Audio and Video Coding

Armando J. Pinho

Departamento de Electrónica, Telecomunicações e Informática

Universidade de Aveiro

ap@ua.pt

http://www.ieeta.pt/~ap

Contents I

Motivation

- In many cases, the information source produces recurring patterns.
- A possibility is to keep a dictionary with these patterns.
- When one of those patterns occurs, it is encoded using a reference to the dictionary.
- If it does not appear in the dictionary, it can be encoded using some other (less efficient) method.
- Therefore, the idea is to split the patterns into two classes: frequent and rare.

Motivation

- Suppose we have a source alphabet with 32 symbols (for example, 26 letters plus some punctuation marks).
- For an iid source, we would need 5 bits/symbol.
- Treating all 32^4 (1 048 576) 4-symbol patterns as equally likely, it would imply 20 bits for each 4-symbol pattern.
- Suppose we put the 256 most common patterns in a dictionary.
- If the pattern is in the dictionary, send '0' followed by the (8-bit) index of the dictionary.
- Else, send '1' followed by the 20 bits encoding the pattern.

Motivation

- If p is the probability of finding the pattern in the dictionary, then the average code-length is

$$r = 9p + 21(1 - p) = 21 - 12p.$$

- To be useful, this scheme should attain $r < 20$.
- This happens for $p \geq 0.084$.
- Notice that, for an iid source, the probability of a 4-symbol pattern over a 32-symbol alphabet is only 0.00025...

Tunstall codes

Principles

- As we have seen, variable-length codes assign variable-length bit sequences to the symbols of the alphabet.
- One of the problems with this approach is that errors in the encoded data propagate easily.
- In the Tunstall code, all codewords are of equal length.
- Instead, each codeword represents variable-length groups of alphabet symbols.

Tunstall codes

Example

- Consider the following 2-bit Tunstall code for the alphabet $\mathcal{A} = \{A, B\}$:

Sequence	Codeword
AAA	00
AAB	01
AB	10
B	11

- Then, the sequence “AAABAABAABAABAAA” is encoded as “001101010100”.

Tunstall codes

Conditions for construction

- The design of a code that has a fixed codeword length but a variable number of symbols per codeword needs to meet the following two conditions:
  - We should be able to parse a source output sequence into sequences of symbols that appear in the codebook (or dictionary).
  - We should maximize the average number of source symbols represented by each codeword.

Tunstall codes

Conditions for construction

- To understand the meaning of the first condition, Consider now the following code:

Sequence	Codeword
AAA	00
ABA	01
AB	10
B	11

- Let us try to encode the same sequence, “AAABAABAABAABAAA”, but now using this new code:
  - First, we encode “AAA” with code “00”.
  - Then, we encode “B” with code “11”.
  - Next, we run into trouble, because there is no way to proceed. . .

Tunstall codes

Construction procedure

- A n-bit Tunstall code for an iid source over a size-N alphabet can be built using the following procedure:
  - We start with the N symbols in the codebook.
  - Then we remove the entry with highest probability and add the N sequences obtained by concatenating this symbol with every symbol in the alphabet.
  - Note that this increases the size of the codebook from N to $N + (N - 1)$, and that the new probabilities are the product of the probabilities of the concatenated symbols.
  - We repeat this procedure K times, subject to $N + K(N - 1) \leq 2^n$, where K is the largest possible integer.

Tunstall codes

Example

- Consider designing a 3-bit Tunstall code for the alphabet

Symbol	Probability
A	0.60
B	0.30
C	0.10

- Because “A” is the most probable symbol, we first get

Symbol	Probability
B	0.30
C	0.10
AA	0.36
AB	0.18
AC	0.06

Tunstall codes

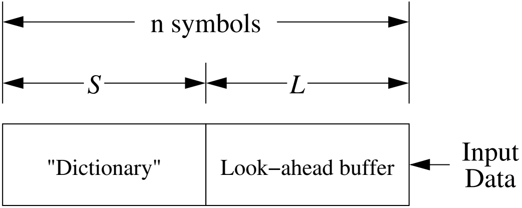
Example

- And finally,

Symbol	Probability	Code
B	0.300	000
C	0.100	001
AB	0.180	010
AC	0.060	011
AAA	0.216	100
AAB	0.108	101
AAC	0.036	110

LZ77 compression

- J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Trans. on Information Theory, 1977, **23**, pp. 337–343.
- The LZ77 algorithm relies on a separation of the data in two parts:
  - Data already encoded.
  - Data that are to be encoded.



LZ77 compression

- During operation, the data go first through a look-ahead buffer and then through a search buffer (the “dictionary”).
- The algorithm searches, in the “dictionary”, the largest sequence of symbols that can be found in the look-ahead buffer.
- The larger the sequences that are found, the higher will be the coding efficiency.
- The algorithm expects that repeating sequences occur close to each other.

LZ77 compression

- The codeword that is generated in each coding step is composed of three components:
  1. A pointer, that indicates the position of the repeating sequence in the “dictionary”.
  2. The length of the sequence.
  3. The first symbol in the look-ahead buffer that follows the matching sequence.
- Example:

10	9	8	7	6	5	4	3	2	1					
										a	c	a	a	(0,0,a)
									a	c	a	a	c	(0,0,c)
								a	c	a	a	c	a	(2,1,a)
						a	c	a	a	c	a	b	c	(3,2,b)
			a	c	a	a	c	a	b	c	a	b	a	...

LZ77 compression

- The look-ahead buffer can also be used as a “dictionary” extension.
- Example:

10	9	8	7	6	5	4	3	2	1					
a	c	a	a	c	a	b	c	a	b	a	b	a	c	(2,3,c)

- A codeword requires

$$\lceil \log_2 S \rceil + \lceil \log_2 (L-1) \rceil + \lceil \log_2 M \rceil$$

bits, where S is the size of the “dictionary”, L is the size of the look-ahead buffer and M is the size of the alphabet.

- Typically, this number of bits can be $11 + 5 + 8 = 24$, much larger than the number of bits required to represent a symbol (8 bits).

LZ77 compression

- There are some problems associated with this compression algorithm:
  - The search time required by each coding step can be high if the dictionary is large.
  - Increasing the size of the dictionary increases the probability of finding matching sequences (the compression efficiency increases, but the processing time also rises).
  - Increasing the size of the look-ahead buffer increases the probability of finding larger sequences, but this also increases the processing time. . .
  - In both cases, more bits are needed to represent the pointers to the dictionary and the size of the matched sequences.
  - Generally, the three component codewords are inefficient (the isolated symbol could be included in the next sequence).

LZ78 compression

- J. Ziv and A. Lempel, *Compression of individual sequences via variable-rate coding*, IEEE Trans. on Information Theory, 1978, **24**, pp. 530–536.
- Uses a distinct approach from that of LZ77 (it is not based on sliding windows):
  - The data are partitioned into strings.
  - Each string is built of the largest matching string found so far, plus the first non-matching character.
  - The new string is represented using the index of the matching string (which is in the dictionary) followed by the code of the unmatched symbol.
  - Finally, this new string is added to the dictionary.

LZ78 compression

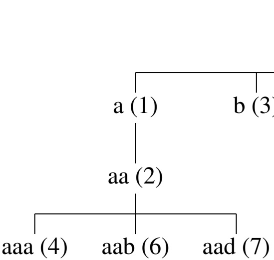
- Example: coding of message "aaabaaadaabaado".

Dictionary:	a	aa	b	aaa	d	aab	aad	o
Index:	1	2	3	4	5	6	7	8
Codeword:	0,a	1,a	0,b	2,a	0,d	2,b	2,d	0,o

- More spaced sequences can be encoded more efficiently than with LZ77.
- Theoretically, LZ78 does not put limitations to the maximum temporal distance for referencing past repeating sequences…

LZ78 compression

- The dictionary can be efficiently implemented using a tree data structure:



- Note that, for an alphabet size of *M* symbols, each node of the tree has a maximum of *M* children.

LZ78 compression

- In practice, the dictionary cannot grow without bound:
  - The size of the codewords is related to the size of the dictionary, because the dictionary entries have associated indexes.
  - For example, for a size n dictionary, the pointers have to be stored with $\lceil \log_2 n \rceil$ bits.
  - The amount of memory for storing the dictionary (both by the encoder and by the decoder) might be too large.
- Therefore, in practice, it is necessary to implement a mechanism for limiting the growth of the dictionary.

LZ78 compression

- Some of those mechanisms could be:
  - “Freezing” the dictionary, when it reaches some predefined size.
  - When the dictionary reaches some predefined size, it is deleted and starts growing again (this is identical to block coding).
  - When the dictionary is full, some entries are deleted, for example, those not used for a longer time.

LZW compression

- In 1984, Terry Welch published a paper where solutions for some of the problems associated to LZ78 are proposed:
  - Symbols seen for the first time are encoded more efficiently (in LZ78, two-component codewords are generated in this case).
  - The number of components of the codewords is reduced to one.
- Initially, all symbols of the alphabet are inserted in the dictionary.
- The algorithm operates as follows:
  - Symbols are read, one by one, from the message to encode.
  - These symbols (**x**) are concatenated to a sequence, **S**, while **Sx** can be found in the dictionary.
  - When it is not possible to add one more symbol, the index of **S** in the dictionary is sent, **Sx** is inserted in the dictionary, and **S** is initialized with symbol **x**.

LZW compression

- Example: encoding of message “aaabaaadaabaado”.
- Initially, the dictionary contains all symbols of the alphabet:

0	...	97	98	99	100	...	255
nul	...	a	b	c	d	...	

- Message: $\boxed{\text{aa}}$abaaadaabaado; Codeword sent: 97

...	97	98	99	100	...	256
...	a	b	c	d	...	aa

- Message: a$\boxed{\text{aab}}$aaadaabaado; Codeword sent: 256

...	97	98	99	100	...	256	257
...	a	b	c	d	...	aa	aab

LZW compression

- Message: aaa $\boxed{\text{ba}}$ aadaabaado; Codeword sent: 98

...	97	98	99	100	...	256	257	258
...	a	b	c	d	...	aa	aab	ba

- Message: aaab $\boxed{\text{aaa}}$ daabaado; Codeword sent: 256

...	97	98	99	100	...	256	257	258	259
...	a	b	c	d	...	aa	aab	ba	aaa

- Message: aaabaa $\boxed{\text{ad}}$ aabaado; Codeword sent: 97

...	97	98	99	100	...	256	257	258	259	260
...	a	b	c	d	...	aa	aab	ba	aaa	ad

LZW compression

- Message: aaabaaa $\boxed{\text{da}}$ abaado; Codeword sent: 100

...	97	98	99	100	...	256	257	258	259	260	261
...	a	b	c	d	...	aa	aab	ba	aaa	ad	da

- Message: aaabaaad $\boxed{\text{aaba}}$ ado; Codeword sent: 257

...	97	98	99	100	...	256	257	258	259	260	261	262
...	a	b	c	d	...	aa	aab	ba	aaa	ad	da	aaba

- Message: aaabaaadaab $\boxed{\text{aad}}$ o; Codeword sent: 256

...	97	98	99	100	...	256	257	258	259	260	261	262	263
...	a	b	c	d	...	aa	aab	ba	aaa	ad	da	aaba	aad

- ...

LZW compression

- Example: decoding.
- Initially, the dictionary contains all symbols of the alphabet:

0	...	97	98	99	100	...	255
nul	...	a	b	c	d	...	

- Codeword received: 97 ; Message: $\boxed{\text{a?}}$

...	97	98	99	100	...	256
...	a	b	c	d	...	a?

- Codeword received: 256 ; Message: a $\boxed{\text{aa?}}$

...	97	98	99	100	...	256	257
...	a	b	c	d	...	aa	aa?

LZW compression

- Codeword received: 98 ; Message: aaa $\boxed{\text{b?}}$

...	97	98	99	100	...	256	257	258
...	a	b	c	d	...	aa	aab	b?

- Codeword received: 256 ; Message: aaab $\boxed{\text{aa?}}$

...	97	98	99	100	...	256	257	258	259
...	a	b	c	d	...	aa	aab	ba	aa?

- Codeword received: 97 ; Message: aaabaa $\boxed{\text{a?}}$

...	97	98	99	100	...	256	257	258	259	260
...	a	b	c	d	...	aa	aab	ba	aaa	a?

LZW compression

- Codeword received: 100 ; Message: aaabaaa $\boxed{\text{d?}}$

...	97	98	99	100	...	256	257	258	259	260	261
...	a	b	c	d	...	aa	aab	ba	aaa	ad	d?

- Codeword received: 257 ; Message: aaabaaad $\boxed{\text{aab?}}$

...	97	98	99	100	...	256	257	258	259	260	261	262
...	a	b	c	d	...	aa	aab	ba	aaa	ad	da	aab?

- Codeword received: 256 ; Message: aaabaaadaab $\boxed{\text{aa?}}$

...	97	98	99	100	...	256	257	258	259	260	261	262	263
...	a	b	c	d	...	aa	aab	ba	aaa	ad	da	aaba	aa?

- ...