Image Processing Homework 3: Image Compression

Andrés Ponce 0616110

January 10, 2021

1 Introduction

Image compression is a task with tremendous importance in the modern world. Countless numbers of images, videos, movies and other visual content are shared every day. Wihtout a way to reduce the memory usage of this content, long-term storage would be impractical and the cost of storing such media would be impractical. A single image with 1920×1080 resolution would consume

$$1920 \times 1080 \times 3 \times 8 = 49,766,400 \text{ bytes}$$

There are various image compression algorithms and standards which allow the easy sharing of such content. Among them is the JPEG compression method, which utilizes many of the methodologies introduced in the textbook. In this report, the following techniques are introduced: Huffman coding, predictive coding, discrete cosine transform.

These methods are implemented in Python and then compared to see the amount of saved storage they allow us to achieve.

2 Methods

2.1 Huffman Coding

Huffman coding remains one of the most famous compression techniques. This technique finds a variable length code such that the most used symbols in our message are assigned the shortest codes. We first calculate the probability of each symbol in the set. We then make a tree by joining the two elements with the lowest probability under a single parent with combined probability. We repeat this procedure until we are left with a single node of probability 1.

This method proves optimal because the lengths of the symbols increases as their frequency decreases. Huffman coding results in a nearly optimal method. Once we have a string with which to encode a message, once converted into a binary format it becomes easier to format. The set of strings will most likely be shorter. In my implementation, the strings with the lowest frequency tended to be much longer on average.

In my experiment, most of the strings ended up being between 3 and 6 bits in length. Since greyscale images were used, this average would be a considerable saving over a definite 8 bytes per pixel. In the decoding stage, we only have to search the tree for the corresponding intensity to that string.

2.2 Discrete Cosine Transform

Similar to the Discrete Fourier Transform, this method decomposes a signal as a combination of trigonometric functions, however the DCT only uses a combination of cosine and scalar values to compress a signal. The formula used in this assignment is

$$f(x,y,u,v) = \alpha_u \alpha_v \sum_{u=1}^{B} \sum_{v=1}^{B} \cos \frac{(2x+1)u\pi}{2B} \ \cos \frac{(2y+1)v\pi}{2B}$$

where B is the block size, which was left to 8 in this assignment.

The implementation was straightforward, however the transformed image contained quite small values. These were extremely tiny values, mostly exponentials with exponent -14 or -15. With this rough implementation it would lead to promising results if a suitable code were chosen for the frequency values.

2.3 Predictive Coding

With predictive coding, we attempt to exploit the similarity in adjacent pixels. We have a predicted value $e(n)$, and we attempt to add it with the sum of previous m samples to obtain our estimate. The formula used in this assignment was

$$f(x, y) = e(n) + \hat{f}(x, y)$$

where $e(n)$ is the difference between the current pixel and the current pixel, $\hat{f}(x, y)$. Specifically, $\hat{f}(x, y)$ is defined as

$$2f(x, y) - f(x-1, y) - f(x, y-1)$$

i.e. we take the difference with the pixel in the previous row and column.

This method, if combined also with a variable length code technique, could also provide a much shorter encoding that 8 bits used for greyscale values. A method such as Huffman could provide a more efficient method than just saving the different smaller integer.

2.4 Run-Length Coding

The main purpose of this type of coding is to reduce many repeated pixel values that are next to each other. For example, if there is a background in an image many pixels might have the exact same value in case of the sky or some building. If we somehow store the value of the pixel and how many consecutive pixels have the same value, we could avoid storing multiple copies of the same value. For example, a tuple with the values (180, 3) would indicate that the following 3 pixels each are of intensity 180.

This coding method can be used to great efficacy when dealing with the raw binary values. BMP files use this compression method as mentioned in the textbook, and for binary images it would probably have the best effect.

3 Code

Following are screenshots from the main code section of each of the methods.

3.1 Huffman

```python
# Replace the pixel values in the image with code from our table
def encode(self):
    self.compressed = []

    for row in range(len(self.gray)):
        tmp = []
        for col in range(len(self.gray[row])):
            tmp.append(self.table[self.gray[row][col]])
        self.compressed.append(tmp)

    return self.compressed

# Take in a deocded image and return the uncompressed version
def decode(self, encoded):
    enc_size = tuple((len(encoded), len(encoded[1])))
    decompressed = np.zeros(enc_size)
    for row in range(len(encoded)):
        for col in range(len(encoded[row])):
            # Retrieve the level corresponding to the symbol
            decompressed[row][col] = self._retrieve_level(encoded[row][col])
    return decompressed
```

```
    def _make_table(self, node, string):
        if node == None:
            return
        elif node.level != -1:
            #print(f'level:{node.level}, prob:{node.prob} code:{st
            self.table[node.level] = string
        self._make_table(node.right, string+'0')
        self._make_table(node.left, string+'1')

    # Given the code, find the corresponding intensity level
    def _retrieve_level(self, symbol):
        node = self.root
        for bit in symbol:
            if bit == '0':
                node = node.right
            elif bit == '1':
                node = node.left
        return node.level

    def _build_tree(self):
        """
            1. Build the nodes priority queue.
            2. Merge the two nodes with the lowest probabilities.
            3. Repeat until done
        """
        print("\tBuilding huffman tree")
        # 1. Build the nodes priority queue
        self._build_nodes_queue()

        # 2. Merge the two nodes with the lowest probability
        while self.queue.qsize() > 1:
            left = self.queue.get()
            right = self.queue.get()

            parent = self._make_parent(left[1], right[1])

            # Return once we have only the root node
            if self.queue.qsize() == 0:
                return parent
            self.queue.put((parent.prob, parent))
```

These two images show the main functions of the Huffman coding procedure: building the tree and getting the frequencies, inserting them into the tree, and then decoding the image.

3.2 Discrete Cosine Transform

```
def encode(self):
    rows, cols = self.gray.shape
    #transform_arr = []
    transform_arr = np.zeros(self.gray.shape, dtype=np.float32)

    # Loop over every pixel
    for i in range(rows):
        tmp = []
        ci = (au_zero if i == 0 else au_non_zero)

        for j in range(cols):
            cj = (au_zero if j == 0 else au_non_zero)
            _sum = 0

            # Loop over neighborhood of each pixel
            for u in range(BLOCK_SIZE):
                for v in range(BLOCK_SIZE):
                    _sum += self.gray[u][v] * ( math.cos(((2*u + 1) * i * math.pi) / (n2))*
                                math.cos(((2*v + 1) * j * math.pi) / n2 ))
            result = ci * cj * _sum
            result /= 4
            transform_arr[i][j] = result
    return transform_arr
```

```python
def decode(self, encoded):
    rows, cols = encoded.shape

    decoded = np.empty(encoded.shape)
    # Loop for every pixel
    for i in range(rows):
        ci = (au_zero if i == 0 else au_non_zero)
        for j in range(cols):
           cj = (au_zero if i == 0 else au_non_zero)

           # The neighborhood of each pixel
           _sum = 0
           for u in range(BLOCK_SIZE):

               for v in range(BLOCK_SIZE):
                   _sum += (encoded[u][v] * math.cos((2 * u + 1) * i * factor) * math.cos((2 * v
           decoded[i][j] = (ci * cj * _sum) / 4
    return decoded
```

3.3 Predictive Coding

```python
    def encode(self):
        rows, cols, channels = self.image.shape
        self.estimator = np.empty(self.gray.shape)

        for row in range(1, rows):
            for col in range(1, rows):
                # Sum of vertical and horizontal until current point
                sum_hor = np.sum(self.gray[row, :col])
                sum_vert = np.sum(self.gray[:row, col])

                # Rounded difference bw current pixel and previous samples?
                self.estimator[row][col] = (int(round(2*self.gray[row][col]
                    - self.gray[row-1][col] - self.gray[row][col-1])))
                #print(self.estimator[row][col])
        return self.estimator
```

```python
    def decode(self):
        rows, cols= self.gray.shape
        self.decoded = np.empty(self.gray.shape)
        for row in range(1, rows):
            for col in range(1, cols):
                self.decoded[row][col] = (self.estimator[row][col] +
                        self.gray[row-1][col] + self.gray[row][col-1]) * 0.5
        return self.decoded
```

3.4 Run-Length Coding

```python
def encode(self):
    row, col = 0, 0
    decoded = []
    while row < self.rows:
        comp_row = []
        while col < self.cols:
            count = 0
            value = self.gray[row][col]

            while(self.gray[row][col] == value):
                col += 1
                count += 1

                if col == self.cols:
                    break
            comp_row.append(tuple((value, count)))
            decoded.append(comp_row)
            if col == self.cols:
                break
        row += 1
    return decoded
```

```python
def decode(self, encoded):

    decoded = np.empty(self.gray.shape)
    for ridx, row in enumerate(encoded):
        if ridx >= self.rows:
            break
        col_count = 0
        for pair in row:
            value = pair[0]
            length = pair[1]
            print(f'setting {value} in {length} spots.')
            for i in range(length):
                decoded[ridx][col_count] = value
                col_count += 1
    print(self.gray == decoded)
    return decoded
```