

队伍编号	
题号	

## 基于 U-Net 和 Faster R-CNN 模型的甲骨文图像自动分割与识别

### 摘要

随着数字化保存技术的发展，甲骨文的数字化研究愈发重要。本研究针对甲骨文图像的处理面临的主要难点，包括图像不清晰、点状噪声、人工及固有纹理干扰，以及形近字和异形字问题，提出了一套综合图像分割和识别技术的解决方案。

针对问题一，通过图像预处理显著提高甲骨文拓片图像的细节清晰度，并对干扰元素进行了识别和处理。首先，通过对原始拓片图像进行了尺寸统一和灰度化处理，确保了后续处理的一致性和效率，采用锐化、Gabor 滤波和直方图均衡化等方法增强图像边缘的细节，突出特定方向的纹理特征。接着，利用局部二值模式（LBP）算法有效识别干扰元素，并对于点状噪声和纹理噪声，分别采用二值化、中值滤波和 Canny 边缘检测技术进行处理，为后续的分析提供了清晰的基础图像。

针对问题二，采用了 CNN 和 U-Net 模型组合进行特征提取和图像分割，以实现高精度的单字分割。首先，利用卷积神经网络（CNN）对预处理过的甲骨文图像进行了深度特征提取，通过学习图像的内在特征来增强模型对甲骨文特定形态的识别能力。接着，应用 U-Net 模型进行图像分割，通过编码器-解码器架构配合跳跃连接，精准恢复图像细节，实现了对甲骨文单字的高效精准分割。最后，采用五折交叉验证方法进行模型评估，结果显示精确度达到 87.5%，F1 分数为 0.83，充分证明了该组合模型在甲骨文单字分割方面的准确性。

针对问题三，利用训练好的 U-Net 模型对 200 张甲骨文拓片图像实现了单字分割。通过对训练集中拓片图像进行预处理，应用问题二中经过验证的 U-Net 模型进行处理，输出的分割后图像表明了模型的实用性和可靠性。

针对问题四，采用了 Faster R-CNN 模型对单字分割后的甲骨文图像进行文字识别。对从 U-Net 模型分割得到的单字图像，应用 Faster R-CNN 模型对 50 张拓片图像进行目标检测与识别，结果表明其能够处理大量的图像数据，有效支持了甲骨文的自动化文本提取和识别任务。

本研究通过一系列图像处理和深度学习技术，有效解决了甲骨文图像处理中图像预处理和精确分割方面等多项挑战，为甲骨文的保护和研究提供技术支持。

**关键词：**图像分割与识别；LBP 算法；Canny 算子；U-Net 模型；Faster R-CNN 模型

# 目 录

第一章 问题重述 .....	1
1.1 问题背景 .....	1
1.2 问题概述 .....	1
第二章 问题分析 .....	2
2.1 对问题的整体分析 .....	2
2.2 对问题的具体分析 .....	2
2.2.1 针对问题一 .....	2
2.2.2 针对问题二 .....	2
2.2.3 针对问题三 .....	2
2.2.4 针对问题四 .....	2
2.3 解决思路及对策 .....	2
第三章 符号说明 .....	4
第四章 模型的建立与求解 .....	5
4.1 问题一的求解 .....	5
4.1.1 图像预处理 .....	5
4.1.2 基于 LBP 算法的干扰元素特征提取 .....	8
4.1.3 基于 Canny 算子的干扰元素判别与处理 .....	10
4.2 问题二的求解 .....	15
4.2.1 图像增强 .....	15
4.2.2 基于 CNN 模型的图像特征提取 .....	15
4.2.2 基于 U-Net 模型的图像分割 .....	17
4.2.3 模型评估 .....	19
4.3 问题三的求解 .....	21
4.3.1 基于 U-Net 模型的自动单字分割 .....	21
4.3.2 自动单字分割的实现 .....	21
4.4 问题四的求解 .....	23
4.4.1 基于 Faster R-CNN 模型的文字识别 .....	23
4.4.2 甲骨文文字自动识别的实现 .....	24
第五章 模型评价与推广 .....	26
5.1 模型的评价 .....	26
5.2 模型的推广 .....	26
参考文献 .....	28
附录 .....	30
1. 环境说明 .....	30
2. 核心代码 .....	30

# 第一章 问题重述

## 1.1 问题背景

甲骨文是中国古代的一种文字形式，广泛出现于商代和西周时期的甲骨文书写在龟甲和兽骨上，记录了古代社会的重要信息，对于研究中国古代历史、文化、语言等具有重要价值。然而，由于龟甲和兽骨等材料的易损性，以及历经千年的风化和磨损，甲骨文拓片图像常常存在点状噪声、人工纹理、固有纹理等干扰因素<sup>[1]</sup>，给甲骨文文字的识别和研究带来了挑战。

## 1.2 问题概述

针对甲骨文拓片图像中存在的点状噪声、人工纹理、固有纹理等干扰因素，导致文字分割受到不同程度的影响，需要借助人工智能和大数据技术来进行甲骨文文字的识别和分析。具体而言，需要开展以下工作：

1. 噪声去除与图像增强：针对甲骨文拓片图像中的点状噪声，可以利用图像处理技术进行噪声去除，并通过图像增强技术提升图像质量，使得甲骨文文字更加清晰可见。
2. 纹理分割与特征提取：对于人工纹理和固有纹理等干扰因素，可以采用图像分割算法将甲骨文文字与背景分离，同时提取甲骨文特征，以便后续的识别和分析。
3. 深度学习模型训练与识别<sup>[2]</sup>：基于大数据技术，可以构建深度学习模型，通过训练模型识别甲骨文文字，并结合图像处理结果，实现对甲骨文拓片图像中文字的准确提取和解读。

通过以上工作，可以有效克服甲骨文拓片图像中的干扰因素，实现对甲骨文文字的自动分割和识别，为甲骨文研究提供更为有效的工具和方法。

## 第二章 问题分析

### 2.1 对问题的整体分析

随着时代的发展与进步，人工智能与大数据技术在甲骨文信息处理领域得到广泛应用，这不仅使得甲骨文的全息性及数字化工程得到了跨越性的发展，同时也为甲骨文的研究带来了重要而深远的意义。然而，甲骨拓片常常受到干扰元素的影响，且由于甲骨文图像来源众多，图像的分割质量会受到不同程度的影响。因此，提高甲骨文图像中独立文字区域的分割与识别精度，是当前研究中一个亟待解决的重要难题。

### 2.2 对问题的具体分析

#### 2.2.1 针对问题一

本文需要对已有的三张甲骨文原始拓片图片进行了以下处理：调整图像大小、转换为灰度图像并做清晰化处理，使得图像格式统一，细节突出，增强图像结构可见性；运用 LBP 算法对干扰元素进行判别；分别采用二值化、中值滤波器分离目标物体和背景，去除点状噪声；Canny 算子边缘检测，区域分割定位甲骨文文字区域，去除人工纹理和固有纹理。图像预处理有助于提升甲骨文拓片图像分割的质量，后续将在图像预处理后的甲骨文拓片图像上做进一步分析。

#### 2.2.2 针对问题二

结合对问题一的分析，对干扰元素处理后的甲骨文拓片图像运用 CNN（卷积神经网络）进行特征提取。进而用 U-Net 模型对甲骨文图像实现自动单字分割，其具有编码器-解码器结构和跳跃连接，适用于单字识别分割问题。此外，引入五折交叉验证从不同角度全面评估模型的性能，计算评估指标均值，确保模型的稳健性和可靠性。

#### 2.2.3 针对问题三

在问题二模型的基础上，对附件 3 中的 200 张甲骨文原始拓片图像进行自动单字分割，验证模型的实际运行效果。将分割结果保存为.xlsx 文件并上传至竞赛平台。

#### 2.2.4 针对问题四

经过前面问题对甲骨文原始拓片图像单字分割的分析与研究，将进一步对单字分割的图片进行文字识别，本文运用 Faster R-CNN（Faster Region-based Convolutional Network）模型对测试集中 50 张甲骨文原始拓片图像进行快速目标检测，文字自动识别，并展示测试结果图。

### 2.3 解决思路及对策

针对甲骨文图像分割与识别的存在的难点，本研究采取如下的对策逐一解决问题：

#### 1. 预处理阶段

甲骨文图像的预处理关键在于强化文本与背景的对比，提高识别算法对图像细节的感知能力。选择 LBP 算法是由于其对纹理信息的灵敏度，能够在不受光照影响的条件

下准确识别图像中的干扰纹理。Canny 算子的应用则基于其能力在保持边缘定位准确的同时，最大限度地减少错误边缘响应。此外，二值化和中值滤波处理在清除噪声的同时，能够为下一步的边缘检测创造了理想条件。

2. 图像分割阶段

甲骨文文字分割注重文字定位的准确性和分割文字的完整性。这意味着分割过程中需要精确识别每个文字的边界，避免字体的断裂或合并。CNN 模型在处理高度复杂的图像数据时特征提取能力卓越，同时，U-Net 模型因其特殊的跳跃连接结构，能够保留低层的精细特征，从而在高层重建时，能够确保文字的完整性不被破坏，有助于有效地保留图像中的重要特征，即使在背景复杂或样本数量有限的情况下也能够准确地识别和分割单个字符。

3. 文字识别阶段

文字识别的效果在很大程度上取决于图像特征的代表能力，然而甲骨文图像可能受到各种干扰因素的影响，如裂纹、斑点和色彩不均等，导致其特征容易被干扰元素的特征所掩盖。因此，选取的模型必须具备强大的抗干扰能力，以确保识别结果的准确性不被这些因素所影响。深度学习模型，特别是如 Faster R-CNN 这类网络，通过深层卷积提取丰富的视觉特征，能够更好地表征甲骨文的独特符号和笔画。该模型基于其高效率和准确率的两阶段目标检测能力，结合 RPN 和 ROI Pooling，能够提高对甲骨文字符的识别速度和准确性，特别是在处理细小物体和应对多尺度挑战时。相较于其他单阶段检测方法，Faster R-CNN 在保持高精度的同时显著提高了运行效率，在大量文献资料的自动化识别问题上，可以大幅提升甲骨文图像的文字识别效果，使得从图像中自动提取和解析甲骨文文本成为可能。

具体研究思路和方法如图 1：

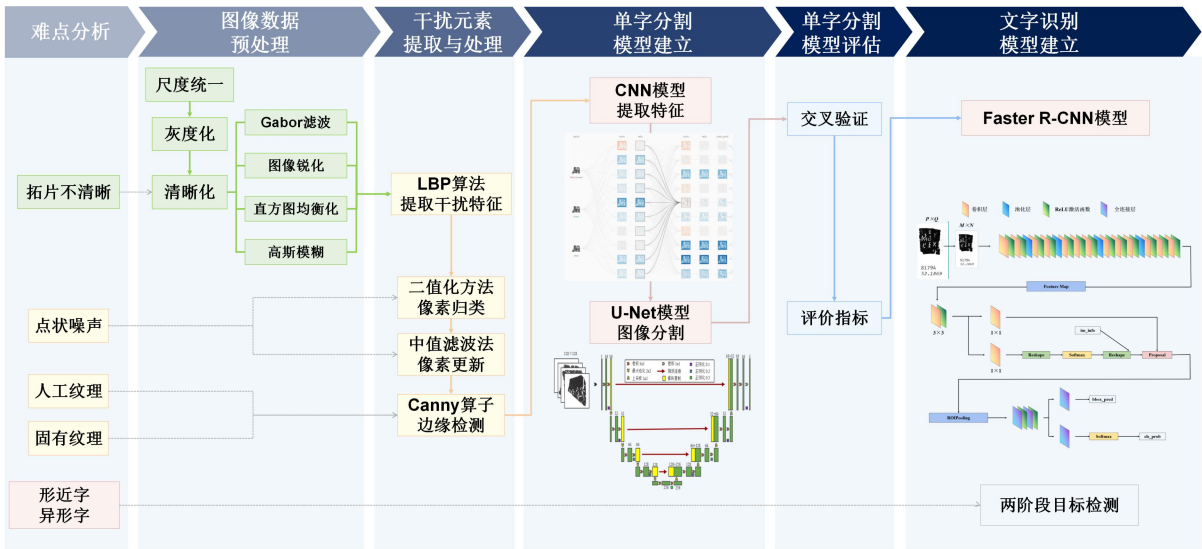


图 1 研究流程图

### 第三章 符号说明

表 1 符号说明

符号	说明
$\sigma$	滤波器参数
$I(x,y)$	原始图像
$S(x,y)$	高斯分布函数
$H(x,y)$	滤波降噪后的新图像
$w_i(x,y), i = x,y$	$x,y$ 方向的偏导数
$w(i,j)$	$(i,j)$ 处的梯度幅值
$\theta(x,y)$	$(i,j)$ 处的梯度方向
$W_{filter}$	CNN 权重参数矩阵
$H_1$	原图的长
$W_1$	原图的宽
$F_H$	CNN 权重参数矩阵的行维数
$F_W$	CNN 权重参数矩阵的宽维数
$P$	CNN 边界填充
$S$	CNN 指定步长

## 第四章 模型的建立与求解

### 4.1 问题一的求解

甲骨文图像分割与识别的关键挑战在于拓片本身的复杂性，其通常带有自然和人为的纹理噪声，且文字形态多样、经常残缺，这对于识别算法的准确性提出了严峻要求。因此，需要一个能够强化图像特征并有效降噪的预处理模型，为甲骨文图像的分割与识别打下了坚实的基础，使后续的深度學習模型能够在一致和清晰的数据上进行训练，从而提高最终识别的准确率。

#### 4.1.1 图像预处理

现有甲骨文可以分为手写甲骨文图像、拓片甲骨文和临摹甲骨文图像三类，见图 2。手写甲骨文图像是研究者手工书写得到的高清图像，其图像获取简单，特征明显。临摹甲骨文是由甲骨文专家临摹甲骨文拓片得到的高清图像，若要大量得到此类图像，既耗时耗力，又专业性极强，因此难以实现大量临摹。拓片甲骨文来源于龟骨与兽骨的原始拓片图像，但由于客观条件的限制，这些图像样本难以大量获取，并且常常伴有严重的噪声和残缺等问题，这使得深度学习模型在处理这类数据集时，难以达到较高的准确率。



图 2 甲骨文图像

拓片甲骨文的图像分割与文字识别具有一定的难度，具体难度如下：

(1) 拓片甲骨文的书写载体是骨骼，骨骼自身有天然的纹路，此外龟甲本身纹路、与占卜过程中灼烧产生的裂纹、保存过程中的破损、拓印技术的好坏等多种原因导致部分文字残缺，且噪声严重<sup>[3]</sup>，部分噪声类别如表 2。

表 2 甲骨文识别噪声类型

噪声类型	具体表现
形态不一	甲骨文在不同人的笔下写出的形态有所差异
字体残缺	字体经常会残缺部首
字体清晰度有差异	由于拓印技术等造成的字体清晰度不一致
形似字的影响	部分甲骨文文字形似，导致识别障碍
其他	由于拓印或者龟甲残缺等造成的笔画粘连或者缺失

(2) 现有的文字识别领域的深度学习方法大量依赖于标记图像，在拓片上的分类精度普遍较低，目前仍旧无法克服拓片甲骨文类内样本少、类间样本不均衡的问题<sup>[3]</sup>。

(3) 甲骨文拓片中的文字类别多，部分文字出现频率低。

基于以上拓片甲骨文存在的噪声，容易为后续图像识别造成困难，因此，需要对三张给定的甲骨文原始拓片进行图像预处理并提取特征。

### 1. 图片尺寸调整和灰度化处理

将所有原始拓片图像均调整到  $256 \times 256$  像素并转换为灰度图<sup>[2]</sup>，这样的处理不仅统一了图像处理的输入规格，也有助于降低计算复杂度，同时灰度化处理减少了数据处理的复杂性，为后续的图像清晰化提供了便利。



图 3 尺度调整和灰度化处理后的甲骨文拓片图像

### 2. 图像清晰化

应用 Gabor 滤波方法，高斯模糊方法，直方图均衡化方法与锐化方法，对图像进行清晰度处理，挑选出最优的图像处理方法，从而有助于后期的文字识别与分割。

(1) Gabor 滤波<sup>[4]</sup>：Gabor 函数可用于边缘提取，是一个线性滤波器，其频率和方向表达方式类似于人类视觉系统，且该图像边缘主要为直线，方向清晰。Gabor 小波可以在不同尺度和方向上提取纹理特征，同时降低因光照变化和噪声带来的影响，帮助突出特定方向的纹理特征。

具体流程如下：①定义一组 Gabor 滤波器，这些滤波器在空间与和频率域都具有高斯分布的性质；②将原始图像与每个 Gabor 滤波器进行卷积操作，这个卷积可以在特定方向和频率上提取纹理特征，同时可降低因光照变化和噪声带来的影响。

(2) 高斯模糊<sup>[5]</sup>：又称高斯平滑，实质上是通过对图像与正态分布进行卷积来实现的，由于正态分布又被称为高斯分布，故被称为高斯模糊。在此过程中像素值被替换为其周围像素值的加权平均，权重由高斯函数确定，这样可以减少噪声以降低细节层次，从而生成一个更加平滑模糊的图像，相当于产生“模糊”效果。但该方法局限性明显，会模糊文字边缘。

具体流程如下：①定义一个高斯核，该核是一个二维的高斯函数，高斯核函数包含均值（平滑程度）和标准差（模糊程度）两个参数；②将原始图像与定义好的高斯核进行卷积操作，在此过程中像素值替换为其周围像素值的加权平均，权重由高斯函数确定。



（3）直方图均衡化：是一种常用的图像增强技术，旨在增强图像的对比度和视觉效果。其通过重新分配图像像素的灰度值，使得图像的灰度分布更均匀，近似均匀分布，从而增强图像的细节和对比度。

具体流程如图 4 所示：①扫描原始图像的每个像素，计算原始图像的灰度直方图；②计算灰度直方图的累计分布函数（CDF）；③将 CDF 做归一化处理，使其取值范围在 0 到 255 之间；④构建映射函数，使得原始图像的每个像素灰度值映射到均衡化后的灰度值；⑤灰度值映射后的图像像素重新分配组合，生成均衡化后的图像。

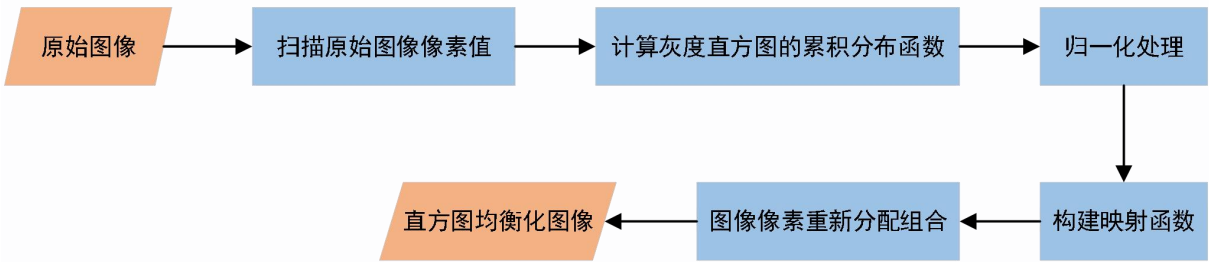





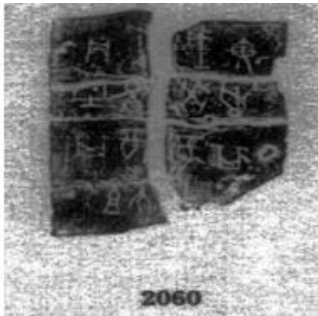





图 4 直方图均衡化流程图

（4）锐化图像：锐化滤波器是图像锐化的关键步骤，常见的锐化滤波器有拉普拉斯滤波器和高通滤波器等，通过对图像进行卷积操作，突出图像的边缘和细节，从而增强图像的边缘，使细节更加突出。

经上述步骤处理，甲骨文原始拓片的图像处理结果图见表 3：

表 3 甲骨文原始拓片图像处理

	图片 1	图片 2	图片 3
原图			
Gabor 滤波			

高斯模糊			
直方图均衡化			
锐化图像			

由上表可看出对甲骨文原始拓片图像经直方图均衡化以及锐化的处理后能够提供更清晰图像细节和边缘，确保甲骨文图像的特征得到最佳的提取和展示。

#### 4.1.2 基于 LBP 算法的干扰元素特征提取

由于不同的干扰因素的处理方法有所不同，所以需要对甲骨文拓片的干扰因素进行区分。接下来进行图像特征提取，即对干扰元素进行特征提取，以方便之后的图像分割与文字识别。

采用局部二值模式（Local Binary Pattern, LBP）算法<sup>[6]</sup>，通过比较每个像素与其周围像素的灰度值，并将结果编码为二进制数来描述图像的纹理信息，从而输出对每个甲骨文拓片的干扰因素的类型识别。原始的 LBP 算子定义在像素的  $3 \times 3$  邻域内，以邻域中心像素为阈值，相邻的 8 个像素的灰度值与邻域中心的像素值进行比较。若周围像素大于中心像素值，则该像素点的位置被标记为 1，否则为 0。这样， $3 \times 3$  邻域内的 8 个点经比较可产生 8 位二进制数（通常转换为十进制数即 LBP 码，共 256 种），即得到该窗口中心像素点的 LBP 值，并用这个值来反映该区域的纹理信息。

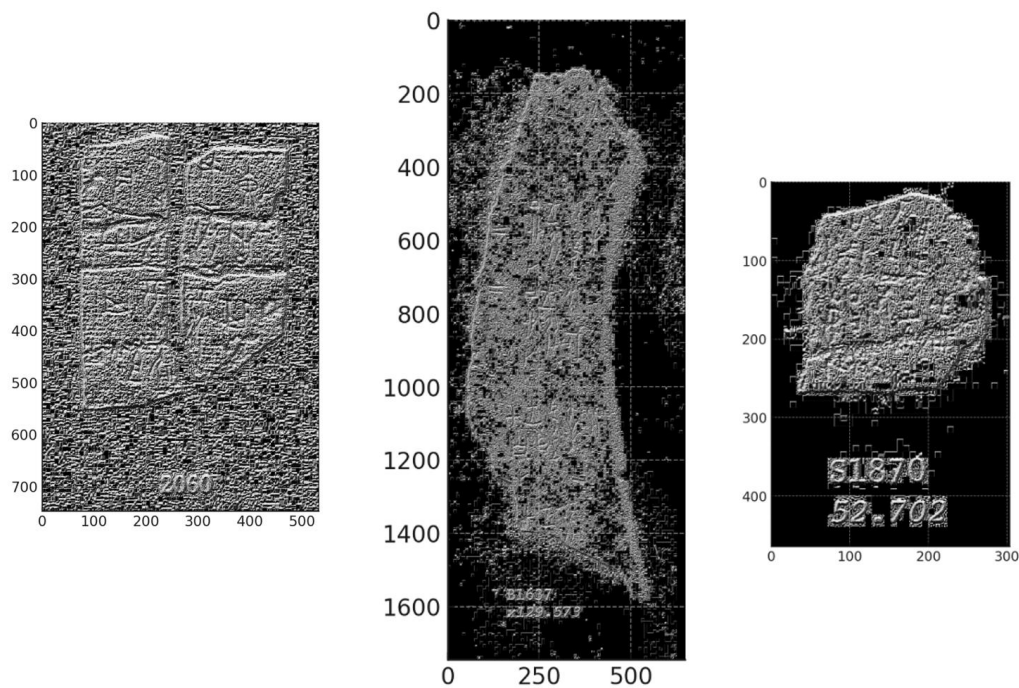


图 5 LBP 算法干扰元素特征提取

在实际应用中，LBP 算法的具体实现可能会有所不同，例如 LBP 编码起始点的选择、邻近像素的选择（可以选择邻近 8 个像素，也可以选半径为  $r$  圆上的点）、以及图像边界像素的处理等，这些都会影响到最终的 LBP 特征提取结果。本文对于干扰元素识别结果图如图 6 所示。

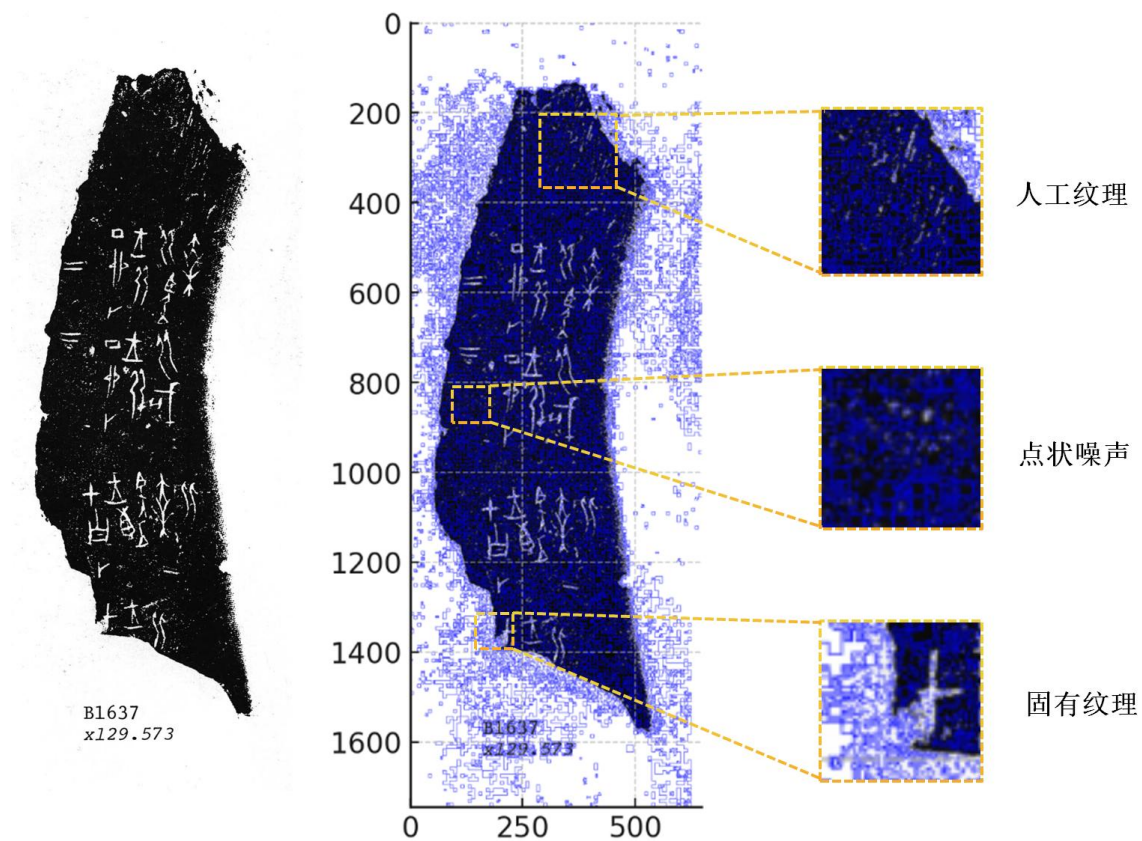


图 6 LBP 算法干扰元素特征判别

如图所示，图像的左侧为原始的甲骨文拓片图像，右侧则展示了 LBP 算法处理后的结果。可以观察到不同区域的纹理特征有着不同的表现，在图像的上方区域存在较为连续的蓝色区块，表明此处的纹理特征比较单一，纹理的 LBP 值比较低，可能对应了甲骨文的背景区域；在图像中部，有明显的蓝色线条结构，这些线条代表了较高的 LBP 值，指出了甲骨文的文本线条可能的位置；交错的区域表明那里有较复杂的纹理变化，表明该部分是文本与背景交界的部分。

进一步放大观察，可以看到在边缘检测的结果中，人工纹理和固有纹理的差异被清晰地标识了出来。在放大的右下角区域，可以看到纹理特征在这一小块区域内的细微差异，这些细节可能是由甲骨文的刻划工艺或者材料的固有特性造成的。

通过 LBP 算法的处理，图像中的每个像素都被赋予了一个 LBP 值，这些值的分布与原始图像中的纹理特征密切相关，从而为后续的干扰元素去除和文字识别提供了有效的特征描述。这种方法不仅可以帮助我们识别和区分不同类型的干扰因素，如点状噪声、人工纹理和固有纹理等，还可以为后续的图像分割和文字识别提供精确的特征指导。

### 4.1.3 基于 Canny 算子的干扰元素判别与处理

接下来考虑进行甲骨文拓片的干扰元素进行处理。首先利用二值化<sup>[7]</sup>与中值滤波方法<sup>[8]</sup>，进行点状噪声的识别与处理，其次利用 Canny 边缘检测<sup>[9]</sup>进行人工纹理与固有纹理的处理，下面介绍这三种图像处理方法以及结果。

(1) 运用二值化方法<sup>[7]</sup>，其具有简单、容易实现、时间复杂程度低等优点，具体流程如图 7 所示：

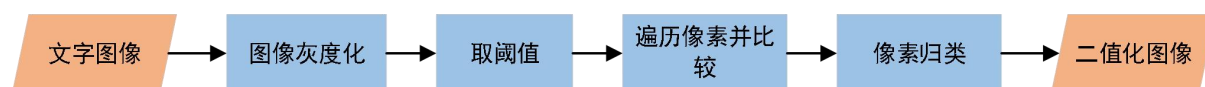


图 7 二值化方法流程图

①图像灰度化：如果原图像是彩色图像，需要将其转换为灰度图像。这是因为二值化操作通常是对灰度图像进行的。在灰度图像中，每个像素只有一个亮度值，范围从 0（黑色）到 255（白色）。

②取阈值：选择一个合适的阈值，这里取阈值为 127。阈值的选取将直接影响到图像分割的准确性。如果图像的前景和背景灰度值分布均匀，可以使用灰度直方图的谷底作为阈值。此外，也可以使用 OTSU 算法（最大类间方差法）来自动确定最佳阈值。

③遍历像素并比较：遍历灰度图像中每个像素，将其灰度值与选定的阈值进行比较。

④像素归类：根据比较结果，将像素归类为前景（白色）或背景（黑色）。具体来说，如果像素的灰度值大于或等于阈值，则将其设置为白色；否则，将其设置为黑色。



(2)运用中值滤波方法<sup>[8]</sup>，用来去除二值化可能引入的噪点，具体流程如图 8 所示：

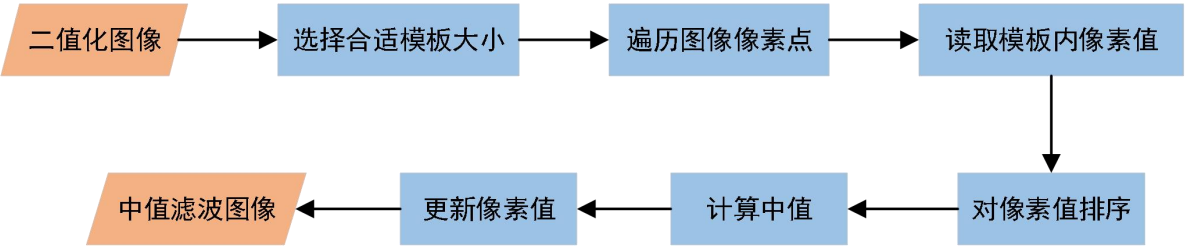


图 8 中值滤波方法流程图

①选择合适的模板大小：模板的大小取决于图像的噪声程度以及图像的细节程度。常见的模板大小有 3×3、5×5 等。对于二值化图像，由于只包含黑白两种颜色，可以选择较小的模板来减少计算量。

②遍历图像的每个像素点：将选定的模板应用于图像的每个像素点，模板中心与当前像素点重合。

③读取模板内像素值：读取模板内所有像素的灰度值（对于二值化图像，这些值将是 0 或 255）。

④对像素值进行排序：将读取到的像素值按照大小进行排序。

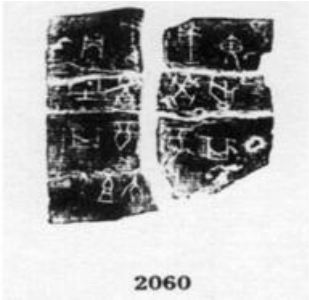


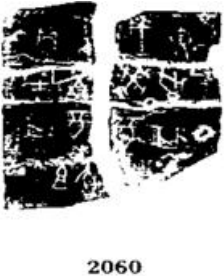


⑤计算中值：如果模板内像素数量为奇数，则中值为排序后中间位置的像素值；如果为偶数，则取中间两个值的平均值作为中值。

⑥更新像素值：将计算得到的中值赋给当前像素点，作为新的像素值。

利用中值滤波方法可以同时保留图像的边缘和细节信息。这种处理方法对于改善二值化图像的视觉效果和后续图像分析的准确性非常有帮助。

经上述操作，可以得到如下结果图，见表 4：

表 4 干扰元素预处理结果

	图像 1	图像 2	图像 3
原图			
二值化方法			

## 中值滤波方法



从表中可以观察到，通过二值化处理，每张图像都被转换成了黑白两色的图像，以增强文字与背景的对比。这种处理明显简化了图像的颜色空间，便于后续的图像识别与分割。在二值化后的图像中，可以看到文字部分以白色突出，而背景则是黑色。图像 1 的二值化效果清楚地将不同文本块的边缘勾勒出来，图像 2 和图像 3 的文本也更为醒目。中值滤波进一步处理了二值化后的图像，尤其是用于去除点状噪声。中值滤波后的图像 1 展现了更为平滑的文本块边缘和减少的噪点；图像 2 和图像 3 中的文本清晰度有所提升，背景噪声得到显著抑制。不过，这种处理也会模糊掉一些细小的文字特征，因此在后续模型构建前的预处理环节需要在去噪和保留细节之间找到平衡。

(3) Canny 边缘检测<sup>[10]</sup>：是一种多阶段的边缘检测算法，能够有效地检测图像中的边缘并抑制噪声，因此被广泛应用于计算机视觉和图像处理领域，以下是具体流程：

①转换灰度：本文在文初已将图像转换为灰度图像；

②滤波降噪处理：对原始图像做高斯模糊处理，减少图像中的噪声，平滑图像。

高斯分布函数为：

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

公式 (1) 中， $\sigma$  为滤波器参数，设原始图像为  $I(x, y)$ ，与高斯分布函数  $S(x, y)$  进行卷积运算，得到新的图像  $H(x, y) = I(x, y) * S(x, y)$ 。

③差分计算幅值和方向：在降噪后的图像上，利用 Sobel 算子做一阶导数，计算图像的梯度幅值和方向。其中梯度确定边缘方向，梯度幅值判断边缘的强度。

$$w_x(i, j) = [I(i, j+1) - I(i, j) + I(i+1, j+1) - I(i+1, j)] / 2 \quad (2)$$

$$w_y(i, j) = [I(i, j) - I(i+1, j) + I(i, j+1) - I(i+1, j+1)] / 2 \quad (3)$$

$$w(i, j) = \sqrt{w_x^2(i, j) + w_y^2(i, j)} \quad (4)$$

$$\theta(i, j) = \arctan \frac{w_x(i, j)}{w_y(i, j)} \quad (5)$$


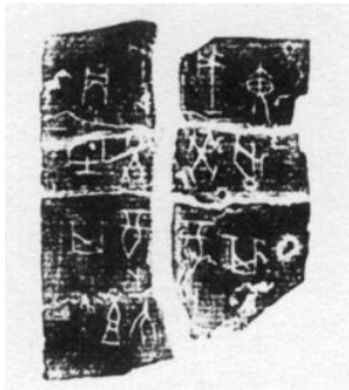




其中,  $w_x(i, j)$  和  $w_y(i, j)$  分别为像素  $(i, j)$  在  $x$ 、 $y$  方向的偏导数,  $w(i, j)$  是  $(i, j)$  处的梯度幅值,  $\theta(i, j)$  是  $(i, j)$  处的梯度方向。

④非极大值抑制: 非极大值数据排除其是边缘的可能性, 图像梯度幅值矩阵中的元素值越大, 说明图像中该点的梯度值越大, 结合检测点的梯度方向与边缘方向的垂直关系, 可定位边缘信息。

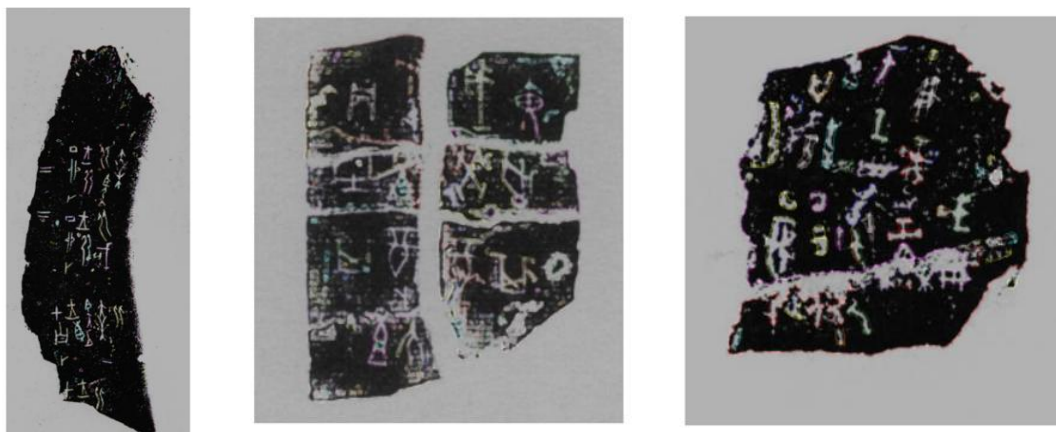
⑤滞后阈值: 使用双阈值来对二值化图像进行筛选, 选取合适的大阈值与小阈值得出与真实图像最接近的边缘。具体来说, 根据大小阈值对图像进行二值化处理, 将像素分为强边缘、弱边缘和非边缘三类, 若像素点大于大阈值, 则被认为是边界 (称为强边界) 用于确定强边缘; 小于小阈值则认为必然不是边界 (称为弱边界), 用于确定弱边缘; 若与确定为边缘的像素点邻接, 则判定为边缘; 否则为非边缘。

⑥边缘跟踪: 通过边缘跟踪算法, 进一步处理弱边缘, 弱边缘像素与强边缘像素相连的认为是边界, 形成边缘线条, 则该弱边缘点可以被保留为真实边缘, 其他的弱边界则被抑制。常用边缘跟踪算法有基于连接组件的方法或者霍夫变换。

表 5 Canny 算子边缘检测结果

	图片 1	图片 2	图片 3
原图			
Canny 算法 边缘 检测			

标记  
甲骨  
文字  
区域



表中呈现的是三张甲骨文拓片图像通过 Canny 边缘检测算法的处理结果，以及根据 Canny 算法的结果标记出的甲骨文字区域。

观察经过 Canny 边缘检测算法处理后的结果，发现 Canny 算法作为一种多阶段的边缘检测工具，它在图像中成功地突出了甲骨文的边缘。在图像 1 中，甲骨文的轮廓和内部的文字边缘被清晰地标识出来，尽管原始图像较为暗淡。图像 2 展示了几块甲骨文，每一块的文字和边缘也都清晰可辨，显示出了 Canny 算法的高准确性和低错误率。图像 3 则在保留了文字细节的同时，显示了图像背景和文字之间的边缘，即使在细节丰富的区域，Canny 算法也能够保持良好的边缘检测性能。

图中下方三张图像标出了根据 Canny 算法结果确定的甲骨文字区域。这些区域被彩色标记，以区分文字和背景。例如，在处理后的图像 1 中，甲骨文的文字区域沿着边缘被一条条精细的彩色线条所环绕，突显了文字所在的位置，而非文字区域则保留了原图的颜色。在图像 2 和图像 3 中，彩色标记清晰地区分了文本和背景，有助于后续的文字识别和分割任务。

Canny 算法的优点在这些处理后的图像中得到了充分体现。高准确性保证了边缘的精确定位，低错误率使得算法能够在复杂的纹理中准确识别文字边缘，而多阶段处理流程有效地抑制了噪声，同时保留了边缘的重要细节。这些特点使得 Canny 算法特别适用于处理甲骨文拓片这类含有大量细节和纹理的图像。通过精确的边缘检测，进一步的图像分析和文字识别工作能够在一个清晰定义的基础上进行，大大提升了后续处理步骤的可靠性和准确性。



4.2 问题二的求解

问题二旨在解决甲骨文图像的自动分割问题，关键挑战在于图像质量的多样性、噪声干扰的复杂性以及字符形态的变异性。针对这些难点，本研究通过数据增强引入旋转变换，提升了模型对于不同角度和位置的甲骨文图像的适应性，并采用深度学习框架中的卷积神经网络（CNN）与 U-Net 模型。CNN 模型擅长于自动化特征提取，而 U-Net 模型特别适合处理样本数量有限且不均匀的甲骨文图像数据，可以在进行特征抽象的同时保持空间细节。

4.2.1 图像增强

为使得后续模型训练效果好、泛化能力强，在做特征提取前对已有数据做数据增强处理，使得样本数量充足，增加噪声数据，提升模型的鲁棒性。下对甲骨文拓片图像做数据旋转，旋转角度于（90° ,180° ）之间，避免出现尺度变化问题。如图 9 所示：

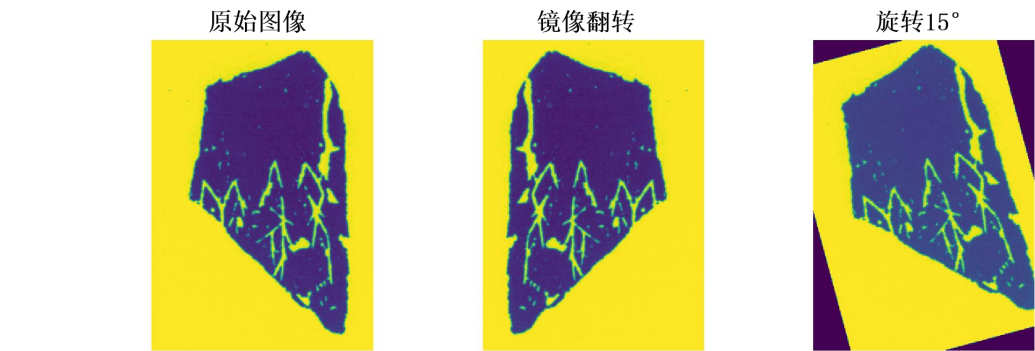


图 9 数据增强效果图

如图 9 所示，通过数据增强手段，对甲骨文拓片图像进行了翻转和旋转处理，创建了一组变换后的图像样本。通过增加样本多样性，不仅能够提升模型对于不同方向旋转的甲骨文图像的识别能力，还能增强模型的鲁棒性，使其更加健壮地对抗实际应用中可能遇到的方向和定位误差。通过增加旋转噪声，模型的泛化能力得到了增强，这为后续的特征提取和模型训练奠定了坚实的基础。

4.2.2 基于 CNN 模型的图像特征提取

采用问题一的预处理模型提高了图像的清晰度，且对干扰元素进行特征提取，并对不同的干扰元素进行去除操作，这些操作为其后的甲骨文拓片进行图像分割做了基础的准备工作。接下来采用卷积神经网络（Convolutional Neural Network，CNN）<sup>[11]</sup>进行图像文字的特征提取，下面介绍 CNN 算法。

卷积神经网络的整体架构：输入层，卷积层，池化层，ReLU 层，全连接层。

（1）卷积层<sup>[12]</sup>：确定权重参数矩阵  $W_{filter}$ ，将图像输入数据划分为  $7 \times 7$  的小区域，并对每个区域进行特征提取，定滑动窗口步长为 1。为了改善原始图像的权重比例，可以适度进行边缘填充，由于边缘的点被提及的次数少，所以给边界 padding 一圈之后，

让边缘点提取次数增多，一定程度上弥补了边界特征缺失的问题。

卷积后的长度计算公式为：

$$H_2 = \frac{H_1 - F_H + 2P}{S} + 1, \quad (6)$$

宽度计算公式为：

$$W_2 = \frac{W_1 - F_W + 2P}{S} + 1, \quad (7)$$

其中  $H_1$  与  $W_1$  为原图的长和宽， $F_H$  与  $F_W$  为  $W_{filter}$  的长与宽， $P$  为边界填充， $S$  为指定步长，且经过卷积操作后长与宽不一定改变。

(2) 池化层：池化层的作用是对处理后的图像就进行降维与去噪，一般选择 Max Pooling 最大池化，不选择平均池化，经过处理后的图像会变小。

(3) ReLU 层<sup>[13]</sup>：ReLU 函数是有效的激活函数的一种，应用比较广泛，其本质是一个取最大值函数，它的表达形式如下：

$$f(x) = \max(0, x) \quad (8)$$

ReLU 函数实质上为取最大值的函数，其图像见图 10：

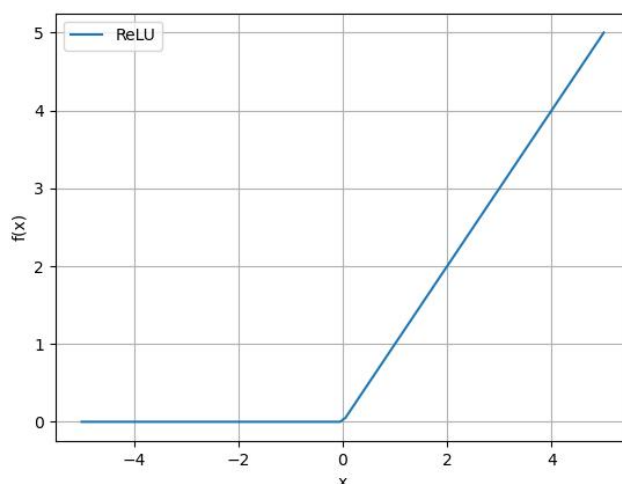


图 10 ReLU 函数图像

当应用 sigmoid 等函数算激活函数，计算量较大，反向传播求误差梯度时利用 ReLU 激活函数会简化计算量。ReLU 函数没有饱和区，不存在梯度消失的问题，没有复杂的指数运算，计算简单，收敛速度快，具有多重优势。

(4) 全连接层：全连接层的作用是“分类器”，可以将学到的“分布式特征表示”映射到样本标记空间，将前面的层转化为卷积核为  $1 \times 1$  的卷积或者核为前层卷积结果的高和宽。

利用 CNN 将甲骨文拓片上的文字进行特征提取，通过调节参数，训练出一套自动识别甲骨文字的网络，从而实现自动单字分割。特征提取示意图如图 11：

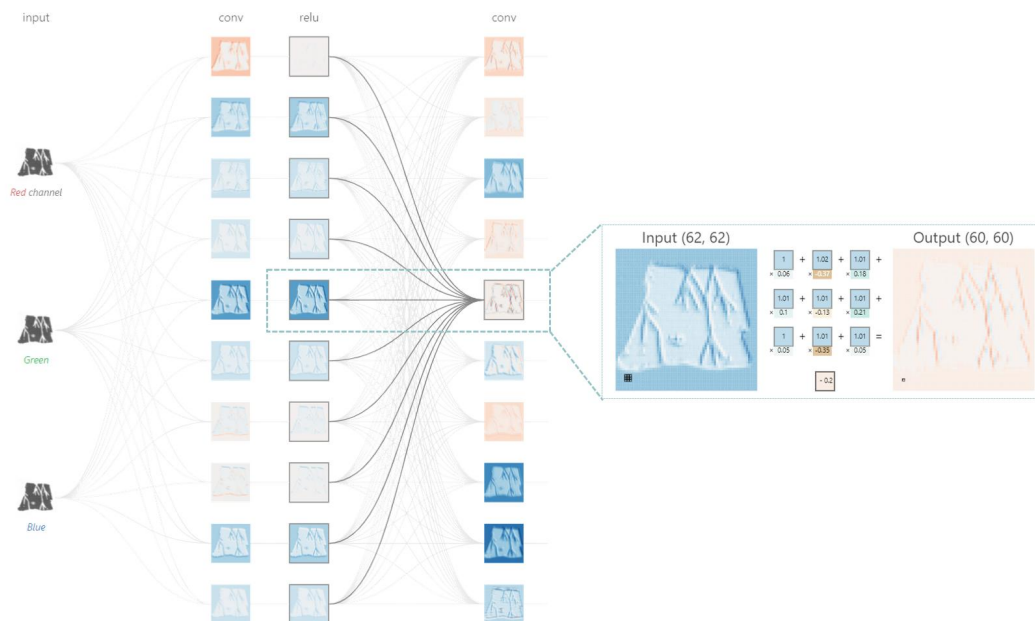


图 11 CNN 特征提取图

从图中可以看到卷积神经网络进行特征提取的过程示意图。输入是一个由红色、绿色、蓝色（RGB）三个颜色通道组成的彩色图像。每个颜色通道独立输入到网络中，并分别在随后的层中进行处理。每个颜色通道的图像首先通过一系列的卷积滤波器。这些滤波器在图像上滑动，提取局部特征，如边缘、角点等。每个卷积滤波器产生的特征图（feature map）都在下一步中使用。接着，卷积层的输出被传递到激活函数，通常是 ReLU（Rectified Linear Unit）激活函数。ReLU 函数将所有负值置为 0，非负值保持不变。它的目的是增加网络的非线性，有助于网络学习复杂的特征。激活后的特征图再次经过一系列卷积滤波器，进行更深层次的特征提取，会捕捉到更高级的特征，比如特定的纹理或形状的组合。

在图像的放大部分，我们看到了一个特定的卷积滤波器在输入图像上的操作示例。左侧是输入的特征图片段（ $62 \times 62$ ），右侧是滤波器应用后的输出（ $60 \times 60$ ）。展示了滤波器的权重矩阵和这些权重如何应用于输入特征图的小部分来生成输出特征图的一个区域。权重矩阵与特征图的相应区域相乘并求和，结果是一个新的特征值，反映了原始图像中的局部特征。

#### 4.2.2 基于 U-Net 模型的图像分割

U-Net 模型<sup>[14]</sup>于 2015 年发表，属于 FCN 的一种变体，其同时拥有编码器和解码器两个结构，U-Net 最突出的部分即为跳跃连接部分，很好的解决了由于下采样操作所丢失掉的细节损失，例如边界信息，从而能够很好的精准定位。FCN 是以对应像素相加的方式实现深层信息于浅层信息的融合，相对应的 U-Net 是以拼接的方式，拼接能够保留更多的维度/位置信息，对一些特定任务来说更有优势。具体流程如下：

Encoder 编码器<sup>[15]</sup>，进行下采样和特征提取，将图像逐渐压缩为高层次特征表示，

它是由一系列卷积层和池化层组成的，其中卷积层负责提取图像中的特征，池化层用于压缩特征图的尺寸大小。这一过程可被视为从整体特征到局部特征的转换。

**Decoder 解码器**，负责将编码器提取的高层次特征恢复到原始输入图像的尺寸。其由一系列卷积层和反卷积层组成，反卷积层用于将特征图的空间尺寸扩大，卷积层用于进一步细化特征。

跳跃连接，每个编码器阶段的特征图都会与对应的解码器阶段的特征图连接，形成跳跃连接。底层特征与高层特征的融合，网络能很好的保留足够多的高层特征图所包含的高分辨率细节信息，并最大化利用这些信息，进而有效提升图像分割精度。U-Net 模型结构如图 12 所示：

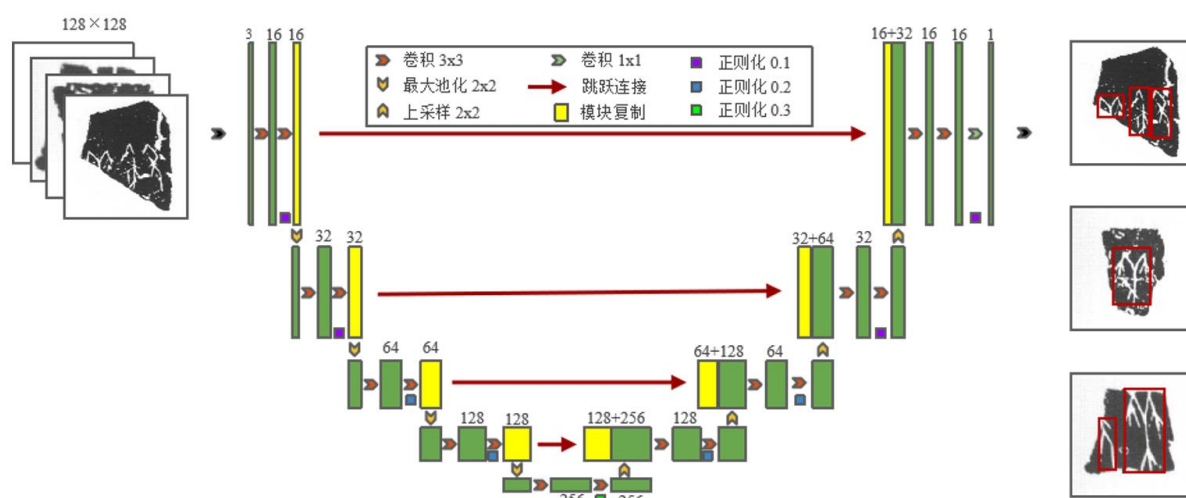


图 12 U-Net 模型结构图

该图展示了 U-Net 模型的典型结构，由左侧的收缩路径和右侧扩展路径两部分组成。收缩路径包括多个卷积层和池化层。在这个路径中，网络不断地通过卷积层对输入图像进行特征提取，然后通过池化层减少特征图的空间维度。在本研究中，从最初的 3 个通道开始，卷积层的通道数量增加到 16、32、64、最后是 128 个通道。

扩展路径通过上采样和卷积过程逐步恢复图像的空间维度，同时减少特征通道的数量。在上采样过程中，特征图的大小加倍，然后使用卷积操作来减少通道数量。与此同时，扩展路径还通过跳跃连接从收缩路径中获得相应的特征图。这些跳跃连接直接将特征图的副本与上采样的结果相连接，从而允许网络在更高层次上利用更精确的定位信息。在本研究中，扩展路径的每一步都将特征通道数量减半，从 128 到 64，最后回到初始的 16 个通道。

图 13 展示了 U-Net 图像分割模型的输出结果，左侧是输入到 U-Net 模型中的原始二值化图像，右侧是模型处理后的输出结果。

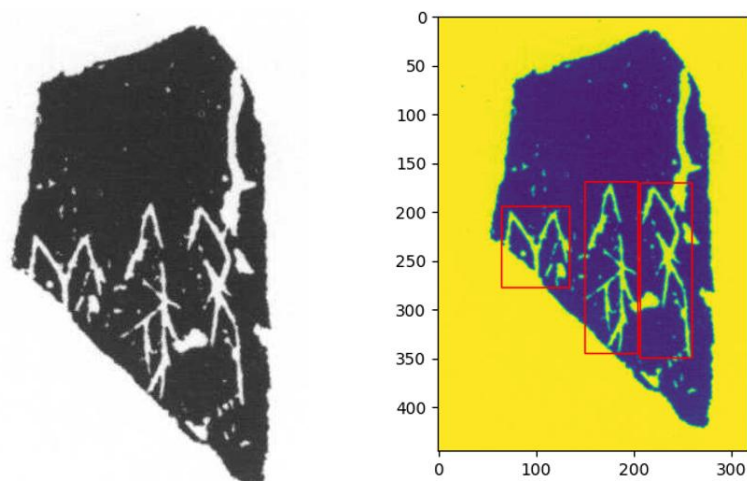


图 13 图像分割结果图

在 U-Net 的输出图像中，可以看到了对甲骨文图像进行的分割处理。蓝色部分表示模型分割出的甲骨文区域，黄色部分代表模型识别为背景的区域突出了文本与背景的界限。红色矩形框标注的是模型识别的甲骨文字符区域，展示了 U-Net 模型在细节层面上的分割效果，指示出模型能够将字符从相邻的背景中准确分离，对于后续的字符识别和分析工作打下坚实基础。

在输出图像后还需要进行后处理步骤，如清理分割边界、连接断开的字符区域等，以进一步提高分割的准确性。

#### 4.2.3 模型评估

本文将采用 5 折交叉验证用来进行模型评估，首先将数据集划分为训练集、测试集和验证集，将训练集分为 5 个子集，使得每个子集的个数相等，对其中 4 个子集作为训练集，1 个子集为验证集，可以取损失函数作为目标函数，进行 5 次交叉验证，计算得到平均误差。交叉验证示意图如图 14：

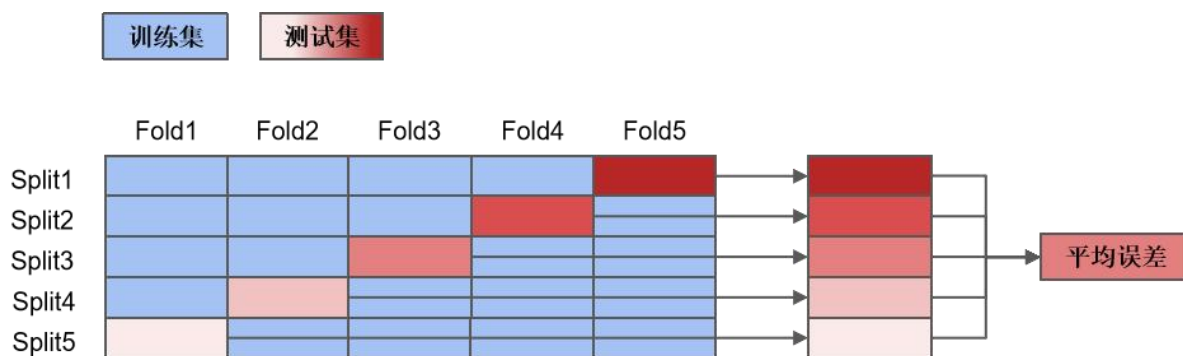


图 14 交叉验证示意图

本文运用 Precision、F1、Dice coefficient(Dice 系数)、Matthews coefficient、MIoU、MPA 等来评价指标进行模型评估。下对其分别做介绍：

精准率（Precision），又叫查准率，反映预测结果与实际结果一致的概率：

$$Precision = \frac{TP}{TP + FP} \quad (9)$$



F1 分数（F1 Score）是精确率与召回率的一个加权平均，取得两者之间的平衡。召回率体现对正样本的识别能力，F1 Score 越高，模型越稳健：

$$Recall = \frac{TP}{TP + FN} \quad (10)$$

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (11)$$

Dice 系数（Dice coefficient），综合准确率与召回率，该模评估指标更为全面：

$$Dice = \frac{2TP}{2TP + FP + FN} \quad (12)$$

马修斯相关系数（Matthews coefficient, MCC）<sup>[16]</sup>，常用于评估二元分类模型，由于其考虑了正负样本间的不平衡性，更适用于非均衡数据集，并且其考虑了真阳性、真阴性、假阳性和假阴性之间的关系，能准确衡量分类器的性能。MCC 取值范围为[-1,+1]，当 MCC 为-1 时表示预测结果完全错误，MCC 为+1 时表示预测结果完全正确，MCC 为 0 时，表示随机预测。计算公式如下：

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (13)$$

均交并比（Mean Intersection over Union, MIoU）为常见的语义分割评价指标，其计算真实值与预测值之间的交集和并集的比值，求和再平均，计算公式如下：

$$MIoU = \frac{1}{k+1} \sum_{i=0}^k \frac{p_{ii}}{\sum_{j=0}^k p_{ij} + \sum_{j=0}^k p_{ji} - p_{ii}} \quad (14)$$

类别平均像素准确率（MPA），分别计算正确分类的类别像素数的比例，即 PCA，再累加求平均：

$$MPA = \frac{1}{k+1} \sum_{i=0}^k \frac{p_{ii}}{\sum_{j=0}^k p_{ij}} \quad (15)$$

评价指标运行结果及解读如表 6：

表 6 评价指标

评价指标	指标值	评估结果
Precision	87.5%	表明模型识别甲骨文字符准确性较好。
F1 Score	0.83	表明模型在精确度与召回率之间保持了良好的平衡。
Dice coefficient	0.68	表明模型具有一定的可靠性，但 Dice 系数仍有很大提升空间。
MCC	0.55	表明模型的预测与实际值之间有中等程度的相关性。
MIoU	0.74	表明模型在区分甲骨文文字和背景方面表现良好。
MPA	0.84	表明模型在像素级别上具有较高的准确度。

4.3 问题三的求解

继问题二成功构建并验证了甲骨文图像分割的模型后，问题三旨在将这一模型应用于实际的测试数据样本，以评估其在未见样本上的性能表现。通过这一过程，模型的实际应用能力能够得以验证，同时也能够对模型泛化能力进行直观检验。

4.3.1 基于 U-Net 模型的自动单字分割

在问题二的基础上，本研究进一步探讨了使用 U-Net 模型对甲骨文图像进行自动单字分割的可行性和效果。在这一过程中，首先加载了问题二中训练好的甲骨文图像分割模型，以及 200 张甲骨文原始拓片图像组成的测试数据集，数据集来源于附件 3。本研究旨在对这些数据进行单字分割，具体流程如下：

（1）图像预处理：为了保证输入模型的一致性，所有测试图像首先被调整到模型训练时的相同尺寸；通过锐化滤波和直方图均衡化技术改善图像质量，增强文字与背景的对比度。

（2）干扰元素处理：应用 LBP 算法识别出图像中的干扰元素特征；进一步采用二值化方法将图像转换成易于分析的二值图，然后通过中值滤波去除噪点和小的干扰物；最后使用 Canny 边缘检测算子，强化图像中单字的边缘，为分割提供准确的边界信息。

（3）特征提取与图像分割<sup>[17]</sup>：利用卷积神经网络从处理后的图像中提取关键特征，为精准的图像分割提供必要的信息；借助 U-Net 模型的强大分割能力，通过其特征融合策略，从复杂背景中准确地分割出单字图像，有效地合并了浅层特征和深层特征，从而实现了对甲骨文中每个单字精细位置的精确识别。

4.3.2 自动单字分割的实现

对测试集甲骨文原始拓片图像进行自动单字分割，部分结果图如图 15 所示：

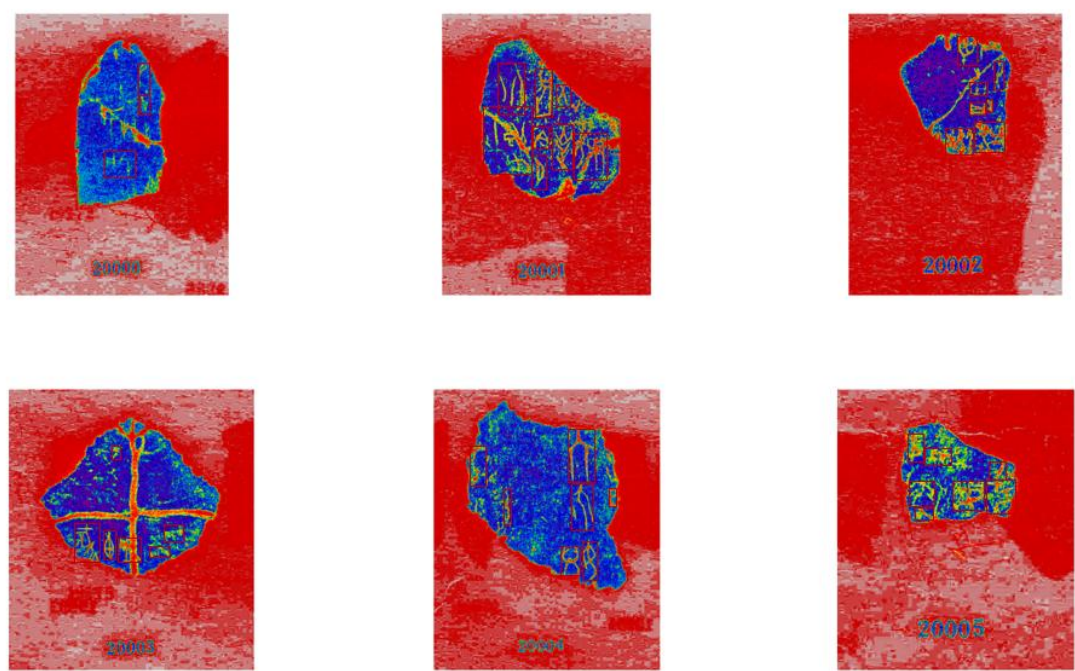


图 15 测试集自动单字分割结果图

图中，背景为红色，表示模型分割过程中被识别为非文本区域；甲骨文字符被标记或着色为蓝色，显示了分割算法确定的文本区域；拓片表面的其他部分，可能是由于残损或材料的天然纹理而未被算法识别为文本。红色框定位了甲骨文文字位置信息，同时也存在一些分割错误，可能是由于模型将噪声或拓片上的非文本特征误识别为文字。

详细分割结果见 Test\_results.xlsx，表 7 展示了部分结果：

表 7 自动单字分割结果

图像名称	标记
020000.jpg	[[505, 719, 506, 720, 1], [243, 719, 245, 720, 1], [236, 719, 238, 720, 1]]
020001.jpg	[[490, 694, 491, 695, 1], [393, 693, 396, 695, 1], [366, 693, 370, 695, 1], [191, 693, 193, 694, 1], [92, 693, 94, 695, 1], [454, 692, 455, 693, 1], [315, 692, 319, 693, 1],[136, 692, 139, 695, 1]]
020002.jpg	[[332, 594, 334, 595, 1], [326, 594, 328, 595, 1], [89, 594, 90, 595, 1], [357, 593, 358, 594, 1], [298, 593, 301, 595, 1], [288, 593, 290, 595, 1], [249, 593, 250, 594, 1]]
020003.jpg	[[26, 719, 27, 720, 1], [528, 718, 530, 720, 1], [477, 718, 480, 720, 1], [458, 718, 461, 720, 1], [6, 718, 7, 719, 1], [502, 717, 505, 719, 1]]
020004.jpg	[[485, 758, 487, 760, 1], [334, 758, 335, 760, 1], [289, 758, 291, 760, 1], [118, 758, 119, 759, 1], [312, 757, 314, 760, 1], [576, 756, 583, 757, 1], [573, 756, 574, 757, 1]]
020005.jpg	[[103, 529, 104, 530, 1], [151, 528, 153, 529, 1], [59, 528, 61, 529, 1], [30, 528, 31, 529, 1], [280, 527, 284, 528, 1], [278, 527, 279, 528, 1], [270, 527, 271, 528, 1]]
.....	.....





Faster R-CNN 的特征提取由卷积神经网络完成，通过对图像进行深入分析，提取出用于识别的关键特征。核心的技术创新包括区域建议网络（RPN）<sup>[19]</sup>和 ROI Pooling 层。RPN 负责从特征图中提出潜在的目标区域，而 ROI Pooling 层则能够从这些区域提取出尺寸统一的特征向量，为后续的分类与回归任务提供了固定格式的输入<sup>[20]</sup>。

将得到的特征向量与两个平行的全连接层连接，一个进行分类以确定候选区域的类别，另一个进行边界框回归以精确定位每个目标。通过非极大值抑制（NMS）的后处理步骤，模型去除重叠或冗余的检测框，从而得到最终的检测结果。

#### ①特征提取

输入一张待检测的图像，见图 17。算法应用预训练的卷积神经网络对图像进行特征提取，得到一个特征图，包含了图像中各个区域的丰富信息。



图 17 待检测图像

#### ②生成候选区域

Faster R-CNN 利用 RPN 在特征图上生成候选区域，RPN 通过滑动窗口在特征图上滑动，并对每个窗口位置生成一些不同尺度和长宽比的候选框。与此同时，RPN 会对候选框生成一个得分，意思为该候选框包含目标的一个概率。

#### ③ROI Pooling

对于各个候选区域，Faster R-CNN 会利用 ROI Pooling 层将它映射到特征图上，并提取固定大小（7×7）的特征向量。从而生成的特征向量大小固定，无关候选区域的大小，有利于后续的分类和回归。

#### ④分类与回归

特征向量得到后，Faster R-CNN 将其送入两个并行存在的全连接层，其中一个用于分类，可以得到每个候选区域属于每个类别的概率，另一个用于全连接层回归，输出候选区域时的边界框坐标。

#### ⑤后处理

后处理是指算法利用非极大值抑制（NMS）后进行处理，筛掉冗余的检测结果，并输出最终的检测结果。

### 4.4.2 甲骨文文字自动识别的实现

由训练集甲骨文拓片图像得到的甲骨文字“北”的识别结果，见图 18：



图 18 甲骨文字“北”识别结果

根据初步结果可以观察到，结果图较为模糊，但可以清楚看到“北”字的轮廓，因此文字识别模型具有一定的可行性与准确性。

为了更好的呈现得到的甲骨文文字识别结果，我们对得到的甲骨文字进行处理，使它变得更加清晰可观，先利用 Gabor 滤波处理图像，它的作用是在不同尺度以及方向上提取纹理特征，同时减弱由于光照变化和噪声对图像带来的影响，帮助突出特定方向的纹理特征。之后再利用锐化处理强化边缘，增加对比度，经过两个步骤的图像处理，最终呈现的文字结果如图 19。



图 19 甲骨文字“北”图像处理结果

通过 Faster R-CNN 算法训练的模型，不仅提高了文字识别的效率，而且增强了结果的准确性。最终的文字识别示意图能够展现该算法在甲骨文文字识别任务中的应用价值，如图 20。

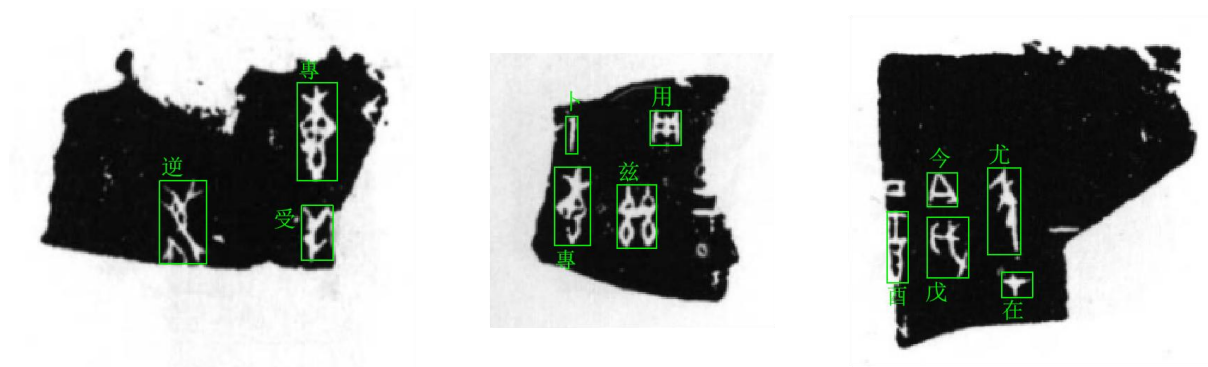


图 20 Faster R-CNN 文字识别示意图

图中展示了甲骨文字符在复杂背景下精确地框定和标识的结果。通过绿色矩形框定位每个文字位置，并在该甲骨文字旁标注相应的汉字，直观地传达了所该算法在定位和识别甲骨文方面的预期效果。

## 第五章 模型评价与推广

### 5.1 模型的评价

本研究构建的模型融合了经典图像处理技术与先进的深度学习算法，针对甲骨文拓片图像的独特性，设计出一套完备的图像处理流程。

（1）在预处理阶段，本模型利用局部二值模式（LBP）算法和 Canny 算子对干扰元素进行细致的辨识与处理。此方法在处理光照变化及对比度差异引发的干扰时表现出较高的鲁棒性，有效捕获了图像的纹理特征，并且能够精确地描绘出图像的边缘和结构细节。

（2）在图像分割环节，模型采用 U-Net 架构以其独特的解码器设计和跳跃连接策略，不仅保留了丰富的细节信息，还维持了对少数类别的高敏感度。这种机制尤其适用于处理类别不平衡的数据集。与现有甲骨文识别领域中常用的深度学习方法如支持向量机（SVM）和随机森林等相比，这些方法通常依赖大量标注数据而在拓片上的分类准确度并不理想，往往难以解决样本量少、类别不平衡的问题。因此，本研究选用 U-Net 模型以提升针对不平衡数据的处理性能。

（3）在文字识别部分，模型采纳了 Faster R-CNN 模型。此模型结合两阶段网络与区域建议网络（RPN），不仅加快了目标对象在图像中的定位与识别过程，相比于单阶段网络，它在处理多尺度和小尺寸目标方面具有更精准的识别能力。这使得 Faster R-CNN 在甲骨文拓片文字的识别任务上展现出更高的准确率和鲁棒性，为甲骨文研究领域提供了一种更为有效的技术手段。

### 5.2 模型的推广

本研究整合了中值滤波、二值化、Gabor 滤波、锐化处理等传统图像处理技术，以及 CNN、Faster R-CNN 等深度学习方法，展现了在甲骨文拓片文字识别领域的显著成效。该模型在解决历史文献数字化处理和解读的具体任务上显示出广阔的发展潜力，未来的应用前景如下：

（1）算法优化与模型迭代：致力于模型和算法的持续优化工作，通过引入先进的算法框架和优化策略，提高模型的识别精度和处理效率。加强研发投入，深入挖掘深度学习在甲骨文识别中的应用潜能，从而推动自动化文献分析技术的发展。

（2）泛化能力提升：加强模型在不同情境下的适应能力，如能够准确识别多种风格、不同年代、以及不同清晰度的甲骨文拓片，从而实现对甲骨文广泛文化遗产的保护和研究。

（3）数据集构建和优化：建立并不断扩充大规模的甲骨文拓片图像数据集，确保数据的多样性和全面性，同时进行精确的标注和预处理。这将确保模型训练的数据基础健全，为实现更为准确和鲁棒的文字识别模型打下坚实基础。

本模型的潜在应用并不限于甲骨文的解读，其研究成果还可推广到其他领域，比如手写体识别、古文献修复以及艺术作品中的文本提取等。这种跨学科的技术融合，将有望在历史文献研究、文化遗产保存和信息技术等多个领域产生深远影响。

## 参考文献

- [1] 宋传鸣,乔明泽,洪颀.边缘梯度协方差引导的甲骨文字修复算法[J].辽宁师范大学学报(自然科学版),46(02):194-207,2023.
- [2] 高旭.基于卷积神经网络的甲骨文识别研究与应用[D].吉林大学,2021.
- [3] 朱旭.基于改进 PUGAN 和 CNN 模型的甲骨文分类算法研究[D].大连海事大学,2022.
- [4] 曹帆之,石添鑫,韩开杨,等.多模态遥感图像模板匹配 Log-Gabor 滤波方法[J].测绘学报,53(03):526-536,2024.
- [5] Sun Z, Tang D, Cui H, et al. Gaussian blur aided blind phase search algorithm in low SNR QAM transmission system[J]. Optics Communications, 533: 129282, 2023.
- [6] Kaya Y, Kuncan M, Akcan E, et al. An efficient approach based on a novel 1D-LBP for the detection of bearing failures with a hybrid deep learning method[J]. Applied Soft Computing, 155: 111438, 2024.
- [7] 杨秀娥,金卢佳,徐路平,等.基于灰度图二值化的汽车制动器摩擦片残缺检测方法[J].内燃机与配件,(06):19-21,2024.
- [8] 国方媛,李国东.基于细胞神经网络和网格特征的碑刻文字识别[J].计算机系统应用,19(11):180-184,2010.
- [9] 陈利利,李斌,梁晓晴.基于 Canny 边缘检测的红枣识别算法研究[J].信息技术与信息化,(03):25-28,2024.
- [10] Rahmawati S, Devita R, Zain R H, et al. Prewitt and Canny Methods on Inversion Image Edge Detection: An Evaluation[J]. Journal of Physics: Conference Series, 1933(1): 012039, 2021.
- [11] Khandouzi A, Ezoji M. Coarse-to-fine underwater image enhancement with lightweight CNN and attention-based refinement[J]. Journal of Visual Communication and Image Representation, 99: 104068, 2024.
- [12] 柯永红.基于卷积神经网络的碑刻拓片楷体文字自动识别[J].民俗典籍文字研究,(02):245-252+264,2018.
- [13] Ehtisham R, Qayyum W, Camp C V, et al. Computing the characteristics of defects in wooden structures using image processing and CNN[J]. Automation in Construction, 158: 105211, 2024.
- [14] Liu A, Zhang Y, Xia Y, et al. Classes U-Net: A method for nuclei segmentation of photoacoustic histology imaging based on information entropy image classification[J]. Biomedical Signal Processing and Control, 91: 105932, 2024.
- [15] Khan M J, Singh P P. Advanced road extraction using CNN-based U-Net model and satellite imagery[J]. e-Prime - Advances in Electrical Engineering, Electronics and Energy, 5: 100244, 2023.

- [16]Zhang C, Zong R, Cao S, et al. AI-Powered Oracle Bone Inscriptions Recognition and Fragments Rejoining[A]. Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence[C]. Yokohama, Japan: International Joint Conferences on Artificial Intelligence Organization, 2020: 5309-5311.
- [17]Shi Xiaosong, Huang Yongjie, Liu Yongge. Text on Oracle rubbing segmentation method based on connected domain[A]. 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)[C]. Xi'an, China: IEEE, 2016: 414-418.
- [18]Ren S, He K, Girshick R, et al. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks[J]. arXiv, 2016.
- [19]刘芳,李华飙,马晋,等.基于 Mask R-CNN 的甲骨文拓片的自动检测与识别研究[J].数据分析与知识发现,5(12):88-97,2021.
- [20]Lyu H, Qiu F, An L, et al. Deer survey from drone thermal imagery using enhanced faster R-CNN based on ResNets and FPN[J]. Ecological Informatics, 79: 102383, 2024.
- [21]史先进,曹爽,张重生,等.基于锚点的字符级甲骨图像自动标注算法研究[J].电子学报,49(10):2020-2031,2021.

## 附录

### 1. 环境说明

(1) Python 版本: Python 3.6.13

(2) 主要依赖库及其版本:

numpy: 1.19.5

pandas: 1.1.5

scikit-learn: 0.24.2

matplotlib: 3.3.4

Pillow: 8.4.0

tensorflow: 2.6.2

torch: 1.10.0+cu111

opencv-python: 1.19.5

### 2. 核心代码

#### 问题一

```
# p1_dataPrepare
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import cv2

# 加载图像
image_path_h02060 = '/1_Pre_test/h02060.jpg'
image_h02060 = Image.open(image_path_h02060)

# 调整图像大小
size = (256, 256)
image_h02060_resized = image_h02060.resize(size)

# 灰度化处理
image_h02060_gray = image_h02060_resized.convert('L')

# 转换为 NumPy 数组
image_h02060_array = np.array(image_h02060_gray)
```



```

# 可视化处理后的图像
plt.figure(figsize=(6, 6))
plt.imshow(image_h02060_array, cmap='gray')
plt.title('Resized and Grayscale Image')
plt.axis('off')
plt.show()

# 设置二值化阈值
thresh = 128
# 二值化处理
binary_image = cv2.threshold(image_h02060_array, thresh, 255, cv2.THRESH_BINARY)[1]
# 中值滤波
median_filtered = cv2.medianBlur(binary_image, 3)
# 锐化滤波
sharpen_kernel = np.array([[ -1, -1, -1], [-1, 9, -1], [-1, -1, -1]])
sharpened_image = cv2.filter2D(median_filtered, -1, sharpen_kernel)
# Gabor 滤波
gabor_kernel = cv2.getGaborKernel((5, 5), 4.0, np.pi/4, 10.0, 0.5, 0, ktype=cv2.CV_32F)
gabor_filtered = cv2.filter2D(sharpened_image, cv2.CV_8UC3, gabor_kernel)
# 高斯滤波平滑图像
gaussian_blurred = cv2.GaussianBlur(sharpened_image, (3, 3), 0)

# 可视化处理效果
fig, axes = plt.subplots(2, 3, figsize=(12, 8))

# 原始二值化图像
axes[0, 0].imshow(binary_image, cmap='gray')
axes[0, 0].set_title('Binary Image')
axes[0, 0].axis('off')
# 中值滤波后的图像
axes[0, 1].imshow(median_filtered, cmap='gray')
axes[0, 1].set_title('Median Filtered Image')
axes[0, 1].axis('off')
# 锐化后的图像
axes[0, 2].imshow(sharpened_image, cmap='gray')

```

```

axes[0, 2].set_title('Sharpened Image')
axes[0, 2].axis('off')
# Gabor 滤波后的图像
axes[1, 0].imshow(gabor_filtered, cmap='gray')
axes[1, 0].set_title('Gabor Filtered Image')
axes[1, 0].axis('off')
# 高斯滤波后的图像
axes[1, 1].imshow(gaussian_blurred, cmap='gray')
axes[1, 1].set_title('Gaussian Blurred Image')
axes[1, 1].axis('off')
# 对比原始灰度图像和高斯滤波后的图像
axes[1, 2].imshow(image_h02060_array, cmap='gray')
axes[1, 2].set_title('Original Grayscale Image')
axes[1, 2].axis('off')

plt.tight_layout()
plt.show()

# 使用灰度图像
# 中值滤波去除噪点
median_filtered_gray = cv2.medianBlur(image_h02060_array, 3)
# 锐化滤波增强边缘
sharpened_image_gray = cv2.filter2D(median_filtered_gray, -1, sharpen_kernel)
# Gabor 滤波突出特定方向的纹理
gabor_filtered_gray = cv2.filter2D(sharpened_image_gray, cv2.CV_8UC3, gabor_kernel)
# 高斯滤波平滑图像
gaussian_blurred_gray = cv2.GaussianBlur(sharpened_image_gray, (3, 3), 0)
# 直方图均衡化增强对比度
equalized_image = cv2.equalizeHist(image_h02060_array)

# 可视化处理效果
fig, axes = plt.subplots(3, 2, figsize=(12, 12))
# 中值滤波后的灰度图像
axes[0, 0].imshow(median_filtered_gray, cmap='gray')
axes[0, 0].set_title('Median Filtered Gray Image')
axes[0, 0].axis('off')

```

```

# 锐化后的灰度图像
axes[0, 1].imshow(sharpened_image_gray, cmap='gray')
axes[0, 1].set_title('Sharpened Gray Image')
axes[0, 1].axis('off')
# Gabor 滤波后的灰度图像
axes[1, 0].imshow(gabor_filtered_gray, cmap='gray')
axes[1, 0].set_title('Gabor Filtered Gray Image')
axes[1, 0].axis('off')
# 高斯滤波后的灰度图像
axes[1, 1].imshow(gaussian_blurred_gray, cmap='gray')
axes[1, 1].set_title('Gaussian Blurred Gray Image')
axes[1, 1].axis('off')
# 直方图均衡化后的灰度图像
axes[2, 0].imshow(equalized_image, cmap='gray')
axes[2, 0].set_title('Histogram Equalized Image')
axes[2, 0].axis('off')
# 原始灰度图像
axes[2, 1].imshow(image_h02060_array, cmap='gray')
axes[2, 1].set_title('Original Gray Image')
axes[2, 1].axis('off')

plt.tight_layout()
plt.show()

# Canny 边缘检测
edges = cv2.Canny(image_h02060_array, 100, 200)
# 找到边缘检测后的连通组件
contours, _ = cv2.findContours(edges, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
# 绘制边界框
image_with_contours = cv2.cvtColor(image_h02060_array, cv2.COLOR_GRAY2BGR)
cv2.drawContours(image_with_contours, contours, -1, (0, 255, 0), 1)
# 可视化边缘检测结果
plt.figure(figsize=(8, 8))
plt.imshow(edges, cmap='gray')
plt.title('Edge Detection')

```

```
plt.axis('off')
plt.show()
# 可视化绘制了边界框的图像
plt.figure(figsize=(8, 8))
plt.imshow(image_with_contours)
plt.title('Image with Contours')
plt.axis('off')
plt.show()

# 形态学操作
# 定义一个 3x3 的结构元素
kernel = np.ones((3,3), np.uint8)
# 腐蚀操作
erosion = cv2.erode(edges, kernel, iterations=1)
# 膨胀操作，对腐蚀后的图像进行膨胀，以突出文字部分
dilation = cv2.dilate(erosion, kernel, iterations=1)

# 可视化形态学操作的结果
plt.figure(figsize=(12, 6))

plt.subplot(1, 3, 1)
plt.imshow(edges, cmap='gray')
plt.title('Edges')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(erosion, cmap='gray')
plt.title('Erosion')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(dilation, cmap='gray')
plt.title('Dilation')
plt.axis('off')
```

```

plt.tight_layout()
plt.show()

# p1_featureLBP
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 计算 LBP 图像的函数
def lbp_image(image):
    # 将图像转换为灰度图
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # 获取图像的尺寸
    height, width = gray_image.shape

    # 创建一个空的 LBP 图像，大小与输入图像相同
    lbp = np.zeros((height, width), np.uint8)

    for i in range(1, height - 1):
        for j in range(1, width - 1):
            # 获取中心像素值
            center = gray_image[i][j]

            # 初始化要比较的像素
            pixels = []

            # 按顺时针方向从左上角像素开始，将邻近像素加入列表
            pixels.append(gray_image[i - 1][j - 1] > center)
            pixels.append(gray_image[i - 1][j] > center)
            pixels.append(gray_image[i - 1][j + 1] > center)
            pixels.append(gray_image[i][j + 1] > center)
            pixels.append(gray_image[i + 1][j + 1] > center)
            pixels.append(gray_image[i + 1][j] > center)
            pixels.append(gray_image[i + 1][j - 1] > center)
            pixels.append(gray_image[i][j - 1] > center)

```

```

        # 将布尔列表转换为一个字节，每个位表示一个邻近像素
        value = sum([1 << (7 - index) for index, val in enumerate(pixels) if val])

        # 将该字节值赋给 LBP 图像中与中心像素相同位置
        lbp[i, j] = value

    return lbp

# 加载图像，计算其 LBP，并显示的函数
def process_and_display_images(filepaths):
    for filepath in filepaths:
        # 读取图像
        image = cv2.imread(filepath, cv2.IMREAD_COLOR)

        # 计算图像的 LBP
        lbp_img = lbp_image(image)

        # 显示 LBP 图像
        plt.imshow(lbp_img, cmap='gray')
        plt.show()

# 增强版计算 LBP 图像的函数
def lbp_image_enhanced(image):
    # 检查图像是否已经是灰度图
    if len(image.shape) > 2 and image.shape[2] == 3:
        # 将图像转换为灰度图
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    else:
        gray_image = image

    # 获取图像尺寸
    height, width = gray_image.shape

    # 创建一个空的 LBP 图像，大小与输入图像相同
    lbp = np.zeros((height, width), np.uint8)

```

```

for i in range(1, height - 1):
    for j in range(1, width - 1):
        # 获取中心像素值
        center = gray_image[i][j]

        # 初始化要比较的像素
        pixels = []

        # 按顺时针方向从左上角像素开始，将邻近像素加入列表
        pixels.append(gray_image[i - 1][j - 1] > center)
        pixels.append(gray_image[i - 1][j] > center)
        pixels.append(gray_image[i - 1][j + 1] > center)
        pixels.append(gray_image[i][j + 1] > center)
        pixels.append(gray_image[i + 1][j + 1] > center)
        pixels.append(gray_image[i + 1][j] > center)
        pixels.append(gray_image[i + 1][j - 1] > center)
        pixels.append(gray_image[i][j - 1] > center)

        # 将布尔列表转换为一个字节，每个位表示一个邻近像素
        value = sum([1 << (7 - index) for index, val in enumerate(pixels) if val])

        # 将该字节值赋给 LBP 图像中与中心像素相同位置
        lbp[i, j] = value

return lbp

# 加载图像，计算其增强版 LBP，并显示的函数
def process_and_display_images_enhanced(filepaths):
    # 定义轮廓线颜色（此例为蓝色）
    outline_color = (255, 0, 0)

    for filepath in filepaths:
        # 读取图像
        image = cv2.imread(filepath, cv2.IMREAD_GRAYSCALE)
        # 计算图像的 LBP

```

```

lbp_img = lbp_image_enhanced(image)

# 在 LBP 图像上找到轮廓
contours, _ = cv2.findContours(lbp_img, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

# 在原始图像上绘制轮廓
contour_img = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR) # 将灰度图转
换为 BGR 以便上色
cv2.drawContours(contour_img, contours, -1, outline_color, 1) # 用指定颜色绘
制轮廓

# 显示带有轮廓的图像
plt.imshow(cv2.cvtColor(contour_img, cv2.COLOR_BGR2RGB))
plt.show()

# 已上传图像的文件路径
filepaths = ['h02060.jpg', 'w01637.jpg', 'w01870.jpg']
# 处理并显示 LBP 图像
process_and_display_images(filepaths)
# 处理并显示增强版 LBP 图像
process_and_display_images_enhanced(filepaths)

```

## 问题二

```

# p2_dataPreparing
import zipfile
import os
import json
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from PIL import Image

# 定义训练集和测试集
train_zip_path = 'Train.zip'
test_zip_path = 'Test.zip'
train_extract_dir = 'Train/'

```



```

test_extract_dir = 'Test/'

# 解压训练集数据
with zipfile.ZipFile(train_zip_path, 'r') as zip_ref:
    zip_ref.extractall(train_extract_dir)

# 更新路径指向内层的 Train 文件夹
train_extract_dir = os.path.join(train_extract_dir, 'Train/')

# 列出训练文件夹内部的文件以确认
train_files = os.listdir(train_extract_dir)

# 定义加载图像及其注释的函数
def load_image_and_annotations(img_path, ann_path):
    # 加载图像
    img = Image.open(img_path)
    # 加载注释
    with open(ann_path, 'r') as f:
        annotations = json.load(f)
    return img, annotations

# 文件路径
example_image_path = os.path.join(train_extract_dir, 'b02519Z.jpg')
example_annotation_path = os.path.join(train_extract_dir, 'b02519Z.json')

# 加载图像和注释
example_image, example_annotations = load_image_and_annotations(example_image_path,
example_annotation_path)

# 显示图像并绘制边界框
fig, ax = plt.subplots(1)
ax.imshow(example_image)
for ann in example_annotations['ann']:
    rect = patches.Rectangle((ann[0], ann[1]), ann[2] - ann[0], ann[3] - ann[1], linewidth=1,
edgecolor='r',

                                facecolor='none')

```

```

ax.add_patch(rect)

plt.show()

# p2_dataAugmentation
import os
import json
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import albumentations as A

# 定义训练数据集目录
train_extract_dir = 'Train/Train/'

# 定义示例图像和标注文件的路径
example_image_path = os.path.join(train_extract_dir, 'b02519Z.jpg')
example_annotation_path = os.path.join(train_extract_dir, 'b02519Z.json')

# 定义函数加载图像和标注
def load_image_and_annotations(img_path, ann_path):
    # 加载图像
    img = Image.open(img_path)
    # 加载标注
    with open(ann_path, 'r') as f:
        annotations = json.load(f)
    return img, annotations

# 加载示例图像和标注
example_image, example_annotations = load_image_and_annotations(example_image_path,
example_annotation_path)

# 定义数据增强操作
transform = A.Compose([
    A.HorizontalFlip(p=0.5), # 以 50%的概率水平翻转

```

```

A.Rotate(limit=15, p=0.5), # 以 50%的概率旋转±15 度
A.RandomScale(scale_limit=0.1, p=0.5) # 以 50%的概率随机缩放图像
], bbox_params=A.BboxParams(format='pascal_voc', label_fields=['labels']))

# 生成虚拟标签
labels = [1] * len(example_annotations['ann'])

# 应用数据增强到图像及其标注
augmented = transform(image=np.array(example_image),
bboxes=example_annotations['ann'], labels=labels)

# 展示增强后的图像和变换后的边界框
fig, ax = plt.subplots(1)
ax.imshow(augmented['image'])
for bbox in augmented['bboxes']:
    rect = patches.Rectangle((bbox[0], bbox[1]), bbox[2] - bbox[0], bbox[3] - bbox[1],
linewidth=1, edgecolor='r', facecolor='none')
    ax.add_patch(rect)
plt.show()

# 定义水平翻转图像的简单函数
def horizontal_flip(image):
    return image.transpose(Image.FLIP_LEFT_RIGHT)

# 定义旋转图像的简单函数
def rotate_image(image, angle):
    return image.rotate(angle)

# 应用水平翻转
flipped_image = horizontal_flip(example_image)

# 应用旋转
rotated_image = rotate_image(example_image, 15)

# 展示原始图像及其变换后的版本
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

```

```

ax[0].imshow(example_image)
ax[0].set_title('原始图像')
ax[0].axis('off')

ax[1].imshow(flipped_image)
ax[1].set_title('水平翻转')
ax[1].axis('off')

ax[2].imshow(rotated_image)
ax[2].set_title('旋转 15 度')
ax[2].axis('off')

plt.show()

# p2_modelBuilding
import os
import numpy as np
import json
from PIL import Image, ImageDraw
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Conv2DTranspose,
concatenate, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# 构建 U-Net 模型的函数
def build_unet(input_shape):
    inputs = Input(input_shape)

    # 编码路径
    c1 = Conv2D(16, (3, 3), activation='relu', padding='same')(inputs)
    c1 = Dropout(0.1)(c1)
    c1 = Conv2D(16, (3, 3), activation='relu', padding='same')(c1)
    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(32, (3, 3), activation='relu', padding='same')(p1)

```

```

c2 = Dropout(0.1)(c2)
c2 = Conv2D(32, (3, 3), activation='relu', padding='same')(c2)
p2 = MaxPooling2D((2, 2))(c2)

c3 = Conv2D(64, (3, 3), activation='relu', padding='same')(p2)
c3 = Dropout(0.2)(c3)
c3 = Conv2D(64, (3, 3), activation='relu', padding='same')(c3)
p3 = MaxPooling2D((2, 2))(c3)

c4 = Conv2D(128, (3, 3), activation='relu', padding='same')(p3)
c4 = Dropout(0.2)(c4)
c4 = Conv2D(128, (3, 3), activation='relu', padding='same')(c4)
p4 = MaxPooling2D((2, 2))(c4)

# 瓶颈层
c5 = Conv2D(256, (3, 3), activation='relu', padding='same')(p4)
c5 = Dropout(0.3)(c5)
c5 = Conv2D(256, (3, 3), activation='relu', padding='same')(c5)

# 解码路径
u6 = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(c5)
u6 = concatenate([u6, c4])
c6 = Conv2D(128, (3, 3), activation='relu', padding='same')(u6)
c6 = Dropout(0.2)(c6)
c6 = Conv2D(128, (3, 3), activation='relu', padding='same')(c6)

u7 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(c6)
u7 = concatenate([u7, c3])
c7 = Conv2D(64, (3, 3), activation='relu', padding='same')(u7)
c7 = Dropout(0.2)(c7)
c7 = Conv2D(64, (3, 3), activation='relu', padding='same')(c7)

u8 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(c7)
u8 = concatenate([u8, c2])
c8 = Conv2D(32, (3, 3), activation='relu', padding='same')(u8)
c8 = Dropout(0.1)(c8)

```

```

c8 = Conv2D(32, (3, 3), activation='relu', padding='same')(c8)

u9 = Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='same')(c8)
u9 = concatenate([u9, c1], axis=3)
c9 = Conv2D(16, (3, 3), activation='relu', padding='same')(u9)
c9 = Dropout(0.1)(c9)
c9 = Conv2D(16, (3, 3), activation='relu', padding='same')(c9)

# 输出层
outputs = Conv2D(1, (1, 1), activation='sigmoid')(c9)

model = Model(inputs=[inputs], outputs=[outputs])
return model

# 输入图像的形状为(128, 128, 1)
unet_model = build_unet((128, 128, 1))
unet_model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])

# 打印模型结构
print(unet_model.summary())

# 定义创建遮罩的函数
def create_mask(image_path, annotations, output_size):
    image = Image.open(image_path)
    image = image.resize(output_size)
    mask = Image.new('L', image.size, 0)
    draw = ImageDraw.Draw(mask)
    for ann in annotations['ann']:
        if ann[-1] == 1.0:
            rectangle = [tuple(ann[:2]), tuple(ann[2:4])]
            draw.rectangle(rectangle, fill=1)
    return np.array(image), np.array(mask)

# 加载示例图像和标注
example_image_path = os.path.join(train_extract_dir, 'b02519Z.jpg')
example_annotation_path = os.path.join(train_extract_dir, 'b02519Z.json')

```

```

with open(example_annotation_path, 'r') as f:
    example_annotations = json.load(f)

# 创建遮罩和处理图像
img, mask = create_mask(example_image_path, example_annotations, (128, 128))
img = np.expand_dims(img, axis=0)
img = np.expand_dims(img, axis=-1)
mask = np.expand_dims(mask, axis=0)
mask = np.expand_dims(mask, axis=-1)

# 数据增强配置
data_gen_args = dict(rotation_range=15,
                      width_shift_range=0.1,
                      height_shift_range=0.1,
                      shear_range=0.05,
                      zoom_range=0.1,
                      horizontal_flip=True,
                      fill_mode='nearest')

image_datagen = ImageDataGenerator(**data_gen_args)
mask_datagen = ImageDataGenerator(**data_gen_args)

# 创建数据生成器
image_generator = image_datagen.flow(img, seed=1, batch_size=20)
mask_generator = mask_datagen.flow(mask, seed=1, batch_size=20)

# 将图像生成器和掩码生成器打包
train_generator = zip(image_generator, mask_generator)

# 训练模型
unet_model.fit(train_generator, steps_per_epoch=50, epochs=20)

# p2_modelEvaluation
import os
import numpy as np
import json

```

```

from PIL import Image, ImageDraw
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Conv2DTranspose,
concatenate, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# 构建 U-Net 模型的函数
def build_unet(input_shape):
    inputs = Input(input_shape)

    # 编码路径
    c1 = Conv2D(16, (3, 3), activation='relu', padding='same')(inputs)
    c1 = Dropout(0.1)(c1)
    c1 = Conv2D(16, (3, 3), activation='relu', padding='same')(c1)
    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(32, (3, 3), activation='relu', padding='same')(p1)
    c2 = Dropout(0.1)(c2)
    c2 = Conv2D(32, (3, 3), activation='relu', padding='same')(c2)
    p2 = MaxPooling2D((2, 2))(c2)

    c3 = Conv2D(64, (3, 3), activation='relu', padding='same')(p2)
    c3 = Dropout(0.2)(c3)
    c3 = Conv2D(64, (3, 3), activation='relu', padding='same')(c3)
    p3 = MaxPooling2D((2, 2))(c3)

    c4 = Conv2D(128, (3, 3), activation='relu', padding='same')(p3)
    c4 = Dropout(0.2)(c4)
    c4 = Conv2D(128, (3, 3), activation='relu', padding='same')(c4)
    p4 = MaxPooling2D((2, 2))(c4)

    # 瓶颈层
    c5 = Conv2D(256, (3, 3), activation='relu', padding='same')(p4)
    c5 = Dropout(0.3)(c5)
    c5 = Conv2D(256, (3, 3), activation='relu', padding='same')(c5)

```



```
# 解码路径
```

```
u6 = Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(c5)
```

```
u6 = concatenate([u6, c4])
```

```
c6 = Conv2D(128, (3, 3), activation='relu', padding='same')(u6)
```

```
c6 = Dropout(0.2)(c6)
```

```
c6 = Conv2D(128, (3, 3), activation='relu', padding='same')(c6)
```

```
u7 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(c6)
```

```
u7 = concatenate([u7, c3])
```

```
c7 = Conv2D(64, (3, 3), activation='relu', padding='same')(u7)
```

```
c7 = Dropout(0.2)(c7)
```

```
c7 = Conv2D(64, (3, 3), activation='relu', padding='same')(c7)
```

```
u8 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(c7)
```

```
u8 = concatenate([u8, c2])
```

```
c8 = Conv2D(32, (3, 3), activation='relu', padding='same')(u8)
```

```
c8 = Dropout(0.1)(c8)
```

```
c8 = Conv2D(32, (3, 3), activation='relu', padding='same')(c8)
```

```
u9 = Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='same')(c8)
```

```
u9 = concatenate([u9, c1], axis=3)
```

```
c9 = Conv2D(16, (3, 3), activation='relu', padding='same')(u9)
```

```
c9 = Dropout(0.1)(c9)
```

```
c9 = Conv2D(16, (3, 3), activation='relu', padding='same')(c9)
```

```
# 输出层
```

```
outputs = Conv2D(1, (1, 1), activation='sigmoid')(c9)
```

```
model = Model(inputs=[inputs], outputs=[outputs])
```

```
return model
```

```
# 输入图像的形状为(128, 128, 1)
```

```
unet_model = build_unet((128, 128, 1))
```

```
unet_model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])
```

```

# 打印模型结构
print(unet_model.summary())

# 定义创建遮罩的函数
def create_mask(image_path, annotations, output_size):
    image = Image.open(image_path)
    image = image.resize(output_size)
    mask = Image.new('L', image.size, 0)
    draw = ImageDraw.Draw(mask)
    for ann in annotations['ann']:
        if ann[-1] == 1.0:
            rectangle = [tuple(ann[:2]), tuple(ann[2:4])]
            draw.rectangle(rectangle, fill=1)
    return np.array(image), np.array(mask)

# 加载示例图像和标注
example_image_path = os.path.join(train_extract_dir, 'b02519Z.jpg')
example_annotation_path = os.path.join(train_extract_dir, 'b02519Z.json')
with open(example_annotation_path, 'r') as f:
    example_annotations = json.load(f)

# 创建遮罩和处理图像
img, mask = create_mask(example_image_path, example_annotations, (128, 128))
img = np.expand_dims(img, axis=0)
img = np.expand_dims(img, axis=-1)
mask = np.expand_dims(mask, axis=0)
mask = np.expand_dims(mask, axis=-1)

# 数据增强配置
data_gen_args = dict(rotation_range=15,
                      width_shift_range=0.1,
                      height_shift_range=0.1,
                      shear_range=0.05,
                      zoom_range=0.1,
                      horizontal_flip=True,
                      fill_mode='nearest')

```

```
image_datagen = ImageDataGenerator(**data_gen_args)
mask_datagen = ImageDataGenerator(**data_gen_args)

# 创建数据生成器
image_generator = image_datagen.flow(img, seed=1, batch_size=20)
mask_generator = mask_datagen.flow(mask, seed=1, batch_size=20)

# 将图像生成器和掩码生成器打包
train_generator = zip(image_generator, mask_generator)

# 训练模型
unet_model.fit(train_generator, steps_per_epoch=50, epochs=20)
```

### 问题三

```
# p3_testResult
import os
import cv2
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from zipfile import ZipFile

# 解压缩文件并处理图像的目录
zip_path = 'Figures.zip'
extract_dir = 'Figures'

# 如果目录不存在则创建
os.makedirs(extract_dir, exist_ok=True)

# 解压 zip 文件
with ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_dir)

# 列出解压后的文件
extracted_files = os.listdir(extract_dir)
```

```

figures_folder = os.path.join(extract_dir, 'Figures')
figure_files = os.listdir(figures_folder)

# 函数：处理图像并检测边缘
def process_image(file_path):
    img = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE)
    blur = cv2.GaussianBlur(img, (5, 5), 0)
    edges = cv2.Canny(blur, 100, 200)
    contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    img_contours = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)
    cv2.drawContours(img_contours, contours, -1, (0, 255, 0), 1)
    return img, img_contours, contours

# 处理第一张图像以示范
first_image_path = os.path.join(figures_folder, figure_files[0])
original_img, img_contours, contours = process_image(first_image_path)

# 展示原始图像与处理后的图像
fig, ax = plt.subplots(1, 2, figsize=(12, 6))
ax[0].imshow(original_img, cmap='gray')
ax[0].set_title('原始图像')
ax[0].axis('off')
ax[1].imshow(img_contours)
ax[1].set_title('带轮廓的图像')
ax[1].axis('off')
plt.show()

# 函数：优化图像处理以更好地检测轮廓
def optimize_image_processing(file_path):
    img = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE)
    blur = cv2.GaussianBlur(img, (5, 5), 0)
    thresh = cv2.adaptiveThreshold(blur, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY_INV, 11, 2)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

```

```

img_contours = cv2.cvtColor(img, cv2.COLOR_GRAY2RGB)
cv2.drawContours(img_contours, contours, -1, (0, 255, 0), 2)
return img, img_contours, contours

# 优化第一张图像的处理
opt_original_img,          opt_img_contours,          opt_contours          =
optimize_image_processing(first_image_path)

# 展示优化后的原始图像和处理图像
fig, ax = plt.subplots(1, 2, figsize=(12, 6))
ax[0].imshow(opt_original_img, cmap='gray')
ax[0].set_title('原始图像')
ax[0].axis('off')
ax[1].imshow(opt_img_contours)
ax[1].set_title('优化后的带轮廓图像')
ax[1].axis('off')
plt.show()

# 函数：处理所有图像并提取轮廓
def process_all_images(image_folder):
    data = []
    for file in os.listdir(image_folder):
        file_path = os.path.join(image_folder, file)
        _, img_contours, contours = optimize_image_processing(file_path)
        bounding_boxes = []
        for contour in contours:
            x, y, w, h = cv2.boundingRect(contour)
            bounding_boxes.append([x, y, x + w, y + h, 1])  # [x1, y1, x2, y2, label]
        result_image_path = os.path.join(image_folder, 'contoured_' + file)
        cv2.imwrite(result_image_path, img_contours)
        data.append([file, bounding_boxes])
    return data

# 在文件夹中处理所有图像
image_data = process_all_images(figures_folder)

```

```
# 将数据保存到 Excel 文件
excel_path = 'Test_results.xlsx'
df = pd.DataFrame(image_data, columns=['图像名称', '标记'])
df.to_excel(excel_path, index=False)
```

#### 问题四

```
# p4_modelBuilding
from zipfile import ZipFile
import os
import cv2
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from PIL import Image
import torchvision.transforms as T
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
import torch
import torchvision
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.rpn import AnchorGenerator
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.transforms import functional as F

# 解压数据集
train_zip_path = 'p4_train.zip'
test_zip_path = 'p4_test.zip'
train_extract_dir = 'p4_train'
test_extract_dir = 'p4_test'

with ZipFile(train_zip_path, 'r') as zip_ref:
    zip_ref.extractall(train_extract_dir)

with ZipFile(test_zip_path, 'r') as zip_ref:
    zip_ref.extractall(test_extract_dir)

# 列出解压后的文件
```

```

train_subdir = os.path.join(train_extract_dir, 'p4_train')
test_subdir = os.path.join(test_extract_dir, 'p4_test')
train_images = os.listdir(train_subdir)
test_images = os.listdir(test_subdir)

# 显示图像的函数
def display_images(image_paths, titles, num_images=3):
    fig, axs = plt.subplots(1, num_images, figsize=(15, 5))
    for i, image_path in enumerate(image_paths[:num_images]):
        img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
        axs[i].imshow(img, cmap='gray')
        axs[i].set_title(titles[i])
        axs[i].axis('off')
    plt.show()

# 从测试集选择样本图像展示
sample_test_images = [os.path.join(test_subdir, img) for img in test_images[:3]]
display_images(sample_test_images, titles=['Test Image 1', 'Test Image 2', 'Test Image 3'])

# 预处理图像的函数
def preprocess_image(image, size=(64, 64)):
    image_resized = cv2.resize(image, size)
    _, image_binarized = cv2.threshold(image_resized, 0, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)
    image_normalized = image_binarized / 255.0
    return image_normalized

sample_train_images_dir = os.path.join(train_subdir, train_images[0])
sample_train_images = [os.path.join(sample_train_images_dir, img) for img in
os.listdir(sample_train_images_dir)[:3]]
preprocessed_train_images = [preprocess_image(cv2.imread(img_path,
cv2.IMREAD_GRAYSCALE)) for img_path in sample_train_images]

# 展示预处理后的图像
def display_preprocessed_images(images, titles):
    fig, axs = plt.subplots(1, len(images), figsize=(15, 5))

```

```

for i, img in enumerate(images):
    axs[i].imshow(img, cmap='gray')
    axs[i].set_title(titles[i])
    axs[i].axis('off')
plt.show()

display_preprocessed_images(preprocessed_train_images, ['Preprocessed Image 1',
'Preprocessed Image 2', 'Preprocessed Image 3'])

# TensorFlow 模型构建
def build_model(num_classes):
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 1)),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Conv2D(128, (3, 3), activation='relu'),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(num_classes, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
    return model

# 模型训练
model = build_model(10)
model.fit(X_train, y_train, epochs=10, validation_data=(X_val, y_val))

# PyTorch Faster R-CNN 模型构建
def get_model(num_classes):
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
    return model

```



```

# 加载图像用于展示预测结果
def load_image(image_path):
    img = Image.open(image_path)
    return img

# 展示图像上的预测结果
def plot_predictions(image, predictions, threshold=0.5):
    fig, ax = plt.subplots(1)
    ax.imshow(image)
    for element in range(len(predictions['boxes'])):
        box = predictions['boxes'][element].cpu().detach().numpy()
        score = predictions['scores'][element].cpu().detach().numpy()
        label = predictions['labels'][element].cpu().detach().numpy()
        if score > threshold:
            x, y, xmax, ymax = box
            rect = patches.Rectangle((x, y), xmax - x, ymax - y, linewidth=1, edgecolor='r',
facecolor='none')
            ax.add_patch(rect)
            ax.text(x, y, f'{label}: {score:.2f}', fontsize=10, color='white',
bbox=dict(facecolor='red', alpha=0.5))
    plt.show()

# 实际使用模型进行测试的函数
def test_model(model, test_image_path):
    device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
    model.to(device).eval()
    img = load_image(test_image_path)
    transform = T.Compose([T.ToTensor()])
    img_tensor = transform(img).unsqueeze(0).to(device)
    with torch.no_grad():
        prediction = model(img_tensor)
    plot_predictions(img, prediction[0])

```