数字图像处理大作业报告

## 背景介绍

鸟类识别任务通常用于生态学研究、物种监测、环境保护等领域。通过鸟类图像识别技术，可以帮助科学家更准确地识别不同鸟种，追踪鸟类种群变化，评估栖息地的健康状况。此外，鸟类是生态系统中的重要组成部分，许多物种面临栖息地丧失、气候变化以及非法捕猎的威胁。因此，通过自动化技术识别鸟类，不仅能够提高监测效率，还能够为物种保护提供支持。例如，在环境保护区、自然保护区，鸟类监测系统可以实时监测鸟类的出现与迁徙，为保护工作提供数据支持。

CUB-200-2011（Caltech-UCSD Birds 200）数据集是一个用于鸟类识别的标准数据集。它包含了200种不同鸟类的图像，共11788张鸟类图像。这个数据集在计算机视觉领域得到了广泛应用，特别是在深度学习和迁移学习的研究中，用于开发和评估鸟类图像识别算法。

通过对CUB-200-2011数据集进行处理和分析，可以帮助开发高效的鸟类识别算法，提高识别准确率，拓宽人工智能在物种识别和生态保护中的应用范围。

对CUB-200-2011数据集进行图像处理，包括几何变换、加噪、平滑、灰度级变换和直方图修正、边缘提取、目标提取等技术，能显著增强数据集的多样性和质量，从而帮助提升鸟类识别的准确性和鲁棒性。这些技术不仅能够改善识别模型的性能，还能够提高其在实际应用中的适用性，尤其是在鸟类监测和物种保护领域中，为生态学家提供更有效的数据支持。

## 二、图像几何变换

1. 图像几何变换过程
2. 先计算新图像的尺寸
3. 根据几何变换的反变换，确定新图像中的每一点再原图像中的对应点。
4. 按对应关系给新图像中各像素赋值。

若原图像中的点对应存在，直接赋值。

若原图像中对应的点坐标超出图像宽高范围，直接赋为背景色。

若原图像中对应的点坐标在图像宽高范围内，但坐标不是整数，则采用插值的方法计算。

1. 图像缩放

原图像中的点(x,y)进行缩放处理后，变换到点(x’,y’)，则缩放处理的矩阵可以表示为



根据该公式，我们可以知道变换后的点对应原图像中的点，从而实现图像缩放。

实现代码：

function new\_img = bilinear\_resize(f, kx, ky)

% 函数功能: 使用双线性插值对图像进行缩放

% 输入参数:

% f - 输入图像 (可以是灰度图像或彩色图像)

% kx - 水平方向缩放因子

% ky - 垂直方向缩放因子

% 输出:

% new\_img - 缩放后的图像

% 获取原图像的大小

[h, w, c] = size(f); % c为通道数（如果是灰度图，c=1）

% 缩放后的图像大小

new\_h = ceil(h \* ky);

new\_w = ceil(w \* kx);

% 初始化放大后的图像

new\_img = zeros(new\_h, new\_w, c);

% 对每个颜色通道分别进行双线性插值操作

for ch = 1:c

for y\_new = 1:new\_h

for x\_new = 1:new\_w

% 计算在原图像中的位置

x\_orig = (x\_new - 1) / kx;

y\_orig = (y\_new - 1) / ky;

% 如果该点在原图像范围内，则进行插值，否则设置为背景色0

if x\_orig >= 0 && x\_orig < w && y\_orig >= 0 && y\_orig < h

% 找到原图像中的四个邻近像素的坐标

x = floor(x\_orig) + 1;

y = floor(y\_orig) + 1;

% 计算插值的权重

a = x\_orig - (x - 1); % 水平方向的权重

b = y\_orig - (y - 1); % 垂直方向的权重

% 如果在图像范围内，使用四个像素进行插值

if x < w && y < h

% 获取四个邻近像素的值

f11 = f(y, x, ch); % 左上角

f12 = f(y + 1, x, ch); % 左下角

f21 = f(y, x + 1, ch); % 右上角

f22 = f(y + 1, x + 1, ch); % 右下角

% 计算插值后的像素值

new\_img(y\_new, x\_new, ch) = (1 - a) \* (1 - b) \* f11 + ...

a \* (1 - b) \* f21 + ...

(1 - a) \* b \* f12 + ...

a \* b \* f22;

% 如果超出图像宽度范围，则在垂直方向进行插值

elseif x >= w && y < h

new\_img(y\_new, x\_new, ch) = f(y, w, ch) + ...

b \* (f(y + 1, w, ch) - f(y, w, ch));

% 如果超出图像高度范围，则在水平方向进行插值

elseif x < w && y >= h

new\_img(y\_new, x\_new, ch) = f(h, x, ch) + ...

a \* (f(h, x + 1, ch) - f(h, x, ch));

% 如果x和y都超出图像范围，则直接使用右下角的像素值

else

new\_img(y\_new, x\_new, ch) = f(h, w, ch);

end

else

% 超出原图像范围时，设置背景色

new\_img(y\_new, x\_new, ch) = 0;

end

end

end

end

% 将结果转换为8位图像

new\_img = uint8(new\_img);

end

1. 图像旋转

原图像中点(x,y)绕原点逆时针旋转θ角后的对应点(x’,y’)可以表示为



绕原点旋转的反变换为



实现代码：

function new\_img = rotate\_image(f, theta)

% 函数功能: 使用双线性插值法对图像进行逆时针旋转

% 输入参数:

% f - 输入图像

% theta - 旋转角度（弧度制）

% 输出:

% new\_img - 旋转后的图像

% 获取原图像的大小

[h, w, c] = size(f);

% 步骤一：先确定旋转后的图像大小

x = [0 w-1 w-1 0];

y = [0 0 h-1 h-1];

x\_new = x\*cos(theta) + y\*sin(theta);

y\_new = -x\*sin(theta) + y\*cos(theta);

% 找新图像的大小

minx = min(x\_new);

miny = min(y\_new);

maxx = max(x\_new);

maxy = max(y\_new);

H = ceil(maxy - miny + 1);

W = ceil(maxx - minx + 1);

% 初始化新图像

new\_img = zeros(H, W, c);

% 对每个通道分别进行处理

for ch = 1:c

for newx = 1:W

for newy = 1:H

% 平移变换

x\_temp = newx - 1 + minx;

y\_temp = newy - 1 + miny;

% 反变换

oldx = x\_temp\*cos(theta) - y\_temp\*sin(theta);

oldy = x\_temp\*sin(theta) + y\_temp\*cos(theta);

% 判断是否在原图像范围内

if oldx < 0 || oldy < 0 || oldx >= w || oldy >= h

new\_img(newy, newx, ch) = 0;

else

x = floor(oldx) + 1;

y = floor(oldy) + 1;

a = oldx - floor(oldx);

b = oldy - floor(oldy);

% 双线性插值

if x < w && y < h

f11 = f(y, x, ch); % 左上角

f12 = f(y + 1, x, ch); % 左下角

f21 = f(y, x + 1, ch); % 右上角

f22 = f(y + 1, x + 1, ch);% 右下角

new\_img(newy, newx, ch) = (1 - a) \* (1 - b) \* f11 + ...

a \* (1 - b) \* f21 + ...

(1 - a) \* b \* f12 + ...

a \* b \* f22;

elseif x >= w && y < h

new\_img(newy, newx, ch) = f(y, w, ch) + b \* (f(y + 1, w, ch) - f(y, w, ch));

elseif x < w && y >= h

new\_img(newy, newx, ch) = f(h, x, ch) + a \* (f(h, x + 1, ch) - f(h, x, ch));

else

new\_img(newy, newx, ch) = f(h, w, ch);

end

end

end

end

end

% 将结果转换为uint8格式

new\_img = uint8(new\_img);

end

1. 图像错切

图像的错切变换是平面景物在投影平面上的非垂直投影。错切变换使图像中的图形产生扭变。这种扭变只在水平或者垂直方向产生时，分别称为水平方向错切和垂直方向错切。

设原图像中的点(x,y)进行错切变换后，变换到点(x’,y’)，则错切变换的矩阵表达式为：



实现代码：

function new\_img = shear\_image(f, shear\_factor\_xy, shear\_factor\_yx)

% 函数功能: 对图像同时进行水平和垂直方向的错切变换

% 输入参数:

% f - 输入图像

% shear\_factor\_xy - 水平方向错切因子

% shear\_factor\_yx - 垂直方向错切因子

% 输出:

% new\_img - 变换后的图像

% 获取原图像的大小

[h, w, c] = size(f);

% 水平和垂直方向的错切变换矩阵

T = [1 shear\_factor\_xy; shear\_factor\_yx 1];

% 计算变换后图像的范围

corners = [1, 1, w, w; 1, h, 1, h]; % 原图像四个角的坐标

new\_corners = T \* corners; % 变换后的四个角坐标

minx = min(new\_corners(1, :));

maxx = max(new\_corners(1, :));

miny = min(new\_corners(2, :));

maxy = max(new\_corners(2, :));

% 计算新图像的大小

new\_w = ceil(maxx - minx);

new\_h = ceil(maxy - miny);

% 初始化新图像

new\_img = zeros(new\_h, new\_w, c);

% 对每个通道分别处理

for ch = 1:c

for newx = 1:new\_w

for newy = 1:new\_h

% 反向变换到原图坐标

pos = inv(T) \* [(newx + minx - 1); (newy + miny - 1)];

oldx = pos(1);

oldy = pos(2);

% 判断是否在原图像范围内

if oldx >= 1 && oldx <= w && oldy >= 1 && oldy <= h

% 双线性插值

x1 = floor(oldx);

x2 = ceil(oldx);

y1 = floor(oldy);

y2 = ceil(oldy);

% 计算插值权重

a = oldx - x1;

b = oldy - y1;

% 获取邻近像素的值

if x1 < 1, x1 = 1; end

if y1 < 1, y1 = 1; end

if x2 > w, x2 = w; end

if y2 > h, y2 = h; end

f11 = double(f(y1, x1, ch)); % 左上角

f12 = double(f(y2, x1, ch)); % 左下角

f21 = double(f(y1, x2, ch)); % 右上角

f22 = double(f(y2, x2, ch)); % 右下角

% 插值计算

new\_img(newy, newx, ch) = (1 - a) \* (1 - b) \* f11 + ...

a \* (1 - b) \* f21 + ...

(1 - a) \* b \* f12 + ...

a \* b \* f22;

else

% 超出原图范围设为背景色0

new\_img(newy, newx, ch) = 0;

end

end

end

end

% 将结果转换为uint8格式

new\_img = uint8(new\_img);

end

1. 图像镜像

设图像的分辨率为M×N，采用像素坐标系，图像镜像变换矩阵公式如下所示，镜像就是左右、上下或者对角对换。

水平镜像：

垂直镜像：

对角镜像：

实现代码：

function new\_img = mirror\_image(f, mode)

% 函数功能: 对图像进行镜像变换

% 输入参数:

% f - 输入图像

% mode - 选择镜像模式, 'horizontal' 水平镜像，'vertical' 垂直镜像, 'diagonal' 对角镜像

% 输出:

% new\_img - 变换后的图像

[h, w, c] = size(f); % 获取原图像大小

% 初始化新图像

new\_img = zeros(h, w, c);

switch mode

case 'horizontal'

% 水平镜像（沿着垂直轴翻转）

for y = 1:h

for x = 1:w

new\_img(y, x, :) = f(y, w - x + 1, :);

end

end

case 'vertical'

% 垂直镜像（沿着水平轴翻转）

for y = 1:h

for x = 1:w

new\_img(y, x, :) = f(h - y + 1, x, :);

end

end

case 'diagonal'

% 对角镜像（沿着对角线翻转）

% 假设输入是方阵，若不是，则以左上角矩阵为主

for y = 1:min(h, w)

for x = 1:min(h, w)

new\_img(x, y, :) = f(y, x, :); % 交换行列位置

end

end

otherwise

error('未知的镜像模式. 请选择 horizontal, vertical 或 diagonal.');

end

% 将新图像转换为uint8类型

new\_img = uint8(new\_img);

end

## 四、直方图修正

### 直方图均衡化

直方图修正法的基本原理就是通过构造灰度级变换函数改造原图像的直方图，使变换后的图像的直方图达到一定要求。

直方图均衡化是采用灰度级r的累积分布函数作为变换函数的直方图修正法。给出一幅数字图像，共有L个灰度级，总像素个数为N，其中，第j级灰度对应的像素数为。对该图像进行灰度直方图统计，则图像进行直方图均衡化处理的变换函数T（r）为



对一幅数字图像进行直方图均衡化处理的算法步骤如下：

1. 统计原始图像的直方图。
2. 计算新的灰度级。
3. 修正sk为合理的灰度级。
4. 计算新的直方图。
5. 用处理后的新灰度代理处理前的灰度，生成新图像。

实现代码如下：

function [NewImage] = histogram\_equalization(input\_img)

% 读取图像并转换为灰度图像

if size(input\_img, 3) == 3

Image = rgb2gray(input\_img); % 将彩色图像转换为灰度图像

else

Image = input\_img; % 如果已经是灰度图像，直接使用

end

% 计算原始灰度图像的直方图

histgram = imhist(Image); % 统计图像直方图

[h, w] = size(Image);

% 初始化均衡化后的图像矩阵

NewImage = zeros(h, w);

% 计算累积直方图

s = zeros(256, 1);

s(1) = histgram(1);

for t = 2:256

s(t) = s(t-1) + histgram(t); % 计算新的灰度值

end

% 生成均衡化后的图像

for x = 1:w

for y = 1:h

NewImage(y, x) = s(Image(y, x) + 1) / (w \* h); % 生成新图像

end

end

% 返回处理后的图像

end

### 直方图匹配

直方图匹配涉及到概率密度函数（PDF）和累积分布函数（CDF）的概念。对于给定的灰度级r，其概率密度函数pr(r)表示该灰度级出现的概率。累积分布函数则是概率密度函数的积分，表示灰度级小于或等于r的所有概率之和。

在直方图匹配中，目标是找到一个变换函数T，使得变换后的图像具有预定的直方图形状。这可以通过以下步骤实现：

1. 计算原始图像的累积分布函数。
2. 计算参考图像的累积分布函数。
3. 对于原始图像中的每个灰度级r，找到参考图像中累积分布函数值与之相等（或最接近）的灰度级z。
4. 使用找到的映射关系替换原始图像中的每个像素点的灰度值。

实现代码如下：

function matched\_img = histogram\_matching(source, reference)

% 如果是彩色图像，分别处理每个通道

if size(source, 3) == 3

matched\_img = source;

for channel = 1:3

matched\_img(:,:,channel) = match\_histogram\_single\_channel(source(:,:,channel), reference(:,:,channel));

end

else

% 如果是灰度图像，直接进行匹配

matched\_img = match\_histogram\_single\_channel(source, reference);

end

end

function matched\_channel = match\_histogram\_single\_channel(source\_channel, reference\_channel)

% 获取源图像和参考图像的直方图

[src\_counts, src\_bins] = imhist(source\_channel);

[ref\_counts, ref\_bins] = imhist(reference\_channel);

% 计算源图像和参考图像的累计分布函数（CDF）

src\_cdf = cumsum(src\_counts) / numel(source\_channel);

ref\_cdf = cumsum(ref\_counts) / numel(reference\_channel);

% 创建一个映射表，将源图像的像素值映射到参考图像的像素值

mapping = zeros(256, 1);

ref\_idx = 1;

for src\_idx = 1:256

% 找到与源图像的CDF最接近的参考图像的CDF

while ref\_cdf(ref\_idx) < src\_cdf(src\_idx) && ref\_idx < 256

ref\_idx = ref\_idx + 1;

end

mapping(src\_idx) = ref\_bins(ref\_idx);

end

% 使用映射表对源图像进行像素值映射

matched\_channel = uint8(mapping(double(source\_channel) + 1));

end

## 五、灰度级变换

灰度级变换就是借助于变换函数将输入的像素灰度值映射成一个新的输出值，通过改变像素的亮度值来增强图像，如公式所示



其中，f(x,y)是输入图像，g(x,y)为变换后的输出图像，T是灰度变换函数。根据变换函数的不同，灰度级变换可以分为线性灰度级变换和非线性灰度级变换。

### 线性灰度级变换

#### 基本线性灰度级变换

最基本的线性灰度级变换如公式————所示。将输入的图像f(x,y)变换为g(x,y)，变换结果由倾角α决定，如公式————所示。



当α为45°时，图像无变化；当α<45°时，变换后的图像均匀变暗；反之，变换后的图像均匀变亮。

实现代码如下：

function transformed\_image = linear\_gray\_transform(image, alpha)

% 检查输入图像是否是彩色图像，如果是，则转换为灰度图像

if size(image, 3) == 3

image = rgb2gray(image); % 使用 rgb2gray 转换彩色图像为灰度图像

end

% 将图像转换为 double 类型，以便进行数值计算

image = double(image);

% 计算 tan(alpha)，需要将角度转换为弧度

alpha\_rad = deg2rad(alpha); % 将角度转换为弧度

tan\_alpha = tan(alpha\_rad); % 计算 tan(alpha)，它是一个标量

% 应用线性灰度级变换，按元素乘以 tan\_alpha

transformed\_image = image .\* tan\_alpha; % 按元素相乘

% 对变换后的图像进行裁剪，确保像素值在 0 到 255 之间

transformed\_image = uint8(max(min(transformed\_image, 255), 0)); % 裁剪并转换回 uint8 类型

end

#### 分段线性灰度级变换

分段线性灰度级变换是将输入的图像f(x,y)的灰度级区间分成两段乃至多段，分别作线性灰度级变换，以获得增强图像g(x,y)。典型的三段线性灰度级变换如公式————所示。



其中，参数a,b,c,d为用于确定三段线段斜率的常数，取值可根据具体变换需求灵活设定。但存在一些情况，用户仅对某个范围内的灰度感兴趣，只需对其进行线性拉伸，而将其余灰度级保持不变或变为某固定值。

实现代码如下：

function output\_img = piecewise\_linear\_gray(input\_img)

% 输入参数：

% input\_img: 输入图像，可以是灰度图像

% a=30, b=100, c=75, d=200 设定的四个阈值，范围是0-255

% 确保输入图像是灰度图像

if size(input\_img, 3) == 3

input\_img = rgb2gray(input\_img); % 如果是彩色图像，转换为灰度图像

end

% 参数设定，范围应在0-255之间

a = 30; % 对应原图中的灰度值

b = 100;

c = 75;

d = 200;

% 将输入图像转化为 double 类型以便处理

input\_img = double(input\_img);

% 获取图像的大小

[rows, cols] = size(input\_img);

% 初始化输出图像

output\_img = zeros(rows, cols);

% 遍历图像中的每个像素，应用三段线性变换

for i = 1:rows

for j = 1:cols

pixel\_value = input\_img(i, j);

if pixel\_value < a

% 第一段：线性变换公式：s = (c/a) \* r

output\_img(i, j) = (c / a) \* pixel\_value;

elseif pixel\_value >= a && pixel\_value < b

% 第二段：线性变换公式：s = ((d - c)/(b - a)) \* (r - a) + c

output\_img(i, j) = ((d - c) / (b - a)) \* (pixel\_value - a) + c;

else

% 第三段：线性变换公式：s = ((255 - d)/(255 - b)) \* (r - b) + d

output\_img(i, j) = ((255 - d) / (255 - b)) \* (pixel\_value - b) + d;

end

end

end

% 限制输出图像像素值的范围，确保在0到255之间

output\_img = uint8(min(max(output\_img, 0), 255));

end

### 非线性灰度级变换

当用某些非线性变换函数作为灰度变换的变换函数时，可实现图像灰度的非线性变换。对数变换、指数变换和幂变换是常见的非线性变换。

#### 对数变换

基于对数变换的非线性灰度级变换如公式————所示。



式中，c是尺度比例常数，其取值可以结合输入图像的范围来定。当希望对图像的低灰度区作较大拉伸、高灰度区压缩时，可采用这种变换，它能使图像的灰度分布与人的视觉特性相匹配。对数变换一般适用于处理过暗图像，本系统固定c值为1。

实现代码如下：

function output = logTransform(inputImage)

% 对数变换函数

% inputImage: 输入图像（灰度图或单通道图像）

% c: 常数系数，用于调整输出图像的强度

% 如果是彩色图像，转换为灰度图像

if size(inputImage, 3) == 3

inputImage = rgb2gray(inputImage); % 将RGB图像转换为灰度图像

end

% 转换为双精度浮点数

inputImage = double(inputImage);

% 对数变换公式：output = c \* log(1 + inputImage)

output = log(1 + inputImage);

% 归一化到 [0, 1] 区间

output = mat2gray(output); % 将输出规范化到 [0, 1] 范围

% 映射到 [0, 255] 并转换为 uint8 类型

output = uint8(output \* 255);

end

#### 指数变换

基于指数变换的非线性灰度变换如式————所示。



式中，a用于决定指数变换函数曲线的初始位置。当f(x,y)=a时，g(x,y)=0，曲线与x轴交叉。b是底数，c用于决定指数变换曲线的陡度。当希望对图像的低灰度区压缩，高灰度区作较大拉伸时，可采用这种变换。指数变换一般适用于处理过亮图像。

实现代码如下：

function output = expTransform(inputImage)

% 幂指数变换函数

% inputImage: 输入图像（灰度图或单通道图像）

% 如果是彩色图像，转换为灰度图像

if size(inputImage, 3) == 3

inputImage = rgb2gray(inputImage); % 将RGB图像转换为灰度图像

end

% 转换为双精度浮点数

inputImage = double(inputImage);

% 幂指数变换公式：output = exp(c \* (inputImage - a)) + 1

output = exp(0.325 \* (inputImage - 225) / 30) + 1;

% 归一化到 [0, 1] 范围

output = mat2gray(output); % 将输出规范化到 [0, 1] 范围

% 映射到 [0, 255] 并转换为 uint8 类型

output = uint8(output \* 255); % 映射并转换为 uint8

end

## 六、边缘提取

本系统实现了一个彩色图像的边缘检测函数，支持四种常用的边缘检测算子：Sobel、Prewitt、Robert 和拉普拉斯（Laplacian）。边缘检测是一种常见的图像处理技术，用于提取图像中的边界信息。输入图像是一个彩色图像，函数通过选择不同的边缘检测算子，对图像的每个颜色通道（红色、绿色、蓝色）进行处理，并返回每个通道的边缘检测结果。

首先，函数将输入图像转换为 double 类型，以便进行后续的计算。然后，可以根据用户输入的算子名称（通过 operator 参数）选择对应的边缘检测滤波器。每种算子都有其独特的滤波器设计：

**Sobel 算子**：用于计算图像的水平方向和垂直方向的梯度，通常用于检测边缘的方向和强度。Sobel算子引入了平均因素，对图像中的随机噪声有一定的平滑作用，相隔两行或两列求差分，故边缘两侧的元素得到了加强，边缘显得粗而亮。

**Prewitt 算子**：与 Sobel 算子类似，但其模板系数略有不同，适用于不同的边缘检测需求。

**Robert 算子**：使用 2x2 小的卷积核，计算水平方向和垂直方向的梯度，常用于图像的细节边缘检测。

**拉普拉斯算子**：与其他算子不同，它通过计算图像的二阶导数来检测边缘，通常用于发现图像中的快速变化区域。

应用对应的滤波器计算水平方向和垂直方向的梯度后，使用公式————计算出梯度，得到图像的梯度幅值。



接下来，通过设定的阈值对计算得到的梯度幅值进行二值化处理，生成边缘图像。二值化的结果是，如果某个像素的梯度幅值大于阈值，就认为该像素是边缘点，赋值为 1，否则为 0。

实现代码如下：

function edgeI = edgeDetection(Image, operator)

% 输入：Image - 彩色图像 (double 类型)

% operator - 边缘检测算子名称 (字符串, 'sobel', 'prewitt', 'robert', 'laplacian')

% 输出：edgeI - 彩色图像的边缘检测结果

threshold=0.4;

Image=im2double(Image);

% 获取图像的大小

[rows, cols, channels] = size(Image);

% 初始化边缘检测结果

edgeI = zeros(rows, cols, channels);

% 根据输入的算子选择对应的滤波器

switch operator

case 'sobel'

% Sobel 算子

H1 = [-1 0 1; -2 0 2; -1 0 1]; % 水平方向

H2 = [-1 -2 -1; 0 0 0; 1 2 1]; % 垂直方向

case 'prewitt'

% Prewitt 算子

H1 = [-1 0 1; -1 0 1; -1 0 1]; % 水平方向

H2 = [-1 -1 -1; 0 0 0; 1 1 1]; % 垂直方向

case 'robert'

% Robert 算子 (通常是 2x2 的小滤波器)

H1 = [1 0; 0 -1]; % 水平方向

H2 = [0 1; -1 0]; % 垂直方向

case 'laplacian'

% 拉普拉斯算子

H1 = [0 1 0; 1 -4 1; 0 1 0]; % 拉普拉斯算子

otherwise

error('Unsupported operator. Choose from "sobel", "prewitt", "robert", or "laplacian".');

end

% 分别处理 RGB 三个通道

for c = 1:channels

% 提取当前通道

channel = Image(:, :, c);

% 计算梯度幅值

if strcmp(operator, 'laplacian')

% 对于拉普拉斯算子，只有一个方向

R1 = imfilter(channel, H1); % 水平方向梯度

gradI = abs(R1);

else

% 其他算子，使用欧几里得距离计算梯度幅值

R1 = imfilter(channel, H1); % 水平方向梯度

R2 = imfilter(channel, H2); % 垂直方向梯度

gradI = abs(R1) + abs(R2);

end

% 阈值处理生成二值边缘

edgeI(:, :, c) = gradI > threshold;

end

end

## 七、图像平滑

### 低通滤波

理想低通滤波是在傅里叶平面上半径为D0的圆形滤波器，其传递函数为



其中，D0>0，为理想滤波器的截止频率，D(u,v)为点(u,v)到傅里叶频率域原点的距离，定义为。

首先，将输入的灰度图像转换到频域，这一步通过傅里叶变换完成，同时进行频谱搬移，使得频域的中心点位于图像的中央。接着，代码计算图像中心的位置，并创建一个理想低通滤波器的模板，用于保留图像的低频成分，同时去除高频成分（如噪声和边缘细节）。

在构建滤波器时，根据每个频域点到中心的距离与设定的截止频率的比较，决定是否保留该频率。距离小于截止频率的点被保留，而其他频率被阻断。

完成滤波后，通过逆傅里叶变换实现将频域的结果转换回空间域，恢复图像。

实现代码如下：

%单通道滤波

function outputImage = ideal\_lowpass\_filter\_single(inputImage, D0)

% 单通道的理想低通滤波器

% 输入：inputImage - 输入的单通道图像（灰度图）

% D0 - 截止频率

% 输出：outputImage - 经过低通滤波后的图像

% 执行傅里叶变换并进行频谱搬移

FImage = fftshift(fft2(double(inputImage))); % 傅里叶变换及频谱搬移

% 获取图像大小

[N, M] = size(FImage);

% 计算图像的中心坐标

r1 = floor(M / 2);

r2 = floor(N / 2);

% 创建空的频率响应矩阵

g = zeros(N, M);

% 生成理想低通滤波器

for x = 1:M

for y = 1:N

d = sqrt((x - r1)^2 + (y - r2)^2); % 计算当前点到频域中心的距离

if d <= D0

h = 1; % 保留低频

else

h = 0; % 阻断高频

end

g(y, x) = h \* FImage(y, x); % 应用低通滤波器

end

end

% 执行逆傅里叶变换，得到滤波后的图像

g = real(ifft2(ifftshift(g))); % 逆变换并去除频谱搬移

% 归一化输出图像，确保值在0-255之间

outputImage = uint8(mat2gray(g) \* 255);

end

function outputImage=ideal\_lowpass\_filter\_color(Image, D0)

if size(Image, 3) == 3

% 如果是彩色图像，分离RGB通道

R = Image(:,:,1);

G = Image(:,:,2);

B = Image(:,:,3);

else

% 如果是灰度图像，直接处理

R = Image;

G = Image;

B = Image;

end

% 执行滤波

% 创建空的输出图像

output\_R = ideal\_lowpass\_filter\_single(R, D0);

output\_G = ideal\_lowpass\_filter\_single(G, D0);

output\_B = ideal\_lowpass\_filter\_single(B, D0);

% 合并三个通道

outputImage = cat(3, uint8(output\_R), uint8(output\_G), uint8(output\_B));

### 双边滤波

双边滤波是一种非线性滤波方法，其核心思想是结合空间相似性和灰度相似性，在平滑图像的同时保留图像的边缘细节信息。它在图像去噪和边缘保持方面有重要应用。

双边滤波对一个像素的输出值由其周围像素的加权平均值决定。加权系数不仅与像素之间的空间距离有关，还与像素的灰度差异有关，具体计算公式为：



其中，表示点p双边滤波的结果，表示点p和点q的灰度值。S表示滤波窗口的范围。为空间邻域标准差，为像素亮度标准差，这是双边滤波的两个关键参数，当变大时，窗口包含的像素变多，距离远的像素也能影响到中心像素点，平滑程度也越高。用来控制灰度邻近度，当变大时，则灰度差值大于的像素将不会参与运算，使得能够保留图像高频边缘的灰度信息。分别为空间邻近度函数和灰度邻近度函数，其形式为高斯函数。是一个标准量，表示灰度权值和空间权值乘积的加权和，其定义为：



对彩色图像的双边滤波，就是对每个通道分别进行双边滤波（使用Matlab的双边滤波函数imbilatfilt），最后，函数将处理后的三个颜色通道（红色、绿色和蓝色通道）重新合并成一个完整的彩色图像，并返回该图像作为滤波结果。通过这种方式，双边滤波能够有效地去除图像中的噪声，同时保留图像的边缘和细节，使得图像在平滑的同时保持视觉质量。

### 模糊滤波

模糊理论对于不确定性问题的处理有独特的优势，以此为基础的模糊滤波器非常适合高斯噪声的滤除。因此基于模糊数学思想，利用模糊隶属度函数的概念，通过对均值滤波器权值加以优化，提高平滑高斯噪声的能力。

模糊加权均值滤波的具体算法步骤如下：

1. 计算以点(x,y)为中心的邻域内灰度差异变化。给定一幅输入图像f(x,y)，S是以点(x,y)为中心的邻域，计算该邻域内灰度变化公式为：



然后计算该邻域内的平均灰度变化：



其中，N是集合S内邻点的数目，不包含点(x,y)。

1. 计算每一邻域点对中心点的模糊隶属度。模糊隶属度公式如下：



其中，为模糊隶属度，且。

1. 计算当前窗口模糊加权均值滤波输出g(x,y)为：



最后，用加权平均值g(x,y)来代替滤波窗口内中心像素点的灰度值f(x,y)，实现对高斯噪声点的滤波处理。

实现代码如下：

function result = blurFilter(Image)

% Image: 输入的图像

% n: 模板的半径，决定邻域大小 (2\*n+1) x (2\*n+1) n=1

n=1;

% 转换为 double 类型，方便进行计算

NoiseI = im2double(Image);

% 图像尺寸获取

[height, width, channels] = size(NoiseI);

% 邻域模板半径

result = zeros(height, width, channels);

% 对每个通道进行模糊滤波处理

for c = 1:channels

% 扩展边缘

hh = height + 2 \* n;

ww = width + 2 \* n;

ff = zeros(hh, ww);

ff(n+1:hh-n, n+1:ww-n) = NoiseI(:, :, c);

% 滤波操作

for i = n+1:hh-n

for j = n+1:ww-n

% 当前窗口

dd = zeros(n\*2+1, n\*2+1);

% 求 d(m,n)

for s = i-n:i+n

for t = j-n:j+n

dd(s+n-i+1, t+n-j+1) = ((ff(i, j) - ff(s, t)))^2;

end

end

% 求 β(x,y)

bb = (sum(dd(:))) / ((n\*2+1)^2 - 1);

% 求 μ(m,n)/β(x,y)

dd = (exp((-dd / bb))) / bb;

sumd = sum(dd(:)) - 1 / bb;

gg = 0;

for s = i-n:i+n

for t = j-n:j+n

gg = gg + dd(s+n-i+1, t+n-j+1) \* ff(s, t);

end

end

gg = gg - (1 / bb) \* ff(i, j);

result(i-n, j-n, c) = gg / sumd;

end

end

end

% 转换为 uint8 并返回结果

result = uint8(result \* 255);

end

### 高斯滤波

**图像的高斯滤波是图像与高斯正态分布函数的卷积运算，适用于抑制服从正态分布的高斯噪声。**

**高斯滤波的基本原理是以某一像素为中心，在他的周围选择一个局部邻域，把邻域内像素的灰度按照高斯正态分布曲线进行统计，分配相应的权值系数，然后将邻域内所有点的加权平均值来代替原来的像素值，通过降低噪声点与周围像素点的差值以去除噪声点。**

**设一个二维零均值高斯滤波器的响应为，对一幅图像M×N的输入图像进行高斯滤波，获得输出图像的过程可以用离散卷积表示为**

****

**其中，k,l是根据所选邻域大小而确定的。高斯模板的特点是按照正态分布曲线的统计，模板上不同位置赋予不同的加权系数值。标准差σ是影响高斯模板生成的关键参数。标准差代表着数据的离散程度。σ越小，分布越集中，生成的高斯模板的中心系数远远大于周围的系数值，则对图像的平滑效果就越不明显；反之，σ越大，分布越分散，生成的高斯模板中的不同系数值差别不大，对图像的平滑效果越明显。**

**实现代码如下：**

function output\_img = gaussian\_filter(input\_img)

% GAUSSIAN\_FILTER 对彩色或灰度图像进行5x5高斯滤波

% 输入:

% input\_img: 待处理的图像（灰度或彩色）

% sigma: 高斯模板的标准差

% 输出:

% output\_img: 高斯滤波后的图像

% 参数初始化

sigma=1;

N\_size = 5; % 高斯模板大小为5x5

center\_N = (N\_size + 1) / 2; % 模板中心

[x, y] = meshgrid(1:N\_size, 1:N\_size); % 创建网格

% 计算高斯模板

G\_ry = exp(-((x - center\_N).^2 + (y - center\_N).^2) / (2 \* sigma^2));

G\_ry = G\_ry / sum(G\_ry(:)); % 归一化

% 判断输入图像的维度

if size(input\_img, 3) == 3

% 彩色图像

output\_img = zeros(size(input\_img), 'like', input\_img); % 初始化输出图像

for c = 1:3 % 分别处理 R、G、B 通道

output\_img(:, :, c) = imfilter(double(input\_img(:, :, c)), G\_ry, 'same', 'replicate');

end

else

% 灰度图像

output\_img = imfilter(double(input\_img), G\_ry, 'same', 'replicate');

end

% 转换为 uint8 类型

output\_img = uint8(output\_img);

end

## 八、基于区域生长算法的目标提取

区域生长是指从图像某个位置开始，使每块区域变大，直到被比较的像素与区域像素具有显著差异为止。具体实现时，在每个要分割的区域内确定一个种子点，判断种子像素周围邻域是否有与种子像素相似的像素，若有则将新的像素包含在区域内，并作为新的种子继续生长，直到没有满足条件的像素点时才停止生长。用户在使用系统时可以自己选择单个种子点生长。本系统采用与种子点灰度距离的度量作为生长准则。

区域生长算法使用的数据结构是队列，使用队列模拟生长的过程。在生长过程中，每次从队列中取出一个点，检查其周围邻域的像素（包括上下左右和对角线上的像素）。对于每个邻域点，如果其灰度值与当前像素灰度值的差异小于或等于设定的阈值，就认为它属于当前区域，并将其加入队列继续生长。此过程不断重复，直到队列为空，即没有更多的像素满足生长条件为止。

实现代码如下：

function segmentedImage = regionGrowingSegmentation(inputImage, seedPoint, threshold)

% 基于区域生长的图像分割

% 输入参数:

% inputImage: 输入图像（灰度图像）

% seedPoint: 种子点坐标 [row, col]

% threshold: 灰度值阈值，用于控制生长条件

% 输出:

% segmentedImage: 分割后的二值图像

% 检查输入图像的维度，确保为灰度图像

if size(inputImage, 3) == 3

inputImage = rgb2gray(inputImage); % 将彩色图像转换为灰度图像

end

inputImage = double(inputImage); % 将图像转换为 double 类型

% 检查种子点是否有效

[rows, cols] = size(inputImage);

if seedPoint(1) < 1 || seedPoint(1) > rows || seedPoint(2) < 1 || seedPoint(2) > cols

error('种子点越界');

end

% 初始化输出图像和访问标记

segmentedImage = false(rows, cols); % 输出二值图像

visited = false(rows, cols); % 记录访问过的像素

seedValue = inputImage(seedPoint(1), seedPoint(2)); % 种子点灰度值

% 初始化队列并添加种子点

queue = seedPoint; % 使用种子点初始化队列

visited(seedPoint(1), seedPoint(2)) = true;

% 区域生长

while ~isempty(queue)

% 取出队列中的第一个点

currentPoint = queue(1, :);

queue(1, :) = []; % 出队

r = currentPoint(1);

c = currentPoint(2);

% 标记当前点为分割区域

segmentedImage(r, c) = true;

% 遍历当前点的8邻域

for i = -1:1

for j = -1:1

if i == 0 && j == 0

continue; % 跳过中心点

end

newRow = r + i;

newCol = c + j;

% 检查是否越界

if newRow >= 1 && newRow <= rows && newCol >= 1 && newCol <= cols

% 检查是否已访问过

if ~visited(newRow, newCol)

% 判断当前点和相邻点灰度值差异

if abs(inputImage(newRow, newCol) - inputImage(r, c)) <= threshold

% 添加到队列

queue = [queue; newRow, newCol];

visited(newRow, newCol) = true;

end

end

end

end

end

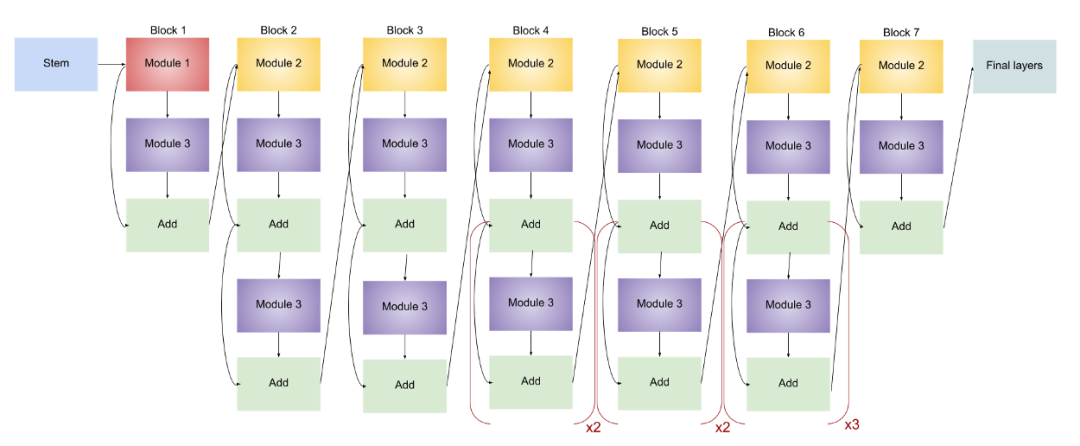
end

end

## 九、基于EfficientNet-B2模型的鸟类分类任务

### 9.1 EfficientNet模型

EfficientNet是由Google Brain提出的一种高效的卷积神经网络（CNN）架构，其核心思想是在网络的深度、宽度和输入分辨率这三方面同时进行复合缩放，从而实现更高的计算效率和更好的精度。EfficientNet的提出是为了应对计算资源和模型精度之间的权衡问题，通过创新的模型设计在减少计算成本的同时提升模型的性能。



EfficientNet的主要创新之一是复合缩放方法。传统的神经网络设计通常在深度、宽度和输入分辨率上独立地进行调整，而EfficientNet通过复合缩放将这三个维度进行有机结合，创建了一个可以在计算资源与精度之间平衡的方案。复合缩放的目标是根据模型的初始版本来优化网络的深度、宽度和分辨率，找到最佳的缩放系数，避免传统方法中可能出现的参数过多或过少的情况。

相比于传统的CNN架构，EfficientNet在相同计算资源下能够提供更好的精度。它能够在减少计算量和参数量的同时提高准确率，从而减少计算开销。

### 9.2 训练模型

模型架构选择预训练的EfficientNet-B2模型，预训练权重来自大规模数据集（ImageNet），能够加速训练并提升性能。EfficientNet是一种高效的卷积神经网络架构，通过复合缩放方法同时优化网络的深度、宽度和分辨率，使其在计算效率和精度之间取得平衡。在本任务中，我们冻结了EfficientNet-B2的预训练权重，仅训练其分类头部分，以适应我们的特定任务。

数据预处理是深度学习中提高模型性能的关键步骤。在本分类任务中，我们对训练集应用了数据增强技术，包括随机水平翻转、随机垂直翻转和颜色扰动等。验证集和测试集则进行了标准化处理，以减少训练数据偏差对模型性能的影响。

模型主干部分：

1. model = torchvision.models.efficientnet\_b2(pretrained=True)
2. for param in model.parameters():
3. param.requires\_grad = False
4. n\_inputs = model.classifier[1].in\_features
5. model.classifier = nn.Sequential(
6. nn.Linear(n\_inputs, 2048),
7. nn.SiLU(),
8. nn.Dropout(0.3),
9. nn.Linear(2048, 2048),
10. nn.SiLU(),
11. nn.Dropout(0.3),
12. nn.Linear(2048, len(classes))
13. )

训练过程中采用了AdamW优化器，它能够有效地控制学习率并减少过拟合的风险。同时，使用了带标签平滑的交叉熵损失函数，标签平滑能够让模型对类别不确定性进行建模，避免过于自信的预测。为了避免模型过拟合，还对全连接层应用了Dropout正则化。

图9.2.1展示了基于EfficientNet-B2模型训练的过程，一共训练19轮，训练集上每轮的批次大小为256。训练结束时，在训练集上的准确度达到95.28%，在验证集的准确度达到89.82%。相较于训练集的准确度，验证集准确度略低，这通常是因为验证集代表了模型未见过的新数据，但差距不大，模型泛化能力较好。

图9.2.2展示了训练过程中损失函数的变化，训练过程中损失在不断下降，说明模型在训练集上的拟合效果越来越好。验证损失在后期有轻微波动，可能表明模型开始出现过拟合现象。训练准确率和验证准确率在后期都趋于平稳，说明模型在一定程度上收敛了。

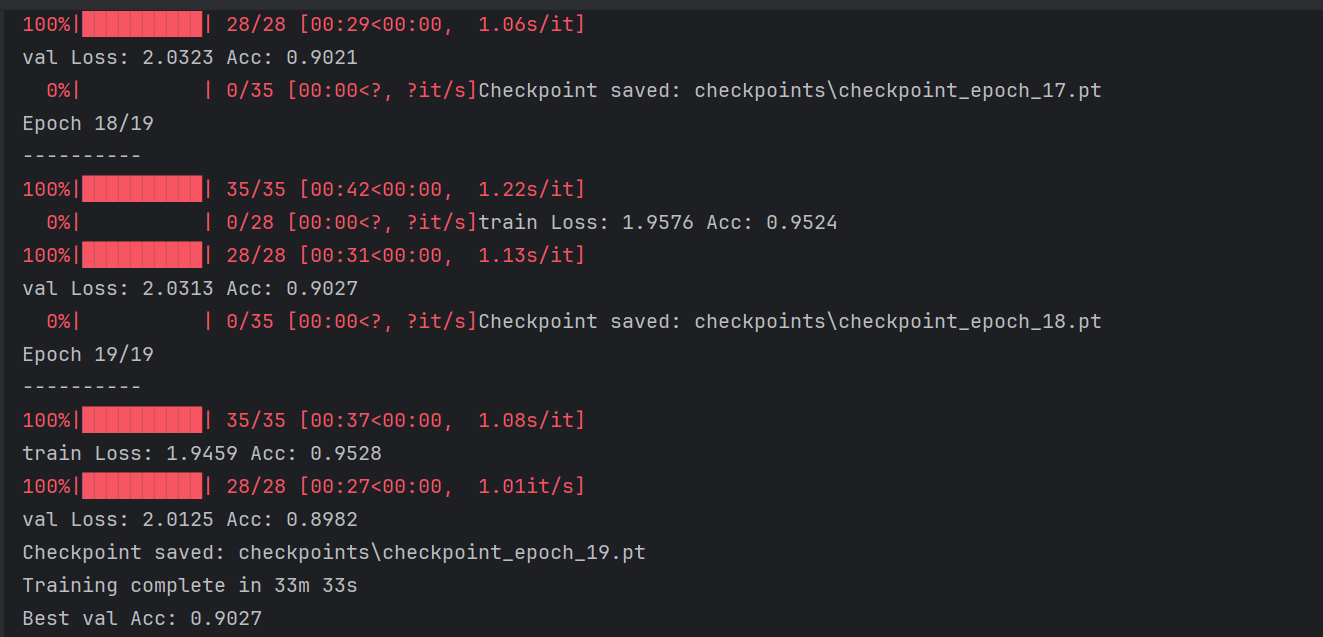


图9.2.1 EfficientNet-B2训练过程

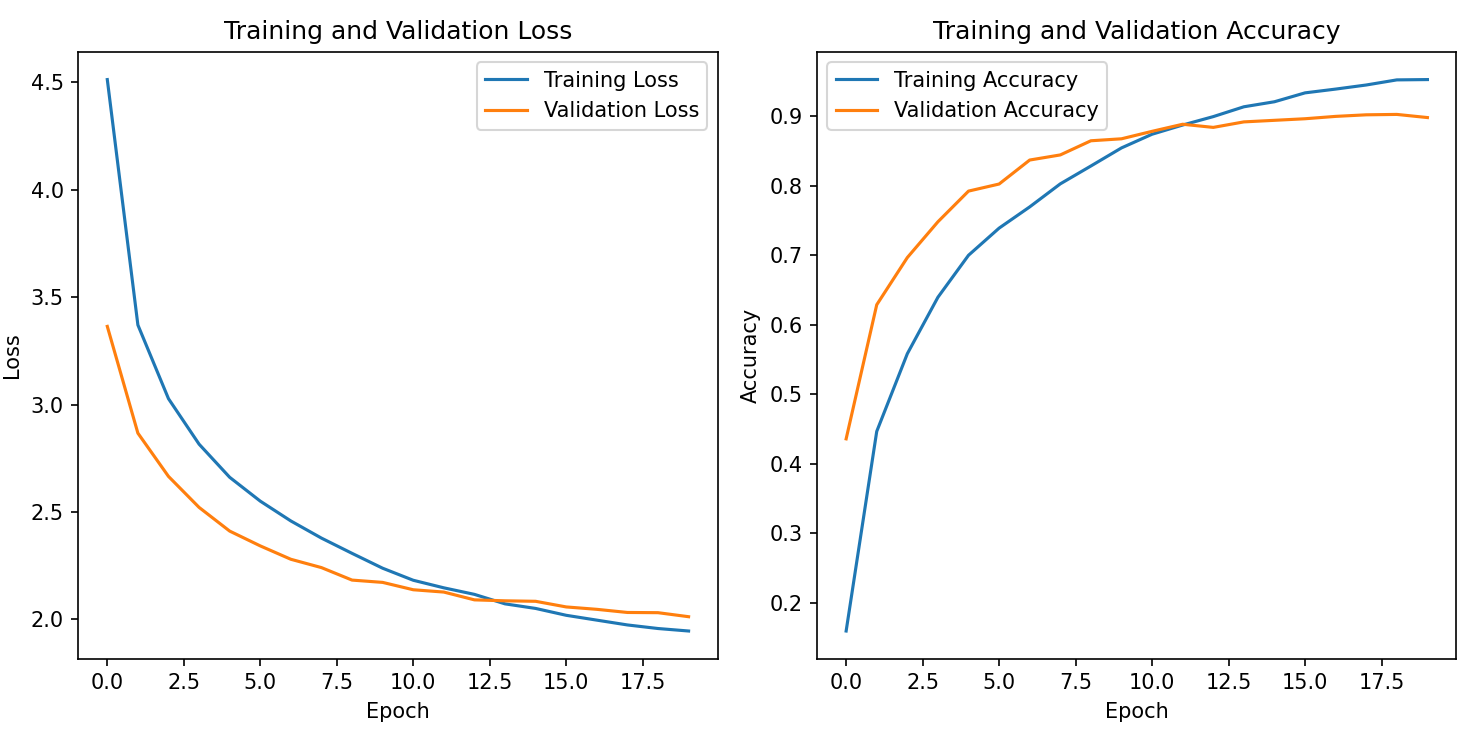
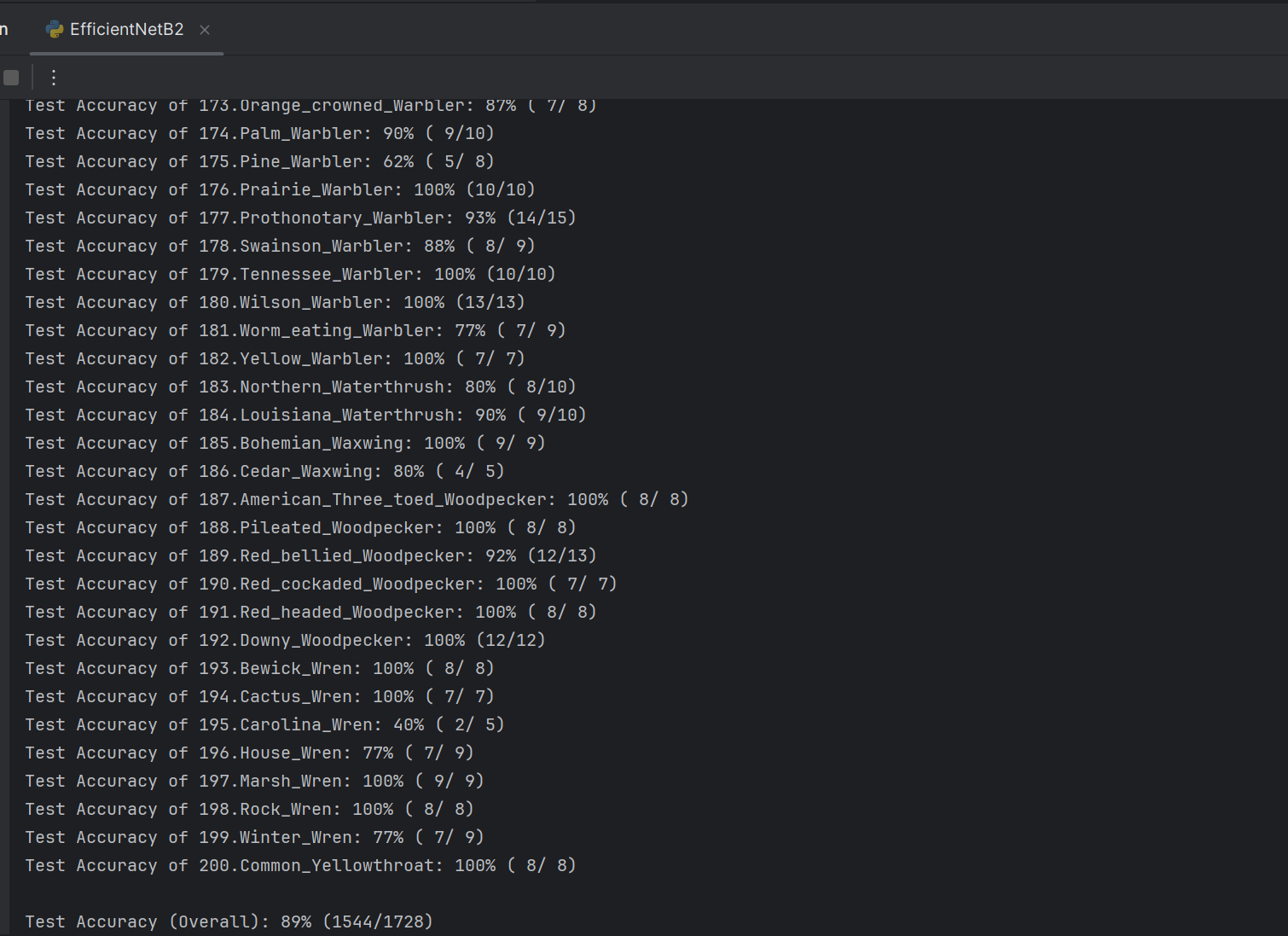


图9.2.2 EfficientNet-B2训练过程损失函数变化

图9.2.3展示了在测试数据集上对不同鸟类的识别准确率。图中列出了多种鸟类的名称及其对应的识别准确率（以百分比表示），以及每种鸟类测试样本的数量（括号内的两个数字分别表示正确识别的样本数量和总样本数量）。



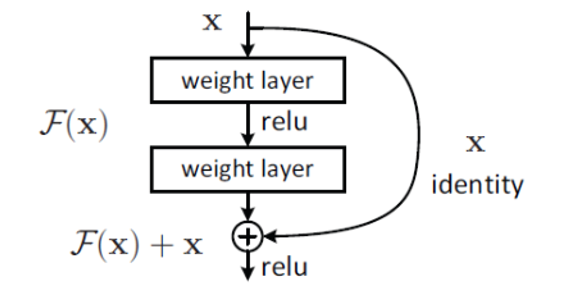
## 十、基于Resnet18模型的鸟类分类任务

### Resnet模型

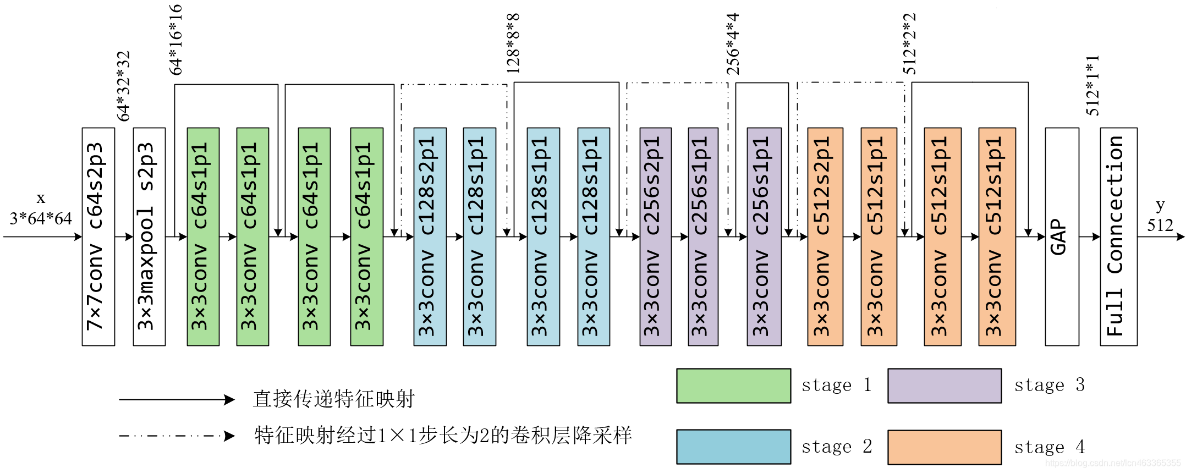
残差神经网络(ResNet)是由微软研究院的何恺明、张祥雨、任少卿、孙剑等人提出的。ResNet 在2015 年的ILSVRC（ImageNet Large Scale Visual Recognition Challenge）中取得了冠军。残差神经网络的主要贡献是发现了“退化现象（Degradation）”，并针对退化现象发明了 “快捷连接（Shortcut connection）”，极大的消除了深度过大的神经网络训练困难问题。其核心思想是通过添加额外的连接来解决深度神经网络训练中的梯度消失和梯度爆炸等问题，从而允许构建非常深的神经网络。

ResNet通过引入快捷连接，允许某一层的输出直接跳过一个或多个层，连接到后续层的输入。这样做的好处是，即使某些层不做任何有意义的变换，它们仍然可以传递之前层的信息，而不会对梯度产生过多的损失。这可以用一个公式来表示：





ResNet18的基本含义是，网络的基本架构是ResNet，网络的深度是18层。但是这里的网络深度指的是网络的权重层，也就是包括池化，激活，线性层。而不包括批量化归一层，池化层。图—————就是一个ResNet18的基本网络架构，其中并未加入批量化归一和池化层。



### 训练模型

模型架构使用的是预训练的 ResNet-18 网络，预训练权重来自大规模数据集（ImageNet），能够加速训练并提升性能，并冻结卷积层的权重，只训练最后的全连接层。我们修改了全连接层的结构，通过增加层数和神经元数量，使其能够处理更复杂的分类任务。

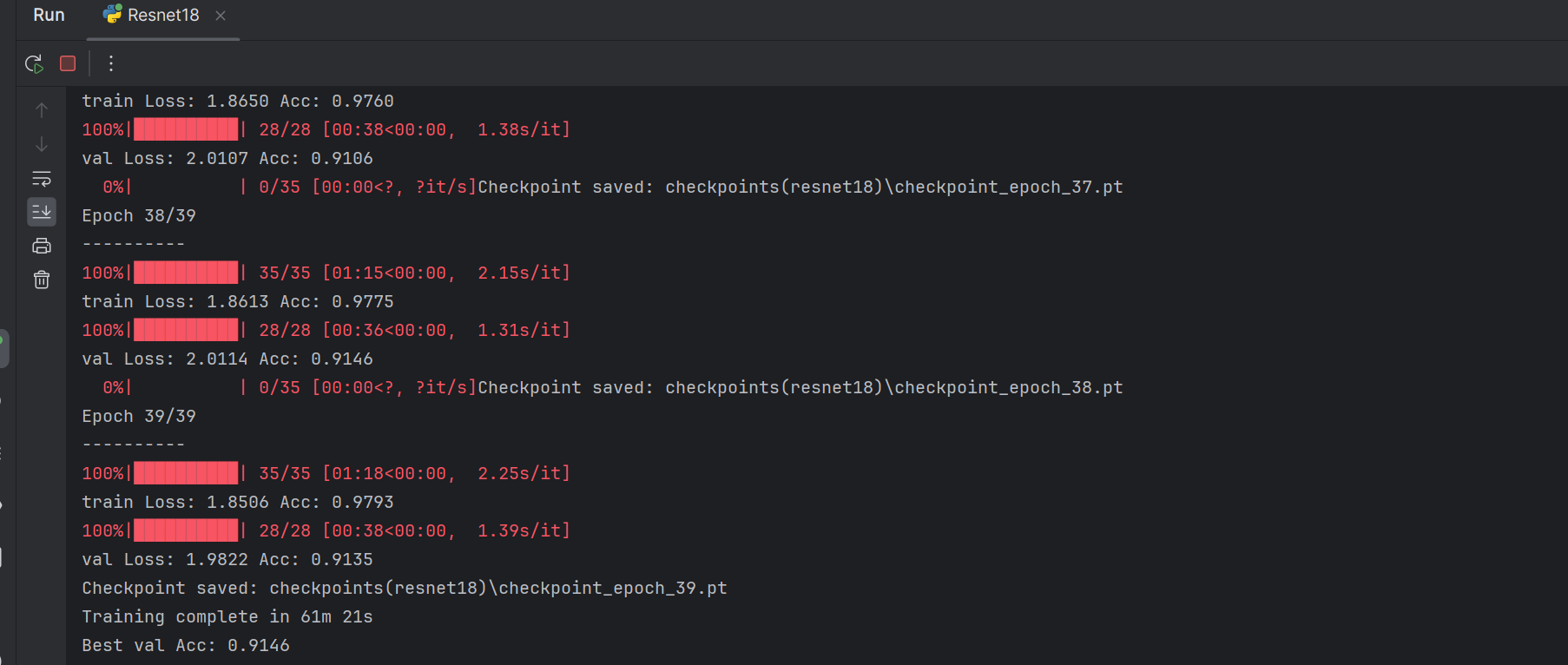
模型主干部分：

1. model = torchvision.models.resnet18(pretrained=True)
2. for param in model.parameters():
3. param.requires\_grad = False
4. n\_inputs = model.fc.in\_features
5. model.fc = nn.Sequential(
6. nn.Linear(n\_inputs, 1024),  # Increase the size of the first fully connected layer
7. nn.SiLU(),
8. nn.Dropout(0.3),
9. nn.Linear(1024, 2048),  # Add another fully connected layer
10. nn.SiLU(),
11. nn.Dropout(0.3),
12. nn.Linear(2048, 2048),  # Add another fully connected layer
13. nn.SiLU(),
14. nn.Dropout(0.3),
15. nn.Linear(2048, len(classes))  # Adjust the output size to match the number of classes
16. )

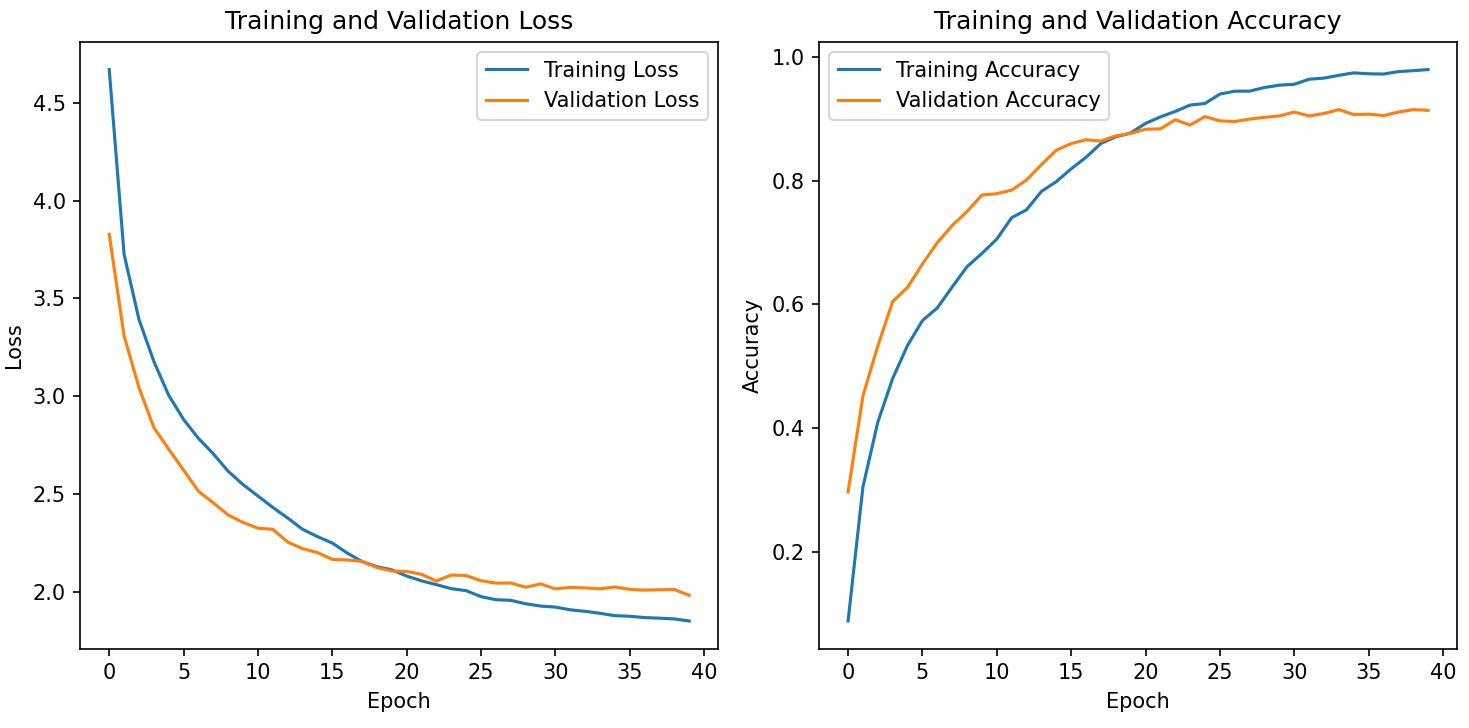
Dropout 层，丢弃率设置为 0.3，帮助防止过拟合；SiLU激活函数，具有良好的非线性表达能力，且在许多任务上性能优于 ReLU。数据预处理与增强、损失函数和优化器与训练EfficientNetB2模型时使用的一致，此处不再赘述。

训练过程如下图，一共训练了40轮，图————训练集的损失（蓝色线）在开始时很高，随着训练轮次的增加逐渐下降。这表明模型在训练过程中不断学习，逐渐减少预测误差。验证集的损失（橙色线）呈现下降趋势，但下降幅度相对较小，并且在后期趋于平稳。这说明模型在验证集上的表现也在改善，但可能存在过拟合的风险，因为验证集损失没有像训练集损失那样持续大幅下降。

训练过程中，在大约第 30 轮之后，训练集损失和验证集损失都趋于平稳，意味着模型已经收敛，进一步训练可能不会带来显著的改进。在第33轮时，验证集损失最小，保存为模型checkpoint\_epoch\_33.pt。



在训练集和验证集上的损失函数、准确率的变化：



在测试集上的测试：

