《数字图像处理》实验报告

姓名：孔馨怡 学号：22122128

实验 6

（本实验将所有任务合并来描述，但会把每个板块分开描述）

一. 任务

a) 核心代码：

板块一：计算频率并构建哈夫曼树

内容：

定义 huffman 节点

计算图片像素点频率（输入原图）

构建 huffman 树（依据像素频率）：用于创建编码

构建 huffman 编码（依据 huffman 树）：输出图片转化为的 code 编码

此部分完成：根据输入的图片数组（或者预测图片数组）变为 huffman 编码

```
# 定义哈夫曼树节点
class HuffmanNode:
    def __init__(self, value, freq):
        self.value = value  # 值
        self.freq = freq  # 频率
        self.left = None  # 左子树
        self.right = None  # 右子树

    # 重载小于 (<) 运算符。
    # 比较两个节点，频率较小的优先
    # 在heap 形成最小堆的时候用到 否则heapq 不知道哪个大
    def __lt__(self, other):
        return self.freq < other.freq

# 计算频率
def calculate_frequency(img):
    # Counter(...): collections 模块中的一个类，专门用于统计可哈希对象的出现次数。
    # {像素值：出现频率}
    return Counter(img.flatten())
```

```python
def build_huffman_tree(frequencies):
    # 构建哈夫曼树
    # frequencies 是一个字典（dict）
    # 形式：{value1: freq1, value2: freq2, ...}，其中value 是要编码的元素（例如字符），freq 是该元素出现的频率。
    heap = [HuffmanNode(value, freq) for value, freq in frequencies.items()]
    # 将这个列表转换为最小堆。
    heapq.heapify(heap)

    # 由heap 这个最小堆序列 构建huff 树
    while len(heap) > 1:
        # heapq.heappop(heap)用于从最小堆中弹出并返回堆顶的元素。（这里是根结点）
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        # 这个节点没有具体的值
        merged = HuffmanNode(None, node1.freq + node2.freq)
        # 哈夫曼树的构建规则之一：每次合并两个频率最小的节点，生成一个新的节点，频率是这两个节点频率之和。
        merged.left = node1
        merged.right = node2
        # 把这个新节点放在堆顶
        heapq.heappush(heap, merged)

    # 返回根结点
    return heap[0]

# 生成哈夫曼编码
def generate_codes(node, code="", codes={}):
    # 不为空
    if node:
        if node.value is not None:
            # 记录这个有值节点的编码
            codes[node.value] = code
        # 递归实现
        generate_codes(node.left, code + "0", codes)
        generate_codes(node.right, code + "1", codes)
    return codes
```

板块二：实现不同形式编码并压缩图片

内容：

通用压缩图片（从数组到 code 的整合）

Huffman 编码的实现

无损预测编码的实现（这里包含预测的部分和计算 error 的部分）

两种实现的总体步骤整合

此部分完成：具体的不同编码输入图像数组的实现，以及图片转化为比特流，成功压缩为比特流

通用压缩函数：

```
def compress_image(img, codes):
    # 把图像和弄好的code 编码和值对应的hash 转为数据（二进制比特流）
    # 连接所有编码组成原始数据
    # 将图片像素转为哈夫曼编码
    # join()字符串方法 元素连接成字符串 ''是空字符串
    # flatten 二维数组展平为一维
    compressed_data = ''.join([codes[pixel] for pixel in img.flatten()])

    # 返回比特流
    return compressed_data
```

Huffman 编码：

Huffman 编码直接用原 img 的数组就可以了

```
# 对图片进行压缩
def huffman_compress(img):
    # 计算频率(用来build huffman)
    frequencies = calculate_frequency(img)

    # 构建哈夫曼树 返回的根结点
    huffman_tree = build_huffman_tree(frequencies)

    # 用构建好的树生成哈夫曼编码(递归实现) 传入根结点
    codes = generate_codes(huffman_tree)
    # codes[像素值] = 编码

    # 压缩图片成为比特流
    compressed_data = compress_image(img, codes)

    # 返回比特流、编码字典和图像形状
    return compressed_data, codes, img.shape
```

无损预测编码：

这里要预测，计算出一个 error 才丢到 huffman 编码里

```
def predict_and_encode(img):
    height, width = img.shape
    # 初始化误差数组
    # 默认是np.uint8 不支持负数 转为dtype=np.int32
```

```
    errors = np.zeros_like(img, dtype=np.int32)

    for i in range(height):
        for j in range(width):
            if(j == 0):
                errors[i, j] = int(img[i, j])
            else:
                # 使用前一个像素值进行预测
                errors[i, j] = int(int(img[i, j]) - int(img[i, j - 1]))

    return errors

def huffman_predict_compress(img):
    shape = img.shape
    errors = predict_and_encode(img)
    frequencies = calculate_frequency(errors)
    tree = build_huffman_tree(frequencies)
    codes = generate_codes(tree)
    compress_data = compress_image(errors, codes)
    return compress_data, codes, shape
```

板块三：实现不同形式编码的解压图片

内容：

通用解压图片（从 data 到 image 的通用函数）

Huffman 编码的实现

无损预测编码的实现

两种实现的总体步骤整合

此部分完成：压缩完成的数据变为可以让 cv2 展示的正常图像

通用压缩函数：

```
def decompress_img(compressed_data, codes, original_shape):
    # 解码比特流
    decoded_pixels = []
    code = ""
    reverse_codes = {code: pixel for pixel, code in codes.items()}

    for bit in compressed_data:
        code += bit
        if code in reverse_codes:
            decoded_pixels.append(reverse_codes[code])
            code = ""
            # 检查解码后的像素长度
```

```
            if len(decoded_pixels) >= original_shape[0] * original_shape[1]:
                break

    # 还原图片
    img = np.array(decoded_pixels).reshape(original_shape)
    return img
```

Huffman 编码：

```
def huffman_decompress(compressed_data, codes, original_shape):
    return decompress_img(compressed_data, codes, original_shape)
```

无损预测编码：还原预测和 error 为正常图像（注意转换数据类型）

```
def predict_decompress(compressed_data, codes, original_shape):
    errors = decompress_img(compressed_data, codes, original_shape)
    img = np.zeros_like(errors, dtype=np.int32)
    height, width = original_shape

    for i in range(height):
        for j in range(width):
            if j == 0:
                img[i, j] = errors[i, j]
            else:
                img[i, j] = errors[i, j] + img[i, j-1]
    # 从np.int32 转到 np.uint8 不然不是正常图片格式
    img = img.astype(np.uint8)
    return img
```

板块四：性能比较

内容：

根据提供的公式计算 RMSE

计算压缩比

打印各计算结果

此部分完成：打印各数据，比较不同编码压缩的性能

计算平均方误差 RMSE：

```
# 计算平均均方误差（RMSE）
def calculate_rmse(original_image, decompressed_image):
    # 计算每个像素的平方误差
    mse = np.mean((original_image - decompressed_image) ** 2)
    rmse = np.sqrt(mse)  # 取平方根
    print(f"RMSE: {rmse:.2f}")
    return rmse

def calculate_ratio(original_image, compressed_data):
```

计算压缩比：

```
    # 计算压缩比
    original_size = original_image.size * 8  # 原始图像大小（以比特计：像素 * 8）
    compressed_size = len(compressed_data)  # 压缩后数据大小（以比特计）
    ratio = original_size / compressed_size  # 压缩比
    print(f"Original size: {original_size} pixels")
    print(f"Compressed size: {compressed_size} bits")
    print(f"Compression Ratio: {ratio:.2f}")
    return ratio
```

板块五：步骤整合

内容：

整合所有步骤为一个函数

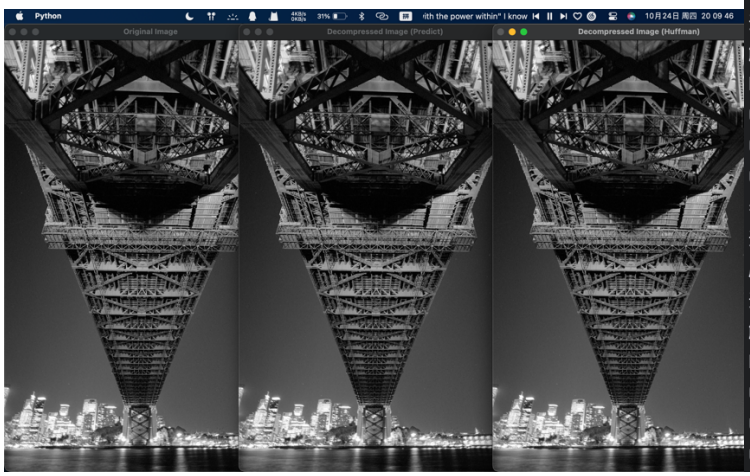
简化并整理测试一张图片时要用到的重复代码片段

此部分完成：实现整个实验的全步骤整合，图片测试的简单化、通用化

```
# ------------------------------------测试图片-----------------------------------
def test_photo(filename):
    print(f"------------------开始测试图像:{filename}------------------")
    # 读取
    image = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)

    # ---------------------- 正常哈夫曼编码 ----------------------
    print(f"========= 正常哈夫曼编码 =========")
    print(f"开始压缩图像（正常哈夫曼编码）: {filename}")
    # 压缩图像
    compressed_data, codes, shape = huffman_compress(image)
    print(f"压缩图像完毕（正常哈夫曼编码）: {filename}")

    print(f"开始解压图像数据（正常哈夫曼编码）: {filename}")
    # 解压图像
    decompressed_image_huffman = huffman_decompress(compressed_data, codes, shape)
    print(f"解压图像数据完毕（正常哈夫曼编码）: {filename}")

    # 计算RMSE 和压缩比
    calculate_rmse(image, decompressed_image_huffman)
    calculate_ratio(image, compressed_data)


    # ---------------------- 无损预测编码 ----------------------
    print(f"========= 无损预测编码 =========")
    print(f"开始压缩图像（无损预测编码）: {filename}")
    compressed_data_predict, codes, shape = huffman_predict_compress(image)
    print(f"压缩图像完毕（无损预测编码）: {filename}")
```

```
    print(f"开始解压图像数据（无损预测编码）: {filename}")
    decompressed_image_predict = predict_decompress(compressed_data_predict, codes,
shape)
    print(f"解压图像数据完毕（无损预测编码）: {filename}")

    # 计算RMSE 和压缩比
    calculate_rmse(image, decompressed_image_predict)
    calculate_ratio(image, compressed_data_predict)

    # 显示解压后的图像
    cv2.imshow("Original Image", image)
    cv2.imshow("Decompressed Image (Huffman)", decompressed_image_huffman)
    cv2.imshow("Decompressed Image (Predict)", decompressed_image_predict)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    return
```

b） 实验结果截图

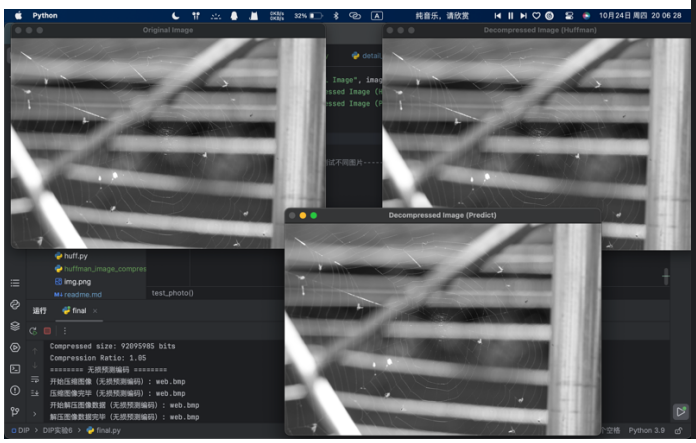



```
------------------开始测试图像:bridge.bmp-----------------
======== 正常哈夫曼编码 ========
开始压缩图像（正常哈夫曼编码）: bridge.bmp
压缩图像完毕（正常哈夫曼编码）: bridge.bmp
开始解压图像数据（正常哈夫曼编码）: bridge.bmp
解压图像数据完毕（正常哈夫曼编码）: bridge.bmp
RMSE: 0.00
Original size: 89045608 pixels
Compressed size: 83002951 bits
Compression Ratio: 1.07
======== 无损预测编码 ========
开始压缩图像（无损预测编码）: bridge.bmp
压缩图像完毕（无损预测编码）: bridge.bmp
开始解压图像数据（无损预测编码）: bridge.bmp
解压图像数据完毕（无损预测编码）: bridge.bmp
RMSE: 0.00
Original size: 89045608 pixels
Compressed size: 53092940 bits
Compression Ratio: 1.68
```

可以看到两种编码（huffman 编码 & 无损预测编码）都是无损压缩：对图片的画质没有任何的影响，RMSE 也都为 0。

Huffman 编码的压缩比接近 1，压缩减少的数据并不是很多。

而无损预测压缩的压缩比到 1.68，压缩效果较 huffman 编码较好。



```
------------------开始测试图像:web.bmp-----------------
======== 正常哈夫曼编码 ========
开始压缩图像（正常哈夫曼编码）: web.bmp
压缩图像完毕（正常哈夫曼编码）: web.bmp
开始解压图像数据（正常哈夫曼编码）: web.bmp
解压图像数据完毕（正常哈夫曼编码）: web.bmp
RMSE: 0.00
Original size: 96968704 pixels
Compressed size: 92095985 bits
Compression Ratio: 1.05
======== 无损预测编码 ========
开始压缩图像（无损预测编码）: web.bmp
压缩图像完毕（无损预测编码）: web.bmp
开始解压图像数据（无损预测编码）: web.bmp
解压图像数据完毕（无损预测编码）: web.bmp
RMSE: 0.00
Original size: 96968704 pixels
Compressed size: 29327077 bits
Compression Ratio: 3.31
```

可以看到两种编码（huffman 编码 & 无损预测编码）都是无损压缩：对图片的画质没有任何的影响，RMSE 也都为 0。

Huffman 编码的压缩比接近 1，压缩减少的数据并不是很多。

而无损预测压缩的压缩比到 3.31，压缩效果非常好。

c） 实验小结

基本思想：具体步骤思想已在代码实现中分板块解释，还可以查看详细的注释。

结果分析：以在以上实验结果截图下方叙述完成。

总结：

通过本实验实现了哈夫曼编码和无损预测编码的压缩与解压过程，并对两者的性能进行了对比分析。实验结果表明，无损预测编码在压缩比上明显优于哈夫曼编码，同时两种编码在解压时都能保证图像质量无损，均达到 RMSE 为 0 的效果。