Unit- 4: Image Compression

1. Explain the need for image compression in multimedia applications. How does compression impact storage and transmission efficiency?

Answer:

Need for Image Compression in Multimedia Applications

Image compression is essential in multimedia applications to efficiently manage and transmit large volumes of visual data. Without compression, images—especially high-resolution ones—consume significant storage space and require high bandwidth for transmission, which impacts performance. Here are key reasons why image compression is needed:

1. **Storage Efficiency**
    - Multimedia systems handle a large number of images, videos, and graphics. High-resolution images, such as 4K or medical scans, can be several megabytes to gigabytes in size. Compression reduces file size, making it feasible to store more content on servers, hard drives, or mobile devices.
2. **Transmission Speed and Bandwidth Optimization**
    - In multimedia applications like video streaming, online galleries, or messaging apps, images must be transmitted quickly. Compressed images reduce the amount of data transmitted, improving loading times and reducing latency. This is especially critical in low-bandwidth environments or mobile networks.
3. **Cost Reduction**
    - Lower storage and bandwidth requirements mean reduced infrastructure costs. This benefits cloud storage providers, streaming services, and users by minimizing data transfer fees.
4. **User Experience**
    - Fast loading of images enhances user satisfaction. In multimedia apps like Instagram or Netflix, quick access to visual data is crucial. Delayed transmission due to uncompressed content can lead to poor user experience and reduced engagement.

Impact of Compression on Storage and Transmission Efficiency

Image compression directly improves the efficiency of both storage and transmission:

1. **Reduced Storage Requirements**
   - Compression techniques like JPEG, PNG, or WebP significantly decrease file sizes. For example, a 10 MB uncompressed image can be reduced to 1-2 MB with lossless compression or even smaller with lossy methods. This helps store more data within limited storage space.
2. **Faster Transmission Times**
   - Smaller file sizes require less time to upload or download. This is crucial in real-time applications like video conferencing, where lag needs to be minimized.
3. **Bandwidth Savings**
   - Compressed images consume less network bandwidth. This improves network efficiency, allowing more users to access content simultaneously without congestion.
4. **Trade-offs Between Quality and Compression**
   - Lossy compression methods (like JPEG) achieve higher compression ratios by removing some image details, which may slightly degrade quality. Lossless compression (like PNG) preserves quality but achieves less reduction in size. Depending on the application, the trade-off between quality and compression must be carefully managed.

---

2. What is redundancy? Explain three types of Redundancy.

Answer:
Redundancy refers to the presence of duplicate or unnecessary data in an image that can be removed without significantly affecting the image's quality. Removing redundancy allows for more efficient storage and transmission by compressing the image.

There are three primary types of redundancy in image compression:

1. Spatial Redundancy (Intra-frame Redundancy)

- **Definition**: It occurs when neighboring pixels in an image have similar or identical intensity values. Since adjacent pixels often carry redundant information, encoding each one separately is wasteful.
- **Example**: In a photo of the sky, many pixels may have nearly the same shade of blue.
- **Compression Technique**:
  - **Run-length encoding (RLE)**: Compresses sequences of identical pixels by recording the value and the number of repetitions.
  - **Predictive coding**: Encodes only the difference between adjacent pixels instead of individual pixel values.

2. Temporal Redundancy (Inter-frame Redundancy)

- **Definition**: This type of redundancy exists in sequences of images (like video frames) where consecutive frames are similar, with only minor changes between them.
- **Example**: In a video, the background of a scene often remains constant over many frames.
- **Compression Technique**:
  - **Motion compensation**: Encodes only the difference between frames rather than storing each frame independently.
  - **Video codecs** like H.264 use this to compress videos efficiently by transmitting changes between frames (delta frames).

3. Coding Redundancy

- **Definition**: Occurs when the data is represented with more bits than necessary for encoding the information. This results from inefficient encoding schemes where certain pixel values or patterns are assigned longer codes than needed.
- **Example**: An 8-bit grayscale image stores 256 levels of gray. However, if only a few shades dominate the image, encoding each level with 8 bits is redundant.

- **Compression Technique**:
  - **Huffman coding**: Assigns shorter codes to frequently occurring values and longer codes to less frequent ones.
  - **Arithmetic coding**: Represents sequences of symbols more efficiently based on their probabilities.

---

3. Define coding redundancy. Provide examples of how coding redundancy is used to reduce image file sizes.

Answer:

Coding redundancy occurs when data is represented using more bits than necessary, leading to inefficient encoding. In the context of image compression, certain pixel values or patterns may appear frequently, but if all values are encoded with the same number of bits (as in fixed-length encoding), the resulting representation is larger than needed. Coding redundancy can be reduced by assigning **shorter codes to more frequent patterns** and **longer codes to rare patterns**.

Examples of How Coding Redundancy is Reduced in Image Compression

1. **Huffman Coding**
   - **Concept**: Assigns variable-length binary codes to symbols (pixel values) based on their frequency of occurrence.
     - **Frequent symbols** get shorter codes.
     - **Rare symbols** get longer codes.
   - **Example**:
     In a grayscale image where the pixel value **"0"** appears 60% of the time and **"255"** only 1%, Huffman coding assigns a short code to **"0"** (e.g., **1**) and a longer one to **"255"** (e.g., **11111**). This reduces the average bit length required to encode the entire image.

Usage: JPEG compression employs Huffman coding during the entropy coding phase to reduce file size.

2. **Arithmetic Coding**
    - **Concept**: Rather than assigning fixed codes to individual symbols, arithmetic coding represents an entire sequence of symbols as a single decimal value between **0 and 1**, based on the probability distribution of the symbols.
    - **Example**:
      In an image, if **grayscale levels 50 and 51** occur with probabilities **0.6** and **0.4** respectively, arithmetic coding can generate a single value representing the whole sequence, instead of coding each pixel individually.

Usage: JPEG and JPEG2000 use arithmetic coding for further compression, especially for images with high redundancy.

3. **Run-Length Encoding (RLE)**
    - **Concept**: RLE exploits repeated sequences of identical symbols (common in images with large flat areas, like logos or drawings) by encoding the symbol and the number of repetitions.
    - **Example**:
      For a row of pixels:
      [255, 255, 255, 0, 0, 0, 0]
      Instead of encoding each value, RLE stores it as: **(255, 3), (0, 4)**.

Usage: PNG uses RLE to compress indexed images, and GIF uses it to compress animated sequences with repetitive content.

---

4. Discuss inter-pixel redundancy and how it is exploited in image compression algorithms. Provide examples of common methods to reduce inter-pixel redundancy.

Answer:
Inter-pixel redundancy occurs when neighboring pixels in an image have similar or identical intensity or color values. This redundancy arises because adjacent pixels in most natural images (such as photos of landscapes or portraits) often contain correlated information. Compression algorithms exploit this redundancy by encoding differences

between neighboring pixels instead of storing each pixel value independently.

How Inter-Pixel Redundancy is Exploited

By reducing inter-pixel redundancy, fewer bits are required to represent the image. Instead of storing each pixel's value, the **differences or patterns** between adjacent pixels are encoded. This process ensures that only changes (which are often smaller) are stored, leading to more efficient compression.

Common Methods to Reduce Inter-Pixel Redundancy

1. Predictive Coding

- **Concept**: Instead of storing the actual pixel values, the algorithm stores the difference between a pixel and its predicted value based on neighboring pixels. The predictor can use **previous pixels** (to the left or above) to guess the current pixel's value.
- **Example**:
  - If the predicted value for a pixel is 120 and the actual value is 125, the difference **(5)** is encoded.
- **Usage**: JPEG-LS (Lossless JPEG) uses predictive coding to encode the differences between adjacent pixel values.

2. Transform Coding (e.g., Discrete Cosine Transform - DCT)

- **Concept**: Transform coding converts the pixel values from the **spatial domain** (raw pixel values) to the **frequency domain**, where inter-pixel correlations appear as low-frequency components.
- **Example**:
  - In the JPEG algorithm, the image is divided into **8×8 blocks**. The DCT converts each block into a matrix representing different frequency components. Most of the visual information is concentrated in the **low-frequency coefficients**, while the high-frequency ones (representing minor details) can be discarded or quantized.

- **Usage**: JPEG leverages DCT to compress photographic images by reducing redundant frequency information.

3. Run-Length Encoding (RLE) on DCT Coefficients

- **Concept**: After applying a transform like DCT, many frequency components in a block become **0** or very small. These repeated zeroes can be efficiently compressed using **run-length encoding** (RLE).
- **Example**:
    - A sequence of DCT coefficients:
      [15, -3, 0, 0, 0, 7]
      Can be encoded as: (15, -3), (3 zeros), (7).
- **Usage**: JPEG uses RLE after quantizing the DCT coefficients to reduce inter-pixel redundancy.

4. Subsampling (for Color Images)

- **Concept**: In color images, human vision is less sensitive to small changes in color than in brightness. Thus, **chroma subsampling** reduces the resolution of color components (like chrominance) relative to the luminance (brightness) component.
- **Example**:
    - A **4:2:0** subsampling scheme reduces the color resolution by averaging color information across a 2×2 pixel block.
- **Usage**: JPEG uses chroma subsampling to reduce redundant color information between neighboring pixels, leading to smaller file sizes without significant quality loss.

5. Compare and contrast lossy and lossless image compression techniques. Provide examples of when each type of compression is more appropriate.

Answer:
Image compression techniques fall into two categories: **lossy** and **lossless**. Each has its unique approach to reducing file size and is suited for different use cases based on the need for quality and efficiency.

1. Lossy Image Compression

Overview

- **Lossy compression** reduces file size by **removing some data** that is considered less important, often exploiting limitations of human perception (like sensitivity to minor color changes).
- The reduction is irreversible, meaning the original image **cannot be fully reconstructed** after decompression.
- It achieves **higher compression ratios** than lossless techniques.

Examples

- **JPEG**: Uses techniques like the **Discrete Cosine Transform (DCT)** to discard high-frequency components that are less noticeable to the human eye.
- **WebP**: A modern lossy format that offers better compression ratios than JPEG.

Use Cases

- **When small file size is critical, and some quality loss is acceptable**:
    1. **Web images**: Faster loading times on websites (e.g., thumbnails, product photos).
    2. **Social media uploads**: Platforms compress images to save storage.
    3. **Streaming and video compression**: Video codecs like H.264 use lossy compression to reduce data size for efficient streaming.

Advantages

- High compression ratios, resulting in **smaller file sizes**.

- Useful for images where some quality loss is not noticeable.

Disadvantages

- Quality deteriorates with repeated compression (generation loss).
- Not suitable for images requiring precise reproduction, like medical images or graphics with text.

2. Lossless Image Compression

Overview

- **Lossless compression** reduces file size without discarding any information, allowing the **exact original image to be reconstructed** after decompression.
- It achieves **lower compression ratios** compared to lossy techniques but retains full data integrity.

Examples

- **PNG**: Uses **deflate compression** and supports transparency, often used for graphics and logos.
- **GIF**: Limited to 256 colors, used for simple animations and images with large flat color areas.
- **TIFF**: Common in professional photography and scanning, where lossless quality is critical.

Use Cases

- **When preserving image quality is crucial**:
    1. **Medical imaging**: Lossless compression ensures no diagnostic data is lost.
    2. **Graphic design and photography**: For intermediate workflows where images are edited multiple times.
    3. **Archiving**: Storing historical photos or artwork where original quality must be preserved.

Advantages

- **No loss in quality**, suitable for applications requiring exact reproduction.
- Can be used repeatedly without generation loss.

Disadvantages

- **Larger file sizes** compared to lossy compression.
- Less suitable for high-volume applications where storage and bandwidth are limited.

Comparison Table:

Aspect	Lossy Compression	Lossless Compression
Data Loss	Irreversible	None
File Size	Smaller	Larger
Compression Ratio	High	Lower
Image Quality	Some loss (depends on level)	Perfect reproduction
Suitable For	Web images, videos, social media	Medical images, archives, graphic design
Examples	JPEG, WebP	PNG, GIF, TIFF

---

6. Explain Compression Ratio with an Example. What other metrics helps in understanding the quality of the compression.

Amswer:

Compression ratio is a metric used to measure how effectively an image or file has been compressed. It is defined as the ratio of the **original file size** to the **compressed file size**.

Compression Ratio=Original Size / Compressed Size

A **higher compression ratio** means the file size has been reduced more effectively.

Example of Compression Ratio

- **Original Image Size**: 10 MB
- **Compressed Image Size**: 2 MB

Compression Ratio=10 MB / 2 MB = 5:1

This means the compressed file is 5 times smaller than the original.

Interpreting Compression Ratio

- **High compression ratio** (e.g., 10:1 or 20:1) indicates **greater size reduction**, but it might involve loss of quality (in lossy compression).
- **Low compression ratio** (e.g., 2:1) typically means the original quality is retained, as seen in lossless compression.

Other Metrics to Assess Compression Quality

1. **Peak Signal-to-Noise Ratio (PSNR)**
   - **Definition**: Measures the difference between the original and compressed images. A higher PSNR value indicates better quality and closer resemblance to the original image.
   - **Typical Range**:
     - Above 40 dB: Excellent quality
     - 30–40 dB: Good quality (acceptable loss for most purposes)
     - Below 30 dB: Noticeable degradation

2. **Mean Squared Error (MSE)**
   - **Definition**: Measures the average squared difference between the original and compressed images. Lower MSE means better quality.

3. **Structural Similarity Index (SSIM)**
   - **Definition**: Evaluates the perceived visual quality by comparing the structure, luminance, and contrast between two images. SSIM values range from **0 to 1**, with 1 indicating identical images.

    - **Usage**: More aligned with human visual perception than PSNR, making it a preferred metric in practical applications.

4. **Compression Time and Decompression Time**
    - **Definition**: Measures how long the algorithm takes to compress and decompress the image. Faster compression is desirable for real-time applications.

5. **Bits Per Pixel (BPP)**
    - **Definition**: Describes the average number of bits required to represent a pixel after compression. Lower BPP indicates better compression efficiency.

---

7. Identify Pros and Cons of the following algorithms.

I. Huffman coding

II. Arithmetic coding

III. LZW coding

IV. Transform coding

V. Run length coding

Answer:

Here are the pros and cons of various image compression algorithms, including Huffman coding, Arithmetic coding, LZW coding, Transform coding, and Run-Length coding.

I. Huffman Coding

Pros:

- **Efficiency**: Assigns shorter codes to frequently occurring symbols, leading to effective compression.
- **Simplicity**: The algorithm is relatively simple to implement.

- **Optimal for Fixed Frequency**: Produces optimal codes when the frequency of symbols is known and static.

Cons:

- **Static Nature**: Performance can degrade if symbol frequencies change significantly during data transmission (static vs. dynamic coding).
- **Overhead**: Requires additional storage for the Huffman tree, which can negate some compression benefits for small files.
- **Not Efficient for Small Datasets**: May not yield significant compression for files with fewer unique symbols.

II. Arithmetic Coding

Pros:

- **High Compression Ratios**: More efficient than Huffman coding for large datasets, as it can encode entire sequences of symbols rather than individual symbols.
- **Adaptive**: Can adapt to changing symbol probabilities, making it suitable for various types of data.
- **No Limit on Symbol Length**: Can encode long sequences without being restricted to fixed-length codes.

Cons:

- **Complexity**: More complex to implement than Huffman coding and can be computationally intensive.
- **Floating-Point Arithmetic**: Requires high precision in arithmetic calculations, which can lead to rounding errors.
- **Decompression Delay**: Requires the entire message for decoding, which can lead to delays in applications requiring real-time processing.

III. LZW Coding (Lempel-Ziv-Welch)

Pros:

- **No Prior Knowledge Required**: Works well without requiring prior knowledge of the frequency of symbols.

- **Simplicity**: The algorithm is straightforward and can be implemented easily.
- **Good for Repetitive Data**: Highly effective for data with many repeated sequences, such as graphics and simple images.

Cons:

- **Dictionary Size**: The dictionary can become large, leading to increased memory usage, especially for large files or very diverse data.
- **Decompression Speed**: Decompression can be slower than some other algorithms due to dictionary management.
- **Not Ideal for Small Data**: For small files, the overhead of dictionary storage can outweigh compression benefits.

IV. Transform Coding (e.g., DCT)

Pros:

- **Effective for Visual Data**: Works well for natural images, concentrating information in a few low-frequency components.
- **High Compression Ratios**: Achieves significant compression ratios while maintaining perceptual quality, especially in lossy formats (e.g., JPEG).
- **Compatibility**: Widely used and supported in various formats and applications.

Cons:

- **Loss of Information**: In lossy implementations, some original data is permanently lost, which can lead to quality degradation.
- **Blocking Artifacts**: In images, the 8x8 block division can cause visible artifacts, particularly at lower bitrates.
- **Complexity**: More complex to implement than basic coding algorithms due to the mathematics involved.

V. Run-Length Coding (RLE)

Pros:

- **Simplicity**: Very simple and easy to implement, making it suitable for basic applications.
- **Effective for Homogeneous Areas**: Works well for images with large areas of uniform color (like logos or icons).
- **Low Overhead**: Minimal additional data is needed for encoding.

Cons:

- **Ineffective for Complex Images**: Not suitable for photographs or images with high detail and variability, as it may not achieve significant compression.
- **Limited Compression**: Compression ratios may be low for data that lacks runs of repeated symbols.
- **Data Expansion**: Can lead to larger files when applied to data without many runs, such as detailed images.

---

8. Perform Huffman coding on a given set of pixel values. Show the step-by-step process and calculate the compression ratio achieved.

Answer:

Consider the following pixel values (grayscale intensities) with their **frequencies**:

Pixel Value	Frequency
0	10
1	15
2	30
3	16
4	29

Step-by-Step Process of Huffman Coding

Step 1: Create a Min-Heap with Frequencies

Each pixel value and its frequency will act as a node in the heap. Initially, every pixel is treated as a **leaf node**.

Heap:

- (10, 0)
- (15, 1)
- (16, 3)
- (29, 4)
- (30, 2)

Step 2: Build the Huffman Tree

We build the tree by **combining the two smallest nodes** repeatedly until only one node (the root) remains.

1. **Combine (10, 0) + (15, 1) = 25**
   - New heap: (16, 3), (29, 4), (30, 2), **(25, {0, 1})**
2. **Combine (16, 3) + 25 = 41**
   - New heap: (29, 4), (30, 2), **(41, {3, {0, 1}})**
3. **Combine (29, 4) + 30 = 59**
   - New heap: **(41, {3, {0, 1}}), (59, {4, 2})**
4. **Combine 41 + 59 = 100**
   - Final root node: **(100, {3, {0, 1}, {4, 2}})**

Step 3: Assign Binary Codes

Using the Huffman tree, assign **binary codes** starting with **0** for the left child and **1** for the right child.

- **Pixel 0**: 000
- **Pixel 1**: 001

- **Pixel 2**: 11
- **Pixel 3**: 01
- **Pixel 4**: 10

Step 4: Encode the Pixel Values

The original pixel values will now be replaced with their corresponding Huffman codes.
For example, a sequence like 0, 1, 2, 2, 3 would be encoded as:
000 001 11 11 01

Step 5: Calculate Original and Compressed Sizes

1. **Original Size**

- Assume each pixel value takes **8 bits** (standard grayscale value).

Total Bits (Original)=(10+15+30+16+29)×8=800 bits

2. **Compressed Size**

- Calculate the total compressed bits using the Huffman codes:

Total Bits (Compressed)=(10×3)+(15×3)+(30×2)+(16×2)+(29×2)

Total Bits (Compressed)=30+45+60+32+58=225 bits

Step 6: Calculate the Compression Ratio

Compression Ratio=Original Size/Compressed Size=800/225≈3.56:1

Conclusion

Using Huffman coding, the original pixel data (800 bits) is reduced to **225 bits**, achieving a **compression ratio of 3.56:1**. This demonstrates the effectiveness of Huffman coding in reducing the size of the image data without loss.

9. Explain the concept of arithmetic coding and how it differs from Huffman coding. Why is arithmetic coding considered more efficient in some cases?

Answer:

Arithmetic coding is an advanced entropy encoding technique used in data compression. Unlike Huffman coding, which assigns fixed-length codes to individual symbols, **arithmetic coding** encodes the entire message as a **single number** (a fractional value between 0 and 1) based on the probabilities of symbols. The more probable a symbol sequence, the smaller the interval representing it, leading to better compression.

How Arithmetic Coding Works (Step-by-Step Overview)

1. **Assign Probability Intervals:**
   Divide the interval [0,1) based on the probabilities of symbols in the message.

Example: For symbols {A, B, C} with probabilities:

   - A: 0.5 → Interval [0, 0.5)
   - B: 0.3 → Interval [0.5, 0.8)
   - C: 0.2 → Interval [0.8, 1)

2. **Narrow the Interval:**
   For a sequence like "AB", you first select the interval for "A" (i.e., [0, 0.5)) and then subdivide it based on the second symbol's probability.
   - "A" gives [0, 0.5)
   - For "B", the interval within [0, 0.5) becomes [0.25, 0.4) (narrowing based on B's probability).

3. **Resulting Interval:**
   The entire message "AB" is now represented by a number between **0.25 and 0.4**.
   This **single number** uniquely identifies the sequence.

4. **Decoding:**
   The decoding process works by reversing the above steps: using the encoded number to progressively identify each symbol based on the predefined intervals.

Differences Between Arithmetic Coding and Huffman Coding

Aspect	**Arithmetic Coding**	**Huffman Coding**
Encoding Unit	Encodes entire message as a **single value**	Encodes each symbol individually with fixed-length or variable-length codes
Handling of Probabilities	Handles fractional probabilities effectively	Requires probabilities rounded to powers of 2
Compression Efficiency	Can achieve closer to the **theoretical entropy limit**	May not always achieve optimal compression, especially with skewed distributions
Output Length	Produces **variable-length bitstreams**	Produces codes of fixed or variable length
Adaptiveness	Easily adapts to changing symbol distributions	Requires adjustments for dynamic symbol changes
Complexity	More complex, involving precise arithmetic	Simpler to implement with trees

Why Arithmetic Coding is More Efficient in Some Cases

1. **Better Compression for Skewed or Complex Probabilities**
   - Huffman coding works best when symbol probabilities are close to powers of 2 (e.g., 0.5, 0.25).
   - If symbol probabilities are non-uniform (e.g., 0.3, 0.2), arithmetic coding can provide more compact output by dividing intervals precisely.
2. **Encoding of Entire Messages**
   - Arithmetic coding encodes the entire message as a single value, which removes the **redundancy introduced by symbol-by-symbol encoding** in Huffman coding.
   - This results in higher compression efficiency for longer sequences.

3. **No Restriction on Symbol Length**
    - Huffman coding produces distinct codes for each symbol, limiting its efficiency. Arithmetic coding avoids these fixed boundaries and allows for **fractional bit-level precision**.

When to Use Arithmetic Coding

- **Video and Image Compression:** Arithmetic coding is used in **H.264** video and **JPEG2000** image formats, where high precision is required for compact data.
- **Adaptive Compression:** It's ideal for scenarios with **dynamic symbol probabilities**, such as text or speech encoding.
- **Applications Needing High Compression Ratios:** In situations where every bit matters (e.g., embedded systems), arithmetic coding achieves closer-to-optimal results than Huffman coding.

---

10. Provide an example of LZW coding on a simple sequence of image pixel values.

Answer:

The **LZW (Lempel-Ziv-Welch)** algorithm is a lossless compression method that builds a dictionary of previously seen sequences to compress the data. Let's walk through a simple example using a sequence of **image pixel values**.

Sequence of Pixel Values

Consider the following sequence of grayscale pixel values:
4, 4, 4, 5, 5, 4, 4, 4, 5

Step-by-Step LZW Coding Process

Step 1: Initialize the Dictionary

The dictionary starts with the **unique symbols** (pixel values) from the input sequence.

Code	**Value**
0	4
1	5

Step 2: Encoding the Sequence

We read the input sequence symbol by symbol, **adding new sequences to the dictionary** as we encounter them. The key idea is to find the **longest matching sequence** already in the dictionary, output its code, and add a new entry.

Encoding Steps:

1. **Start with the first symbol 4.**
   - Output: 0 (code for 4)
2. **Next symbol is another 4.**
   - Sequence so far: 4, 4
   - Not in the dictionary, so add it:

Code	**Value**
2	4, 4

   - Output the code for the first 4: 0
3. **Next symbol is 4.**
   - Sequence: 4
   - Already in the dictionary, continue reading.
4. **Next symbol is 5.**
   - Sequence: 4, 5
   - Not in the dictionary, so add it:

Code	**Value**
3	4, 5

   - Output the code for 4: 0
5. **Next symbol is 5.**

- Sequence: 5
- Already in the dictionary, continue reading.

6. **Next symbol is 4.**
   - Sequence: 5, 4
   - Not in the dictionary, so add it:

Code	**Value**
4	5, 4

   - Output the code for 5: 1
7. **Next symbol is 4.**
   - Sequence: 4
   - Already in the dictionary, continue reading.
8. **Next symbol is 4.**
   - Sequence: 4, 4
   - Already in the dictionary, continue reading.
9. **Next symbol is 5.**
   - Sequence: 4, 4, 5
   - Not in the dictionary, so add it:

Code	**Value**
5	4, 4, 5

   - Output the code for 4, 4: 2
10. **End of Sequence.**
    - Output the code for the last 5: 1

Final Encoded Output

The original sequence was:
4, 4, 4, 5, 5, 4, 4, 4, 5

The **LZW encoded output** is:
0, 0, 1, 2, 1

Step 3: Compression Achieved

1. **Original Size:**
   - Each pixel value takes **8 bits**.
   - Total bits for the original sequence = 9×8=729 \times 8 = 729×8=72 bits.
2. **Compressed Size:**
   - The encoded output has **5 codes**. Assuming each code takes **3 bits** (sufficient for small dictionaries):
     - Total compressed size = 5×3=155 \times 3 = 155×3=15 bits.

Compression Ratio

Compression Ratio=Original Size/Compressed Size=7215≈4.8:1

---

11. What is transform coding? Explain how it helps in compressing image data by reducing redundancies in the frequency domain.

Answer:

Transform coding is a technique used in image compression that leverages the **frequency domain** to reduce redundancies in image data. In transform coding, the original spatial data (pixel intensities in the image) are transformed into a different domain, typically the **frequency domain**, using mathematical transformations like the **Discrete Cosine Transform (DCT)** or **Discrete Fourier Transform (DFT)**. In this transformed domain, the image data is represented in terms of its frequency components rather than pixel intensities, making it easier to identify and eliminate less important information.

How Transform Coding Works

1. **Transformation to Frequency Domain**:
   - An image is divided into smaller blocks (e.g., 8x8 pixels for JPEG compression).

    - Each block undergoes a mathematical transformation (commonly DCT) that decomposes the spatial pixel values into **frequency coefficients**. These coefficients represent different frequency components in the block:
        - **Low-frequency components**: Contain most of the image's significant visual information (e.g., large, smooth areas).
        - **High-frequency components**: Represent rapid changes in the image, such as edges and fine details.
2. **Coefficient Quantization**:
    - After transformation, many of the high-frequency components tend to have small or zero values because smooth areas dominate most images.
    - These high-frequency coefficients are quantized or approximated to reduce their values significantly, often setting many to zero, which removes visually insignificant details.
    - The low-frequency coefficients are preserved more accurately because they contain essential image information.
3. **Encoding the Transformed Data**:
    - The remaining coefficients, now reduced in number, are encoded using techniques like **Huffman coding** or **Run-Length Encoding (RLE)** to compress the data further.

How Transform Coding Reduces Redundancies

- **Spatial Redundancy**: In natural images, neighboring pixels tend to have similar values, creating spatial redundancy. Transform coding leverages this by concentrating important image information in a few low-frequency coefficients, which can represent the smooth areas and essential details of an image.
- **Psychovisual Redundancy**: Human vision is more sensitive to low-frequency components than high-frequency components. By selectively reducing high-frequency details, transform coding exploits this redundancy to reduce data while maintaining perceived image quality.
- **Efficient Data Representation**: The transformed data has a more compact representation, where fewer coefficients carry the main image

information. This allows efficient compression by discarding or approximating insignificant data (high-frequency noise).

Example: JPEG Compression Using DCT

In JPEG compression:

- An 8x8 block of pixels undergoes a DCT transformation to produce an 8x8 matrix of frequency coefficients.
- The coefficients are then quantized, reducing precision based on how significant each coefficient is to image quality.
- The quantized coefficients are then encoded, achieving high compression rates by eliminating redundancies that would otherwise consume storage.

Advantages of Transform Coding in Image Compression

- **High Compression Ratios**: By removing spatial and psychovisual redundancies, transform coding can achieve significant compression ratios.
- **Perceived Quality Preservation**: By preserving essential low-frequency components and discarding high-frequency noise, transform coding maintains image quality in lossy formats like JPEG.
- **Broad Application**: Transform coding is fundamental to various image and video compression standards, including JPEG, MPEG, and H.264, due to its effectiveness and adaptability.

---

12. Discuss the significance of sub-image size selection and blocking in image compression. How do these factors impact compression efficiency and image quality?

Answer:

Sub-image size selection and blocking are crucial elements in image compression, especially in transform-based techniques like **JPEG compression**, where an image is divided into smaller blocks (typically **8x8** pixels) before applying transformations like the **Discrete Cosine Transform (DCT)**. The choice of block size and the process of dividing an image into smaller subimages (blocks) have a direct impact on **compression efficiency** and **image quality**.

Significance of Sub-Image Size and Blocking in Image Compression

1. **Localized Transformation and Redundancy Reduction**:
   - Smaller blocks allow compression algorithms to focus on localized areas of an image, where pixel values tend to be similar, particularly in regions of low detail. This enables the algorithm to reduce spatial redundancy efficiently within each block.
   - In transform coding, each block is transformed separately, concentrating most of the image's important visual information in a few coefficients. With smaller blocks, these coefficients represent localized regions more effectively, making compression more efficient.
2. **Handling of Image Details**:
   - In areas with significant detail or high-frequency changes (such as edges), smaller blocks can retain more detailed information. However, larger blocks tend to average out details, leading to loss of fine image structures.
   - Compression algorithms can thus balance the amount of data retained based on the degree of detail in each block. For example, smooth areas can be represented with fewer bits, while high-detail areas might need more data to preserve quality.
3. **Quantization and Blocking Artifacts**:
   - In transform-based compression (e.g., JPEG), quantization after transformation often leads to the **discarding of high-frequency components**, which helps achieve higher compression ratios. However, since each block is processed independently, abrupt changes between blocks can occur, resulting in visible **blocking artifacts** at lower bit rates.
   - Smaller blocks may reduce the visibility of these artifacts, but using too small a block size can lower the compression efficiency and introduce its own challenges, such as higher encoding time and reduced quality in highly compressed images.

Impact on Compression Efficiency and Image Quality

1. **Compression Efficiency**:
   - **Smaller Block Sizes**: Higher compression efficiency in areas of uniform color or texture, as redundant information can be more easily removed. However, more blocks mean that each block needs individual processing, which can increase overhead and reduce compression gains.
   - **Larger Block Sizes**: May allow higher compression ratios by averaging over larger areas, but can reduce adaptability to high-frequency details, leading to loss of detail in textured regions.
2. **Image Quality**:
   - **Smaller Block Sizes**: Better representation of fine details, with reduced blocking artifacts. However, if compressed heavily, more granular artifacts may appear across the image.
   - **Larger Block Sizes**: Can lead to noticeable blocking artifacts, especially at lower bit rates, as individual blocks become visible due to loss of coherence between adjacent blocks.

Examples of Block Size Selection in Compression Standards

- **JPEG Compression**: Uses 8x8 blocks as a compromise between quality and efficiency. This block size is small enough to manage localized image details while large enough to reduce blocking artifacts effectively at moderate bit rates.
- **H.264/AVC and HEVC Video Compression**: Employs variable block sizes for different image areas. Large blocks (e.g., 16x16) are used for smooth regions, while smaller blocks are used for high-detail areas, balancing compression efficiency and image quality dynamically.

13. Explain the process of implementing Discrete Cosine Transform (DCT) using Fast Fourier Transform (FFT). Why is DCT preferred in image compression?

Answer:

The **Discrete Cosine Transform (DCT)** is widely used in image compression due to its ability to efficiently represent image data in terms of its frequency components, allowing for effective compression by concentrating most of the signal's energy in a few coefficients. Implementing DCT directly can be computationally intensive, but it can be efficiently computed using the **Fast Fourier Transform (FFT)**, a technique that speeds up the transformation process significantly.

Implementing DCT Using FFT

The DCT is closely related to the **Discrete Fourier Transform (DFT)**, which is efficiently computed by the FFT algorithm. The DCT can be viewed as a **special case of the DFT** applied to real, even-symmetrical signals, which allows us to leverage the efficiency of FFT to compute DCT. Here's a simplified outline of how this can be achieved:

1. **Create Symmetry (Even Extension)**:
   - The DCT of a sequence can be computed by **extending the original signal to an even-symmetrical form**. For a sequence of length NNN, we construct an **even extension** of the signal by mirroring it. This manipulation allows the DFT to produce only cosine terms, which is the basis for the DCT.
2. **Apply FFT to the Extended Signal**:
   - The DCT can then be computed by applying the FFT to this symmetrically extended sequence, as the Fourier transform of an even-symmetric function produces only real values.
3. **Extract the DCT Coefficients**:
   - The FFT result is adjusted to obtain the **DCT coefficients** from the transformed data. Only the real part of the FFT is taken, as the imaginary part in this setup corresponds to sine terms, which are zero for an even signal.

This process allows us to compute the DCT quickly using an existing FFT implementation, reducing the complexity from O(N2)O(N^2)O(N2) to O(NlogN)O(N \log N)O(NlogN), where NNN is the number of data points.

Why DCT is Preferred in Image Compression

The DCT is preferred in image compression, especially in formats like **JPEG**, due to several key advantages:

1. **Energy Compaction**:
   - DCT is highly effective at **energy compaction**, meaning it concentrates most of the signal's energy in a few low-frequency components (coefficients). In an 8x8 pixel block, for example, the majority of the image information is stored in the top left corner (low-frequency components), allowing the high-frequency components (mostly noise and detail) to be discarded with minimal quality loss.
2. **Human Visual Perception Alignment**:
   - DCT aligns well with the characteristics of human vision, which is more sensitive to low-frequency information than to high-frequency details. By discarding high-frequency components, we can reduce file size while maintaining visual quality, as the loss is less perceptible to the human eye.
3. **Efficient Quantization**:
   - The low-frequency DCT coefficients are easier to quantize without significant image quality degradation. Higher-frequency coefficients, which contribute less to the visual quality, can be heavily quantized or even zeroed out to achieve higher compression rates. This selective quantization maximizes compression efficiency by prioritizing perceptually important information.
4. **Compression Standards Compatibility**:
   - The DCT is used in a wide range of image and video compression standards, such as **JPEG** and **MPEG**, making it widely compatible and well-suited for standardized digital image processing.

---

14. Describe how run-length coding is used in image compression, particularly for images with large areas of uniform color. Provide an example to illustrate your explanation.

Answer:

Run-Length Coding (RLC) is a simple yet effective compression technique used in image compression, especially beneficial for images with large areas of **uniform color** (low-frequency information). RLC reduces data size by representing consecutive, identical values (or "runs") as a single value and a count, which efficiently compresses sequences of repeated pixel values.

How Run-Length Coding Works in Image Compression

In images with large uniform areas, such as monochromatic backgrounds, the pixel values in these regions are often the same or change only slightly. Instead of storing each pixel individually, RLC stores:

1. The **value of the pixel** (or color).
2. The **number of times it repeats consecutively** (the "run length").

This process can be applied on a **row-by-row** or **column-by-column** basis to compress the image.

Example of Run-Length Coding

Suppose we have a grayscale image row with the following pixel values, where each value represents a pixel's intensity:

10, 10, 10, 10, 15, 15, 20, 20, 20, 20, 20, 10, 10

Without RLC, storing these pixel values requires storing each intensity individually. However, with RLC, we can compress this by recording the intensity and its run length as:

(10, 4), (15, 2), (20, 5), (10, 2)

Here:

- **10** repeats **4** times.
- **15** repeats **2** times.
- **20** repeats **5** times.
- **10** repeats **2** times again.

Instead of storing **13 pixel values**, RLC only stores **8 values** (4 pairs of intensity and run-length), achieving compression by reducing redundancy.

Advantages of Run-Length Coding for Images with Uniform Colors

- **Efficient Compression**: RLC is highly effective for compressing images with **large uniform regions**, as it significantly reduces storage needs by representing long runs of identical values with just two numbers.
- **Simple Implementation**: RLC is simple to implement and requires minimal processing, making it efficient for low-complexity systems.

Limitations of Run-Length Coding

- **Not Ideal for High-Frequency Images**: For images with high detail or frequent changes in color (such as photographs with textures), RLC may increase the data size, as run lengths will be short, reducing the effectiveness of compression.

Applications of Run-Length Coding

- RLC is widely used in compressing **bitmap images**, **fax transmissions**, **scanned documents**, and **black-and-white images**, where large regions often contain uniform color.

Conclusion

Run-Length Coding is particularly useful for images with large uniform areas, where it effectively reduces data size by storing runs of identical values compactly. However, for complex images with high variability, RLC's effectiveness is limited, and it is typically used alongside other compression methods (like DCT in JPEG) for enhanced results in such cases.