Module 2

LOSSLESS IMAGE COMPRESSION SYSTEMS

Lesson 4

Arithmetic and Lempel-Ziv Coding

Version 2 ECE IIT, Kharagpur

Instructional Objectives

At the end of this lesson, the students should be able to:

1. Explain why Huffman coding is not optimal.
2. State the basic principles of arithmetic coding.
3. Encode a sequence of symbols into an arithmetic coded bit stream.
4. Decode an arithmetic coded bit stream.
5. State the coding efficiency limitations of arithmetic coding.
6. State the basic principles of Lempel-Ziv coding
7. Encode a sequence of symbols into a Lempel-Ziv coded bit stream.
8. Decode a Lempel-Ziv coded bit stream.

4.0 Introduction

In lesson-3, we have studied Huffman coding, one of the popular lossless compression schemes. However, we have seen from our examples that although the average code word length is much less as compared to that of fixed-length coding, it is still higher than the entropy and consequently, the coding efficiency is less than one. It is because, we have to encode one symbol at a time and each symbol translates into an integral number of bits. If instead, we sacrifice the one-to-one mapping between the symbol and its code word and encode the entire sequence of source symbols into one single code word, we may expect the coding efficiency to be better and the average code word length approach the lower bound given by Shannon's noiseless coding theorem. This is exactly the basis of *arithmetic coding*, which has emerged as a strong alternative to Huffman coding.

This lesson begins with a discussion on *arithmetic coding* techniques. We shall first present the basic principles of *arithmetic coding* and then study how to encode a sequence of symbols into a bit stream and decode the bit stream back to obtain the sequence of symbols. We shall see that although the coding efficiency is high, *arithmetic coding* is still not optimal due to some deficiencies inherent in the coding scheme. The rest of this lesson is devoted to *Lempel-Ziv coding,* which is another popular lossless compression scheme that assigns fixed-length codes to a variable group of symbols. This is a dictionary-based coding that finds its use in a variety of image representation formats.

4.1 Basic Principles of Arithmetic Coding

Like *Huffman coding*, this too is a Variable Length Coding (VLC) scheme requiring *a priori* knowledge of the symbol probabilities. The basic principles of *arithmetic coding* are as follows:

a) Unlike *Huffman coding*, which assigns variable length codes to a fixed group symbols (usually of length one), *arithmetic coding* assigns variable length codes to a variable group of symbols.

b) All the symbols in a message are considered together to assign a single arithmetic code word.

c) There is no one-to-one correspondence between the symbol and its corresponding code word.

d) The code word itself defines a real number within the half-open interval [0,1) and as more symbols are added, the interval is divided into smaller and smaller subintervals, based on the probabilities of the added symbols.

4.1.2 Algorithm for Arithmetic Coding

The steps of the encoding algorithm are as follows:

Step-1: Consider a range of real numbers in [0,1). Subdivide this range into a number of sub-ranges that is equal to the total number of symbols in the source alphabet. Each sub-range spans a real value equal to the probability of the source symbol.

Step-2: Consider a source message and take its first symbol. Find to which sub-range does this source symbol belongs.

Step-3: Subdivide this sub-range into a number of next level sub-ranges, according to the probability of the source symbols.

Step-4: Now parse the next symbol in the given source message and determine the next-level sub-range to which it belongs.

Step-5: Repeat step-3 and step-4 until all the symbols in the source message are parsed. The message may be encoded using any real value in the last sub-range so formed. The final message symbol is reserved as a special end-of-symbol message indicator.

4.2 Encoding a Sequence of Symbols using Arithmetic Coding

We are now going to illustrate the *arithmetic coding* process with a 6-symbol message $a_2a_1a_3a_4a_1a_2$ generated from a 4-symbol source $\{a_1, a_2, a_3, a_4\}$ having probabilities of the symbols $P(a_1) = 0.4, P(a_2) = 0.3, P(a_3) = 0.2$ and $P(a_4) = 0.1$.

Step-1: Draw a line, indicating an open interval of real numbers [0,1) and divide this line into a number of subintervals corresponding to the number of symbols in the source (in this example, 4 symbols- a_1, a_2, a_3, a_4). The lengths of the subintervals are equal to the probabilities of the symbols, starting with the interval [0,*P*(a_1)) from the bottom and ending with the subinterval [*P*(a_3),1) in this example. Click. On the graphics window, a vertical line will be displayed, having markings 0 at the bottom and 1 on top. The line will have 3 other markings at 0.4, 0.7 and 0.9. a_1, a_2, a_3, a_4 will be marked besides these subintervals from the bottom.

4[1].2.1a.swf

Fig 4.2.1

Step-2: Examine the first symbol in the sequence. Expand the corresponding subinterval to the full length of the line and mark the corresponding reduced overall range. Divide this line too into the subintervals as before and determine the positions of the subintervals corresponding to the reduced range. Click. The diagram will now get modified, adding a second vertical line that is pointed to by the interval a_2 from the first line. The top of this second line will be marked as 0.7 and the bottom as 0.4. This line will also have 3 other markings at the proportional positions and the numerical values corresponding to those positions should be indicated (0.52, 0.61 and 0.67). a_1, a_2, a_3, a_4 will be marked besides these subintervals from the bottom.

4[1].2.2a.swf

Fig 4.2.2

In this example, the first symbol being a_2, the corresponding subinterval [0.4, 0.7) is expanded for further encoding.

Step-3: Examine the next symbol in the sequence and repeat the operations performed in *step-2.* Repeat this step, until all the symbols in the sequence has been considered. Click once. The diagram will now get modified, adding a third vertical line that is pointed to by the interval a_1 from the second line. The top of the third line will be marked as 0.52 and the bottom as 0.4. This line will also have 3 other markings at the proportional positions and the numerical values corresponding to those positions should be indicated (0.448, 0.484, and 0.508). a_1, a_2, a_3, a_4 will be marked besides these subintervals from the bottom.

4[1].2.3a.swf

Fig 4.2.3

We have therefore added the next symbol a_1 for encoding and the corresponding subinterval [0.4, 0.52) is expanded for further encoding.
Click once more. A new vertical line will come up, pointed to by the interval a_3 from the previous line. Mark as before. The positions will now be (0.484, 0.4936, 0.5008, 0.5056 and 0.508).

4[1].2.4a.swf

Fig 4.2.4

We have now added the next symbol a_3
Click once more. A new vertical line will come up, pointed to by the interval a_4 from the previous line. Mark as before. The positions will now be (0.5056, 0.50656, 0.50728, 0.50776 and 0.508).

4[1].2.5a.swf

Fig 4.2.5

We have now added the next symbol a_4.

Click once more. A new vertical line will come up, pointed to by the interval a_1 from the previous line. Mark as before. The positions will now be (0.5056, 0.505984, 0.506272, 0.506464 and 0.50656).


4[1].2.6a.swf

Fig 4.2.6

We have now added the next symbols a_1 and a_2 and hence considered the entire message for encoding.

Step-4: Generate an arithmetic code corresponding to the last subinterval that remained. In this example, [0.505984, 0.506272) should now be arithmetically encoded. Any real number within this subinterval can be considered for encoding and we can choose 0.506, which requires a precision of three digits after the decimal point. Encoding the example sequence of length 6 symbols therefore requires (3/6), i.e., 0.5 digits/ symbol, which is close to its entropy of 0.554 digits/symbol. The bit stream corresponding to real number 0.506 can be generated by any appropriate coding schemes.

4.3 Decoding an Arithmetic-Coded Bit Stream

The decoding process of arithmetic coded bit stream is very straightforward. We should follow these steps:

Step-1: Identify the message bit stream. Convert this to the real decimal number and determine its position within the subintervals identified at the beginning of encoding process. The corresponding symbol is the first one in the message.

In this example, the message bit stream corresponds to decimal number 506 i.e., 0.506 in the real valued interval [0,1). This corresponds to the subinterval [0.4, 0.7) of a_2 in *step-1* of encoding (Section-4.2). The first symbol is therefore a_2.

Step-2: Consider the expanded subinterval of the previous decoded symbol and map the real number within this to determine the next subinterval and obtain the next decoded symbol. Repeat this step until the end-of-message indicator is parsed.

In this example, following *step-1* and *step-2,* we obtain the complete sequence $a_2 a_1 a_3 a_4 a_1 a_2$.

4.4 Coding Efficiency Limitations of Arithmetic-Coded Bit Stream

Although we may expect coding efficiency close to unity for *arithmetic coding*, its performance falls short of the Shannon's noiseless coding theorem bounds, due to the following limitations:

a) Every message ends with a special end-of-message symbol. This adds to an overhead in encoding and optimal performance can only be reached for very long messages.

b) Finite precision arithmetic also restricts the coding performance. This problem has been addressed by Langdon and Rissanen [1] through the introduction of a scaling and rounding strategy.

4.5 Basic Principles of Lempel-Ziv Coding

We now consider yet another popular lossless compression scheme, which is originally called *Lempel-Ziv coding*, and also referred to as *Lempel-Ziv-Welch (LZW)* coding, following the modifications of Welch for image compression. This coding scheme has been adopted in a variety of imaging file formats, such as the *graphic interchange format* (GIF), *tagged image file format* (TIFF) and the *portable document format* (PDF). The basic principles of this encoding scheme are:

a) It assigns a fixed length codeword to a variable length of symbols.

b) Unlike *Huffman coding* and *arithmetic coding*, this coding scheme does not require *a priori* knowledge of the probabilities of the source symbols.

c) The coding is based on a “dictionary” or “codebook” containing the source symbols to be encoded. The coding starts with an initial dictionary, which is enlarged with the arrival of new symbol sequences.

d) There is no need to transmit the dictionary from the encoder to the decoder. A Lempel-Ziv decoder builds an identical dictionary during the decoding process.

4.6 Encoding a sequence of symbols using Lempel-Ziv Coding

The encoding process will now be illustrated for one line of image having the following intensity values in sequence:

32 32 34 32 34 32 32 33 32 32 32 34

This indicates a near uniform intensity line strip with a little perturbation, that is commonly encountered in practical images.

To begin the encoding process, we consider a dictionary of size 256 locations (numbered 0 to 255) that contain entries corresponding to each pixel intensity value in the range 0-255. When we encounter the first pixel of intensity 32, we do not encode it and wait for the second pixel to arrive. When the next pixel intensity of 32 is encountered, we encode the first pixel as 32, corresponding to its dictionary location number, but at the same time, we make a new entry to the

dictionary at the location number 256 to include the newly detected sequence 32-32 (a short form of writing 32 followed by 32), so that next time we encounter this sequence 32-32, we are going to encode as 256, instead of two consecutive code words of 32. The arrival of the third pixel of intensity 34 makes a new sequence entry 32-34 at the location number 257 and the previous pixel (the second one) will be encoded as 32. The fourth pixel of intensity 32 will add another code entry at location number 258 to include the sequence 34-32 and the third pixel will be encoded as 34. The arrival of fifth pixel of intensity 34 will find a matching sequence of 32-34 at location number 257. We do not encode any pixel now and wait for the arrival of the sixth pixel of intensity 32. The previously found matching sequence of 32-34 will be encoded as 257. With the arrival of the sixth pixel, we also recognize the sequence 34-32 and add a triplet of sequence 32-34-32 to the dictionary. We can now follow the rest of the coding process illustrated in Table-4.1.

It is to be noted that the encoding process of the pixel intensities and the sequence of pixel intensities will require 9 bits, as long as the dictionary size is restricted to 2^9 (=512) locations. There is considerable bit savings when the pixel sequences find matching with a dictionary entry. For example, encoding the sequence 32-34 requires only 9-bits, as compared to 16-bits, which would have been required to independently encode the two pixels. Longer matching sequences (triplets and higher) results in substantial bit savings. In fact, it will be more and more efficient to encode larger file sizes. However, the size of the dictionary is a very crucial parameter. If it is too small, we are going to have less frequent matches and consequently less compression. On the other hand, too large a dictionary size would lead to increased bit consumption in encoding unmatched sequence of pixels.

Currently Recognized Sequence	Pixel being processed	Encoded Output	Dictionary Location (Code word)	Dictionary Entry
	32			
32	32	32	256	32-32
32	34	32	257	32-34
34	32	34	258	34-32
32	34			
32-34	32	257	259	32-34-32
32	32			
32-32	33	256	260	32-33
33	32	33	261	33-32
32	32			
32-32	32	256	262	32-32-32
32	34			
32-34		257		

Table 4.1 Lempel-Ziv coding example

4.7 Decoding a Lempel-Ziv encoded sequence

Decoding of Lempel-Ziv encoded sequence is quite straightforward. If we follow the example shown in Table-4.1, we find that the encoded sequence of symbols will be 32 32 34 257 256 33 256 257

Like the encoder, the decoder also starts with the initial dictionary entries of 0-255 only. When the decoder receives this sequence at the input, it first receives 32, which can be directly decoded as a pixel of intensity of 32. Next received symbol, as well as the next pixel is also 32. However, the dictionary now makes a new entry of 32-32 sequence at the location 256. Next received symbol, as well as the next pixel is 34. The dictionary now enters the sequence 32-34 at the location 257. Having made this entry, when the next received symbol is 257, the decoder detects two consecutive pixels 32, followed by 34. The next received symbol is 256, i.e., 32 followed by 32. The reader should verify that this way we decode the sequence of pixels as 32 32 34 32 34 32 32 33 32 32 32 34, which is the original sequence.

Questions

NOTE: The students are advised to thoroughly read this lesson first and then answer the following questions. Only after attempting all the questions, they should click to the solution button and verify their answers.

PART-A

A.1. Explain why Huffman coding fails to achieve the optimal coding efficiency
A.2. Explain why arithmetic coding can achieve better coding efficiency than Huffman.
A.3. State the basic principles of arithmetic coding.
A.4. Make a comparison between Huffman coding and arithmetic coding.
A.5. Discuss the limitations of arithmetic coding in achieving optimal coding efficiency.
A.6. State the basic principles of Lempel-Ziv coding.

PART-B: Multiple Choice

In the following questions, click the best out of the four choices.
Radio buttons will be provided to the left of each choice. Only one out of the four buttons can be chosen.

B.1 Which of the following statements is not true for arithmetic coding

(A) Integral number of bits is assigned to each symbol.

(B) A real number in the interval [0, 1) indicates the entire coding sequence.

(C) Coding requires *a priori* knowledge of the probabilities of source symbols.

(D) Longer sequence of source symbols leads to longer code words.

B.2 The coding efficiency of arithmetic coding is expected to improve if

(A) The interval of real numbers is increased to [0, 10)

(B) Shorter sequence of source symbols is encoded.

(C) Longer sequence of source symbols is encoded.

(D) None of the above.

B.3 Which one of the following leads to a performance limitation of arithmetic coding

(A) There is no one-to-one mapping between source symbols and code words.

(B) It is a variable length coding.

(C) The entire sequence of source symbols in a message is encoded together.

(D) Use of finite precision arithmetic.

B.4 A source of 4 symbols a_1, a_2, a_3, a_4 having probabilities $P(a_1) = 0.5, P(a_2) = 0.25, P(a_3) = P(a_4) = 0.125$ is used for arithmetic coding. The source symbol sequence a_2a_1 will correspond to the interval (mark the closest answer)

(A) [0.25, 0.375)

(B) [0.5, 0.625)

(C) [0.75, 0.875)

(D)[0,1)

B.5 A sequence of three symbols is arithmetically encoded by the above source to a real number 0.8. The original sequence is

(A) $a_4 a_2 a_3$

(B) $a_2 a_4 a_3$

(C) $a_3 a_1 a_3$

(D) $a_3 a_1 a_4$

B.6 Which of the following statements is not true for Lempel-Ziv coding

(A) It is a fixed-length coding for variable-length symbol sequence.

(B) It does not require *a priori* knowledge of the source symbol probabilities.

(C) Larger image/file size leads to poorer compression.

(D) Decoder dictionary can be derived from the encoded sequence.

B.7 A very large-sized dictionary in Lempel-Ziv coding has the following advantage

(A) Group of symbols can find more frequent matches.

(B) Faster encoding and decoding is possible.

(C) Isolated symbols without groups can be represented with less number of bits.

(D) Large-sized images and files can be encoded.

B.8 A Lempel-Ziv dictionary starts with two entries – “0” and “1”. The dictionary size after parsing the symbol stream 00101100 is

(A) 4

(B) 5

(C) 6

(D)7

B.9 Consider the Lempel-Ziv coding example presented in Table-4.1. The compression ratio (compressed bits : uncompressed bits) achieved for the example sequence is

(A) 1:4

(B) 3:4

(C) 7:8

(D) 5:6

PART-C: Problems

C-1.
(a) Construct an arithmetic code in real decimal number for the word "APPLE" formed out of a 4-symbol alphabet – "A", "P", "L" and "E" having probabilities 0.2, 0.4, 0.2 and 0.2 respectively.

(b) Compare the codeword length in digits/symbol with its entropy expressed in the same units. Calculate its coding efficiency.

C-2. A Lempel-Ziv dictionary having maximum size of 32 has 10 entries to start with – the decimal numbers "0" to "9".

(a) Encode the digit sequence 8,2,8,2,2,8,2,2,2,8 using Lempel-Ziv coding and determine the compression ratio (compressed bits: uncompressed bits).
(b) Repeat part (a) for the digit sequence 9,7,2,0,6,1,5,3,4,8.
(c) Why is the compression ratio better in (a), as compared to (b)?

SOLUTIONS

A.1 Huffman coding fails to achieve optimal coding efficiency because it encodes one symbol at a time and each symbol translates into an integral number of bits

A.2

A.3

A.4

A.5

A.6

B.1 (A) B.2 (C) B.3 (D) B.4 (B) B.5 (D)

B.6 (C) B.7 (A) B.8 (D) B.9 (B).

C.1

C.2

REFERENCES

1. Langdon, G.C. and Rissanen, J.J., “Compression of Black-White Images with Arithmetic Coding”, IEEE Transactions on Communications, Vol.COM-29, No.6, pp.858-867, 1981.