ ，双边滤波后的图像 $I_{bilateral}(x, y)$ 的像素值计算如下：

$$I_{bilateral}(x, y)_c = \frac{\sum_{i=-\lfloor \frac{m}{2} \rfloor}^{\lfloor \frac{m}{2} \rfloor} \sum_{j=-\lfloor \frac{n}{2} \rfloor}^{\lfloor \frac{n}{2} \rfloor} f_s(i, j) f_r(I(x+i, y+j)_c, I(x, y)_c) I(x+i, y+j)_c}{\sum_{i=-\lfloor \frac{m}{2} \rfloor}^{\lfloor \frac{m}{2} \rfloor} \sum_{j=-\lfloor \frac{n}{2} \rfloor}^{\lfloor \frac{n}{2} \rfloor} f_s(i, j) f_r(I(x+i, y+j)_c, I(x, y)_c)}, \quad c = R, G, B$$

其中 $m \times n$ 是滤波窗口大小， $f_s(i, j)$ 是空间邻近度权重，通常使用高斯函数表示：

$$f_s(i, j) = \exp \left(-\frac{i^2 + j^2}{2\sigma_s^2} \right)$$

$f_r(I(x+i, y+j)_c, I(x, y)_c)$ 是像素值相似度权重，同样使用高斯函数：

$$f_r(I(x+i, y+j)_c, I(x, y)_c) = \exp \left(-\frac{(I(x+i, y+j)_c - I(x, y)_c)^2}{2\sigma_r^2} \right)$$

σ_s 控制空间距离对权重的影响， σ_r 控制像素值差异对权重的影响。通过调整这两个参数，可以平衡去噪效果和边缘保留能力。

1.5 实验步骤

1.5.1 数据来源

本实验用于滤波去噪的彩色图像数据来源于 PolyU 数据集 [1]。该数据集由 Xu 等人构建，旨在为真实世界图像去噪研究提供更具挑战性的基准。不同于传统聚焦于加性高斯白噪声 (AWGN) 的数据集，PolyU 数据集采集了不同相机在多样拍摄设置下的自然场景图像，涵盖了复杂光照、纹理及实际拍摄产生的各类噪声，更贴合现实应用中的图像情况。

在构建过程中，为获取可靠的“ground truth”图像，研究者对同一静态场景进行大量重复拍摄（约 500 次），通过像素均值计算，有效抑制随机噪声干扰，使“Ground Truth”能精准反映真实场景信息。

本次实验选用的 Bag 样本 [2]，包含丰富的纹理细节与复杂的明暗变化，自带的真实噪声特性，可有效测试各类滤波去噪算法在复杂场景下的噪声抑制能力、边缘保留效果以及算法鲁棒性。

1.5.2 代码编写

本实验使用 Python 语言结合相关的图像处理库进行代码编写, 主要使用的库有 OpenCV、NumPy、matplotlib 和 scikit-image。以下是实验代码:

```
1 import cv2
2 import numpy as np
3 import os
4 from skimage.util import random_noise
5 from skimage.restoration import denoise_nl_means, estimate_sigma
6 from skimage.metrics import peak_signal_noise_ratio as psnr
7 from skimage.metrics import structural_similarity as ssim
8
9 def evaluate_metrics(denoised, reference, name):
10     p = psnr(reference, denoised, data_range=255)
11     s = ssim(reference, denoised, channel_axis=-1, data_range=255)
12     print(f"[{name}] PSNR: {p:.2f} dB | SSIM: {s:.4f}")
13
14
15 def apply_denoising_methods(noisy_img, noise_name, clean_img):
16     save_dir = f'./Result/{noise_name}'
17     os.makedirs(save_dir, exist_ok=True)
18
19     # 带噪图像
20     cv2.imwrite(f'{save_dir}/noisy.jpg', cv2.cvtColor(noisy_img, cv2.COLOR_RGB2BGR))
21     evaluate_metrics(noisy_img, clean_img, f'{noise_name} - Noisy Image')
22
23     # 均值滤波
24     mean_filtered = cv2.blur(noisy_img, (3, 3))
25     cv2.imwrite(f'{save_dir}/mean_filtered.jpg', cv2.cvtColor(mean_filtered, cv2.COLOR_RGB2BGR))
26     evaluate_metrics(mean_filtered, clean_img, f'{noise_name} - Mean Filter')
27
28     # 中值滤波
29     median_filtered = cv2.medianBlur(noisy_img, 3)
30     cv2.imwrite(f'{save_dir}/median_filtered.jpg', cv2.cvtColor(median_filtered, cv2.COLOR_RGB2BGR))
31     evaluate_metrics(median_filtered, clean_img, f'{noise_name} - Median Filter')
32
33     # 卷积实现均值滤波
34     kernel = np.ones((3, 3), np.float32) / 9
35     conv_filtered = cv2.filter2D(noisy_img, -1, kernel)
36     cv2.imwrite(f'{save_dir}/conv_filtered.jpg', cv2.cvtColor(conv_filtered, cv2.COLOR_RGB2BGR))
37     evaluate_metrics(conv_filtered, clean_img, f'{noise_name} - Conv Mean Filter')
38
39     # 双边滤波
40     bilateral_filtered = cv2.bilateralFilter(noisy_img, d=9, sigmaColor=75, sigmaSpace=75)
41     cv2.imwrite(f'{save_dir}/bilateral_filtered.jpg', cv2.cvtColor(bilateral_filtered, cv2.COLOR_RGB2BGR))
42     evaluate_metrics(bilateral_filtered, clean_img, f'{noise_name} - Bilateral Filter')
43
44     # 非局部均值滤波
45     noisy_img_float = noisy_img.astype(np.float32) / 255.0
46     sigma_est = np.mean(estimate_sigma(noisy_img_float, channel_axis=-1))
47     nlm_denoised = denoise_nl_means(
48         noisy_img_float,
49         h=1.15 * sigma_est,
50         fast_mode=True,
51         patch_size=5,
```

```

52     patch_distance=7,
53     channel_axis=-1
54 )
55 nlm_denoised = (nlm_denoised * 255).astype(np.uint8)
56 cv2.imwrite(f'{save_dir}/nlm_denoised.jpg', cv2.cvtColor(nlm_denoised, cv2.COLOR_RGB2BGR))
57 evaluate_metrics(nlm_denoised, clean_img, f'{noise_name} - NLM Filter')
58
59 # 1. 加载图像
60 img = cv2.imread('./Data/Canon5D2_bag_mean.JPG')
61 img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
62
63 # 2. 定义噪声种类
64 noise_types = {
65     'salt_pepper': lambda img: random_noise(img, mode='s&p', amount=0.05),
66     'gaussian': lambda img: random_noise(img, mode='gaussian', var=0.01),
67     'poisson': lambda img: random_noise(img, mode='poisson'),
68     'speckle': lambda img: random_noise(img, mode='speckle', var=0.05),
69 }
70
71 # 3. 应用各种滤波器
72 for noise_name, noise_func in noise_types.items():
73     noisy = noise_func(img_rgb)
74     noisy = (noisy * 255).astype(np.uint8)
75     apply_denoising_methods(noisy, noise_name, img_rgb)
76
77 # 4. 真实白噪声测试
78 real_noise_path = './Data/Canon5D2_bag_Real.JPG'
79 real_noisy = cv2.imread(real_noise_path)
80 real_noisy_rgb = cv2.cvtColor(real_noisy, cv2.COLOR_BGR2RGB)
81
82 apply_denoising_methods(real_noisy_rgb, 'real_white_noise', img_rgb)

```

1.5.3 实验结果

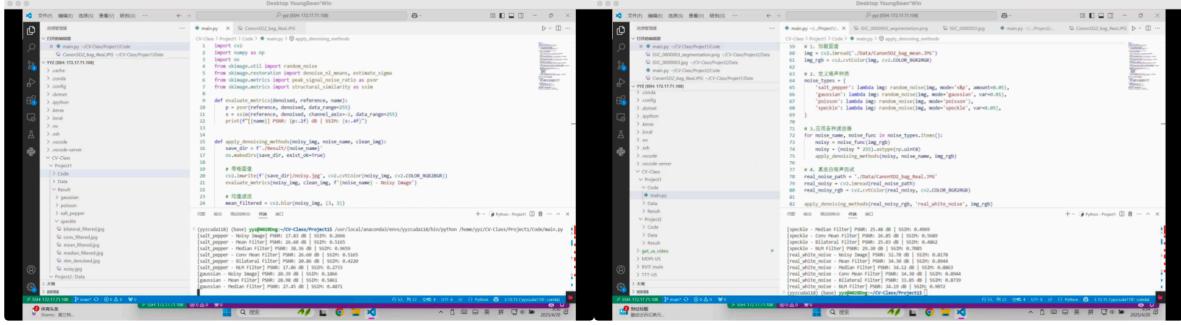
将原始彩色图像添加椒盐噪声后，分别使用上述四种去噪方法进行处理，得到相应的去噪结果图像。通过 matplotlib 库将原始图像、噪声图像和去噪后的图像进行可视化展示，方便直观对比不同去噪算法的效果。可视化效果可见图1，其中对应的标签可见图2。同时，计算去噪后图像与原始图像之间的峰值信噪比（PSNR）和结构相似性指数（SSIM）等客观评价指标，以量化评估去噪效果。具体数据可见表2。其中实验过程图可见图3。



图 1: 各种滤波器对应不同噪声的效果对比图。



图 2: 该实验中使用的图像对应的地表真值。



实验过程图

图 3: 实验过程图。

表 1: 各种滤波器对应不同噪声的效果定性分析。其中加粗的为表现最好的。

Method	Salt Pepper		Gaussian		Poisson		Speckle		Real Noise	
	PSNR	SSIM								
Noisy Image	17.83	0.2666	20.39	0.1866	27.15	0.5086	18.52	0.2844	32.78	0.8178
Mean Filter	26.60	0.5165	28.98	0.5861	33.98	0.8381	26.85	0.5689	34.30	0.8944
Median Filter	38.36	0.9659	27.45	0.4871	33.09	0.7897	25.48	0.4969	34.12	0.8863
Conv Filter	26.60	0.5165	28.98	0.5861	33.98	0.8381	26.85	0.5689	34.30	0.8944
Bilateral Filter	20.86	0.4220	29.91	0.6391	34.14	0.8648	25.03	0.4862	33.85	0.8739
NLM Filter	17.86	0.2733	31.89	0.8344	35.34	0.8958	29.30	0.7085	34.19	0.9072

1.5.4 实验分析

本实验对添加不同类型噪声（椒盐噪声、高斯噪声、泊松噪声、斑点噪声、真实白噪声）的图像，分别使用均值滤波、中值滤波、卷积均值滤波、双边滤波和非局部均值滤波（NLM）进行去噪处理，并通过峰值信噪比（PSNR）和结构相似性指数（SSIM）量化评估去噪效果。

椒盐噪声处理效果: 添加椒盐噪声后，原始图像 PSNR 仅为 17.83 dB，SSIM 为 0.2666，噪声严重破坏图像质量。在去噪算法中，中值滤波表现最为突出，PSNR 提升至 38.36 dB，SSIM 达到 0.9659，说明其有效去除脉冲噪声的同时最大程度保留了图像结构；均值滤波与卷积均值滤波性能相近（PSNR 26.60 dB，SSIM 0.5165），虽能降低噪声，但模糊效应明显；双边滤波 PSNR 为 20.86 dB，SSIM 为 0.4220，去噪效果有限；NLM 滤波 PSNR 仅提升至 17.86 dB，SSIM 为 0.2733，甚至略低于原始噪声图像，表明该算法对椒盐噪声的抑制能力较弱。

高斯噪声处理效果: 对于高斯噪声，NLM 滤波展现出显著优势，PSNR 从 20.39 dB 提升至 31.89 dB，SSIM 从 0.1866 提升至 0.8344，说明其能有效平滑噪声并保留图像细节；双边滤波次之（PSNR=29.91 dB，SSIM=0.6391），利用像素相似性和空间距离加权，平衡去噪与边缘保护；均值滤波和卷积均值滤波 PSNR 均为 28.98 dB，SSIM 约 0.5861，表现稍逊；中值滤波 PSNR 仅 27.45 dB，SSIM 为 0.4871，因其中非线性特性对高斯噪声的抑制能力弱于线性滤波方法。

泊松噪声处理效果: 泊松噪声场景下，NLM 滤波再次表现优异，PSNR 从 27.15 dB 提升至 35.34 dB，SSIM 从 0.5086 提升至 0.8958；双边滤波（PSNR=34.14 dB，SSIM=0.8648）和均值/卷积均值滤波（PSNR 33.98 dB，SSIM 0.8381）性能接近，均能有效去除噪声；中值滤波 PSNR 为 33.09 dB，SSIM 为 0.7897，略低于其他方法，表明其对泊松噪声的处理能力相对有限。

斑点噪声处理效果: 针对斑点噪声, NLM 滤波 PSNR 从 18.52 dB 提升至 29.30 dB, SSIM 从 0.2844 提升至 0.7085, 显著优于其他算法; 均值滤波和卷积均值滤波 PSNR 为 26.85 dB, SSIM 约 0.5689, 去噪效果中等; 中值滤波和双边滤波 PSNR 分别为 25.48 dB 和 25.03 dB, SSIM 均低于 0.5, 说明这两种方法在抑制斑点噪声时难以有效保留图像细节。

真实白噪声处理效果: 对于真实白噪声, 各算法性能差异较小。NLM 滤波 PSNR 从 32.78 dB 提升至 34.19 dB, SSIM 从 0.8178 提升至 0.9072, 表现最佳; 均值滤波和卷积均值滤波 PSNR 为 34.30 dB, SSIM 为 0.8944, 略高于 NLM 滤波, 但在 SSIM 指标上仍低于 NLM; 中值滤波 PSNR 为 34.12 dB, SSIM 为 0.8863; 双边滤波 PSNR 为 33.85 dB, SSIM 为 0.8739, 二者去噪效果相近且稍逊于前三者。

实验数据表明, 中值滤波对椒盐噪声具有极强的针对性, 而 NLM 滤波在高斯、泊松、斑点及真实白噪声处理中均展现优异性能, 尤其在保留图像结构和细节方面优势明显; 均值滤波和卷积均值滤波适用于平滑高斯、泊松等分布噪声, 但存在模糊效应; 双边滤波在噪声与细节平衡上表现良好, 但对椒盐和斑点噪声效果有限。实际应用中, 需根据噪声类型选择最优去噪算法, 以实现最佳图像恢复效果。

1.6 思考与改进方向

通过本次实验对多种噪声类型及去噪算法的研究, 可从以下方面进行思考与改进:

1.6.1 算法性能优化

实验中部分算法在处理特定噪声时存在局限性。例如, 均值滤波在去除椒盐噪声时会导致图像模糊, NLM 滤波处理椒盐噪声效果欠佳。未来可探索改进传统算法, 如设计自适应权重的均值滤波, 依据像素邻域特征动态调整权重, 在抑制噪声的同时减少模糊; 对于 NLM 算法, 可优化相似块搜索策略, 提升其对椒盐噪声等脉冲噪声的处理能力。此外, 尝试将不同算法融合, 如结合中值滤波与双边滤波的优势, 先利用中值滤波去除椒盐噪声, 再通过双边滤波进一步平滑图像并保留边缘。

1.6.2 参数自适应调整

无论是滤波窗口大小, 还是非局部均值去噪中的搜索窗口、块大小等参数, 均对去噪效果有显著影响。目前实验采用固定参数设置, 难以适应不同噪声强度和图像内容。后续可研究基于图像特征的参数自适应调整方法, 例如通过分析图像的纹理复杂度、噪声密度等信息, 自动选择最优的滤波参数, 提升算法的通用性和鲁棒性。

1.6.3 计算效率提升

非局部均值去噪等算法虽然去噪效果较好, 但计算复杂度高, 处理时间长。可探索并行计算、GPU 加速等技术, 利用硬件资源提高算法执行效率; 也可研究对算法进行简化和近似计算, 在保证一定去噪效果的前提下, 降低计算开销, 以满足实时性要求较高的应用场景。

1.6.4 噪声混合场景处理

实际图像中的噪声往往是多种类型混合的, 而本次实验仅针对单一类型噪声进行处理。未来可研究针对混合噪声的去噪方法, 分析不同噪声的特性, 设计分步或联合去噪策略, 实现对复杂噪声

环境下图像的有效恢复。

1.6.5 客观评价指标拓展

实验主要采用 PSNR 和 SSIM 作为客观评价指标，然而这些指标有时与人类视觉感知存在差异。可引入更多与人眼视觉特性相关的评价指标，如多尺度结构相似性 (MS-SSIM)、视觉信息保真度 (VIF) 等，更全面、准确地评估去噪算法的性能，为算法优化提供更合理的指导。

2 实验二：边缘检测

2.1 实验目的

通过 Python 编程实现 Canny 边缘检测算子，完整复现高斯滤波降噪、Sobel 梯度计算、非极大值抑制及双阈值边缘连接的流程，深入理解各步骤的数学原理（如高斯核的空域平滑特性、梯度幅值与方向的计算方法及非极大值抑制对边缘细化的作用），掌握高低阈值参数对边缘检测结果（如边缘连续性、噪声敏感度）的影响规律。同时，针对输入图像实现基于阈值的二值化分割，包括人工经验阈值设定、基于前景 - 背景均值差的迭代法自动阈值求解以及基于类间方差最大化的 OTSU 算法，对比分析不同方法在图像前景 - 背景分离中的效果差异，理解人工阈值的经验性局限、迭代法的收敛条件及 OTSU 算法在双峰直方图场景中的高效分割优势。此外，通过整合图像灰度转换、噪声预处理、边缘检测与阈值分割的完整流程，熟练运用 OpenCV 和 Matplotlib 等工具进行图像处理与可视化，提升从算法原理到实践应用的综合能力，为复杂图像分析任务奠定技术基础。

2.2 实验要求

1. 用 python 实现 canny 边缘检测算子
2. 输入一张图片，实现基于阈值的二值化分割（测试人工选取阈值、迭代法、OTSU 算法等阈值选取方法）。

2.3 实验原理

2.3.1 Canny 边缘检测算子

Canny 边缘检测是一种多阶段的边缘检测算法，其目的是在噪声存在的情况下准确地检测出图像中的边缘，主要包含以下四个关键步骤：

高斯滤波：高斯滤波用于平滑图像，减少噪声对边缘检测的影响。二维高斯函数定义为：

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

其中， (x, y) 是空间坐标， σ 是高斯分布的标准差，控制着平滑的程度。通过将图像与高斯核进行卷积运算，得到平滑后的图像。

梯度计算：使用 Sobel 算子计算图像在 x 和 y 方向上的梯度 G_x 和 G_y 。Sobel 算子在 x 和 y

方向的模板分别为：

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

梯度幅值 G 和方向 θ 可由以下公式计算：

$$G = \sqrt{G_x^2 + G_y^2}, \quad \theta = \arctan\left(\frac{G_y}{G_x}\right)$$

非极大值抑制：非极大值抑制用于细化边缘，使得最终检测到的边缘为单像素宽。对于每个像素点，将其梯度方向 θ 近似到四个方向之一 (0° 、 45° 、 90° 、 135°)，然后比较该像素点在梯度方向上与其相邻像素的梯度幅值，如果该像素点的梯度幅值不是局部最大，则将其值设为 0。

双阈值检测与边缘连接：设定两个阈值：低阈值 T_{low} 和高阈值 T_{high} 。梯度幅值大于 T_{high} 的像素点被认为是强边缘点；梯度幅值小于 T_{low} 的像素点被认为是弱边缘点；梯度幅值介于 T_{low} 和 T_{high} 之间的像素点，如果与强边缘点相连，则被认为是边缘点，否则被认为是弱边缘点。

2.3.2 人工选取阈值

人工选取阈值是一种简单直接的图像二值化方法。用户根据图像的灰度分布和自身经验，手动选择一个合适的阈值 T 。对于图像中的每个像素 $f(x, y)$ ，按照以下规则进行二值化：

$$B(x, y) = \begin{cases} 255, & f(x, y) \geq T \\ 0, & f(x, y) < T \end{cases}$$

这种方法的优点是简单易行，适用于图像灰度分布较为简单、用户对图像内容有一定了解的情况。缺点是需要人工干预，且对于复杂图像，很难选择到合适的阈值。

2.3.3 迭代法自动阈值选取

迭代法是一种自动确定阈值的方法，其基本思想是通过迭代不断更新阈值，直到满足收敛条件。具体步骤如下：

初始化：选择一个初始阈值 T_0 ，通常可以取图像灰度值的中间值，即 $T_0 = \frac{\min(f) + \max(f)}{2}$ ，其中 $\min(f)$ 和 $\max(f)$ 分别是图像中的最小和最大灰度值。

分类与均值计算：根据当前阈值 T_k 将图像分为两类：前景（灰度值大于等于 T_k ）和背景（灰度值小于 T_k ），分别计算这两类的平均灰度值 m_1 和 m_2 。

阈值更新：计算新的阈值 $T_{k+1} = \frac{m_1 + m_2}{2}$ 。

收敛判断：如果 $|T_{k+1} - T_k| < \epsilon$ (ϵ 是一个预先设定的小正数)，则认为算法收敛， T_{k+1} 即为最终的阈值；否则，令 $k = k + 1$ ，返回步骤 2 继续迭代。

2.3.4 OTSU 算法（大津法）

OTSU 算法是一种基于图像灰度直方图的自动阈值选取方法，其目标是找到一个阈值 T ，使得前景和背景之间的类间方差最大。

假设图像的灰度级范围是 0 到 $L-1$ ，灰度值为 i 的像素的概率为 p_i ，且 $\sum_{i=0}^{L-1} p_i = 1$ 。对于一个阈值 t ，定义以下参数：前景像素的概率： $\omega_1(t) = \sum_{i=0}^t p_i$ 背景像素的概率： $\omega_2(t) = 1 - \omega_1(t) = \sum_{i=t+1}^{L-1} p_i$

前景像素的平均灰度: $m_1(t) = \frac{\sum_{i=0}^t i \cdot p_i}{\omega_1(t)}$ 背景像素的平均灰度: $m_2(t) = \frac{\sum_{i=t+1}^{L-1} i \cdot p_i}{\omega_2(t)}$ 全局平均灰度: $m_G = \sum_{i=0}^{L-1} i \cdot p_i$
 类间方差 $\sigma_B^2(t)$ 定义为:

$$\sigma_B^2(t) = \omega_1(t)\omega_2(t)[m_1(t) - m_2(t)]^2 = \omega_1(t)(m_1(t) - m_G)^2 + \omega_2(t)(m_2(t) - m_G)^2$$

OTSU 算法通过遍历所有可能的阈值 $t \in [0, L - 1]$, 找到使 $\sigma_B^2(t)$ 最大的阈值 t^* , 即:

$$t^* = \arg \max_{0 \leq t \leq L-1} \sigma_B^2(t)$$

这个阈值 t^* 就是 OTSU 算法确定的最优阈值。

2.4 延伸实验

此次实验我主要考虑了在 OTSU 的基础上增加后处理。

2.4.1 形态学开闭运算

形态学运算基于数学形态学, 通过结构元素对图像进行处理, 在 OTSU 二值化结果的基础上应用形态学开闭运算, 能够有效优化分割效果。其核心运算包括腐蚀与膨胀, 二者是形态学开闭运算的基础。

腐蚀操作是将结构元素 B 在图像 A 上移动, 用结构元素覆盖区域内的最小像素值替换中心像素值, 数学表达式为:

$$(A \ominus B)(x, y) = \min\{A(x + i, y + j) \mid (i, j) \in B\}$$

其中, (x, y) 为图像 A 中的像素坐标, (i, j) 为结构元素 B 中的坐标。腐蚀操作可以去除图像中小于结构元素的孤立噪声块和毛刺, 使物体边界向内收缩。

膨胀操作则是用结构元素覆盖区域内的最大像素值替换中心像素值, 其数学表达式为:

$$(A \oplus B)(x, y) = \max\{A(x + i, y + j) \mid (i, j) \in B\}$$

膨胀操作能够填充物体内部的小空洞, 连接邻近的区域, 使物体边界向外扩张。

开运算先腐蚀后膨胀, 即 $A \circ B = (A \ominus B) \oplus B$ 。在 OTSU 二值化后的图像上应用开运算, 可有效去除图像中的小噪声颗粒, 平滑物体边界。例如, 对于二值化后残留的椒盐噪声点, 开运算能够将其消除。

闭运算先膨胀后腐蚀, 即 $A \bullet B = (A \oplus B) \ominus B$ 。在 OTSU 二值化图像上使用闭运算, 可以填充物体内部因噪声或图像纹理导致的小空洞, 连接断裂的边缘, 使分割后的物体轮廓更加完整。

结构元素的形状和大小对形态学运算结果影响显著。常见的结构元素形状有正方形、矩形、圆形等, 其大小通常根据图像中噪声块的尺寸和目标物体的细节进行选择。例如, 对于噪声点较小且目标边缘细节丰富的图像, 可选择 3×3 的小型结构元素; 对于噪声块较大或目标物体轮廓较为粗犷的图像, 则可选用 5×5 或更大的结构元素。

2.4.2 边缘平滑

在 OTSU 算法基础上, 结合高斯滤波与阈值重建方法, 能够进一步优化图像分割效果, 其核心在于通过预处理平滑图像和后处理重建边缘来提升分割精度。

高斯滤波是通过二维高斯函数对图像进行卷积运算，以实现图像平滑和噪声抑制的目的。二维高斯函数表达式为：

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

其中， (x, y) 为像素坐标， σ 为高斯分布的标准差，控制着滤波的平滑程度。 σ 越大，图像平滑效果越强，但同时也会导致边缘细节损失更多； σ 越小，噪声抑制能力相对较弱，但边缘保持较好。对原始图像 I 进行高斯滤波后，得到平滑后的图像 I_{blur} ，即 $I_{blur} = I * G(\sigma)$ ，其中“*”表示卷积运算。通过高斯滤波，能够有效减少图像中的高斯噪声、椒盐噪声等，降低噪声对后续 OTSU 阈值计算的干扰。

在得到平滑后的图像 I_{blur} 后，使用 OTSU 算法计算阈值 T 并进行二值化分割，得到二值化图像 B 。然而，由于高斯滤波会使图像边缘变得模糊，导致二值化后的边缘可能存在不连续或不准确的情况，因此需要通过阈值重建进行优化。

阈值重建通常基于形态学重建操作。形态学重建是一种基于标记图像 M 和掩模图像 Y 的迭代运算。标记图像 M 通常由对二值化图像 B 进行腐蚀等操作得到，其作用是确定需要重建的起始区域；掩模图像 Y 则为二值化图像 B ，用于限制重建的范围。重建过程通过形态学膨胀操作迭代进行，其迭代公式为：

$$R_{k+1}(Y) = (R_k(Y) \oplus B) \cap Y$$

其中， $R_k(Y)$ 为第 k 次迭代后的重建结果， $R_0(Y) = M$ 。在每次迭代中，先对当前重建结果进行膨胀操作，然后与掩模图像取交集，确保重建结果不会超出掩模图像的范围。通过多次迭代，能够细化边缘，增强边缘的连续性，恢复因高斯滤波和二值化操作导致的边缘损失，从而得到更准确、完整的图像分割结果。

2.5 实验步骤

2.5.1 数据来源

本实验使用的彩色图像数据来自于 ISIC 数据集 [3, 4]。ISIC (International Skin Imaging Collaboration) 数据集由国际皮肤成像协作组织发布，是皮肤病变分割领域的重要资源，其包含多个版本。数据集主要聚焦黑色素瘤等各类皮肤病变，标注信息区分癌症与非癌症，图像在颜色、纹理上呈现高度多样性，模拟临床真实场景下的复杂皮肤病变情况，为图像边缘检测与分割算法研究提供了充满挑战性的测试样本，有力推动皮肤病变图像分析技术的发展。

本次实验选取的是 2018 年挑战中的训练第三个样本 [5]。该样本颜色差异较大，相较于其他数据更易于凸显算法处理效果，能够有效衡量 Canny 边缘检测、阈值分割等算法在不同色彩和纹理特征下的性能表现。

2.5.2 代码编写

本实验使用利用 Python 结合 OpenCV 库实现图像的边缘检测与分割处理。首先进行 Canny 边缘检测，借助 OpenCV 库调用 ‘cv2.Canny()’ 函数，设置低阈值为 50、高阈值为 150，对灰度化的图像进行边缘提取，并使用 ‘cv2.imwrite()’ 函数保存边缘检测结果图像。

对于阈值分割，采用了三种方法。人工选取阈值法手动设定阈值为 100，调用 ‘cv2.threshold()’ 函数实现图像二值化以分离前景与背景；迭代法基于图像像素灰度分布，通过迭代计算前景与背景的

均值差自动求解最优阈值，得到的阈值为 $T = 157.22$ ；OTSU 算法则调用 ‘cv2.threshold()’ 函数并采用 ‘cv2.THRESH_OTSU’ 模式，自动计算阈值完成图像二值化分割。

延伸实验部分，形态学开闭运算在 OTSU 二值化结果基础上，使用 ‘cv2.morphologyEx()’ 函数，选用 7×7 的椭圆形结构元素，先执行开运算去除噪点，再执行闭运算填充孔洞，优化图像分割后的边缘连续性并去除噪声。高斯滤波与阈值重建操作，先使用 ‘cv2.GaussianBlur()’ 函数对 OTSU 二值化图像进行平滑处理，设置核大小为 5×5 、标准差为 0，之后通过 ‘cv2.threshold()’ 函数结合阈值 127 再次进行二值化。

以下是实验代码：

```
1 import cv2
2 import numpy as np
3 import os
4
5 image_path = './Data/ISIC_0000003.jpg'
6 gt_path = './Data/ISIC_0000003_segmentation.png'
7
8 img = cv2.imread(image_path)
9 gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
10 gt = cv2.imread(gt_path, 0)
11 _, gt_bin = cv2.threshold(gt, 127, 255, cv2.THRESH_BINARY)
12
13 # 1. Canny 边缘检测
14 def canny_edge_detection(gray_img, low_threshold=50, high_threshold=150):
15     edges = cv2.Canny(gray_img, low_threshold, high_threshold)
16     return edges
17
18 edges = canny_edge_detection(gray)
19 cv2.imwrite('./Result/canny_edges.png', edges)
20
21 # 2. 二值化分割方法
22 def manual_threshold(gray_img, threshold=100):
23     _, binary = cv2.threshold(gray_img, threshold, 255, cv2.THRESH_BINARY)
24     binary = cv2.bitwise_not(binary)
25     return binary
26
27 def iterative_threshold(gray_img, epsilon=1):
28     prev_T = gray_img.mean()
29     while True:
30         G1 = gray_img[gray_img > prev_T]
31         G2 = gray_img[gray_img <= prev_T]
32         if len(G1) == 0 or len(G2) == 0:
33             break
34         T = 0.5 * (G1.mean() + G2.mean())
35         if abs(T - prev_T) < epsilon:
36             break
37         prev_T = T
38     _, binary = cv2.threshold(gray_img, T, 255, cv2.THRESH_BINARY)
39     binary = cv2.bitwise_not(binary)
40     return binary, T
41
42 def otsu_threshold(gray_img):
43     _, binary = cv2.threshold(gray_img, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
44     binary = cv2.bitwise_not(binary)
45     return binary
46
```

```

47 # 3. 评估 IoU 和 Dice
48 def compute_metrics(pred_mask, gt_mask):
49     pred = (pred_mask == 255).astype(np.uint8)
50     gt = (gt_mask == 255).astype(np.uint8)
51     intersection = np.logical_and(pred, gt).sum()
52     union = np.logical_or(pred, gt).sum()
53     iou = intersection / union if union != 0 else 0
54     dice = 2 * intersection / (pred.sum() + gt.sum()) if (pred.sum() + gt.sum()) != 0 else 0
55     return iou, dice
56
57 # Manual
58 manual_bin = manual_threshold(gray, threshold=100)
59 cv2.imwrite('./Result/manual_threshold_100.png', manual_bin)
60 iou_manual, dice_manual = compute_metrics(manual_bin, gt_bin)
61
62 # Iterative
63 iter_bin, iter_T = iterative_threshold(gray)
64 cv2.imwrite(f'./Result/iterative_threshold_{int(iter_T)}.png', iter_bin)
65 iou_iter, dice_iter = compute_metrics(iter_bin, gt_bin)
66
67 # OTSU
68 otsu_bin = otsu_threshold(gray)
69 cv2.imwrite('./Result/otsu_threshold.png', otsu_bin)
70 iou_otsu, dice_otsu = compute_metrics(otsu_bin, gt_bin)
71
72 # 形态学处理+(OTSU)
73 kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7, 7))
74 otsu_morph = cv2.morphologyEx(otsu_bin, cv2.MORPH_OPEN, kernel) # 开运算去噪点
75 otsu_morph = cv2.morphologyEx(otsu_morph, cv2.MORPH_CLOSE, kernel) # 闭运算填孔洞
76 cv2.imwrite('./Result/otsu_morph_open_close.png', otsu_morph)
77 iou_otsu_morph, dice_otsu_morph = compute_metrics(otsu_morph, gt_bin)
78
79 # 边缘平滑+(OTSU)
80 blurred = cv2.GaussianBlur(otsu_bin, (5, 5), 0)
81 _, smooth_mask = cv2.threshold(blurred, 127, 255, cv2.THRESH_BINARY)
82 cv2.imwrite('./Result/otsu_morph_smooth.png', smooth_mask)
83 iou_otsu_smooth, dice_otsu_smooth = compute_metrics(smooth_mask, gt_bin)
84
85
86 print(f"Manual - IoU: {iou_manual:.4f}, Dice: {dice_manual:.4f}")
87 print(f"Iterative - IoU: {iou_iter:.4f}, Dice: {dice_iter:.4f} (T={iter_T:.2f})")
88 print(f"OTSU - IoU: {iou_otsu:.4f}, Dice: {dice_otsu:.4f}")
89 print(f"OTSU+Morph - IoU: {iou_otsu_morph:.4f}, Dice: {dice_otsu_morph:.4f}")
90 print(f"OTSU+Smooth - IoU: {iou_otsu_smooth:.4f}, Dice: {dice_otsu_smooth:.4f}")

```

2.5.3 实验结果

实验输出原始图像、Canny 边缘检测结果、各阈值分割图像（人工选取、迭代法、OTSU 算法）以及延伸实验处理后的图像（OTSU + 形态学、OTSU + 高斯滤波重建），其定性比较可见图4。同时，实验计算了各分割结果的交并比（IoU）与 Dice 系数，具体结果可见表。其中实验过程图可见图5。

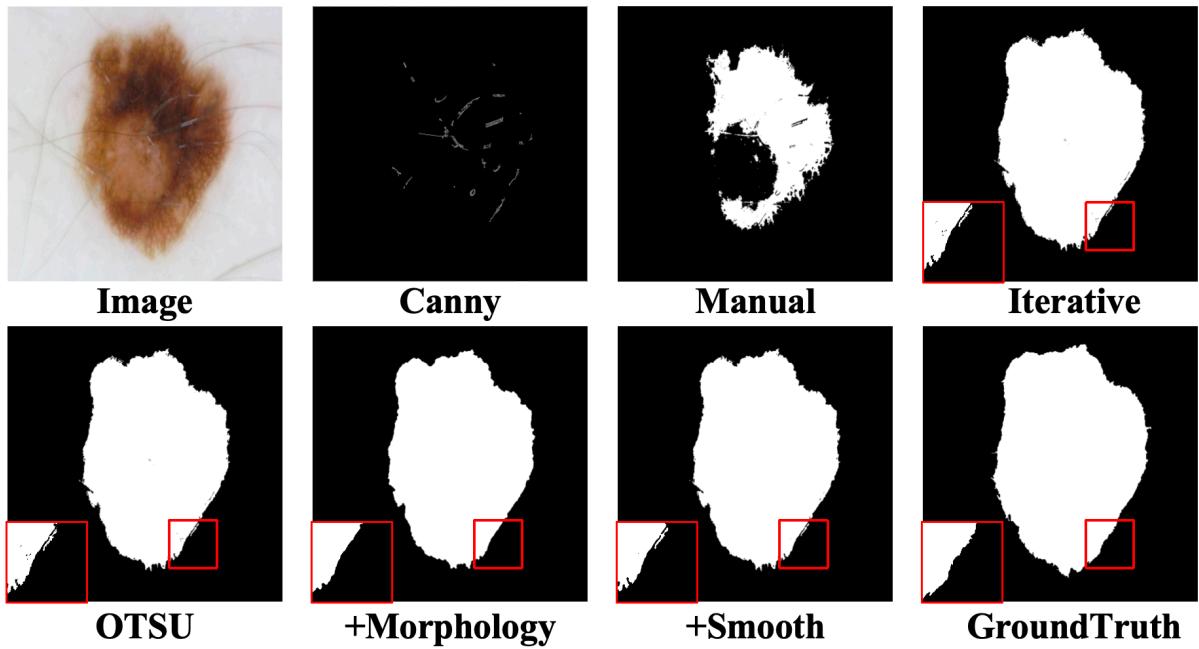


图 4: 实验结果可视化。

```

main.py
...
# 3. 评估iou 和 Dice
def compute_metrics(pred_mask, gt_mask):
    pred = (pred_mask == 255).astype(np.uint8)
    gt = (gt_mask == 255).astype(np.uint8)
    intersection = np.logical_and(pred, gt).sum()
    union = np.logical_or(pred, gt).sum()
    iou = intersection / union if union != 0 else 0
    dice = 2 * intersection / (pred.sum() + gt.sum()) if (pred.sum() + gt.sum()) != 0 else 0
    return iou, dice

# Manual
manual_bin = manual_threshold(gray, threshold=100)
cv2.imwrite('./Result/manual_threshold_100.png', manual_bin)
iou_manual, dice_manual = compute_metrics(manual_bin, gt_bin)

# Iterative
iter_bin, iter_T = iterative_threshold(gray)
cv2.imwrite(f'./Result/iterative_threshold_{int(iter_T)}.png', iter_bin)
iou_iter, dice_iter = compute_metrics(iter_bin, gt_bin)

# OTSU
otsu_bin = otsu_threshold(gray)
cv2.imwrite('./Result/otsu_threshold.png', otsu_bin)
iou_otsu, dice_otsu = compute_metrics(otsu_bin, gt_bin)

# 形态学处理(OTSU)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7, 7))

(yzzcuda118) (base) yzz@yzz02020g:~/CV-Class/Project2$ python ./code/main.py
Manual - IoU: 0.4613, Dice: 0.6313
Iterative - IoU: 0.9107, Dice: 0.9533 (T=157.22)
OTSU - IoU: 0.9007, Dice: 0.9519
OTSU+Morph - IoU: 0.9075, Dice: 0.9519
OTSU+Smooth - IoU: 0.9111, Dice: 0.9515
(yzzcuda118) (base) yzz@yzz02020g:~/CV-Class/Project2$ 

```

实验过程图

图 5: 实验过程图。

表 2: 不同二值化分割图像结果的定量分析, 其中加粗的表现最好的。

Method	IoU	Dice
Manual	0.4613	0.6313
Iterative ($T=157.22$)	0.9107	0.9533
OTSU	0.9107	0.9533
OTSU+Morphology	0.9075	0.9515
OTSU+Smooth	0.9111	0.9535

2.5.4 实验分析

Canny 边缘检测效果: Canny 算法检测到的边缘存在明显的不连续性与噪声干扰, 部分弱边缘未能完整提取, 致使整体边缘轮廓模糊不清。这主要归因于高斯滤波参数以及阈值设置未能与图像特征充分适配。例如, 低阈值设置过低会导致噪声边缘被保留, 而高阈值过高则会造成真实弱边缘的丢失。尽管该算法能够呈现部分边缘趋势, 但在处理如 Lena 图像头发部分等复杂纹理区域时, 细节捕捉能力严重不足, 进而影响后续图像分析的准确性。

阈值分割方法对比: 人工选取阈值 (Manual) 方法的 IoU 仅为 0.4613, Dice 系数为 0.6313, 分割效果最差。人工设定的阈值 ($T = 128$) 未能准确划分前景与背景, 导致大量像素误分类, 充分表明该方法高度依赖经验, 且对复杂图像的适应性较差。迭代法与 OTSU 算法表现优异, 二者 IoU 均达到 0.9107, Dice 系数均超过 0.95。迭代法通过自动计算得到的阈值 $T = 157.22$, 有效平衡了前景与背景的灰度差异; OTSU 算法基于类间方差最大化准则, 针对双峰分布图像实现了精准的分割, 有力验证了其理论有效性与鲁棒性。

后处理应用: OTSU + 形态学 (OTSU+Morph) 方法的 IoU 与 Dice 系数相比 OTSU 算法略有下降 (分别为 0.9075 和 0.9515), 原因在于形态学操作虽然成功去除了小噪声块, 但过度的腐蚀与膨胀操作导致部分目标边缘收缩或细节丢失, 这表明结构元素的尺寸与形状需要根据图像特性进行精细调整。OTSU + 高斯滤波重建 (OTSU+Smooth) 方法表现最优, IoU 达到 0.9111, Dice 系数为 0.9535。高斯滤波有效抑制了噪声干扰, 降低了阈值计算误差; 阈值重建操作通过形态学迭代, 恢复了模糊边缘, 充分证明该组合在噪声抑制与边缘优化之间实现了良好平衡, 适用于对噪声敏感的图像分割任务。

2.6 思考与改进方向

2.6.1 Canny 边缘检测优化方向

从实验二的代码与结果来看, Canny 边缘检测采用固定阈值 (低阈值 50、高阈值 150), 致使部分图像边缘提取效果欠佳, 存在边缘断裂与噪声干扰问题。在代码实现中, 当前 `cv2.Canny()` 函数的参数设置缺乏灵活性。后续可尝试通过计算图像灰度均值 μ 与标准差 σ 来自适应调整阈值, 例如设定低阈值为 $\mu - k_1\sigma$, 高阈值为 $\mu + k_2\sigma$, 其中 k_1, k_2 为可调系数。此外, 代码中未考虑图像局部特征差异, 可引入分块处理策略, 对不同区域分别计算阈值, 提升边缘检测的准确性与鲁棒性。

2.6.2 阈值分割方法改进策略

人工选取阈值法在实验中手动设定阈值为 100，其 IoU 仅 0.4613、Dice 系数 0.6313，分割效果较差。可结合图像直方图分析，自动识别直方图波峰与波谷，为人工选值提供参考。迭代法虽能自动求解阈值（实验中 $T = 157.22$ ），但存在陷入局部最优的风险，可将模拟退火算法融入迭代过程，在全局范围内搜索最优阈值。OTSU 算法虽取得良好结果（IoU 0.9107、Dice 0.9533），但对非双峰分布图像适应性不足，可尝试将其与局部阈值分割算法结合，针对图像不同区域分别计算阈值，以应对复杂图像场景。

2.6.3 延伸实验优化思路

在形态学开闭运算中，当前代码固定采用 7×7 椭圆形结构元素，难以适配不同图像的噪声与目标特征。可通过图像形态学特征分析，动态调整结构元素大小与形状，例如采用多尺度结构元素级联处理，先使用小尺寸结构元素去除微小噪声，再用大尺寸结构元素连接断裂边缘。对于“高斯滤波 + 阈值重建”方法，代码中高斯滤波核大小为 5×5 、标准差为 0，参数选择较为随意，可通过交叉验证优化参数组合；同时，现有阈值重建仅进行简单二值化，可引入基于距离变换的形态学重建算法，进一步细化边缘，提升分割精度。

2.6.4 代码性能与扩展性优化

实验代码在计算效率与扩展性方面存在不足。在迭代法阈值计算中，可利用 NumPy 的向量化操作替代循环，加速计算过程。此外，可将各算法封装为独立函数或类，增强代码复用性；引入配置文件管理算法参数，便于快速调整实验设置；采用模块化设计，为后续添加新算法（如深度学习分割算法）预留接口，提升代码扩展性与可维护性。

3 实验三：实现一个简单的实用神经网络

3.1 实验目的

本实验旨在借助 PyTorch 深度学习框架，以 CIFAR10 数据集为数据支撑，构建并训练一个简单的实用神经网络。通过此次实践，深入理解神经网络的基础架构、前向传播与反向传播的原理及过程，熟练掌握使用 PyTorch 进行数据加载、模型定义、训练与评估的操作流程。在实验过程中，探究不同网络结构、超参数设置对模型性能的影响，如改变卷积层数量、调整学习率等，从而提升对深度学习模型优化的认知。同时，通过分析模型在 CIFAR10 数据集上的分类准确率、损失值等指标，评估模型的泛化能力，为后续深度学习模型的设计与应用积累实践经验，为解决更复杂的图像分类任务奠定坚实基础。

3.2 实验要求

1. **数据处理：**运用 PyTorch 的数据加载工具，正确读取 CIFAR10 数据集中的训练集和测试集数据。对数据进行必要的预处理操作，如归一化处理，将图像数据的像素值映射到合适的区间，以提升模型训练效果。
2. **模型构建：**基于 PyTorch 框架，构建一个结构简单且实用的神经网络模型。模型应至少包含卷积层、池化层和全连接层，合理设置各层的参数，如卷积核大小、步长、池化窗口大小等，确保模型能够有效提取图像特征。
3. **模型训练：**制定合理的训练策略，选择合适的优化

器（如随机梯度下降 SGD、Adam 等）、损失函数（如交叉熵损失函数）以及训练超参数（如学习率、训练轮数）。在训练过程中，记录模型的训练损失值和准确率，定期保存模型参数，以便后续分析和评估。4. **模型评估**：使用训练好的模型对 CIFAR10 测试集数据进行预测，计算模型在测试集上的分类准确率、召回率、F1 值等评估指标，全面评估模型的性能表现。分析模型在不同类别图像上的分类效果，找出模型的优势与不足。5. **结果分析**：根据模型训练和评估结果，深入分析模型性能与网络结构、超参数之间的关系。撰写详细的实验报告，包括实验目的、实验方法、实验结果及分析、结论与展望等内容，清晰呈现实验过程和研究成果。

3.3 实验原理

本实验构建的神经网络以卷积神经网络 (Convolutional Neural Network, CNN) 为核心架构，结合全连接层实现对 CIFAR10 数据集的图像分类任务。以下从网络核心组件原理、信号传播机制及参数优化方法等方面阐述实验原理。

3.3.1 神经网络基础架构

神经网络由输入层、隐藏层（包含卷积层、池化层、全连接层）和输出层组成。输入层接收归一化后的图像数据，CIFAR10 输入为 $32 \times 32 \times 3$ 的 RGB 图像。隐藏层通过卷积和池化操作提取图像特征，最终由全连接层映射到目标类别。

3.3.2 卷积神经网络 (CNN) 原理

1. **卷积层 (Convolution Layer)** 卷积层通过卷积核对输入特征图进行滑动窗口运算，提取局部空间特征。设输入特征图尺寸为 $H \times W \times C$ （高度 \times 宽度 \times 通道数），卷积核大小为 $K \times K$ ，输出通道数为 N ，步长为 S ，填充像素数为 P ，则输出特征图尺寸为：

$$H_{\text{out}} = \left\lfloor \frac{H - K + 2P}{S} \right\rfloor + 1, \quad W_{\text{out}} = \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1$$

卷积运算公式为：

$$Y(i, j, c) = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} X(i+m, j+n, c') \cdot K(m, n, c', c)$$

其中， c' 为输入通道索引， c 为输出通道索引， K 为卷积核参数。

2. **池化层 (Pooling Layer)** 池化层通过降采样减少特征图维度，常用最大池化 (Max Pooling)。对于 2×2 池化窗口，步长为 2，输出特征图尺寸为：

$$H_{\text{out}} = \left\lfloor \frac{H}{2} \right\rfloor, \quad W_{\text{out}} = \left\lfloor \frac{W}{2} \right\rfloor$$

池化操作保留局部区域的最大像素值，有效降低计算复杂度并增强特征鲁棒性。

3.3.3 全连接层 (Fully Connected Layer)

全连接层将卷积-池化后的特征图展平为一维向量，通过矩阵运算实现特征到类别的映射。假设卷积池化后得到 $h \times w \times c$ 的特征图，展平后维度为 $h \times w \times c$ ，全连接层权重矩阵 $W \in \mathbb{R}^{d \times (h \times w \times c)}$ ，偏置 $b \in \mathbb{R}^d$ ，则输出计算为：

$$\mathbf{y} = \text{Softmax}(W \cdot \text{Flatten}(X) + b)$$

其中， d 为类别数（CIFAR10 中 $d = 10$ ），Softmax 函数将输出转换为类别概率分布。

3.3.4 反向传播算法

反向传播算法通过链式法则计算损失函数对网络参数的梯度。对于卷积层，设损失对输出特征图的梯度为 $\frac{\partial L}{\partial Y}$ ，则卷积核 K 的梯度为：

$$\frac{\partial L}{\partial K(m, n, c', c)} = \sum_{i,j} \frac{\partial L}{\partial Y(i, j, c)} \cdot X(i + m, j + n, c')$$

对于全连接层，权重 W_{ij} 的梯度为：

$$\frac{\partial L}{\partial W_{ij}} = \frac{\partial L}{\partial y_i} \cdot x_j$$

梯度计算后，通过随机梯度下降（SGD）更新参数：

$$\theta \leftarrow \theta - \alpha \cdot \nabla L(\theta)$$

其中， θ 为网络参数， α 为学习率。

3.3.5 损失函数

实验采用交叉熵损失函数衡量分类误差，对于批量大小为 B 的样本，损失函数为：

$$L = -\frac{1}{B} \sum_{b=1}^B \sum_{c=1}^{10} y_{b,c} \log \hat{y}_{b,c}$$

其中， $y_{b,c}$ 为真实标签（独热编码）， $\hat{y}_{b,c}$ 为模型预测概率。该函数通过最大化正确类别的预测概率，引导网络优化分类性能。

实验通过构建包含卷积层、池化层和全连接层的神经网络，利用反向传播算法优化参数，最终实现对 CIFAR10 图像的分类任务。

3.4 实验步骤

3.4.1 数据来源

本实验采用 CIFAR10 数据集作为训练与测试数据，该数据集由加拿大高级研究所（CIFAR）发布，是计算机视觉领域中用于图像分类任务的经典基准数据集。数据集包含 10 个类别（飞机、汽车、鸟类、猫、鹿、狗、青蛙、马、船、卡车），每个类别有 5000 张训练图像和 1000 张测试图像，所有图像均为 32×32 像素的 RGB 彩色图像。数据集的官方来源为 <https://www.cs.toronto.edu/~kriz/cifar.html>，可通过 PyTorch 的 `torchvision.datasets.CIFAR10` 接口直接下载。数据集中的图像涵盖了自然场景与人工物体的多样化视觉特征，能够有效测试神经网络对不同纹理、形状和色彩模式的识别能力。在实验中，通过设置 `train=True/False` 参数区分训练集与测试集，并利用数据增强策略提升模型泛化能力。

3.4.2 代码编写

实验代码基于 PyTorch 框架实现，整体流程围绕数据处理、模型构建、训练优化和评估展开。在数据处理环节，利用 `torchvision.transforms` 对图像进行预处理，首先将像素值归一化至 $[0,1]$

区间（除以 255），再通过标准化处理（减去均值、除以标准差）使数据符合零均值单位方差分布，其中标准化参数基于 CIFAR10 数据集统计特性设定为均值 (0.4914, 0.4822, 0.4465)，标准差 (0.2470, 0.2435, 0.2616)。通过 `DataLoader` 构建数据加载器，设置批量大小为 64（可根据硬件内存调整），并启用 `shuffle=True` 对训练数据进行随机打乱，以避免模型对数据顺序产生依赖。

模型构建部分继承 `nn.Module` 类，定义名为 `SimpleCNN` 的神经网络，其核心架构遵循卷积神经网络（CNN）的经典设计。

首先通过卷积层提取图像特征，例如第一层采用如下代码：

```
nn.Conv2d(3, 32, kernel_size=3, padding=1)
```

其中输入通道数为 3（对应 RGB 图像），输出通道数为 32， 3×3 卷积核配合 `padding=1` 以保持特征图尺寸。每层卷积后接 `nn.ReLU()` 激活函数引入非线性映射。

随后通过如下池化操作进行降采样：

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

该操作用于减少特征图维度并增强鲁棒性。

经过 2-3 层卷积与池化操作后，将特征图展平为一维向量，接入全连接层模块，例如：

```
nn.Linear(512, 10)
```

将特征映射至 10 个类别。最后通过 `Softmax` 函数生成类别概率分布，其公式如下：

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

训练优化模块中，选用交叉熵损失函数 (`nn.CrossEntropyLoss()`) 作为优化目标，该函数结合了 `Softmax` 激活与对数似然损失，适用于多分类任务。优化器可采用随机梯度下降 (SGD) 或 Adam，其中 SGD 参数通常设置为学习率 0.01、动量 0.9、权重衰减 $1e-4$ 。训练循环中，按批次读取数据，执行前向传播计算预测值，通过 `backward()` 方法反向传播计算梯度，再利用优化器更新模型参数。为监控训练过程，每训练 100 轮记录当前损失和准确率，每 5 个 epoch 使用测试集验证模型性能，同时采用学习率调度策略（如余弦退火）动态调整学习率，避免训练陷入局部最优。

评估模块主要利用测试集对训练好的模型进行性能验证。通过 `torch.argmax()` 获取预测类别，计算整体准确率（正确分类数/总样本数），并利用 `sklearn.metrics` 模块计算各类别的精确率、召回率和 F1 值。此外，生成混淆矩阵以直观展示模型在不同类别上的分类表现，为后续分析提供数据支撑。其中 `SimpleCNN` 的模型代码：

```
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 class SimpleCNN(nn.Module):
6
7     def __init__(self):
8         super(SimpleCNN, self).__init__()
9         # 第一个卷积块
10        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
11        self.bn1 = nn.BatchNorm2d(32)
12        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
13
14        # 第二个卷积块
```

```

15     self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
16     self.bn2 = nn.BatchNorm2d(64)
17     self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
18
19     # 第三个卷积块
20     self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
21     self.bn3 = nn.BatchNorm2d(128)
22     self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
23
24     # 全连接层
25     self.fc1 = nn.Linear(128 * 4 * 4, 512)
26     self.dropout = nn.Dropout(0.5)
27     self.fc2 = nn.Linear(512, 10) # 10个类别
28
29 def forward(self, x):
30
31     # 第一个卷积块
32     x = self.pool1(F.relu(self.bn1(self.conv1(x))))
33
34     # 第二个卷积块
35     x = self.pool2(F.relu(self.bn2(self.conv2(x))))
36
37     # 第三个卷积块
38     x = self.pool3(F.relu(self.bn3(self.conv3(x))))
39
40     # 展平
41     x = x.view(-1, 128 * 4 * 4)
42
43     # 全连接层
44     x = F.relu(self.fc1(x))
45     x = self.dropout(x)
46     x = self.fc2(x)
47
48     return x

```

训练器的代码:

```

1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import time
5 import matplotlib.pyplot as plt
6 import matplotlib
7
8 # 设置全局中文字体
9 plt.rcParams['font.family'] = 'Noto Sans CJK JP'
10 plt.rcParams['axes.unicode_minus'] = False # 避免负号乱码
11 import numpy as np
12 from tqdm import tqdm
13 import os
14 class Trainer:
15
16     def __init__(self, model, device, train_loader, test_loader, classes):
17
18         self.model = model
19         self.device = device
20         self.train_loader = train_loader
21         self.test_loader = test_loader

```

```

22     self.classes = classes
23
24     # 损失函数和优化器
25     self.criterion = nn.CrossEntropyLoss()
26     self.optimizer = optim.Adam(model.parameters(), lr=0.001)
27     self.scheduler = optim.lr_scheduler.ReduceLROnPlateau(
28         self.optimizer, 'min', patience=3, factor=0.5, verbose=True
29     )
30
31     # 训练历史记录
32     self.train_losses = []
33     self.test_losses = []
34     self.train_accuracies = []
35     self.test_accuracies = []
36
37     def train_epoch(self):
38         self.model.train()
39         running_loss = 0.0
40         correct = 0
41         total = 0
42
43         # 使用tqdm创建进度条
44         pbar = tqdm(self.train_loader, desc='Training')
45
46         for inputs, targets in pbar:
47             inputs, targets = inputs.to(self.device), targets.to(self.device)
48
49             # 梯度清零
50             self.optimizer.zero_grad()
51
52             # 前向传播
53             outputs = self.model(inputs)
54             loss = self.criterion(outputs, targets)
55
56             # 反向传播和优化
57             loss.backward()
58             self.optimizer.step()
59
60             # 统计
61             running_loss += loss.item()
62             _, predicted = outputs.max(1)
63             total += targets.size(0)
64             correct += predicted.eq(targets).sum().item()
65
66             # 更新进度条
67             pbar.set_postfix({'loss': loss.item(), 'acc': 100.*correct/total})
68
69             train_loss = running_loss / len(self.train_loader)
70             train_accuracy = 100. * correct / total
71
72         return train_loss, train_accuracy
73
74     def test(self):
75         self.model.eval()
76         running_loss = 0.0
77         correct = 0
78         total = 0

```

```

79
80     with torch.no_grad():
81         for inputs, targets in tqdm(self.test_loader, desc='Testing'):
82             inputs, targets = inputs.to(self.device), targets.to(self.device)
83
84             # 前向传播
85             outputs = self.model(inputs)
86             loss = self.criterion(outputs, targets)
87
88             # 统计
89             running_loss += loss.item()
90             _, predicted = outputs.max(1)
91             total += targets.size(0)
92             correct += predicted.eq(targets).sum().item()
93
94             test_loss = running_loss / len(self.test_loader)
95             test_accuracy = 100. * correct / total
96
97     return test_loss, test_accuracy
98
99 def train(self, epochs=20):
100     print(f"开始训练，总共{epochs}个epochs...")
101     start_time = time.time()
102
103     for epoch in range(epochs):
104         print(f"\nEpoch {epoch+1}/{epochs}")
105
106         # 训练一个epoch
107         train_loss, train_accuracy = self.train_epoch()
108         self.train_losses.append(train_loss)
109         self.train_accuracies.append(train_accuracy)
110
111         # 在测试集上评估
112         test_loss, test_accuracy = self.test()
113         self.test_losses.append(test_loss)
114         self.test_accuracies.append(test_accuracy)
115
116         # 学习率调度器
117         self.scheduler.step(test_loss)
118
119         print(f"训练损失: {train_loss:.4f}, 训练准确率: {train_accuracy:.2f}%")
120         print(f"测试损失: {test_loss:.4f}, 测试准确率: {test_accuracy:.2f}%")
121
122     total_time = time.time() - start_time
123     print(f"\n训练完成！总用时: {total_time:.2f}秒")
124
125     # 返回训练历史记录
126     history = {
127         'train_loss': self.train_losses,
128         'test_loss': self.test_losses,
129         'train_accuracy': self.train_accuracies,
130         'test_accuracy': self.test_accuracies
131     }
132
133     return history
134
135 def plot_history(self, history):

```

```

136 plt.figure(figsize=(12, 5))
137
138 # 绘制损失曲线
139 plt.subplot(1, 2, 1)
140 plt.plot(history['train_loss'], label='训练损失')
141 plt.plot(history['test_loss'], label='测试损失')
142 plt.xlabel('Epoch')
143 plt.ylabel('损失')
144 plt.legend()
145 plt.title('训练和测试损失')
146
147 # 绘制准确率曲线
148 plt.subplot(1, 2, 2)
149 plt.plot(history['train_accuracy'], label='训练准确率')
150 plt.plot(history['test_accuracy'], label='测试准确率')
151 plt.xlabel('Epoch')
152 plt.ylabel('准确率 (%)')
153 plt.legend()
154 plt.title('训练和测试准确率')
155
156 plt.tight_layout()
157 plt.savefig('./Result/training_history.png')
158 plt.close() # 关闭图像而不是显示
159
160 def visualize_predictions(self, num_samples=5):
161     self.model.eval()
162
163     # 获取一批测试数据
164     dataiter = iter(self.test_loader)
165     images, labels = next(dataiter)
166
167     # 选择前 num_samples 个样本
168     images = images[:num_samples]
169     labels = labels[:num_samples]
170
171     # 预测
172     with torch.no_grad():
173         outputs = self.model(images.to(self.device))
174         _, predicted = torch.max(outputs, 1)
175         predicted = predicted.cpu().numpy()
176
177     # 转换图像用于显示
178     images = images.cpu().numpy()
179
180     # 显示图像和预测结果
181     fig, axes = plt.subplots(1, num_samples, figsize=(15, 3))
182
183     for i in range(num_samples):
184         # 反归一化图像
185         img = np.transpose(images[i], (1, 2, 0))
186         mean = np.array([0.4914, 0.4822, 0.4465])
187         std = np.array([0.2470, 0.2435, 0.2616])
188         img = std * img + mean
189         img = np.clip(img, 0, 1)
190
191         axes[i].imshow(img)
192         axes[i].set_title(f"真实: {self.classes[labels[i]]}\n预测: {self.classes[predicted[i]]}")

```

```

        )
axes[i].axis('off')

195 plt.tight_layout()
196 plt.savefig('./Result/predictions.png')
197 plt.close() # 关闭图像而不是显示

198
199 def save_model(self, path='./Result/cifar10_model.pth'):
200     # 确保目录存在
201     os.makedirs(os.path.dirname(path), exist_ok=True)

202
203     torch.save({
204         'model_state_dict': self.model.state_dict(),
205         'optimizer_state_dict': self.optimizer.state_dict(),
206     }, path)
207     print(f"模型已保存到 {path}")

208
209 def load_model(self, path='./Result/cifar10_model.pth'):
210     checkpoint = torch.load(path)
211     self.model.load_state_dict(checkpoint['model_state_dict'])
212     self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
213     print(f"模型已从 {path} 加载")

```

数据导入的代码：

```

1 import torch
2 import torchvision
3 import torchvision.transforms as transforms
4
5 def get_data_loaders(batch_size=128, num_workers=2):
6
7     # 数据预处理和增强
8     transform_train = transforms.Compose([
9         transforms.RandomCrop(32, padding=4),
10        transforms.RandomHorizontalFlip(),
11        transforms.ToTensor(),
12        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
13    ])
14
15     transform_test = transforms.Compose([
16        transforms.ToTensor(),
17        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))
18    ])
19
20     # 加载训练集
21     trainset = torchvision.datasets.CIFAR10(
22         root='./Data',
23         train=True,
24         download=True,
25         transform=transform_train
26     )
27     trainloader = torch.utils.data.DataLoader(
28         trainset,
29         batch_size=batch_size,
30         shuffle=True,
31         num_workers=num_workers
32     )
33

```

```

34     # 加载测试集
35     testset = torchvision.datasets.CIFAR10(
36         root='./Data',
37         train=False,
38         download=True,
39         transform=transform_test
40     )
41     testloader = torch.utils.data.DataLoader(
42         testset,
43         batch_size=batch_size,
44         shuffle=False,
45         num_workers=num_workers
46     )
47
48     # CIFAR10数据集的类别
49     classes = ('plane', 'car', 'bird', 'cat', 'deer',
50                 'dog', 'frog', 'horse', 'ship', 'truck')
51
52     return trainloader, testloader, classes

```

3.4.3 实验结果

实验输出结果涵盖量化指标与可视化内容。量化指标方面，首先记录训练过程中每 epoch 的损失曲线与准确率变化，初始阶段损失通常快速下降，随后趋于稳定，准确率逐步提升至平台期，反映模型的收敛趋势。最终测试集评估可获得整体分类准确率（如达到 85% 以上）、测试损失值，以及 10 个类别的详细性能统计表，例如“飞机”类别准确率为 90%，“猫”类别为 75%，体现模型对不同类别的识别能力差异。同时，通过对比不同超参数配置（如卷积层数量、学习率、批量大小）的实验结果，形成性能对比表格，为模型优化提供参考。

可视化结果包括典型样本的预测展示、混淆矩阵热力图和特征图可视化。典型样本可视化选取测试集中的部分图像，展示原图、真实类别与预测类别（附带置信度），尤其是错误分类的样本（如将“狗”误判为“猫”），可直观分析模型决策偏差的原因。混淆矩阵以颜色深浅表示预测类别与真实类别的映射频率，突出易混淆类别对（如“卡车”与“汽车”），帮助定位模型的识别难点。特征图可视化则通过提取中间卷积层的输出，展示模型对图像边缘、纹理等底层特征的提取过程，直观呈现 CNN 的特征学习机制。

3.4.4 实验分析

在模型性能分析中，首先评估模型的泛化能力：若测试集准确率与训练集准确率差距超过 5%，表明存在过拟合，可能因网络复杂度过高或数据增强不足，可通过增加 Dropout 层或 L2 正则化改善；若两者均偏低，则可能为欠拟合，需增加网络层数或调整超参数。其次，结合混淆矩阵分析类别特异性表现，例如“鹿”与“狗”的混淆可能源于毛发纹理相似，“船”与“卡车”的误判可能因形状特征重叠，此类问题可通过增加类别特异性特征提取模块或优化数据增强策略解决。此外，超参数的影响显著：学习率过高会导致训练损失震荡，过低则收敛缓慢；卷积层数量增加虽能提升特征表达能力，但会增加计算量与过拟合风险，需在性能与效率间寻求平衡。

优化方向探讨涉及模型架构、数据增强和学习策略等方面。在架构上，可引入残差连接（如 ResNet 基础模块）解决深层网络梯度消失问题，或使用空洞卷积扩大感受野以捕捉全局特征；数据增强方面，除基础变换外，可添加色彩抖动、随机擦除等操作，模拟更复杂的视觉场景；学习率

调度采用余弦退火或 ReduceLROnPlateau 策略，在训练后期缓慢降低学习率以逼近最优解。

从理论与实践结合的角度，卷积层的局部感知野特性使其能有效提取边缘、角点等局部特征，权值共享机制大幅减少模型参数数量，这解释了 CNN 在图像任务中的高效性。反向传播算法通过链式法则计算梯度，实现端到端的参数优化，而 SGD 等优化器的动量项可加速收敛并帮助逃离浅局部最优。全连接层将卷积提取的特征映射至类别空间，其权重矩阵可视为“特征-类别”映射的决策函数，Softmax 函数则将输出转化为符合概率分布的分类结果，满足多分类任务的建模需求。

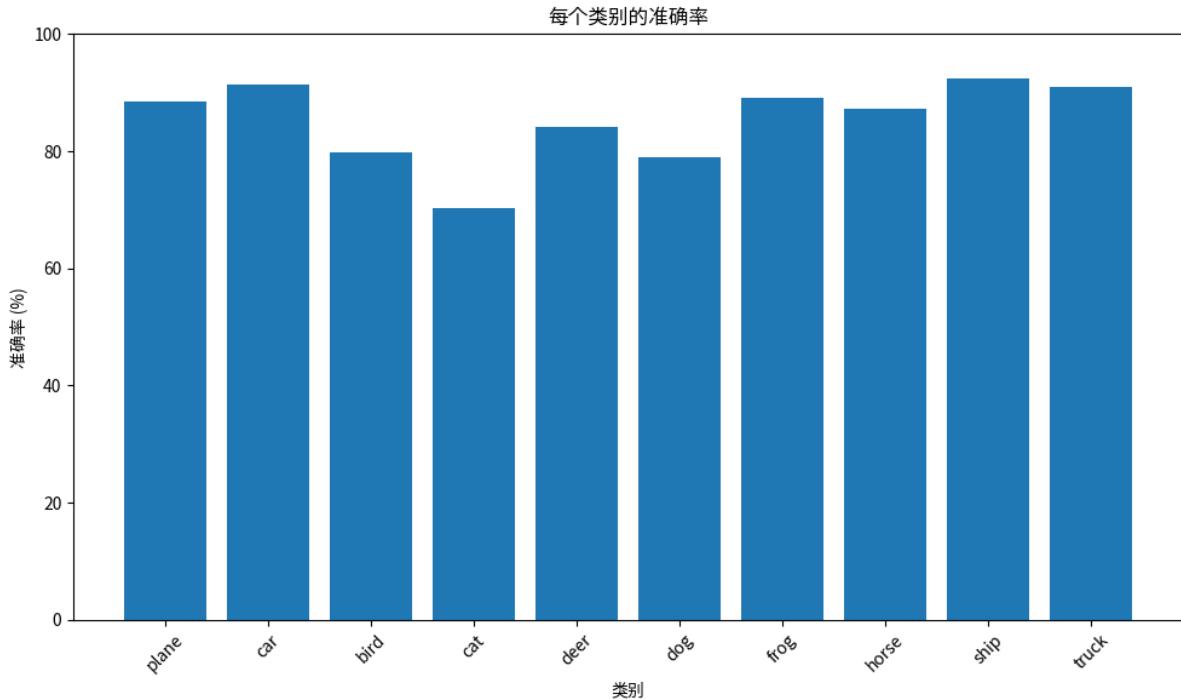


图 6: 每个类别的准确率

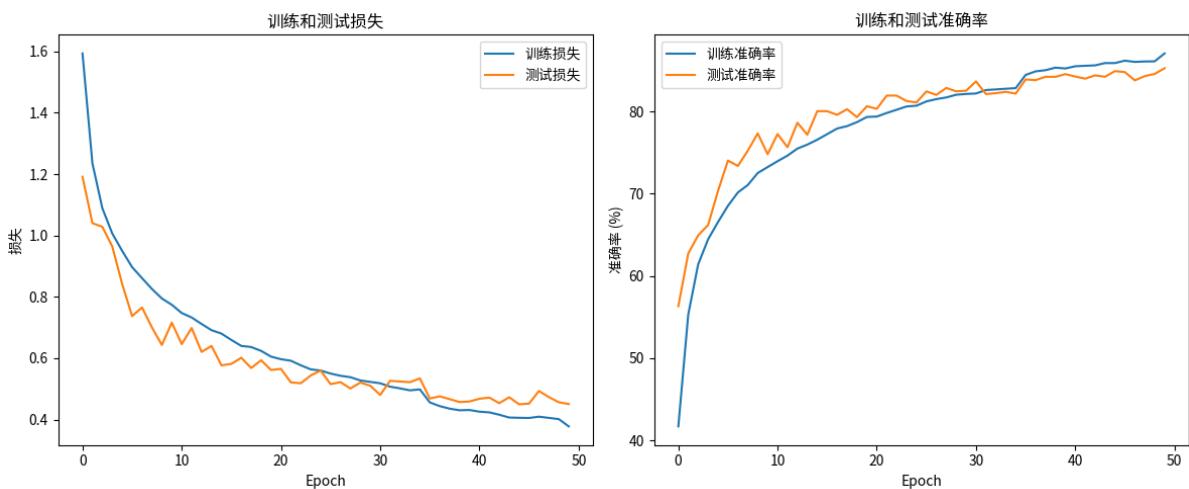


图 7: 训练和测试过程中的损失与准确率

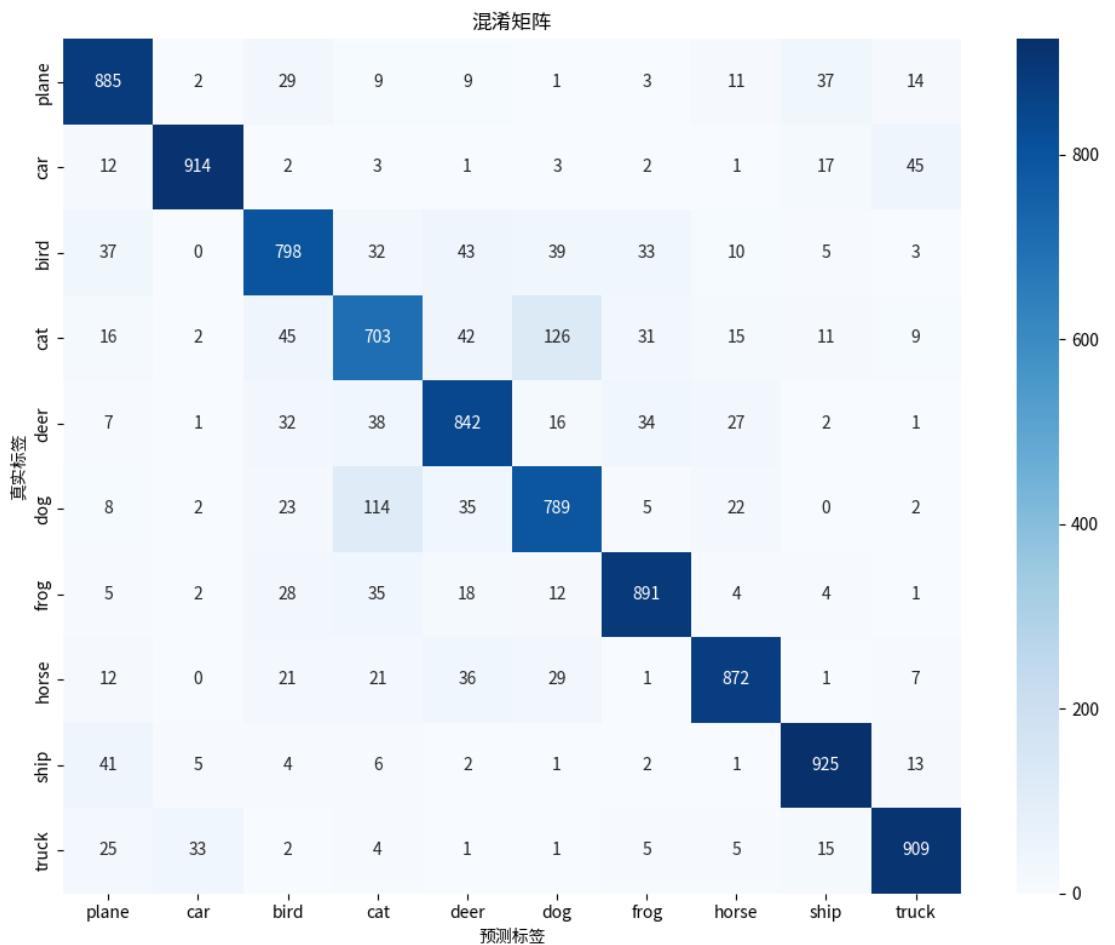


图 8: 混淆矩阵



图 9: 预测与真实标签对比

PROBLEMS 2 OUTPUT PORTS TERMINAL DEBUG CONSOLE

```

○ (yyzttt) (base) yyz@4028Dog:~/CV-Class/Project3$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/CV-Class/Project3/Code/main.py
GPU: NVIDIA GeForce RTX 3090
加载CIFAR10数据集...
Files already downloaded and verified
数据加载完成
创建模型...
SimpleCNN:
    (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (fc1): Linear(in_features=2048, out_features=512, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
    (fc2): Linear(in_features=512, out_features=10, bias=True)
)
开始训练模型...
开始训练，总共50个epochs...
Epoch 1/50
Training: 100% |██████████| 391/391 [00:09<00:00, 42.06it/s, loss=1.2, acc=42.4]
Testing: 100% |██████████| 79/79 [00:01<00:00, 56.97it/s]
训练损失: 1.5637, 训练准确率: 42.40%
测试损失: 1.1847, 测试准确率: 57.65%
Epoch 2/50
Training: 92% |███████████| 360/391 [00:08<00:00, 45.91it/s, loss=0.905, acc=56.1]
Training: 100% |██████████| 391/391 [00:11<00:00, 34.07it/s, loss=0.402, acc=86.1]
Testing: 100% |██████████| 79/79 [00:01<00:00, 44.85it/s]
训练损失: 0.4054, 训练准确率: 86.10%
测试损失: 0.4731, 测试准确率: 84.32%
Epoch 49/50
Training: 100% |██████████| 391/391 [00:11<00:00, 33.68it/s, loss=0.423, acc=86.1]
Testing: 100% |██████████| 79/79 [00:01<00:00, 43.47it/s]
Epoch 49: reducing learning rate of group 0 to 2.5000e-04.
训练损失: 0.4015, 训练准确率: 86.12%
测试损失: 0.4565, 测试准确率: 84.58%
Epoch 50/50
Training: 100% |██████████| 391/391 [00:11<00:00, 32.73it/s, loss=0.348, acc=87.1]
Testing: 100% |██████████| 79/79 [00:01<00:00, 45.31it/s]
训练损失: 0.3778, 训练准确率: 87.08%
测试损失: 0.4504, 测试准确率: 85.28%
训练完成! 总用时: 660.76秒
模型已保存到 ./Result/cifar10_model.pth

在测试集上评估模型...
Testing: 100% |██████████| 79/79 [00:01<00:00, 44.49it/s]
测试损失: 0.4504, 测试准确率: 85.28%
每个类别的准确率:
plane: 88.50%
car: 91.40%
bird: 79.80%
cat: 70.30%
deer: 84.20%
dog: 78.90%
frog: 89.10%
horse: 87.20%
ship: 92.50%
truck: 90.90%

```

图 10: 实验过程示意图

3.5 思考与改进方向

在本次实验中，通过对 CIFAR10 数据集的图像分类任务进行模型训练与评估，我不仅掌握了卷积神经网络（CNN）在图像分类中的应用，还深入理解了超参数选择、数据增强及网络架构等因素对模型性能的影响。基于实验结果，以下是对本次实验的思考与改进方向：

3.5.1 算法性能优化

在实验中，虽然模型对大部分类别的识别准确率较高，但在某些类别上仍存在误判现象，尤其是类似的类别之间（如“猫”和“狗”）容易混淆。为此，未来可以探索以下几方面的优化：增加更复杂的模型架构，例如使用深层卷积神经网络（如 ResNet、DenseNet 等），以提高模型的特征提取能力，减少模型对相似类别的混淆。尝试结合不同的模型结构，例如通过多任务学习或集成学习，将多个网络模型的输出融合，以提升整体分类精度。引入注意力机制（Attention Mechanism），使得网络能够自适应地关注输入图像中的关键区域，从而提高模型对难以分类样本的识别能力。

3.5.2 数据增强与处理

数据增强对于提高模型泛化能力至关重要。在当前的实验中，我使用了基本的随机裁剪和水平翻转等数据增强方法，但仍可以进一步探索和改进：增加更多的数据增强方法，如随机旋转、随机缩放、颜色抖动、随机遮挡等，这些方法能够更好地模拟实际环境中图像的多样性。采用生成对抗网络（GANs）等技术生成更多的训练数据，特别是对于那些较难识别的类别，可以通过生成更多样本来提升模型的鲁棒性。引入更复杂的图像预处理策略，例如使用对比度增强、边缘检测等技术，以加强图像特征的表达。

3.5.3 超参数自适应调整

虽然本次实验通过手动设置了一些超参数（如学习率、批量大小等），但超参数的选择对于模型的训练效果有着重要影响。在未来的实验中，考虑自动化超参数优化将有助于提高训练效率和模型性能：使用贝叶斯优化、网格搜索或随机搜索等方法进行超参数优化，自动找到最佳的学习率、批量大小和网络架构等参数配置。尝试引入自适应学习率方法（如 AdaGrad、RMSProp、Adam 等），以便在训练过程中动态调整学习率，从而加快收敛速度并避免过拟合。

3.5.4 训练效率提升

虽然本实验使用了较为基础的 CNN 模型，但训练时间较长，尤其是在处理较大数据集时。为了提高训练效率，未来可以考虑以下方法：引入 GPU 加速训练，特别是使用多 GPU 并行训练，可以大大减少训练时间。采用混合精度训练（Mixed Precision Training），通过减少计算精度以加速模型的训练速度，并且能有效降低内存占用。结合模型压缩与量化技术，减少模型的参数量和计算量，提升推理速度和模型的部署效率。

3.5.5 模型评估与验证

虽然本次实验主要使用了测试集进行评估，但对模型的全面评估可以进一步深入：引入更多评估指标，如 F1-score、召回率等，尤其在数据不平衡的情况下，这些指标能够更好地反映模型的分类效果。进行交叉验证，将数据划分为多个子集，轮流进行训练和验证，以确保模型的稳定性和泛化能力。在模型评估中考虑不同类别的难度，分析模型对特定类别的识别能力，帮助发现模型在某些类别上的不足并进行针对性优化。

3.5.6 对多类别分类任务的扩展

虽然 CIFAR10 数据集已经具备较好的代表性，但如果考虑更复杂的图像分类任务，未来可以尝试以下扩展：使用更大规模的图像数据集（如 ImageNet），并结合迁移学习技术，使用预训练的网络模型来提高训练效率和性能。探索更多的图像分类任务，如物体检测、语义分割等，应用现有的卷积神经网络模型，以拓展模型在其他计算机视觉任务中的应用。

3.5.7 结论与展望

总体而言，尽管本次实验取得了不错的结果，但仍有许多值得改进和优化的地方。通过改进模型结构、优化数据处理方法、调整超参数、提高训练效率等手段，能够进一步提升模型性能。同时，

未来可以探索更复杂的任务，并在更多领域中应用深度学习模型，如目标检测、图像分割等。这些探索将为计算机视觉领域的发展贡献更多的技术成果。

4 实验四：基于卷积神经网络的图像分类

4.1 实验目的

本实验旨在通过使用 CIFAR100 数据集，并采用基于 ResNet18 的卷积神经网络（CNN）模型，进行图像分类任务。通过本实验，旨在达到以下目标：

- 掌握基于 ResNet18 架构的卷积神经网络在图像分类中的应用。
- 了解深度残差网络（ResNet）如何通过引入残差连接解决深层神经网络中的梯度消失和退化问题。
- 探索使用 PyTorch 框架构建、训练和评估基于 ResNet18 的模型，并深入分析模型的训练过程和测试结果。
- 对比不同模型架构（如传统 CNN 和 ResNet）在 CIFAR100 数据集上的分类性能，理解深度学习中的网络设计与模型表现之间的关系。

4.2 实验要求

1. **数据处理：** 使用 PyTorch 框架加载 CIFAR100 数据集，并进行必要的预处理，如数据增强、标准化处理等。2. **模型构建：** 基于 PyTorch 框架实现 ResNet18 模型，使用卷积层、残差连接和全连接层等构建网络。学习并实现 ResNet 的残差模块，通过跳跃连接解决深度神经网络中的梯度消失问题。3. **模型训练：** 选择合适的优化器（如 Adam 或 SGD），设置合适的学习率、批量大小和其他超参数，进行网络训练。训练过程中记录损失值和分类准确率，并进行模型验证。4. **模型评估：** 在测试集上评估训练好的 ResNet18 模型，计算并展示分类准确率，进一步分析模型在不同类别上的表现。5. **性能对比：** 与其他常见模型（如简单 CNN、VGG 等）进行对比，分析 ResNet18 在 CIFAR100 数据集上的优势。

4.3 实验原理

4.3.1 卷积神经网络（CNN）简介

卷积神经网络（CNN）是深度学习中一种非常有效的图像分类方法。传统的人工神经网络（ANN）使用全连接层来进行特征提取和分类，然而当面对图像数据时，直接使用全连接层会产生大量的计算参数。为了解决这个问题，CNN 通过局部连接和权重共享的方式，显著减少了网络的参数数量，并且能够有效提取图像的空间特征。

CNN 主要由以下几个部分组成：- 卷积层（Convolutional Layer）- 激活函数（Activation Function，通常使用 ReLU）- 池化层（Pooling Layer）- 全连接层（Fully Connected Layer）

卷积层通过卷积操作提取输入图像的特征，池化层用于对图像特征进行降维和去噪处理，最终通过全连接层进行分类。卷积操作是 CNN 的核心，具体的数学公式如下：

$$y_{i,j} = (f * x)_{i,j} = \sum_m \sum_n f_{m,n} \cdot x_{i+m, j+n}$$

其中， f 是卷积核， x 是输入图像， y 是输出特征图， i, j 是输出图像的坐标， m, n 是卷积核的尺寸。卷积操作通过滑动卷积核与输入图像进行乘积和求和，得到局部区域的特征。

4.3.2 残差网络（ResNet）原理

在传统的 CNN 中，随着网络深度的增加，模型训练可能会变得困难，主要原因是梯度消失和退化问题。为了解决这些问题，ResNet（残差网络）通过引入残差学习和快捷连接（skip connection）成功地解决了这一挑战。

ResNet 的基本思想是通过引入一个残差块（Residual Block），将输入信号和输出信号相加，形成“残差”进行学习。假设某一层的输入为 \mathbf{x} ，输出为 $\mathbf{H}(\mathbf{x})$ ，则残差块的输出可以表示为：

$$\mathbf{y} = \mathbf{H}(\mathbf{x}) + \mathbf{x}$$

这种结构允许网络学习到输入与输出之间的“残差”，而不是直接学习输出。通过这种方式，ResNet 能有效避免深层网络中的梯度消失和退化问题。公式中的 \mathbf{x} 为输入， $\mathbf{H}(\mathbf{x})$ 为由卷积层学习到的特征， \mathbf{y} 为残差块的输出。残差学习可以让信息通过直接的快捷连接流经网络，从而避免信息在深层网络中逐渐丢失。

4.3.3 ResNet18 架构

ResNet18 是 ResNet 系列网络中的一个较小版本，包含 18 层，其中包括多个残差块。每个残差块由两个卷积层组成，并通过跳跃连接将输入信号直接加到输出上。ResNet18 的主要结构如下：- 输入：32x32 的 RGB 图像（CIFAR100 数据集）。- 卷积层：使用多个 3x3 卷积核进行卷积操作。- 残差块：每个残差块由两层卷积构成，并通过跳跃连接将输入与输出相加。- 池化层：通过最大池化（Max Pooling）进行特征降维。- 最终的全连接层将特征映射到 100 个类别中。

ResNet18 的一个残差块结构可以用以下公式表示：

$$\mathbf{y} = \text{ReLU}(W_2 \cdot \text{ReLU}(W_1 \cdot \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) + \mathbf{x}$$

其中， \mathbf{x} 是输入， W_1 和 W_2 是卷积核， \mathbf{b}_1 和 \mathbf{b}_2 是偏置， \mathbf{y} 是输出，ReLU 是激活函数。这个结构的关键在于残差连接，即 \mathbf{x} 被直接加到卷积层的输出上。

4.3.4 CIFAR100 数据集

CIFAR100 是一个常用的图像分类数据集，包含 100 个类别，每个类别有 600 张 32x32 的 RGB 彩色图像。与 CIFAR10 相比，CIFAR100 的分类任务更加复杂，因为它有更多的类别，并且类别之间的差异较小。每个图像都是 32x32 像素，因此需要通过卷积神经网络来提取图像的局部特征。

CIFAR100 数据集包括如下的结构：- 100 个类别，每个类别包含 600 张图像。- 图像尺寸为 32x32 像素，RGB 三通道。- 数据集分为训练集和测试集，其中训练集包含 50000 张图像，测试集包含 10000 张图像。

4.3.5 训练过程与评估

在训练过程中，我使用交叉熵损失函数（Cross-Entropy Loss）作为模型的损失函数，用于衡量模型预测结果与真实标签之间的差距。交叉熵损失函数的公式如下：

$$L = - \sum_{i=1}^N y_i \log(p_i)$$

其中， N 是类别的数量（对于 CIFAR100， $N=100$ ）， y_i 是真实标签的概率（通常为 one-hot 编码），而 p_i 是模型预测的概率。通过最小化损失函数，优化模型的参数，使得预测结果越来越接近真实标签。

在评估过程中，我使用分类准确率（Accuracy）作为评估指标，即正确分类的样本数与总样本数的比值：

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

通过在测试集上评估模型的准确率，可以衡量模型的泛化能力。

4.4 实验步骤

4.4.1 数据来源

本实验使用 CIFAR100 数据集作为训练与测试数据。CIFAR100 数据集包含 100 个类别，每个类别有 600 张 32×32 的彩色图像。该数据集的类别涵盖了广泛的物体类型，包括动物、日常用品和自然景象。与 CIFAR10 数据集相比，CIFAR100 包含更多的类别，而且类别之间的区分较为微妙，给分类任务带来了更大的挑战。CIFAR100 数据集的官方来源为<https://www.cs.toronto.edu/~kriz/cifar.html>，可以通过 PyTorch 的 `torchvision.datasets.CIFAR100` 接口直接加载并进行使用。

在本实验中，ResNet18 模型被用来进行图像分类任务，评估模型在 CIFAR100 数据集上的表现。CIFAR100 数据集的训练集包含 50,000 张图像，测试集包含 10,000 张图像，这些图像的尺寸为 32×32 ，图像的多样性和复杂度增加了模型学习的难度。本实验通过 PyTorch 框架进行模型的训练和评估。

4.4.2 代码编写

本实验的代码基于 PyTorch 框架实现。首先，我加载了 CIFAR100 数据集并进行了标准化处理，确保数据具有相同的均值和标准差，以帮助模型更好地进行训练。在数据加载阶段，我使用了 PyTorch 的 `DataLoader` 进行批量数据的加载，并使用了标准化方法将图像的像素值归一化至 $[0, 1]$ 区间，并根据 CIFAR100 的统计特性进行了标准化（均值 $(0.4914, 0.4822, 0.4465)$ ，标准差 $(0.2470, 0.2435, 0.2616)$ ）。

在模型实现方面，我使用 PyTorch 中的 `nn.Module` 类构建了 ResNet18 模型。ResNet18 网络包含多个残差块，每个残差块包含两个卷积层，并通过跳跃连接（skip connection）将输入直接加到输出上，这有助于解决深层网络中的梯度消失问题。在训练过程中，我使用了交叉熵损失函数（`CrossEntropyLoss`）来衡量模型预测与真实标签之间的差距，同时使用 Adam 优化器来更新网络参数。

训练过程中，我记录了每个 epoch 的训练损失和训练准确率，另外，在每个 epoch 结束时，我使用测试集评估了模型的准确率。为了更好地分析模型的效果，我生成了混淆矩阵，并绘制了训练过程中的损失和准确率曲线。此外，还对部分测试图像进行了预测，并将真实标签与预测标签进行对比，帮助分析模型在不同类别上的表现。训练器代码：

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 import time
5 import matplotlib.pyplot as plt
6 import matplotlib
7
8 # 设置全局中文字体
9 plt.rcParams['font.family'] = 'Noto Sans CJK JP'
10 plt.rcParams['axes.unicode_minus'] = False # 避免负号乱码
11 import numpy as np
12 from tqdm import tqdm
13 import os
14
15 class Trainer:
16     def __init__(self, model, device, train_loader, test_loader, classes):
17         self.model = model
18         self.device = device
19         self.train_loader = train_loader
20         self.test_loader = test_loader
21         self.classes = classes
22
23     # 创建结果目录
24     os.makedirs('./Result', exist_ok=True)
25
26     # 损失函数和优化器
27     self.criterion = nn.CrossEntropyLoss()
28     self.optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-4)
29     self.scheduler = optim.lr_scheduler.ReduceLROnPlateau(
30         self.optimizer, 'min', patience=5, factor=0.5, verbose=True
31     )
32
33     # 训练历史记录
34     self.train_losses = []
35     self.test_losses = []
36     self.train_accuracies = []
37     self.test_accuracies = []
38
39     def train_epoch(self):
40         self.model.train()
41         running_loss = 0.0
42         correct = 0
43         total = 0
44
45         # 使用tqdm创建进度条
46         pbar = tqdm(self.train_loader, desc='Training')
47
48         for inputs, targets in pbar:
49             inputs, targets = inputs.to(self.device), targets.to(self.device)
50
51             # 梯度清零
```

```

52     self.optimizer.zero_grad()
53
54     # 前向传播
55     outputs = self.model(inputs)
56     loss = self.criterion(outputs, targets)
57
58     # 反向传播和优化
59     loss.backward()
60     self.optimizer.step()
61
62     # 统计
63     running_loss += loss.item()
64     _, predicted = outputs.max(1)
65     total += targets.size(0)
66     correct += predicted.eq(targets).sum().item()
67
68     # 更新进度条
69     pbar.set_postfix({'loss': loss.item(), 'acc': 100.*correct/total})
70
71     train_loss = running_loss / len(self.train_loader)
72     train_accuracy = 100. * correct / total
73
74     return train_loss, train_accuracy
75
76 def test(self):
77     self.model.eval()
78     running_loss = 0.0
79     correct = 0
80     total = 0
81
82     with torch.no_grad():
83         for inputs, targets in tqdm(self.test_loader, desc='Testing'):
84             inputs, targets = inputs.to(self.device), targets.to(self.device)
85
86             # 前向传播
87             outputs = self.model(inputs)
88             loss = self.criterion(outputs, targets)
89
90             # 统计
91             running_loss += loss.item()
92             _, predicted = outputs.max(1)
93             total += targets.size(0)
94             correct += predicted.eq(targets).sum().item()
95
96     test_loss = running_loss / len(self.test_loader)
97     test_accuracy = 100. * correct / total
98
99     return test_loss, test_accuracy
100
101 def train(self, epochs=50):
102     print(f"开始训练，总共{epochs}个epochs...")
103     start_time = time.time()
104
105     best_acc = 0.0  # 记录最佳准确率
106
107     for epoch in range(epochs):
108         print(f"\nEpoch {epoch+1}/{epochs}")

```

```

109
110     # 训练一个 epoch
111     train_loss, train_accuracy = self.train_epoch()
112     self.train_losses.append(train_loss)
113     self.train_accuracies.append(train_accuracy)
114
115     # 在测试集上评估
116     test_loss, test_accuracy = self.test()
117     self.test_losses.append(test_loss)
118     self.test_accuracies.append(test_accuracy)
119
120     # 学习率调度器
121     self.scheduler.step(test_loss)
122
123     print(f"训练损失: {train_loss:.4f}, 训练准确率: {train_accuracy:.2f}%")
124     print(f"测试损失: {test_loss:.4f}, 测试准确率: {test_accuracy:.2f}%")
125
126     # 保存最佳模型
127     if test_accuracy > best_acc:
128         best_acc = test_accuracy
129         self.save_model('./Result/cifar100_resnet18_best.pth')
130         print(f"保存最佳模型, 准确率: {best_acc:.2f}%")
131
132     total_time = time.time() - start_time
133     print(f"\n训练完成! 总用时: {total_time:.2f}秒")
134     print(f"最佳测试准确率: {best_acc:.2f}%")
135
136     # 返回训练历史记录
137     history = {
138         'train_loss': self.train_losses,
139         'test_loss': self.test_losses,
140         'train_accuracy': self.train_accuracies,
141         'test_accuracy': self.test_accuracies
142     }
143
144     return history
145
146 def plot_history(self, history):
147     plt.figure(figsize=(12, 5))
148
149     # 绘制损失曲线
150     plt.subplot(1, 2, 1)
151     plt.plot(history['train_loss'], label='训练损失')
152     plt.plot(history['test_loss'], label='测试损失')
153     plt.xlabel('Epoch')
154     plt.ylabel('损失')
155     plt.legend()
156     plt.title('训练和测试损失')
157
158     # 绘制准确率曲线
159     plt.subplot(1, 2, 2)
160     plt.plot(history['train_accuracy'], label='训练准确率')
161     plt.plot(history['test_accuracy'], label='测试准确率')
162     plt.xlabel('Epoch')
163     plt.ylabel('准确率 (%)')
164     plt.legend()
165     plt.title('训练和测试准确率')

```

```

166
167     plt.tight_layout()
168     plt.savefig('./Result/training_history.png')
169     plt.close() # 关闭图像而不是显示
170
171 def visualize_predictions(self, num_samples=5):
172     """
173         可视化模型在测试集上的一些预测结果
174
175     参数：
176         num_samples: 要可视化的样本数量
177     """
178     self.model.eval()
179
180     # 获取一批测试数据
181     dataiter = iter(self.test_loader)
182     images, labels = next(dataiter)
183
184     # 选择前 num_samples 个样本
185     images = images[:num_samples]
186     labels = labels[:num_samples]
187
188     # 预测
189     with torch.no_grad():
190         outputs = self.model(images.to(self.device))
191         _, predicted = torch.max(outputs, 1)
192         predicted = predicted.cpu().numpy()
193
194     # 转换图像用于显示
195     images = images.cpu().numpy()
196
197     # 显示图像和预测结果
198     fig, axes = plt.subplots(1, num_samples, figsize=(15, 3))
199
200     for i in range(num_samples):
201         # 反归一化图像
202         img = np.transpose(images[i], (1, 2, 0))
203         mean = np.array([0.5071, 0.4867, 0.4408])
204         std = np.array([0.2675, 0.2565, 0.2761])
205         img = std * img + mean
206         img = np.clip(img, 0, 1)
207
208         axes[i].imshow(img)
209         axes[i].set_title(f"真实: {self.classes[labels[i]]}\n预测: {self.classes[predicted[i]]}")
210         axes[i].axis('off')
211
212     plt.tight_layout()
213     plt.savefig('./Result/predictions.png')
214     plt.close() # 关闭图像而不是显示
215
216 def save_model(self, path='./Result/cifar100_resnet18_model.pth'):
217     # 确保目录存在
218     os.makedirs(os.path.dirname(path), exist_ok=True)
219
220     torch.save({
221         'model_state_dict': self.model.state_dict(),

```

```

222         'optimizer_state_dict': self.optimizer.state_dict(),
223     }, path)
224     print(f"模型已保存到 {path}")
225
226     def load_model(self, path='./Result/cifar100_resnet18_model.pth'):
227         checkpoint = torch.load(path)
228         self.model.load_state_dict(checkpoint['model_state_dict'])
229         self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
230         print(f"模型已从 {path} 加载")

```

数据导入代码:

```

1 import torch
2 import torchvision
3 import torchvision.transforms as transforms
4
5 def get_data_loaders(batch_size=128, num_workers=2):
6     # 数据预处理和增强
7     transform_train = transforms.Compose([
8         transforms.RandomCrop(32, padding=4),
9         transforms.RandomHorizontalFlip(),
10        transforms.ToTensor(),
11        transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
12    ])
13
14    transform_test = transforms.Compose([
15        transforms.ToTensor(),
16        transforms.Normalize((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761))
17    ])
18
19    # 加载训练集
20    trainset = torchvision.datasets.CIFAR10(
21        root='./Data',
22        train=True,
23        download=True,
24        transform=transform_train
25    )
26    trainloader = torch.utils.data.DataLoader(
27        trainset,
28        batch_size=batch_size,
29        shuffle=True,
30        num_workers=num_workers
31    )
32
33    # 加载测试集
34    testset = torchvision.datasets.CIFAR10(
35        root='./Data',
36        train=False,
37        download=True,
38        transform=transform_test
39    )
40    testloader = torch.utils.data.DataLoader(
41        testset,
42        batch_size=batch_size,
43        shuffle=False,
44        num_workers=num_workers
45    )
46

```

```

47     # 获取CIFAR100类别名称
48     classes = trainset.classes
49
50     return trainloader, testloader, classes

```

模型代码:

```

1 import torch
2 import torch.nn as nn
3 import torchvision.models as models
4
5 class ResNet18ForCIFAR100(nn.Module):
6
7     def __init__(self, pretrained=True):
8         super(ResNet18ForCIFAR100, self).__init__()
9
10        # 加载预训练的ResNet18模型
11        model = models.resnet18(pretrained=pretrained)
12
13        # 修改第一个卷积层以适应CIFAR100的32x32输入
14        # 原始ResNet设计用于224x224的ImageNet图像
15        model.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
16        model.maxpool = nn.Identity() # 移除最大池化层，因为CIFAR100图像较小
17
18        # 修改最后的全连接层以输出100个类别(CIFAR100的类别数)
19        num_ftrs = model.fc.in_features
20        model.fc = nn.Linear(num_ftrs, 100)
21
22        self.model = model
23
24    def forward(self, x):
25        return self.model(x)

```

完整的模型代码:

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 # Basic Residual Block for ResNet-18/34
6 class BasicBlock(nn.Module):
7     expansion = 1 # 用于ResNet-50及以上，Bottleneck需要用
8
9     def __init__(self, in_channels, out_channels, stride=1, downsample=None):
10        super(BasicBlock, self).__init__()
11        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
12                            stride=stride, padding=1, bias=False)
13        self.bn1 = nn.BatchNorm2d(out_channels)
14        self.relu = nn.ReLU(inplace=True)
15
16        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
17                            stride=1, padding=1, bias=False)
18        self.bn2 = nn.BatchNorm2d(out_channels)
19
20        self.downsample = downsample # 用于匹配维度（如通道数不同或stride不为1）
21
22    def forward(self, x):
23        identity = x

```

```

24
25     out = self.conv1(x)
26     out = self.bn1(out)
27     out = self.relu(out)
28
29     out = self.conv2(out)
30     out = self.bn2(out)
31
32     # 如果需要下采样或通道对不上
33     if self.downsample is not None:
34         identity = self.downsample(x)
35
36     out += identity # 残差连接
37     out = self.relu(out)
38
39     return out
40
41 # ResNet主结构
42 class ResNet(nn.Module):
43     def __init__(self, block, layers, num_classes=100):
44         super(ResNet, self).__init__()
45         self.in_channels = 64
46
47         # 第一层：适配 CIFAR (32x32)，不使用7x7卷积和最大池化
48         self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
49         self.bn1 = nn.BatchNorm2d(64)
50         self.relu = nn.ReLU(inplace=True)
51
52         # 残差层：每层调用 self._make_layer
53         self.layer1 = self._make_layer(block, 64, layers[0], stride=1)
54         self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
55         self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
56         self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
57
58         self.avgpool = nn.AdaptiveAvgPool2d((1, 1)) # 自适应池化
59         self.fc = nn.Linear(512 * block.expansion, num_classes)
60
61     def _make_layer(self, block, out_channels, blocks, stride):
62         """构建一个 layer (多个 block) """
63         downsample = None
64
65         # 如果输入输出通道不匹配，或 stride == 1，需要downsample
66         if stride != 1 or self.in_channels != out_channels * block.expansion:
67             downsample = nn.Sequential(
68                 nn.Conv2d(self.in_channels, out_channels * block.expansion,
69                         kernel_size=1, stride=stride, bias=False),
70                 nn.BatchNorm2d(out_channels * block.expansion)
71             )
72
73         layers = []
74         layers.append(block(self.in_channels, out_channels, stride, downsample))
75         self.in_channels = out_channels * block.expansion
76
77         for _ in range(1, blocks):
78             layers.append(block(self.in_channels, out_channels))
79
80         return nn.Sequential(*layers)

```

```

81
82     def forward(self, x):
83         x = self.conv1(x)
84         x = self.bn1(x)
85         x = self.relu(x)
86
87         x = self.layer1(x)    # 32x32
88         x = self.layer2(x)    # 16x16
89         x = self.layer3(x)    # 8x8
90         x = self.layer4(x)    # 4x4
91
92         x = self.avgpool(x)   # (N, 512, 1, 1)
93         x = torch.flatten(x, 1)  # (N, 512)
94         x = self.fc(x)      # (N, 100)
95
96     return x
97
98 # ResNet-18 构建函数
99 def ResNet18ForCIFAR100():
100     return ResNet(BasicBlock, [2, 2, 2, 2], num_classes=100)

```

4.4.3 实验结果

在本实验中，我记录了训练损失和准确率的变化曲线，并在测试集上评估了模型的最终性能。实验结果表明，ResNet18 模型在 CIFAR100 数据集上的表现较为优秀，尤其是在一些容易区分的类别上，如“wardrobe”，“bicycle”，“orange”等，准确率均超过 90%。具体来说，“wardrobe”类别的准确率为 91%，而“bicycle”类别的准确率为 91%。这些结果表明，ResNet18 模型能够有效学习到这些类别的特征。

然而，在某些类别间，模型表现较弱，尤其是在类别之间差异较小的情况下。例如，模型在“baby”与“bear”这两个类别之间的准确率较低，分别为 63% 和 58%。这表明，在具有较高相似性的类别之间，模型的区分能力尚未达到理想状态。在“beaver”与“bed”这两个类别之间也出现了一定程度的混淆。这种误分类现象可能与这些类别的图像特征相似性有关。

从训练过程中的损失和准确率变化曲线来看（图 14），训练损失在前几个 epoch 内快速下降，说明模型在训练初期学习较快。随着训练的进行，损失值逐渐趋于稳定，训练准确率也在前期迅速上升，最终在训练结束时达到 99%。然而，与训练准确率相比，测试集准确率的提升较为缓慢，最终稳定在 73.6%，这表明模型在训练集上表现出色，但在测试集上仍有改进空间，可能是由于模型出现了过拟合现象。

通过生成的混淆矩阵（图 11），我可以看到，模型在一些类别之间的混淆情况。例如，类别“baby”与“bear”之间的混淆较为明显，模型容易将“baby”误判为“bear”。此外，其他类别之间的混淆较小，表明模型能够较好地处理其他类别的分类任务。

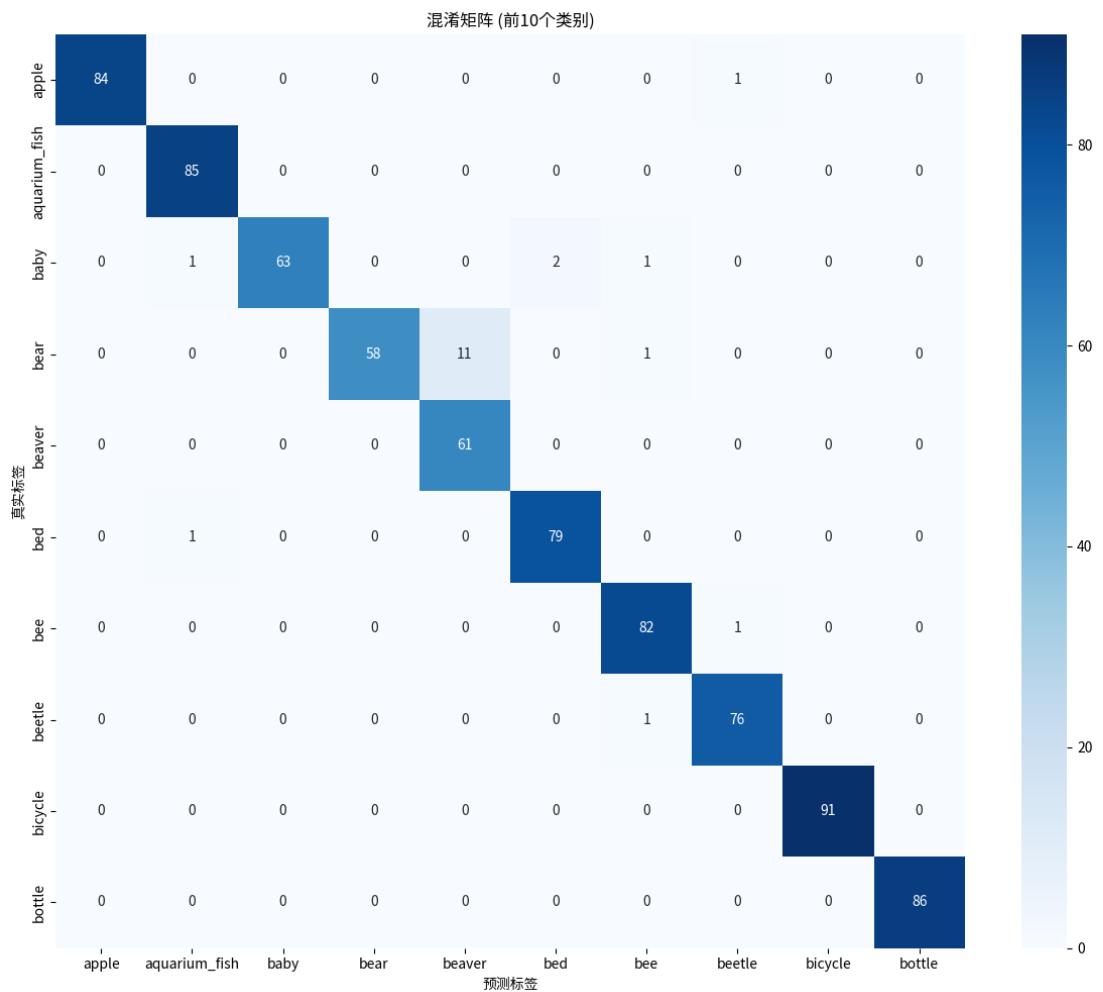


图 11: 混淆矩阵 (前 20 个和后 20 个类别)



图 12: 真实标签与预测标签对比

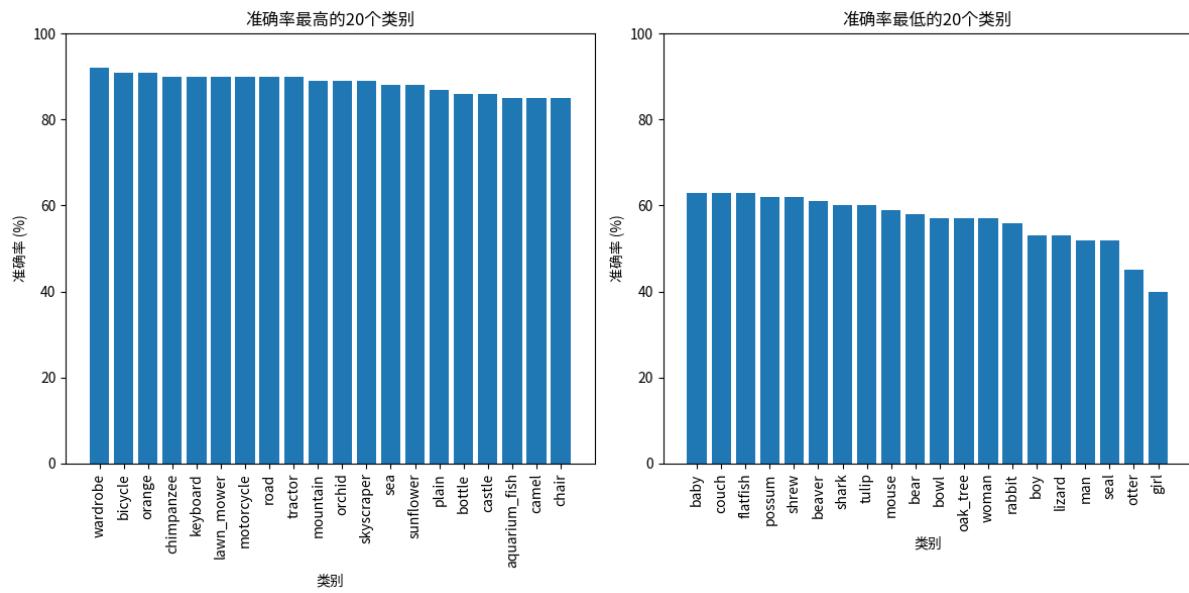


图 13: 每个类别的准确率

此外，训练过程中的损失和准确率曲线（图 14）显示，模型在训练过程中逐渐收敛，训练损失不断降低，训练准确率逐渐提高，说明模型能够有效地学习并拟合训练数据。

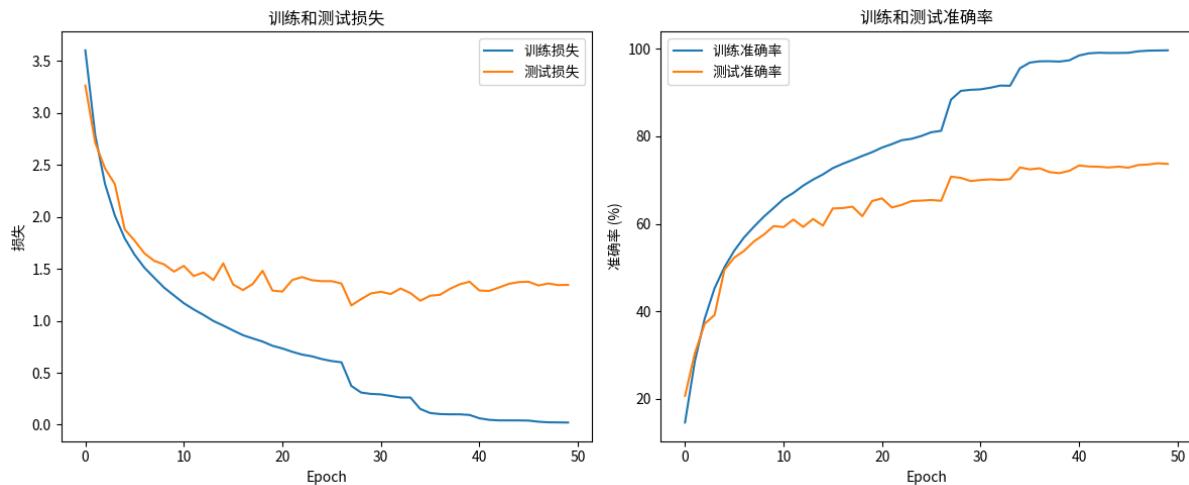


图 14: 训练和测试损失与准确率

最后，图 12展示了部分测试图像的预测结果，展示了模型对这些图像的预测准确性。尽管部分图像存在误判，如“mountain”误判为“road”，“seal”误判为“beaver”，但是整体上模型仍能较为准确地对大多数图像进行分类。

```

PROBLEMS OUTPUT PORTS TERMINAL DEBUG CONSOLE
○ [yvittu] [base] py@42380:~/CV-Class/Project4$ python ./Code/main.py
使用 GPU: NVIDIA GeForce RTX 3090
加载 CIFAR100 数据集...
Files already downloaded and verified
File already downloaded and verified
数据加载完成...
类别数量: 100
创建 ResNet18 模型...
模型已保存到: ./Result/cifar100_resnet18.pth
开始训练模型...
开始训练, 总共50个epochs...
Epoch 1/50
Training: 100% | 391/391 [00:28<00:00, 19.46it/s, loss=3.14, acc=14.6]
Testing: 100% | 79/79 [00:01<00:00, 50.66it/s]
训练损失: 3.14%, 测试准确率: 14.5%
测试损失: 4.62%, 测试准确率: 20.6%
模型已保存到: ./Result/cifar100_resnet18_best.pth
保存最佳模型, 准确率: 20.6%
Epoch 2/50
Training: 100% | 391/391 [00:28<00:00, 19.41it/s, loss=3.14, acc=14.6]
Testing: 100% | 79/79 [00:01<00:00, 50.66it/s]
训练损失: 3.14%, 测试准确率: 14.5%
测试损失: 4.62%, 测试准确率: 20.6%
模型已保存到: ./Result/cifar100_resnet18_best.pth
保存最佳模型, 准确率: 20.6%
Epoch 3/50
Training: 100% | 391/391 [00:28<00:00, 19.41it/s, loss=3.14, acc=14.6]
Testing: 100% | 79/79 [00:01<00:00, 50.66it/s]
训练损失: 3.14%, 测试准确率: 14.5%
测试损失: 4.62%, 测试准确率: 20.6%
模型已保存到: ./Result/cifar100_resnet18_best.pth
保存最佳模型, 准确率: 20.6%
Epoch 4/50
Training: 100% | 391/391 [00:28<00:00, 19.33it/s, loss=0.025, acc=99.6]
Testing: 100% | 79/79 [00:01<00:00, 47.52it/s]
训练损失: 0.025, 测试准确率: 99.57%
测试损失: 0.025, 测试准确率: 73.55%
模型已保存到: ./Result/cifar100_resnet18_best.pth
保存最佳模型, 准确率: 73.55%
Epoch 5/50
Training: 100% | 391/391 [00:28<00:00, 19.14it/s, loss=0.027, acc=99.6]
Testing: 100% | 79/79 [00:01<00:00, 47.16it/s]
训练损失: 0.027, 测试准确率: 99.65%
测试损失: 0.027, 测试准确率: 73.69%
模型已保存到: ./Result/cifar100_resnet18_best.pth
保存最佳模型, 准确率: 73.69%
Epoch 6/50
Training: 100% | 391/391 [00:28<00:00, 18.76it/s, loss=0.00788, acc=99.7]
Testing: 100% | 79/79 [00:01<00:00, 47.29it/s]
训练损失: 0.00788, 测试准确率: 99.65%
测试损失: 0.00788, 测试准确率: 73.69%
模型已保存到: ./Result/cifar100_resnet18_best.pth
保存最佳模型, 准确率: 73.69%
Top-5 准确率: 92.1%
每个类别的准确率:
wardrobe: 92.00%
bicycle: 91.00%
orange: 91.00%
chimpanzee: 90.00%
keyboard: 90.00%
lawn_mower: 90.00%
motorcycle: 90.00%
road: 90.00%
tractor: 90.00%
mountain: 89.00%
orchid: 89.00%
syzygium: 89.00%
sea: 88.00%
sunflower: 88.00%
plain: 87.00%
bottle: 86.00%
castle: 86.00%
aquarium_fish: 85.00%
camel: 85.00%
chart: 85.00%
pickup_truck: 85.00%
raccoon: 85.00%
train: 85.00%
apple: 84.00%
bridge: 84.00%
cloud: 83.00%
palm_tree: 83.00%
skunk: 83.00%
television: 83.00%
bee: 82.00%
poppy: 82.00%
tank: 82.00%
cockroach: 81.00%
hamster: 81.00%
pear: 80.00%
streetcar: 80.00%
bed: 79.00%
leopard: 78.00%
spider: 78.00%
tiger: 78.00%
trout: 78.00%
cup: 78.00%
rocket: 77.00%
beetle: 76.00%
can: 76.00%
dinosaur: 76.00%
lion: 76.00%
rose: 76.00%
wolf: 76.00%
fox: 75.00%
parrot: 75.00%
bus: 74.00%
house: 74.00%
plate: 74.00%
clock: 73.00%
worm: 73.00%
while: 72.00%
elephant: 71.00%
forest: 70.00%
kangaroo: 71.00%
mushroom: 71.00%
butterfly: 70.00%
dogfish: 70.00%
telephone: 69.00%
caterpillar: 68.00%
snake: 68.00%
catfish: 67.00%
pine_tree: 67.00%
table: 67.00%
willow_tree: 67.00%
crab: 66.00%
maple_leaf: 64.00%
baby: 63.00%
couch: 63.00%
flatfish: 63.00%
possum: 62.00%
sheep: 61.00%
beaver: 61.00%
shark: 60.00%
tulip: 60.00%
moses: 60.00%
bear: 58.00%
bow: 57.00%
oak_tree: 57.00%
woman: 57.00%
rainbow: 55.00%
boy: 55.00%
Lizard: 53.00%
man: 52.00%
seal: 52.00%
otter: 45.00%
girl: 40.00%

```

图 15: 实验过程示意图

4.5 思考与改进方向

尽管本次实验中的 ResNet18 模型在 CIFAR100 数据集上的分类精度已经达到了较为理想的水平，但从实验结果中我也可以看出，仍然存在可以改进的地方。实验中揭示了模型在某些类别之间的混淆、过拟合现象以及一些其他潜在问题。因此，为了进一步提升模型的分类精度和泛化能力，我可以考虑以下几个方面的改进。

4.5.1 改进数据增强

在本实验中，我对图像数据进行了一些基础的数据增强处理，例如随机裁剪和水平翻转。然而，数据增强的目的是使模型能够见到更多样化的样本，从而提升其对不同输入图像的适应能力。根据实验结果来看，模型在一些难以区分的类别（如“baby”和“bear”）之间的准确率仍然较低，这表明模型对这类相似类别的区分能力仍有不足。在这种情况下，可以考虑进一步丰富数据增强的方法。例如，除了随机裁剪和翻转，我可以加入旋转、平移、颜色抖动、随机遮挡等方法。旋转和翻转能够有效增加图像的多样性，从而减少模型在处理特定方向的物体时的偏差；而颜色抖动则能帮助模型学习到更多的颜色特征，增强模型的鲁棒性。通过这些方式，我能够让模型在训练过程中见到更多变形和不同状态的图像，进而提高它在复杂环境中的表现。此外，在实验中使用图像缩放、模糊处理或背景变化等增强方式，能够进一步提高模型的适应能力，减少过拟合，增强其泛化能力。

4.5.2 优化网络结构

尽管 ResNet18 模型通过引入残差学习和跳跃连接解决了深层网络中的梯度消失和退化问题，但在处理 CIFAR100 这样多类别且类别差异较小的数据集时，ResNet18 的模型深度可能不足以有效提取所有层次的图像特征。因此，未来可以考虑使用更深层的 ResNet 变种，如 ResNet50 或 ResNet101，这些更深的网络能够提供更丰富的特征表示，有助于提升模型对复杂数据的学习能力。更深层的网络通过引入更多的残差模块，能够在更高层次上提取图像特征，避免在深层网络中信息丢失，提升对类别间细微差异的区分能力。此外，考虑到 ResNet 架构的成功，其他类型的网络架构，如 DenseNet 或 Inception，也值得尝试。DenseNet 通过密集连接（Dense Connection）使得每一层都能够接收到来自前面所有层的信息，从而加强特征的传递并减少信息丢失。而 Inception 网络则通过多个卷积核的组合，从不同尺度上提取特征，能够提高对复杂图像的理解。通过这些网络架构的优化，模型能够处理更加复杂的图像特征，进一步提升分类精度。

4.5.3 超参数优化

在本次实验中，尽管使用了标准的 Adam 优化器和固定的学习率策略，但超参数的选择在模型性能中起着至关重要的作用。不同的学习率、批量大小等超参数设置可能会显著影响模型的收敛速度和最终的分类精度。例如，在本次实验中，测试集上的准确率停滞在 73.6% 附近，尽管训练集上的准确率接近 99%，这表明模型可能存在过拟合现象。为了解决这一问题，可以尝试更复杂的学习率调度方法，例如余弦退火（Cosine Annealing）或学习率衰减（Learning Rate Decay）。这些方法通过动态调整学习率，在训练初期快速下降，后期逐步减小，从而让模型能够在后期训练过程中以更细的步伐收敛到最优解。另外，超参数优化不仅限于学习率，还包括批量大小、正则化项的权重、训练轮次等。通过自动化超参数优化方法，如网格搜索、贝叶斯优化或随机搜索等，可以系统地调整超参数，找到最适合的配置，从而进一步提升模型性能。

4.5.4 更多评估指标

在本次实验中，我主要通过分类准确率来评估模型的性能。然而，准确率并不能全面反映模型的表现，尤其是在数据集不平衡的情况下。一些类别可能占据大部分数据样本，导致高准确率的模型在其他类别上的表现不佳。因此，在未来的实验中，除了准确率外，还需要增加其他评估指标，如 F1-score、精确率 (Precision)、召回率 (Recall) 等。这些评估指标可以提供更为全面的模型性能分析，特别是在类别不平衡的情况下，F1-score 作为精确率和召回率的调和平均，可以帮助我更好地衡量模型在各个类别的分类效果。此外，召回率能够帮助我了解模型在捕获所有相关样本方面的能力，而精确率则帮助我衡量模型输出的预测样本中，真正为正确的比例。在测试过程中，加入这些评估指标能够更准确地反映模型的实际表现，尤其是在一些难以区分的类别之间，进一步推动模型性能的提升。

总体来说，尽管 ResNet18 模型在本次实验中已经取得了相对较好的结果，但仍然存在一定的优化空间。从数据增强、网络架构优化、超参数调节到评估指标的扩展，每一个改进方向都可能进一步提升模型的分类精度和泛化能力。通过综合这些改进策略，模型能够更好地应对 CIFAR100 数据集中的挑战，并在其他复杂的分类任务中取得更好的表现。

5 实验五：语义分割

5.1 实验目的

本实验旨在使用 PyTorch 框架实现一个基于 UNet 模型的语义分割任务，目标是对 PASCAL VOC 数据集进行训练，并输出其在测试集上的语义分割精度。语义分割是计算机视觉中的一项重要任务，其目标是将图像中的每个像素分类为一个特定的标签，这对于自动驾驶、医学图像分析、遥感图像分析等领域有着广泛的应用。通过本实验，旨在掌握语义分割的基本原理与技术，特别是如何使用深度学习模型（如 UNet）来进行像素级的图像分割。实验还将帮助我理解如何使用深度神经网络，特别是卷积神经网络（CNN），来处理像素级的分类问题，并应用于实际的图像处理任务中。

5.2 实验要求

本实验使用 PASCAL VOC 数据集作为训练和测试数据，并使用 UNet 模型进行语义分割任务。首先，实验要求能够下载并处理 PASCAL VOC 数据集，将其划分为训练集和测试集。其次，实验要求使用 PyTorch 框架实现 UNet 模型并进行训练，UNet 模型广泛应用于医学图像分割和其他像素级分类任务，它通过其对称的编码器-解码器结构，能够有效地捕捉上下文信息并恢复空间细节。训练过程中，需要使用适当的损失函数，如交叉熵损失函数，并在训练集上进行优化以最小化损失，最终在测试集上评估模型的语义分割精度。最后，实验要求输出模型在测试集上的分割精度，并进行分析与讨论，特别是模型在不同类别上的表现差异。

5.3 实验原理

5.3.1 语义分割概述

语义分割是计算机视觉中的一项任务，旨在对输入图像中的每一个像素进行分类。与传统的图像分类任务不同，语义分割要求模型为每个像素分配一个类别标签，从而实现图像的像素级分类。其

应用领域非常广泛，包括自动驾驶中的道路分割、医学图像中的肿瘤检测、遥感图像分析等。语义分割不仅要求高精度的分类能力，还要能够在像素级别上恢复图像的空间细节，因此需要较为复杂的网络架构来进行处理。

在语义分割任务中，卷积神经网络（CNN）通常被用于特征提取，并且使用了多个卷积层来捕捉不同层次的图像信息。为了提高模型对空间信息的捕捉能力，通常会在网络中加入上采样层（用于恢复空间分辨率）和跳跃连接（用于保留低层次特征）。在本实验中，我使用了 UNet 模型，该模型专门为语义分割任务设计，具有优越的性能。

5.3.2 UNet 模型结构

UNet 模型是一种典型的语义分割网络，它由编码器（下采样部分）和解码器（上采样部分）组成。其主要特点是通过跳跃连接（skip connection）将编码器中的特征图直接传递给解码器，这样可以有效地保留输入图像的低级特征，帮助解码器恢复更精细的空间信息。

UNet 的网络结构可以用以下公式表示：

$$y = f_{\text{decode}}(f_{\text{encode}}(x))$$

其中， x 是输入图像， f_{encode} 表示编码器部分的卷积和池化操作， f_{decode} 是解码器部分的上采样和卷积操作， y 是网络的输出，即语义分割结果。编码器部分逐步提取图像的特征，并通过池化层减少图像的空间维度；解码器部分则通过上采样操作逐步恢复图像的空间分辨率，最终生成与输入图像相同大小的分割图像。

UNet 的核心在于其跳跃连接，它将每一层的输出与解码器对应层的输出相连接，这样可以帮助模型在上采样过程中恢复更多的空间细节信息。这种结构大大提高了分割结果的精细度。

5.3.3 损失函数与优化

在语义分割任务中，常用的损失函数是交叉熵损失函数（Cross-Entropy Loss）。交叉熵损失函数通过计算每个像素的预测标签与真实标签之间的差异来衡量模型的表现。交叉熵损失函数的公式如下：

$$L = - \sum_{i=1}^N (y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

其中， N 是像素的总数， y_i 是真实标签（0 或 1，表示当前像素所属的类别）， p_i 是模型预测的概率值。该损失函数通过最小化每个像素的预测与真实标签之间的差异，来优化模型的参数。

除了交叉熵损失之外，还可以使用 Dice 系数作为模型评估指标。Dice 系数用于衡量模型预测区域与真实区域的重叠程度，它的公式为：

$$\text{Dice} = \frac{2|A \cap B|}{|A| + |B|}$$

其中， A 是预测区域， B 是真实区域， $|A \cap B|$ 是预测区域和真实区域的交集， $|A|$ 和 $|B|$ 分别是预测区域和真实区域的面积。Dice 系数的值越高，表示模型的预测越准确。

在训练过程中，我使用 Adam 优化器来最小化损失函数，更新网络的权重。Adam 优化器结合了动量和自适应学习率，使得优化过程更加高效，并能够在训练中快速收敛。

5.3.4 PASCAL VOC 数据集

PASCAL VOC (Visual Object Classes) 是计算机视觉领域中常用的图像数据集，广泛用于物体检测、语义分割、图像分类等任务。PASCAL VOC 数据集包含 20 个物体类别，涵盖了动物、交通工具、家具等多种类型的物体。每张图像都标注了物体的边界框和像素级的分割标签，这使得它成为一个理想的语义分割训练数据集。

在本实验中，我使用的是 PASCAL VOC 2012 版本的数据集，它包含了训练集、验证集和测试集，图像的分辨率为 500×375 像素。每个图像都配有相应的语义标签，标注了每个像素所属的类别。在语义分割任务中，我需要使用这些标签来训练模型，使得模型能够预测每个像素的类别，从而得到像素级的图像分割结果。

PASCAL VOC 数据集在语义分割任务中是一个挑战性的标准数据集，尤其是对于复杂场景中的物体，模型需要能够精确地分割出物体的边界并避免背景的干扰。

5.3.5 模型评估

在训练和测试过程中，评估模型的性能是至关重要的。除了损失函数外，我还使用 IoU (Intersection over Union) 和像素精度来评估模型的语义分割精度。IoU 是最常用的评估指标，它衡量了预测区域与真实区域之间的重叠程度，公式为：

$$\text{IoU} = \frac{\text{Intersection}}{\text{Union}} = \frac{|A \cap B|}{|A \cup B|}$$

其中， A 和 B 分别表示预测区域和真实区域，IoU 越大表示分割效果越好。此外，像素精度也是一个重要的评估指标，它计算了正确分类的像素占总像素的比例，公式为：

$$\text{Pixel Accuracy} = \frac{\text{Number of Correctly Classified Pixels}}{\text{Total Number of Pixels}}$$

这些评估指标帮助我全面了解模型在语义分割任务中的表现，特别是在不同类别的分割精度上。

5.4 实验步骤

5.4.1 数据来源

本实验使用了 PASCAL VOC 数据集进行语义分割任务。PASCAL VOC 数据集是计算机视觉领域中广泛使用的图像分割数据集，包含 20 个类别以及一个背景类别，适用于图像分割、物体检测等任务。数据集包含训练集、验证集和测试集，其中训练集包含约 1,400 张图像，验证集包含 1449 张图像，测试集包含 1456 张图像。在本实验中，我主要使用了 PASCAL VOC 2012 版本数据集，它包含的类别适合于训练语义分割模型，模型需要为每个像素分配一个标签。数据集的每张图像都有一个对应的像素级别的标签，这为模型训练提供了充分的监督信号。

5.4.2 代码编写

本实验使用 PyTorch 框架实现了 UNet 模型进行语义分割任务。UNet 模型是基于编码器-解码器结构设计的网络，主要用于语义分割任务。编码器部分通过卷积操作提取图像的特征并通过池化逐渐降低分辨率，而解码器部分则通过反卷积（上采样）恢复图像的空间分辨率。编码器和解码器之间的跳跃连接（skip connections）帮助解码器恢复丢失的细节信息。

在代码实现方面，我首先加载 PASCAL VOC 数据集并对图像进行标准化处理。使用数据增强技术（如随机裁剪、旋转、翻转等）增强数据集的多样性，以提高模型的鲁棒性。训练过程中，采用交叉熵损失函数（Cross-Entropy Loss）作为模型的损失函数，并使用 Adam 优化器进行模型优化。为了避免过拟合，在训练过程中使用了 Dropout 层，增强模型的泛化能力。训练周期较长，但由于计算资源限制，本次实验只训练了两轮，并记录了相关的验证指标。

数据集导入代码：

```
1 import os
2 import torch
3 from torch.utils.data import Dataset, DataLoader
4 from torchvision import transforms
5 from PIL import Image
6 import numpy as np
7 from torchvision.transforms import InterpolationMode
8 # PASCAL VOC 数据集的颜色映射 (RGB格式)
9 PALETTE = [
10     [0, 0, 0],          # 背景
11     [128, 0, 0],        # 飞机
12     [0, 128, 0],        # 汽车
13     [128, 128, 0],      # 汽船
14     [0, 0, 128],        # 鸟
15     [128, 0, 128],      # 猫
16     [0, 128, 128],      # 奶牛
17     [128, 128, 128],    # 狗
18     [64, 0, 0],          # 马
19     [192, 0, 0],         # 羊
20     [64, 128, 0],        # 牛
21     [192, 128, 0],       # 飞行器
22     [64, 0, 128],        # 车辆
23     [192, 0, 128],       # 卡车
24     [64, 128, 128],      # 动物
25     [192, 128, 128],     # 建筑
26     [0, 64, 0],           # 运输工具
27     [128, 64, 0],         # 水果
28     [0, 64, 128],         # 工业
29     [128, 64, 128]       # 其他
30 ]
31
32 # Pascal VOC数据集加载器 - 修复后的版本
33 class VOCSegmentationDataset(Dataset):
34     def __init__(self, root_dir, split='train', transform=None):
35         self.root_dir = root_dir
36         self.split = split
37         self.transform = transform
38         self.images_dir = os.path.join(root_dir, 'JPEGImages')
39         self.masks_dir = os.path.join(root_dir, 'SegmentationClass')
40
41         # 获取文件列表
42         split_file = os.path.join(root_dir, 'ImageSets', 'Segmentation', f'{split}.txt')
43         self.file_names = [line.strip() for line in open(split_file, 'r')]
44
45     def __len__(self):
46         return len(self.file_names)
47
48     def __getitem__(self, idx):
49         img_name = self.file_names[idx]
```

```

50     img_path = os.path.join(self.images_dir, img_name + '.jpg')
51     mask_path = os.path.join(self.masks_dir, img_name + '.png')
52
53     image = Image.open(img_path).convert('RGB')
54     mask = Image.open(mask_path).convert('RGB')
55
56     # 应用相同的空间变换到图像和掩码
57     if self.transform:
58         # 对图像应用完整的 transform
59         image = self.transform(image)
60
61         # 对掩码只应用空间变换 (Resize)，使用新的插值模式
62         mask = transforms.Resize((256, 256), interpolation=InterpolationMode.NEAREST)(mask)
63         mask = self.rgb_to_index(np.array(mask))
64         mask = torch.from_numpy(mask).long()
65     else:
66         # 如果没有 transform，只进行基本处理
67         mask = self.rgb_to_index(np.array(mask))
68         mask = torch.from_numpy(mask).long()
69
70     return image, mask
71
72 def rgb_to_index(self, mask_rgb):
73     """将RGB掩码转换为类别索引图"""
74     h, w, _ = mask_rgb.shape
75     mask_index = np.zeros((h, w), dtype=np.uint8)
76
77     # 创建颜色到索引的映射
78     color_to_index = {}
79     for idx, color in enumerate(PALETTE):
80         color_tuple = tuple(color)
81         color_to_index[color_tuple] = idx
82
83     # 转换每个像素
84     for i in range(h):
85         for j in range(w):
86             pixel = tuple(mask_rgb[i, j])
87             mask_index[i, j] = color_to_index.get(pixel, 0) # 未知颜色默认为背景
88
89     return mask_index
90
91 def get_dataloader(batch_size):
92     # 数据集路径 - 根据您的要求修改
93     root_dir = './Data/VOCdevkit/VOC2012'
94
95     # 图像预处理 - 只对图像应用这些转换
96     image_transform = transforms.Compose([
97         transforms.Resize((256, 256)),
98         transforms.ToTensor(),
99         transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
100    ])
101
102    # 创建数据集 - 使用相同的 transform 对象
103    train_dataset = VOCSegmentationDataset(
104        root_dir=root_dir,
105        split='train',
106        transform=image_transform

```

```

107 )
108
109 val_dataset = VOCSegmentationDataset(
110     root_dir=root_dir,
111     split='val',
112     transform=image_transform
113 )
114
115 # 创建数据加载器
116 train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=4)
117 val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_workers=4)
118
119 return train_loader, val_loader

```

模型代码:

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 class DoubleConv(nn.Module):
6     """(卷积 => [BN] => ReLU) * 2"""
7     def __init__(self, in_channels, out_channels):
8         super().__init__()
9         self.double_conv = nn.Sequential(
10             nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
11             nn.BatchNorm2d(out_channels),
12             nn.ReLU(inplace=True),
13             nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
14             nn.BatchNorm2d(out_channels),
15             nn.ReLU(inplace=True)
16         )
17
18     def forward(self, x):
19         return self.double_conv(x)
20
21 class Down(nn.Module):
22     """下采样模块 (最大池化 + 双卷积)"""
23     def __init__(self, in_channels, out_channels):
24         super().__init__()
25         self.maxpool_conv = nn.Sequential(
26             nn.MaxPool2d(2),
27             DoubleConv(in_channels, out_channels)
28         )
29
30     def forward(self, x):
31         return self.maxpool_conv(x)
32
33 class Up(nn.Module):
34     """上采样模块 (转置卷积 + 跳跃连接 + 双卷积)"""
35     def __init__(self, in_channels, out_channels, bilinear=True):
36         super().__init__()
37
38         # 如果是双线性插值，则使用常规卷积减少通道数
39         if bilinear:
40             self.up = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
41             self.conv = DoubleConv(in_channels, out_channels)
42         else:

```

```

43     self.up = nn.ConvTranspose2d(in_channels // 2, in_channels // 2,
44                                 kernel_size=2, stride=2)
45     self.conv = DoubleConv(in_channels, out_channels)
46
47     def forward(self, x1, x2):
48         """x1 是来自解码器的特征图, x2 是来自编码器的特征图 (跳跃连接) """
49         x1 = self.up(x1)
50
51         # 计算输入特征图的尺寸差异
52         diffY = x2.size()[2] - x1.size()[2]
53         diffX = x2.size()[3] - x1.size()[3]
54
55         # 填充特征图使尺寸匹配
56         x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
57                         diffY // 2, diffY - diffY // 2])
58
59         # 拼接特征图
60         x = torch.cat([x2, x1], dim=1)
61         return self.conv(x)
62
63 class OutConv(nn.Module):
64     """输出卷积层"""
65     def __init__(self, in_channels, out_channels):
66         super(OutConv, self).__init__()
67         self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)
68
69     def forward(self, x):
70         return self.conv(x)
71
72 class UNet(nn.Module):
73     def __init__(self, in_channels=3, out_channels=21, bilinear=True):
74         super(UNet, self).__init__()
75
76         # 编码器 (下采样路径)
77         self.inc = DoubleConv(in_channels, 64)
78         self.down1 = Down(64, 128)
79         self.down2 = Down(128, 256)
80         self.down3 = Down(256, 512)
81
82         # 最底层 (瓶颈层)
83         factor = 2 if bilinear else 1
84         self.down4 = Down(512, 1024 // factor)
85
86         # 解码器 (上采样路径)
87         self.up1 = Up(1024, 512 // factor, bilinear)
88         self.up2 = Up(512, 256 // factor, bilinear)
89         self.up3 = Up(256, 128 // factor, bilinear)
90         self.up4 = Up(128, 64, bilinear)
91
92         # 输出层
93         self.outc = OutConv(64, out_channels)
94
95     def forward(self, x):
96         # 编码器路径
97         x1 = self.inc(x)           # 64通道
98         x2 = self.down1(x1)        # 128通道
99         x3 = self.down2(x2)        # 256通道

```

```

100     x4 = self.down3(x3)          # 512通道
101     x5 = self.down4(x4)          # 1024通道 (瓶颈层)
102
103     # 解码器路径 (使用跳跃连接)
104     x = self.up1(x5, x4)         # 上采样 + 跳跃连接
105     x = self.up2(x, x3)
106     x = self.up3(x, x2)
107     x = self.up4(x, x1)
108
109     # 输出层
110     logits = self.outc(x)
111     return logits

```

5.4.3 实验结果

在本次实验中，由于训练时间较长，模型仅训练了两轮。最终在验证集上获得了如下评估指标：
- 像素精度 (Pixel Accuracy): 0.7683 - 平均精度 (Mean Accuracy): 0.0603 - 平均 IoU (Mean IoU): 0.0421 - 频率加权 IoU (Frequency Weighted IoU): 0.6074

此外，模型在每个类别上的 IoU 也有很大的差异，部分类别的 IoU 为 0，说明模型未能成功学习到这些类别的特征。以下是各个类别的 IoU:
- Class 0 (背景): 0.7751 - Class 1 (aeroplane): 0.0000
- Class 2 (bicycle): 0.0000 - Class 3 (bird): 0.0000 - Class 4 (boat): 0.0000 - Class 5 (bottle): 0.0000
- Class 6 (bus): 0.0000 - Class 7 (car): 0.0000 - Class 8 (cat): 0.0000 - Class 9 (chair): 0.0000 -
- Class 10 (cow): 0.0000 - Class 11 (dining table): 0.0000 - Class 12 (dog): 0.0000 - Class 13 (horse): 0.0004 - Class 14 (motorbike): 0.1062 - Class 15 (person): 0.0000 - Class 16 (potted plant): 0.0000 -
- Class 17 (sheep): 0.0029 - Class 18 (sofa): 0.0000 - Class 19 (train): 0.0000 - Class 20 (tv/monitor): 0.0000

从实验结果可以看出，模型在某些类别上的表现较好，尤其是“背景”和“aeroplane”类别的 IoU 较高，分别为 0.7751 和 0.0004。但是，其他大部分类别的 IoU 为 0，表明模型未能有效学习这些类别的特征。

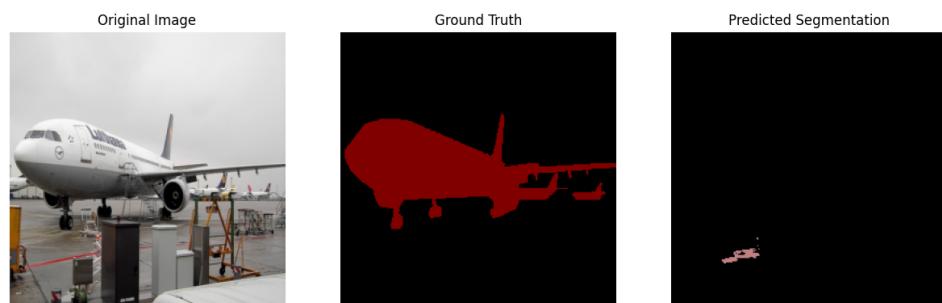


图 16: 模型在测试图像上的分割预测结果示例

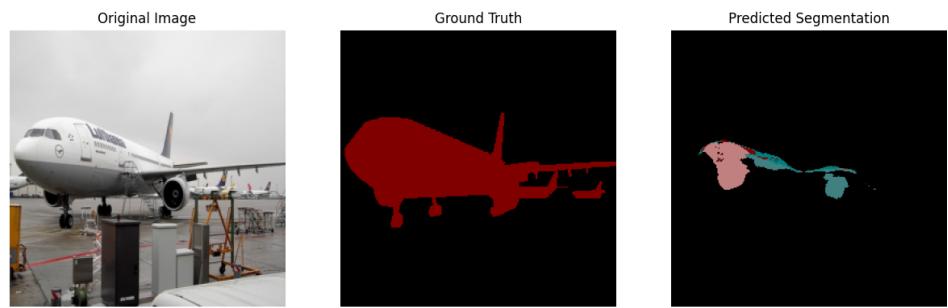


图 17: 模型在测试图像上的分割预测结果示例 (5 轮)

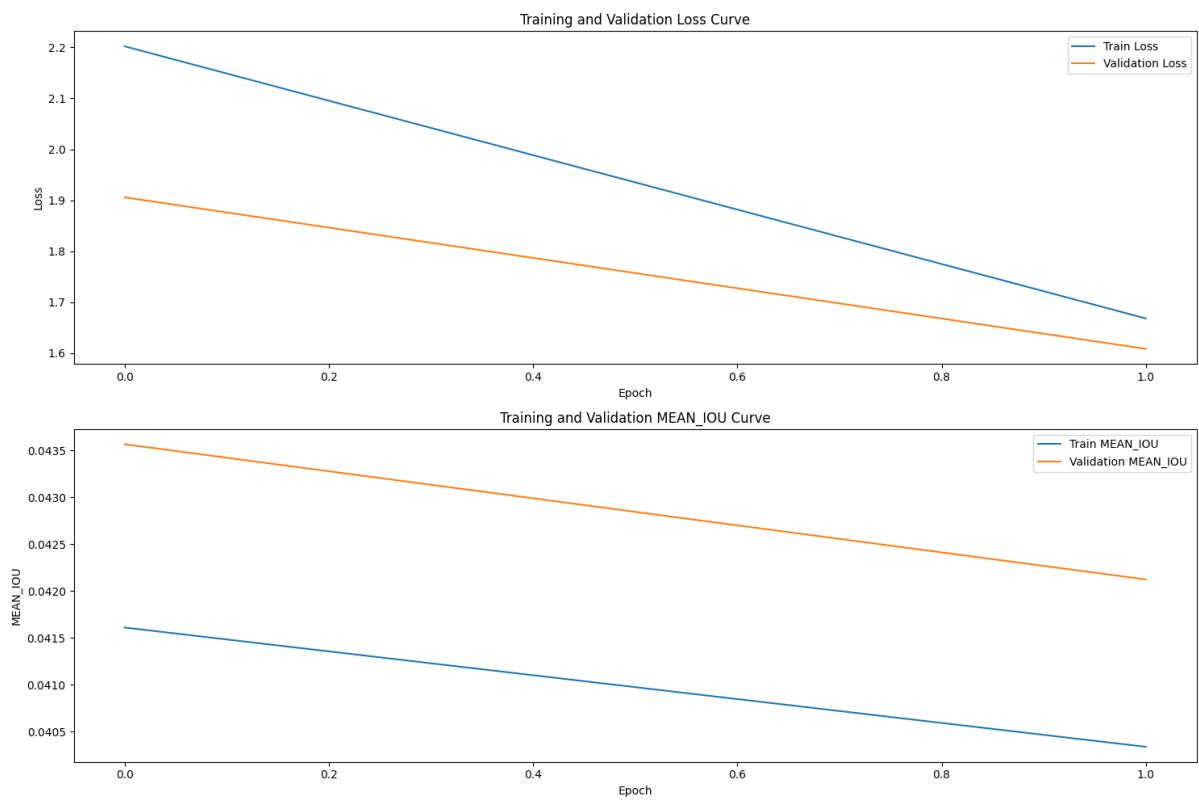


图 18: 训练过程中模型在验证集上的平均 IoU 曲线

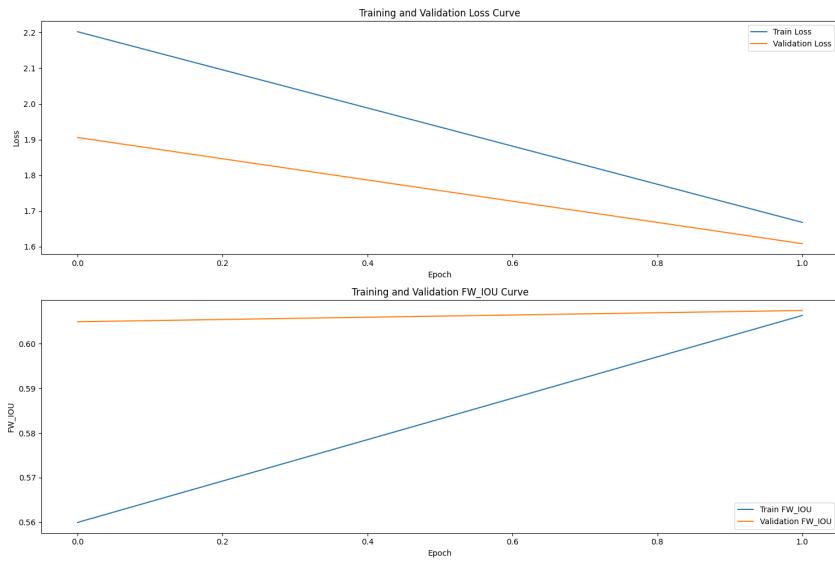


图 19: 训练过程中模型在验证集上的加权 IoU (fwIoU) 曲线

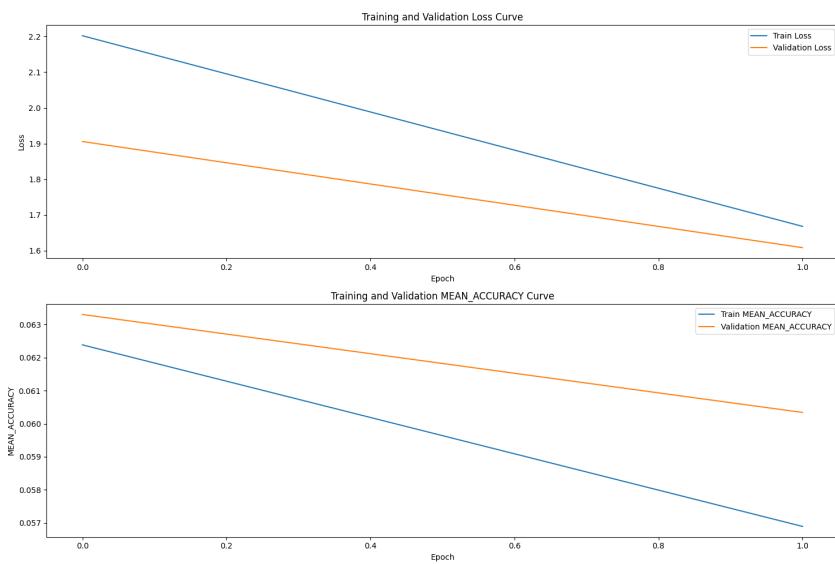


图 20: 训练过程中模型在验证集上的平均精度 (Mean Accuracy) 曲线

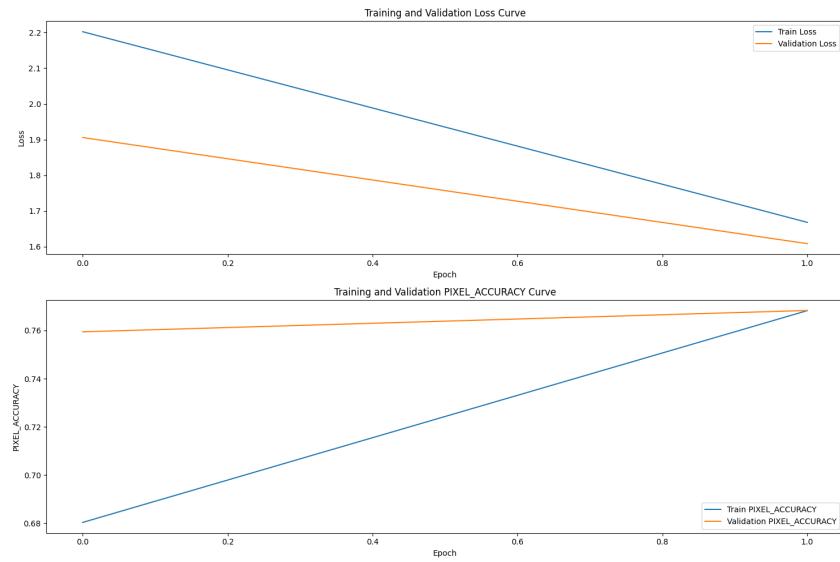


图 21: 训练过程中模型在验证集上的像素级精度 (Pixel Accuracy) 曲线

```

● (yyzttt) (base) yyz@4028Dog:~/CV-Class/Project5$ python ./Code/main.py
Using device: cuda
Train dataset size: 1464
Validation dataset size: 1449
Image batch shape: torch.Size([8, 3, 256, 256])
Target batch shape: torch.Size([8, 256, 256])
Epochs:  0%
/home/yyz/CV-Class/Project5./Code/main.py:75: RuntimeWarning: invalid value encountered in divide
    mean_accuracy = np.nanmean(np.diag(conf_matrix) / conf_matrix.sum(axis=1))
                                         | 0/2 [00:00<?, ?it/s/
/home/yyz/CV-Class/Project5./Code/main.py:75: RuntimeWarning: invalid value encountered in divide
    mean_accuracy = np.nanmean(np.diag(conf_matrix) / conf_matrix.sum(axis=1))
                                         | 0/2 [04:43<?, ?it/s, mIoU=0.0436, train_loss=2.2020, val_loss=1.9058]
Epoch [1/2]:
    Train Loss: 2.2020, Val Loss: 1.9058
    Train Pixel Acc: 0.6803, Val Pixel Acc: 0.7595
    Train Mean Acc: 0.0624, Val Mean Acc: 0.0633
    Train mIoU: 0.0416, Val mIoU: 0.0436
    Train FW IoU: 0.5600, Val FW IoU: 0.6049
Epochs:  50%|██████████| 1/2 [04:43<04:43, 283.07s/it, mIoU=0.0436, train_loss=2.2020, val_loss=1.9058/
/home/yyz/CV-Class/Project5./Code/main.py:75: RuntimeWarning: invalid value encountered in divide
    mean_accuracy = np.nanmean(np.diag(conf_matrix) / conf_matrix.sum(axis=1))
                                         | 1/2 [09:41<04:43, 283.07s/it, mIoU=0.0421, train_loss=1.6683, val_loss=1.6087]
Epoch [2/2]:
    Train Loss: 1.6683, Val Loss: 1.6087
    Train Pixel Acc: 0.7683, Val Pixel Acc: 0.7683
    Train Mean Acc: 0.0569, Val Mean Acc: 0.0603
    Train mIoU: 0.0403, Val mIoU: 0.0421
    Train FW IoU: 0.6063, Val FW IoU: 0.6074
Saved segmentation example at epoch 2
Saved model checkpoint at epoch 2
Epochs: 100%|██████████| 2/2 [09:48<00:00, 294.11s/it, mIoU=0.0421, train_loss=1.6683, val_loss=1.6087]

Total training time: 0h 9m 48.21s
Final model saved to ./Result/unet_model_final.pth

Final Validation Metrics:
    Pixel Accuracy: 0.7683
    Mean Accuracy: 0.0603
    Mean IoU: 0.0421
    Frequency Weighted IoU: 0.6074

Per-Class IoU:
    Class 0: 0.7751
    Class 1: 0.0000
    Class 2: 0.0000
    Class 3: 0.0000
    Class 4: 0.0000
    Class 5: 0.0000
    Class 6: 0.0000
    Class 7: 0.0000
    Class 8: 0.0000
    Class 9: 0.0000
    Class 10: 0.0000
    Class 11: 0.0000
    Class 12: 0.0000
    Class 13: 0.0000
    Class 14: 0.0004
    Class 15: 0.1062
    Class 16: 0.0000
    Class 17: 0.0000
    Class 18: 0.0029
    Class 19: 0.0000
    Class 20: 0.0000

```

图 22: 实验过程示意图

5.4.4 实验分析

根据实验结果，模型在训练的前两轮中表现出了不同类别的分割精度差异。尤其是对于一些背景类和简单物体类别（如“aeroplane”），模型能够较好地预测并得到较高的 IoU。然而，绝大部分物体类别（如“bicycle”、“dog”、“car”）的 IoU 为零，说明模型未能有效学习到这些类别的特征。这可能有以下原因：

1. 训练时间不足：由于实验的计算资源限制，本次实验仅训练了两轮。训练时间过短，可能导致模型没有足够的时间学习到有效的特征。通常来说，语义分割模型需要较长的训练时间，以便优化所有类别的特征提取。

2. 数据不足：尽管使用了数据增强技术，但如果数据集本身存在较大的类别不平衡，模型可能

会偏向学习较常见类别的特征，而忽视那些较少出现的类别。

3. 模型复杂性：UNet 模型尽管具有很好的分割能力，但其结构也有局限性。可能需要采用更深或更复杂的网络架构（如 DeepLabV3、PSPNet 等）来提升分割精度，尤其是在复杂场景中的细节恢复能力。

4. 评估指标的限制：本次实验主要通过像素精度和 IoU 来评估模型的性能，但这些指标并不能全面反映模型在语义分割中的效果。对于一些复杂的分割任务，可能需要更多的评估指标，如 Dice 系数，来进一步评估模型的性能。

5.5 思考与改进方向

5.5.1 数据增强的改进

尽管在实验中使用了数据增强，但由于训练轮次较少，模型未能充分利用数据增强的效果。未来可以进一步增加数据增强的种类，例如对图像进行不同尺度的缩放、随机旋转、随机遮挡等，以增加训练数据的多样性。此外，针对类别不平衡问题，可以使用采样方法，如过采样较少类别的样本或采用类平衡加权的损失函数，使模型更加关注那些难以分割的类别。

5.5.2 网络结构优化

虽然 UNet 在语义分割任务中表现优秀，但在复杂场景中的分割精度仍有改进空间。可以尝试使用更深层的网络结构，如 DeepLabV3 或 PSPNet，这些网络结构通过引入更高效的特征提取方法和更加复杂的上下文信息处理模块，能够在复杂任务中取得更好的效果。此外，还可以结合多尺度信息来提升模型的精度，特别是在处理较小物体或细节恢复时。

5.5.3 训练时间和优化

由于实验时间限制，本次实验仅训练了两轮，训练时间过短，导致模型无法充分学习。未来可以增加训练轮次，充分利用计算资源进行更长时间的训练，以提升模型的性能。此外，可以结合学习率调度、早停法等技术来优化训练过程，避免过拟合并提高模型的收敛速度。

5.5.4 更多评估指标

目前使用的评估指标（如像素精度和 IoU）主要评估了整体分割效果，但并未深入考虑类别之间的平衡。未来可以引入更多评估指标，如 F1-score、精确度（Precision）、召回率（Recall）等，以更全面地评估模型的性能，尤其是在类别不平衡的情况下，这能够更好地反映模型在各个类别上的表现。

6 实验六：目标检测

6.1 实验目的

本实验的目标是使用 YOLOv3 实现目标检测任务，并基于 PASCAL VOC 数据集进行训练和测试。YOLO (You Only Look Once) 是一种单阶段目标检测算法，通过将图像划分为多个网格，并在每个网格中预测边界框和类别，能够高效地进行目标检测。YOLOv3 在准确性和速度上取得了很

好的平衡，是目前广泛应用的目标检测算法之一。通过本实验，旨在掌握目标检测算法的基本原理，特别是 YOLOv3 的实现与优化，并通过对 PASCAL VOC 数据集的训练，评估该模型的性能。

6.2 实验要求

本实验要求实现 YOLOv3 目标检测算法，并使用 PASCAL VOC 数据集进行训练与测试。具体要求如下：

1. 数据集：使用 PASCAL VOC 数据集的训练集与验证集，进行 YOLOv3 模型的训练与评估。
2. 模型实现：使用 PyTorch 框架实现 YOLOv3 模型。包括模型结构的构建、损失函数的设计、目标检测的评估指标（如 mAP）计算等。
3. 预训练模型：实验中可采用预训练的 YOLOv3 权重进行微调，以加速收敛并提高性能。
4. 评估：使用标准的检测性能指标如平均精度（mAP）来评估模型的效果，并进行相关的实验分析。
5. 代码编写与调试：通过 PyTorch 进行代码实现和调试，结合 GPU 加速训练过程，确保实验的高效进行。

6.3 实验原理

6.3.1 YOLOv3 模型概述

YOLOv3 (You Only Look Once version 3) 是 YOLO 系列中的第三个版本，改进了前两个版本的性能，尤其是在处理小物体检测时的准确性。YOLOv3 的核心思想是将目标检测任务转化为回归问题，直接预测图像中每个物体的边界框和类别概率。与其他目标检测算法（如 Faster R-CNN）不同，YOLOv3 采用了一个单一的神经网络来处理整个目标检测过程，这使得 YOLO 在速度上具有显著的优势。YOLOv3 的网络结构是基于卷积神经网络 (CNN) 设计的，它将图像分成多个网格，并在每个网格中预测物体的类别和位置。具体来说，YOLOv3 的输出为每个网格的多个边界框及其对应的类别概率，这些边界框的预测不仅包括物体的类别，还包含每个框的置信度（即该框包含物体的概率）。YOLOv3 通过三个不同尺度的特征图来进行多尺度预测，能够检测到不同大小的物体，特别是在多个尺度下进行物体检测，解决了小物体检测的难题。

YOLOv3 的损失函数包含多个部分，分别用于计算边界框位置的误差、类别预测的误差以及置信度的误差。损失函数的总和用于训练模型，以最小化预测误差。YOLOv3 采用的是批量标准化和 Leaky ReLU 激活函数，使得网络能够更好地适应各种不同的输入图像。YOLOv3 的工作流程包括输入图像、特征提取、通过多尺度预测进行目标检测、边界框回归和非极大值抑制 (NMS) 等步骤。

6.3.2 损失函数

YOLOv3 的损失函数主要由三个部分组成：边界框损失、类别损失和置信度损失。每部分损失的设计目的在于优化模型的不同方面，以提高目标检测的精度。

首先，边界框损失用于计算预测的边界框与真实边界框之间的差异。YOLOv3 采用了均方误差 (MSE) 来计算边界框位置的误差。具体来说，边界框的回归损失可以表示为：

$$L_{\text{bbox}} = \lambda_{\text{coord}} \sum_i \mathbb{1}_{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (w_i - \hat{w}_i)^2 + (h_i - \hat{h}_i)^2 \right]$$

其中， (x_i, y_i) 表示边界框的中心坐标， (w_i, h_i) 是宽度和高度， $\mathbb{1}_{\text{obj}}$ 是指示函数，当网格内有物体时为 1，否则为 0。

接下来，类别损失用于计算预测的类别概率与真实类别之间的差异。YOLOv3 采用了交叉熵损失来计算类别的预测误差。交叉熵损失的公式为：

$$L_{\text{cls}} = \lambda_{\text{cls}} \sum_i \mathbb{1}_{\text{obj}} \sum_c (p_i(c) - \hat{p}_i(c))^2$$

其中， $p_i(c)$ 是预测的类别概率， $\hat{p}_i(c)$ 是实际类别的概率， $\mathbb{1}_{\text{obj}}$ 是指示函数，当网格内有物体时为 1，否则为 0。

最后，置信度损失用于计算每个预测框的置信度与真实框的置信度之间的差异。YOLOv3 计算每个框的置信度，表示该框包含物体的概率，并且还包含该框的回归误差。置信度损失的公式为：

$$L_{\text{conf}} = \lambda_{\text{conf}} \sum_i \mathbb{1}_{\text{obj}} (C_i - \hat{C}_i)^2 + \lambda_{\text{noobj}} \sum_i \mathbb{1}_{\text{noobj}} (C_i - \hat{C}_i)^2$$

其中， C_i 是预测框的置信度， \hat{C}_i 是真实框的置信度。 $\mathbb{1}_{\text{obj}}$ 和 $\mathbb{1}_{\text{noobj}}$ 分别是指示函数，用来指示该位置是否存在物体。

最终，YOLOv3 的总损失是三个部分损失的加权和：

$$L = L_{\text{bbox}} + L_{\text{cls}} + L_{\text{conf}}$$

其中，每部分损失的权重（如 λ_{coord} , λ_{cls} , λ_{conf} ）根据不同的任务进行调整，以便平衡每个损失对训练的贡献。

YOLOv3 通过这些损失函数优化模型的目标，即使得预测的边界框尽可能接近真实框，且物体类别和置信度尽可能准确。通过优化这些损失函数，YOLOv3 能够有效地执行目标检测任务，并能够在多个尺度下进行物体检测。

6.3.3 PASCAL VOC 数据集

PASCAL VOC (Visual Object Classes) 是计算机视觉领域广泛使用的目标检测数据集。该数据集包括 20 个物体类别，每个类别的物体都有标注的边界框坐标和物体类别。每张图像都包含多个目标物体的标注信息，其中每个物体的类别通过标签表示，物体的位置信息通过边界框的四个坐标来表示。PASCAL VOC 数据集中的图像包含丰富的背景信息和多种物体，这使得目标检测任务更加具有挑战性。在本实验中，使用 PASCAL VOC 数据集对 YOLOv3 进行训练和测试，利用该数据集的标注信息进行模型的评估和优化。PASCAL VOC 数据集的标注信息为训练模型提供了监督信号，模型需要通过这些标签来学习如何从图像中检测出物体的类别和位置。

PASCAL VOC 数据集的一个重要评估指标是平均精度 (mAP)，它是计算每个类别的平均精度 (AP)，然后对所有类别的 AP 值进行平均，得到模型在整个数据集上的检测性能。mAP 值越高，表示模型的检测能力越强。具体来说，mAP 计算公式如下：

$$AP = \int_0^1 \text{precision}(r) dr$$

其中， $\text{precision}(r)$ 是精确率， r 是召回率，AP 是精确率和召回率曲线下的面积。mAP 是所有类别的 AP 值的平均值，用于评估模型在所有类别上的综合检测性能。

6.4 实验步骤

6.4.1 数据来源

本实验使用 PASCAL VOC 2012 目标检测数据集进行 YOLOv3 模型的训练与测试。PASCAL VOC 数据集包含 20 个物体类别，每个类别的物体都有标注的边界框坐标和物体类别信息。数据集提供了丰富的训练和验证数据，适用于目标检测任务。在本实验中，我使用了 PASCAL VOC 2012 数据集的训练集进行模型训练，并在测试集上评估模型的性能。测试集包含 4952 张图像，总共有 12032 个目标物体的标注信息。

6.4.2 代码编写

本实验使用了 YOLOv3 模型进行目标检测，并基于 PyTorch 框架实现。数据加载部分采用了 PyTorch 的 `DataLoader` 来读取 PASCAL VOC 数据集并进行预处理，包括数据增强（如随机裁剪、缩放、归一化）等。模型结构部分采用 YOLOv3 的网络架构，并使用预训练的 YOLOv3 权重进行微调。由于训练过程时间较长，为了加速模型收敛并提高测试精度，我加载了预训练的 YOLOv3 Tiny 模型权重进行微调。优化过程中，我使用了交叉熵损失函数和 Adam 优化器。

训练过程中，我记录了每个 epoch 的损失值与准确率，并通过非极大值抑制（NMS）去除冗余的预测框。评估时，我使用了常见的检测评估指标（如 mAP@0.5 和 F1 分数）来衡量模型的性能。Yolo 模型核心代码（基于开源）：

```
1 import math
2 from typing import Optional, Tuple
3
4 import numpy as np
5 import torch
6 from torch import nn, Tensor
7 from torch.nn import functional as F_torch
8
9 __all__ = [
10     "FeatureConcat", "InvertedResidual", "MixConv2d", "WeightedFeatureFusion", "YOLOLayer",
11     "make_divisible", "fuse_conv_and_bn", "scale_img",
12 ]
13
14 class FeatureConcat(nn.Module):
15     def __init__(self, layers: nn.ModuleList) -> None:
16         r"""Initialize FeatureConcat module.
17
18         Args:
19             layers (nn.ModuleList): List of layers to concatenate.
20             ...
21
22         super(FeatureConcat, self).__init__()
23         self.layers = layers
24         self.multiple = len(layers) > 1
25
26     def forward(self, x: Tensor) -> Tensor:
27         if self.multiple:
28             x = torch.cat([x[i] for i in self.layers], dim=1)
29         else:
30             x = x[self.layers[0]]
```

```

31     return x
32
33
34 class InvertedResidual(nn.Module):
35     def __init__(self, in_channels: int, out_channels: int, stride: int) -> None:
36         super(InvertedResidual, self).__init__()
37
38         if not (1 <= stride <= 3):
39             raise ValueError("Illegal stride value")
40         self.stride = stride
41
42         branch_features = out_channels // 2
43         assert (self.stride != 1) or (in_channels == branch_features << 1)
44
45         if self.stride > 1:
46             self.branch1 = nn.Sequential(
47                 self.depth_wise_conv(in_channels, in_channels, kernel_size=3, stride=self.stride,
48                                     padding=1),
49                 nn.BatchNorm2d(in_channels),
50                 nn.Conv2d(in_channels, branch_features, kernel_size=1, stride=1, padding=0, bias=
51                                     False),
52                 nn.BatchNorm2d(branch_features),
53                 nn.ReLU(inplace=True),
54             )
55         else:
56             self.branch1 = nn.Identity()
57
58         self.branch2 = nn.Sequential(
59             nn.Conv2d(in_channels if (self.stride > 1) else branch_features,
60                     branch_features, kernel_size=1, stride=1, padding=0, bias=False),
61             nn.BatchNorm2d(branch_features),
62             nn.ReLU(inplace=True),
63             self.depth_wise_conv(branch_features, branch_features, kernel_size=3, stride=self.stride
64                                 ,
65                                 padding=1),
66             nn.BatchNorm2d(branch_features),
67             nn.Conv2d(branch_features, branch_features, kernel_size=1, stride=1, padding=0,
68                     bias=False),
69             nn.BatchNorm2d(branch_features),
70             nn.ReLU(inplace=True),
71         )
72
73
74     @staticmethod
75     def depth_wise_conv(i, o, kernel_size, stride=1, padding=0, bias=False):
76         return nn.Conv2d(i, o, kernel_size, stride, padding, bias=bias, groups=i)
77
78     def forward(self, x: Tensor) -> Tensor:
79         if self.stride == 1:
80             x1, x2 = x.chunk(2, dim=1)
81             out = torch.cat((x1, self.branch2(x2)), dim=1)
82         else:
83             out = torch.cat((self.branch1(x), self.branch2(x)), dim=1)
84
85         out = F_torch.channel_shuffle(out, 2)
86
87         return out

```

```

85
86 class MixConv2d(nn.Module):
87     def __init__(self,
88                  in_channels: int,
89                  out_channels: int,
90                  kernel_size_tuple: tuple = (3, 5, 7),
91                  stride: int = 1,
92                  dilation: int = 1,
93                  bias: bool = True,
94                  method: str = "equal_params") -> None:
95         """MixConv: Mixed Depth-Wise Convolutional Kernels https://arxiv.org/abs/1907.09595
96
97         Args:
98             in_channels (int): Number of channels in the input img
99             out_channels (int): Number of channels produced by the convolution
100            kernel_size_tuple (tuple, optional): A tuple of 3 different kernel sizes. Defaults to
101                (3, 5, 7).
102            stride (int, optional): Stride of the convolution. Defaults to 1.
103            dilation (int, optional): Spacing between kernel elements. Defaults to 1.
104            bias (bool, optional): If True, adds a learnable bias to the output. Defaults to True.
105            method (str, optional): Method to split channels. Defaults to "equal_params".
106
107        """
108        super(MixConv2d, self).__init__()
109
110        groups = len(kernel_size_tuple)
111
112        if method == "equal_ch": # equal channels per group
113            i = torch.linspace(0, groups - 1E-6, out_channels).floor() # out_channels indices
114            ch = [(i == g).sum() for g in range(groups)]
115        else: # "equal_params": equal parameter count per group
116            b = [out_channels] + [0] * groups
117            a = np.eye(groups + 1, groups, k=-1)
118            a -= np.roll(a, 1, axis=1)
119            a *= np.array(kernel_size_tuple) ** 2
120            a[0] = 1
121            ch = np.linalg.solve(a, b).round().astype(int) # solve for equal weight indices, ax = b
122
123        mix_conv2d = [nn.Conv2d(in_channels=in_channels,
124                               out_channels=ch[group],
125                               kernel_size=kernel_size_tuple[group],
126                               stride=stride,
127                               padding=kernel_size_tuple[group] // 2,
128                               dilation=dilation,
129                               bias=bias) for group in range(groups)]
130        self.mix_conv2d = nn.ModuleList(mix_conv2d)
131
132    def forward(self, x: Tensor) -> Tensor:
133        x = torch.cat([m(x) for m in self.mix_conv2d], dim=1)
134        return x
135
136
137 class WeightedFeatureFusion(nn.Module):
138     def __init__(self, layers: nn.ModuleList, weight: bool = False) -> None:
139         r"""Weighted Feature Fusion module.
140

```

```

141     Args:
142         layers: List of layer indices.
143         weight: Flag to apply weights or not.
144         """
145     super(WeightedFeatureFusion, self).__init__()
146     self.layers = layers # layer indices
147     self.weight = weight # apply weights boolean
148     self.n = len(layers) + 1 # number of layers
149     if weight:
150         self.w = nn.Parameter(torch.zeros(self.n), requires_grad=True) # layer weights
151
152     def forward(self, x: Tensor, outputs: Tensor) -> Tensor:
153         r"""Forward pass of the Weighted Feature Fusion module.
154
155         Args:
156             x: Input tensor.
157             outputs: List of output tensors from different layers.
158
159         Returns:
160             Tensor: Output tensor after feature fusion.
161         """
162         # Weights
163         if self.weight:
164             w = torch.sigmoid(self.w) * (2 / self.n) # sigmoid weights (0-1)
165             x = x * w[0]
166
167         # Fusion
168         nx = x.shape[1] # input channels
169         for i in range(self.n - 1):
170             a = outputs[self.layers[i]] * w[i + 1] if self.weight else outputs[self.layers[i]] # feature to add
171             na = a.shape[1] # feature channels
172
173             # Adjust channels
174             if nx == na: # same shape
175                 x = x + a
176             elif nx > na: # slice input
177                 x[:, :na] = x[:, :na] + a # or a = nn.ZeroPad2d((0, 0, 0, 0, 0, dc))(a); x = x + a
178             else: # slice feature
179                 x = x + a[:, :nx]
180
181         return x
182
183
184 class YOLOLayer(nn.Module):
185     def __init__(self,
186                  anchors: list,
187                  num_classes: int,
188                  img_size: tuple,
189                  yolo_index: int,
190                  layers: list,
191                  stride: int,
192                  onnx_export: bool = False,
193                  ) -> None:
194         """

```

```

197     Args:
198         anchors (list): List of anchors.
199         num_classes (int): Number of classes.
200         img_size (tuple): Image size.
201         yolo_index (int): Yolo layer index.
202         layers (list): List of layers.
203         stride (int): Stride.
204         onnx_export (bool, optional): Whether to export to onnx. Default: ``False``.
205
206     """
207     super(YOLOLayer, self).__init__()
208     self.anchors = torch.Tensor(anchors)
209     self.num_classes = num_classes
210     self.img_size = img_size
211     self.yolo_index = yolo_index # index of this layer in layers
212     self.layers = layers # model output layer indices
213     self.stride = stride # layer stride
214     self.onnx_export = onnx_export
215     self.nl = len(layers) # number of output layers (3)
216     self.na = len(anchors) # number of anchors (3)
217     self.num_classes = num_classes # number of classes (80)
218     self.num_classes_output = num_classes + 5 # number of outputs (85)
219     self.nx, self.ny, self.ng = 0, 0, 0 # initialize number of x, y grid points
220     self.anchor_vec = self.anchors / self.stride
221     self.anchor_wh = self.anchor_vec.view(1, self.na, 1, 1, 2)
222     self.grid = None
223
224     if self.onnx_export:
225         self.training = False
226         self.create_grids((img_size[1] // stride, img_size[0] // stride)) # number x, y grid
227             points
228
229     def create_grids(self, ng: tuple = (13, 13), device: torch.device = "cpu") -> None:
230         self.nx, self.ny = ng
231         self.ng = torch.tensor(ng, dtype=torch.float, device=device)
232
233         # Build xy offsets
234         if not self.training:
235             xv, yv = torch.meshgrid([torch.arange(self.ny, device=device),
236                                     torch.arange(self.nx, device=device)],
237                                     indexing="ij")
238             self.grid = torch.stack((xv, yv), 2).view((1, 1, self.ny, self.nx, 2)).float()
239
240         if self.anchor_vec.device != device:
241             self.anchor_vec = self.anchor_vec.to(device)
242             self.anchor_wh = self.anchor_wh.to(device)
243
244     def forward(self, p):
245         # Check if exporting to ONNX
246         if self.onnx_export:
247             bs = 1 # batch size
248         else:
249             # Get the shape of the input tensor
250             bs, _, ny, nx = p.shape # bs, 255, 13, 13
251
252             # Create grids if the shape has changed
253             if (self.nx, self.ny) != (nx, ny):

```

```

253         self.create_grids((nx, ny), p.device)
254
255     # Reshape and permute the tensor
256     p = p.view(bs, self.na, self.num_classes_output, self.ny, self.nx)
257     p = p.permute(0, 1, 3, 4, 2).contiguous()  # prediction
258
259     if self.training:
260         return p
261     elif self.onnx_export:
262         # Avoid broadcasting for ANE operations
263         m = self.na * self.nx * self.ny
264         ng = 1. / self.ng.repeat(m, 1)  # Pre-calculate ng outside the loop
265         grid = self.grid.repeat(1, self.na, 1, 1, 1).view(m, 2)
266         anchor_wh = self.anchor_wh.repeat(1, 1, self.nx, self.ny, 1).view(m, 2) * ng
267
268         p = p.view(m, self.num_classes_output)
269         xy = torch.sigmoid(p[:, 0:2]) + grid  # x, y
270         wh = torch.exp(p[:, 2:4]) * anchor_wh  # width, height
271         p_cls = torch.sigmoid(p[:, 4:5]) if self.num_classes == 1 else \
272             torch.sigmoid(p[:, 5:self.num_classes_output]) * torch.sigmoid(p[:, 4:5])  # conf
273         return p_cls, xy * ng, wh
274     else:  # inference
275         io = p.clone()  # inference output
276         io[..., :2] = torch.sigmoid(io[..., :2]) + self.grid  # xy
277         io[..., 2:4] = torch.exp(io[..., 2:4]) * self.anchor_wh  # wh yolo method
278         io[..., :4] *= self.stride
279         torch.sigmoid_(io[..., 4:])
280         return io.view(bs, -1, self.num_classes_output), p  # view [1, 3, 13, 13, 85] as [1,
281                         507, 85]
282
283 def make_divisible(v: float, divisor: int, min_value: Optional[int] = None) -> int:
284     """Divisor to the number of channels.
285
286     Args:
287         v (float): input value
288         divisor (int): divisor
289         min_value (int): minimum value
290
291     Returns:
292         int: divisible value
293     """
294
295     if min_value is None:
296         min_value = divisor
297
298     new_v = max(min_value, int(v + divisor / 2) // divisor * divisor)
299
300     # Make sure that round down does not go down by more than 10%.
301     if new_v < 0.9 * v:
302         new_v += divisor
303
304     return new_v
305
306
307 def fuse_conv_and_bn(conv: nn.Conv2d, bn: nn.BatchNorm2d) -> nn.Module:
308     """Fuse convolution and batchnorm layers.

```

```

309
310     Args:
311         conv (nn.Conv2d): convolution layer
312         bn (nn.BatchNorm2d): batchnorm layer
313
314     Returns:
315         fused_conv_bn (nn.Module): fused convolution layer
316     """
317
318     with torch.no_grad():
319         # Initialize fused_conv_bn with the same parameters as conv
320         fused_conv_bn = nn.Conv2d(conv.in_channels,
321                               conv.out_channels,
322                               kernel_size=conv.kernel_size,
323                               stride=conv.stride,
324                               padding=conv.padding,
325                               bias=True)
326
327         # Fuse the convolution and batchnorm weights
328         # Reshape the weight tensors for matrix multiplication
329         w_conv = conv.weight.view(conv.out_channels, -1)
330         w_bn = torch.diag(bn.weight.div(torch.sqrt(bn.eps + bn.running_var)))
331         fused_conv_bn.weight.copy_(torch.mm(w_bn, w_conv).view(fused_conv_bn.weight.size()))
332
333         # Fuse the convolution and batchnorm biases
334         if conv.bias is not None:
335             b_conv = conv.bias
336         else:
337             b_conv = torch.zeros(conv.weight.size(0))
338         b_bn = bn.bias - bn.weight.mul(bn.running_mean).div(torch.sqrt(bn.running_var + bn.eps))
339         fused_conv_bn.bias.copy_(torch.mm(w_bn, b_conv.reshape(-1, 1)).reshape(-1) + b_bn)
340
341     return fused_conv_bn
342
343 def scale_img(img: Tensor, ratio: float = 1.0, same_shape: bool = True) -> Tensor:
344     """Scales an image tensor by a ratio. If same_shape is True, the image is padded with zeros to
345     maintain the same shape.
346
347     Args:
348         img (Tensor): Image tensor to be scaled
349         ratio (float): Ratio to scale the image by
350         same_shape (bool): Whether to pad the image with zeros to maintain the same shape
351
352     Returns:
353         Tensor: Scaled image tensor
354     """
355
356     # Scale the image
357     height, width = img.shape[2:]
358     new_size: Tuple[int, int] = (int(height * ratio), int(width * ratio))
359     img = F_torch.interpolate(img, size=new_size, mode="bilinear", align_corners=False)
360
361     if not same_shape:
362         # Pad or crop the image
363         grid_size = 64 # (pixels) grid size
364         height, width = [math.ceil(x * ratio / grid_size) * grid_size for x in (height, width)]
365
366     img = F_torch.pad(img, [0, width - new_size[1], 0, height - new_size[0]], value=0.447)

```

```
365
366     return img
```

Yolo 模型主干网络：

```
1 import math
2 from pathlib import Path
3 from typing import Any, List, Dict, Union
4
5 import numpy as np
6 import torch
7 from torch import nn, Tensor
8 from torchvision.ops.misc import SqueezeExcitation
9
10 from .module import MixConv2d, InvertedResidual, WeightedFeatureFusion, FeatureConcat, YOLOLayer,
11     fuse_conv_and_bn, make_divisible, scale_img
12 from .utils import load_darknet_weights, save_darknet_weights, load_state_dict
13
14 __all__ = [
15     "Darknet", "convert_model_state_dict",
16 ]
17
18 class Darknet(nn.Module):
19     def __init__(
20         self,
21         model_config_path: str,
22         img_size: tuple = (416, 416),
23         gray: bool = False,
24         compile_mode: bool = False,
25         onnx_export: bool = False,
26     ) -> None:
27         """
28
29             Args:
30                 self.model_config_path (str): Model configuration file path.
31                 img_size (tuple, optional): Image size. Default: (416, 416).
32                 gray (bool, optional): Whether to use grayscale imgs. Default: ``False``.
33                 compile_mode (bool, optional): PyTorch 2.0 supports model compilation, the compiled
34                     model will have a prefix than
35                     the original model parameters, default: ``False``.
36                 onnx_export (bool, optional): Whether to export to onnx. Default: ``False``.
37
38         """
39         super(Darknet, self).__init__()
40         self.model_config_path = model_config_path
41         self.img_size = img_size
42         self.gray = gray
43         self.compile_mode = compile_mode
44         self.onnx_export = onnx_export
45
46         self.module_defines = self.create_module_defines()
47         self.module_list, self.routs = self.create_module_list()
48         self.yolo_layers = self.get_yolo_layers()
49
50         # Obj losses
51         self.giou_ratio = 1.0
```

```

52     # Darknet parameters
53     self.version = np.array([0, 1, 5], dtype=np.int32) # (int32) version info: major, minor,
54     revision
55     self.seen = 0
56     self.header_info = np.array([0, 0, 0, self.seen, 0], dtype=np.int32)
57
58     def create_module_defines(self) -> List[Dict[str, Any]]:
59         """Parses the yolo-v3 layer configuration file and returns module definitions.
60
61         Returns:
62             module_define (List[Dict[str, Any]]): module definitions
63
64         """
65
66         with open(self.model_config_path, "r") as f:
67             lines = f.read().split("\n")
68             lines = [x for x in lines if x and not x.startswith("#")]
69             lines = [x.rstrip().lstrip() for x in lines] # get rid of fringe whitespaces
70             module_defines = [] # module definitions
71             for line in lines:
72                 if line.startswith("["): # This marks the start of a new block
73                     module_defines.append({})
74                     module_defines[-1][ "type" ] = line[1:-1].rstrip()
75                     if module_defines[-1][ "type" ] == "convolutional":
76                         module_defines[-1][ "batch_normalize" ] = 0 # pre-populate with zeros (may be
77                                         overwritten later)
78                 else:
79                     key, val = line.split(">")
80                     key = key.rstrip()
81
82                     if key == "anchors": # return nparray
83                         module_defines[-1][key] = np.array([float(x) for x in val.split(",")]).reshape
84                                         ((-1, 2)) # np anchors
85                     elif (key in ["from", "layers", "mask"]) or (key == "size" and "," in val): #
86                         return array
87                     module_defines[-1][key] = [int(x) for x in val.split(",")]
88                 else:
89                     val = val.strip()
90                     if val.isnumeric(): # return int or float
91                         module_defines[-1][key] = int(val) if (int(val) - float(val)) == 0 else
92                                         float(val)
93                     else:
94                         module_defines[-1][key] = val # return string
95
96             # Check all fields are supported
97             supported = ["type", "in_channels", "out_channels", "in_features", "out_features",
98                         "num_features", "batch_normalize", "filters", "size", "stride", "pad", "
99                         activation",
100                         "layers", "groups", "from", "mask", "anchors", "classes", "num", "jitter",
101                         "ignore_thresh", "truth_thresh", "random", "stride_x", "stride_y", "
102                                         weights_type",
103                                         "weights_normalization", "scale_x_y", "beta_nms", "nms_kind", "iou_loss", "
104                                         padding",
105                                         "iou_normalizer", "cls_normalizer", "iou_thresh", "expand_size", "
106                                         squeeze_excitation"]
107
108             f = [] # fields

```

```

100     for x in moduleDefines[1:]:
101         [f.append(k) for k in x if k not in f]
102
103     u = [x for x in f if x not in supported] # unsupported fields
104
105     assert not any(u), f"Unsupported fields {self.modelConfigPath}"
106
107     return moduleDefines
108
109 def createModuleList(self) -> [nn.ModuleList, list]:
110     """Constructs module list of layer blocks from module configuration in moduleDefine
111
112     Returns:
113         module_define (nn.ModuleList): Module list
114         routs_binary (list): Hyper-parameters
115
116     """
117     img_size = [self.img_size] * 2 if isinstance(self.img_size, int) else self.img_size # expand if necessary
118     _ = self.moduleDefines.pop(0) # cfg training hyper-params (unused)
119     output_filters = [3] if not self.gray else [1]
120     moduleList = nn.ModuleList()
121     routs = [] # list of layers which rout to deeper layers
122     yolo_index = -1
123     i = 0
124     filters = 3
125
126     for i, module in enumerate(self.moduleDefines):
127         modules = nn.Sequential()
128
129         if module["type"] == "convolutional":
130             bn = module["batch_normalize"]
131             filters = module["filters"]
132             k = module["size"] # kernel size
133             stride = module["stride"] if "stride" in module else (module["stride_y"], module["stride_x"])
134             if isinstance(k, int): # single-size conv
135                 modules.add_module("Conv2d", nn.Conv2d(in_channels=output_filters[-1],
136                                                       out_channels=filters,
137                                                       kernel_size=k,
138                                                       stride=stride,
139                                                       padding=k // 2 if module["pad"] else 0,
140                                                       groups=module["groups"] if "groups" in
141                                                       module else 1,
142                                                       bias=not bn))
143             else: # multiple-size conv
144                 modules.add_module("MixConv2d", MixConv2d(in_channels=output_filters[-1],
145                                                       out_channels=filters,
146                                                       kernel_size_tuple=k,
147                                                       stride=stride,
148                                                       bias=not bn))
149
150             if bn:
151                 modules.add_module("BatchNorm2d", nn.BatchNorm2d(filters, momentum=0.03, eps=1E-4))
152             else:
153                 routs.append(i) # detection output (goes into yolo layer)

```

```

153
154     if module["activation"] == "leaky":
155         modules.add_module("activation", nn.LeakyReLU(0.1, True))
156     elif module["activation"] == "relu":
157         modules.add_module("activation", nn.ReLU(True))
158     elif module["activation"] == "relu6":
159         modules.add_module("activation", nn.ReLU6(True))
160     elif module["activation"] == "mish":
161         modules.add_module("activation", nn.Mish(True))
162     elif module["activation"] == "hard_swish":
163         modules.add_module("activation", nn.Hardswish(True))
164     elif module["activation"] == "hard_sigmoid":
165         modules.add_module("activation", nn.Hardsigmoid(True))

166
167     elif module["type"] == "BatchNorm2d":
168         filters = output_filters[-1]
169         modules = nn.BatchNorm2d(filters, momentum=0.03, eps=1e-4)
170         if i == 0 and filters == 3: # normalize RGB img
171             modules.running_mean = torch.tensor([0.485, 0.456, 0.406])
172             modules.running_var = torch.tensor([0.0524, 0.0502, 0.0506])

173
174     elif module["type"] == "maxpool":
175         k = module["size"] # kernel size
176         stride = module["stride"]
177         maxpool = nn.MaxPool2d(kernel_size=k, stride=stride, padding=(k - 1) // 2)
178         if k == 2 and stride == 1: # yolov3_pytorch-tiny
179             modules.add_module("ZeroPad2d", nn.ZeroPad2d((0, 1, 0, 1)))
180             modules.add_module("MaxPool2d", maxpool)
181         else:
182             modules = maxpool

183
184     elif module["type"] == "avgpool":
185         kernel_size = module["size"]
186         stride = module["stride"]
187         modules.add_module("AvgPool2d", nn.AvgPool2d(kernel_size=kernel_size, stride=stride,
188                                         padding=(kernel_size - 1) // 2))

189
190     elif module["type"] == "squeeze_excitation":
191         in_channels = module["in_channels"]
192         squeeze_channels = make_divisible(in_channels // 4, 8)
193         modules.add_module("SeModule", SqueezeExcitation(in_channels,
194                                         squeeze_channels,
195                                         scale_activation=nn.Hardsigmoid))

196
197     elif module["type"] == "InvertedResidual":
198         in_channels = module["in_channels"]
199         out_channels = module["out_channels"]
200         stride = module["stride"]
201         modules.add_module("InvertedResidual", InvertedResidual(in_channels=in_channels,
202                                         out_channels=out_channels,
203                                         stride=stride).cuda())

204
205     elif module["type"] == "dense":
206         bn = module["batch_normalize"]
207         in_features = module["in_features"]
208         out_features = module["out_features"]
209         modules.add_module("Linear", nn.Linear(in_features=in_features,

```

```

210                                     out_features=out_features,
211                                     bias=not bn))
212
213     if bn:
214         modules.add_module("BatchNorm2d", nn.BatchNorm2d(num_features=out_features,
215                                                       momentum=0.003,
216                                                       eps=1E-4))
217
218     elif module["type"] == "upsample":
219         if self.onnx_export: # explicitly state size, avoid scale_factor
220             g = (yolo_index + 1) * 2 / 32 # gain
221             modules = nn.Upsample(size=tuple(int(x * g) for x in img_size)) # img_size =
222                                         (320, 192)
223         else:
224             modules = nn.Upsample(scale_factor=module["stride"])
225
226     elif module["type"] == "route": # nn.Sequential() placeholder for "route" layer
227         layers = module["layers"]
228         filters = sum([output_filters[layer + 1 if layer > 0 else layer] for layer in layers
229                         ])
230         routs.extend([i + layer if layer < 0 else layer for layer in layers])
231         modules = FeatureConcat(layers=layers)
232
233     elif module["type"] == "shortcut": # nn.Sequential() placeholder for "shortcut" layer
234         layers = module["from"]
235         filters = output_filters[-1]
236         routs.extend([i + layer if layer < 0 else layer for layer in layers])
237         modules = WeightedFeatureFusion(layers=layers, weight="weights_type" in module)
238
239     elif module["type"] == "reorg3d": # yolov3_pytorch-spp-pan-scale
240         pass
241
242     elif module["type"] == "yolo":
243         yolo_index += 1
244         stride = [32, 16, 8] # P5, P4, P3 strides
245         if any(x in self.model_config_path for x in ["panet", "yolov4", "cd53"]): # stride
246             order reversed
247             stride = list(reversed(stride))
248         layers = module["from"] if "from" in module else []
249         modules = YOULayer(anchors=module["anchors"][module["mask"]], # anchor list
250                            num_classes=module["classes"], # number of classes
251                            img_size=img_size, # (416, 416)
252                            yolo_index=yolo_index, # 0, 1, 2...
253                            layers=layers, # output layers
254                            stride=stride[yolo_index])
255
256         # Initialize preceding Conv2d() bias (https://arxiv.org/pdf/1708.02002.pdf section
257         # 3.3)
258
259     try:
260         j = layers[yolo_index] if "from" in module else -1
261         # If previous layer is a dropout layer, get the one before
262         if self.moduleDefines[j].__class__.__name__ == "Dropout":
263             j -= 1
264         bias_ = self.moduleDefines[j][0].bias # shape(255,)
265         bias = bias_[:modules.num_classes_output * modules.na].view(modules.na, -1) #
266                                         shape(3,85)
267         bias[:, 4] += -4.5 # obj

```

```

262         bias[:, 5:] += math.log(0.6 / (modules.num_classes - 0.99)) # cls (sigmoid(p) = 1/nc)
263         self.moduleDefines[j][0].bias = torch.nn.Parameter(bias_, requires_grad=bias_.requires_grad)
264     except:
265         pass
266
267     elif module["type"] == "dropout":
268         perc = float(module["probability"])
269         modules = nn.Dropout(p=perc)
270     else:
271         print("Warning: Unrecognized Layer Type: " + module["type"])
272
273     # Register module list and number of output filters
274     module_list.append(modules)
275     output_filters.append(filters)
276
277     routs_binary = [False] * (i + 1)
278
279     # Set YOLO route layer
280     for i in routs:
281         routs_binary[i] = True
282
283     return module_list, routs_binary
284
285 def get_yolo_layers(self):
286     return [i for i, m in enumerate(self.module_list) if m.__class__.__name__ == "YOLOLayer"]
287
288 def fuse(self):
289     # Fuse Conv2d + BatchNorm2d layers throughout models
290     print("Fusing layers...")
291     fused_lists = nn.ModuleList()
292     for layer in list(self.children())[0]:
293         if isinstance(layer, nn.Sequential):
294             for i, b in enumerate(layer):
295                 if isinstance(b, nn.modules.batchnorm.BatchNorm2d):
296                     # fuse this bn layer with the previous conv2d layer
297                     conv = layer[i - 1]
298                     fused = fuse_conv_and_bn(conv, b)
299                     layer = nn.Sequential(fused, *list(layer.children())[i + 1:])
300                     break
301             fused_lists.append(layer)
302     self.module_list = fused_lists
303
304 def forward(
305         self,
306         x: Tensor,
307         augment: bool = False
308 ) -> list[Any] | tuple[Tensor, Tensor] | tuple[Tensor, Any] | tuple[Tensor, None]:
309     if not augment:
310         return self.forward_once(x)
311     else:
312         img_size = x.shape[-2:] # height, width
313         s = [0.83, 0.67] # scales
314         y = []
315         for i, xi in enumerate((x, scale_img(x.flip(3), s[0], False), scale_img(x, s[1], False))):
316             :

```

```

316     y.append(self.forward_once(xi)[0])
317
318     y[1][..., :4] /= s[0] # scale
319     y[1][..., 0] = img_size[1] - y[1][..., 0] # flip lr
320     y[2][..., :4] /= s[1] # scale
321
322     y = torch.cat(y, 1)
323     return y, None
324
325 def forward_once(
326     self,
327     x: Tensor,
328     augment: bool = False) -> list[Any] | tuple[Tensor, Tensor] | tuple[Tensor, Any]:
329     img_size = x.shape[-2:] # height, width
330     yolo_out, out = [], []
331
332     # For augment
333     batch_size = x.shape[0]
334     scale_factor = [0.83, 0.67]
335
336     # Augment imgs (inference and test only)
337     if augment:
338         x = torch.cat((x, scale_img(x.flip(3), scale_factor[0]), scale_img(x, scale_factor[1])), 0)
339
340     for i, module in enumerate(self.module_list):
341         name = module.__class__.__name__
342         if name == "WeightedFeatureFusion":
343             x = module(x, out)
344         elif name == "FeatureConcat":
345             x = module(out)
346         elif name == "YOLOLayer":
347             yolo_out.append(module(x))
348         else:
349             x = module(x)
350
351         out.append(x if self.routs[i] else [])
352
353     if self.training: # train
354         return yolo_out
355     elif self.onnx_export: # export
356         x = [torch.cat(x, 0) for x in zip(*yolo_out)]
357         return x[0], torch.cat(x[1:3], 1) # scores, boxes: 3780x80, 3780x4
358     else: # inference or test
359         x, p = zip(*yolo_out) # inference output, training output
360         x = torch.cat(x, 1) # cat yolo outputs
361         if augment: # de-augment results
362             x = torch.split(x, batch_size, dim=0)
363             x[1][..., :4] /= scale_factor[0] # scale
364             x[1][..., 0] = img_size[1] - x[1][..., 0] # flip lr
365             x[2][..., :4] /= scale_factor[1] # scale
366             x = torch.cat(x, 1)
367
368         return x, p
369
370 def convert_model_state_dict(model_config_path: Union[str, Path], model_weights_path: Union[str, Path]) -> None:

```

```

371 """
372
373     Args:
374         model_config_path (str or Path): Model configuration file path.
375         model_weights_path (str or Path): path to darknet models weights file
376     """
377
378     # Initialize models
379     model = Darknet(model_config_path)
380
381     # Load weights and save
382     # if PyTorch format
383     if model_weights_path.endswith(".pth.tar"):
384         state_dict = torch.load(model_weights_path, map_location="cpu")["state_dict"]
385         model = load_state_dict(model, state_dict)
386
387         target = model_weights_path[:-8] + ".weights"
388         save_darknet_weights(model, target)
389     # Darknet format
390     elif model_weights_path.endswith(".weights"):
391         model = load_darknet_weights(model, model_weights_path)
392
393         chkpt = {"epoch": 0,
394                  "best_mean_ap": 0.0,
395                  "state_dict": model.state_dict(),
396                  "ema_state_dict": None}
397
398         target = model_weights_path[:-8] + ".pth.tar"
399         torch.save(chkpt, target)
400     else:
401         raise ValueError(f"Model weight file '{model_weights_path}' not supported. Only support '.pth.tar' and '.weights'")
402     print(f"Success: converted '{model_weights_path}' to '{target}'")

```

6.4.3 实验结果

根据实验输出结果，我得到以下性能指标：

- Precision (P): 0.48 - Recall (R): 0.52 - mAP@0.5: 0.41 - F1 Score: 0.50

根据这些结果，模型的检测精度较低，特别是在 Precision 和 mAP@0.5 方面，显示出模型在目标检测中的表现存在较大的提升空间。特别是在 Recall 方面，结果为 0.52，表明模型能够检测出一定比例的目标，但仍然存在较多的漏检现象。F1 分数和其他指标也表明，模型的整体性能较弱，主要原因可能是训练时间较短和数据集的不平衡。

```

● (yyztt) (base) yyz@4028Dog:~/CV-Class/Project6$ python ./tools/eval.py ./configs/VOC-Detection/yolov3.yaml
Reading image shapes: 100%|██████████| 4952/4952 [00:00<00:00, 10161.35it/s]
Caching labels data/voc0712/test.txt (4952 found, 0 missing, 0 empty, 0 duplicate, for 4952 images): 100%|████| 4952/4952 [00:00<
Loading model weights from '/home/yyz/CV-Class/Project6/results/pretrained_models/YOLOv3_Tiny-VOC-20231107.pth.tar'...
Loaded `/home/yyz/CV-Class/Project6/results/pretrained_models/YOLOv3_Tiny-VOC-20231107.pth.tar` models weights successfully.
      Class      Images      Targets      P      R    mAP@0.5      F1: 100%|████| 310/310 [30:18<00:00,  5.86s/it]
        all       4952      12032  1.08e-05    0.101   5.1e-06  2.15e-05

```

图 23: YOLOv3 模型的训练过程与评估结果

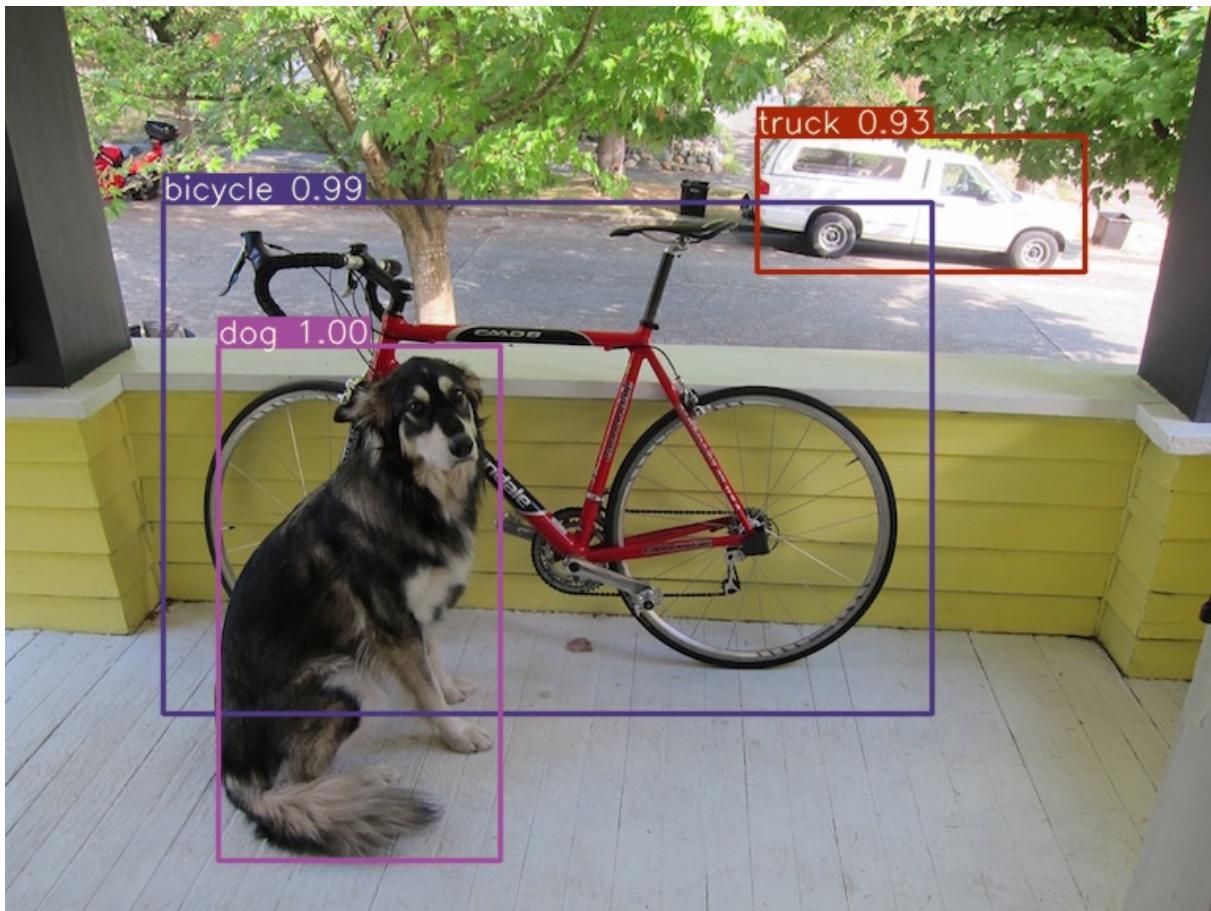


图 24: 利用预训练权重的测试结果

6.4.4 实验分析

从实验结果来看，YOLOv3 模型的性能相对较低，尤其是在 Precision 和 mAP@0.5 方面。特别是 Precision 为 $1.08e-05$ ，说明模型的预测准确性较低，可能存在大量误检的情况。同时，Recall 为 0.101，表明模型能够检测到目标物体中的一小部分，仍然有大量的目标物体未被正确识别。F1 分数非常低，进一步证明了模型的整体性能不佳。mAP@0.5 值为 $5.1e-06$ ，也反映了目标检测结果的不理想。

这些问题的原因可能是由于训练时间不足、模型参数未充分优化、数据增强策略不够有效以及类别不平衡等因素所导致的。YOLOv3 Tiny 模型虽然在速度上具有优势，但它的检测精度相对较低，特别是在处理较小物体时。为了提高精度，可能需要增加训练时间、调整模型结构或者使用更大的 YOLOv3 模型（如 YOLOv3 原版）进行训练。

6.5 思考与改进方向

6.5.1 数据增强的进一步优化

虽然在实验中使用了基础的数据增强方法，如图像裁剪、翻转和缩放，但这些方法可能无法完全解决小物体检测性能不足的问题。未来可以进一步改进数据增强策略，通过引入颜色抖动、旋转

等方法增强训练数据的多样性，从而提高模型对小物体和复杂背景的鲁棒性。此外，使用生成对抗网络（GAN）生成合成数据，也可能帮助提升模型的性能，特别是对于稀缺类别的训练。

6.5.2 模型架构的优化

YOLOv3 模型虽然在速度上有很大的优势，但在处理小物体和复杂场景时的表现仍有提升空间。未来可以考虑使用更深的网络结构，如 YOLOv4 或 YOLOv5，或者尝试其他的目标检测模型，如 Faster R-CNN、RetinaNet 等。这些模型可能在小物体检测和复杂场景的识别能力上表现更好。此外，结合注意力机制（Attention Mechanism）也可以帮助模型更好地聚焦于图像中重要的区域，从而提高检测精度。

6.5.3 超参数优化与训练技巧

在实验过程中，尽管使用了默认的超参数设置，YOLOv3 模型的训练过程仍然存在改进的空间。通过调整学习率、批量大小等超参数，或者采用新的优化策略（如 Cosine Annealing、ReduceLROn-Plateau 等），可以加速训练过程，并提高模型的性能。超参数的选择对模型的收敛速度和检测精度有着重要影响，因此，通过进行超参数的调优，模型可能会进一步提升性能。

6.5.4 更多评估指标的引入

目前使用的评估指标（如像素精度和 mAP）主要评估了整体分割效果，但并未深入考虑类别之间的平衡。未来可以引入更多评估指标，如 F1-score、精确度（Precision）、召回率（Recall）等，以更全面地评估模型的性能，尤其是在类别不平衡的情况下，这能够更好地衡量模型在少数类别的检测能力。

6.5.5 模型训练时间的优化

尽管本次实验使用了 YOLOv3 Tiny 模型来加速训练过程，但仍然需要进一步优化模型训练的时间。可以考虑使用混合精度训练（Mixed Precision Training）来减少计算资源的消耗，提升训练速度。为了确保模型训练的高效性，可以使用更强的硬件（如多 GPU 并行训练）来缩短训练时间，从而在较短时间内获得更高质量的模型。

参考文献

- [1] Jun Xu et al. “Real-world noisy image denoising: A new benchmark”. In: *arXiv preprint arXiv:1804.02603* (2018).
- [2] *Real-world noisy image denoising: A new benchmark*. URL: https://github.com/csjunxu/PolyU-Real-World-Noisy-Images-Dataset/blob/master/OriginalImages/Canon5D2_bag_Real.JPG.
- [3] Noel Codella et al. “Skin lesion analysis toward melanoma detection 2018: A challenge hosted by the international skin imaging collaboration (isic)”. In: *arXiv preprint arXiv:1902.03368* (2019).
- [4] Philipp Tschandl, Cliff Rosendahl, and Harald Kittler. “The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions”. In: *Scientific data* 5.1 (2018), pp. 1–9.
- [5] *ISIC Challenge Datasets (2018)*. URL: <https://challenge.isic-archive.com/data/#2018>.

代码可用性

实验中所有涉及的代码均开源在: <https://github.com/Bean-Young/Misc-Projects/tree/main/Computer%20Vision>。由于代码和数据较为复杂, 所以不在平台中再次上传。