**实验4 数字图像编码实验**

**作业一: 无损编码/压缩算法实验**

问题1: 实现行程编码压缩, 肉眼观察压缩效果，并计算原图和压缩以后的尺寸，计算压缩率并比较分析；

class RLE:

    def \_\_init\_\_(self, path):

        self.path = path

        self.image\_list = [x for x in listdir(path) if is\_image\_file(x)]

        self.image\_list = sorted(self.image\_list)

    def matrix2list(self, matirx):

        """ 按照行程编码样式将2维数组展开为一维数组 """

        mrows, mcols = matirx.shape[:2]

        mrows -= 1

        mcols -= 1

        mlen = min(mrows, mcols)

        rmatrix = []

        rmatrix.append(matirx[0][0])

        rmatrix.extend(self.first\_encode(matirx, mlen))

        if mcols > mrows:

            rmatrix.extend(

                self.colmore\_middle\_encode(matirx, mlen, mcols, mrows))

            rmatrix.extend(self.colmore\_last\_encode(matirx, mlen, mcols,

                                                    mrows))

        else:

            rmatrix.extend(

                self.rowmore\_middle\_encode(matirx, mlen, mcols, mrows))

            rmatrix.extend(self.rowmore\_last\_encode(matirx, mlen, mcols,

                                                    mrows))

        rmatrix.append(matirx[-1][-1])

        return rmatrix

    def first\_encode(self, matirx, mlen):

        rmatrix = []

        for len in range(1, mlen + 1):

            if (len % 2 == 1):

                for i in range(0, len + 1):

                    rmatrix.append(matirx[i][len - i])

            else:

                for i in range(0, len + 1):

                    rmatrix.append(matirx[len - i][i])

        return rmatrix

    def colmore\_middle\_encode(self, matirx, mlen, mcols, mrows):

        rmatrix = []

        if mlen % 2 == 0:

            for extra in range(mcols - mrows):

                if extra % 2 == 0:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[i][mlen - i + extra + 1])

                else:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[mlen - i][i + extra + 1])

        else:

            for extra in range(mcols - mrows):

                if extra % 2 == 1:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[i][mlen - i + extra + 1])

                else:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[mlen - i][i + extra + 1])

        return rmatrix

    def colmore\_last\_encode(self, matirx, mlen, mcols, mrows):

        rmatrix = []

        if mcols % 2 == 0:

            for len in range(0, mlen - 1):

                if len % 2 == 0:

                    for i in range(mlen - len):

                        rmatrix.append(

                            matirx[mlen - (mlen - 1 - len - i)][mlen - i +

                                                                mcols - mrows])

                else:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen -

                                              i][mlen - (mlen - 1 - len - i) +

                                                 mcols - mrows])

        else:

            for len in range(0, mlen - 1):

                if len % 2 == 1:

                    for i in range(mlen - len):

                        rmatrix.append(

                            matirx[mlen - (mlen - 1 - len - i)][mlen - i +

                                                                mcols - mrows])

                else:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen -

                                              i][mlen - (mlen - 1 - len - i) +

                                                 mcols - mrows])

        return rmatrix

    def rowmore\_middle\_encode(self, matirx, mlen, mcols, mrows):

        rmatrix = []

        if mlen % 2 == 0:

            for extra in range(mrows - mcols):

                if extra % 2 == 1:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[mlen - i + extra + 1][i])

                else:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[i + extra + 1][mlen - i])

        else:

            for extra in range(mrows - mcols):

                if extra % 2 == 0:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[mlen - i + extra + 1][i])

                else:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[i + extra + 1][mlen - i])

        return rmatrix

    def rowmore\_last\_encode(self, matirx, mlen, mcols, mrows):

        rmatrix = []

        if mrows % 2 == 0:

            for len in range(0, mlen - 1):

                if len % 2 == 0:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen - (mlen - 1 - len - i) +

                                              mrows - mcols][mlen - i])

                else:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen - i + mrows -

                                              mcols][mlen -

                                                     (mlen - 1 - len - i)])

        else:

            for len in range(0, mlen - 1):

                if len % 2 == 1:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen - (mlen - 1 - len - i) +

                                              mrows - mcols][mlen - i])

                else:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen - i + mrows -

                                              mcols][mlen -

                                                     (mlen - 1 - len - i)])

        return rmatrix

    def encode(self, lst):

        lst\_encode = np.array([(len(list(group)), name)

                               for name, group in groupby(lst)])

        return lst\_encode.flatten()

    def decode(self, lst\_encode):

        lst = []

        for i in range(0, len(lst\_encode), 2):

            print(lst\_encode[i])

            length = int(lst\_encode[i])

            for j in range(length):

                lst.append(lst\_encode[i + 1])

        return lst

    def test(self):

        img3 = np.array(range(1, 9 + 1)).reshape(3, 3)

        img5 = np.array(range(1, 25 + 1)).reshape(5, 5)

        img4 = np.array(range(1, 16 + 1)).reshape(4, 4)

        img = np.array([[5, 4, 1, 2], [4, 3, 2, 1], [3, 3, 2, 1], [2, 3, 1,

                                                                   0]])

        col\_img46 = np.array(range(1, 24 + 1)).reshape(4, 6)

        col\_img45 = np.array(range(1, 20 + 1)).reshape(4, 5)

        col\_img35 = np.array(range(1, 15 + 1)).reshape(3, 5)

        row\_img53 = np.array(range(1, 15 + 1)).reshape(5, 3)

        row\_img54 = np.array(range(1, 20 + 1)).reshape(5, 4)

        row\_img64 = np.array(range(1, 24 + 1)).reshape(6, 4)

        # code = self.encode(self.matrix2list(col\_img))

        # print(self.decode(code))

        print(self.matrix2list(row\_img64))

    def compress(self, index):

        try:

            image\_path = os.path.join(self.path, self.image\_list[index - 1])

        except:

            print("ERROR！ 并不包含你想要进行RGB处理的这张图片")

        else:

            image\_name = re.findall(r'(.+?)\.', self.image\_list[index - 1])

            image = default\_loader(image\_path)

            size = sys.getsizeof((image.flatten()))

            print("Image {}:".format(index))

            print("Origin Image's Size is {:.2f} KB.".format(size / 1024))

            [b, g, r] = cv2.split(image)

            r\_b = self.encode(self.matrix2list(b)).astype(np.uint8)

            r\_g = self.encode(self.matrix2list(g)).astype(np.uint8)

            r\_r = self.encode(self.matrix2list(r)).astype(np.uint8)

            # # 通过打印下面的语句，可以证明最终的存储比例超过200%，是因为对于

            # # 多维度图片的每个维度通道新建不同的数组进行保存的时候，开辟新数组

            # # 空间导致的过大的存储消耗，与压缩算法本身无关。

            # print(b.flatten().shape, r\_b.shape)

            # print(g.flatten().shape, r\_g.shape)

            # print(r.flatten().shape, r\_r.shape)

            # print(sys.getsizeof(np.array([])))

            r\_size = sys.getsizeof((r\_b)) + sys.getsizeof(

                (r\_g)) + sys.getsizeof((r\_r))

            print(

                "After Run Length Encoding Image's Size is  {:.2f} KB.\nCompressed Image's size is {:.2%} of Origin Image."

                .format(r\_size / 1024, r\_size / size))

            # print("Origin Image's Array Size is {} .".format(

            #     len(image.flatten())))

            # print(

            #     "After Run Length Encoding Image's Array Size is  {} .\nCompressed Image's Array size is {:%} of Origin Image."

            #     .format(

            #         len(r\_b) + len(r\_g) + len(r\_r),

            #         (len(r\_b) + len(r\_g) + len(r\_r)) / len(image.flatten())))

            print()

问题2: 实现哈夫曼压缩, 肉眼观察压缩效果，并计算原图和压缩以后的尺寸，计算压缩率并比较分析；

class HuffmanLetter:

    def \_\_init\_\_(self, letter, freq):

        self.letter = letter

        self.freq = freq

        self.bitstring = ""

    def \_\_repr\_\_(self):

        return f"{self.letter}"

class HuffmanTreeNode:

    def \_\_init\_\_(self, freq, left, right):

        self.freq = freq

        self.left = left

        self.right = right

class Huffman:

    """

    Huffman coding compress for rgb image,

    using variable-length binary replace fixed-length coding

    to reduce image size.

    """

    def \_\_init\_\_(self, path):

        self.path = path

        self.image\_list = [x for x in listdir(path) if is\_image\_file(x)]

        self.image\_list = sorted(self.image\_list)

    def byte\_cut(self, image):

        """

        Split the image according to the length of the Byte (8 bits).

        """

        image\_list = image.flatten()

        chars = {}

        for c in image\_list:

            chars[c] = chars[c] + 1 if c in chars.keys() else 1

        return sorted([HuffmanLetter(c, f) for c, f in chars.items()],

                      key=lambda l: l.freq)

    def build\_tree(self, letters):

        """

        Build huffman tree structure according to original character segment.

        """

        while len(letters) > 1:

            left = letters.pop(0)

            right = letters.pop(0)

            total\_freq = left.freq + right.freq

            node = HuffmanTreeNode(total\_freq, left, right)

            letters.append(node)

            letters.sort(key=lambda l: l.freq)

        return letters[0]

    def traverse\_tree(self, root, bitstring):

        """

        Re-encoding according to the huffman tree structure,

        getting huffman code.

        """

        if type(root) is HuffmanLetter:

            root.bitstring = bitstring

            return [root]

        letters = []

        letters += self.traverse\_tree(root.left, bitstring + "0")

        letters += self.traverse\_tree(root.right, bitstring + "1")

        return letters

    def test(self):

        test\_image = np.array(np.random.randint(0, 25, size=[5, 5]))

        print(test\_image.flatten())

        letters\_list = self.byte\_cut(test\_image)

        print(letters\_list)

        root = self.build\_tree(letters\_list)

        letters = self.traverse\_tree(root, "")

        dict = {}

        for letter in letters:

            dict[letter.letter] = letter.bitstring

        # print(dict)

        compress = ""

        for bs in test\_image.flatten():

            compress += dict[bs]

            # print(bs)

            # print(dict[bs])

        # for c in test\_image:

        #     compress += letter.bitstring

        print(sys.getsizeof(test\_image.flatten()))

        print(sys.getsizeof(compress))

    def huffman\_change(self, image):

        letters\_list = self.byte\_cut(image)

        root = self.build\_tree(letters\_list)

        letters = self.traverse\_tree(root, "")

        dict = {}

        for letter in letters:

            dict[letter.letter] = letter.bitstring

        compress = ""

        for bs in image.flatten():

            compress += dict[bs]

        return compress, dict

    def compress(self, index):

        try:

            image\_path = os.path.join(self.path, self.image\_list[index - 1])

        except:

            print("ERROR！ 并不包含你想要进行RGB处理的这张图片")

        else:

            image\_name = re.findall(r'(.+?)\.', self.image\_list[index - 1])

            image = default\_loader(image\_path)

            size = sys.getsizeof((image.flatten()))

            print("Image {}:".format(index))

            print("Origin Image's Size is {:.2f} KB.".format(size / 1024))

            [b, g, r] = cv2.split(image)

            r\_b, r\_b\_dict = self.huffman\_change(b)

            r\_g, r\_g\_dict = self.huffman\_change(g)

            r\_r, r\_r\_dict = self.huffman\_change(r)

            r = []

            r.append(r\_b)

            r.append(r\_g)

            r.append(r\_r)

            # print(r)

            r\_size = sys.getsizeof(r)

            r\_dict\_size = sys.getsizeof(r\_b\_dict) + sys.getsizeof(

                r\_g\_dict) + sys.getsizeof(r\_r\_dict)

            r\_size\_all = r\_size + r\_dict\_size

            print("After Huffman Encoding Image's Size is  {:.2f} KB.\

                    \nCompressed Image's Huffman coding size is {:.2f} KB.\

                    \nCompressed Image's Huffman coding dictonary size is {:.2f} KB.\

                    \nCompressed Image's size is {:.2%} of Origin Image.".

                  format(r\_size\_all / 1024, r\_size / 1024, r\_dict\_size / 1024,

                         r\_size\_all / size))

            print()

问题3: 实现一维无损预测压缩, 肉眼观察压缩效果，并计算原图和压缩以后的尺寸，计算压缩率并比较分析.

class PredictCode:

    """

    Linear prediction coding

    """

    def \_\_init\_\_(self, path):

        self.path = path

        self.image\_list = [x for x in listdir(path) if is\_image\_file(x)]

        self.image\_list = sorted(self.image\_list)

    def predict\_f(self, x):

        """

        Prediction function: y = x

        """

        return x

    def matrix2list(self, matirx):

        """ 按照行程编码样式将2维数组展开为一维数组 """

        mrows, mcols = matirx.shape[:2]

        mrows -= 1

        mcols -= 1

        mlen = min(mrows, mcols)

        rmatrix = []

        rmatrix.append(matirx[0][0])

        rmatrix.extend(self.first\_encode(matirx, mlen))

        if mcols > mrows:

            rmatrix.extend(

                self.colmore\_middle\_encode(matirx, mlen, mcols, mrows))

            rmatrix.extend(self.colmore\_last\_encode(matirx, mlen, mcols,

                                                    mrows))

        else:

            rmatrix.extend(

                self.rowmore\_middle\_encode(matirx, mlen, mcols, mrows))

            rmatrix.extend(self.rowmore\_last\_encode(matirx, mlen, mcols,

                                                    mrows))

        rmatrix.append(matirx[-1][-1])

        return rmatrix

    def first\_encode(self, matirx, mlen):

        rmatrix = []

        for len in range(1, mlen + 1):

            if (len % 2 == 1):

                for i in range(0, len + 1):

                    rmatrix.append(matirx[i][len - i])

            else:

                for i in range(0, len + 1):

                    rmatrix.append(matirx[len - i][i])

        return rmatrix

    def colmore\_middle\_encode(self, matirx, mlen, mcols, mrows):

        rmatrix = []

        if mlen % 2 == 0:

            for extra in range(mcols - mrows):

                if extra % 2 == 0:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[i][mlen - i + extra + 1])

                else:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[mlen - i][i + extra + 1])

        else:

            for extra in range(mcols - mrows):

                if extra % 2 == 1:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[i][mlen - i + extra + 1])

                else:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[mlen - i][i + extra + 1])

        return rmatrix

    def colmore\_last\_encode(self, matirx, mlen, mcols, mrows):

        rmatrix = []

        if mcols % 2 == 0:

            for len in range(0, mlen - 1):

                if len % 2 == 0:

                    for i in range(mlen - len):

                        rmatrix.append(

                            matirx[mlen - (mlen - 1 - len - i)][mlen - i +

                                                                mcols - mrows])

                else:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen -

                                              i][mlen - (mlen - 1 - len - i) +

                                                 mcols - mrows])

        else:

            for len in range(0, mlen - 1):

                if len % 2 == 1:

                    for i in range(mlen - len):

                        rmatrix.append(

                            matirx[mlen - (mlen - 1 - len - i)][mlen - i +

                                                                mcols - mrows])

                else:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen -

                                              i][mlen - (mlen - 1 - len - i) +

                                                 mcols - mrows])

        return rmatrix

    def rowmore\_middle\_encode(self, matirx, mlen, mcols, mrows):

        rmatrix = []

        if mlen % 2 == 0:

            for extra in range(mrows - mcols):

                if extra % 2 == 1:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[mlen - i + extra + 1][i])

                else:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[i + extra + 1][mlen - i])

        else:

            for extra in range(mrows - mcols):

                if extra % 2 == 0:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[mlen - i + extra + 1][i])

                else:

                    for i in range(mlen + 1):

                        rmatrix.append(matirx[i + extra + 1][mlen - i])

        return rmatrix

    def rowmore\_last\_encode(self, matirx, mlen, mcols, mrows):

        rmatrix = []

        if mrows % 2 == 0:

            for len in range(0, mlen - 1):

                if len % 2 == 0:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen - (mlen - 1 - len - i) +

                                              mrows - mcols][mlen - i])

                else:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen - i + mrows -

                                              mcols][mlen -

                                                     (mlen - 1 - len - i)])

        else:

            for len in range(0, mlen - 1):

                if len % 2 == 1:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen - (mlen - 1 - len - i) +

                                              mrows - mcols][mlen - i])

                else:

                    for i in range(mlen - len):

                        rmatrix.append(matirx[mlen - i + mrows -

                                              mcols][mlen -

                                                     (mlen - 1 - len - i)])

        return rmatrix

    def predict(self, image):

        # Using RLE to improve prediction coding efficiency.

        image\_list = self.matrix2list(image)

        predict\_list = []

        update\_list = []

        predict\_list.append(image\_list[0])

        update\_list.append(0)

        for c in image\_list[1:]:

            pred = self.predict\_f(predict\_list[-1])

            e = c - pred

            predict\_list.append(pred)

            update\_list.append(e)

        return np.int8(predict\_list), np.int8(update\_list)

    def test(self):

        test\_image = np.array(np.random.randint(0, 25, size=[5, 5]))

        predict\_list, update\_list = self.predict(test\_image)

        print(predict\_list)

        print(predict\_list)

        print(sys.getsizeof(test\_image.flatten()))

        # print(sys.getsizeof(predict\_list))

        print(sys.getsizeof(update\_list))

    def compress(self, index):

        try:

            image\_path = os.path.join(self.path, self.image\_list[index - 1])

        except:

            print("ERROR！ 并不包含你想要进行RGB处理的这张图片")

        else:

            image\_name = re.findall(r'(.+?)\.', self.image\_list[index - 1])

            image = default\_loader(image\_path)

            size = sys.getsizeof((image.flatten()))

            print("Image {}:".format(index))

            print("Origin Image's Size is {:.2f} KB.".format(size / 1024))

            [b, g, r] = cv2.split(image)

            r\_b\_pred, r\_b\_update = self.predict(b)

            r\_g\_pred, r\_g\_update = self.predict(g)

            r\_r\_pred, r\_r\_update = self.predict(r)

            r = []

            r.append(r\_b\_update)

            r.append(r\_g\_update)

            r.append(r\_r\_update)

            # print(r\_b\_update)

            # print(r)

            r\_size = sys.getsizeof(r)

            print("After Predict Encoding Image's Size is  {:.2f} KB.\

                    \nCompressed Image's size is {:.2%} of Origin Image.".

                  format(r\_size / 1024, r\_size / size))

            print()

**作业二: 有损压缩/压缩算法实验**

查阅JPEG编码的有关资料，对图像进行JPEG压缩，算法步骤必须包括如下几个部分：图像分块，离散余弦变换，量化，ac和dc系数的Z字形编排。

问题1: 质量因子分别选为20，60，80，对比显示原图与不同质量因子下解码后的图像；

问题2: 记录图像大小、压缩比、均方根误差；对结果进行分析。

class JPEGEncode:

    """

    JPEG 格式压缩

    参考链接：https://blog.csdn.net/yuyangyg/article/details/77968494

    """

    def \_\_init\_\_(self, path):

        self.path = path

        self.image\_list = [x for x in listdir(path) if is\_image\_file(x)]

        self.image\_list = sorted(self.image\_list)

    def compress(self, index, q\_factor):

        try:

            image\_path = os.path.join(self.path, self.image\_list[index - 1])

        except:

            print("ERROR！ 并不包含你想要进行RGB处理的这张图片")

        else:

            image\_name = re.findall(r'(.+?)\.', self.image\_list[index - 1])

            # Step 1: convert rgb image space tp YCrCb space

            image = ycrcb\_loader(image\_path)

            # 图像尺寸调整，以适应分块

            height, width = image.shape[:2]

            if height % 8 != 0 or width % 8 != 0:

                image = np.pad(image, ((0, (8 - height % 8) % 8),

                                       (0, (8 - width % 8) % 8), (0, 0)),

                               "edge")

            height, width = image.shape[:2]

            size = sys.getsizeof((image.flatten()))

            print("Image {}:".format(index))

            print("Origin Image's Size is {:.2f} KB.".format(size / 1024))

            [y, cr, cb] = cv2.split(image)

            # Step 2: DCT decomposition, transform from time-domain to

            # frequency-domain, and choose 8\*8 block

            image\_dct = []

            for img in [y, cr, cb]:

                f\_patches = []

                fi\_patches = []

                # 图像分块

                h\_patches = np.vsplit(img, height // 8)

                for i in range(height // 8):

                    wh\_patches = np.hsplit(h\_patches[i], width // 8)

                    f\_patch = []

                    fi\_patch = []

                    for j in range(width // 8):

                        # DCT 变换

                        patch\_dct = cv2.dct(wh\_patches[j].astype(np.float))

                        f\_patch.append(patch\_dct)

                    f\_patchs = np.hstack(f\_patch)

                    f\_patches.append(f\_patchs)

                img\_dct = np.vstack(f\_patches)

                image\_dct.append(img\_dct)

            image\_dct = np.moveaxis(image\_dct, 0, 2)

            # Step 3: 量化

            image\_dct = np.around(image\_dct / q\_factor)

            # Step 4: 行程编码，转换为一维数组

            rle = RLE()

            [d\_y, d\_cr, d\_cb] = cv2.split(image\_dct)

            image\_rle = []

            for dct in [d\_y, d\_cr, d\_cb]:

                dct\_rle = rle.compress(dct)

                image\_rle.append(dct\_rle)

            # 图像大小计算，压缩比计算

            r\_size = sys.getsizeof((image\_rle))

            print("After Run JPEG Compress Image's Size is  {:.2f} KB.\

                    \nCompressed Image's size is {:.2%} of Origin Image.".

                  format(r\_size / 1024, r\_size / size))

            # Step 5: Huffman Transformation

            ## 不想写了，Step 4 和 Step 5 直接省略，

            ## 如果需要实现，可以参考 作业四的 work1，

            ## 直接在量化之后进行 IDCT 还原

            image\_iq = image\_dct \* q\_factor

            [r\_y, r\_cr, r\_cb] = cv2.split(image\_iq)

            image\_back = []

            for img in [r\_y, r\_cr, r\_cb]:

                f\_patches = []

                # 图像分块

                h\_patches = np.vsplit(img, height // 8)

                for i in range(height // 8):

                    wh\_patches = np.hsplit(h\_patches[i], width // 8)

                    f\_patch = []

                    fi\_patch = []

                    for j in range(width // 8):

                        # IDCT 变换

                        patch\_dct = cv2.idct(wh\_patches[j].astype(np.float))

                        f\_patch.append(patch\_dct)

                    f\_patchs = np.hstack(f\_patch)

                    f\_patches.append(f\_patchs)

                img\_back = np.vstack(f\_patches).astype(np.uint8)

                image\_back.append(img\_back)

            image\_back = np.moveaxis(image\_back, 0, 2)

            # YCrCb 空间转换回 RGB 空间

            image\_back = cv2.cvtColor(image\_back, cv2.COLOR\_YCrCb2BGR)

            cv2.imwrite(

                str(\*image\_name) + " " + str(q\_factor) + " IDCT.png",

                image\_back)

            # 计算均方根误差

            mse = ((image - image\_back)\*\*2).mean()

            print("Compressed Image's MSE is {:.2f}".format(mse))

            print()