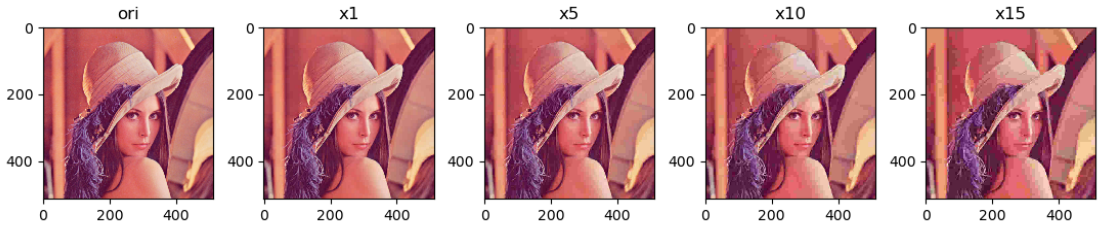
第三次大作业技术报告

环境配置

使用python 3.8 以及numpy、 cv2、 matplotlib等基础库

实验概述

- 本实验实现了基础的**JPEG算法**，将图片通过JPEG压缩称为二进制文件（.txt文件），并且通过JPEG逆变换完成了重新解压
  进一步本文实现了**可控压缩比。**根据用户输入的参数，可以动态调节图片的压缩率，在内存和图像质量间进行权衡
  最后本文分析了**工业JPEG算法与理论算法的区别**，比较了各自的特点与优势



- JPEG算法是工业界使用最为广泛的有损压缩技术。总的来数，其可以将图片中的不重要信息过滤掉，只保留关键信息。使得图片在仍然较为清晰的前提下，大小压缩为原来的10-20分之一。其具体的步骤及其意义如下：
  - **编码流程**
    - **图片预处理**：YUV转换（完成RGB到YUV的映射）、填充（使图片能划分成8×8的小块）、分割（划分成8×8的小块）
    - 离散余弦变换**DCT**：关键技术，分开高频和低频信息，是量化与编码的重要前提
    - **量化**：将不重要信息变为0，从而在之后的步骤中丢弃。是JPG有损的源头
    - **编码**：ZigZag、RLE、VLI。将图片变成数字串，并进一步压缩
    - **Huffman编码**：常用的压缩技术
    - **写入二进制文件**：将二进制数写入文件保存
  - **解码流程**
    - **读二进制文件**：将之前保留的文件信息，压缩图像数据读出
    - **逆Huffman编码**：解码
    - **逆编码**：逆VLI变换、逆RLE变换、ZigZag。恢复图像的尺寸
    - **反量化**：此时不重要信息无法被恢复（可视作丢弃）
    - **逆DCT**：转化为像素信息
    - **图片处理**：恢复图片尺寸、RGB转换
- 接下来将详细阐述实验的实现细节，以及各部分的原理。其中很多的内容与图像参考了他人的博客，链接放在了下方的参考文献中。

实现细节

图像预处理

- 在图像预处理部分，主要涉及到图像的读入、颜色空间转换、填充与分割三部分。其中图像的读入较为简单，下面将详细阐述颜色空间转换的原因以及填充与分割的实现细节。

RGB转换YUV

- **颜色空间转换的步骤，实际上是第一次将图像的像素信息按照重要程度进行区分的操作**
- 当我们读入一张.bmp图像后，其shape为(x,x,3)，所存储的是RGB的三通道信息，三个颜色的占比是相通的，一致的。但是，对于人的眼睛来说，实际上对于三个颜色的注意需求是不同的。这是因为人的眼睛通常对于图像的亮度信息较为敏感，而RGB三个颜色对于亮度的贡献占比不同所导致的。YUV颜色空间，就是将亮度信息单独分出，并且保留色差信息（UV），既可以保留原图像的全部信息，又可以将重要程度进行区分。下面这张图生动的展示了该转化的意义


Y


Cb


Cr

- 至此，加入我们将亮度信息尽量保留，并且找到合适的策略将Cb，Cr信息适度忽略，就可以在保证人眼分辨不出来的前提下，完成图片的压缩，使得效果最优。其计算公式如下：

$$
\begin{aligned}
Y &= 0.299\cdot R + 0.5870\cdot G + 0.114\cdot B \\
C_b &= -0.1687\cdot R - 0.3313\cdot G + 0.5\cdot B \\
C_r &= 0.5\cdot R - 0.4187\cdot G - 0.0813\cdot B
\end{aligned}
$$

填充&分割

- 在将图像转化到YUV颜色空间了之后，其依然保留了原来图片的尺寸。但是，在接下来的量化过程中，需要操作的对象是(8,8,1)的尺寸的像素单元，这就意味着我们需要确保下一环节图像的尺寸应该是8的倍数，所以就需要对图像进行填充补0，其代码如下：

```
    def __Fill(self, matrix):
        fh, fw = 0, 0
        if self.height % 16 != 0:
            fh = 16 - self.height % 16
        if self.width % 16 != 0:
            fw = 16 - self.width % 16
        res = np.pad(matrix, ((0, fh), (0, fw)), 'constant',
constant_values=(0, 0))
        return res
```

- 在填充好后，我们进一步将其分成8×8的小块，方便之后的步骤单独处理

离散余弦变换DCT

- **DCT变换的步骤，实际上是第二次将图像的像素信息按照重要程度进行区分的操作**
- 接下来便是关键的DCT变换。由于上一次实验已经完成了对于DCT的实现，并且分析了其图像信息的分部，其原理便不在此赘述了。
  我们可以知道，图像经过DCT变换后，其每一个位置的值会变小，并且大部分的低频信息会被集中在左上角，决定着图像的大部分特征。但是按照理论的算法有着计算速度慢的缺点，于是我采用了矩阵运算的DCT替代公式，如下所示：

$$预先变换矩阵C(CosineTransformMatrix)满足：$$

$$C(i,j)=\frac{1}{\sqrt{N}}\ \ if\ \ i=0$$

$$C(i,j)=\sqrt{\frac{2}{N}}\cos[\frac{(2j+1)i\pi}{2N}]\ \ if\ \ i>0$$

$$计算得到C后，设C的转置矩阵为C_i，则DCT可以表示为：$$

$$DCT=C\times 像素矩阵\times C_i$$

-
- 对于每一个8×8的小块来说，在已经计算好C的前期下，其DCT变换的代码就较为简单，如下所示：

```
    def __Dct(self, block):
        res = np.dot(self.__dctA, block)
        res = np.dot(res, np.transpose(self.__dctA))
        return res
```

  计算过后，图片的绝大多数信息都被集中在了矩阵的左上角，也被称为图像的直流信息。而其他的表示图像边缘的高频信息则被称作是交流信息。

量化与压缩率

- **量化的步骤，实际上将图像分离出来的不重要像素信息进行过滤**
- 接下来是整个JPEG算法最重要的一步：量子化(Quantization)。在经过DCT变换后，每一个8×8的像素单元已经被转化为了频域的数据，但是由于DCT变换的输出范围是从-1024到1023，**占11位**，保留全部数据会消耗大量内存，起不到压缩的效果。所以需要使用量子化，通过整除运算**减少输出值的存储位数**。量子化公式为：

$$量化后的值(i,j)=Round(\frac{DCT(i,j)}{量子(i,j)})$$

  其中，每一位对应的除数也就是量子，组成了量子化矩阵，其大小也为8×8。工业JPEG算法提供了两张标准的量化系数矩阵，已经被验证有着很好的量化效果，可以用于处理亮度数据Y和色差数据Cr以及Cb，如下所示：

$$Q_Y=\begin{bmatrix}16&11&10&16&24&40&51&61\\12&12&14&19&26&58&60&55\\14&13&16&24&40&57&69&56\\14&17&22&29&51&87&80&62\\18&22&37&56&68&109&103&77\\24&35&55&64&81&104&113&92\\49&64&78&87&103&121&120&101\\72&92&95&98&112&100&103&99\end{bmatrix}$$

标准亮度量化表

$$Q_C=\begin{bmatrix}17&18&24&47&99&99&99&99\\18&21&26&66&99&99&99&99\\24&26&56&99&99&99&99&99\\47&66&99&99&99&99&99&99\\99&99&99&99&99&99&99&99\\99&99&99&99&99&99&99&99\\99&99&99&99&99&99&99&99\\99&99&99&99&99&99&99&99\end{bmatrix}$$

标准色差量化表

可以看到，量化表的大致规律是沿着左上角到右下角的值逐渐变大，其目的就是使得DCT得到的频域数据，左上角的低频数据尽量被保留，而右下角的高频数据适当被丢弃。通过量化，不仅可以完成所有**图像数据的存储位数压缩**，更是可以完成**高低频数据的过滤**，其实现的代码如下所示：

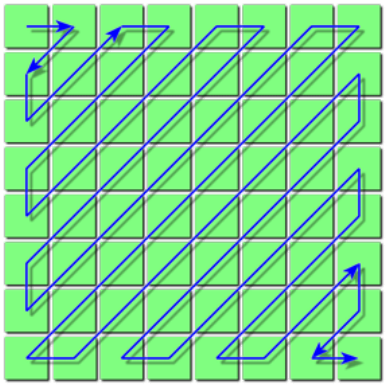
- 除此之外，本文实现的**参数可控压缩率**也是在量化这一部分进行的操作。
  在实际的压缩过程中，还可以根据需要在这些系数的基础上再乘以一个系数，以使更多或更少的数据变成0，我们平时使用的图像处理软件在生成jpg文件时，在控制压缩质量的时候，就是控制的这个系数。但是工业上的标准我并没有找到，所以就自己设计了一个简单的变换，代码如下所示：

```
temp = (self.__lq[i] * self.__quality_scale + 50) / 2
```

经过测试，压缩率系数可以很好地控制图片的压缩质量与大小

ZigZag编码&RLE编码&变长整数编码

- **编码的是三个步骤，实际上将图像二维的信息压缩转化为一维信息。**
- 经过上述变换后，我们已经完成了图像信息按照重要性的分离。但是所有的数据仍然是按照图像的格式进行存储，并且还没有将不重要的信息进行丢弃。通过编码，我们可以将二维信息转化为一维信息，并完成压缩。
- 首先是**ZigZag编码**：
  目前我们的图像信息还是以二维的格式存储，转化为一维信息，遍历是不可或缺的。那么我们如何能找到一种方式，使得遍历之后的一维数据信息数据有序？当前的8×8信息矩阵有着信息集中在左上角的规律，矩阵的右下部分大多为0，如果我们能制定一个遍历顺序，使得大部分的数据集中在前面，后面是全0，那么就可以将全0替换掉，压缩数据序列。所以工业界就制定了一个ZigZag遍历顺序，如下图所示：



其体现在代码中就是一个遍历顺序的矩阵，如下所示：

```
# ZigZag正编码表
self.__zig = np.array([
    0, 1, 8, 16, 9, 2, 3, 10,
    17, 24, 32, 25, 18, 11, 4, 5,
    12, 19, 26, 33, 40, 48, 41, 34,
    27, 20, 13, 6, 7, 14, 21, 28,
    35, 42, 49, 56, 57, 50, 43, 36,
    29, 22, 15, 23, 30, 37, 44, 51,
    58, 59, 52, 45, 38, 31, 39, 46,
    53, 60, 61, 54, 47, 55, 62, 63
])
```

- 接下来是**RLE编码**：

当前我们的数据是，以64唯一单位，大部分非零数据集中在前部。再考虑到数据中0的大量存在，所以需要有针对性的设计一种编码方式，压缩数据串，也就是RLE编码。首先要做的事情，是对串中的0进行处理，把数据中的非零的数据，以及数据前面0的个数作为一个处理单元。如果其中某个单元的0的个数超过16，则需要分成每16个一组。每组由两个元素组成，前面一个元素表示前面的0的个数，后面的元素表示该元素的值。编码的过程如下例：

$$\text{原始数据：}35,7,0,0,0,-6,-2,0,0,-9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,0,0,0,\ldots,0$$
$$35\ 7\ 0,0,0,-6\ \ -2\ 0,0,-9\ 0,0,\ldots,0,8\ 0,0,\ldots,0$$
$$RLE\text{编码：}(0,35)(0,7)(3,-6)(0,-2)(2,-9)(15,0)(2,8)$$
$$\text{其中}(15,0)\text{表示}16\text{个}0$$

至此，对数据串的初步压缩就完成了。对于上述的例子，原本的29个数就被压缩为了14个。代码如下：

```
    def __Rle(self, blist):
        res = []
        cnt = 0
        for i in range(len(blist)):
            if blist[i] != 0:
                res.append(cnt)
                res.append(int(blist[i]))
                cnt = 0
            elif cnt == 15:
                res.append(cnt)
                res.append(int(blist[i]))
                cnt = 0
            else:
                cnt += 1
        # 末尾全是0的情况
        if cnt != 0:
            res.append(cnt - 1)
            res.append(0)
        return res
```

- 最后是**变长整数编码**：

  对于目前已经得到的RLE编码，我们发现其有这样的特点：一组的前一个元素的范围是0-15，后一个元素的值不确定。针对这样的特点，我们需要对后一个元素进行变长整数编码。这是由于如果把整数都按照相同的位数进行存储，但是有的数字的有效位数是1，有的数字的有效位数是100，就会造成大量的内存浪费，压缩效果很差。所以对于每一个整数，首先将其转换为二进制数，先拿一个4位存储他的位数，之后在存储他的有效位。这样就可以做到变长的整数编码。代码如下所示：

```
    def __VLI(self, n):
        ts, tl = 0, 0
        if n > 0:
            ts = bin(n)[2:]
            tl = len(ts)
        elif n < 0:
            tn = (-n) ^ 0xFFFF
            tl = len(bin(-n)[2:])
            ts = bin(tn)[-tl:]
        else:
            tl = 0
            ts = '0'
        return (tl, ts)
```

另外要小心的是，对于负数要单独进行处理，操作过程如代码所示。

- 至此，初步的图像信息序列化以及压缩的工作就已经完成了。

Huffman编码

- Huffman编码是整个压缩的过程的最后一步。Huffman编码的编码原理耳熟能详，就不在此赘述了。本文针对于序列中的所有数字，按照出现的次数，先创建节点，再构造Huffman树，最终生成编码后的序列，其代码如下：

```
class node(object):
    def __init__(self,value = None,left = None,right = None,father = None):

    def build_father(left,right):

    def encode(n):
        if n.father == None:
            return b''
        if n.father.left == n:
            return node.encode(n.father) + b'0'
        else:
            return node.encode(n.father) + b'1'
class tree(object):
    def __init__(self):

    def build_tree(self, l):

    def encode(self, echo):
        for x in self.node_dict.keys():
            self.ec_dict[x] = node.encode(self.node_dict[x])
            if echo == True:
                print(x)
                print(self.ec_dict[x])
```

- 对于树中的每一个节点，通过遍历父节点的方式完成编码。至此，所有的压缩工作就已经结束了。

二进制文件读写

- 这一部分是二进制文件的读写。由于实验需要看到压缩前和压缩后的文件大小对比，所以需要将JPEG压缩后的图像数据信息写入到二进制文件中。这一部分的工作内容很多很复杂，这是因为在写入文件时，并不能简简单单地只把编码后的数据写入。还需要将图片的长度宽度信息、Huffman编码的对应关系、不同通道数据的长度信息全部作为文件头写入到二进制文件中。这样才可以在解码的时候恢复成.bmp文件。
- 这一部分的工作需要使用文件流，按二进制位一位一位地将数据写入，其代码如下所示：

```
for ti in range(4):
    buff = (buff << 1) | ((td & 0x08) >> 3)
    td <<= 1
    bcnt += 1
    if bcnt == 8:
        o.write(buff.to_bytes(1, byteorder='big'))
        o.flush()
        buff = 0
        bcnt = 0
```

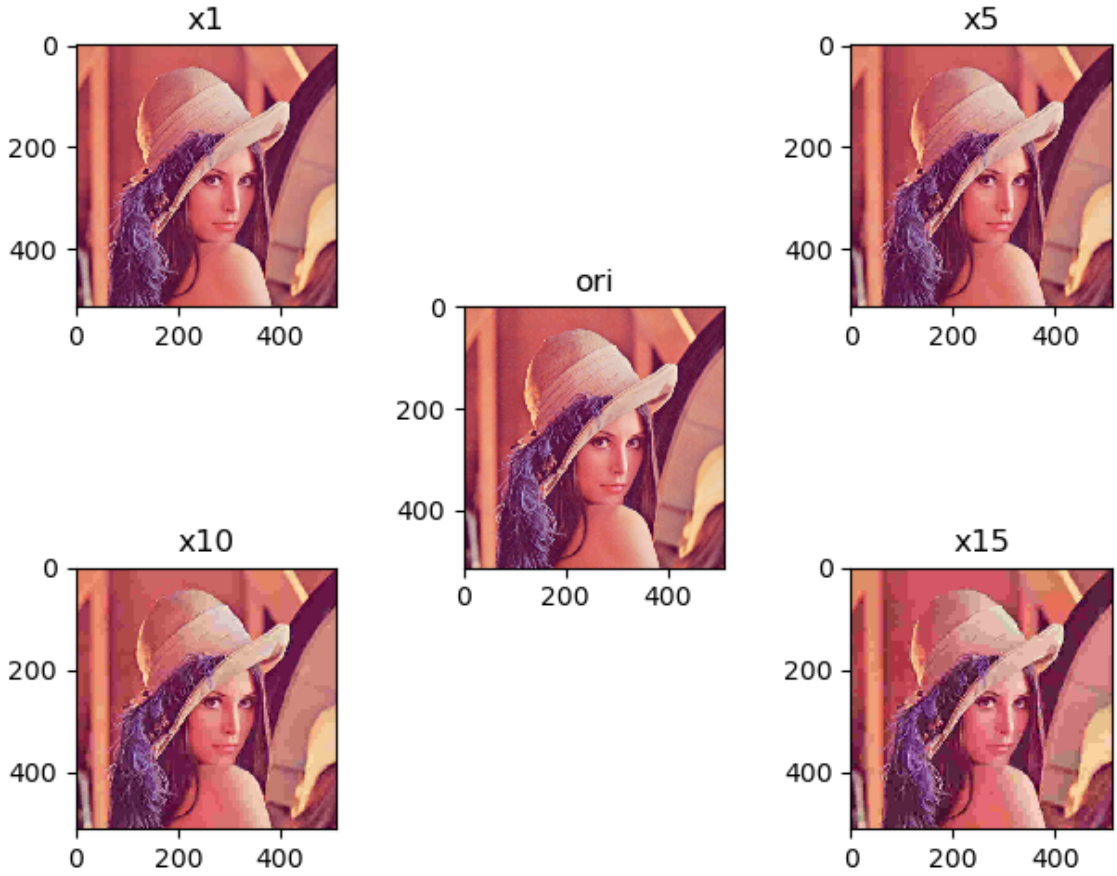
当写够一个BYTE时，便将数据flush进文件中。至此，JPEG的正向的压缩工作就全部完成了。

反变换

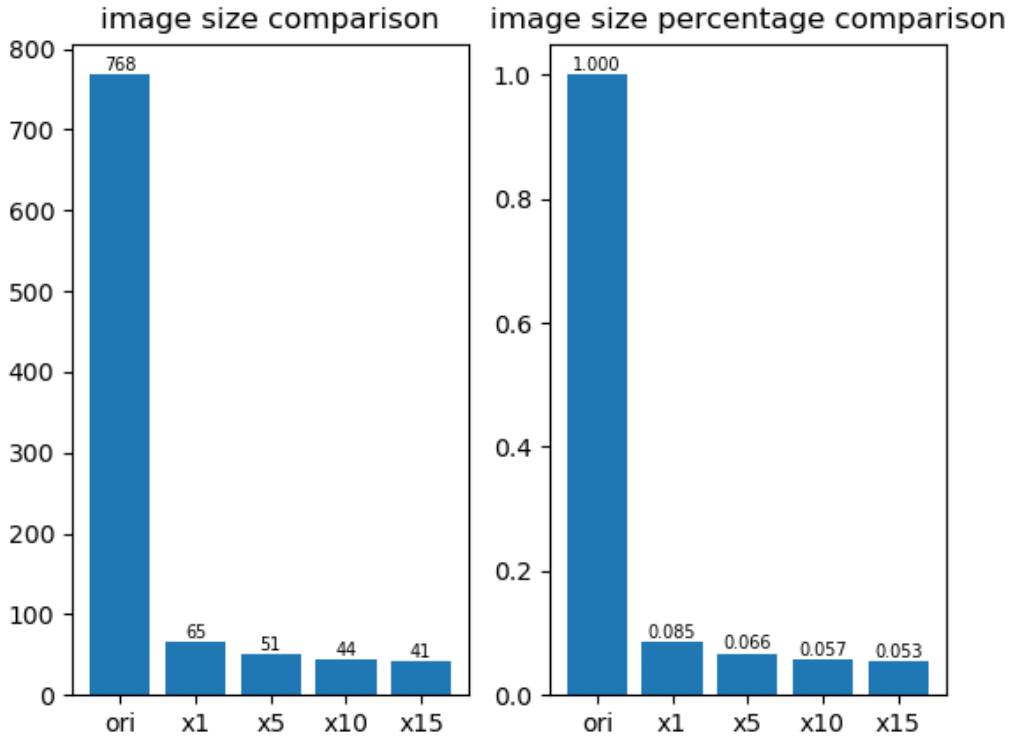
- 为了看到压缩后的图像信息相比于原图的差别，我们需要将二进制文件反变换成.bmp文件
- 反变换部分的工作流程与原理实际上与正向的压缩过程没有任何区别。只是需要注意在读取文件的时候要按照约定先读取头信息，进一步完成全部的反变换工作。

实验结果

- 本实验选用经典lenna图片，分别在压缩系数为1、5、10、15的条件下对图片进行了压缩与恢复，实验结果如下：



- 可以看到所有的压缩图像均保持了较好的图像形态，这表明本压缩算法的正确性良好。进一步，随着压缩系数的逐渐增大，可以看到图像的细节逐渐丢失。这是由于当压缩系数很大时，会使得量化后的矩阵中有更多的数变为0，从而在恢复中无法复原，成为损失。进一步，我测量了图像在压缩前后的大小，并绘制了相关曲线：



- 可以看待当系数为1的时候，压缩比就可以达到10倍，并且随着系数的增大，压缩效果也在不断地变好。尽管size的变化并不大，但这是由于源文件本身就不大的原因所导致的，对于压缩比来说，当系数为15的时候，其压缩比可以接近20倍。可见，通过控制压缩系数，我们可以在10-20间调整JPEG的压缩比，与图片效果做一个权衡。

反思体会

- 在完成实验的前期阶段，我查阅了很多博客与工业的规格，但是处于实现难度等原因，最终本实验的JPEG算法与当下的工业算法有所**差别**，大概是以下几点：
    - 首先最重要的是**Huffman编码**。工业界的编码没有采用建树的这样的纯理论做法。而是根据统计学规律有固定的编码表，直接可以完成编码的映射。这样做得好处是减少了代码的冗余，降低了内存消耗，提高了计算效率。但是可能对于单一的文件来说，其压缩率并没有那么高。
    - 第二点是**.jpg文件格式**。当我们使用上树的一系列JPEG算法完成图像的压缩之后，理论上是可以存成.jpg文件格式，并且在windows上通过图像查看器直接点开的。但是这实现起来有一定难度。首先是.jpg文件有固定的文件头，需要我那个里面写很多东西，这无疑会增加一定的冗余，干扰试验结果。第二点是，.jpg的工业化实现需要把Huffman编码的对应表以矩阵形式写进去，才能让图像查看器解码，但是我没有具体研究，所以并不了解。
- 这一次的作业真的难度很大很综合。不过好在JPEG是很成熟的技术，有很多的代码、文章可以参考，这对于理解原理的帮助巨大，我也都进行了整理，放在了参考文献部分。一个简简单单的JPEG，原先在windows的一键转格式，没想到底层的原理和推导如此复杂；原先双击就可以完成.jpg文件的展示，没想到图像查看器居然做了解压的一系列工作。这在没有做这次的实验之前是没有机会去了解的。
- 这一学期的课程在这里差不多也就到达尾声了。理论课+三次作业真的令我学到了很多，只是有的时候可能精力不足，部分工作做的仍有提高的余地。在这里感谢老师与助教的付出！

参考文献

- ZigZag的原理与用法
- Huffman编码的原理与用法
- JPEG的流程讲解（写的很好）
- JPEG编码器与反编码器讲解
- 变长整数编码