

Jpeg 图像压缩项目报告

胡晟

19214785

更新: *December 7, 2019*

1 代码文件组织结构及运行方法

1.1 代码组织结构

- encoder.py: 包含 encoder 主体代码
- decoder.py: 包含 decoder 主体代码
- huffman.py: 包含哈夫曼树构建以及对应哈夫曼表输出的代码
- utils.py: 包含编码主要步骤用到的函数, 以及一些常量定义
- Controller.py: 包含控制编解码的代码

1.2 代码运行方法以及注意事项

运行 Controller.py 会自动对图片进行编码, 生成中间文件并保存, 然后在从中间文件中重建出图像。

原始图像的地址, 中间文件输出地址, 重建图像输出地址由 utils.py 中的几个常量定义。用于编码的图片长宽需要为 8 的倍数。

2 图片压缩相关工作回顾

图像压缩技术是数据压缩技术在数字图像上的应用, 目的是较少图像数据中的冗余信息, 并以更高效的格式保存并传输图像。图像压缩可以分为无损压缩以及有损压缩。其中无损压缩主要有以下几种方法: 哈夫曼编码, 游程编码, 熵编码以及 LZW 自适应字典算法。主要用于对数据精度有要求的压缩场景。而有损编码主要采用变化色彩空间, 色度抽样, 变换编码, 分行压缩等, 主要用于对数据大小敏感的场景。

由于无损压缩提供的压缩率优先, 无法满足大量图片存储以及传输的需要, 所以对有损压缩算法的研究十分火热, 其中 DCT 变化被广泛应用。经典的 JPEG 算法即使用了 DCT 变换作为算法核

心。随着技术的发展，人们逐渐使用小波变换作为图像压缩的核心算法，经典的 JPEG2000 即使用了这种方法。

今年来随着深度学习的崛起，越来越多的人开始使用深度学习模型对图像压缩进行尝试，CNN 模型在图像处理方面的能力十分强大，为图像压缩有更高的压缩率以及更好的信息保存率提供了可能。

本次工作主要着眼于经典的 JPEG 算法，对 JPEG 编解码器设计实现。

3 JPEG 编码器

3.1 JPEG 编码器原理

Jpeg 编码中主要涉及以下几个操作：

- 图像分割
- 色相变换
- DCT 变换
- 量化
- 熵编码

编码器的第一步操作是将读入的图片分割成 8×8 的小块，每个小块在压缩过程中是单独处理的，可以并行处理加速 Jpeg 压缩。

第二步是将图片的颜色空间从 RGB 转换成 $YCbCr$ ，因为人眼对图片明暗的变化更容易感知，而 $YCbCr$ 可以将图片中重要的信息和不重要的信息分开，可以通过下采样去除对图片视觉效果影响不大的数据，达到有损压缩的目的。

第三步是 DCT 变换，图像经过 DCT 变换后，转换成代表不同频率分量的系数集，这使得能量大部分集中在频域的一个小范围内，可以减少描述不重要分量的比特数，与此同时，频率域分解映射了人类视觉系统的处理过程，为后继的量化过程满足其灵敏度要求。从 DCT 变换后的矩阵可以发现，矩阵的能量主要集中在左上角部分（直流分量），矩阵的其余部分（交流分量）的值都相对较小。DCT 变换是无损的，并没有起到压缩的作用，但是为后面压缩的奠定了基础。

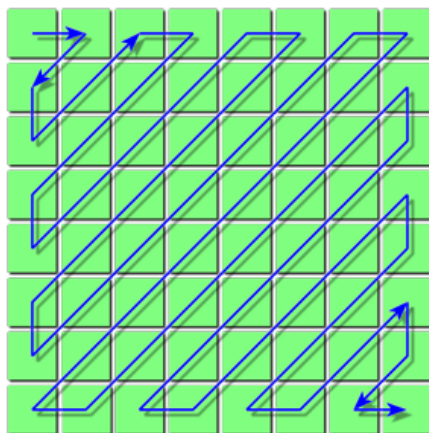
$$\begin{bmatrix} 34 & 34 & 34 & 33 & 34 & 28 & 35 & 32 \\ 34 & 34 & 34 & 33 & 34 & 28 & 35 & 32 \\ 34 & 34 & 34 & 33 & 34 & 28 & 35 & 32 \\ 34 & 34 & 34 & 33 & 34 & 28 & 35 & 32 \\ 36 & 36 & 29 & 27 & 33 & 31 & 30 & 31 \\ 32 & 32 & 35 & 30 & 32 & 33 & 31 & 27 \\ 30 & 30 & 27 & 28 & 30 & 30 & 28 & 29 \end{bmatrix} \xrightarrow{\text{DCT}} \begin{bmatrix} 257.1 & 6.4 & 2.5 & -0.3 & 0.4 & 0.1 & -6.0 & 6.9 \\ 8.4 & 0.0 & 0.5 & -5.0 & 1.9 & 3.4 & -4.2 & 3.3 \\ -5.3 & -1.0 & -1.4 & 1.3 & -0.7 & -0.5 & 2.1 & -1.7 \\ 2.4 & 1.7 & 1.5 & 1.5 & -0.6 & -1.5 & 0.2 & 0.4 \\ -1.1 & -1.6 & -0.2 & -1.8 & 1.6 & 1.2 & -1.4 & -0.1 \\ 1.4 & 0.9 & -1.9 & -0.1 & -2.0 & 0.9 & 1.5 & 0.7 \\ -2.0 & -0.1 & 3.1 & 2.0 & 1.8 & -2.7 & -0.9 & -1.3 \\ 1.5 & -0.2 & -2.3 & -1.9 & -1.0 & 2.3 & 0.3 & 1.1 \end{bmatrix}$$

图 1: DCT 变换示意图

第四步是数据量化。之前的步骤都是无损的，数据量化通过损失一部分精度的方法使用更少的

$$B_{i,j} = \text{round}(\frac{G_{i,j}}{Q_{i,j}})i, j = 0, 1, 2, \dots, 7 \quad (1)$$
$$\begin{bmatrix} -415.38 & -30.19 & -61.20 & 27.24 & 56.12 & -20.10 & -2.39 & 0.46 \\ 4.47 & -21.86 & -60.76 & 10.25 & 13.15 & -7.09 & -8.54 & 4.88 \\ -46.83 & 7.37 & 77.13 & -24.56 & -28.91 & 9.93 & 5.42 & -5.65 \\ -48.53 & 12.07 & 34.10 & -14.76 & -10.24 & 6.30 & 1.83 & 1.95 \\ 12.12 & -6.55 & -13.20 & -3.95 & -1.87 & 1.75 & -2.79 & 3.14 \\ -7.73 & 2.91 & 2.38 & -5.94 & -2.38 & 0.94 & 4.30 & 1.85 \\ -1.03 & 0.18 & 0.42 & -2.42 & -0.88 & -3.02 & 4.12 & -0.66 \\ -0.17 & 0.14 & -1.07 & -4.19 & -1.17 & -0.10 & 0.50 & 1.68 \end{bmatrix} \xrightarrow{\text{Quantization}} \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

为了方便哈夫曼压缩，JPEG 采用 zigzag 的方法，这是因为研究量化后的矩阵可以发现高能量的数据主要集中在左上角，0 主要集中在右下部分，为了将 0 尽可能集中在一起，所以采用以下遍历顺序，得到一维整数数组。



最后一步是熵编码（哈夫曼编码），哈夫曼压缩根据数据中元素的使用频率，调整元素的编码长度，以得到更高的压缩比。经过数据量化得到的一维数组中 0 出现的概率非常高，所以需要对其做进一步处理，JPEG 中采取的方法是使用游长编码 (Run Length Coding)，具体做法如图 4 所示

①原始数据	35,7,0,0,0,-6,-2,0,0,-9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,0,0,0,...,0							
②RLE编码	35	7	0,0,0,-6	-2	0,0,-9	0,0,...,0,8		0,0,...,0
	35	7	0,0,0,-6	-2	0,0,-9	0,0,...,0	0,0,8	0,0,...,0
	(0,35)	(0,7)	(3,-6)	(0,-2)	(2,-9)	(15,0)	(2,8)	EOB

3

RLE 将每个非零元素与前面连续的 0 作为一组，值得注意的是如果某个单元 0 的个数超过 16，需要分为 16 个一组，如果最后一个单元全为 0，那么需要使用“EOB”特数表示。JPEG 还为数值编码提供了一张标准的编码表。根据这张数字编码表可以将一个数字用位长和二进制表示，那么每

图 5: 数字编码表

[illegible]

图 6: 游长编码表示

3.2 JPEG 编码器关键代码

```
image = Image.open(IMAGE_PATH)
ycbcr = image.convert('YCbCr')
```



```

:param array:
:return: (RUNLENGTH, BITLENGTH)
'''

last_nonzero = -1
for i, elem in enumerate(array):
    if elem != 0: last_nonzero = i

# (runlength, bitlength)
left_part = []
# bit represent of value
value = []

cur_length = 0

for i, elem in enumerate(array):
    if i > last_nonzero:
        # (0,0) stands for EOB
        left_part.append((0,0))
        value.append(bin_repr(0))
        break
    elif elem == 0 and cur_length < 15:
        cur_length += 1
    else:
        length = bits_required(elem)
        left_part.append((cur_length, length))
        value.append(bin_repr(elem))
        cur_length = 0
return left_part, value

```

哈夫曼表的构造中，使用一个优先队列辅助哈夫曼树的构建，并通过哈夫曼树返回哈夫曼映射表。

```

class HuffmanTree:

    class __Node:
        def __init__(self, freq, value, left, right):
            self.freq, self.value, self.left, self.right = freq, value, left, right

        def __lt__(self, other):
            return self.freq < other.freq

        def __eq__(self, other):
            l = self.freq, self.value, self.left, self.right
            r = other.freq, other.value, other.left, other.right
            return l == r

```

```

def __init__(self, array):
    q = PriorityQueue()

    for val, freq in self.__frequency(array).items():
        q.put(self.__Node(freq, val, None, None))

    self.__buildTree(q)
    self.__root = q.get()

    self.__huffman_repr = dict()

def __frequency(self, array):
    frequency = dict()
    for num in array:
        if num in frequency.keys():
            frequency[num] += 1
        else: frequency[num] = 1
    return frequency

def __buildTree(self, pq):
    while pq.qsize() >= 2:
        tmp1 = pq.get()
        tmp2 = pq.get()
        newNode = self.__Node(tmp1.freq+tmp2.freq, None, tmp1, tmp2)
        pq.put(newNode)

def value_to_huffman_repr(self):
    if len(self.__huffman_repr.keys()) == 0:
        self.build_huffman_table()
    return self.__huffman_repr

def build_huffman_table(self):
    def traverse(node, huffmancode=''):
        if not node: return
        elif not node.left and not node.right:
            self.__huffman_repr[node.value] = huffmancode
            return
        traverse(node.left, huffmancode+'0')
        traverse(node.right, huffmancode+'1')
    traverse(self.__root)

```

最后保存 ac, dc, 通过二者对应的 Y, C 分量的哈夫曼映射表可以得到二者的压缩表示并保存即可完成编码器工作。

4 JPEG 解码器

4.1 JPEG 解码器原理

JPEG 解码器是编码器的逆过程，主要包含以下几个步骤：

- 解哈夫曼序列：对编码后的序列通过哈夫曼映射表，恢复出原 ac，dc 序列。
- 反向 zigzag 扫描：将一维序列恢复成矩阵形式。
- 反量化：乘以量化矩阵得到数据的原表示近似。
- 逆 DCT 操作：对数据进行 DCT 逆变换。
- 色域转换：将图片从 $YCbCr$ 转换为 RGB 格式。

解码的原理与编码器类似，就不做赘述了。

4.2 JPEG 解码器关键代码

解码器最复杂的部分是哈夫曼解码部分，需要正确地从中间文件中获得解码所需的信息。首先需要设计从文件中恢复哈夫曼映射表的函数，其中因为 DC 分量前面没有表示 RunLength 量所以只需要 4 位，即 `DC_CODE_LENGTH_BITS` 长度为 4，而 AC 分量需要包含 RunLength 量以及位数 `AC_CODE_LENGTH_BITS` 长度为 8，正确读出数据后保存在一个字典中并返回。

```
def read_dc_table(self):
    table = dict()

    table_size = self.__read_uint(self.TABLE_SIZE_BITS)
    for _ in range(table_size):
        category = self.__read_uint(self.CATEGORY_BITS)
        code_length = self.__read_uint(self.DC_CODE_LENGTH_BITS)
        code = self.__read_str(code_length)
        table[code] = category
    return table

def read_ac_table(self):
    table = dict()

    table_size = self.__read_uint(self.TABLE_SIZE_BITS)
    for _ in range(table_size):
        run_length = self.__read_uint(self.RUN_LENGTH_BITS)
        size = self.__read_uint(self.SIZE_BITS)
        code_length = self.__read_uint(self.AC_CODE_LENGTH_BITS)
        code = self.__read_str(code_length)
        table[code] = (run_length, size)
    return table
```


接下来是哈夫曼解码的主体，根据哈夫曼映射表，恢复出 dc,ac 序列。首先恢复 dc 分量，读出 category 表示 dc 分量值的位长，根据位长获取正确的值的位表示；接着读取 ac 分量，首先获得 RunLength 以及位长，接着对 0 进行填充，并恢复出二进制表示。

```
for block_index in range(blocks_count):
    for component in range(3):
        dc_table = tables['dc_y'] if component == 0 else tables['dc_c']
        ac_table = tables['ac_y'] if component == 0 else tables['ac_c']

        category = reader.read_huffman_code(dc_table)
        dc[block_index, component] = reader.read_int(category)

        cells_count = 0

        while cells_count < 63:
            run_length, size = reader.read_huffman_code(ac_table)

            if (run_length, size) == (0, 0):
                while cells_count < 63:
                    ac[block_index, cells_count, component] = 0
                    cells_count += 1
            else:
                for i in range(run_length):
                    ac[block_index, cells_count, component] = 0
                    cells_count += 1
                if size == 0:
                    ac[block_index, cells_count, component] = 0
                else:
                    value = reader.read_int(size)
                    ac[block_index, cells_count, component] = value
                cells_count += 1
```

通过哈夫曼解码得到 dc, ac 序列后，解码器对数据进行余下操作如下：

```
for block_index in range(blocks_number):
    i = block_index // block_pre_line * BLOCK_SIDE
    j = block_index % block_pre_line * BLOCK_SIDE

    for c in range(3):
        rezigzagged = zigzag_to_block([dc[block_index, c]] + list(ac[block_index, :,
            c]))
        dequantized = dequantize(rezigzagged, 1 if c == 0 else 2)
        idcted = iDCT2D(dequantized)
        matrix[i:i+8, j:j+8, c] = idcted + 128
image = Image.fromarray(matrix, 'YCbCr')
```

```
image = image.convert('RGB')
image.save(REBUILD_PATH)
```

5 实验结果

实验中使用的测试图片为 lena，一张 512 色的 tiff 格式文件，原始尺寸为 769KB 通过 JPEG 编


 lena512color.tiff	2019/12/5 21:36	TIFF 文件	769 KB
---	-----------------	---------	--------

图 8: 原始图片信息

码器得到中间文件如下：其中保存的是 01 值，从文件大小来看编码后的文件为 585KB，小于源文

 encoded	2019/12/6 19:18	文件	585 KB
---	-----------------	----	--------

图 9: 中间文件

件大小，但是变小的不是特别明显，分析原因是保存时将 0, 1 分别用字符'0','1' 保存了，这增加了文件存储量占用空间，如果将位流用一个二进制位表示中间文件会更小。接下来对中间文件进行解码操作，得到重建图片信息如图 10 所示。图 11 展示原图像与重建图像之间的比较，可以看到重建


 rebuild.jpg	2019/12/6 19:18	JPG 文件	51 KB
--	-----------------	--------	-------

图 10: 重建图像

图片相对于原图片稍微有些模糊，这是因为 JPEG 是有损压缩算法，在压缩过程中丢失了部分图片信息。

如果需要进一步提升图像压缩率，可以在图像从 RGB 色域空间转换到 YC_bC_r 后进行采样操作，丢弃一部分对视觉效果影响不大的数据，达到更高的压缩率效果。



原图: lena512color.tiff



重建图像: rebuild.jpq

图 11: 原图与重建图片比较