Image Compression Implementation

Project by: 1) Kunal Chaudhari (22B1060)
2) Viraansh Sontakkey (22B1250)
3) Neha Pedgaonkar (22B1805)

PART 1]

Objective: The objective of this task is to implement a basic image compression engine using principles similar to the JPEG algorithm to grayscale images. The algorithm should perform the following steps:

1. **Compute the 2D Discrete Cosine Transform (DCT) coefficients** of non-overlapping image patches.
2. **Quantize the DCT coefficients** based on a scaling factor, which is influenced by the quality factor.
3. **Apply Huffman coding** to compress the quantized DCT coefficients.
4. **Save the compressed data** in a specified format, allowing the compressed image to be decompressed and displayed accurately.
5. **Evaluate compression performance** by computing the Root Mean Squared Error (RMSE) and Bits Per Pixel (BPP) for multiple images with different quality factors.

Steps and Implementation Details:

1. Loading the Images:

The script (Img_compr_singleImg.py) works on a .png image and performs compression. The image is read using OpenCV's `cv2.imread` method with the `cv2.IMREAD_GRAYSCALE` flag to ensure it is treated as a grayscale image.

The input image is divided into non-overlapping 8x8 blocks, ensuring each block is independently processed. In cases where the image size is not divisible by 8, padding is added to ensure all blocks are complete.

{The script (Img_comp_fromFolder.py) loads all 20 grayscale images from a specified folder and processes them one by one.}

2. DCT Coefficients Computation:

For each image, the script performs a 2D Discrete Cosine Transform (DCT) on non-overlapping image patches. The DCT is an essential step in JPEG compression, converting the image data from the spatial domain to the frequency domain. The `cv2.dct` function can be utilized for this purpose, which calculates the DCT coefficients.

The DCT is applied to each block to convert the spatial domain data into frequency domain data. This step is critical as it allows for the compression of image data, focusing on significant frequencies and discarding less important ones.

3. Quantization:

The DCT coefficients are quantized using a quantization matrix scaled by a quality factor. The quality factor ranges from 10 to 100, where a higher quality factor results in less compression (and thus less distortion in the decompressed image). The quantization matrix is scaled according to the current quality factor, and any value below 1 is clipped to 1 to ensure valid quantization.

4. Huffman Coding:

After quantization, the DCT coefficients are compressed using Huffman coding. Huffman coding is a lossless data compression algorithm that replaces frequently occurring values with shorter codes and less frequent values with longer codes. The Huffman tree is generated and the quantized coefficients are encoded into a binary stream, which is written to a file for later decompression.

5. Saving Compressed Data:

The compressed data is saved in a binary file format. The script writes the following information to the file:

- Image dimensions (to be used for decompression)
- The number of blocks processed
- The scaled quantization matrix
- The encoded Huffman data

The binary data is saved in a `.bin` file, which contains all the necessary information for the decompression process.

6. Decompression:

The compressed binary data is read back, and the Huffman tree is decoded to retrieve the quantized DCT coefficients. These coefficients are then de-quantized using the original (scaled) quantization matrix. Finally, the 2D inverse DCT is applied to reconstruct the image.

7. Evaluation Metrics:

The performance of the compression algorithm is evaluated using two key metrics:

- **RMSE (Root Mean Squared Error):** The RMSE is calculated between the original and decompressed images to measure the quality of the compression. It indicates the accuracy of the compression. A lower RMSE indicates better reconstruction quality.

- **BPP (Bits Per Pixel):** The BPP is calculated by dividing the compressed file size (in bits) by the total number of pixels in the image. This metric measures the compression ratio: lower BPP values indicate higher compression efficiency.

8. Plotting RMSE vs BPP:

For each image, RMSE and BPP are calculated for multiple quality factors. The script plots these values to visualize the trade-off between compression efficiency (BPP) and image quality (RMSE). The plot is generated for all processed images, allowing a comparison of the compression performance across different images.

9. Output:

- The script outputs the RMSE and BPP for each quality factor, and these values are plotted on a graph.
- The decompressed images are saved in the same folder as the original images for comparison.

The results show how different quality factors impact both the image quality and the file size. Images with higher quality factors tend to have lower RMSE (better quality) but higher BPP (larger file sizes). Conversely, lower quality factors result in higher RMSE and smaller BPP.

Key Results:

For each image, the script prints the following values for each quality factor:

- **Quality Factor (QF):** Varies from 10 to 100.
- **RMSE:** Measures the quality of the compression by comparing the original and decompressed images.
- **BPP:** Measures the compression efficiency by comparing the file size to the number of pixels.

After processing all images, the script generates a plot that shows how RMSE and BPP vary with the quality factor for each image. A typical plot would show a curve where RMSE decreases and BPP increases as the quality factor increases.

Please refer to the script (Img_compr_singleImg.py) to see the detailed implementation of all these steps.

{The script (Img_comp_fromFolder.py) fulfills the same purpose but for 20 different images and plots RMSE V/s BPP of all 20 images (for 10 different quality factors for each) in a single plot (Plot_20.png).}

Conclusion:

This basic implementation of the JPEG compression algorithm performs the required steps of DCT computation, quantization, and Huffman coding. The evaluation of the compression quality and efficiency is done through the RMSE and BPP metrics. The plots generated provide a clear visual representation of the trade-offs between compression efficiency and image quality, helping to understand how different quality factors affect the performance of the compression algorithm.

PART 2]

Objective: To implement the paper on “Edge-Based Image Compression with Homogeneous Diffusion” from scratch.

Main Idea:

This repository implements an encoder-decoder system for image compression and decompression, based on edge detection, quantized pixel storage, and homogeneous diffusion. The whole procedure can be divided into a pipeline, which makes it easy to understand.

Pipeline:

Encoding:

1. Edge Detection:

- Extracts significant edges from the input image using Marr-Hildreth edge detection

2. Contour Pixel Extraction:

- Extracts pixel values from areas near edges and along image boundaries.
- Subsamples these values for efficient storage.

3. Quantization and Compression:

- Quantizes contour pixel values to 2^q levels. We also select every d-th pixel to minimize space required to store.
- The edge data is compressed using JBIG and quantized pixel values using PAQ

4. File Encoding:

- Combines header, JBIG data, and PAQ data into a single encoded file.

Decoding (Decompression):

1. File Decoding:

- Splits the encoded file into its components (header, JBIG data, and PAQ data).

2. Edge Reconstruction:

- Decodes the JBIG data to reconstruct the edge image.

3. Pixel Value Reconstruction:

- Decodes the PAQ data to retrieve quantized pixel values.
- Maps these values to their corresponding positions around edges.

4. Inpainting:

- Reconstructs missing regions using i dont know what diffusion to produce the final image.

Usage:

Encoder

Run the encoder to compress an image:

```
python encoder.py <image_path> <quantization_q> <sampling_d>
```

- `<image_path>`: Path to the input image (e.g., `input_image.jpg`).
- `<quantization_q>`: Quantization parameter (e.g., `4` for 242^424 levels).
- `<sampling_d>`: Sampling distance (e.g., `5`).

This will generate:

- `encoded_image_file.bin`: The compressed file in the main directory.. All of the supplement files will be redirected to the tmp folder.

Decoder

Run the decoder to decompress the encoded file:

```
python decoder.py <encoded_file_path>
```

- `<encoded_file_path>`: Path to the encoded file (e.g., `encoded_image_file.bin`).

This will generate:

- `final_image.png`: The reconstructed image in the main directory. All of the helper files will be redirected to the tmp folder.

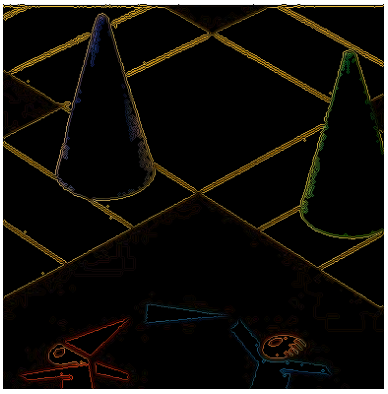
Requirements:

The following are central to the working of our model in addition to the required python libraries.

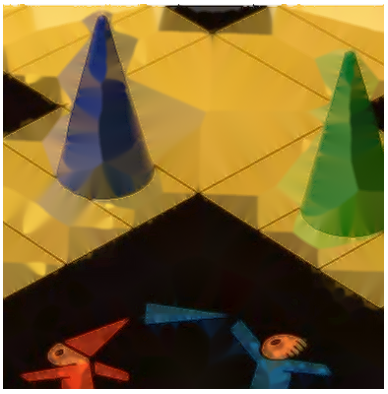
- jbigkit (for JBIG compression)
- paq8px (for PAQ compression)

Stages of Pipeline:

(a) Original Image



(b) Image with contour pixels



(c) Final Image

Bad aspects about approach:

We could not implement a perfect implementation of the paper albeit the method outlined being simplistic. Therefore the analysis of PSNR was not done, as visual inspection leads to noticeable defects :(Our diffusion is not implemented from scratch as outlined in the paper as well. The black portion is not being captured properly due to our lazily written code 😭

Also the main idea of the paper seems to only be working for images with cartoonish style, mainly due to less changes throughout. It is like demanding all the favourable

factors at once. Also the parameters have been eyeballed for edge detection, which makes it hard to show optimal performance for diverse sets of images.

Good aspects about the approach:

We could approach a compression rate of almost 43 times from the original png image to the one in the final image. We followed through all of the pipeline and made an organised step by step working encoder, all in one single consolidated file.

We feel we put a lot of effort into making it organised and as functional as possible. Final image is somewhat coherent haha 🙂