Compression: Basic Algorithms

Recap: The Need for Compression

Raw Video, Image and Audio files can very large:

Uncompressed Audio

1 minute of Audio:

Audio Type	44.1 KHz	22.05 KHz	11.025 KHz
16 Bit Stereo	10.1 Mb	5.05 Mb	2.52 Mb
16 Bit Mono	5.05 Mb	2.52 Mb	1.26 Mb
8 Bit Mono	2.52 Mb	1.26 Mb	630 Kb

Uncompressed Images:

Image Type	File Size
512 x 512 Monochrome	0.25 Mb
512 x 512 8-bit colour image	0.25 Mb
512 x 512 24-bit colour image	0.75 Mb

Back
Close

Video

Can also involve: Stream of audio **plus** video imagery.

Raw Video – Uncompressed Image Frames, 512x512 True Color PAL 1125 Mb Per Min

DV Video — 200-300 Mb per Min (Approx) Compressed

HDTV — **Gigabytes** per second Compressed.

- Relying on higher bandwidths is not a good option — M25 Syndrome.
- Compression **HAS TO BE** part of the representation of audio, image and video formats.

Classifying Compression Algorithms

What is Compression?

E.g.: Compression ASCII Characters `EIEIO`

$$\overbrace{01000101}^{\texttt{E(69)}}\overbrace{01001001}^{\texttt{I(73)}}\overbrace{01000101}^{\texttt{E(69)}}\overbrace{01001001}^{\texttt{I(73)}}\overbrace{01001111}^{\texttt{O(79)}} = 5 \times 8 = 40 \text{ bits}$$

The **Main aim** of Data Compression is find a way to use **less bits** per character, E.g.:

$$\overbrace{xx}^{\texttt{E(2bits)}}\ \overbrace{yy}^{\texttt{I(2bits)}}\ \overbrace{xx}^{\texttt{E(2bits)}}\ \overbrace{yy}^{\texttt{I(3bits)}}\ \overbrace{zzz}^{\texttt{O(3bits)}} = \overbrace{(2 \times 2)}^{2\times\texttt{E}} + \overbrace{(2 \times 2)}^{2\times\texttt{I}} + \overbrace{3}^{O} = 11$$

bits

Note: We usually consider character sequences here for simplicity. Other token streams can be used — e.g. Vectorised Image Blocks, Binary Streams.

Back
Close

Compression in Multimedia Data

Compression basically employs redundancy in the data:

- Temporal — in 1D data, 1D signals, Audio etc.
- Spatial — correlation between neighbouring pixels or data items
- Spectral — correlation between colour or luminescence components.
  This uses the frequency domain to exploit relationships between frequency of change in data.
- Psycho-visual — exploit perceptual properties of the human visual system.

Lossless v Lossy Compression

Compression can be categorised in two broad ways:

Lossless Compression — after decompression gives an exact copy of the original data

Examples: Entropy Encoding Schemes (Shannon-Fano, Huffman coding), arithmetic coding,LZW algorithm used in GIF image file format.

Lossy Compression — after decompression gives ideally a 'close' approximation of the original data, in many cases perceptually lossless but a byte-by-byte comparision of files shows differences.

Examples: Transform Coding —
FFT/DCT based quantisation used in JPEG/MPEG differential encoding, vector quantisation

Why do we need Lossy Compression?

- Lossy methods for **typically** applied to high resoultion audio, image compression
- **Have to be employed** in video compression (apart from special cases).

Basic reason:

- *Compression ratio* of lossless methods (e.g., Huffman coding, arithmetic coding, LZW) is not high enough.

Lossless Compression Algorithms: Repetitive Sequence Suppression

- Fairly straight forward to understand and implement.
- Simplicity is their downfall: **NOT best compression ratios**.
- Some methods have their applications, *e.g. Component of JPEG, Silence Suppression*.

Simple Repetition Suppression

If a sequence a series on n successive tokens appears

- Replace series with a token and a count number of occurrences.
- Usually need to have a special *flag* to denote when the repeated token appears

For Example:

```
89400000000000000000000000000000000
```

we can replace with:

```
894f32
```

where `f` is the flag for zero.

Simple Repetition Suppression: How Much Compression?

Compression savings depend on the content of the data.

Applications of this simple compression technique include:

- Suppression of zero's in a file (*Zero Length Suppression*)
  - Silence in audio data, Pauses in conversation *etc.*
  - Bitmaps
  - Blanks in text or program source files
  - Backgrounds in simple images
- Other regular image or data tokens

◀◀
▶▶
◀
▶
Back
Close

Lossless Compression Algorithms: Run-length Encoding (RLE)

This encoding method is frequently applied to graphics-type images (or pixels in a scan line) — simple compression algorithm in its own right.

It is also a component used in **JPEG compression pipeline.**

Basic RLE Approach:

- Sequences of image elements $X_1, X_2, \ldots, X_n$ (Row by Row)
- Mapped to pairs $(c_1, l_1), (c_2, l_2), \ldots, (c_n, l_n)$

  where c_i represent image intensity or colour and l_i the length of the ith run of pixels
- (Not dissimilar to zero length suppression above).

Run-length Encoding Example

Original Sequence (1 Row):

```
111122233333311112222
```

can be encoded as:

```
(1,4),(2,3),(3,6),(1,4),(2,4)
```

How Much Compression?

The savings are dependent on the data: In the **worst case** (**Random Noise**) encoding is more heavy than original file:

2*integer rather than 1* integer if original data is integer vector/array.

MATLAB example code:

runlengthencode.m , runlengthdecode.m

Lossless Compression Algorithms: Pattern Substitution

This is a simple form of statistical encoding.

Here we substitute a frequently repeating pattern(s) with a code.

The code is shorter than than pattern giving us compression.

A simple Pattern Substitution scheme could employ predefined codes

◀◀
▶▶
◀
▶
Back
Close

Simple Pattern Substitution Example

For example replace all occurrences of pattern of characters ‘`and`’ with the predefined code ’&’.

So:

```
and you and I
```

Becomes:

```
& you & I
```

Similar for other codes — commonly used words

Token Assignment

More typically tokens are assigned to according to frequency of occurrence of patterns:

- Count occurrence of tokens
- Sort in Descending order
- Assign some symbols to highest count tokens

A predefined symbol table may be used *i.e.* assign code i to token T. (**E.g. Some dictionary of common words/tokens**)

However, it is more usual to dynamically assign codes to tokens.

The entropy encoding schemes **below** basically attempt to decide the optimum assignment of codes to achieve the best compression.

Lossless Compression Algorithms Entropy Encoding

- Lossless Compression frequently involves some form of **entropy encoding**
- Based on **information theoretic techniques**.

Basics of Information Theory

According to Shannon, the **entropy** of an information source S is defined as:

$$H(S) = \eta = \sum_i \; p_i \log_2 \frac{1}{p_i}$$

where p_i is the probability that symbol S_i in S will occur.

- $\log_2 \frac{1}{p_i}$ indicates the amount of information contained in S_i, i.e., the number of bits needed to code S_i.
- For example, in an image with uniform distribution of gray-level intensity, i.e. p_i = 1/256, then
  - The number of bits needed to code each gray level is 8 bits.
  - The entropy of this image is 8.

◀◀
▶▶
◀
▶
Back
Close

The Shannon-Fano Algorithm — Learn by Example

This is a basic information theoretic algorithm.

A simple example will be used to illustrate the algorithm:

A finite token Stream:

```
ABBAAAACDEAAABBBDDEEAAA........
```

Count symbols in stream:

Symbol	A	B	C	D	E
Count	15	7	6	6	5

Back

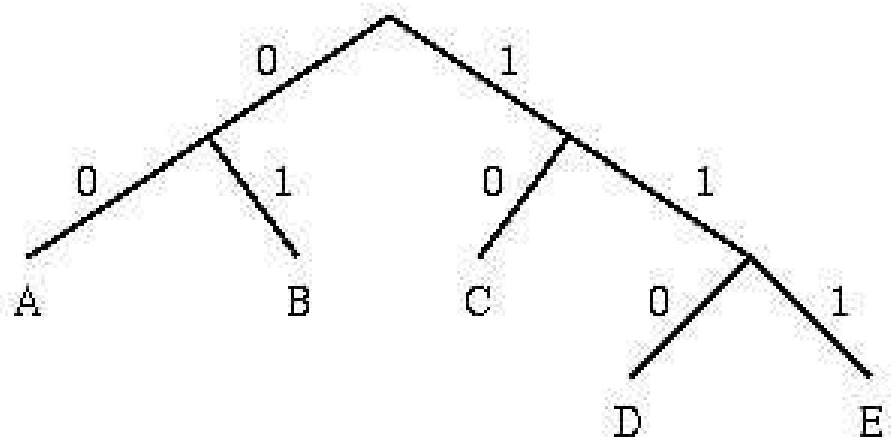
Close

Encoding for the Shannon-Fano Algorithm:

- A top-down approach

1. Sort symbols (Tree Sort) according to their frequencies/probabilities, e.g., ABCDE.

2. Recursively divide into two parts, each with approx. same number of counts.



◀◀
▶▶
◀
▶
Back
Close

3. Assemble code by depth first traversal of tree to symbol node

```
Symbol     Count     log(1/p)      Code        Subtotal (# of bits)
------     -----     --------    ---------     --------------------
  A         15        1.38          00             30
  B          7         2.48          01            14
  C          6         2.70          10            12
  D          6         2.70         110            18
  E          5         2.96         111            15
                            TOTAL (# of bits):     89
```

4. Transmit Codes instead of Tokens

- Raw token stream 8 bits per (39 chars) token = 312 bits
- Coded data stream = 89 bits

Back
Close

Huffman Coding

- Based on the frequency of occurrence of a data item (pixels or small blocks of pixels in images).
- Use a lower number of bits to encode more frequent data
- Codes are stored in a **Code Book** — as for Shannon (previous slides)
- Code book constructed for each image or a set of images.
- Code book **plus** encoded data **must** be transmitted to enable decoding.

Encoding for Huffman Algorithm:

- A bottom-up approach

1. Initialization: Put all nodes in an OPEN list, keep it sorted at all times (e.g., ABCDE).

2. Repeat until the OPEN list has only one node left:

(a) From OPEN pick two nodes having the lowest frequencies/probabilities, create a parent node of them.

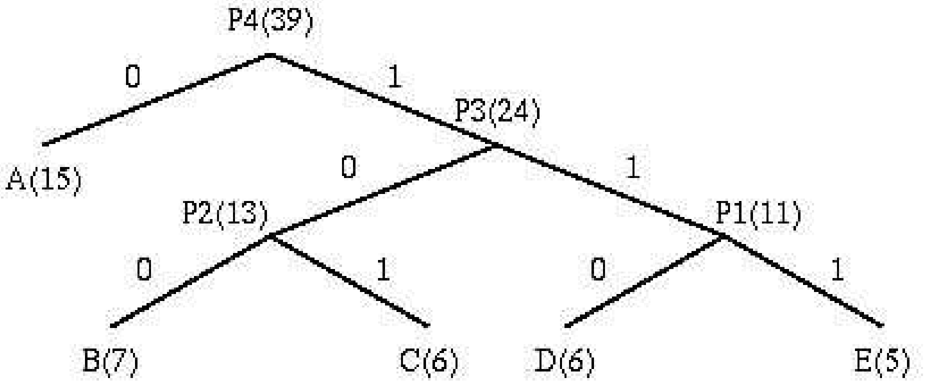
(b) Assign the sum of the children's frequencies/probabilities to the parent node and insert it into OPEN.

(c) Assign code 0, 1 to the two branches of the tree, and delete the children from OPEN.

3. Coding of each node is a top-down label of branch labels.

Huffman Encoding Example:

ABBAAAACDEAAABBBDDEEAAA........ (Same as Shannon-Fano E.g.)



Symbol	Count	log(1/p)	Code	Subtotal (# of bits)
A	15	1.38	0	15
B	7	2.48	100	21
C	6	2.70	101	18
D	6	2.70	110	18
E	5	2.96	111	15
		TOTAL	(# of bits):	87

Huffman Encoder Analysis

The following points are worth noting about the above algorithm:

- Decoding for the above two algorithms is trivial as long as the coding table/book is sent before the data.
  - There is a bit of an overhead for sending this.
  - But negligible if the data file is big.
- **Unique Prefix Property**: no code is a prefix to any other code (all symbols are at the leaf nodes) –> great for decoder, unambiguous.
- If prior statistics are available and accurate, then Huffman coding is very good.

Huffman Entropy

In the above example:

$$\begin{aligned}\mathrm{Idealentropy} &= (15 * 1.38 + 7 * 2.48 + 6 * 2.7 \\ &\quad +6 * 2.7 + 5 * 2.96)/39 \\ &= 85.26/39 \\ &= 2.19\end{aligned}$$

Number of bits needed for Huffman Coding is: $87/39 = 2.23$

Huffman Coding of Images

In order to encode images:

- Divide image up into (typically) 8x8 blocks
- Each block is a symbol to be coded
- Compute Huffman codes for set of block
- Encode blocks accordingly
- In **JPEG**: Blocks are DCT coded first before Huffman may be applied (More soon)

Coding image in blocks is common to all image coding methods

MATLAB Huffman coding example:
huffman.m (Used with JPEG code later),
huffman.zip (Alternative with tree plotting)

Arithmetic Coding

- A widely used entropy coder
- Also used in **JPEG — more soon**
- Only problem is it's speed due possibly complex computations due to large symbol tables,
- Good compression ratio (better than Huffman coding), entropy around the Shannon Ideal value.

Why better than Huffman?

- **Huffman coding etc.** use an integer number (k) of bits for each symbol,
  - hence k is never less than 1.
- Sometimes, e.g., when sending a 1-bit image, compression **becomes impossible**.

◀◀
▶▶
◀
▶
Back
Close

Decimal Static Arithmetic Coding

- Here we describe basic approach of Arithmetic Coding
- Initially basic static coding mode of operation.
- Initial example **decimal coding**
- Extend to Binary and then machine word length later

Basic Idea

The idea behind arithmetic coding is

- To have a probability line, 0–1, and
- Assign to every symbol a range in this line based on its probability,
- The higher the probability, the higher range which assigns to it.

Once we have defined the ranges and the probability line,

- Start to encode symbols,
- Every symbol defines where the output floating point number lands within the range.

◀◀
▶▶
◀
▶
Back
Close

Simple Basic Arithmetic Coding Example

Assume we have the following token symbol stream

`BACA`

Therefore

- A occurs with **probability 0.5**,
- B and C with **probabilities 0.25**.

Basic Arithmetic Coding Algorithm

Start by assigning each symbol to the probability range 0–1.

- Sort symbols highest probability first

Symbol	Range
A	[0.0, 0.5)
B	[0.5, 0.75)
C	[0.75, 1.0)

- The first symbol in our example stream is B

We now know that the code will be in the range 0.5 to $0.74999\ldots$.

Back
Close

Range is not yet unique

- Need to narrow down the range to give us a unique code.

Basic arithmetic coding iteration

- Subdivide the range for the first token given the probabilities of the second token then the third etc.

Subdivide the range as follows

For all the symbols:

```
range = high - low;
high = low + range * high_range of the symbol being coded;
low = low + range * low_range of the symbol being coded;
```

Where:

- `range`, keeps track of where the next range should be.
- `high` and `low`, specify the output number.
- Initially `high = 1.0`, `low = 0.0`

◀◀
▶▶
◀
▶
Back
Close

Back to our example

The second symbols we have
(now `range` = 0.25, `low` = 0.5, `high` = 0.75):

Symbol	Range
B**A**	[0.5, 0.625)
BB	[0.625, 0.6875)
BC	[0.6875, 0.75)

Back
Close

Third Iteration

We now reapply the subdivision of our scale again to get for our third symbol

(range = 0.125, low = 0.5, high = 0.625):

Symbol	Range
BAA	[0.5, 0.5625)
BAB	[0.5625, 0.59375)
BA**C**	[0.59375, 0.625)

Fourth Iteration

Subdivide again
(`range` = 0.03125, `low` = 0.59375, `high` = 0.625):

Symbol	Range
BAC**A**	[0.59375, 0.60937)
BACB	[0.609375, 0.6171875)
BACC	[0.6171875, 0.625)

So the (Unique) output code for BACA is any number in the range:

[0.59375, 0.60937).

Decoding

To **decode** is essentially the opposite

- We compile the table for the sequence given probabilities.
- Find the range of number within which the code number lies and carry on

Binary static algorithmic coding

This is very similar to above:

- **Except** we us **binary fractions**.

Binary fractions are simply an extension of the binary systems into fractions much like decimal fractions.

Binary Fractions — Quick Guide

Fractions in **decimal**:

0.1 decimal = $\frac{1}{10^1} = 1/10$
0.01 decimal = $\frac{1}{10^2} = 1/100$
0.11 decimal = $\frac{1}{10^1} + \frac{1}{10^2} = 11/100$

So in **binary** we get

0.1 binary = $\frac{1}{2^1} = 1/2$ decimal
0.01 binary = $\frac{1}{2^2} = 1/4$ decimal
0.11 binary = $\frac{1}{2^1} + \frac{1}{2^2}$ = 3/4 decimal

◀◀
▶▶
◀
▶
Back
Close

Binary Arithmetic Coding Example

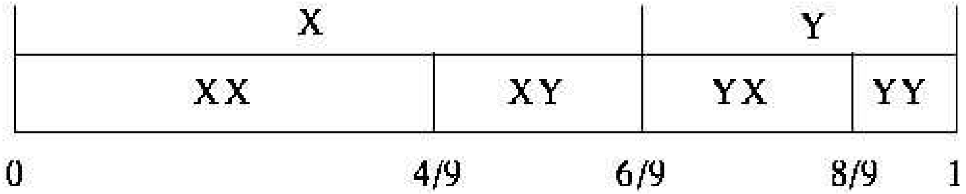
- Idea: Suppose alphabet was `X, Y` and token stream:

  `XXY`

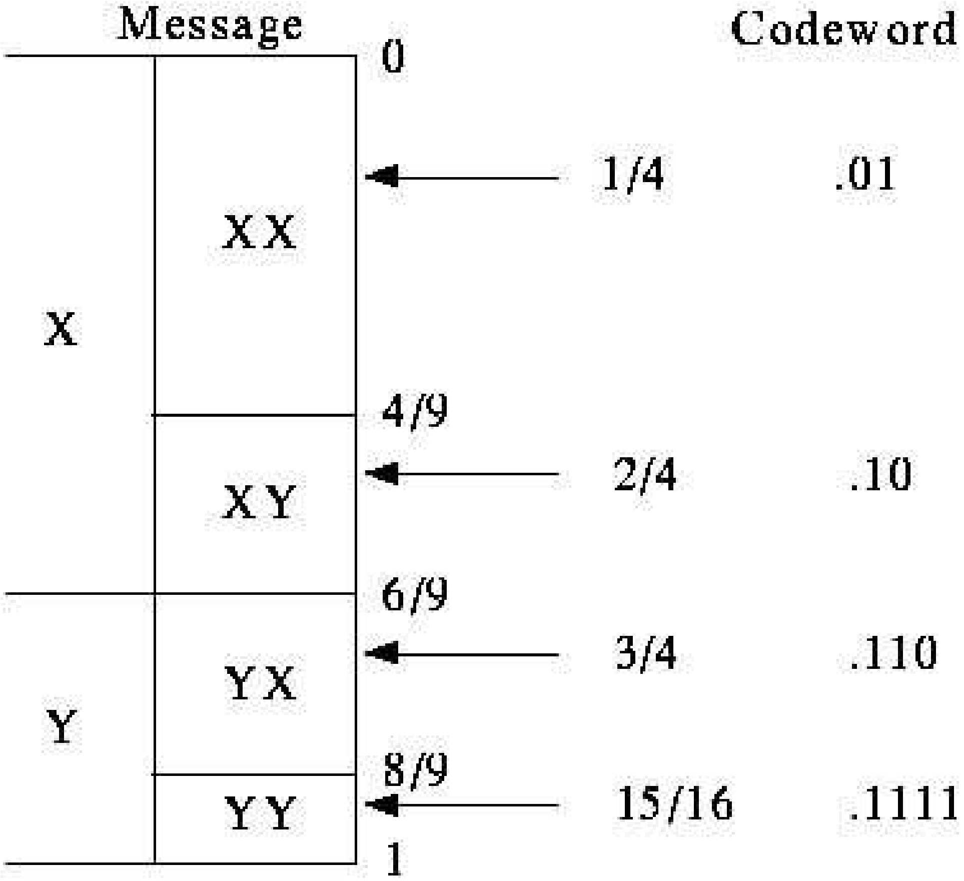
  Therefore:

  ```
  prob(X) = 2/3
  prob(Y) = 1/3
  ```

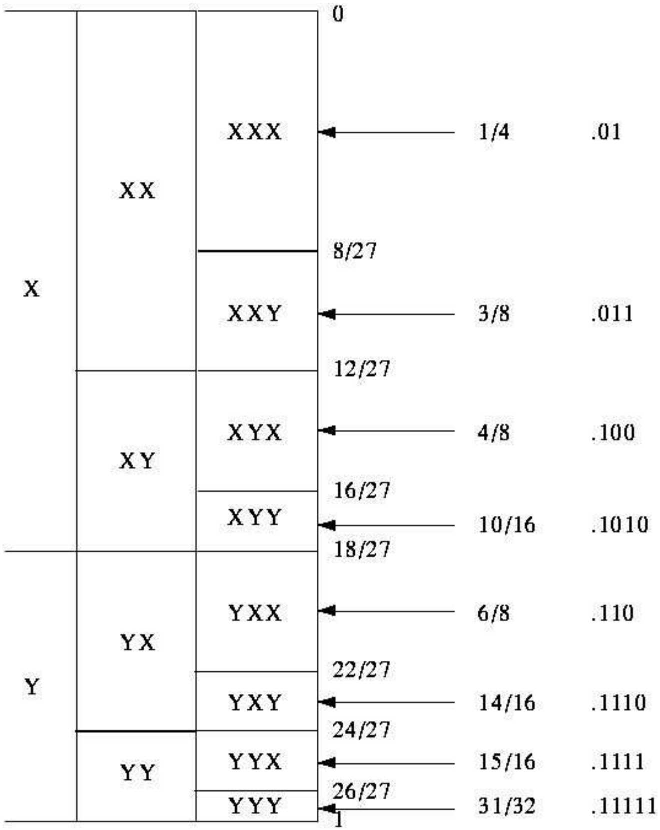
- If we are only concerned with encoding length 2 messages, then we can map all possible messages to intervals in the range [0..1]:



- To encode message, just send enough bits of a binary fraction that uniquely specifies the interval.



- Similarly, we can map all possible length 3 messages to intervals in the range [0..1]:



Implementation Issues

FPU Precision

- Resolution of the of the number we represent is limited by FPU precision
- Binary coding extreme example of rounding
- Decimal coding is the other extreme — theoretically no rounding.
- Some FPUs may us up to 80 bits
- As an example let us consider working with 16 bit resolution.

16-bit arithmetic coding

We now encode the range 0–1 into 65535 segments:

0.000	0.250	0.500	0,750	1.000
0000h	4000h	8000h	C000h	FFFFh

If we take a number and divide it by the maximum (FFFFh) we will clearly see this:

```
0000h: 0/65535 = 0.0
4000h: 16384/65535 = 0.25
8000h: 32768/65535 = 0.5
C000h: 49152/65535 = 0.75
FFFFh: 65535/65535 = 1.0
```

◀◀
▶▶
◀
▶
Back
Close

The operation of coding is similar to what we have seen with the binary coding:

- Adjust the probabilities so the bits needed for operating with the number aren't above 16 bits.
- Define a new interval
- The way to deal with the infinite number is
  - to have only loaded the 16 first bits, and when needed shift more onto it:

    ```
    1100 0110 0001 000 0011 0100 0100 ...
    ```
  - work only with those bytes
  - as new bits are needed they'll be shifted.

Memory Intensive

What about an alphabet with 26 symbols, or 256 symbols, ...?

- In general, number of bits is determined by the size of the interval.
- In general, (from entropy) need $-\log p$ bits to represent interval of size p.
- Can be memory and CPU intensive

MATLAB Arithmetic coding examples:
Arith06.m (Version 1),
Arith07.m (Version 2)

Lempel-Ziv-Welch (LZW) Algorithm

- A very common compression technique.
- Used in GIF files (LZW), Adobe PDF file (LZW), UNIX `compress` (LZ Only)
- Patented — LZW not LZ.

Basic idea/Example by Analogy:

Suppose we want to encode the Oxford Concise English dictionary which contains about 159,000 entries.

Why not just transmit each word as an 18 bit number?

Back

Close

LZW Constructs Its Own Dictionary

Problems:

- Too many bits per word,
- Everyone needs a dictionary,
- Only works for English text.

Solution:

- Find a way to build the dictionary adaptively.
- Original methods (LZ) due to Lempel and Ziv in 1977/8.
- Quite a few variations on LZ.
- Terry Welch improvement (1984), **Patented LZW Algorithm**
  - LZW introduced the idea that only the **initial dictionary** needs to be transmitted to enable **decoding**:
    The decoder is able to **build** the **rest** of the table from the **encoded sequence**.

◀◀
▶▶
◀
▶
Back
Close

LZW Compression Algorithm

The LZW Compression Algorithm can summarised as follows:

```
w = NIL;
while ( read a character k )
  {    if wk exists in the dictionary
            w = wk;
         else
            { add wk to the dictionary;
               output the code for w;
                w = k;
             }
   }
```

- Original LZW used dictionary with 4K entries, first 256 (0-255) are ASCII codes.

LZW Compression Algorithm Example:

Input string is "^WED^WE^WEE^WEB^WET".

w	k	output	index	symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			
^W	E	256	260	^WE
E	^	E	261	E^
^	W			
^W	E			
^WE	E	260	262	^WEE
E	^			
E^	W	261	263	E^W
W	E			
WE	B	257	264	WEB
B	^	B	265	B^
^	W			
^W	E			
^WE	T	260	266	^WET
T	EOF	T		

- A 19-symbol input has been reduced to 7-symbol plus 5-code output. Each code/symbol will need more than 8 bits, say 9 bits.
- Usually, compression doesn't start until a large number of bytes (e.g., $>$ 100) are read in.

Back
Close

LZW Decompression Algorithm

The LZW Decompression Algorithm is as follows:

```
read a character k;
output k;
w = k;
while ( read a character k )
/* k could be a character or a code. */
  {
          entry = dictionary entry for k;
          output entry;
          add w + entry[0] to dictionary;
          w = entry;
  }
```

Note (Recall):

LZW decoder only needs the **initial dictionary**:

The decoder is able to **build** the **rest** of the table from the **encoded sequence**.

Back
Close

LZW Decompression Algorithm Example:

Input string is

"^WED<256>E<260><261><257>B<260>T"

w	k	output	index	symbol
^	^			
^	W	W	256	^W
W	E	E	257	WE
E	D	D	258	ED
D	<256>	^W	259	D^
<256>	E	E	260	^WE
E	<260>	^WE	261	E^
<260>	<261>	E^	262	^WEE
<261>	<257>	WE	263	E^W
<257>	B	B	264	WEB
B	<260>	^WE	265	B^
<260>	T	T	266	^WET

Back

Close

MATLAB LZW Code

norm2lzw.m: LZW Encoder

lzw2norm.m: LZW Decoder

lzw_demo1.m: Full MATLAB demo

More Info on MATLAB LZW code

Lossy Compression: Source Coding Techniques

Source coding is based changing on the content of the original signal.

Also called *semantic-based coding*

High compression rates may be high but a price of loss of information. Good compression rates make be achieved with source encoding with (occasionally) **lossless** or (mostly) little **perceivable** loss of information.

There are three broad methods that exist:

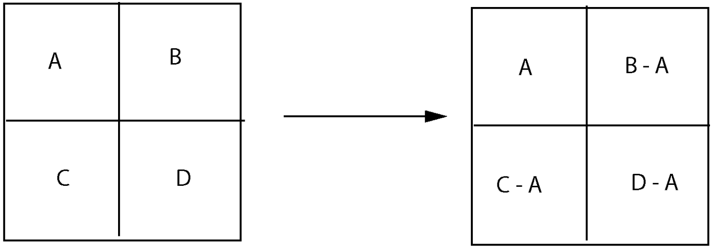
- Transform Coding
- Differential Encoding
- Vector Quantisation

Transform Coding

A simple transform coding example

A Simple Transform Encoding procedure maybe described by the following steps for a 2x2 block of monochrome pixels:

1. Take top left pixel as the base value for the block, pixel A.
2. Calculate three other transformed values by taking the difference between these (respective) pixels and pixel A, **Ii.e. B-A, C-A, D-A.**
3. Store the base pixel and the differences as the values of the transform.



A	B
C	D

A	B - A
C - A	D - A

Back
Close

Simple Transforms

Given the above we can easily form the forward transform:

$$
\begin{aligned}
X_0 &= A \\
X_1 &= B - A \\
X_2 &= C - A \\
X_3 &= D - A
\end{aligned}
$$

and the inverse transform is:

$$
\begin{aligned}
A_n &= X_0 \\
B_n &= X_1 + X_0 \\
C_n &= X_2 + X_0 \\
D_n &= X_3 + X_0
\end{aligned}
$$

Compressing data with this Transform?

Exploit redundancy in the data:

- Redundancy transformed to values, X_i.
- Compress the data by using fewer bits to represent the differences — *Quantisation*.
  - I.e if we use 8 bits per pixel then the 2x2 block uses 32 bits
  - If we keep 8 bits for the base pixel, X0,
  - Assign 4 bits for each difference then we only use 20 bits.
  - Better than an average 5 bits/pixel

Example

Consider the following 4x4 image block:

120	130
125	120

then we get:

$$
\begin{aligned}
X_0 &= 120 \\
X_1 &= 10 \\
X_2 &= 5 \\
X_3 &= 0
\end{aligned}
$$

We can then compress these values by taking less bits to represent the data.

Back
Close

Inadequacies of Simple Scheme

- It is **Too Simple**
- Needs to operate on larger blocks (typically 8x8 min)
- Simple encoding of differences for large values will result in loss of information
  - V. poor losses possible here 4 bits per pixel = values 0-15 unsigned,
  - Signed value range: $-7 - 7$ so either quantise in multiples of 255/max value or massive overflow!!

◀◀
▶▶
◀
▶
Back
Close

Differential Transform Coding Schemes

- **Differencing** is used in some compression algorithms:
  - Later part of JPEG compression
  - Exploit static parts (*e.g.* background) in MPEG video
  - Some speech coding and other simple signals
  - **Good** on repetitive sequences
- **Poor** on highly varying data sequences
  - *e.g* interesting audio/video signals

MATLAB Simple Vector Differential Example

diffencodevec.m: Differential Encoder
diffdecodevec.m: Differential Decoder
diffencodevecTest.m: Differential Test Example

Back
Close

Differential Encoding

Simple example of transform coding mentioned earlier and instance of this approach.

Here:

- The difference between the actual value of a sample and a prediction of that values is encoded.
- Also known as **predictive encoding**.
- Example of technique include: differential pulse code modulation, delta modulation and adaptive pulse code modulation — differ in prediction part.
- Suitable where successive signal samples do not differ much, but are not zero. E.g. Video — difference between frames, some audio signals.

Differential Encoding Methods

- **Differential pulse code modulation** (DPCM)

Simple prediction (also used in JPEG):

$$f_{predict}(t_i) = f_{actual}(t_{i-1})$$

I.e. a simple Markov model where current value is the predict next value.

So we simply need to encode:

$$\Delta f(t_i) = f_{actual}(t_i) - f_{actual}(t_{i-1})$$

If successive sample are close to each other we only need to encode first sample with a large number of bits:

Back
Close

Simple Differential Pulse Code Modulation Example

Actual Data: 9 10 7 6

Predicted Data: 0 9 10 7

$\Delta f(t)$: +9, +1, -3, -1.

MATLAB Complete (with quantisation) DPCM Example

dpcm_demo.m.m: DPCM Complete Example
dpcm.zip.m: DPCM Support FIles

Differential Encoding Methods (Cont.)

- **Delta modulation** is a special case of DPCM:
  - Same predictor function,
  - Coding error is a **single bit or digit** that indicates the current sample should be increased or decreased by a step.
  - Not Suitable for rapidly changing signals.
- **Adaptive pulse code modulation**

Fuller Temporal/Markov model:

- Data is extracted from a function of a series of previous values
- **E.g.** Average of last n samples.
- Characteristics of sample better preserved.

◀◀
▶▶
◀
▶
Back
Close

Frequency Domain Methods

Another form of Transform Coding

Transformation from one domain —time (*e.g.* 1D audio, video:2D imagery over time) or Spatial (*e.g.* 2D imagery) domain to the **frequency** domain via

- **Discrete Cosine Transform (DCT)**— Heart of **JPEG** and **MPEG Video**, (alt.) MPEG Audio.
- **Fourier Transform (FT)** — **MPEG Audio**

Theory already studied earlier

Back

Close

RECAP — Compression In Frequency Space

How do we achieve compression?

- Low pass filter — ignore high frequency noise components
- Only store lower frequency components
- High Pass Filter — Spot Gradual Changes
- If changes to low Eye does not respond so ignore?

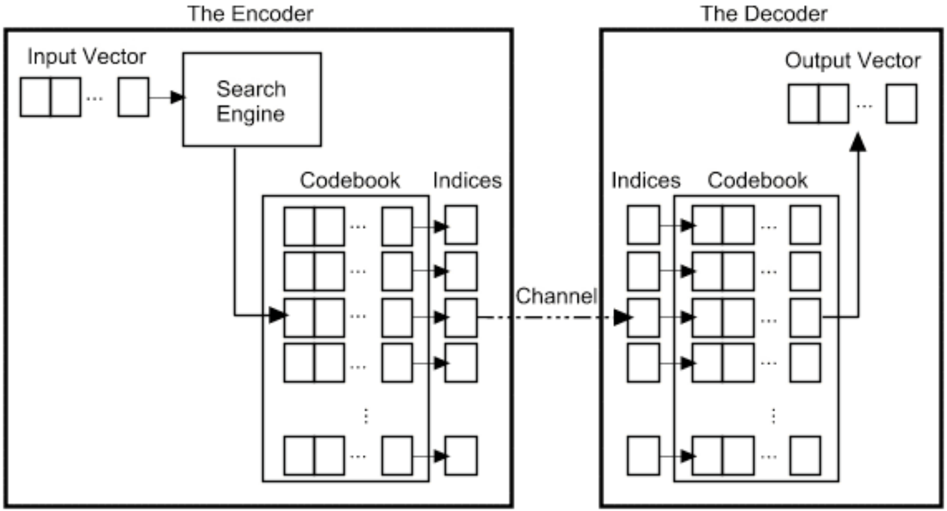
Back
Close

Vector Quantisation

The basic outline of this approach is:

- Data stream divided into (1D or 2D square) blocks — **vectors**
- A table or **code book** is used to find a pattern for each block.
- Code book can be dynamically constructed or predefined.
- Each pattern for block encoded as a look value in table
- Compression achieved as data is effectively subsampled and coded at this level.
- Used in MPEG4, Video Codecs (Cinepak, Sorenson), Speech coding, Ogg Vorbis.

Vector Quantisation Encoding/Decoding

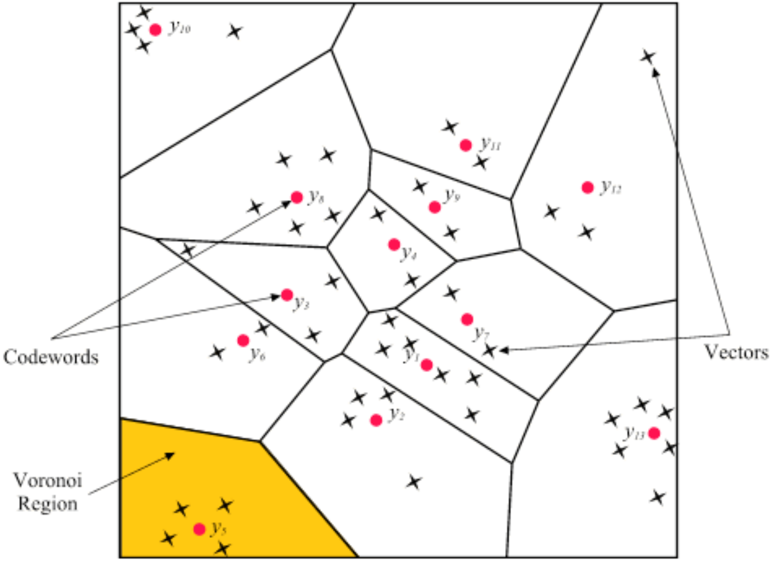


- **Search Engine**:
  - Group (Cluster) data into vectors
  - Find closest code vectors
- On decode output need to *unblock* (smooth) data

Vector Quantisation Code Book Construction

How to cluster data?

- Use some clustering technique,
  e.g. K-means, Voronoi decomposition
  Essentially cluster on some closeness measure, minimise inter-sample variance or distance.

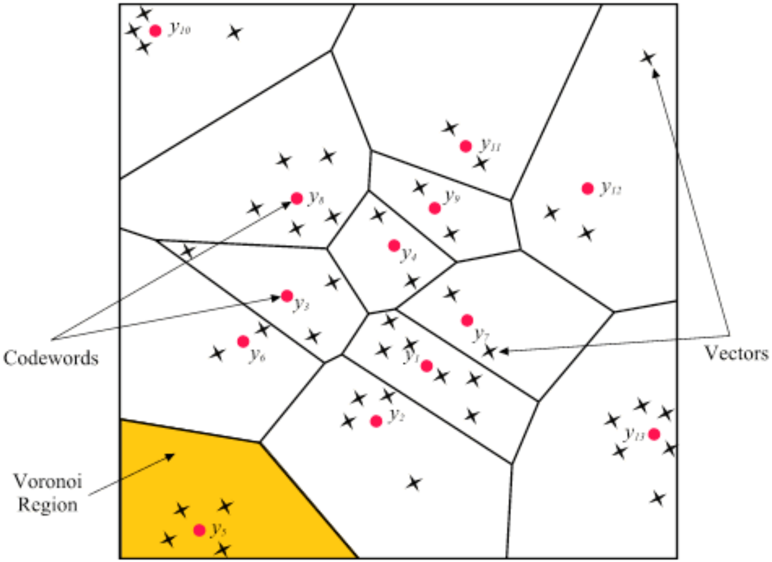


Back
Close

Vector Quantisation Code Book Construction

How to code?

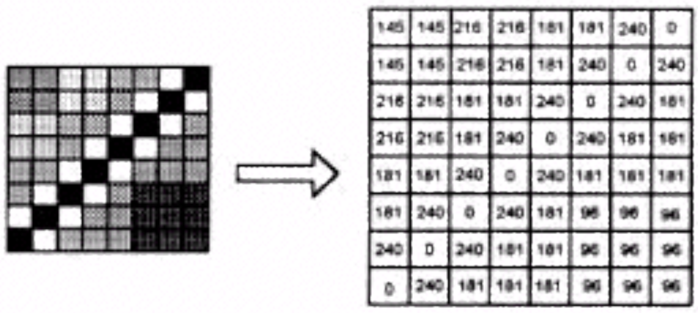
- For each cluster choose a mean (median) point as representative code for all points in cluster.



Back
Close

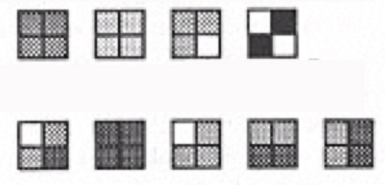
Vector Quantisation Image Coding Example

- A small block of images and intensity values



145	145	216	216	181	181	240	0
145	145	216	216	181	240	0	240
216	216	181	181	240	0	240	181
216	216	181	240	0	240	181	181
181	181	240	0	240	181	181	181
181	240	0	240	181	96	96	96
240	0	240	181	181	96	96	96
0	240	181	181	181	96	96	96

- Consider Vectors of 2x2 blocks, and only allow 8 codes in table.
- 9 vector blocks present in above:

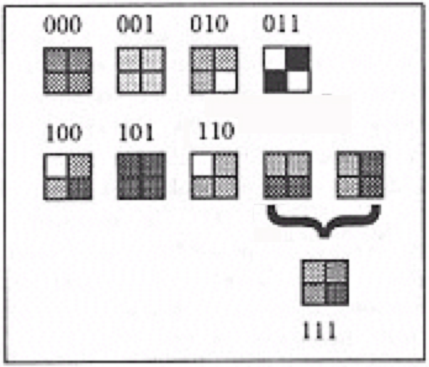


Back

Close

Vector Quantisation Image Coding Example (Cont.)

- 9 vector blocks, so **only one** has to be **vector quantised** here.
- Resulting code book for above image



MATLAB EXAMPLE: vectorquantise.m