We may already have heard of codes used to represent letters and symbols. For instance, the ASCII code describes all the uppercase and lowercase letters in the English alphabet using a sequence of zeros and ones. Morse code uses a sequence of dashes (long pulses) and dots (short pulses).
Entropy of real signals

Recall: real signals have a lot of redundancy. This means that data *compression* can reduce the memory required to store or transmit the contained information without significant losses.
Fixed and variable length codes

In a fixed length code, there are the **same number of bits** in each *message* used to represent a *symbol* (single piece of information).

A variable length code uses **different numbers of bits** to represent different pieces of information (like letters or image intensities).

Fixed length codes have a **simple decoding process** – the beginning and the end of each *code word* or message is well defined.

But, fixed length codes are very **wasteful**. You use the same number of bits to represent things that occur frequently and infrequently.
Fixed length codes – a waste of space!

A fixed length code would use the same number of bits to describe each image pixel:

The Persistence of Memory, 1931 - Salvador Dali

No. 4, 1964 - Mark Rothko
Examples of Fixed Length Codes

A number stored in a computer (e.g., 32-bit integer)

Credit card magnetic stripe

DNA and RNA – a sequence of 3 nucleic acids represent a codon for a specific amino acid when constructing a protein

American Standard Code for Information Interchange (ASCII)

- Originally a 7-bit code for alphanumeric symbols and some keyboard commands like ENTER or ESC.
- 7 bits gives you $2^7 = 128$ possible symbols (seems like a lot).
- Extended ASCII allows 8 bits, permits some additional characters, including those used in some other languages. Beyond extended ASCII, there are other “character sets” to cover even more languages. Unicode, usually 16 bits, is a standard scheme for unifying all these.
- What is used to store text data on a phone or computer?
ASCII

Uppercase B is:

- 01000010 in binary
- 64+2=66 in decimal
- 42 in hex (groups of four bits)
- 102 in octal (groups of three starting from right)

- We’ll talk more about these different number systems later in the course.
Variable-length Code

Example: Morse Code
- The longest-lasting electrical signaling code in existence

Variable length coding is an essential step in encoding audio and images.
- MP3 uses a variable-length coding method called the Huffman code (later today).

Idea: Use short code words for common (high probability) symbols. Use longer code words to represent less probable symbols.
- Advantages: less storage space, faster transmission
- Disadvantage: more difficult to decode (computation/circuit cost)
Variable-length Code

Other examples of variable-length codes:
- English language (words have different lengths)
- Fax machine
- Compression programs like WinZip (Lempel-Ziv)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Word</th>
<th>Rank</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>the</td>
<td>901</td>
<td>shot</td>
</tr>
<tr>
<td>2</td>
<td>of</td>
<td>902</td>
<td>poet</td>
</tr>
<tr>
<td>3</td>
<td>and</td>
<td>903</td>
<td>seven</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>904</td>
<td>moving</td>
</tr>
<tr>
<td>5</td>
<td>to</td>
<td>905</td>
<td>mass</td>
</tr>
<tr>
<td>6</td>
<td>in</td>
<td>906</td>
<td>plane</td>
</tr>
<tr>
<td>7</td>
<td>is</td>
<td>907</td>
<td>proper</td>
</tr>
<tr>
<td>8</td>
<td>be</td>
<td>908</td>
<td>propose</td>
</tr>
<tr>
<td>9</td>
<td>that</td>
<td>909</td>
<td>drink</td>
</tr>
<tr>
<td>10</td>
<td>was</td>
<td>910</td>
<td>obviously</td>
</tr>
</tbody>
</table>
Variable-length coding: Fax

Fax Machines... going back to 1840’s

- The fax machine will speed up during the white space and slow down where a picture exists.
- The fax machine actually scans the image and breaks the image up into pixels (picture elements) of size 125 microns x 250 microns.
- Each pixel is either light (0) or dark (1).
Run-length coding

To speed up transmissions, fax machines use a run-length code.

Suppose that a scan line has...

- 104 consecutive white pixels then 25 black pixels, 50 white, 10 black, 50 white
- This could be encoded as W104,25,50,10,50

Since long runs of black and white are common, we’re assigning short code words to them

- But how to encode the numbers? Use short codes words for common values, long for uncommon.
Entropy and average code length

What is the smallest number of bits/symbol for storing data produced by a source? How is this related to entropy?

- Example: Storing DNA data. Symbols = {A, C, G, T}. If we map A→00, C→01, G→10, T→11, we use 2 bits/symbol. If all symbols are equally likely, then H=log₂4=2 bits. So our code matches the entropy of the signal.

- Example: Unequal probabilities. What if p(A)=1/2, p(C)=1/4, p(G)=1/8, p(T)=1/8. Then the entropy is H=1.75 bits, which is less than 2. Can we encode this source so that we use 1.75 bits/symbol? Yes, we can use a variable length code:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probability</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>1/8</td>
</tr>
<tr>
<td>Code</td>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>Information</td>
<td>log 2=1</td>
<td>log 4=2</td>
<td>log 8=3</td>
<td>log 8=3</td>
</tr>
</tbody>
</table>

- What is the average code length? Note: code word length matches information for symbol here.
Entropy and average code length

If a message can take values $x$ with probability $P(x)$, the entropy describes the expected information from the message.

- The minimum number of bits per message needed to store $K$ messages converges to $H$ as $K$ approaches infinity.
- Note: each message is encoded independently. (Why is this important?)
- Compression techniques that achieve this limit include Huffman* and arithmetic coding.

* Huffman codes achieve limit if $1/P(x_i)$ is a power of two for all $x_i$. Otherwise, the codes just get close to the limit.
Prefix code – dealing with variable length

One difficulty with variable length codes is knowing when one code word ends and another begins. A prefix code is one way to solve this problem.

- A prefix code assigns a unique bit string to each possible symbol in the message such that no code is the prefix of another code.

- Example (digits 0-9):
  0 = “0000”, 1 = “0001”, 2 = “001”,
  3 = “0100”, 4 = “0101”, 5 = “0110”,
  6 = “0111”, 7 = “100”, 8 = “101”,
  9 = “11”

Decode left-to-right “0010011100000100”: 22903
Prefix code optimality

Suppose a prefix code has code words of length $l(x)$ for each symbol $x$.

- The average length of a prefix code is $\sum_x p(x)l(x)$.

To minimize the number of bits, more likely symbols should have shorter bit strings. (just like run-length encoding!)

Goal: use bit string length that is nearly the information value for that symbol (e.g., if $I_A = 2$, then use 2 bits).

- Then the average length per code word approaches the entropy of the original signal.
- How do we do this? Answer: Huffman coding
Huffman coding

A Huffman code is a prefix code with a *binary tree* structure. It is generated on-the-fly using the following algorithm:

1. Initialize: each symbol $x_i$ gets its own tree with weight $P(x_i)$.
2. Iterate: choose pair of trees with lowest weights and combine into single tree with summed weight. Repeat until a single tree remains.
3. Assign left branches “0” or “1” and right branches the other.

Let’s see an example that will illustrate what a binary tree is.
Example

Suppose \( p(0) = 1/4, p(1) = 1/4, p(2) = 1/4, p(3) = 1/8, p(4) = 1/16, p(5) = 1/16. \) Then:

<table>
<thead>
<tr>
<th>Trees</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w )</td>
<td>1/4</td>
<td>1/4</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
<td>1/16</td>
</tr>
</tbody>
</table>
Example

Suppose $p(0) = 1/4$, $p(1) = 1/4$, $p(2) = 1/4$, $p(3) = 1/8$, $p(4) = 1/16$, $p(5) = 1/16$. Then:

Final Result:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code word</th>
<th>$P(x)$</th>
<th>$I_x$</th>
<th>$I(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>1/4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1/4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1/4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>1/8</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1110</td>
<td>1/16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1111</td>
<td>1/16</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Example

Suppose \( p(0) = 1/4, p(1) = 1/4, p(2) = 1/4, p(3) = 1/8, p(4) = 1/16, p(5) = 1/16 \). Then:

- Our result: average code word length = \( H = 2.375 \) bits
- What would have been the fixed code length? (6 symbols → 3 bits)

- If we were to encode 1000 of these symbols, how much memory would we save on average?

- Note: Huffman coding average word length won’t exactly match \( H \), unless the probabilities are all powers of two. But, it will still be a lot closer for nonuniform data than fixed length codes.
Other coding methods

Arithmetic coding – prefix code generated on the fly from input data
- Generates codes for sequences of symbols.
- Asymptotically achieves entropy limit, even if relative frequencies in data are not exactly powers of two.
- Note, we have to know or measure the probabilities of different symbols we encounter. This may not always be given. Fortunately, adaptive methods are available...

Shannon-Fano coding – a simple prefix code generator
- Guaranteed to generate a prefix code that is reasonably good, but not optimal
- Also constructs a binary tree, by sorting and splitting sets of symbols by relative frequency
Computing probabilities

These coding methods are probabilistic; they require probabilities for each symbol.

Encoding a static document: measure probabilities of each symbol over the whole document/signal and send along with encoded message.

Dynamic estimation: update probabilities on the fly; code for a symbol depends on distribution up to that point. No need to send probabilities along
Decoding messages

Storing or transmitting these bit strings is called encoding.

The act of reconstructing the original message is called *decoding*.

**Lossless/lossy encoding**
- Encoding is *lossless* if it is guaranteed to recover the original message exactly.
- Encoding is *lossy* if the recovery error (distortion) is nonzero (likely small in practice).
- Examples?
Lempel-Ziv compression

Lempel-Ziv (LZ) compression builds a table of strings; it does not use or track probabilities. It also operates “on the fly”, constructing a table reading one symbol at a time.
- The original 1978 LZ algorithm builds a table of strings of increasing length from the input.
- Variants include Lempel-Ziv-Welch (LZW), the sliding window (LZ 1977), and others.
Lempel-Ziv compression

Here are steps of the LZ compression algorithm:

1. Initialize: Start with dictionary containing alphabet \(D\), empty string \(w\).
2. Iterate: Read a character \(x\), and append to form \(wx\). If \(wx\) is in \(D\), let \(w = wx\) and don’t output anything. Otherwise, output new \(w\) string, add \(wx\) to \(D\), and set \(w = x\).
3. Repeat step 2 until the string is done, and output the final \(w\) string.
4. Compute relative probabilities of strings in dictionary and encode each using a method like Huffman coding.
Example

Encode the string “abacaba”:

<table>
<thead>
<tr>
<th>w (previous)</th>
<th>x</th>
<th>wx</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>“”</td>
<td>a</td>
<td>a</td>
<td>“a” is in D; w = wx</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>ab</td>
<td>“ab” not in D; output “a”, add “ab” to D; w = x</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>ba</td>
<td>“ba” not in D; output “b”, add “ba” to D; w = x</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>ac</td>
<td>“ac” not in D; output “a”, add “ac” to D; w = x</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>ca</td>
<td>“ca” not in D; output “c”, add “ca” to D; w = x</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>ab</td>
<td>“ab” is in D; w = wx</td>
</tr>
<tr>
<td>ab</td>
<td>a</td>
<td>aba</td>
<td>“aba” not in D; output “ab”, add “aba” to D; w=x</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>output “a”</td>
</tr>
</tbody>
</table>

- **Output string divides “abacaba” into:** a b a c (ab) a
- **Assign a number to each string used (e.g., a = 0, b = 10, c = 100, ab = 101).**
Lempel-Ziv-Welch

Like the 1978 LZ method, Lempel-Ziv-Welch (LZW) also constructs a dictionary as it reads the string.

However, LZW only appends an entry to the dictionary when its prefix (e.g., “Jo” for “Joe”) is already in the dictionary. Thus, the dictionary grows a bit more slowly than LZ.

For either LZ or LZW, when the dictionary gets too large, methods like GIF compression throw away the dictionary and start over. This avoids retaining too many words that are not used throughout the entire signal.
LZW procedure

1. Initialize: construct a dictionary containing all possible symbols as strings.

2. Iterate: If the current input symbol is in the dictionary, see if the first pair of input symbols is in the dictionary. Then, see if the first triple of input symbols is in the dictionary. Continue until we have a sequence of symbols not in the dictionary. Output the longest matching string in the dictionary as our next string and add the new string to the dictionary. (Note: prefix-based string construction.)

3. Repeat until entire signal is processed.
Example

Encode the string “abacaba”: dictionary initially contains “a”, “b”, “c”.

- Output string is still: a b a c (ab) a
- Assign a number to each string used.
Announcements

Next time: Quantization and data compression

Homework #5 out today (due on March 7).

Lab 5a today (two-part lab).