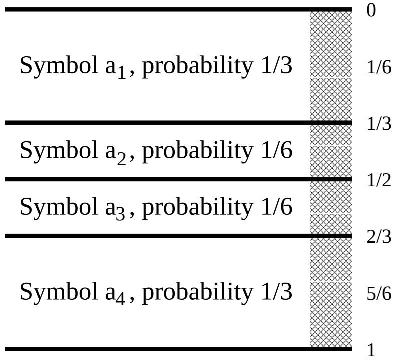
Lecture 7:

Stream Codes & Arithmetic Coding

October 2, 2006

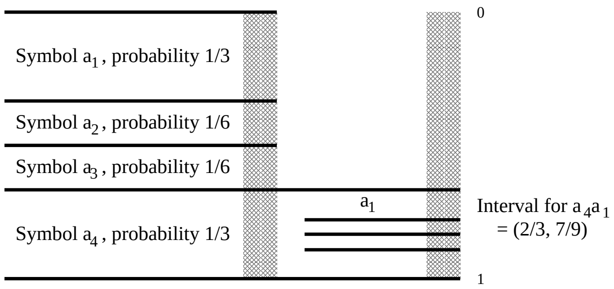
Viewing the Codespace as the Interval [0,1) 1

- Even if we could solve the problem of how to generate codewords corresponding arbitrary divisions of codespace, how can we handle symbols with probabilities like $1/3$, which aren't multiples of 2^{-k}?
- A solution: Consider the codespace to be the interval of real numbers between 0 and 1. Example:



Key Concept: Encode Blocks by Subdiving Further 2

- Consider the source with probabilities $\{1/3, 1/6, 1/6, 1/3\}$.
- Suppose we want to encode blocks of two symbols from this source. We can do this by just subdividing the interval corresponding to the first symbol in the block, in the same way as we subdivided the original interval.
- Here's, how we encode the block $a_4\,a_1$:



Encoding Large Blocks as Intervals 3

- A general scheme for encoding a block of N symbols, $a_{i_1}, \ldots, a_{i_N}$

1) Initialize the interval to $\left[u^{(0)},\ v^{(0)}\right)$; $u^{(0)} = 0$ and $v^{(0)} = 1$.

2) For $k = 1, \ldots, N$:

Let $u^{(k)} = u^{(k-1)} + \left(v^{(k-1)} - u^{(k-1)}\right) \sum_{j=1}^{i_k - 1} p_j$

Let $v^{(k)} = u^{(k)} + \left(v^{(k-1)} - u^{(k-1)}\right) p_{i_k}$

3) Output a codeword that corresponds (somehow) to the final interval, $\left[u^{(N)},\ v^{(N)}\right)$.

- This scheme is known as *arithmetic coding*, since codewords are found using arithmetic operations on the probabilities.

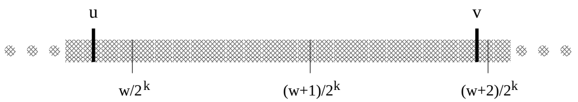
Finding a Codeword for an Interval

- The last step requires that we be able to find a codeword to represent the final interval. We'll insist on an instantaneous code, for which no codeword is a prefix of another codeword.
- Observation: any binary codeword defines a number in $[0, 1)$, found by putting a "binary point" at its left end. Eg, the codeword 101... defines the number $1 \times (1/2) + 0 \times (1/4) + 1 \times (1/8) \ldots$
- Based on this, we'll choose a codeword such that:
  - The codeword defines a point in the final interval.
  - If we added any string of bits to the end of the codeword, it would still define a point in the final interval.

  This is equivalent to finding the largest interval of the form $[w/2^k, (w+1)/2^k)$ that fits entirely within $[u, v)$.
- Codewords chosen in this way will form a prefix code for the blocks.

How Long Will the Codewords Be?

- Here's a picture of how we pick a codeword for an interval:



- Here, the interval $[w/2^k,\ (w+1)/2^k)$ fits entirely within $[u,\ v)$, the final interval found when encoding the block. We can therefore use the k-bit binary representation of w as the codeword for this block.
- This can only be true if $v - u \geq 1/2^k$. Also, we will always be able to find such a codeword of length k if $v - u \geq 2/2^k = 1/2^{k-1}$.
- **Conclusion:** We can pick a codeword of length k for a block of probability $p = (v - u)$ if $k \geq \log(1/p) + 1$.
  So codewords need be no longer than $\lceil \log(1/p) \rceil + 1$.

Arithmetic Coding Gets Close to the Entropy

- We encode symbols from $\mathcal{A}_X$ in blocks of size N (ie, we use the N-th extension, $\mathcal{A}_X^N$), with N being quite large.
- Assuming independence, the probability of the block $a_{i_1}, \ldots, a_{i_N}$ is $p_b = p_{i_1} \cdots p_{i_N}$.
- We can find the interval for this block *without* explicitly considering all possible blocks by subdividing $(0, 1)$ N times according to the i_k.
- We can then find a binary codeword for this block no longer than $\lceil \log(1/p_b) \rceil + 1$ which is less than $\log(1/p_b) + 2$.
- The average codeword length for blocks will be less than

$$2 + \sum_b p_b \log(1/p_b) \ = \ 2 + H(\mathcal{A}_X^N) \ = \ 2 + NH(\mathcal{A}_X)$$

- The average number of bits transmitted per symbol of $\mathcal{A}_X$ will be less than $H(\mathcal{A}_X) + 2/N$; without ever considering all blocks.

How Well it Works (So Far)

- **Big advantage:**
  We can get arbitrarily close to the entropy using big blocks, without an exponential growth in complexity with block size.
- **Big disadvantage (so far):**
  If we use big blocks, most block probabilities will be tiny. Therefore, the interval corresponding to the block will be very narrow. To represent this interval in a computer program using the procedure we described, we would have to use highly precise arithmetic.
- In fact, the amount of precision needed for an accurate approximation will go up linearly with blocksize, and the time for arithmetic operations involving such operands will also grow linearly.
- *Fortunately, this disadvantage can be overcome.*

Stream Codes: Transmitting Bits as We Go 8

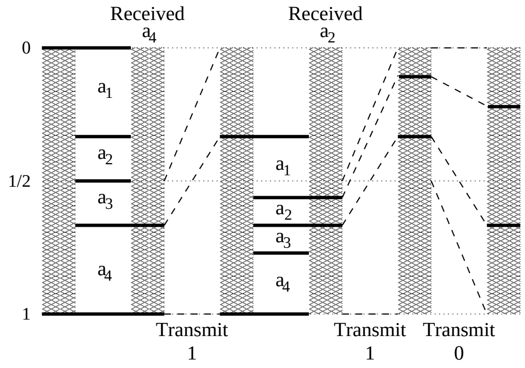
- The problem of needing high-precision arithmetic makes aritmetic coding potentially impractical. We'll try to solve it by transmitting bits as soon as they are determined.
- **Example:** After looking at the first few symbols in our block, our interval has been reduced to $[0.625,\ 0.875)\ =\ [0.101_2,\ 0.111_2)$.
- *Any* number in this interval that we might eventually transmit will start with a 1 bit. So we can transmit this bit immediately, without even looking at what symbols come next!
- This scheme is called *stream coding* because we just receive an incoming stream of source symbols and output bits of the encoding as we compute them.
- There is no need for an explicit block length anymore! We are in effect transmitting the entire message as a single block by specifying (to adequate precision) the corresponding interval in $[0, 1)$.

Expanding the Interval After Transmitting a Bit 9

- Once we transmit a bit that is determined by the current interval, we can throw that bit away, and then expand the interval by moving the "bit point" one place to the right and doubling.
- **Example:** Continuing from the previous slide, the interval $[0.625,\ 0.875)\ =\ [0.101_2,\ 0.111_2)$ results in transmission of a 1. We then throw out the 1, and double the bounds, giving the interval $[0.010_2,\ 0.110_2)$.
- Hopefully, expanding the interval will allow us to use numerical representations of the bounds, u and v, that are of lower precision.

Picture Of How it Works 10

Suppose we are encoding symbols from the alphabet $\{a_1,\ a_2,\ a_3,\ a_4\}$, with probabilities $1/3,\ 1/6,\ 1/6,\ 1/3$.
Here's how the interval changes as we encode the message $a_4,\ a_2,\ \ldots$



Arithmetic Coding Without Blocks (ver 1.0) 11

1) Initialize interval $[u,\ v)$ to $u = 0$ and $v = 1$.

2) For each source symbol, a_i, in turn:
   Compute $r = v - u$.
   Let $u = u + r \sum_{j=1}^{i-1} p_j$. Let $v = u + rp_i$.
   While $u \geq 1/2$ or $v \leq 1/2$:
      If $u \geq 1/2$:
         Transmit a 1 bit.
         Let $u = 2(u - 1/2)$ and $v = 2(v - 1/2)$.
      If $v \leq 1/2$:
         Transmit a 0 bit.
         Let $u = 2u$ and $v = 2v$.

3) Transmit enough final bits to specify a number in $[u,\ v)$.

Precision Might Still be a Problem... 12

- We hope that by transmitting bits early and expanding the interval, we can avoid tiny intervals, requiring high precision to represent.
- *Problem:* What if the interval gets smaller and smaller, *but it always includes 1/2?*
- For example, as we encode symbols, we might get intervals of:
  $[0.00000_2,\ 1.00000_2)$
  $[0.01010_2,\ 0.11001_2)$
  $[0.01101_2,\ 0.10100_2)$
  $[0.01111_2,\ 0.10010_2)$
  ...
- Although the interval is getting smaller and smaller, we still can't tell whether the next bit to transmit is a 0 or a 1.

A Solution 13

- When a narrow interval straddles $1/2$, it will have the form

$$[0.01xxx,\ 0.10xxx)$$

- So although we don't know what the next it to transmit is, we *do* know that the bit transmitted after the next will be the opposite.
- We can therefore expand the interval around the *middle* of the range, remembering that the next bit output should be followed by an opposite bit.
- If we need to do several such expansions, there will be several opposite bits to output.

Arithmetic Coding Without Blocks (ver 1.1) 14

1) Initialize the interval $[u,\ v)$ to $u = 0$ and $v = 1$.
   Initialize the "opposite bit count" to $c = 0$.

2) For each source symbol, a_i, in turn:
   Compute $r = v - u$.
   Let $u = u + r \sum_{j=1}^{i-1} p_j$. Let $v = u + rp_i$.
   While $u \geq 1/2$ or $v \leq 1/2$ or $u \geq 1/4$ and $v \leq 3/4$:
      If $u \geq 1/2$:
         Transmit a 1 bit followed by c 0 bits. Set c to 0.
         Let $u = 2(u-1/2)$ and $v = 2(v-1/2)$.
      If $v \leq 1/2$:
         Transmit a 0 bit followed by c 1 bits. Set c to 0.
         Let $u = 2u$ and $v = 2v$.
      If $u \geq 1/4$ and $v \leq 3/4$:
         Set c to $c + 1$.
         Let $u = 2(u-1/4)$ and $v = 2(v-1/4)$.

3) Transmit enough final bits to specify a number in $[u,\ v)$.

What Have We Gained? 15

- By expanding the interval in this way, we ensure that the size of the (expanded) interval, $v - u$, will always be at least 1/4.
- We can now represent u and v with a *fixed* amount of precision — we *don't* need more precision for longer messages.
- We will use a *fixed point* (scaled integer) representation for u and v.
- Why not floating point?
  - Fixed point arithmetic is faster on most machines.
  - Fixed point arithmetic is well defined. Floating point arithmetic may vary slightly from machine to machine. The effect? Machine B might not correctly decode a file encoded on Machine A!