Image and Audio Compression

Harald Nautsch
harald.nautsch@liu.se
ISY Informationskodning, Linköpings universitet

https://icg.isy.liu.se/en/courses/TSBK38/

Lectures, preliminary program

1. Introduction. Source models, source coding.
2. Huffman coding. Entropy.
3. Golomb codes. Arithmetic coding
4. Lempel-Ziv coding. Lossless image coding.
   GIF, PNG, lossless JPEG, JPEG-LS
5. Amplitude continuous source models. Scalar quantization.
6. Vector quantization
7. Linear predictive coding
8. Transform coding. JPEG
9. Transform coding. Subband coding, wavelets. JPEG-2000
10. Psychoacoustics. Audio coding. mp3, AAC, Dolby Digital, Ogg Vorbis
11. Video coding. H.26x, MPEG
12. Video coding. Speech coding. CELP

Labs

There are 5 labs. You can work alone or in groups of two.

1. Entropy estimation for audio and still images.
   Report.
2. Lossless coding of audio and still images.
   Report.
3. Quantization.
   Report.
4. Linear predictive coding of audio.
   Report.
5. Transform coding of still images
   Done on one of the scheduled 4h times

Prerequisites

- Probability theory
- Random processes
- Linear algebra (matrices, vectors)
- Basic calculus
- Basic transform theory (DFT)
- Signal processing (linear systems)
- Matlab

Course literature

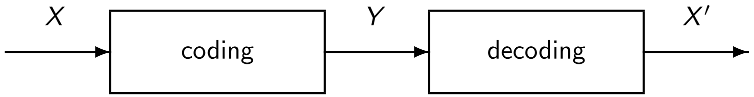
Recommended book: Khalid Sayood, *Introduction to Data Compression*. Third, fourth and fifth edition all work. The third edition is available as an electronic book.

Exercise collection, tables and formulas, see the course web pages.

Electronic books, see the course web pages.

Lab compendium, lecture slides, solutions for Sayood, et c., see the course web pages.

What is data compression?


$$\begin{tikzcd} \arrow[r, "X"] & \boxed{\text{coding}} \arrow[r, "Y"] & \boxed{\text{decoding}} \arrow[r, "X'"] & {} \end{tikzcd}$$

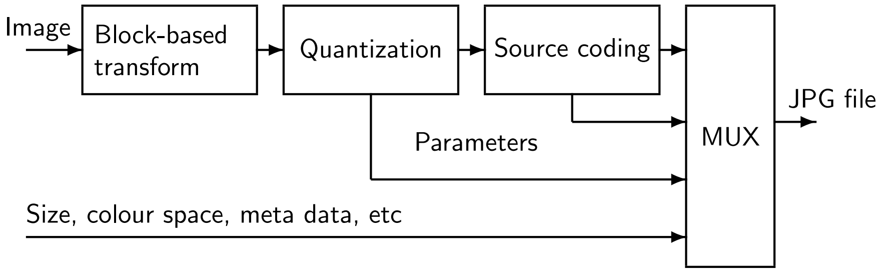
Y needs less “space” than X.

Lossless coding: $X = X'$

Lossy coding: $X \approx X'$

Example: JPEG coding

Simplified description of a JPEG still image coder:



The decoder does everything in reverse order.

Examples

Examples of lossless coding:

General:	zip, gzip, bzip, compress
Still images:	GIF, PNG, lossless JPEG, JPEG-LS, fax coding
Music:	FLAC

Examples of lossy coding:

Still images:	JPEG, JPEG 2000
Video:	H.261, H.263, H.264, HEVC, MPEG-1 (VideoCD), MPEG-2 (DVD, DVB), MPEG-4 (DivX, XviD), DV
Audio:	MPEG-1 layer 3 (mp3), AC-3 (Dolby Digital), Ogg Vorbis, AAC, ATRAC (MiniDisc)
Speech:	CELP, GSM

Performance measures

How much *compression* do we get from our method?

Compression ratio

Ex. Signal to code: grayscale image 256×256 pixels, 1 byte (8 bits) per pixels. Suppose the coded signal is 16384 bytes.
The compressions ratio is then $\frac{256 \cdot 256 \cdot 1}{16384} = 4$.

Average data rate in bits/symbol (bits/pixel, bits/sample)

Ex. Our image above: $\frac{16384 \cdot 8}{256 \cdot 256} = 2$ bits/pixel

For video and audio signals the rate is often given as bits/s.

Performance measures, cont.

How much *distortion* do we get?

Human evaluation

Mathematical measures

- Mean square error
- SNR
- SNR weighted to mimic human vision or hearing.

What is the original?

For audio and images the original signals are *sampled* and *finely quantized* amplitude signals.

The signal can be either scalar (mono sound, grayscale images) or vector valued (RGB, CMYK, multispectral images, stereo sound, surround sound).

Even though the original signal is almost always quantized we can still often use amplitude continuous models for it (lecture 5 and later.)

Properties to use

What properties of the signal can we use to achieve compression?

- All symbols are not equally common. For instance in a music signal small amplitude values are more common than large amplitude values. A good code will have short descriptions for common values and longer descriptions for uncommon values.
- Dependence between symbols (samples, pixels). For instance in an image two pixels next to each other usually have almost the same value. A good coding algorithm will take advantage of this dependence.
- Properties of the human vision or hearing system. We can remove information that a human can not see or hear anyway.

Discrete sources

A *source* is something that produces a sequence of *symbols*.

The symbols are elements in a discrete *alphabet* $\mathcal{A} = \{a_1, a_2, \ldots, a_L\}$ of size L.

We will mostly deal with finite alphabets, but infinite alphabets can also be used.

In most cases we only have access to a symbol sequence generated by the source and we will have to model the source from the given sequence.

Random source models

The source models we will focus on are *random models*, where we assume that the symbols are generated by random variables or random processes.

The simplest random model for a source is a discrete random variable X.

Distribution

$$Pr(X = a_i) = P_X(a_i) = P(a_i) = p_i$$

$$P(a_i) \geq 0 \ , \quad \forall a_i$$

$$\sum_{i=1}^{L} P(a_i) = 1$$

Random source models, cont.

Better source models: discrete stationary random processes.

A random process X_t can be viewed as a sequence of random variables, where we get an outcome in each time instance t.

Conditional probabilities:
The output of the source at two times t and s

$$P(x_t, x_s) = Pr(X_t = x_t, X_s = x_s)$$

$$P(x_s|x_t) = \frac{P(x_t, x_s)}{P(x_t)}$$

$$P(x_t, x_s) = P(x_t) \cdot P(x_s|x_t)$$

Memory sources

Dependence between the signal at different times is called *memory*.

If X_t and X_{t+k} are independent for all $k \neq 0$ the source is *memoryless*.

For a memoryless source we have:

$$P(x_t, x_{t+k}) = P(x_t) \cdot P(x_{t+k})$$

$$P(x_{t+k} | x_t) = P(x_{t+k})$$

Markov sources

A *Markov source* of order k is a memory source with limited memory k steps back in the sequence.

$$P(x_n|x_{n-1}x_{n-2}\ldots) = P(x_n|x_{n-1}\ldots x_{n-k})$$

If the alphabet is $\mathcal{A} = \{a_1, a_2, \ldots, a_L\}$, the Markov source can be described as a state model with L^k states $(x_{n-1}\ldots x_{n-k})$ where we at time n move from state $(x_{n-1}\ldots x_{n-k})$ to state $(x_n\ldots x_{n-k+1})$ with probability $P(x_n|x_{n-1}\ldots x_{n-k})$. These probabilities are called *transition probabilities*

The sequence of states is a random process $S_n = (X_n, X_{n-1}\ldots, X_{n-k+1})$ with alphabet $\{s_1, s_2, \ldots, s_{L^k}\}$ of size L^k.

Markov sources, cont.

The Markov source can be described using its starting state and its *transition matrix* $\mathbf{P}$. This quadratic matrix has in row r and column k the transition probability from state s_r to s_k.

If it is possible to move, with positive probability, from every state to every other state in a finite number of steps, the Markov source is called *irreducible*.

If we at time n are in state s_i with the probability p_i^n, we can calculate the probabilities for time $n+1$ as

$$(p_1^{n+1} \; p_2^{n+1} \; \ldots \; p_{L^k}^{n+1}) = (p_1^n \; p_2^n \; \ldots \; p_{L^k}^n) \cdot \mathbf{P}$$

A distribution over the states such that the distribution at time $n+1$ is the same as at time n is called a *stationary distribution*.

If the Markov source is irreducible and aperiodic the stationary distribution is unique and every starting distribution will approach the stationary distribution as the time goes to infinity.

Stationary distribution

We denote the stationary probabilities w_i and define the row vector

$$\bar{w} = (w_1, w_2, \ldots, w_{L^k})$$

If the stationary distribution exists, it can be found as the solution of the equation system

$$\bar{w} = \bar{w} \cdot \mathbf{P}$$

or

$$\bar{w} \cdot (\mathbf{P} - \mathbf{I}) = \bar{0}$$

This equation system is under-determined (if $\bar{w}$ is a solution then $c \cdot \bar{w}$ is also a solution). To find the correct solution we add the equation $\sum_{j=1}^{L^k} w_j = 1$ (w_j are probabilities and therefore their sum is 1).

(If you prefer equation systems with column vectors, you can just transpose the entire expression and solve $\bar{w}^T = \mathbf{P}^T \cdot \bar{w}^T$ instead.)

Random modeling

Give a long symbol sequence from a source, how do we make a random model for it?

Relative frequencies: To get the probability for a symbol, count the number of times that symbol appears and divide by the total number of symbols in the sequence. In the same way this can be done for pair probabilities, triple probabilities, conditional probabilities et c.

Random model from given sequence

Example: Alphabet $\{a, b\}$. Given data: *bbbbaabbbaaaaabbbbbabaaabbbb*.

To estimate the symbol probabilities we count how often each symbol appears: *a* appears 11 times, *b* 17 times. The estimated probabilities $P(x_t)$ are then:

$$P(a) = \frac{11}{28}, \quad P(b) = \frac{17}{28}$$

For pair probabilities and conditional probabilities we instead count how often the different symbol pairs appear. *aa* appears 7 times, *ab* 4 times, *ba* 4 times and *bb* 12 times. The estimated probabilities $P(x_t, x_{t+1})$ and $P(x_{t+1}|x_t)$ are:

$$P(aa) = \frac{7}{27}, \quad P(ab) = \frac{4}{27}, \quad P(ba) = \frac{4}{27}, \quad P(bb) = \frac{12}{27}$$
$$P(a|a) = \frac{7}{11}, \quad P(b|a) = \frac{4}{11}, \quad P(a|b) = \frac{4}{16}, \quad P(b|b) = \frac{12}{16}$$

Source coding

Source coding means mapping sequences of symbols from a source alphabet onto binary sequences (called *codewords*).

The set of all codewords is called a *code*.

A code where all the codewords have the same length (number of bits) is called a *fixed-length code*.

Example: $\mathcal{A} = \{a, b, c, d\}$

Symbol	Code 1	Code 2	Code 3	Code 4	Code 5
a	00	0	0	0	0
b	01	0	1	10	01
c	10	1	00	110	011
d	11	10	11	111	111

Source coding

Code the sequence *abbacddcd* using our five codes

Symbol	Code 1	Code 2	Code 3	Code 4	Code 5
a	00	0	0	0	0
b	01	0	1	10	01
c	10	1	00	110	011
d	11	10	11	111	111

Code 1:	000101001011111011
Code 2:	000011010110
Code 3:	01100011110011
Code 4:	01010011011111110111
Code 5:	00101001111111011111

Properties of codes

If you from any sequence of codewords can recreate the original symbol sequence, the code is called *uniquely decodable*.

If you can recognize the codewords directly while decoding, the code is called *instantaneous*.

If no codeword is a prefix to another codeword, the code is called a *prefix code* (in some literature they are called *prefix free* codes). These codes are *tree codes*, ie each codeword can be described as the path from the root to a leaf in a binary tree.

All prefix codes are instantaneous and all instantaneous codes are prefix codes.

Example

Example, $\mathcal{A} = \{a, b, c, d\}$

Symbol	Code 1	Code 2	Code 3	Code 4	Code 5
a	00	0	0	0	0
b	01	0	1	10	01
c	10	1	00	110	011
d	11	10	11	111	111

Code 1	Uniquely decodable, instantaneous (tree code)
Code 2	Not uniquely decodable
Code 3	Not uniquely decodable
Code 4	Uniquely decodable, instantaneous (tree code)
Code 5	Uniquely decodable, not instantaneous

Uniquely decodable or not?

How can you check if a given code is uniquely decodable or not?
Make a list of all codewords. Examine every pair of elements in the list to see if any element is a prefix to another element. In that case add the suffix to the list, if it's not already in the list. Repeat until one of two things happen:

1. You find a suffix that is a codeword.
2. You find no more new suffixes to add to the list.

In case 1 the code is not uniquely decodable, in case 2 the code is uniquely decodable.

Code performance

How good a code is is determined by its *mean data rate R* (usually just rate or data rate) in bits/symbol.

$$R = \frac{\text{average number of bits per codeword}}{\text{average number of symbols per codeword}}$$

Since we're doing compression we want R to be as small as possible.

For a random source, there is a theoretical lower bound on the rate.

Note that R is a measure of how good the code is *on average* over all possible sequences from the source. It tells us nothing of how good the code is for a particular sequence.

Kraft's inequality, mean codeword length

An instantaneous code (prefix code, tree code) with the codeword lengths $l_1, \ldots, l_L$ exists if and only if

$$\sum_{i=1}^{L} 2^{-l_i} \leq 1$$

The inequality also holds for all uniquely decodable codes. It is then called Kraft-McMillan's inequality.
Mean codeword length:

$$\bar{l} = \sum_{i=1}^{L} p_i \cdot l_i \quad \text{[bits/codeword]}$$

if we code one symbol with each codeword we have

$$R = \bar{l}$$

Entropy as a lower bound

There is a lower bound on the mean codeword length of a uniquely decodable code:

$$\bar{l} \geq -\sum_{i=1}^{L} p_i \cdot \log_2 p_i = H(X_t)$$

$H(X_t)$ is the *entropy* of the source (more on entropy later).

Entropy as a lower bound, cont

Proof of $\bar{l} \geq H(X_t)$

$$\begin{aligned}
H(X_t) - \bar{l} &= -\sum_{i=1}^{L} p_i \cdot \log_2 p_i - \sum_{i=1}^{L} p_i \cdot l_i = \sum_{i=1}^{L} p_i \cdot (\log_2 \frac{1}{p_i} - l_i) \\
&= \sum_{i=1}^{L} p_i \cdot (\log_2 \frac{1}{p_i} - \log_2 2^{l_i}) = \sum_{i=1}^{L} p_i \cdot \log_2 \frac{2^{-l_i}}{p_i} \\
&\leq \frac{1}{\ln 2} \sum_{i=1}^{L} p_i \cdot (\frac{2^{-l_i}}{p_i} - 1) = \frac{1}{\ln 2} (\sum_{i=1}^{L} 2^{-l_i} - \sum_{i=1}^{L} p_i) \\
&\leq \frac{1}{\ln 2}(1 - 1) = 0
\end{aligned}$$

where we used the inequality $\ln x \leq x - 1$ and Kraft-McMillan's inequality.

Optimal codes

A code is called *optimal* if no other code exists (for the same probability distribution) that has a lower mean codeword length.

There are of course several codes with the same mean codeword length. The simplest example is to just switch all ones to zeros and all zeros to ones in the codewords.

Even codes with different sets of codeword lengths can have the same mean codeword length.

Upper bound for optimal codes

Given that we code one symbol at a time, an optimal code satisfies $\bar{l} < H(X_t) + 1$

Let $l_i = \lceil -\log p_i \rceil$. We have that $-\log p_i \leq \lceil -\log p_i \rceil < -\log p_i + 1$.

$$\begin{aligned} \sum_{i=1}^{L} 2^{-l_i} &= \sum_{i=1}^{L} 2^{-\lceil -\log p_i \rceil} \\ &\leq \sum_{i=1}^{L} 2^{\log p_i} \\ &= \sum_{i=1}^{L} p_i \; = \; 1 \end{aligned}$$

Kraft's inequality is satisfied, therefore a tree code with the given codeword lengths exists.

Upper bound for optimal codes, cont.

What's the mean codeword length of this code?

$$\begin{aligned}\bar{l} &= \sum_{i=1}^{L} p_i \cdot l_i = \sum_{i=1}^{L} p_i \cdot \lceil -\log p_i \rceil \\ &< \sum_{i=1}^{L} p_i \cdot (-\log p_i + 1) \\ &= -\sum_{i=1}^{L} p_i \cdot \log p_i + \sum_{i=1}^{L} p_i = H(X_t) + 1\end{aligned}$$

An optimal code can't be worse than this code, then it wouldn't be optimal. Thus, the mean codeword length for an optimal code also satisfies $\bar{l} < H(X_t) + 1$.

NOTE: If $p_i = 2^{-k_i}, \forall i$ for integers k_i, we can construct a code with codeword lengths k_i and $\bar{l} = H(X_t)$.

Huffman coding

A simple method for constructing optimal tree codes.

Start with symbols as leaves.

In each step connect the two least probable nodes to an inner node. The probability for the new node is the sum of the probabilities of the two original nodes. If there are several nodes with the same probability to choose from it doesn't matter which ones we choose.

When we have constructed the whole code tree, we create the codewords by setting 0 and 1 on the branches in each node. Which branch that is set to 0 and which that is set to 1 doesn't matter.

Extended codes

For small alphabets with skewed distributions, or sources with memory, a Huffman code can be relatively far from the entropy bound. This can be solved by *extending* the source, ie by coding multiple symbols with each codeword.

If we code n symbols with each codeword, and the code has the mean codeword length $\bar{l}$ the rate will be

$$R = \frac{\bar{l}}{n}$$

The maximal redundancy of an optimal code (the difference between the rate and the entropy) is $\frac{1}{n}$ when we code n symbols at a time.

Side information

So far we have only considered the rate R of the code to determine how good or bad it is. In a practical application we also have to consider that we have to transmit *side information* for the receiver to be able to decode the sequence.

For instance we might have to tell the receiver what alphabet is used, what the code tree looks like and how long the decoded sequence should be. Exactly what side information needs to be transmitted depends on the situation.

A straightforward method of sending a code tree: For each symbol in the alphabet we first send the codeword length and then the actual codeword. With an alphabet of size L we will need $L \cdot \lceil \log L \rceil + \sum_i l_i$ extra bits.

Side information, cont.

Given the codeword lengths (and hence the rate) it's usually not important exactly what codewords that are used. If both the coder and the decoder uses the same algorithm to construct a code given the codeword lengths, we only need to send the lengths.
In practice the coder will use the Huffman algorithm to construct an optimal code. The coder notes the codeword lengths, throws away the code and constructs a new code with these lengths. The new code will have the same rate as the first code. The decoder can construct the same code. In this case we only need

$$L \cdot \lceil \log L \rceil$$

bits of side information.

If the amount of data to be sent is large the side information will only have a small affect on the total data rate, but if the amount of data is small the side information can be a substantial portion of the total rate.

Runlength coding

Sometimes we have sources that produce long partial sequences consisting of the same symbol. It can then be practical to view the sequence as consisting of *runs* instead of symbols. A run is a tuple describing what symbol that is in the run and how long the run is.

For example, the sequence

$$aaaabbbbbbbccbbbbaaaa$$

can be described as

$$(a, 4)(b, 7)(c, 2)(b, 4)(a, 4)$$

Basically we have switched to another alphabet than the original one.

The gain is that it might be easier to find a good code for the new alphabet, and that it's easier to take advantage of the memory of the source.

Golomb codes

$\mathcal{A} = \{0, 1, 2, \ldots\}$

Choose the parameter m. In practice, m is usually chosen to be an integer power of two, but it can be any positive integer.

Represent the integer n with $q = \lfloor \frac{n}{m} \rfloor$ and $r = n - qm$.

Code q with a unary code.

If m is an integer power of two, code r binary with $\log m$ bits.

If m is not an integer power of two:

$0 \leq r < 2^{\lceil \log m \rceil} - m$	Code r binary with $\lfloor \log m \rfloor$ bits
$2^{\lceil \log m \rceil} - m \leq r \leq m - 1$	Code $r + 2^{\lceil \log m \rceil} - m$ binary with $\lceil \log m \rceil$ bits

Examples of Golomb codes

Symbol	$m = 1$	$m = 2$	$m = 3$	$m = 4$
0	0	0 0	0 0	0 00
1	10	0 1	0 10	0 01
2	110	10 0	0 11	0 10
3	1110	10 1	10 0	0 11
4	11110	110 0	10 10	10 00
5	111110	110 1	10 11	10 01
6	1111110	1110 0	110 0	10 10
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Golomb codes are optimal for distributions of the type

$$p(i) = q^i \cdot (1 - q) \quad ; \quad 0 < q < 1$$

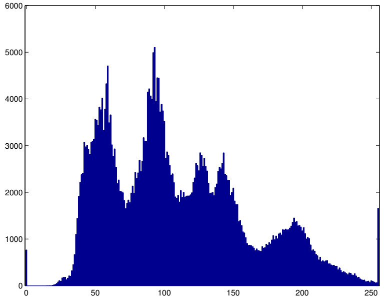
if we choose $m = \lceil -\frac{1}{\log q} \rceil$
Golomb codes are for instance used in the image coding standard JPEG-LS and in the video coding standard H.264.

Example, image coding

Original image, 768×512 pixels, 8 bits per pixel

Huffman coding

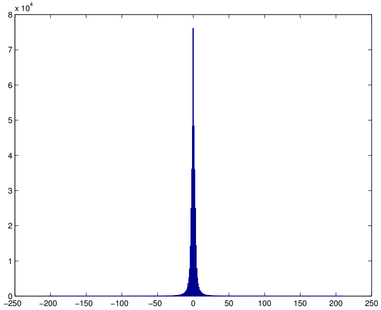
We use the histogram (ie the relative frequencies) as an estimate of the probability distribution over the pixel values.



A Huffman code for this distribution will code the image using 2954782 bits, plus 1280 bits of side information. This gives us a rate of 7.51 bits/pixel.

Huffman coding of pixel differences

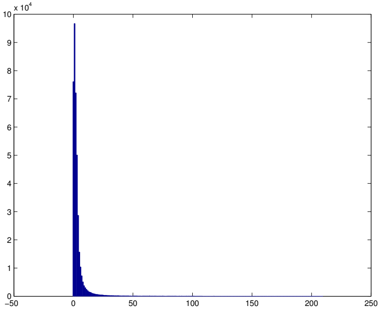
To take advantage of the memory we take the difference between a pixel and the pixel above it. Pixels outside the image area are assumed to be 128.



A Huffman code for this distribution will code the image using 1622787 bits, plus 2075 bits of side information. This gives us a rate of 4.13 bits/pixel.

Golomb coding of pixel differences I

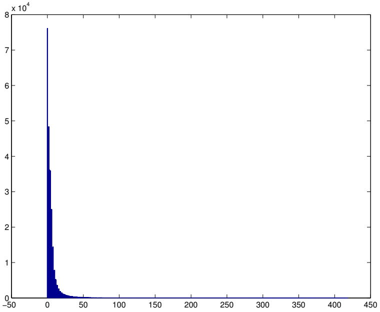
In order to do Golomb coding of the pixel differences we need to have an alphabet of non-negative values. For example we can code the absolute value of the differences using a Golomb code and then send a sign bit for each non-zero value.



The best Golomb code for this distribution is $m = 3$. This gives us 1438806 bits, 317104 sign bits and 8 bits of side information. The rate is 4.47 bits/pixel.

Golomb coding of pixel differences II

Another variant is to represent positive values as even numbers and negative values as odd numbers.



The best Golomb code for this distribution is $m = 3$. This gives us 1760618 bits and 8 bits of side information. The rate is 4.48 bits/pixel.

Information and entropy

X is a discrete random variable.

Distribution

$$p_i = P(a_i) = Pr(X = a_i)$$

Self information of the outcomes

$$i(a_i) = -\log p_i$$

The logarithm can be taken in any base. For practical reasons, base 2 is usually used. The unit is then called *bits*. We can then easily compare the entropy with the rate of a binary code.

The lower the probability of an outcome is, the larger the information of the outcome is

$$p_i \to 0 \implies i(a_i) \to \infty$$

$$p_i = 1 \implies i(a_i) = 0$$

Entropy

The mean value of the information is called *entropy*.

$$H(X) = \sum_{i=1}^{L} p_i \cdot i(a_i) = -\sum_{i=1}^{L} p_i \cdot \log\ p_i$$

The entropy can be seen as a measure of the average information in X, or a measure of the uncertainty of X.

$$0 \leq H(X) \leq \log\ L$$

The entropy is maximized when all outcomes are equally probable.

If any one of the outcomes has probability 1 (and thus all the other outcomes have probability 0) the entropy is 0, ie there is no uncertainty.

Entropy, cont.

Two random variables X and Y with alphabets $\mathcal{A}$ and $\mathcal{B}$.

$$P_{XY}(a_i, b_j) = P_X(a_i) \cdot P_{Y|X}(b_j|a_i) = P_Y(b_j) \cdot P_{X|Y}(a_i|b_j)$$

Conditional entropy

$$H(Y|X) = -\sum_{i,j} P_{XY}(a_i, b_j) \cdot \log P_{Y|X}(b_j|a_i)$$

$H(Y|X) \leq H(Y)$ with equality if X and Y are independent.

Block entropy

$$H(X, Y) = -\sum_{i,j} P_{XY}(a_i, b_j) \cdot \log P_{XY}(a_i, b_j)$$

$$H(X, Y) = H(X) + H(Y|X) = H(Y) + H(X|Y)$$

Entropy of sources

Source X_t (stationary random process)
First order entropy of the source

$$H(X_t) = \sum_{i=1}^{L} p_i \cdot i(a_i) = -\sum_{i=1}^{L} p_i \cdot \log\ p_i$$

Conditional entropy

$$H(X_t|X_{t-1}) \leq H(X_t)$$

with equality if X_t is memoryless.

Second order entropy

$$H(X_{t-1}, X_t) = H(X_{t-1}) + H(X_t|X_{t-1}) \leq 2 \cdot H(X_t)$$

n-th order entropy

$$H(X_1, \ldots, X_n) = H(X_1) + H(X_2|X_1) + \ldots + H(X_n|X_1 \ldots X_{n-1}) \leq n \cdot H(X_t)$$

Entropy of sources, cont.

Entropy of the source (also called *entropy rate*).

$$\lim_{n\to\infty} \frac{1}{n} H(X_1 \ldots X_n) = \lim_{n\to\infty} H(X_n|X_1 \ldots X_{n-1})$$

For a memoryless source the entropy rate is equal to the first order entropy.

For a Markov source of order k the entropy rate is

$$H(X_t|X_{t-1} \ldots X_{t-k})$$

The entropy rate gives a lower bound on the data rate of a uniquely decodable code for the source.

Problems with Huffman coding

Huffman coding is optimal in theory, but it can be impractical to use for skewed distributions and/or when extending the source.
Ex: $\mathcal{A} = \{a, b, c\}, \quad P(a) = 0.95, P(b) = 0.02, P(c) = 0.03$

The entropy of the source is approximately 0.3349. The mean codeword length of a Huffman code is 1.05 bits/symbol, ie more than 3 times the entropy. If we want the rate to be no more than 5% larger than the entropy we have to extend the source and code 8 symbols at a time. This gives a Huffman code with $3^8 = 6561$ codewords.

We would like to have coding method where we can directly find the codeword for a given sequence, without having to determine the codewords for *all* possible sequences. One way of doing this is *arithmetic coding.*

Arithmetic coding

Suppose that we have a source X_t taking values in the alphabet $\{1, 2, \ldots, L\}$. Suppose that the probabilities for all symbols are strictly positive: $P(i) > 0, \ \forall i$.

The cumulative distribution function $F(i)$ is defined as

$$F(i) = \sum_{k \leq i} P(k)$$

$F(i)$ is a step function where the step in k has the height $P(k)$.

Example:
$\mathcal{A} = \{1, 2, 3\}$
$P(1) = 0.5, \ P(2) = 0.3, \ P(3) = 0.2$
$F(0) = 0, \ F(1) = 0.5, \ F(2) = 0.8, \ F(3) = 1$

Arithmetic coding

Suppose that we want to code a sequence $\mathbf{x} = x_1, x_2, \ldots, x_n$.
Start with the whole probability interval $[0, 1)$. In each step j divide the interval proportional to the cumulative distribution $F(i)$ and choose the subinterval corresponding to the symbol x_j that is to be coded.

If we have a memory source the intervals are divided according to the conditional cumulative distribution function.

Each symbol sequence of length n uniquely identifies a subinterval. The codeword for the sequence is a number in the interval. The number of bits in the codeword depends on the interval size, so that a large interval (ie a sequence with high probability) gets a short codeword, while a small interval gives a longer codeword.

Iterative algorithm

Suppose that we want to code a sequence $\mathbf{x} = x_1, x_2, \ldots, x_n$. We denote the lower limit in the corresponding interval by $l^{(n)}$ and the upper limit by $u^{(n)}$. The interval generation is the given iteratively by

$$\begin{cases} l^{(j)} = l^{(j-1)} + (u^{(j-1)} - l^{(j-1)}) \cdot F(x_j - 1) \\ u^{(j)} = l^{(j-1)} + (u^{(j-1)} - l^{(j-1)}) \cdot F(x_j) \end{cases}$$

Starting values are $l^{(0)} = 0$ and $u^{(0)} = 1$.

The interval size is of course equal to the probability of the sequence

$$u^{(n)} - l^{(n)} = P(\mathbf{x})$$

Codeword

The codeword for an interval is the shortest bit sequence $b_1 b_2 \dots b_k$ such that the binary number $0.b_1 b_2 \dots b_k$ is in the interval and that all other numbers staring with the same k bits are also in the interval.
Given a binary number a in the interval $[0, 1)$ with k bits $0.b_1 b_2 \dots b_k$.
All numbers that have the same k first bits as a are in the interval $[a, a + \frac{1}{2^k})$.
A necessary condition for all of this interval to be inside the interval belonging to the symbol sequence is that it is less than or equal in size to the symbol sequence interval, ie

$$P(\mathbf{x}) \geq \frac{1}{2^k} \quad \Rightarrow \quad k \geq \lceil -\log P(\mathbf{x}) \rceil$$

We can't be sure that it is enough with $\lceil -\log P(\mathbf{x}) \rceil$ bits, since we can't place these intervals arbitrarily. We can however be sure that we need at most one extra bit The codeword length $l(\mathbf{x})$ for a sequence $\mathbf{x}$ is thus given by

$$l(\mathbf{x}) = \lceil -\log P(\mathbf{x}) \rceil \quad \text{or} \quad l(\mathbf{x}) = \lceil -\log P(\mathbf{x}) \rceil + 1$$

Average codeword length

$$\begin{aligned}
\bar{l} &= \sum_{\mathbf{x}} P(\mathbf{x}) \cdot l(\mathbf{x}) \leq \sum_{\mathbf{x}} P(\mathbf{x}) \cdot (\lceil -\log P(\mathbf{x}) \rceil + 1) \\
&< \sum_{\mathbf{x}} P(\mathbf{x}) \cdot (-\log P(\mathbf{x}) + 2) = -\sum_{\mathbf{x}} P(\mathbf{x}) \cdot \log P(\mathbf{x}) + 2 \cdot \sum_{\mathbf{x}} P(\mathbf{x}) \\
&= H(X_1 X_2 \ldots X_n) + 2
\end{aligned}$$

The resulting data rate is thus bounded by

$$R < \frac{1}{n} H(X_1 X_2 \ldots X_n) + \frac{2}{n}$$

This is a little worse than the rate for an extended Huffman code, but extended Huffman codes are not practical for large n. The complexity of an arithmetic coder, on the other hand, is independent of how many symbols n that are coded. In arithmetic coding we only have to find the codeword for a particular sequence and not for all possible sequences.

Memory sources

When doing arithmetic coding of memory sources, we let the interval division depend on earlier symbols, ie we use different F in each step depending on the value of earlier symbols.

For example, if we have a binary Markov source X_t of order 1 with alphabet $\{1,\ 2\}$ and transition probabilities $P(x_t|x_{t-1})$

$$P(1|1) = 0.8, \quad P(2|1) = 0.2, \quad P(1|2) = 0.1, \quad P(2|2) = 0.9$$

we will use two conditional cumulative distribution functions $F(x_t|x_{t-1})$

$$F(0|1) = 0, \quad F(1|1) = 0.8, \quad F(2|1) = 1$$

$$F(0|2) = 0, \quad F(1|2) = 0.1, \quad F(2|2) = 1$$

For the first symbol in the sequence we can either choose one of the two distributions or use a third cumulative distribution function based on the stationary probabilities.

Practical problems

When implementing arithmetic coding we have a limited precision and can't store interval limits and probabilities with aribtrary resolution.

We want to start sending bits without having to wait for the whole sequence with n symbols to be coded.

One solution is to send bits as soon as we are sure of them and to rescale the interval when this is done, to maximally use the available precision.

If the first bit in both the lower and the upper limits are the same then that bit in the codeword must also take this value. We can send that bit and thereafter shift the limits left one bit, ie scale up the interval size by a factor 2.

Fixed point arithmetic

Arithmetic coding is most often implemented using fixed point arithmetic.

Suppose that the interval limits l and u are stored as integers with m bits precision and that the cumulative distribution function $F(i)$ is stored as integers with k bits precision. The algorithm can then be modified to

$$l^{(j)} = l^{(j-1)} + \lfloor \frac{(u^{(j-1)} - l^{(j-1)} + 1)F(x_j - 1)}{2^k} \rfloor$$

$$u^{(j)} = l^{(j-1)} + \lfloor \frac{(u^{(j-1)} - l^{(j-1)} + 1)F(x_j)}{2^k} \rfloor - 1$$

Starting values are $l^{(0)} = 0$ and $u^{(0)} = 2^m - 1$.

Note that previosuly when we had continuous intervals, the upper limit pointed to the first number in the next interval. Now when the intervals are discrete, we let the upper limit point to the last number in the current interval.

Interval scaling

The cases when we should perform an interval scaling are:

1. The interval is completely in $[0, 2^{m-1} - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 0. Shift out the most significant bit of $l^{(j)}$ and $u^{(j)}$ and send it. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$.
2. The interval is completely in $[2^{m-1}, 2^m - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 1. Shift out the most significant bit of $l^{(j)}$ and $u^{(j)}$ and send it. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$. The same operations as in case 1.

When we have coded our n symbols we finish the codeword by sending all m bits in $l^{(n)}$. The code can still be a prefix code with fewer bits, but the implementation of the decoder is much easier if all of l is sent. For large n the extra bits are neglible. We probably need to pack the bits into bytes anyway, which might require padding.

More problems

Unfortunately we can still get into trouble in our algorithm, if the first bit of l always is 0 and the first bit of u always is 1. In the worst case scenario we might end up in the situation that $l = 011\dots11$ and $u = 100\dots00$. Then our algorithm will break down.

Fortunately there are ways around this. If the first two bits of l are 01 and the first two bits of u are 10, we can perform a bit shift, without sending any bits of the codeword. Whenever the first bit in both l and u then become the same we can, besides that bit, also send one extra inverted bit because we are then sure of if the codeword should have 01 or 10.

Interval scaling

We now get three cases

1. The interval is completely in $[0, 2^{m-1} - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 0. Shift out the most significant bit of $l^{(j)}$ and $u^{(j)}$ and send it. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$.
2. The interval is completely in $[2^{m-1}, 2^m - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 1. The same operations as in case 1.
3. We don't have case 1 or 2, but the interval is completely in $[2^{m-2}, 2^{m-1} + 2^{m-2} - 1]$, ie the two most significant bits are 01 in $l^{(j)}$ and 10 in $u^{(j)}$. Shift out the most significant bit from $l^{(j)}$ and $u^{(j)}$. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$. Invert the new most significant bit in $l^{(j)}$ and $u^{(j)}$. Don't send any bits, but keep track of how many times we do this kind of rescaling. The next time we do a rescaling of type 1, send as many extra ones as the number of type 3 rescalings. In the same way, the next time we do a rescaling of type 2 we send as many extra zeros as the number of type 3 rescalings.

Demands on the precision

We must use a datatype with at least $m + k$ bits to be able to store partial results of the calculations.

We also see that the smallest interval we can have without performing a rescaling is of size $2^{m-2} + 1$, which for instance happens when $l^{(j)} = 2^{m-2} - 1$ and $u^{(j)} = 2^{m-1}$. For the algorithm to work, $u^{(j)}$ can never be smaller than $l^{(j)}$ (the same value is allowed, because when we do a rescaling we shift zeros into l and ones into u). In order for this to be true, all intervals of the fixed point version of the cumulative distribution function must fulfill (with a slight overestimation)

$$F(i) - F(i-1) \geq 2^{k-m+2} \quad ; \quad i = 1, \ldots, L$$

Decoding

Start the decoder in the same state (ie $l = 0$ and $u = 2^m - 1$) as the coder. Introduce t as the m first bits of the bit stream (the codeword). At each step we calculate the number

$$\lfloor \frac{(t - l + 1) \cdot 2^k - 1}{u - l + 1} \rfloor$$

Compare this number to F to see what probability interval this corresponds to. This gives one decoded symbol. Update l and u in the same way that the coder does. Perform any needed shifts (rescalings). Each time we rescale l and u we also update t in the same way (shift out the most significant bit, shift in a new bit from the bit stream as new least significant bit. If the rescaling is of type 3 we invert the new most significant bit.) Repeat until the whole sequence is decoded.

Note that we need to send the number of symbols in the codeword as side information, so the decoder knows when to stop decoding. Alternatively we can introduce an extra symbol into the alphabet, with lowest possible probability, that is used to mark the end of the sequence.

Lempel-Ziv coding

Code symbol sequences using references to previous data in the sequence.

There are two main types:

- Use a history buffer, code a partial sequence as a pointer to when that particular sequence last appeared (LZ77).
- Build a dictionary of all unique partial sequences that appears. The codewords are references to earlier words (LZ78).

Lempel-Ziv coding, cont.

The coder and decoder don't need to know the statistics of the source. Performance will asymptotically reach the entropy rate. A coding method that has this property is called *universal*.

Lempel-Ziv coding in all its different variants are the most popular methods for file compression and archiving, eg zip, gzip, ARJ and compress.

The image coding standards GIF and PNG use Lempel-Ziv.

The standard V.42bis for compression of modem traffic uses Lempel-Ziv.

LZ77

Lempel and Ziv 1977.

View the sequence to be coded through a sliding window. The window is split into two parts, one part containing already coded symbols (search buffer) and one part containing the symbols that are about to coded next (look-ahead buffer).

Find the longest sequence in the search buffer that matches the sequence that starts in the look-ahead buffer. The codeword is a triple $< o, l, c >$ where o is a pointer to the position in the search buffer where we found the match (offset), l is the length of the sequence, and c is the next symbol that doesn't match. This triple is coded using a fixlength codeword. The number of bits required is

$$\lceil \log S \rceil + \lceil \log(W+1) \rceil + \lceil \log L \rceil$$

where S is the size of the search buffer, W is the size of the look-ahead buffer and L is the alphabet size.

Improvements of LZ77

It is unnecessary to send a pointer and a length if we don't find a matching sequence. We only need to send a new symbol if we don't find a matching sequence. Instead we can use an extra flag bit that tells if we found a match or not. We either send $< 1, o, l >$ or $< 0, c >$. This variant of LZ77 is called LZSS (Storer and Szymanski, 1982).

Depending on buffer sizes and alphabet sizes it can be better to code short sequences as a number of single symbols instead of as a match.

In the beginning, before we have filled up the search buffer, we can use shorter codewords for o and l.

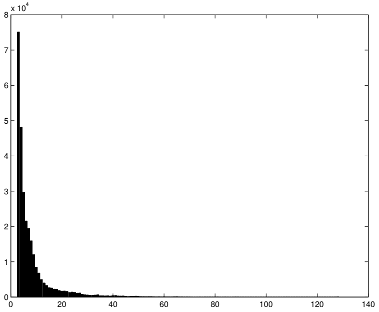
All o, l and c are not equally probable, so we can get even higher compression by coding them using variable length codes (eg Huffman codes).

Buffer sizes

In principle we get higher compression for larger search buffers. For practical reasons, typical search buffer sizes used are around $2^{15} - 2^{16}$.

Very long match lengths are usually not very common, so it is often enough to let the maximum match length (ie the look-ahead buffer size) be a couple of hundred symbols.

Example: LZSS coding of a text file, buffer size 32768, match lengths 3-130 (128 possible values). Histogram for match lengths:



DEFLATE

Deflate is a variant of LZ77 that uses Huffman coding. It is the method used in zip, gzip and PNG.

Data to be coded are bytes.

The data is coded in blocks of arbitrary size (with the exception of uncompressed blocks that can be no more than 65536 bytes).

The block is either sent uncompressed or coded using LZ and Huffman coding.

The match lengths are between 3 and 258. Offset can be between 1 and 32768.

The Huffman coding is either fixed (predefined codewords) or dynamic (the codewords are sent as side information).

Two Huffman codes are used: one code for single symbols (literals) and match lengths and one code for offsets.

Symbols and lengths

The Huffman code for symbols and lengths use the alphabet $\{0, 1, \ldots, 285\}$ where the values 0-255 are used for symbols, the value 256 marks the end of a block and the values 257-285 are used to code lengths together with extra bits:

	extra bits	length		extra bits	length		extra bits	length
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Offset

The Huffman code for offset uses the alphabet $\{0, \ldots, 29\}$. Extra bits are used to exactly specify offset 5-32768

	extra bits	offset		extra bits	offset		extra bits	offset
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

Fixed Huffman codes

Codewords for the symbol/length alphabet:

value	number of bits	codewords
0 - 143	8	00110000 - 10111111
144 - 255	9	110010000 - 111111111
256 - 279	7	0000000 - 0010111
280 - 287	8	11000000 - 11000111

The reduced offset alphabet is coded with a five bit fixlength code.

For example, a match of length 116 at offset 12 is coded with the codewords 11000000 0001 and 00110 11.

Dynamic Huffman codes

The codeword lengths for the different Huffman codes are sent as extra information.

To get even more compression, the sequence of codeword lengths are runlength coded and then Huffman coded (!). The codeword lengths for this Huffman code are sent using a three bit fixlength code.

The algorithm for constructing codewords from codeword lengths is specified in the standard.

Coding

What is standardized is the syntax of the coded sequence and how it should be decoded, but there is no specification for how the coder should work. There is a recommendation about how to implement a coder.

The search for matching sequences is not done exhaustively through the whole history buffer, but rather with hash tables. A hash value is calculated from the first three symbols next in line to be coded. In the hash table we keep track of the offsets where sequences with the same hash value start (hopefully sequences where the first three symbol correspond, but that can't be guaranteed). The offsets that have the same hash value are searched to find the longest match, starting with the most recent addition. If no match is found the first symbol is coded as a literal. The search depth is also limited to speed up the coding, at the price of a reduction in compressionen. For instance, the compression parameter in gzip controls how long we search.

Documents

See also:
`ftp://ftp.uu.net/pub/archiving/zip/`
`ftp://ftp.uu.net/graphics/png/`
`http://www.ietf.org/rfc/rfc1950.txt`
`http://www.ietf.org/rfc/rfc1951.txt`
`http://www.ietf.org/rfc/rfc1952.txt`
`http://www.ietf.org/rfc/rfc2083.txt`

LZ78

Lempel and Ziv 1978.

A dictionary of unique sequences is built. The size of the dictionary is S. In the beginning the dictionary is empty, apart from index 0 that means "no match".

Every new sequence that is coded is sent as the tuple $< i, c >$ where i is the index in the dictionary for the longest matching sequence we found and c is the next symbol of the data that didn't match. The matching sequence plus the next symbol is added as a new word to the dictionary.

The number of bits required is

$$\lceil \log S \rceil + \lceil \log L \rceil$$

The decoder can build an identical dictionary.

LZ78, cont.

What to do when the dictionary becomes full? There are a few alternatives:

- Throw away the dictionary and start over.
- Keep coding with the dictionary, but only send index and do not add any more words.
- As above, but only as long as the compressionen is good. If it becomes too bad, throw away the dictionary and start over. In this case we might have to add an extra symbol to the alphabet that informs the decoder to start over.

LZW

LZW is a variant of LZ78 (Welch, 1984).

Instead of sending a tuple $< i, c >$ we only send index i in the dictionary. For this to work, the starting dictionary must contain words of all single symbols in the alphabet.

Find the longest matching sequence in the dictionary and send the index as a new codeword. The matching sequence plus the next symbol is added as a new word to the dictionary.

GIF (Graphics Interchange Format)

Two standards: GIF87a and GIF89a.

A virtual screen is specified. On this screen rectangular images are placed. For each little image we send its position and size.

A colour table of maximum 256 colours is used. Each subimage can have its own colour table, but we can also use a global colour table.

The colour table indices for the pixels are coded using LZW. Two extra symbols are added to the alphabet: ClearCode, which marks that we throw away the dictionary and start over, and EndOfInformation, which marks that the code stream is finished.

Interlace: First lines $0, 8, 16, \ldots$ are sent, then lines $4, 12, 20, \ldots$ then lines $2, 6, 10, \ldots$ and finally lines $1, 3, 5, \ldots$

In GIF89a things like animation and transparency are added.

PNG (Portable Network Graphics)

Partly introduced as a replacement for GIF because of patent issues with LZW (these patents have expired now).

Uses deflate for compression.

Colour depth up to 3×16 bits.

Alpha channel (general transparency).

Can exploit the dependency between pixels (do a prediction from surrounding pixels and code the difference between the predicted value and the real value), which makes it easier to compress natural images.

PNG, cont.

Supports 5 different predictors (called filters):

0 No prediction

1 $\hat{I}_{ij} = I_{i,j-1}$

2 $\hat{I}_{ij} = I_{i-1,j}$

3 $\hat{I}_{ij} = \lfloor (I_{i-1,j} + I_{i,j-1})/2 \rfloor$

4 Paeth (choose the one of $I_{i-1,j}, I_{i,j-1}$ and $I_{i-1,j-1}$ which is closest to $I_{i-1,j} + I_{i,j-1} - I_{i-1,j-1}$)

Lossless JPEG

JPEG is normally an image coding method that gives distortion, but there is also a lossless mode in the standard.

The pixels are coded row-wise from the top down.

The pixel I_{ij} on position (i, j) is predicted from neighbouring pixels.
There are 7 predictors to choose from:

1. $\hat{I}_{ij} = I_{i-1,j}$
2. $\hat{I}_{ij} = I_{i,j-1}$
3. $\hat{I}_{ij} = I_{i-1,j-1}$
4. $\hat{I}_{ij} = I_{i,j-1} + I_{i-1,j} - I_{i-1,j-1}$
5. $\hat{I}_{ij} = I_{i,j-1} + \lfloor (I_{i-1,j} - I_{i-1,j-1})/2 \rfloor$
6. $\hat{I}_{ij} = I_{i-1,j} + \lfloor (I_{i,j-1} - I_{i-1,j-1})/2 \rfloor$
7. $\hat{I}_{ij} = \lfloor (I_{i,j-1} + I_{i-1,j})/2 \rfloor$

Lossless JPEG, cont.

The difference $d_{ij} = I_{ij} - \hat{I}_{ij}$ is coded either by an adaptive arithmetic coder, or using a Huffman code.
Huffman coding is not performed directly on the differences. Instead cathegories

$$k_{ij} = \lceil \log(|d_{ij}| + 1) \rceil$$

are formed. Statistics for the cathegories are calculated and a Huffman tree is constructed.
The codeword for a difference d_{ij} consists of the Huffman codeword for k_{ij} plus k_{ij} extra bits used to exactly specify d_{ij}.

k_{ij}	d_{ij}	extra bits
0	0	–
1	$-1, 1$	$0, 1$
2	$-3, -2, 2, 3$	$00, 01, 10, 11$
3	$-7, \ldots, -4, 4, \ldots, 7$	$000, \ldots, 011, 100, \ldots, 111$
$\vdots$	$\vdots$	$\vdots$

JPEG-LS

Standard for lossles and near lossless coding of images. Near lossless means that we allow the pixel values of the decoded image to be a little different from the original pixels.

The pixels are coded row-wise from the top down.

When pixel (i, j) is to be coded you first look at the surrounding pixels in position $(i, j-1)$, $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$. A *context* is formed by first calculating the gradients

$$\begin{aligned} D_1 &= I_{i-1,j+1} - I_{i-1,j} \\ D_2 &= I_{i-1,j} - I_{i-1,j-1} \\ D_3 &= I_{i-1,j-1} - I_{i,j-1} \end{aligned}$$

JPEG-LS, cont.

The gradients D_k are quantized to three integers Q_k such that $-4 \leq Q_k \leq 4$. The quantizer bounds can be chosen by the coder. Each Q_k takes 9 possible values, which means that we have 729 possible combinations. A pair of combinations with inverted signs counts as the same context which finally gives us 365 different contexts.

A *median edge detector (MED)* prediction of I_{ij} is done according to:
If $I_{i-1,j-1} \geq \max(I_{i,j-1}, I_{i-1,j}) \Rightarrow \hat{I}_{ij} = \min(I_{i,j-1}, I_{i-1,j})$
if $I_{i-1,j-1} \leq \min(I_{i,j-1}, I_{i-1,j}) \Rightarrow \hat{I}_{ij} = \max(I_{i,j-1}, I_{i-1,j})$
Otherwise: $\hat{I}_{ij} = I_{i,j-1} + I_{i-1,j} - I_{i-1,j-1}$

For each context q we keep track if the prediction has a systematic error, if that is the case the prediction is adjusted a little in the correct direction.

JPEG-LS, cont.

The difference between the real pixel value and the predicted value $d_{ij} = I_{ij} - \hat{I}_{ij}$ is coded using a Golomb code with parameter $m = 2^{k_q}$. For each context q we keep track of the best Golomb code, and each k_q is constantly adjusted during the coding process.

The coder also detects if we get long runs of the same value on a row. In that case the coder switches to coding run-lengths instead.

Test image, grayscale

Original image, 768 $\times$ 512 pixels, 8 bits per pixel

Test image, colour

Original image, 768×512 pixels, 24 bits per pixel

Coding with lossless JPEG

Number of bits per pixel when coding the two test images using different predictors.

	grayscale	colour
Predictor 1	4.35	13.15
Predictor 2	4.17	12.58
Predictor 3	4.55	13.79
Predictor 4	4.26	12.83
Predictor 5	4.15	12.51
Predictor 6	4.09	12.33
Predictor 7	4.00	12.07

Coding results

Method	grayscale	colour
Huffman	7.51	
Huffman, difference data	4.13	
Golomb, difference data, var. I	4.47	
Golomb, difference data, var. II	4.48	
Lossless JPEG	4.00	12.07
JPEG-LS	3.54	10.60
GIF	7.07	
PNG	3.97	11.36