CS 490: Computer Vision
Image Compression
Fall 2015
Dr. Michael J. Reale
INTRODUCTION AND FUNDAMENTALS
Why do we need compression?

- Consider a two hour DVD movie (standard definition)
  - 720 x 480 resolution
  - 24 bits per pixel = 3 bytes per pixel
  - 30 frames per second
  - 2 hours = 60 x 60 x 2 seconds

- $30 \times (720 \times 480) \times 3 \times 60 \times 60 \times 2 = 2.24 \times 10^{11}$ bytes = about 224 GB!
  - DVDs only hold about 8.5GB…
Why do we need compression?

- Internet transmission
  - Images
  - Video streaming
  - Videoconferencing
- Storage media (CDs, DVDs, Blu-Rays, etc.)
- Device local storage (e.g., flash memory card in a digital camera)
- Archiving
- FAX machines
Definitions

- **Data Compression**
  - Process of **reducing the amount of data** required **to represent** a given quantity of **information**

- **Data**
  - Means by which information is conveyed
  - How information is represented
  - *Example*: 10x10 binary image (all pixels black or white)
    - Information $\rightarrow$ the idea that each pixel is either black or white
    - Data $\rightarrow$ depends on representation:
      - 1 byte per pixel $\rightarrow$ black = 0, white = 255 $\rightarrow$ 100 bytes of data = **800 bits of data**
      - 1 bit per pixel $\rightarrow$ black = 0, white = 1 $\rightarrow$ **100 bits of data**
Redundant Data

- If data representation contains irrelevant or repeated information → contains redundant data
Compression Ratio

- \( a = \# \) of bits for first representation
- \( b = \# \) of bits for second representation
- **Compression Ratio \( C \):**

\[
C = \frac{a}{b}
\]

- Compares two representations \( a \) and \( b \)
  - *Example*: \( C = 10 \) \( \rightarrow \) \( a \) has 10 bits for every 1 bit of \( b \)
Relative Data Redundancy

- **Relative Data Redundancy** $R$ of $a$ and $b$:
  \[
  R = 1 - \frac{1}{C}
  \]

- Tells us how much of $a$’s data is redundant (compared to $b$)

- **Example:**
  - $C = 10$
  - $R = 1 - 0.1 \Rightarrow R = 0.9$
  - 90% of $a$’s data is redundant
2D Arrays

- Human beings like to view images as 2D arrays of data
- However, not optimal for storage
Types of Image Data Redundancies

• Three principal types of data redundancies in 2D intensity arrays:
  1) Coding redundancy
  2) Spatial and temporal redundancy
  3) Irrelevant information
I) Coding Redundancy

- **Code** = system of symbols (letters, numbers, bits, etc.) to represent information
- Every piece of information/event → assigned **code word**
  - **Code word** = sequence of code symbols
  - **Length** = # of symbols in each code word

- Typically use 8-bit codes to represent intensity → usually more bits than we actually need

Only 3 intensities: 0, 128, and 255
A histogram gives us the probability of encountering a given intensity:

\[ p_r(r_k) = \frac{n_k}{MN} \]

- \( r_k \) = discrete random variable in range \([0, U-1]\)
  - i.e., pixel intensity value, usually \([0, 256-1] \rightarrow [0, 255]\)

- \( n_k \) = # of pixels with intensity value \( k \)

- \( MN \) = width x height of image = total number of pixels
Number of Bits to Represent Values

- $u(r_k) =$ # of bits used to represent each value of $r_k$
- Average # of bits required to represent each pixel:
  \[ L_{avg} = \sum_{k=0}^{U-1} u(r_k) p_r(r_k) \]
- Total # of bits for the image = $MNL_{avg}$
“Natural” Codes

- **Natural m-bit fixed length code**
  - “Natural” binary code $\rightarrow$ use m-bit binary counting sequence with $2^m$ values
  - All $u(r_k)$ are the same $\rightarrow$ $u(r_k) = m$ for all $r_k$
  - $L_{avg}$ becomes $m$
  - *Example*: $m = 8$ $\rightarrow$ one byte per pixel $\rightarrow$ pretty much how we’ve been doing things so far
Variable-Length Codes

• **Variable-length code**
  ◦ Assign different length codes to different possible values
  ◦ More probable value $\rightarrow$ assign shorter code
Variable-Length Code Example

- 3 values in image (0, 128, and 255)
- Histogram of image:

<table>
<thead>
<tr>
<th>Intensity</th>
<th>Histogram Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>0.727</td>
</tr>
<tr>
<td>128</td>
<td>0.196</td>
</tr>
<tr>
<td>0</td>
<td>0.077</td>
</tr>
<tr>
<td>All other values</td>
<td>0</td>
</tr>
</tbody>
</table>
Variable-Length Code Example

- If we assign the following codes:

<table>
<thead>
<tr>
<th>Intensity</th>
<th>Histogram Value</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>0.727</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>0.196</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0.077</td>
<td>00</td>
</tr>
<tr>
<td>All other values</td>
<td>0</td>
<td>--</td>
</tr>
</tbody>
</table>

- The average bit length will be:

\[ L_{avg} = 0.727(1) + 0.196(2) + 0.077(2) = 1.273 \]

- Our compression ratio C and relative redundancy R will be:

\[ C = \frac{8}{1.273} \approx 6.28 \quad \quad R = 1 - \frac{1}{6.28} \approx 0.841 \]

- 84.1% of the data in the original 8-bit image is redundant!
2) Spatial and Temporal Redundancy

- **Spatial Redundancy**
  - Most pixels in 2D image correlated spatially
    - I.e., each pixel similar to or dependent on neighbor pixels

- **Temporal Redundancy**
  - In videos, pixels in a given frame similar to corresponding pixels in next (or previous) frames

- *Idea*: exploit duplicate information in space and time
Spatial Redundancy Example

- All 256 values are equally likely
  - Histogram is flat

- Each intensity selected randomly, so no connection between lines (in vertical direction)

- However, same intensity on each line
Run-Length Pairs

- Represent each line as a run-length pair
  - Specify the intensity and the number of consecutive pixels with that intensity

- Two numbers per line:

\[
C = \frac{(256 \times 256) \times 8}{(256 + 256) \times 8} = 128 \quad R = 1 - \frac{1}{128} \approx 0.992
\]

- So, 99.2% of the data in the original representation was redundant!
Mappings

- Run-lengths and other methods we will see are called **mappings**
  - *Reversible mapping* – can get original image without error
  - *Irreversible mapping* – some information is lost
3) Irrelevant Information

- Most images contain information either:
  - Ignored by human visual system
  - Not used in intended application

- Redundant here means “not used”

Can reduce this to single intensity (128)
Irrelevant Information Example

- Image is mostly 128 with some image noise
- Histogram:

  ![Histogram Image]

- Can use a single intensity value to represent whole image (128), so $C$ equals:

  $$C = \frac{(256 \times 256) \times 8}{8} = 65536$$
Difference from Previous Approaches

- Before, we were looking for repeated information
  - i.e., Coding Redundancy, Spatial/Temporal Redundancy
- Here, we are purposely removing non-essential information
  - Means that we removed quantitative information (some information was lost)
  - This removal process is referred to as quantization
THEORY BEHIND COMPRESSION
How Small Can We Shrink the Data?

- What is the minimum number of bits we need to represent a given piece of information?
  - Need a lower bound

- Turn to **Information Theory**
  - Models information as a *probabilistic process*
  - A possible value is considered a *random event*
    - *Example:* a possible *pixel intensity* would be a single random event
Measuring Information

- We can measure how much information a random event \( E \) (possible value) with probability \( P(E) \) gives us with:

\[
I(E) = \log \frac{1}{P(E)} = -\log P(E)
\]

- If \( P(E) = 1 \) (event ALWAYS occurs), then \( I(E) = 0 \)
  - No information really given by telling us \( E \) happened: \( E \) always happens around here.

- Base of log \( \rightarrow \) unit of measuring information
  - Base 2 \( \rightarrow \) 1 bit = unit of information

- Example: \( P(E) = 0.5 \) \( \rightarrow \) \( I(E) = -\log_2 2 \) \( \rightarrow \) \( I(E) = 1 \) \( \rightarrow \) only need 1 bit to say whether the event occurred (e.g., coin toss)
Entropy

- Let’s say we have several random events \{a_1, a_2, \ldots, a_J\} with associated probabilities \{P(a_1), \ldots, P(a_J)\}
  - The a’s are statistically independent random events
  - a’s \rightarrow called source symbols
    - Statistically independent random events, so \rightarrow source itself = zero-memory source

- Entropy = average information per source output:

  \[ H = - \sum_{j=1}^{J} P(a_j) \log P(a_j) \]
Image Entropy

- If image = output of imaginary zero-memory “intensity source” \( \rightarrow \) use histogram for the probabilities:

\[
H = - \sum_{k=0}^{L-1} p_r(r_k) \log_2 p_r(r_k)
\]

- \( p_r(r_k) \) = histogram value for intensity \( r_k \)
- \( L = \) maximum \# of possible values (e.g., 256)

- It is not possible to code the intensity values of the imaginary source (i.e., sample image) with fewer than \( H \) bits/pixel.
# Image Entropy Example

<table>
<thead>
<tr>
<th>Intensity</th>
<th>Histogram Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>0.727</td>
</tr>
<tr>
<td>128</td>
<td>0.196</td>
</tr>
<tr>
<td>0</td>
<td>0.077</td>
</tr>
<tr>
<td>All other values</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
H = -\left[0.727 \log_2 0.727 + 0.196 \log_2 0.196 + 0.077 \log_2 0.077\right] \\
= -\left[0.727(-0.459973) + 0.196(-2.351074) + 0.077(-3.698998)\right] \\
\approx 1.080
\]

So, the smallest number of bits we could use per value is 1.080 bits/pixel.
Image Entropy Example

- Before, though, our average bits/pixels was 1.273 (larger than 1.080)

- Is it possible to make it smaller?
Shannon’s First Theorem

- According to Shannon’s First Theorem, we can represent the image with as few as $H$ bits per pixel
  - Also called noiseless coding theorem
    - Proof involves representing $n$ consecutive source symbols with a single code word as $n$ goes to infinity

- So, entropy gives us lower bound on the # of bits per pixel was can use
Problem with Shannon’s First Theorem

- We can actually go even *lower*, if the pixels are correlated in some way (spatially, for instance)
  - See our example from before with run-length encoding

- When output of information source depends on finite number of preceding outputs → source called a *Markov* or *finite memory source*
Two Kinds of Information Sources

- **Zero-memory source**
  - Events statistically independent
  - E.g., pixels have no correlation, spatial or otherwise
  - Shannon’s first theorem applies

- **Markov (finite-memory) source**
  - Output depends on finite number of preceding outputs
    - E.g., some kind of correlation among pixels
  - May be able to go lower than Shannon’s first theorem dictates
How Do We Evaluate Lossy Compression

- In lossy compression, some information is lost
  - How do we evaluate whether 1) the information was important and 2) how much was really lost?

1) Objective Fidelity Criteria
2) Subjective Fidelity Criteria
Objective Fidelity Criteria

- $f(x,y) = \text{original image}$
- $g(x,y) = \text{reconstructed image}$
- **Error per pixel:** $e(x, y) = g(x, y) - f(x, y)$
- **Total error per pixel:** $\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [g(x, y) - f(x, y)]$

**Root-mean-square error:**

$$e_{rms} = \left[ \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [g(x, y) - f(x, y)]^2 \right]^{1/2}$$

- Akin to Euclidean distance
Mean-Square Signal-to-Noise Ratio

- If you think of $g(x,y) = f(x,y) + e(x,y)$ ($e()$ is the “noise”), then the mean-square signal-to-noise ratio is:

$$SNR_{ms} = \frac{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} g(x, y)^2}{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [g(x, y) - f(x, y)]^2}$$
Subjective Fidelity Criteria

- Images are ultimately viewed by humans
- Humans rate images on some kind of discrete scale comparing the original with the reconstructed image
  - E.g., Much worse, worse, slightly worse, the same, slightly better, better, much better
THE COMPRESSION PROCESS
Introduction

- Image compression system has:
  - **Encoder** → performs compression
  - **Decoder** → reverses process of encoder

- Both can be done in software or hardware/firmware

- **Codec** = device or program capable of both encoding or decoding
Lossy vs Lossless

- $f(x,y)$ is original, $g(x,y)$ is decoded image

- If $f(x,y) = g(x,y)$ exactly, compression system is error-free, lossless, or information preserving

- If not (some error in reconstruction/decoding), compression system is lossy
Overview of Encoding Process

- **Step 1: Mapper**
  - Minimizes spatial/temporal redundancy
  - Generally reversible
  - May reduce amount of data

- **Step 2: Quantizer**
  - Minimizes irrelevant information
  - Omitted for lossless / error-free compression
  - Not reversible
  - Reduces accuracy of mapper output according to fidelity criterion
    - *Example:* fit to bit rate (bits/second)

- **Step 3: Symbol Coder**
  - Minimizes coding redundancy
  - Generates fixed or variable length code to represent quantizer output
Overview of Decoding Process

- **Step 1: Symbol Decoder**
  - *Inverse of Symbol Encoder*

- **Step 2: Inverse Mapper**
  - *Inverse of Mapper*

- **Note:** Quantization not reversible, so not “inverse quantizer”
Image File Formats, Containers, and Compression Standards

- **Image File Format**
  - Standard way to organize and store image data
  - Defines how data is arranged and compression used

- **Image Container**
  - Similar to file format, but handles multiple types of image data
  - *Example*: QuickTime MOV

- **Image Compression Standard**
  - Defines procedures for compressing and decompressing images
METHODS: REDUCING CODING REDUNDANCY
Overview

- Huffman Coding
- Golomb Coding
- Arithmetic Coding
Huffman Coding

• One of the most popular techniques for removing coding redundancy

• Yields smallest possible number of code symbols per source symbol
  ◦ In terms of Shannon’s First Theorem → optimal for fixed value of n (if source symbols coded one at a time)
Huffman Coding: Part 1

- Build a kind of “symbol tree”
  - Get probabilities of symbols
    - E.g., histogram
  - Sort symbols by probabilities (largest first)
  - Do the following:
    - Combine smallest two probabilities → these form a “compound symbol”
      - Called a source reduction
    - Remove old probabilities and insert new “compound symbol” probability
    - Repeat until you have only two symbols left
Let’s say we have 6 symbols:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_2</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.6</td>
</tr>
<tr>
<td>a_6</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>a_1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>a_4</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a_3</td>
<td>0.06</td>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a_5</td>
<td>0.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Huffman Encoding: Part 2

• Building a code string for each possible symbol (called code assignments)
  ◦ Assign ‘0’ and ‘1’ to the two symbols at the far right
  ◦ Do the following:
    • Follow the tree down
    • Append a ‘0’ and ‘1’ to the next two symbols
### Huffman Coding: Part 2 Example

<table>
<thead>
<tr>
<th>Prob.</th>
<th>Code</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_2)</td>
<td>0.4</td>
<td>1</td>
<td>0.4</td>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>(a_6)</td>
<td>0.3</td>
<td>00</td>
<td>0.3</td>
<td>00</td>
<td>0.3</td>
</tr>
<tr>
<td>(a_{41})</td>
<td>0.1</td>
<td>011</td>
<td>0.1</td>
<td>011</td>
<td>0.2</td>
</tr>
<tr>
<td>(a_{43})</td>
<td>0.1</td>
<td>0100</td>
<td>0.1</td>
<td>0100</td>
<td>0.1</td>
</tr>
<tr>
<td>(a_{43})</td>
<td>0.06</td>
<td>01010</td>
<td>0.1</td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>(a_{45})</td>
<td>0.04</td>
<td>01011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Advantages of Huffman Coding

- Code generated $\rightarrow$ *instantaneous, uniquely decodable* block code
  - *Block code* $\rightarrow$ each source symbol mapped into a fixed sequence of code symbols
  - *Instantaneous* $\rightarrow$ can decode each code word without reference to other symbols
  - *Uniquely decodable* $\rightarrow$ any string of code symbols can only be decoded one way (assuming you read them left to right)
Disadvantages of Huffman Coding

- Large number of symbols $\rightarrow$ construction of Huffman code nontrivial
  - J symbols $\rightarrow$ J – 2 source reductions $\rightarrow$ J – 2 code assignments

- However, if source probabilities known ahead of time, can **precomputed Huffman codes**
  - *Example*: JPEG, MPEG, and others specify default Huffman coding tables pre-computed from experimental data
Golomb Coding

- Assuming inputs:
  - Are non-negative integer inputs
  - Have exponentially decaying probability distributions
    - I.e., small integers $\rightarrow$ larger probabilities

- Advantages:
  - Can get optimal code (Shannon’s First Theorem)
  - Computationally simpler than Huffman codes
Golomb Coding: Step 1

- **Given:**
  - $n = \text{non-negative integer}$
  - $m = \text{positive integer divisor} \ (m > 0)$

- **Step 1:** Get unary code of $\lfloor \frac{n}{m} \rfloor$
  - **Unary code of** $q \rightarrow q$ 1’s followed by a 0
    - *Example:* $n = 5, m = 1$
      - $q = \text{floor}(n/m) = \text{floor}(5/1) = 5$
      - Unary code for 5 $\rightarrow 11110$
Golomb Coding: Step 2

• Step 2: \[ k = \lceil \log_2 m \rceil \] → Min. # of bits for m
  \[ c = 2^k - m \] → Max value + 1 minus m
  \[ r = n \mod m \] → Remainder of n/m

  - Find truncated remainder \( r' \) such that:

    \[ r' = \begin{cases} 
    r & \text{truncated to } k-1 \text{ bits} \quad 0 \leq r < c \\
    r + c & \text{truncated to } k \text{ bits} \quad \text{otherwise}
    \end{cases} \]
Golomb Coding: Step 3

- Step 3: Concatenate results from steps 1 and 2
Golomb-Rice Codes

- If $m = 2^k$, then:
  - $c = 0$
  - $r' = r = n \mod m$ truncated to $k$ bits for all $n$

- Basically, nothing but binary shift operations

- The codes generated are called Golomb-Rice or Rice codes.
## Golomb-Rice Code Examples

<table>
<thead>
<tr>
<th>n</th>
<th>( m = 2^0 = 1 )</th>
<th>( m = 2^1 = 2 )</th>
<th>( m = 2^2 = 4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( G_1(n) )</td>
<td>( G_2(n) )</td>
<td>( G_4(n) )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>10 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>3</td>
<td>1110</td>
<td>10 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>110 0</td>
<td>10 0 0</td>
</tr>
<tr>
<td>5</td>
<td>111110</td>
<td>110 1</td>
<td>10 0 1</td>
</tr>
<tr>
<td>6</td>
<td>1111110</td>
<td>1110 0</td>
<td>10 1 0</td>
</tr>
<tr>
<td>7</td>
<td>11111110</td>
<td>1110 1</td>
<td>10 1 1</td>
</tr>
</tbody>
</table>

### Unary code of

\[
\left\lfloor \frac{n}{m} \right\rfloor
\]

### Truncated Remainder \( r' \)

\[
r' = r = n \mod m
\]

**Min # of bits:**

\[
k = \left\lceil \log_2 m \right\rceil
\]
How do you pick m?

- One question remains: how do you pick the best m?

- If data is geometrically distributed with probability mass function (PMF)

$$P(n) = (1 - p)p^n$$

- for some $0 < p < 1$, Golomb codes are optimal when:

$$m = \left\lfloor \frac{\log_2(1 + p)}{\log_2(1/p)} \right\rfloor$$
Coding Images

- *Problem*: Most intensity distributions won’t look like this:

- On the other hand, if we use it to code *image differences*…
  - Such as the error from reconstruction?
How to Handle Negative Numbers with Image Differences

- Obviously, image differences have negative values, so use mapping like this:

\[ M(n) = \begin{cases} 
2n & n \geq 0 \\
2|n| - 1 & n < 0 
\end{cases} \]

- Reordering integers, alternating positive and negative values
Arithmetic Coding

- With Huffman and Golomb → assigning block code to each symbol

- Arithmetic Coding
  - No 1-to-1 symbol/code word correspondence
  - Sequence of source symbols (message) → single arithmetic code word
  - Can get nearly optimal results
Arithmetic Coding: Method

- Given symbol probabilities, assign decimal range for each symbol:

<table>
<thead>
<tr>
<th>Source Symbol</th>
<th>Probability</th>
<th>Initial Subinterval</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1</td>
<td>0.2</td>
<td>[0.0, 0.2)</td>
</tr>
<tr>
<td>a_2</td>
<td>0.2</td>
<td>[0.2, 0.4)</td>
</tr>
<tr>
<td>a_3</td>
<td>0.4</td>
<td>[0.4, 0.8)</td>
</tr>
<tr>
<td>a_4</td>
<td>0.2</td>
<td>[0.8, 1.0)</td>
</tr>
</tbody>
</table>

- Note that the interval is open on one side
Arithmetic Coding: Method

- Say we have a five-symbol sequence (or message) → $a_1a_2a_3a_4$

- We want to construct a single decimal number representing this message
Arithmetic Coding:

Keep making smaller and smaller subinterval until message is finished \( \rightarrow \) can pick any number in final interval (0.068)
Arithmetic Coding: Problems

- Often need an “end of message” character
- Finite precision arithmetic
  - Solution: effectively end up using integer digits strung together
- Many arithmetic coding techniques protected by U.S. Patent Law
  - Hence, JPEG officially has options for Huffman and Arithmetic, but typically implementation just has Huffman
METHODS: REDUCING SPATIAL AND TEMPORAL REDUNDANCY
Overview

- LZW Coding
- Run-Length Encoding (RLE)
- Symbol-Based Coding
- Bit-Plane Coding
- Block Transform Coding
  - WHT
  - DCT
- Predictive Coding
  - Including Motion Compensation
LZW Coding

- **Lempel-Ziv-Welch (LZW) coding**
  - Lossless
  - Assigns **fixed-length code words** to **variable length sequences** of source symbols
  - Does NOT require knowledge of probabilities of source symbols
  - Removes **spatial redundancy**
LZW: Method

- Construct codebook (or dictionary) containing source symbols
  - For 8-bit grayscale image, first 256 words of dictionary assigned to intensities [0,255]
- As encoder goes through data, looks for sequences it hasn’t seen yet
  - Adds new ones to dictionary
    - Example: sees “255,255” → might be assigned code 256
  - When it sees sequence again, uses new code word
  - For 9-bit, 512 word dictionary, using 9 bits for (8+8) original bits
LZW: Dictionary Size

- Too small
  - Less likely to find matching intensity-level sequences

- Too large
  - Size of code words will negatively affect compression
LZW: Encoding Example

<table>
<thead>
<tr>
<th>Currently Recognized Sequence</th>
<th>Pixel Being Processed</th>
<th>Encoded Output</th>
<th>Dictionary Location (Code Word)</th>
<th>Dictionary Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>39</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>39</td>
<td>256</td>
<td>39-39</td>
</tr>
<tr>
<td>39</td>
<td>126</td>
<td>39</td>
<td>257</td>
<td>39-126</td>
</tr>
<tr>
<td>126</td>
<td>126</td>
<td>126</td>
<td>258</td>
<td>126-126</td>
</tr>
<tr>
<td>126</td>
<td>39</td>
<td>126</td>
<td>259</td>
<td>126-39</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>126</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>126-126</td>
<td>39</td>
<td>258</td>
<td>261</td>
<td>126-126-39</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39-39</td>
<td>126</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LZW: Aside

- Unique algorithm in that *code book created while data is encoded*
  - Decoder actually builds identical code book as it decodes the data
### LZW: Decoding Example

<table>
<thead>
<tr>
<th>Encoded Input</th>
<th>Decoded Output</th>
<th>New Sequence</th>
<th>Dictionary Location (Code Word)</th>
<th>Dictionary Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>39</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>126</td>
<td>39-126</td>
<td>257</td>
<td>39-126</td>
</tr>
<tr>
<td>126</td>
<td>126</td>
<td>126-126</td>
<td>258</td>
<td>126-126</td>
</tr>
<tr>
<td>256</td>
<td>39</td>
<td>126-39</td>
<td>259</td>
<td>126-39</td>
</tr>
<tr>
<td>39</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LZW: Dictionary Overflow

- What happens when the dictionary is full (dictionary overflow)?

- **Possible solutions:**
  - Flush/reinitialize dictionary and continue coding with reinitialized dictionary
    - *More complex*: Monitor compression performance → flush dictionary if compression poor
  - Track and replace least used dictionary entries
Run-Length Encoding

- **Run-length encoding (RLE)**
  - Represents repeated intensities with *run-length pairs*
    - Start position + # of pixels with same intensity
  - Developed in 1950s → standard for FAX compression
    - CCITT Group 3 and 4 standards
RLE: Binary Images

- RLE particularly effective with **binary images**
  - Only two possible values (black and white)
  - Method:
    - Code each contiguous group (run) with length
    - Set some convention for which value the line starts with
      1) Specify first value
      2) Assume starts with white (but allow length to be zero)
RLE: Coding the Run-Lengths

- On top of using RLE, you can also use **variable-length coding on the run-length pairs themselves**
  - Usually do white and black runs separately
  - *Example:* set $a_j = \text{black run of length } j$
    - Estimate probability by $(\# \text{ of black run lengths of length } j \text{ in image})/(\text{total number of black runs})$
Symbol-Based Coding

- **Symbol-Based (or Token-Based) Coding**
  - Image represented as collection of frequently-used subimages, called **symbols**
  - **Symbol dictionary** contains all symbols
  - Image coded as triplets \((x,y,t)\)
    - \((x,y)\) → coordinate
    - \((t)\) → symbol (token) index
  - Particular useful for **document compression**
    - Repeated letters
Symbol-Based Coding

- Proposed in early 1970s → only recently more practical because of symbol-matching algorithms
  - Encoding can be time-consuming, but decoding is very fast
- Can also compress the symbols themselves
Symbol-Based Coding

- **Lossless** → only exact symbol matches allowed

- **Lossy** → small differences permitted in symbol matching
Bit-Plane Coding

- Previous two approaches used *binary images*:
  - Run-Length Encoding
  - Symbol-Based Encoding

- To use on multiple intensities (even color), break image into bit planes
  - Called *bit-plane coding*
Recall: Bit-Plane Slicing

- Can decompose 8-bit image into “bit planes”
- Highest-order bits contain most information

http://hwshow-ipc.blogspot.com/2009/12/hw5-8-bit-plane-slicing.html
Bit-Plane Coding: Problem

- **Problem**: small intensity differences → increase complexity of planes
  - Example: values 127 and 128 → 011111111 and 100000000 in binary
    - Creates 0-1 and 1-0 transitions on all planes
Bit-Plane Coding: Gray Codes

- *Alternative:* first represent image by *m*-bit Gray code
  - \( a_{m-1} \ldots a_2 a_1 a_0 \rightarrow \) m-bit intensity value
  - \( g_{m-1} \ldots g_2 g_1 g_0 \rightarrow \) m-bit Gray code
  
  \[
  g_i = a_i \oplus a_{i+1} \quad 0 \leq i \leq m - 2
  \]

  \[
  g_{m-1} = a_{m-1}
  \]

  - \( \oplus = \) exclusive OR
  - Successive code words differ in only one bit position
    \( \rightarrow \) small changes unlikely to affect complexity of bit planes
Block Transform Coding

**Procedure:**

- Divide image into small, non-overlapping blocks of equal size (e.g., 8x8)
- Process blocks independently using 2D transform
  - Use reversible, linear transform
  - Map each block/subimage → set of transform coefficients
  - Quantize coefficients and code them
Block Transform Coding: Encoder

**Encoder:**

Input image (M x N) → Construct n x n subimages → Forward Transform → Quantizer → Symbol Encoder → Compressed image
Block Transform Coding: Decoder

Decoder:

Compressed image → Symbol Decoder → Inverse Transform → Merge n x n subimages → Decompressed image
Block Transform Coding

- Any or all of the transform encoding steps can be:
  - **Adaptive** → adapted to local image content
  - **Nonadaptive** → fixed for all subimages
Picking a Transformation

- But which transformation, you may ask, should we use?
General Transformation

- Given an image of size n×n, we can perform a discrete transformation \( T(u,v) \)
  \[
  T(u,v) = \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} f(x,y) r(x,y,u,v)
  \]

- The inverse is given by:
  \[
  f(x,y) = \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} T(u,v) s(x,y,u,v)
  \]

- \( r(x,y,u,v) \) and \( s(x,y,u,v) \) are the **forward and inverse transformation kernels**
  - These are also referred to as **basis functions** or **basis images**
Transform Coefficients

- $T(u,v)$ for $u,v = 0,1,2,...n-1$ are the transform coefficients
Relation to Fourier Transform

- If we use the following kernels:

\[ r(x, y, u, v) = e^{-j2\pi(ux+vy)/n} \]

\[ s(x, y, u, v) = \frac{1}{n^2} e^{j2\pi(ux+vy)/n} \]

- ...and put them in the previous equations, we have a simplified form of the Discrete Fourier Transform!
Kernels

- By setting different kernels, we can get different transformations
  - Some will be more useful than others, obviously.
Walsh-Hadamard Kernels (WHT)

- Walsh-Hadamard Kernels (WHT)
  - Use alternating +1 and -1 pattern
    - Checkerboard patterns

**FIGURE 8.29** Walsh-Hadamard basis functions for $N = 4$. The origin of each block is at its top left.
Discrete Cosine Transform (DCT)

- Discrete Cosine Transform (DCT)
  - Curiously, both the forward and inverse kernels are equal:

\[
\begin{align*}
    r(x, y, u, v) &= s(x, y, u, v) \\
    &= \alpha(u)\alpha(v) \cos\left(\frac{(2x+1)u\pi}{2n}\right) \cos\left(\frac{(2y+1)v\pi}{2n}\right)
\end{align*}
\]

\[
\alpha(u) = \begin{cases} 
    \sqrt{\frac{1}{n}} & \text{for } u = 0 \\
    \frac{2}{n} & \text{for } u = 1, 2, \ldots, n-1
\end{cases}
\]
Discrete Cosine Transform (DCT)
Basis Functions

- Looking back on the original inverse transform:
  \[
  f(x, y) = \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} T(u, v)s(x, y, u, v)
  \]

- We can look as \(s(x,y,u,v)\) as a matrix \(S_{uv}\), where:
  \[
  S_{uv} = \begin{bmatrix}
  s(0,0,u,v) & s(0,1,u,v) & \ldots & s(0,n-1,u,v) \\
  s(1,0,u,v) & \vdots & \ddots & \vdots \\
  \vdots & \vdots & \ddots & \vdots \\
  s(n-1,0,u,v) & s(n-1,1,u,v) & \ldots & s(n-1,n-1,u,v)
  \end{bmatrix}
  \]

- Each one of these is a **basis function** or **basis image**
  - Reconstructed image built from these basis images
How Many Coefficients?

- We can use the mean-square error between the original image and the reconstructed image:

\[
e_{ms} = E \left\{ \left\| F - G \right\|^2 \right\}
\]

- F = original
- G = reconstructed
Which Kernel is Better?

- **DCT** better at packing information than DFT or WHT
  - **DCT** = major international standard for encoding information

- **PCA**, however, is actually optimal
  - However, requires training set
  - Best if you know images in certain domain
DCT: Advantages

- Good compromise between information packing ability and computational complexity
- Can be implemented in a single integrated circuit
- Minimizes blocking artifact
  - When boundaries of subimages become visible
  - DCT reduces this because of 2n-point periodicity
Block Transform Coding: Picking Subimage Size

- Usually pick power of 2 for subimage size
  - Simplifies computation for most approaches

- Larger the subimage size →
  - Increases level of compression
  - Increases computational complexity

- 8x8 and 16x16 → most popular sizes
Block Transform Coding: Storing Transform Coefficients

- Error of reconstruction dependent on:
  - Number and importance of retained transform coefficients
    - We will discuss two approaches for picking coefficients later
  - Precision used to store coefficients
    - *Bit allocation* → process of truncating, quantizing, and coding coefficients
Predictive Coding

- Predictive Coding
  - Eliminates redundancies of closely-spaced pixels in space and/or time →
  - Extracts/codes only *new information* in each pixel
    - *New information* = difference between actual and predicted value of pixel
Predictive Coding: Advantages

- Simple

- Achieves good compression without significant computational overhead

- Can be error-free or can be lossy
Predictor and Prediction Error

- **Predictor**
  - Uses past samples (in space and/or time) to predict current value of pixel
  - Output rounded to nearest integer

- **Prediction Error**
  - If:
    - \( f(n) = \) original function (image)
    - \( g(n) = \) predictor
  - Then error \( e(n) \) is:
    - \( e(n) = f(n) - g(n) \)
Lossless Predictive Coding

- Identical predictor in both encoder and decoder

**Encoding:**
- Get error $e(n) = f(n) - g(n)$
- Encode error as variable-length code
  - Note: all you are storing is the error from the prediction!

**Decoding:**
- Decode error
- Add to predictor to get true value: $f(n) = e(n) + g(n)$
Linear Predictor

- Often linear combination of $m$ previous samples:

$$g(n) = \text{round} \left[ \sum_{i=1}^{m} a_i f(n-i) \right]$$

  - $m$ = “order” of predictor
  - Round = round to nearest integer operation
  - $a_i$ = prediction coefficients

- First $m$ values must be encoded as well (e.g., using Huffman code)
Linear Predictive Coding

- If $f(n)$ image, then $m$ samples come from
  - Current scan line $\rightarrow$
    1D linear predictive coding
  - Current and previous scan lines $\rightarrow$
    2D linear predictive coding
  - Current image and previous images $\rightarrow$
    3D linear predictive coding

[Diagram of linear predictive coding with examples of scan lines and previous images]
Simple First-Order Linear Predictor for Images

- If $m = 1$, then:

$$g(x, y) = af(x, y - 1)$$

- Called a previous pixel predictor
  - Procedure $\rightarrow$ *differential coding* or *previous pixel coding*
Why Previous Pixel Predictors Can Work

• It seems like such a simple predictor wouldn’t work that well.

• However, we are encoding the error:
  ◦ (standard deviation of error) << (standard deviation of image)
  ◦ Entropy is also much lower
    • Even though we need (k+1) bits for the error of k-bit images
Estimating Lower Bound for Predictive Coding

- Generally, best you can do is:
  \[ \frac{k}{e} \]

- where:
  - \( k \) = # of bits for each pixel in original image
  - \( e \) = estimated entropy of prediction error
The prediction error sequence is often called the **prediction residual**
Redundancies in Time

- You can also use a first-order linear predictor in time:

\[ g(x, y, t) = \text{round}[af(x, y, t - 1)] \]

- Can get much better results than from using spatial information alone
Problems with Time

• If sequence contains:
  ◦ Rapidly moving objects
  ◦ Camera zooms and pans
  ◦ Cuts, fade-ins, fade-outs, etc.

then there’s little temporal redundancy
  ◦ Can actual get *data expansion* using prediction system!
Avoiding Data Expansion

- Two ways to avoid data expansion in video compression:
  - 1) Track objects and compensate for movement during prediction/differencing process
    - Called **motion compensation**
  - 2) Switch to alternate coding method when interframe (between frame) correlation is too low
    - Often use block-oriented 2D transform (e.g., JPEG’s DCT-based coding)
    - Frames compressed this way → **intraframes** or **independent frames (I-frames)**
I-Frames

- **I-Frames (Independent Frames or Intraframes)**
  - Resemble JPEG encoded images
  - Encoded without reference to previous frames
  - Starting points for prediction
  - High degree of random access
  - Ease of editing
  - Resistant to propagation of transmission error

- All standards require periodic insertion of I-frames in a compressed video stream
Motion Compensation

- Divide frame into **macroblocks**
  - Non-overlapping rectangular regions, usually 4x4 to 16x16

- **Reference frame** = “most likely” position of macroblock

- Motion of macroblock $\rightarrow$ encoded as **motion vector**
  - Displacements in $x$ and $y$
  - May use subpixel accuracy (e.g., $\frac{1}{4}$ pixel)
    - Have to use interpolation from reference frame
Motion Compensation (cont.)

- Encoded frame based
  - Previous frame $\rightarrow$ **forward prediction**
    - Frame called a *Predictive frame (P-frame)*
  - Subsequent frame $\rightarrow$ **backward prediction**
    - Frame called a *Bidirectional frame (B-frame)*
    - Frames have to be reordered going into the decoder
Motion Estimation

- Different approaches and error measures
  - One of the most common error measures → *mean absolute distortion (MAD)*
    
    $$MAD(x, y) = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} |f(x + i, y + j) - p(x + i + dx, y + j + dy)|$$
    
    - $p(x,y) = \text{predicted macroblock values}$
    - $dx, dy = \text{displacement}$

  - **Block matching** – searching for $dx, dy$ that minimizes $MAD(x,y)$ over allowed range of motion vector displacements
    - Exhaustive search → guarantees best results, but computationally expensive
    - Faster searches → may not find best results

  - Fortunately, only encoder has to do motion estimation
METHODS: REDUCING IRRELEVANT INFORMATION
Block Transform Coding: Picking Coefficients

- By picking the most important coefficients, we are purposely excluding (hopefully) irrelevant information.

- Two approaches:
  - Zonal coding
  - Threshold coding
Zonal Coding

- Zonal Coding
  - Pick coefficients based on *maximum variance*
    - More variance $\rightarrow$ more uncertainty $\rightarrow$ more information
  - Either calculate variance directly OR based on assumed image model
  - Construct a *zonal mask*
    - Usually *single, fixed mask* for all subimages
    - 1 $\rightarrow$ keep coefficient, 0 $\rightarrow$ drop coefficient
  - May also have mask for bit allocation
    - Each entry shows how many bits to use
# Zonal Mask and Bit Allocation

<table>
<thead>
<tr>
<th>Zonal Mask</th>
<th>Zonal Bit Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Zonal Mask" /></td>
<td><img src="image" alt="Zonal Bit Allocation" /></td>
</tr>
</tbody>
</table>

- **Zonal Mask**:
  - 0 represent non-zonal areas.
  - 1 represent zonal areas.

- **Zonal Bit Allocation**: 8, 7, 6, 4, 3, 2, 1, 0
Threshold Coding

- **Threshold Coding**
  - Pick coefficients based on *maximum magnitude*
    - Largest coefficients → make largest contribution to image
  - Approach inherently adaptive
  - **Procedure:**
    - Threshold coefficients
    - Reorder coefficients as single $n^2$ coefficient sequence
    - Represent thresholded sequence with run-length encoding
Threshold Coding: Reordering “Mask”

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>5</th>
<th>6</th>
<th>14</th>
<th>15</th>
<th>27</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>13</td>
<td>16</td>
<td>26</td>
<td>29</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
<td>12</td>
<td>17</td>
<td>25</td>
<td>30</td>
<td>41</td>
<td>43</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>11</td>
<td>18</td>
<td>24</td>
<td>31</td>
<td>40</td>
<td>44</td>
<td>53</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>19</td>
<td>23</td>
<td>32</td>
<td>39</td>
<td>45</td>
<td>52</td>
<td>54</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>22</td>
<td>33</td>
<td>38</td>
<td>46</td>
<td>51</td>
<td>55</td>
<td>60</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>34</td>
<td>37</td>
<td>47</td>
<td>50</td>
<td>56</td>
<td>59</td>
<td>61</td>
</tr>
<tr>
<td>35</td>
<td>35</td>
<td>36</td>
<td>48</td>
<td>49</td>
<td>57</td>
<td>58</td>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

Reordered as follows:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
<th>...</th>
<th>63</th>
</tr>
</thead>
</table>

Reordered as follows:
Threshold Coding: Thresholding Methods

- Threshold in three ways:
  - 1) *Single Global Threshold*
    - Gives unknown number of coefficients
  - 2) *N-Largest coding*
    - Always discard same number of coefficients
  - 3) *Threshold varies depending on coefficient position*
    - Gives unknown number of coefficients
    - Can combine thresholding and quantization
Threshold Coding: Transform Normalization

- Coefficients divided by appropriate mask value $\rightarrow$ round to get thresholded value
  - $T(u,v) = \text{transform coefficients}$
  - $Z(u,v) = \text{transform normalization array}$
  - $G(u,v) = \text{thresolded and quantized approximation of } T(u,v)$

$$G(u,v) = \text{round} \left[ \frac{T(u,v)}{Z(u,v)} \right]$$

Matrix used in JPEG standard

$$Z(u,v) = \begin{bmatrix}
16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\
12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\
14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\
14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\
18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\
24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\
49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\
72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \\
\end{bmatrix}$$