Image Compression and Coding

Chapter_5

Why we need image compression?

- One 90 minutes color movie, each second plays 24 frames. When we digitize it, each frame has 512×512 pixels, each pixel has three components R, G, B each one occupies 8 bits respectively, the total byte number is:

$$90 \times 60 \times 24 \times 3 \times 512 \times 512 = 97{,}200\text{MB}$$

- A CD may save 600 megabytes data, the movie needs 160 CDs to save.

Image Compression

- Image compression refers to the process of reducing the amount of data required to represent a digital image.
- It is achieved by removing redundancies and irrelevancies in image data to reduce storage requirements and facilitate faster transmission over networks.
- The goal is to preserve as much visual quality as possible while minimizing the file size.

- There are two main types of image compression:
- **Lossless Compression**: The original image can be perfectly reconstructed from the compressed image (e.g., Huffman Coding, LZW).
- **Lossy Compression**: Some information is lost in the process, but it often yields much higher compression ratios and is acceptable in many applications (e.g., JPEG, MPEG).

- Mathematically, compression seeks to encode image data with fewer bits by:
- Exploiting **statistical redundancy** (e.g., repeated pixel values)
- Exploiting **perceptual irrelevance** (e.g., discarding details invisible to the human eye)

- Compression can be characterized by:
- **Compression Ratio (CR)**: CR=Original Size/Compressed Size
- **Fidelity Measures**: Like PSNR or SSIM, which assess image quality degradation
- Examples:
- A 1024×1024 8-bit grayscale image requires 1 MB of storage.
- Using JPEG compression, the file can be reduced to ~50–100 KB with minimal quality loss.

Examples:

- A 1024×1024 8-bit grayscale image contains 1,048,576 pixels (1024 × 1024).
- Since each pixel in an 8-bit grayscale image requires 1 byte (8 bits), the total size is:
- 1024 * 1024 * 1 =1048756 – 1M
- Depending on the compression quality setting, the image file size can be reduced to approximately **50 KB to 100 KB**, achieving a **compression ratio (CR)** of:
- CR= 1048756/100000(for 100 KB)
- CR= 1048756/50000(for 50 KB)

- Image compression plays a vital role in:
  - Reducing storage costs in cloud and local systems
  - Enabling real-time image sharing and video streaming
  - Preserving bandwidth in communication networks
  - Optimizing image datasets for machine learning and AI

Need for Compression

- **Storage Efficiency**: Digital images consume large amounts of storage space. Compression helps in saving disk space.
- **Transmission Bandwidth**: Reducing file size leads to faster image transmission across networks.
- **Cost Reduction**: Reduced size lowers storage and communication costs.
- **Data Handling**: Easier to manage, transfer, and archive compressed images.

Lossy & Lossless Compression, Issues of Compression

Lossless Compression

- Lossless compression techniques preserve the original image data entirely. No information is lost during the process, so the original image can be perfectly reconstructed.
- **Examples**:
- Run Length Encoding (RLE)
- Huffman Coding
- LZW Coding

- **Examples**:
- Medical imaging
- Technical drawings
- Legal documents
- **Advantages**:
- Perfect reconstruction
- No degradation in quality
- **Disadvantages**:
- Lower compression ratios compared to lossy methods

Lossy Compression

- Lossy compression allows some loss of data to achieve much higher compression ratios. It removes less perceptible details that are not easily noticed by the human visual system.
- **Examples**:
- JPEG
- JPEG2000 (with lossy settings)
- MPEG (for video)

- **Example:**
- Web images
- Streaming videos
- Mobile image sharing
- **Advantages**:
- High compression ratio
- Reduced storage and bandwidth requirements
- **Disadvantages**:
- Irreversible loss of quality
- May introduce artifacts if over-compressed

Common Issues in Image Compression

1. **Trade-off Between Quality and Compression Ratio**:
   1. Higher compression reduces quality; finding the right balance is critical.
2. **Compression Artifacts**:
   1. Lossy compression may produce visual artifacts (e.g., blockiness in JPEG).
3. **Computational Complexity**:
   1. Some algorithms (e.g., Huffman, DCT) can be computationally intensive.

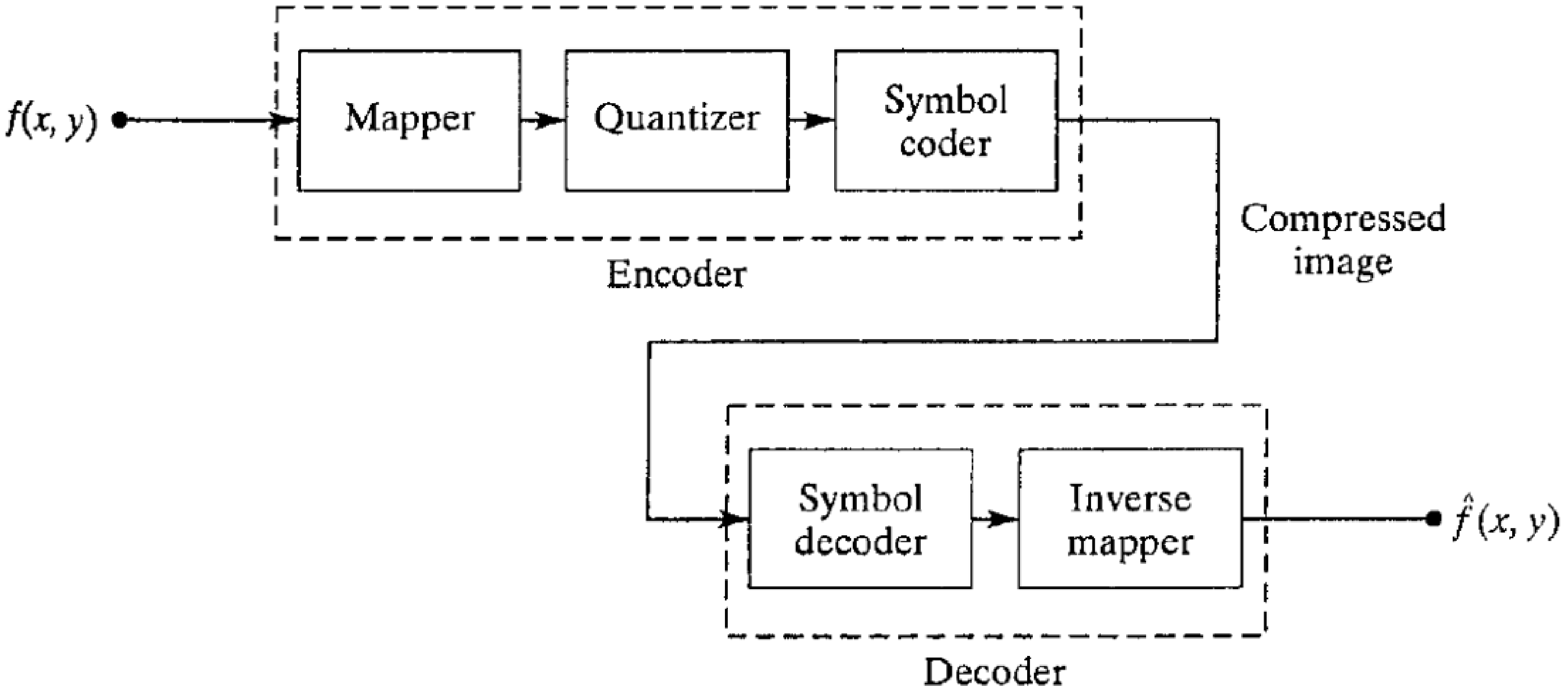
Common Issues in Image Compression

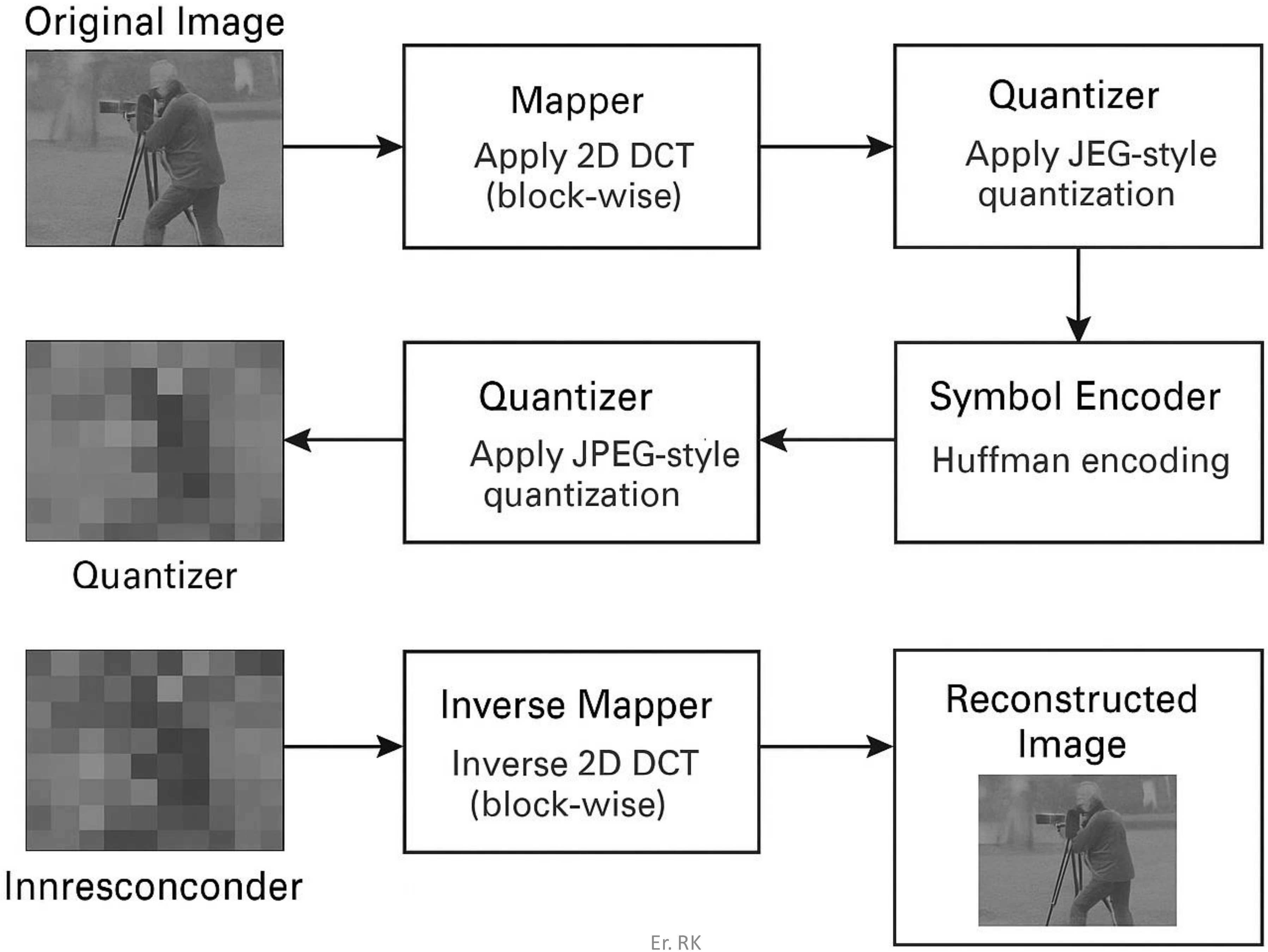
1. **Compatibility**:
    1. Not all formats support lossless compression (e.g., standard JPEG is lossy).
2. **Latency in Real-Time Systems**:
    1. In video streaming or live transmission, high compression may introduce delays.
3. **Perceptual Variability**:
    1. Different users may perceive quality differently, complicating standardization.

Image Compression System

- The process of compression involves two major components:
- **Encoder** (compressor): Transforms input data into a compact representation.
- **Decoder** (decompressor): Reconstructs the original (or an approximation of the original) from the compressed data.

Image Compression System




Original Image
Mapper
Apply 2D DCT
(block-wise)
Quantizer
Apply JEG-style
quantization
Symbol Encoder
Huffman encoding
Quantizer
Apply JPEG-style
quantization
Quantizer
Innresconconder
Inverse Mapper
Inverse 2D DCT
(block-wise)
Reconstructed
Image

A. Encoder

- **i. Mapper**
- **Function**: Transforms the image data into a different representation that is more compressible.
- **Example**:
  - Predictive coding
  - Transform coding (DCT, Wavelet)
- **Goal**: Reduce spatial or perceptual redundancies.

A. Encoder

- **ii. Quantizer *(Used in Lossy Compression Only)***
- **Function**: Reduces precision of mapped values.
- **Effect**: Introduces controlled loss to reduce data size.
- **Example**: JPEG quantization matrix applied to DCT coefficients.
- **Goal**: Exploit human visual system limitations by discarding imperceptible details.

A. Encoder

- **iii. Symbol Encoder**
- **Function**: Performs entropy coding.
- **Techniques**:
  - Huffman coding
  - Arithmetic coding
  - Run-Length Encoding (RLE)
- **Goal**: Remove statistical redundancy.

B. Decoder

- **i. Symbol Decoder**
- **Function**: Reverses entropy coding to recover quantized values.
- **ii. Inverse Mapper**
- **Function**: Applies inverse of the transformation used in the mapper.
- **Goal**: Reconstructs the original (or approximated) data from the transformed domain.

Element of Information Theory

- **Elements of Information Theory** refer to the foundational concepts used to **quantify and model information**, especially in the context of **data compression**, **communication**, and **coding**.
- These elements provide mathematical tools to measure how much "information" is in a message, and how efficiently it can be stored or transmitted.
- These concepts were introduced by **Claude Shannon** in his landmark 1948 paper, *A Mathematical Theory of Communication*.

1. Self-Information (Surprisal)

- **Self-Information**, also known as **Surprisal**, is a concept from **Information Theory** that measures how much **information** or **surprise** is associated with a specific event occurring.
- The **less probable** an event is, the **more surprising** it is — and therefore, the **more information** it carries.
- Self-information measures the **amount of surprise or information** associated with a single event occurring. It depends on how **unlikely** the event is:
- **More unlikely = More surprising = More information**

Mathematical Definition

Let x be a random event (e.g., a symbol in an image), and let $P(x)$ be the **probability** of occurrence of x.
Then the self-information of event x is defined as:

$$I(x) = -\log_b P(x)$$

Where:

- $I(x)$ is the self-information (in **bits** if $b = 2$, in **nats** if $b = e$)
- $P(x)$ is the probability of event x
- $\log_b$ is the logarithm to base b

> The base $b = 2$ is most common in digital systems, giving results in **bits**.

Intuitive Meaning

Probability P(x)	Self-Information I(x)	Interpretation
1	0 bits	No surprise — it's certain
0.5	1 bit	Moderate surprise
0.25	2 bits	Higher surprise
0.01	~6.64 bits	Very rare, high information

Use

- In image coding:
- Rare pixel values (like sharp transitions or edges) carry **more self-information**
- Frequent pixel values (like background regions) carry **less self-information**
- Compression schemes like **Huffman coding** or **Arithmetic coding** assign **shorter codes** to symbols with low self-information (high probability), and **longer codes** to those with high self-information (low probability)
  - This allows efficient representation of image data.

Units of Self-Information

🔷 1. In Bits (Base $b = 2$)

$$I(x) = -\log_2 P(x)$$

- This is the **most commonly used** form, especially in **digital systems**, compression, coding, and computing.
- The result is in **bits**, which represents the number of **binary yes/no decisions** needed to identify the event.

✔ **Example:**

If $P(x) = 0.25$, then:

$$I(x) = -\log_2(0.25) = 2 \text{ bits}$$

> It takes 2 binary decisions to identify that event.

🔷 2. In Nats (Base $b = e$)

$$I(x) = -\log_e P(x) = -\ln P(x)$$

- Unit is called a **nat**.
- Used in **information theory**, **thermodynamics**, **statistics**, and **machine learning** (especially when dealing with continuous variables and natural logarithms).
- 1 nat ≈ 1.4427 bits

✔ **Example:**

If $P(x) = 0.25$, then:

$$I(x) = -\ln(0.25) \approx 1.3863 \text{ nats}$$

🔷 3. In Hartleys (Base $b = 10$)

$$I(x) = -\log_{10} P(x)$$

- Less common.
- Used in early information theory, with the unit called a **Hartley**.

Comparing All Three Bases

Probability P(x)	In Bits (log2)	In Nats	In Hartleys (log_{10})
1	0	0	0
0.5	1 bit	0.693 nats	0.301 Hartleys
0.25	2 bits	1.386 nats	0.602 Hartleys
0.1	~3.32 bits	~2.302 nats	1 Hartley

When to Use Each:

Base	Use Case	Unit
2	Digital systems, binary compression (e.g., Huffman)	**Bits**
e	Continuous probabilities, ML loss functions (e.g., cross-entropy in softmax)	**Nats**
10	Older info theory, log-scale systems	**Hartleys**

2. Entropy (Average Information)

- Entropy is the **average self-information** across all symbols from a source — a measure of the **source's uncertainty** or **information content**.
- **Entropy** measures the **average amount of information (or uncertainty)** produced by a **random source of data**. It tells you how **unpredictable** or **informative** a message is, on average.

Mathematical Formula

For a discrete random variable X with outcomes $x_1, x_2, \ldots, x_n$ and corresponding probabilities $P(x_1), P(x_2), \ldots, P(x_n)$:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_b P(x_i)$$

Where:

- $H(X)$ = Entropy of source X
- $P(x_i)$ = Probability of symbol x_i
- b = Logarithm base:
  - Base 2 → entropy in **bits**
  - Base e → **nats**
  - Base 10 → **Hartleys**

Image Compression Case

- Suppose a grayscale image has pixel values with these probabilities:

Pixel Value	Probability
0	0.5
128	0.25
255	0.25

Then the entropy is:

$$H(X) = -[0.5 \log_2 0.5 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25] = 0.5 + 0.5 + 0.5 = 1.5 \text{ bits/symbol}$$

This means: on average, **1.5 bits are needed per pixel** if we compress this optimally.

Role of Entropy in Compression

Concept	Role
Entropy	The **lower bound** on number of bits needed to encode a message
Huffman Coding	Uses symbol probabilities to minimize code length close to entropy
Arithmetic Coding	Achieves compression **closer to entropy** than Huffman
JPEG, PNG, MPEG	All use entropy coding in final compression stage

Data Compression

•*Data compression* refers to the process of reducing the amount of data required to represent a given quantity of information

- *data* are the means where *information* is conveyed
- relative data redundancy R_D:

$$R_D = 1 - \frac{1}{C_R}$$

where C_R is the compression ratio:

$$C_R = \frac{n_1}{n_2},$$

n_1 and n_2 denote the number of information carrying units

Data Redundancy

- **Data redundancy** refers to the presence of **extra or duplicate information** in data that is **not necessary for understanding or reconstructing the content**.
- It contributes to **larger file sizes** without adding meaningful value.
- In the context of **image compression**, **removing redundancy** allows the same image to be represented with **fewer bits**, improving efficiency in **storage** and **transmission**.

Types of Redundancy

- Claude Shannon identified **three types** of redundancy in digital signals and images:

1. **Spatial Redundancy**

- Due to **correlation between neighboring pixels**.
- In natural images, adjacent pixels tend to have **similar intensity values**.
- Example: Sky or background in an image.
- **Compression techniques**: Predictive coding, Transform coding (e.g., DCT in JPEG)

- **2. Spectral Redundancy**
- Occurs in **color images** where the RGB channels carry overlapping information.
- The human eye is more sensitive to luminance (brightness) than chrominance (color), so color can be downsampled.
- **Compression techniques**: YCbCr color space in JPEG; Chroma subsampling

- **3. Psycho-visual Redundancy**
- Based on **limitations of human perception**.
- Some information can be removed **without noticeable quality loss**.
- Example: Discarding high-frequency DCT coefficients in JPEG.
- **Compression techniques**: Quantization, Perceptual weighting

Role of Redundancy in Compression

Redundancy Type	Eliminated By	Used In Compression
Spatial	Prediction, DCT, Wavelets	JPEG, JPEG2000
Coding	Huffman, Arithmetic coding	PNG, ZIP
Psycho-visual	Quantization	JPEG, MP3
Spectral	Chroma subsampling	JPEG, MPEG

Coding Redundancy

- **Coding Redundancy** arises when data is **not encoded in the most efficient way**, especially when:
- **Fixed-length codes** are used to represent symbols with **unequal probabilities**
- More **frequent symbols** are not given **shorter codes**
- It is the **redundancy introduced by inefficient symbol coding**, and it can be **eliminated** using optimal encoding techniques like **Huffman coding** or **Arithmetic coding**.

Coding Redundancy

- Assume a discrete random variable r_k in the interval [0,1] represents the grey levels of an image and that each r_k occurs with probability $p_r(r_k)$:

$$p_r(r_k) = \frac{n_k}{n} \quad k = 0,1,\ldots, L-1$$

and

$$L_{avg} = \sum_{k=0}^{L-1} l(r_k) p_r(r_k)$$

is the average number of bits used to represent each pixel, where $l(r_k)$ is the number of bits required for each value r_k

Coding Redundancy

r_k	$p_r(r_k)$	Code 1	$l_1(r_k)$	Code 2	$l_2(r_k)$
$r_0 = 0$	0.19	000	3	11	2
$r_1 = 1/7$	0.25	001	3	01	2
$r_2 = 2/7$	0.21	010	3	10	2
$r_3 = 3/7$	0.16	011	3	001	3
$r_4 = 4/7$	0.08	100	3	0001	4
$r_5 = 5/7$	0.06	101	3	00001	5
$r_6 = 6/7$	0.03	110	3	000001	6
$r_7 = 1$	0.02	111	3	000000	6

TABLE 8.1 Example of variable-length coding.

$$L_{avg} = \sum_{k=0}^{7} l_2(r_k) p_r(r_k)$$

$$= 2(0.19) + 2(0.25) + 2(0.21) + 3(0.16) + 4(0.08)$$

$$+ 5(0.06) + 6(0.03) + 6(0.02)$$

$$= 2.7 \text{ bits},$$

$$R_D = 1 - \frac{1}{1.11} = 0.099$$

Coding Redundancy

r_k	$p_r(r_k)$	Code 1	$l_1(r_k)$	Code 2	$l_2(r_k)$
$r_0 = 0$	0.19	000	3	11	2
$r_1 = 1/7$	0.25	001	3	01	2
$r_2 = 2/7$	0.21	010	3	10	2
$r_3 = 3/7$	0.16	011	3	001	3
$r_4 = 4/7$	0.08	100	3	0001	4
$r_5 = 5/7$	0.06	101	3	00001	5
$r_6 = 6/7$	0.03	110	3	000001	6
$r_7 = 1$	0.02	111	3	000000	6

L_{avg} 3 bits/symbol

L_{avg} 2.7 bits/symbol

Coding Redundancy

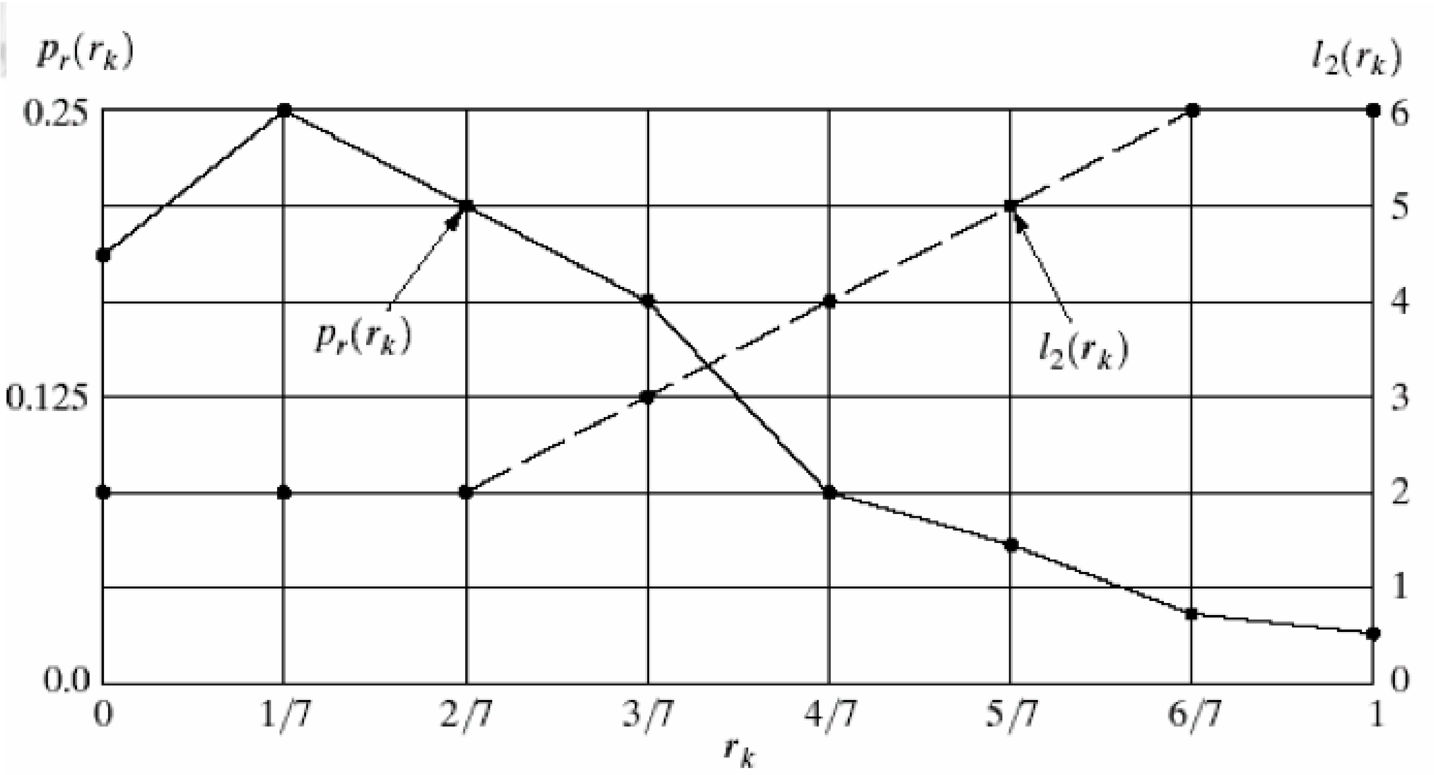


FIGURE 8.1 Graphic representation of the fundamental basis of data compression through variable-length coding.

Concept: assign the longest code word to the symbol with the least probability of occurrence.

Huffman coding

Huffman code: Consider a 6 symbol source

	a_1	a_2	a_3	a_4	a_5	a_6
$p(a_i)$	0.1	0.4	0.06	0.1	0.04	0.3

Huffman coding

- Huffman coding: give the smallest possible number of code symbols per source symbols.

Step 1: Source reduction

Original source		Source reduction			
Symbol	Probability	1	2	3	4
a_2	0.4	0.4	0.4	0.4	0.6
a_6	0.3	0.3	0.3	0.3	0.4
a_1	0.1	0.1	0.2	0.3	
a_4	0.1	0.1	0.1		
a_3	0.06	0.1			
a_5	0.04				

Huffman coding

Step 2: Code assignment procedure

Original source			Source reduction							
Sym.	Prob.	Code	1		2		3		4	
a_2	0.4	1	0.4	1	0.4	1	0.4	1	0.6	0
a_6	0.3	00	0.3	00	0.3	00	0.3	00	0.4	1
a_1	0.1	011	0.1	011	0.2	010	0.3	01		
a_4	0.1	0100	0.1	0100	0.1	011				
a_3	0.06	01010	0.1	0101						
a_5	0.04	01011								

The code is instantaneous uniquely decodable without referencing succeeding symbols.

Average length:

(0.4) (1) + 0.3 (2) + 0.1 X 3 + 0.1 X 4 + (0.06 + 0.04) 5 = 2.2 bits/symbol

(Imp)

Entropy Coding: (Replace input string by code word)

Entropy ~~codes~~ encodes coding the given set of symbols with minimum number of bits required to represent them.

The most important entropy coding is Huffman Coding. The theoritical minimum average of bits that are required to transmit a particular source string is known as entropy of source and it can be computed by using following formula:

$$\boxed{\text{Entropy } (H) = -\sum_{i=1}^{N} P_i \log_2 P_i}$$

Huffman Coding Algorithm:

(1) Arrange the symbol probabilites 'Pi' in a decreasing order and consider them as leaf node of a tree.

(2) While there is more than one node;

(a) Manage the 2 nodes with smallest probability to form a new node.

(b) Assign '1' & '0' to each pair of branches merging into a node.

(3) Read sequentically from root node to leaf node.

(Imp) Q Source generates the symbol S_1, S_2, S_3, S_4 and S_5 randomly with probability $P_1 = 0.4$, $P_2 = 0.2$, $P_3 = 0.2$, $P_4 = 0.1$ and $P_5 = 0.1$ respectively.

Generate the code word for each symbol using Huffman coding. Also, calculate the compression ratio and efficiency of the system.

step(1):

soln:

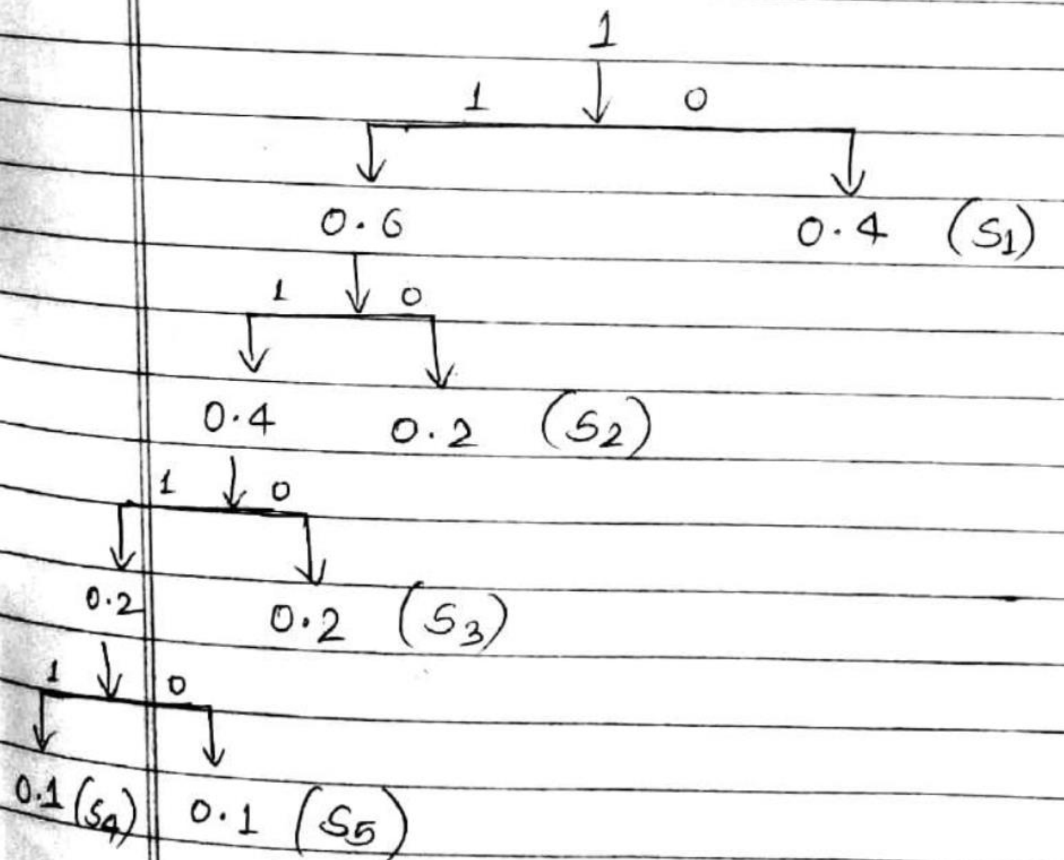
Symbol	Original size Probability (P_i)	Step 1	Step 2	Step 3	Step 4
S_1	0.4	0.4	0.4	0.6	1
S_2	0.2	0.2	0.4	0.4	
S_3	0.2	0.2	0.2		
S_4	0.1	0.2			
S_5	0.1				

step(2):

Step(2):

Construct a Huffman Tree:

Assume, Highest probability 0.6 = 1
Lowest probability 0.4 = 0



Step(3): Generate code Word length:

Symbol	Probability(P_i)	Code length(l_i)
S_1	0.4	0 = 1
S_2	0.2	10 = 2
S_3	0.2	110 = 3
S_4	0.1	1111 = 4
S_5	0.1	1110 = 4

Now

Now,

$$\text{Compression Ratio (or } L_{avg}) = \sum_{i=1}^{N} P_i \cdot l_i$$

$$= 0.4 \times 1 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 4 + 0.1 \times 4$$

$$= 2.2 \text{ bits / symbol}$$

$$\therefore \text{Entropy } (H) = -\sum_{i=1}^{N} P_i \log_2 P_i$$

$$= -[0.4 \times \log_2 0.4 + 0.2 \times \log_2 0.2 + 0.2 \times \log_2 0.2 + 0.1 \times \log_2 0.1 + 0.1 \times \log_2 0.1]$$

$$= 2.12$$

Hence, $\text{Efficiency} = \dfrac{\text{Entropy}}{L_{avg}} \times 100$

$$= \frac{2.12}{2.2} \times 100\%$$

$$= 96.36\%$$

Q Calculate the compression ratio and efficiency from below image:

Gray level (r)	0	1	2	3	4	5	6	7
No. of Pixel (n_k)	400	1350	659	2034	816	2560	250	1500

Run Length Coding

Run-L

- **Run-Length Encoding** is a **lossless data compression** method that replaces **sequences of repeated symbols (runs)** with a **single symbol and a count**.
  - **Key Idea**: Instead of storing each repeated symbol individually, store just **one copy** and the **number of repetitions**.
- **How It Works**
  - **Input:**
  - A sequence of **repeating** symbols (characters, numbers, or pixel values).
  - **Output:**
  - A sequence of **(value, count)** pairs.

Original Image

Pixel Intensity Matrix

0	0	0	0	0	255	255	255
0	0	0	0	0	255	255	255
0	0	0	0	0	255	255	255
255	255	255	0	0	0	0	0

Run Length Encoded Data

```
RLE Output:
(0, 5)
(255, 3)
(0, 5)
(255, 3)
(0, 5)
(255, 6)
(0, 5)
```

Run-Length Encoding (RLE)

- **Run-Length Encoding** is a **lossless data compression** method that replaces **sequences of repeated symbols (runs)** with a **single symbol and a count**.
  - **Key Idea**: Instead of storing each repeated symbol individually, store just **one copy** and the **number of repetitions**.
- **How It Works**
  - **Input:**
  - A sequence of **repeating** symbols (characters, numbers, or pixel values).
  - **Output:**
  - A sequence of **(value, count)** pairs.

Example 1: Simple Text

- Input:
- AAAAABBBCCDAA
- Output
- (‘A’,5), (‘B’,3), (‘C’,2), (‘D’,1), (‘A’,2)

Example 2: 1D Binary Image Row

- Input
- [1 1 1 1 0 0 0 1 1 0 0]
- RLE Output:
- (1,4), (0,3), (1,2), (0,2)

Example 3: 2D Grayscale Image (MATLAB Style)

- Suppose we have an image row:
- [100 100 100 200 200 50 50 50 50]
- RLE Output:
- (100,3), (200,2), (50,4)

RUN LENGTH CODING:

Simplest Data Compression technique.

In this runs of data are stored as a single data value and count rather than original run.

run → Sequence of Same Symbol/data Value.

BBBBBBBB → B 8

Long Sequence of 'B'

B — data Value

8 → (Eight)

8 → Count of data Value

Eg. 1. BBBBBBBBB AAAAA N GG MMM

20. [09 05 01 02 03

↓ RLE

15. B09A05N01G02M03

More efficient

Eg.2 00000000000000 | 0000 | | 000000000000

14 4 0 12

RLE

1110 0100 0000 1100

Applications of RLE

Domain	Application Example
Image compression	Fax machines, BMP, TIFF, PCX files
Video compression	Background scenes with little movement
Data transmission	Repeating headers, padding removal
Font storage	Bitmap fonts, where characters have repeating pixels

LZW Coding

- LZW (Lempel-Ziv-Welch) is a **lossless data compression algorithm** widely used in formats like **GIF, TIFF**, and **PDF**.
  - It compresses data by **replacing strings of characters with shorter codes**.
  - Works **without prior knowledge** of symbol frequencies.
  - Builds a **dictionary dynamically** during compression and decompression.

Key Features

Feature	Description
Type	Lossless Compression
Algorithm Family	Dictionary-based
Advantage	Fast, efficient for text and image data
Use cases	GIF images, TIFF, UNIX compress utility

Working Mechanism (Compression Algorithm)

- **Step-by-step:**

1. **Initialize the dictionary** with all individual characters (e.g., A-Z or ASCII 0–255).
2. Read characters one by one to form the longest match w that exists in the dictionary.
3. Output the **code for w**.
4. Add w + k (next character) to the dictionary.
5. Repeat until all input is processed.

- **Advantages of LZW**
- Efficient for **repetitive data**
- No need to transmit the dictionary
- Fast **encoding/decoding**
- **Limitations**
- Not ideal for **random or very diverse** data
- Dictionary can **grow large** (handled with max size or reset)

Error-Free Compression (Lossless)

- Error-free compression means lossless compression
- In numerous applications error-free compression is the only acceptable means of data reduction.
- They normally provide compression ratios of 2 to 10.

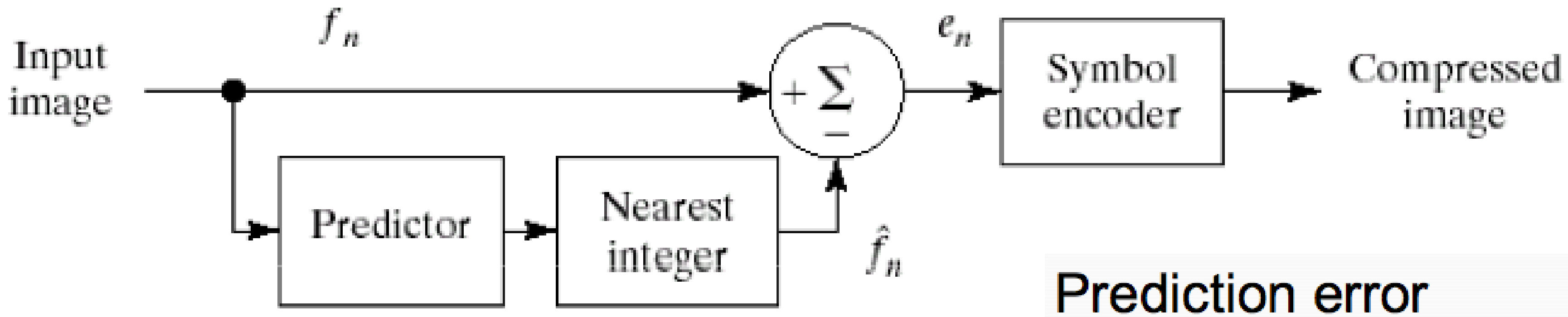
Lossless Predictive Coding Model

Lossless Predictive Coding is a compression technique that:

- Predicts the value of a pixel using neighboring pixels.
- Calculates the **difference (error)** between the actual and predicted value.
- Encodes this difference instead of the original pixel.
- During decoding, the original pixel is **reconstructed exactly** using the prediction and the error.

Because we only store the **prediction error**, which often has **lower entropy**, we can compress it more efficiently using entropy coding (like Huffman or arithmetic coding).

Lossless Predictive Coding Model



Prediction error

$$e_n = f_n - \hat{f}_n$$

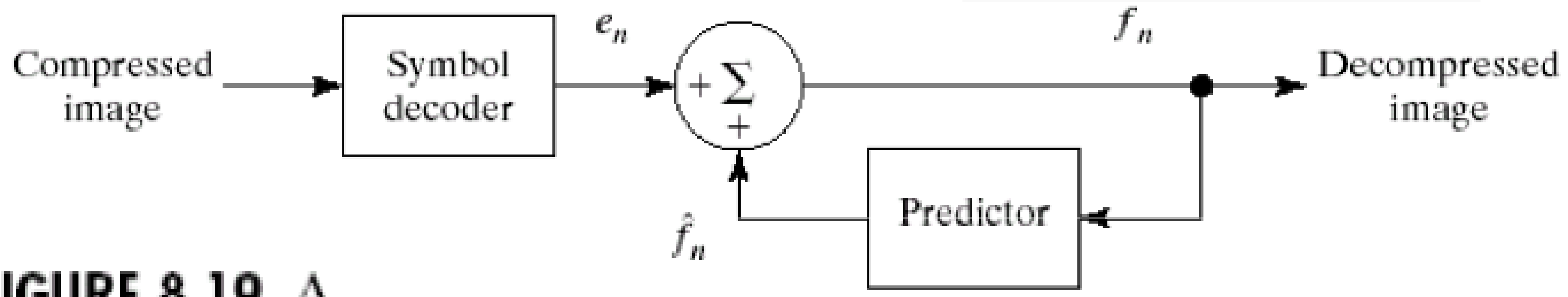


FIGURE 8.19 A lossless predictive coding model: (a) encoder; (b) decoder.

Prediction error:

$e_n = f_n - \hat{f}_n$

e_n is coded using a variable length code

$f_n = e_n + \hat{f}_n$

Lossless Predictive Coding Model

Encoder Model – Step-by-Step

Let:

- f_n: Actual pixel value at position *n*
- $\hat{f}_n$: Predicted value of the pixel
- e_n: Prediction error $e_n = f_n - \hat{f}_n$

Encoding Process:

1. **Prediction:**
   - Use nearby pixels to predict the current pixel $\hat{f}_n$
   - E.g., for pixel (i, j): $\hat{f}_n = f(i, j-1)$ or an average of neighbors
2. **Compute Error:**
   - $e_n = f_n - \hat{f}_n$
   - This error is usually small and centered around 0

3. **Quantization**:
   Round to nearest integer if needed

4. **Entropy Coding**:
   Encode the error values using a variable-length code (e.g., Huffman)

Decoder Model – Step-by-Step

1. **Decode Error**:
   - Receive e_n from entropy decoder
2. **Prediction**:
   - Use same predictor as in encoding to compute $\hat{f}_n$
3. **Reconstruction**:
   - $f_n = e_n + \hat{f}_n$
   - Get the original pixel value exactly

Assume a 1D row of pixels:

$$f = [100, 102, 105, 108, 110]$$

Step 1: Use previous pixel as predictor

Pixel Index	Actual f_n	Predictor $\hat{f}_n$	Error $e_n = f_n - \hat{f}_n$
1	100	0 (initial)	100
2	102	100	2
3	105	102	3
4	108	105	3
5	110	108	2

So instead of storing `[100, 102, 105, 108, 110]`, we store the **prediction errors** `[100, 2, 3, 3, 2]`.

Since the **error values are smaller and more repetitive**, they require fewer bits and can be compressed better with entropy encoding.

Reconstruction (Decoder Side):

Start with first pixel as-is: 100

Use predictor + error:

Step	Prediction	Error	Reconstructed Pixel
1	-	100	100
2	100	2	102
3	102	3	105
4	105	3	108
5	108	2	110

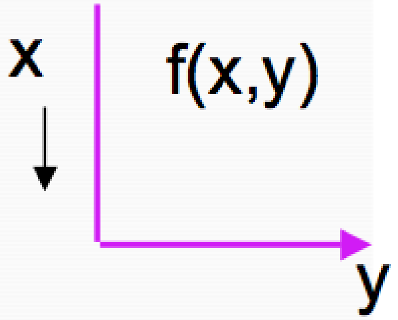
✔ **Perfect reconstruction. No data loss.**

Lossless Predictive Coding Model

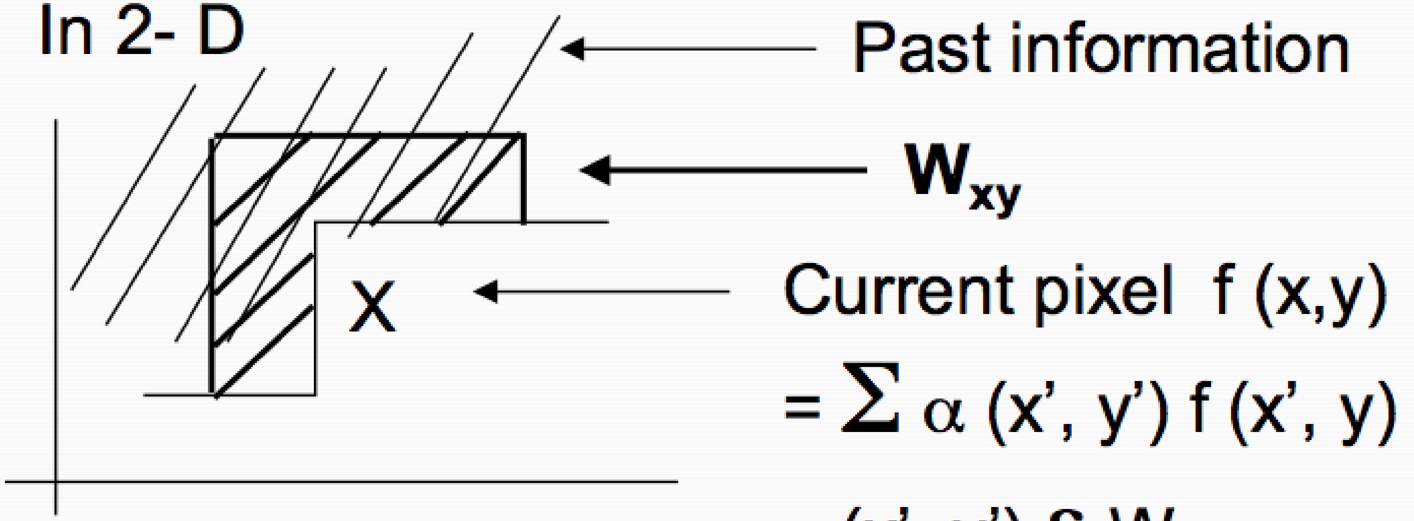
Example 1: $\hat{f}_n = \text{Int}\left(\sum_{i=1}^{m} \alpha_i f_{n-i}\right)$

→ Linear predictor ; m = order of predictor

Example 2: $\hat{f}_n(x,y) = \text{Int}\left(\sum_{i=1}^{m} \alpha_i f(x, y-i)\right)$



In 2- D



$= \sum \alpha(x', y') f(x', y)$

$(x', y') \in W_{xy}$

Lossy Compression

- Unlike the error-free approaches, lossy encoding is based on the concept of compromising the accuracy of the reconstructed image in exchange for increased compression.

Lossy Compression

Lossy Predictive Coding is an image compression technique that:

- **Predicts** the value of a pixel based on its neighbors (like in lossless predictive coding),
- **Computes the prediction error**, but instead of preserving it exactly,
- **Quantizes** the error (introducing some loss),
- Then **encodes** it using entropy coding.

The **goal** is to significantly reduce the number of bits required to represent an image, by allowing a controlled amount of distortion.

Let:

- f_n: Original pixel value
- $\hat{f}_n$: Predicted pixel value
- $e_n = f_n - \hat{f}_n$: Prediction error
- $Q(e_n)$: Quantized prediction error

➤ Step-by-Step:

1. **Prediction:**
   - Estimate $\hat{f}_n$ using previous pixels (e.g., left, above, diagonal)
   - Common: $\hat{f}_n = f(i, j-1)$, or average of neighbors
2. **Error Calculation:**
   - $e_n = f_n - \hat{f}_n$
3. **Quantization (Lossy Step):**
   - $Q(e_n) = \text{round}(e_n/\Delta) \times \Delta$
   - Where Δ is the quantization step size
   - Introduces irreversible loss of data
4. **Entropy Encoding:**
   - Compress the quantized error values using variable-length codes

1. **Entropy Decoding**:
    - Retrieve quantized error values $Q(e_n)$
2. **Prediction**:
    - Recompute predictor $\hat{f}_n$ using already decoded pixels
3. **Reconstruction**:
    - $\tilde{f}_n = \hat{f}_n + Q(e_n)$
    - Output is **not identical** to original f_n, but **close**

Suppose:

- $f = [100, 103, 105, 107]$
- Use left pixel as predictor
- Quantization step $\Delta = 2$

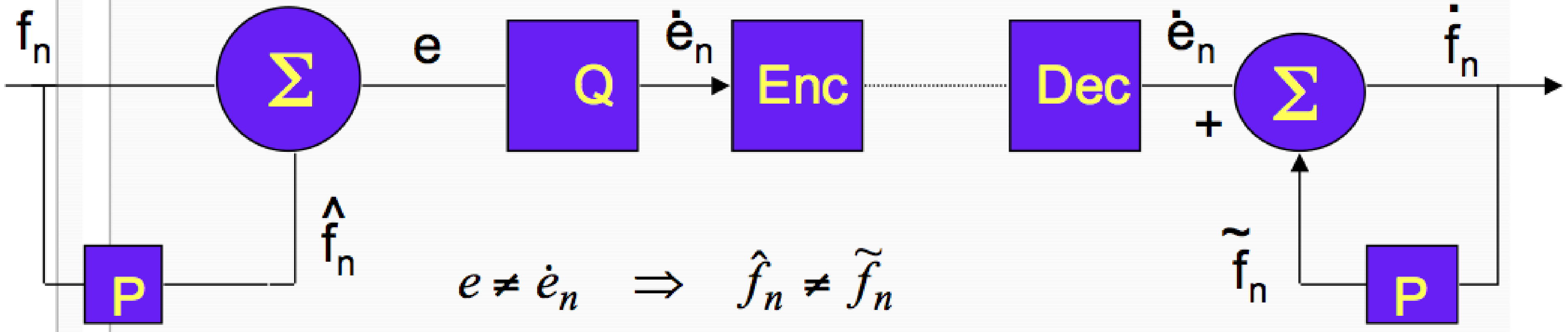
Pixel Index	f_n	$\hat{f}_n$	e_n	$Q(e_n)$	Stored	Reconstructed $\tilde{f}_n$
1	100	0	100	100	100	100
2	103	100	3	4	4	104
3	105	104	1	0	0	104
4	107	104	3	4	4	108

Notice:

- The reconstructed values differ slightly from the original.
- This **small loss** allows better compression, especially after entropy encoding.

Lossy Compression

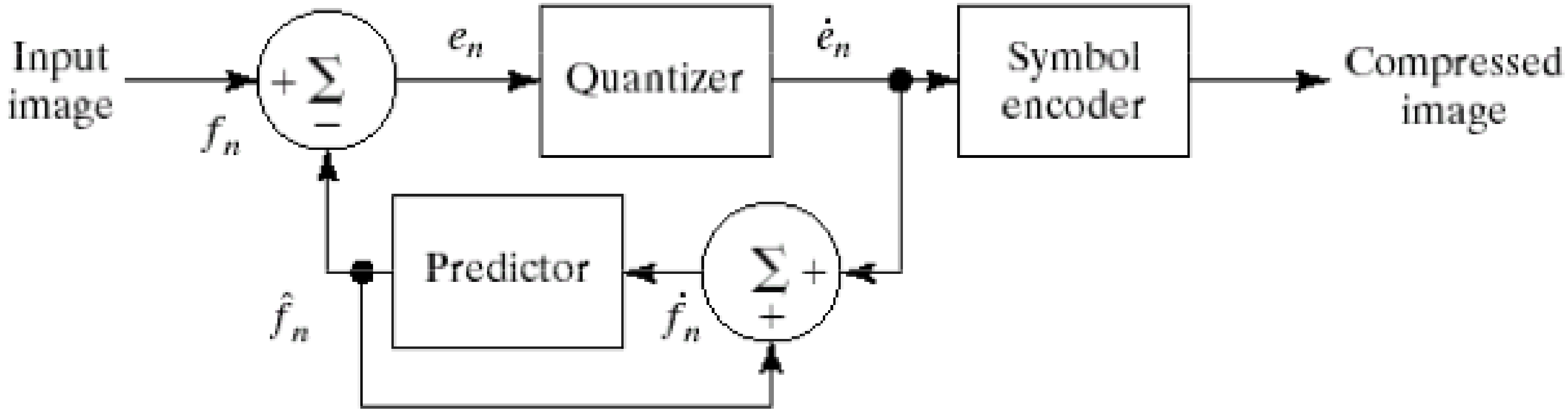
Lossy compression: uses a quantizer to compress further the number of bits required to encode the 'error'.
First consider this:

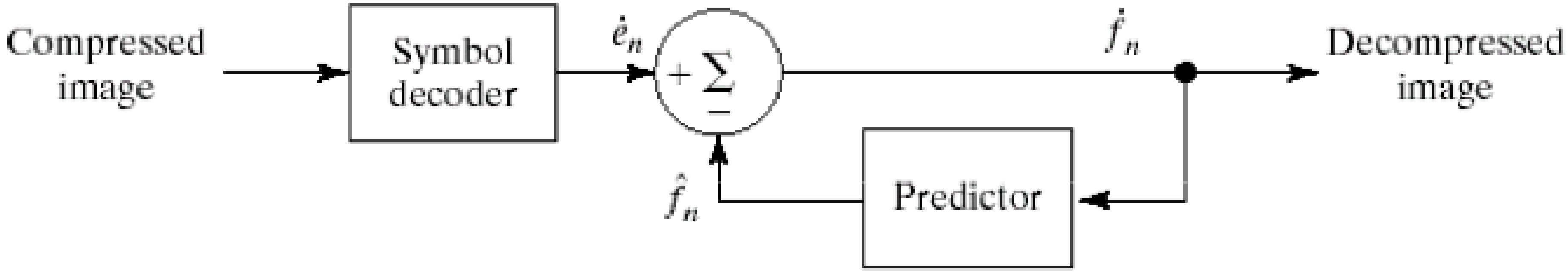


$$e \neq \dot{e}_n \quad \Rightarrow \quad \hat{f}_n \neq \tilde{f}_n$$

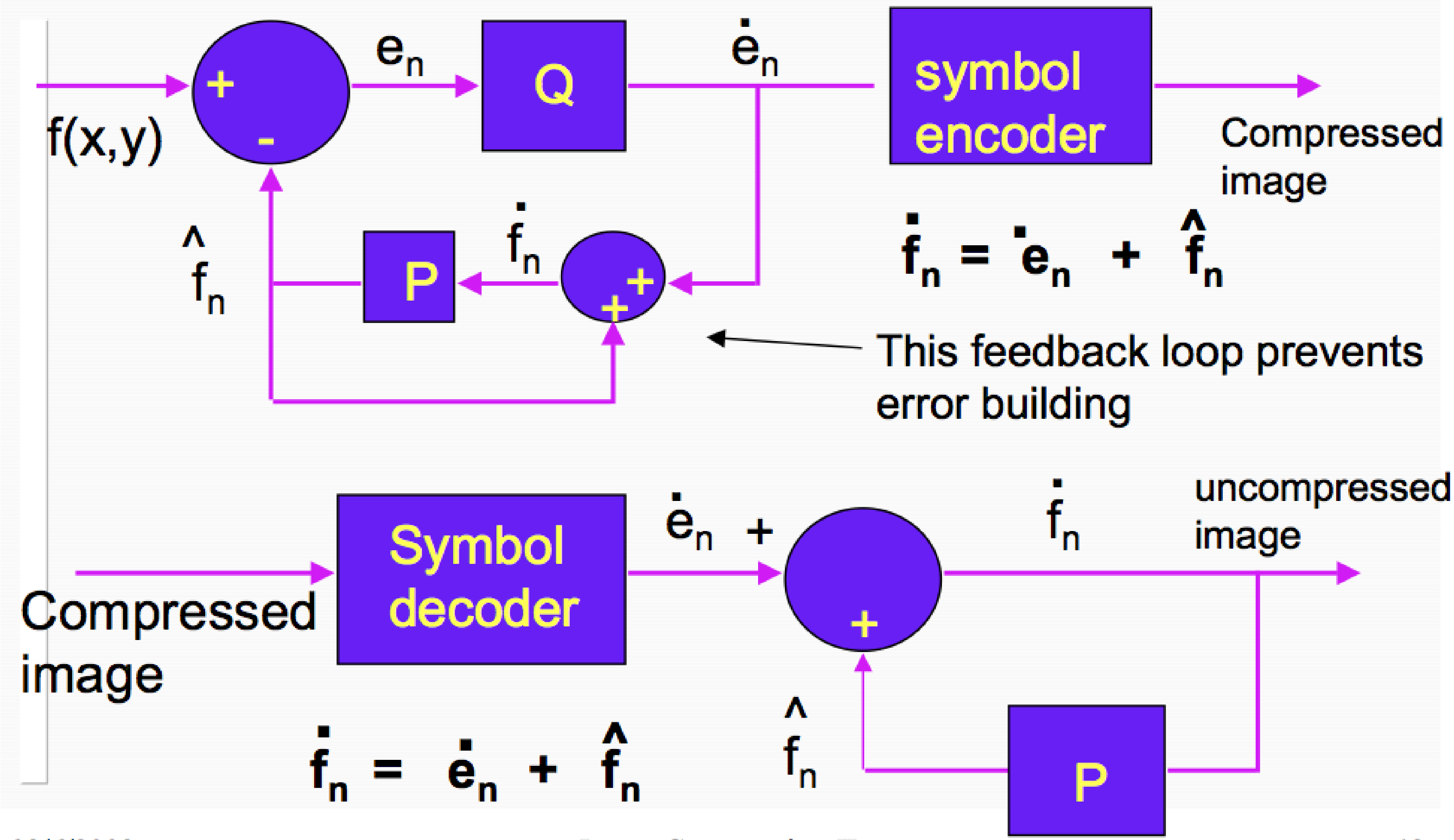
Notice that, unlike in the case of loss-less prediction, in lossy prediction the predictors P "see" different inputs at the encoder and decoder

Lossy Compression





Lossy Compression



Lossy Compression

Example:

$$\hat{f}_n = \alpha \dot{f}_{n-1}$$

$0 < \alpha < 1$
prediction coefficient

and $$\dot{e}_n = \begin{cases} +\xi & e_n > 0 \\ -\xi & e_n < 0 \end{cases}$$

$$\dot{f}_n = \dot{e}_n + \hat{f}_n$$
$$= \dot{e}_n + \alpha \dot{f}_{n-1}$$

Lossless Vs. Lossy Coding