Problem Set 7

published: 9 June 2021
discussion: 14 June 2021

Data Compression With Deep Probabilistic Models
Prof. Robert Bamler, University of Tuebingen

Course material available at https://robamler.github.io/teaching/compress21/

Problem 7.1: Streaming ANS I — Implementation

The accompanying jupyter notebook contains our naive implementation of the Asymmetric Numeral Systems (ANS) algorithm. As we discussed at the end of the lecture, this implementation is not yet practically useful because the runtime cost for encoding k symbols scales quadratically in k. In this problem, you will implement the last missing piece of the ANS algorithm, which will reduce the runtime cost to just $O(k)$.

Problem. The reason why the current implementation suffers from $O(k^2)$ runtime cost is that the cost of each one of the k encoding or decoding operations is proportional to the amount of data that has already been compressed. Consider, e.g., the following line in the `decode` method:

```
self.compressed = self.compressed * scaled_probabilities[symbol] + z
```

Here, `self.compressed` is an integer whose binary expansion is the compressed representation of the symbols you've encoded so far. Thus, if you've already compressed, say, one megabyte worth of information content then `self.compressed` will be an eight-million-bit long integer. Multiplying it with `scaled_probabilities[symbol]` as in the above statement will, in general, change *all* of these eight million bits, which is expensive.

Fractional Bit Shifts. Figure 1 (i) provides a suggestion for a more intuitive interpretation of the encoding and decoding operations. Picture some buffer that stores the value of `self.compressed`. The allocated memory for this buffer will typically be a few bits larger than precisely necessary, and so the binary expansion of `self.compressed` will be padded with some leading zero bits (blue). The *valid bits* (i.e., everything except leading zeros) fit into a region of size $\log_2($`self.compressed`$)$ (up to rounding effects) that is *right-aligned* within the allocated memory (red box in Figure 1 (i)).

Let's now look at the above line of code, which multiplies `self.compressed` with `scaled_probabilities[symbol]`. We will denote the latter as $m(x_i)$ from here on in parity with the lecture notes. If $m(x_i)$ is an integer power of two, i.e., if $m(x_i) = 2^r$ for some $r \in \mathbb{N}$ then the multiplication with $m(x_i)$ results in a simple left shift by $\log_2 m(x_i) = r$ bits (illustrated for $r = 3$ in Figure 1 (i)). Alternatively, we could simply append r zero bits at the end of the allocated memory, which would amount to only a constant run-time cost.[1]

[1] Technically, the *amortized* cost is constant, provided that we use a dynamic array (aka a "vector").

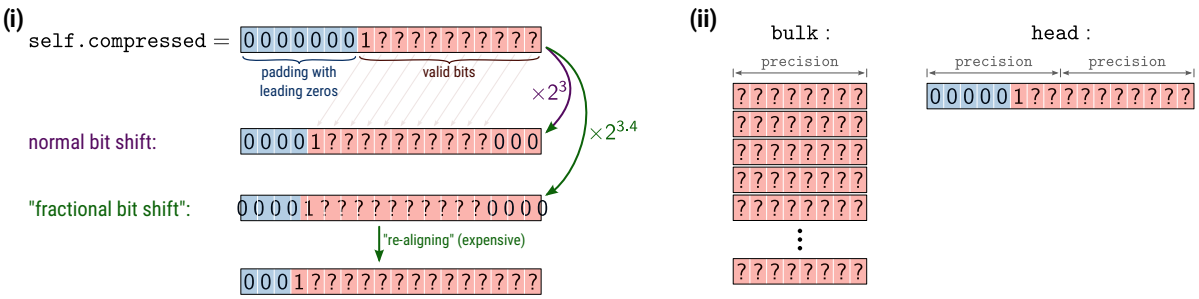

Figure 1: (i) Multiplying an integer by a power of two results in a bit shift. Analogously, we can interpret a general multiplication as a bit shift by a fractional number of bits, followed by an (expensive) operation that has to "re-align" all valid bits to integer offsets. (ii) separation of the compressed representation into a `bulk` and a `head`.

Now consider the general case where $m(x_i)$ is not necessarily an integer power of two. Multiplying $m(x_i)$ to `self.compressed` grows the number of valid bits again by roughly $\log_2 m(x_i)$ bits (up to rounding effects). In fact, it is useful to think of the multiplication as a left bit shift by the non-integer amount of $\log_2 m(x_i)$ bits. In this picture, after the left-shift, the bits will *not* land at any positions that can actually store bits but they will instead land somewhere in-between, i.e., at a fractional alignment (see Figure 1 (i), which illustrates a bit shift by $\log_2 m(x_i) = 3.4$ bits). You can think of the multiplication with $m(x_i)$ as a fractional bit shift by $\log_2 m(x_i)$, followed by the task of "re-aligning" the bits to valid positions, which is expensive since it has to recalculate every valid bit.

To summarize,

- When we encode a symbol, we shift the existing bits on the buffer to the left by the information content of the symbol, and we then write the symbol into the generated space at the right (i.e., least significant) end of the buffer.
- When we decode a symbol, we read it off from the right end and then shift all bits to the right by the consumed information content.
- Integer bit shifts (aka normal bit shifts) are cheap, even on very large numbers, because we can implement them by growing or shrinking the buffer rather than actually shifting the bits.
- By multiplying with or dividing by an arbitrary number (which is not necessarily an integer power of two), we can achieve the equivalent of a "fractional bit shift". However, such fractional bit shifts become increasingly expensive as the buffer grows and should therefore be avoided on large buffers.

Strategy. Motivated by the above observations, we will split `self.compressed` into two parts (see Figure 1 (ii)):

- a `bulk` part that always stores an integer amount of information content; and
- a `head` part that stores the remaining fractional amount of information content.

When we encode and decode symbols, we mostly operate on the `head` (which is therefore indeed the head of the stack). Importantly the `head` has a finite capacity, so that the cost for encoding and decoding operations on the `head` is bounded by a constant. Only when the `head` would overflow or underflow do we access the `bulk` to transfer some valid bits between the `head` and the `bulk`. These transfers will also have a constant (amortized) run-time cost because we only ever transfer an *integer* amount of bits.

Your task will be to figure out *when* and *how* exactly you have to transfer data between the `head` and the `bulk`.

(a) Recall that we approximate probabilities by rational numbers, $P_{\text{ANS}}(X_i = x_i) = \frac{m(x_i)}{n}$ with $n = 2^{\texttt{precision}}$. What are the minimal and maximal information contents that a symbol with nonzero probability can have in this approximation?

We will represent the `head` as a single integer and the `bulk` as a vector (in Python: a list) of integers. A popular (albeit not the only possible) strategy to manage the `bulk` and `head` is to uphold the following two invariants:

- $\texttt{head} < 2^{2\times\text{precision}}$ (always); and
- $\texttt{head} \geq 2^{\text{precision}}$ if the `bulk` is not empty.

The jupyter notebook has a skeleton implementation of a class `StreamingAnsCoder` whose constructor initializes `self.head` and `self.bulk` in a way that trivially satisfies the above invariants. However, the `encode` and `decode` methods don't uphold these invariants yet. Instead, they currently just mirror the implementation from the lecture, except that they encode and decode to and from `self.head` instead of `self.compressed`.

(b) Consider the `encode` method and assume that both of the above invariants hold at its entry. Show that the invariants will be violated at method exit *exactly* if, at method entry,

```
(self.head >> self.precision) >= scaled_probabilities[symbol]
```

(c) Modify the `encode` method so that it transfers `precision` bits from `self.head` to `self.bulk` if the case from part (b) arises at method entry. More precisely, it should pop the `precision` lowest significant bits off `self.head` and append a single item to `self.bulk` that represents these bits. Convince yourself that this operation will never transfer any (meaningless) leading zero bits to `self.bulk`. Then show that, with your modification, the method now upholds *both* of the above invariants (i.e., both invariants hold at method exit provided that they hold at method entry).

(d) Now modify the `decode` method so that it becomes again the inverse of `encode` (i.e., if you start from any valid state and then call `encode` followed by `decode`, you always end up again in the original state). In detail, you will have to conditionally transfer some data from `bulk` to `head`, and you should figure out both *where in the code* and *under which condition* this transfer must happen. Use the provided test to debug your implementation.

(e) Now, add a simple method `get_compressed` that one can call at the end of encoding to get the full compressed representation, i.e., the concatenation of `self.bulk` and `self.head`. The method should split up `self.head` into two halves of `precision` bits each, and return a list of integers that is `self.bulk` followed the two halves of `self.head`, starting with the lower significant half. If the higher-significant half of `self.head` is zero then it should not be included in the returned value (according to the above invariants, this can only happen if `self.bulk` is empty). Finally, modify the constructor (`__init__`) so that it can accept an initial compressed representation (in the form returned by `get_compressed`). Debug your implementations again with the provided test.

(f) *(Bonus Question: Random Access)* Imagine you use your `StreamingAnsCoder` to compress some data, and you want the decoder to be able to quickly jump to certain (predefined) positions within the message without having to first decode everything that has been encoded on top of it in the stack. For example, you may want to allow the decoder to quickly jump to any symbol x_i with $i = 1000j$ for arbitrary integers j, and to then resume decoding from there. Thus, you'd package the compressed representation into some container format that also contains a *jump table*. What information should the jump table store for each desired target position so that the decoder can use this information to quickly jump there?

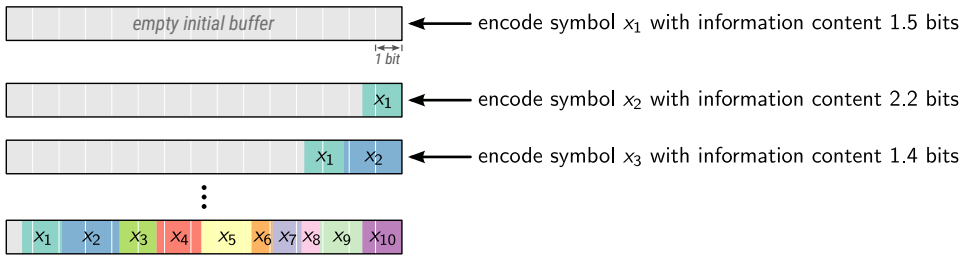

Figure 2: Stream coding with the naive ANS implementation from the lecture notes.

Problem 7.2: Streaming ANS II — Mondrian

This problem continues the discussion from Problem 7.1 but can be solved independently from the implementation of the algorithm.

Figure 2 illustrates how the naive ANS implementation from the lecture packs the encoded information content tightly into a bit string (as opposed to a symbol code, which would align each symbol to a bit boundary, thus introducing "gaps"). The illustrated compressed representation is the result of encoding the following 10 symbols, in this order:[2]

- a symbol x_1 with information content 1.5 bits;
- a symbol x_2 with information content 2.2 bits;
- a symbol x_3 with information content 1.4 bits;
- a symbol x_4 with information content 1.7 bits;
- a symbol x_5 with information content 1.9 bits;
- a symbol x_6 with information content 0.8 bits;
- a symbol x_7 with information content 1.1 bits;
- a symbol x_8 with information content 0.7 bits;
- a symbol x_9 with information content 1.6 bits; and
- a symbol x_{10} with information content 1.5 bits.

What would your `StreamingAnsCoder` from Problem 7.1 do if you used it to encode the same example sequence of symbols and then called `get_compressed()`? Draw a figure analogous to Figure 2 to sketch what the resulting compressed representation would look like (i.e., which parts of it correspond to which symbols). Assume `precision = 4`

[2] We'll gloss over the fact that, in reality, ANS wouldn't be able to represent these *precise* information contents since the corresponding probabilities $2^{-(\text{information content})}$ aren't rational numbers.

(which would be an unreasonably low precision for real applications but suffices here for demonstration purpose).

You should find that some of the symbols get "split up" into two or even three non-neighboring parts, and that the very first symbol x_1 doesn't get flushed from the `head` to the `bulk` until the very end. More precisely, you should end up with a compressed representation that is a sequence of four integers, each one carrying `precision = 4` bits of information content where

- the first integer encodes x_3, x_4, and a part of x_2;
- the second integer encodes x_6, x_7, x_8, and a part of x_5;
- the third integer encodes x_9, x_{10}, another part of x_2, and another part of x_5; and
- the fourth integer encodes x_1 and yet another part of x_2.

Don't forget to provide anonymous feedback to this problem set in the corresponding poll on moodle.