Lec 05
Arithmetic Coding

Zhu Li

Course Web:
http://l.web.umkc.edu/lizhu/teaching/2016sp.video-communication/main.html

Outline

- Lecture 04 ReCap
- Arithmetic Coding
- About Homework-1 and Lab

JPEG Coding

- Block (8x8 pel) based coding
- DCT transform to find sparse representation
- Quantization reflects human visual system
- Zig-Zag scan to convert 2D to 1D string
- Run-Level pairs to have even more compact representation
- Hoffman Coding on Level Category
- Fixed on the Level with in the category

Quant Table:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>11</td>
<td>10</td>
<td>16</td>
<td>24</td>
<td>40</td>
<td>51</td>
<td>61</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>14</td>
<td>19</td>
<td>26</td>
<td>58</td>
<td>60</td>
<td>55</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>16</td>
<td>24</td>
<td>40</td>
<td>57</td>
<td>60</td>
<td>56</td>
</tr>
<tr>
<td>14</td>
<td>17</td>
<td>22</td>
<td>29</td>
<td>35</td>
<td>33</td>
<td>37</td>
<td>46</td>
</tr>
<tr>
<td>14</td>
<td>22</td>
<td>37</td>
<td>56</td>
<td>68</td>
<td>109</td>
<td>103</td>
<td>77</td>
</tr>
<tr>
<td>24</td>
<td>35</td>
<td>55</td>
<td>64</td>
<td>81</td>
<td>104</td>
<td>113</td>
<td>92</td>
</tr>
<tr>
<td>49</td>
<td>64</td>
<td>78</td>
<td>87</td>
<td>100</td>
<td>112</td>
<td>120</td>
<td>104</td>
</tr>
<tr>
<td>72</td>
<td>92</td>
<td>95</td>
<td>98</td>
<td>112</td>
<td>180</td>
<td>103</td>
<td>99</td>
</tr>
</tbody>
</table>

Coding of AC Coefficients

Zigzag scanning:

Example:

<table>
<thead>
<tr>
<th>8</th>
<th>24</th>
<th>-2</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-31</td>
<td>-4</td>
<td>6</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-12</td>
<td>-1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: zigzag scanning result

24 -31 0 -4 -2 0 6 -12 0 0 0 -1 -1 0 0 0 2 -2 0 0 0 0 0 -1 EOB

(Run, level) representation:

- (0, 24), (0, -31), (1, -4), (0, -2), (1, 6), (0, -12), (3, -1), (0, -1), (3, 2), (0, -2), (5, -1), EOB
Coding of AC Coefficients

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EOB</td>
<td>1010</td>
<td>-</td>
<td>-</td>
<td>...</td>
<td>ZRL</td>
</tr>
<tr>
<td>0/1</td>
<td>00</td>
<td>1/1</td>
<td>1100</td>
<td>...</td>
<td>15/1</td>
</tr>
<tr>
<td>0/2</td>
<td>01</td>
<td>1/2</td>
<td>11011</td>
<td>...</td>
<td>15/2</td>
</tr>
<tr>
<td>0/3</td>
<td>100</td>
<td>1/3</td>
<td>1111001</td>
<td>...</td>
<td>15/3</td>
</tr>
<tr>
<td>0/4</td>
<td>1011</td>
<td>1/4</td>
<td>111110110</td>
<td>...</td>
<td>15/4</td>
</tr>
<tr>
<td>0/5</td>
<td>11010</td>
<td>1/5</td>
<td>11111110110</td>
<td>...</td>
<td>15/5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- ZRL: represent 16 zeros when number of zeros exceeds 15.
  - Example: 20 zeros followed by -1: (ZRL), (4, -1).
- (Run, Level) sequence: (0, 24), (0, 31), (1, -4), …
- Run/Cat. Sequence: 0/5, 0/5, 1/3, …
- 24 is the 24-th entry in Category 5 ⇒ (0, 24): 11010 11000
- -4 is the 3-th entry in Category 3 ⇒ (1, -4): 1111001 011

Outline

- Lecture 04 ReCap
  - Arithmetic Coding
    - Basic Encoding and Decoding
    - Uniqueness and Efficiency
    - Scaling and Incremental Coding
    - Integer Implementation
- About Homework-1 and Lab

Arithmetic Coding – The SciFi Story

- When I was in my 5th grade….

Aliens visit earth….
Recall table look-up decoding of Huffman code
- \( N \): alphabet size
- \( L \): Max code word length
- Divide \([0, 2^L]\) into \( N \) intervals
- One interval for one symbol
- Interval size is roughly proportional to symbol prob.

Arithmetic coding applies this idea recursively
- Normalizes the range \([0, 2^L]\) to \([0, 1]\).
- Map an input sequence (multiple symbols) to a unique tag in \([0, 1]\).

Some Questions to think about:
- Why compression is achieved this way?
- How to implement it efficiently?
- How to decode the sequence?
- Why is it better than Huffman code?

Example:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>0.02</td>
</tr>
<tr>
<td>3</td>
<td>0.18</td>
</tr>
</tbody>
</table>

Map to real line range \([0, 1]\)
Order does not matter
- Decoder needs to use the same order
  - Disjoint but complete partition:
    - 1: \([0, 0.8]\): 0, 0.799999...9
    - 2: \([0.8, 0.82]\): 0.8, 0.819999...9
    - 3: \([0.82, 1]\): 0.82, 0.999999...9
Concept of the encoding (not practical)

Input sequence: “1321”

- Range 0.00288
  - 1
  - 0.7712
  - 0.77408
  - 0.773504
  - 0.7735616

- Range 0.144
  - 1
  - 0.656
  - 0.77408
  - 0.7712
  - 0.8

- Range 0.8
  - 1
  - 0.64
  - 0.656
  - 0.8

Termination: Encode the lower end or midpoint to signal the end.

Difficulties:
1. Shrinking of interval requires very high precision for long sequence.
2. No output is generated until the entire sequence has been processed.

Encoder Pseudo Code

- Keep track of LOW, HIGH, RANGE
- Any two are sufficient, e.g., LOW and RANGE.

```
LOW=0.0, HIGH=1.0;
while (not EOF) {
    n = ReadSymbol();
    RANGE = HIGH - LOW;
    HIGH = LOW + RANGE * CDF(n);
    LOW = LOW + RANGE * CDF(n-1);
}
output LOW;
```

Input | HIGH | LOW | RANGE |
-----|------|-----|-------|
Initial | 1.0  | 0.0 | 1.0   |
1      | 0.0+1.0*0.8=0.8 | 0.0+1.0*0 = 0.0 |         |
3      | 0.0 + 0.8*1=0.8 | 0.0 + 0.8*0.82=0.656 | 0.8     |
2      | 0.656+0.144*0.8=0.77408 | 0.656+0.144*0.8=0.77408 | 0.00288 |
1      | 0.7712+0.00288*0=0.7712 | 0.7712+0.00288*0.8=0.773504 | 0.002304 |

Concept of the Decoding

- Arithmetic Decoding (conceptual)
  - Suppose encoder encodes the lower end: 0.7712

Drawback: need to recalculate all thresholds each time.
Receive 0.7712
Decode 1
\[ x = \frac{0.7712 - 0}{0.8} = 0.964 \]
Decode 3
\[ x = \frac{0.964 - 0.82}{0.18} = 0.8 \]
Decode 2
\[ x = \frac{0.8 - 0.8}{0.02} = 0 \]
Decode 1. Stop.

Uniqueness

Termination: Encode the midpoint (or lower end) to signal the end.

How to represent the final tag uniquely and efficiently?

Answer: Take the binary value of the tag \( T(X) \) and truncate to \( X \) is the sequence \( \{x_1, \ldots, x_m\} \), not individual symbol

\[
I(X) = \left\lceil \log \frac{1}{p(X)} \right\rceil + 1 \text{ bits.}
\]

1 bit longer than Shannon code

Proof: Assuming midpoint tag: \( T(X) = F(X-1) + \frac{1}{2} p(X), \ p(X) > 0. \)

First show the truncated code is unique, that is, code is within \([F(X-1), F(X))\).

1. \[
T(X)_{\lfloor X \rceil} \leq T(X) < F(X)
\]

So \( T(X)_{\lfloor X \rceil} \) is below the high end of the interval.
Uniqueness and Efficiency

2). \[ 2^{-l(X)} = 2^{-\left[ \log_2 \frac{1}{p(X)} + 1 \right]} \leq 2^{-\left[ \log_2 \frac{1}{p(X)} + 1 \right]} = \frac{1}{2} p(X) \]

By def,
\[ T(X) = F(X-1) + \frac{1}{2} p(X), \quad p(X) > 0. \]

\[ T(X) - F(X - 1) = \frac{1}{2} p(X) \geq 2^{-l(X)} \]

Together with
\[ T(X) - [T(X)]_{I(X)} \leq 2^{-l(X)} \]

Thus
\[ F(X - 1) \leq [T(X)]_{I(X)} < F(X) \]

So the truncated code is still in the interval. This proves the uniqueness.

Uniqueness and Efficiency

Efficiency of arithmetic code:
\[ l(X)^m_n = \left\lfloor \log_2 \frac{1}{p(X)^m_n} \right\rfloor + 1 \text{ bits.} \]

\[ X^m_n : \{x_1, \ldots, x_m\} \]

\[ L = E\left\{p(X)^m_n l(X)^m_n \right\} = \sum P(X)^m_n \left\lfloor \log_2 \frac{1}{p(X)^m_n} \right\rfloor + 1 \]

\[ \leq \sum P(X)^m_n \left( \log_2 \frac{1}{p(X)^m_n} + 1 + 1 \right) = H(X)^m_n + 2 \]

\[ l(X) \text{ is the bits to code a sequence } \{x_1, x_2, \ldots, x_n\}. \]

Assume iid sequence, \[ H(X)^m_n = mH(X) \]

\[ H(X) \leq \frac{L}{m} \leq H(X) + \frac{2}{m} \]

\[ L/m \rightarrow H(X) \text{ for large } m, \text{ stronger than prev results.} \]

Uniqueness and Efficiency

Prove the code is uniquely decodable (prefix free):

Any code with prefix \[ [T(X)]_{I(X)} \text{ is in } [T(X)]_{I(X)} + [T(X)]_{I(X)} + \frac{1}{2^{l(X)}} \]

Need to show that this is in \([F(X-1), F(X))\):

We already show \[ F(X) - [T(X)]_{I(X)} > \frac{1}{2^{l(X)}} \]

Only need to show \[ F(X) - [T(X)]_{I(X)} > F(X) - T(X) = \frac{p(X)}{2} > \frac{1}{2^{l(X)}} \]

\[ [T(X)]_{I(X)} \text{ is prefix free if } l(X) = \left\lfloor \log_2 \frac{1}{p(X)} \right\rfloor + 1 \text{ bits.} \]

Comparison with Huffman code:

- Expected length of Huffman code:

\[ H(X) \leq L' \leq H(X) + 1 \]

- Huffman code can reduce the overhead by jointly encoding more symbols, but needs much larger alphabet size.

- Arithmetic coding is more efficient for longer sequences.
Scaling and Incremental Coding

- Problems of Previous examples:
  - Need high precision
  - No output is generated until the entire sequence is encoded.
  - Decoder needs to read all input before decoding.

- Key Observation:
  - As the RANGE reduces, many MSB's of LOW and HIGH become identical:
    - Example: Binary form of 0.7712 and 0.773504:
      - 0.1100010..., 0.1100011...
  - We can output identical MSB's and re-scale the rest:
    - Incremental encoding
    - Can achieve infinite precision with finite-precision integers.
  - Three scaling scenarios: E1, E2, E3

- Important Rules: Apply as many scalings as possible before further encoding and decoding.

E₁(lower half) and E₂(higher half) Scaling

- E₁: [LOW, HIGH) in [0, 0.5)
  - LOW: 0.0xxxxxxx (binary),
  - HIGH: 0.0xxxxxxx.
  - Output 0, then shift left by 1 bit
    - [0, 0.5) → [0, 1):
      - E₁(x) = 2x

- E₂: [LOW, HIGH) in [0.5, 1)
  - LOW: 0.1xxxxxxx,
  - HIGH: 0.1xxxxxxx.
  - Output 1, subtract 0.5, shift left by 1 bit
    - [0.5, 1) → [0, 1):
      - E₂(x) = 2(x - 0.5)

- The 3rd scaling, E₃(mid), will be studied later:
  - LOW < 0.5, HIGH > 0.5, but range < 0.5.

Encoding with E₁ and E₂

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>0.02</td>
</tr>
<tr>
<td>3</td>
<td>0.18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Input 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3568</td>
<td>0.3392</td>
<td>0.0848</td>
</tr>
<tr>
<td>0.6784</td>
<td>0.3932</td>
<td>0.09632</td>
</tr>
<tr>
<td>0.3568</td>
<td>0.3568</td>
<td>0.1696</td>
</tr>
<tr>
<td>0.54112</td>
<td>0.54112</td>
<td>0.19264</td>
</tr>
</tbody>
</table>

- Encode any value in the tag, e.g., 0.5
  - Output 1
  - E₂: Output 1
  - E₂: Output 0
  - E₁: Output 0
  - E₂: Output 0
  - E₁: Output 1
  - Encode any value in the tag, e.g., 0.5

All: 1100011 (0.7734)
To verify

- LOW = 0.5424 (0.100010101... in binary),
  HIGH = 0.54816 (0.100011001... in binary).
- So we can send out 10001 (0.53125)
  - Equivalent to E2 → E1 → E1 → E1 → E2
- After left shift by 5 bits:
  - LOW = (0.5424 – 0.53125) x 32 = 0.3568
  - HIGH = (0.54816 – 0.53125) x 32 = 0.54112
  - Same as the result in the last page.

- (In this example, suppose 7 bits are enough for the decoding)

Comparison with Huffman

- Rule: Complete all possible scaling before encoding the next symbol
- Input Symbol 1 does not cause any output
- Input Symbol 3 generates 1 bit
- Input Symbol 2 generates 5 bits
  - Symbols with larger probabilities generates less number of bits.
  - Sometimes no bit is generated at all → Advantage over Huffman coding
- Large probabilities are desired in arithmetic coding
  - Can use context-adaptive method to create larger probability and to improve compression ratio.

### Symbol Prob.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>0.02</td>
</tr>
<tr>
<td>3</td>
<td>0.18</td>
</tr>
</tbody>
</table>

### Incremental Decoding

If input is 110001, the 1st symbol can be decoded without ambiguity after 5 bits are read. When 6 bits are read, the status is:

- Read 5 bits: Decode 1.
  - After reading 6 bits:
    - Tag: 110001, 0.765625
    - No scaling.

- Decode 3, E2 scaling:
  - Shift out 1 bit, read 1 bit
  - Tag: 100011 (0.546875)
  - Decode 2, E2 scaling
    - Tag: 000110 (0.09375)
    - E1: Tag: 001100 (0.1875)

- Decode 1

Encoding Pseudo Code with E1, E2

- Rule: Complete all possible scalings before further decoding. Adjust LOW, HIGH and Tag together.

### Symbol Prob. CDF

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Prob.</th>
<th>CDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>0.02</td>
<td>0.82</td>
</tr>
<tr>
<td>3</td>
<td>0.18</td>
<td>1</td>
</tr>
</tbody>
</table>

### (For floating-point implementation)

EncodeSymbol(n) {
  //Update variables
  RANGE = HIGH - LOW;
  HIGH = LOW + RANGE * CDF(n);
  LOW = LOW + RANGE * CDF(n-1);

  //Apply all possible scalings before encoding the next symbol
  while LOW, HIGH in [0, 0.5) or [0.5, 1) {
    Output 0 for E1 and 1 for E2
    scale LOW, HIGH by E1 or E2 rule
  }
}
Decoding Pseudo Code with E1, E2

```
DecodeSymbol(Tag) {
    RANGE = HIGH - LOW;
    n = 1;
    While ((tag - LOW) / RANGE >= CDF(n)) {
        n++;
    }
    HIGH = LOW + RANGE * CDF(n);
    LOW = LOW + RANGE * CDF(n-1);
    //keep scaling before decoding next symbol
    while LOW, HIGH in [0, 0.5) or [0.5, 1) {
        scale LOW, HIGH by E1 or E2 rule
        read one more bit, update Tag
    }
    return n;
}
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Prob.</th>
<th>CDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>0.02</td>
<td>0.82</td>
</tr>
<tr>
<td>3</td>
<td>0.18</td>
<td>1.0</td>
</tr>
</tbody>
</table>

(For floating-point implementation)

E3 Scaling: $[0.25, 0.75) \Rightarrow [0, 1)$

- If RANGE straddles 1/2, E1 and E2 cannot be applied, but the range can be quite small
  - Example: LOW=0.4999, HIGH=0.5001
    - Binary: LOW=0.01111..., HIGH=0.1000...
  - We may not have enough bits to represent the interval.

- E3 Scaling:
  $[0.25, 0.75) \Rightarrow [0, 1):$
  \[
  E3(x) = 2(x - 0.25)
  \]

Example

```
Example

Input 1
0
0
0
0.312
0.0848
0.1696
0.124
0.1696

Input 3
0
0
0
0.312
0.0848
0.1696
0.124
0.1696

Input 2
0.124
0.5848
0.59632
0.124
0.1696

Input 3
0.312
0.5424
0.54816
0.6
0.312
0.124
0.5848
0.59632

Input 2
0.1696
0.19264
0.1696
0.19264

With E3:

(E2: Output 1)
```

Encoding Operation with E3

```
Encoding Operation with E3

Without E3:

With E3:

Don't send anything when E3 is used, but send a 0 after E2:
  - Same output, same final state \(\Rightarrow\) Equivalent operations

```
Decoding for E3

Input 110001
Read 6 bits:
Tag: 110001 (0.765625)
Decode 1

With E3:
Tag: 10001 (0.546875)

Apply E3 whenever it is possible, everything else is same.

Summary of Different Scalings

Need E1 scaling
Need E2 scaling
Need E3 scaling
No scaling is required.
Continue to encode/decode the next symbol.

Binary Arithmetic Coding

- Arithmetic coding is slow in general:
  To decode a symbol, we need a series of decisions and multiplications:

  \[ \text{While } (\text{Tag} > \text{LOW} + \text{RANGE} \times \text{CDF}(n) / N - 1) \{
  \text{n++;}
  \}

- The complexity is greatly reduced if we have only two symbols: 0 and 1.
  - Only two intervals: [0, x), [x, 1)

Encoding of Binary Arithmetic Coding

\[ HIGH \leftarrow LOW + RANGE \times CDF(n) \]
\[ LOW \leftarrow LOW + RANGE \times CDF(n - 1) \]

LOW = 0, HIGH = 1
Prob(0)=0.6. Sequence: 0110

LOW = 0, HIGH = 0.6

LOW = 0.36, HIGH = 0.6

LOW = 0.504, HIGH = 0.6

LOW = 0.504, HIGH = 0.5616

Only need to update LOW or HIGH for each symbol.
Decoding of Binary Arithmetic Coding

General case (integer implementation):

While (Tag > LOW + RANGE * Sum(n) / N - 1) {
    n++;
}

Binary case: only one condition to check

if (Tag > LOW + RANGE * Sum(Symbol0) / N - 1) {
    n = 1;
} else {
    n = 0;
}

Applications of Binary Arithmetic Coding

- Increasingly popular:
  - JBIG, JBIG2, JPEG2000, H.264
- Convert non-binary signals into binary:
  - Golomb-Rice Code: used in H.264.
- Bit-plane coding: used in JPEG2000.
  - B = [B0, B1, ……, Bk-1]: binary representation of B.
  - Chain rule:
    - \( H(B) = H(B_0) + H(B_1 | B_0) + …… + H(B_{k-1} | B_0, … B_{k-2}) \).
  - To code \( B_0 \), needs \( P_0(0) \): Prob(\( B_0 = 0 \)).
  - To code \( B_1 \), needs \( P_1(0 | i) \): Prob(\( B_1 = 0 | B_0 = i \)), i = 0, 1.
  - … …
- More details:
  - AVC Binary Adaptive Arithmetic Coding: CABAC
  - HEVC Arithmetic Coding:

Summary

- VLC is the real world image coding solution
- Elements of Hoffman and Golomb coding schemes are incorporated
- JPEG: introduced DC prediction, AC zigzag scan, run-level VLC
- H264: introduced reverse order coding.