Getting Close to the Entropy Using Arithmetic Coding

- We encode symbols from $\mathcal{S}$ in blocks of size n, where n is quite large.
- Assuming independence, the probability of the block $s_{i_1}, \ldots, s_{i_n}$ is $p_b = p_{i_1} \cdots p_{i_n}$.
- We can find the interval for this block by subdividing $(0,1)$ n times — *without* explicitly considering all possible blocks.
- We can then find a binary codeword for this block that is no longer than

$$\lceil \log(1/p_b) \rceil + 1 < \log(1/p_b) + 2$$

- The average codeword length for blocks will be less than

$$2 + \sum_b p_b \log(1/p_b) \;=\; 2 + H(\mathcal{S}^n) \;=\; 2 + nH(\mathcal{S})$$

- The average number of bits transmitted per symbol of $\mathcal{S}$ will be less than $H(\mathcal{S}) + 2/n$.

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, many block probabilities will be tiny. For the procedure to work, we will have to use highly precise arithmetic.

(The number of bits of precision needed for a good approximation will go up linearly with blocksize, and the time for arithmetic involving such operands will also grow linearly.)

Fortunately, this disadvantage can be overcome.

Encoding the Whole Message, Transmitting Bits as We Go

To get close to the entropy, we need big blocks. Why not go all the way? — Just transmit the entire message as one block.

The problem that we need high-precision arithmetic is now even worse. We'll try to solve it by transmitting bits as soon as they are determined.

Example: After coding some symbols, our interval is $[0.625, 0.875) \;=\; [0.101_2, 0.111_2)$. *Any* number in this interval that we might eventually transmit will start with a 1 bit. We can transmit this bit immediately.

Expanding the Interval After Transmitting a Bit

Once we transmit a bit that is determined by the current interval, we can throw it away, and then expand the interval by 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, giving the interval $[0.001_2, 0.011_2)$, and double the bounds, giving $[0.010_2, 0.110_2)$.

Expanding the interval will allow us to use representations of l and u that are of lower precision.

Arithmetic Coding Without Blocks (Preliminary Version)

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

2) For each source symbol, s_i, in turn:

   Compute $r = u - l$.

   Let $l = l + r \sum_{j=1}^{i-1} p_j$.

   Let $u = l + rp_i$.

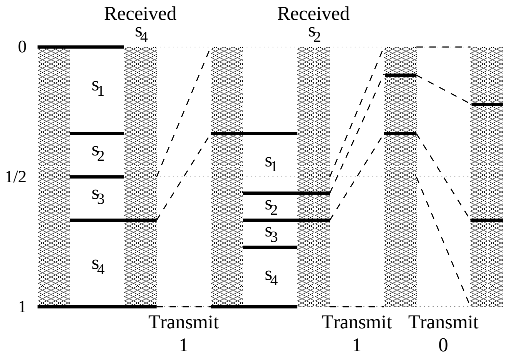
   While $l \geq 1/2$ or $u \leq 1/2$:
   - If $l \geq 1/2$:
     - Transmit a 1 bit
     - Let $l = 2(l - 1/2)$ and $u = 2(u - 1/2)$.
   - If $u \leq 1/2$:
     - Transmit a 0 bit
     - Let $l = 2l$ and $u = 2u$.

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

A Picture Of How it Works

Suppose we are encoding symbols from the alphabet $\{s_1, s_2, s_3, s_4\}$, with probabilities 1/3, 1/6, 1/6, 1/3.

Here's how the interval changes as we encode the message s_4, s_2, ...



A Problem

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

$$[0.00000_2,\ 1.00000_2)$$
$$[0.01010_2,\ 0.11001_2)$$
$$[0.01101_2,\ 0.10100_2)$$
$$[0.01111_2,\ 0.10010_2)$$
$$\cdots$$

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

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 (Revised Version)

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

2) For each source symbol, s_i, in turn:

   Compute $r = u - l$.

   Let $l = l + r \sum_{j=1}^{i-1} p_j$.

   Let $u = l + rp_i$.

   While $l \geq 1/2$ or $u \leq 1/2$ or $l \geq 1/4$ and $u \leq 3/4$:
   - If $l \geq 1/2$:
     - Transmit a 1 bit followed by c 0 bits
     - Set c to 0
     - Let $l = 2(l - 1/2)$ and $u = 2(u - 1/2)$.
   - If $u \leq 1/2$:
     - Transmit a 0 bit followed by c 1 bits
     - Set c to 0
     - Let $l = 2l$ and $u = 2u$.
   - If $l \geq 1/4$ and $u \leq 3/4$:
     - Set c to $c + 1$
     - Let $l = 2(l - 1/4)$ and $u = 2(u - 1/4)$.

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

What Have We Gained?

By expanding the interval in this way, we ensure that the size of the (expanded) interval, $u - l$, will always be at least 1/4.

We can now represent l and u 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 l and u.

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!