Solutions to Problem Set 6

discussed:
17 June 2022

Data Compression With And Without Deep Probabilistic Models
Prof. Robert Bamler, University of Tuebingen

Course materials available at https://robamler.github.io/teaching/compress22/

Problem 6.1: Understanding Information Content

This problem aims to give you a some additional intuition for the concept of "information content" by considering the most trivial compression problem that one could think of. It ties back to the `UniformCoder` that we used in class as a starting point for ANS coding.

The strategy that we use in this problem is something that you might find useful for approaching *any* new topic, not just in this course. Whenever there's something you feel you don't quite understand yet, I strongly encourage you to act like you're debugging some code that doesn't work: reduce the problem to the absolute simplest form, understand that simplified problem, and then gradually build up from that.

Problem Setup. Consider a data source that generates messages $\mathbf{x}$. Different from the problems so far, we don't care what these messages are—they could be sequences of symbols but they don't have to be. All we care about is that the messages come from some finite set $\mathbb{X}$ (which again, could be the product space of some finite alphabet, i.e., $\mathbb{X} = \mathfrak{X}^k$, but it doesn't have to be). Now, we consider the simplest possible probabilistic model: we assume that the messages are uniformly distributed, i.e., $P(\mathbf{X}{=}\mathbf{x}) = 1/|\mathbb{X}|\ \forall \mathbf{x} \in \mathbb{X}$.

(a) What is the information content of any message $\mathbf{x} \in \mathbb{X}$?

Solution:

$$-\log_2 P(\mathbf{X}{=}\mathbf{x}) = -\log_2 \frac{1}{|\mathbb{X}|} = \log_2 |\mathbb{X}|$$

■

(b) Take a step back from the setup for a moment and consider the binary representation of a positive integer $n \in \mathbb{N}$. How long is this binary representation, i.e., how many bits does it contain? Express your result as a mathematical function of n.

Solution: The length of the binary representation of a positive integer n is $\lceil \log_2(n+1) \rceil$, where $\lceil \cdot \rceil$ denotes rounding up to the nearest integer.

To prove this claim, it's easier to first think about the inverse problem: what are *all* the positive integers whose binary representations have some given length $\ell \in \mathbb{N}$? The smallest of these integers is represented in binary as a single "1" followed by $\ell{-}1$ zeros, i.e., its value is $n_{\text{smallest},\ell} = 2^{\ell-1}$; and the largest integer whose binary representation has length ℓ is represented in binary as a string of ℓ "1"-bits, i.e., its value is $n_{\text{largest},\ell} = 2^\ell - 1$.

Let's now calculate $f(n) := \lceil \log_2(n+1) \rceil$ for each $n \in \{n_{\text{smallest},\ell}, \ldots, n_{\text{largest},\ell}\}$.

- $f(n_{\text{largest},\ell}) = \lceil \log_2(n_{\text{largest},\ell} + 1) \rceil = \lceil \log_2(n^\ell - 1 + 1) \rceil = \lceil \ell \rceil = \ell$;
- for $n_{\text{smallest},\ell}$, we note that $\log_2(n_{\text{smallest},\ell} + 1) > \log_2(n_{\text{smallest},\ell}) = \log_2(2^{\ell-1}) = \ell - 1$ where we used that the logarithm is a strictly increasing function; therefore, when we round up to the nearest integer, we find $f(n_{\text{smallest},\ell}) = \lceil \log_2(n_{\text{smallest},\ell} + 1) \rceil > \ell - 1$, i.e., $f(n_{\text{smallest},\ell}) \geq \ell$ (since f maps to integers);
- since the function f is monotonically nondecreasing, we find for all $n \in \{n_{\text{smallest},\ell}, \ldots, n_{\text{largest},\ell}\}$ that $f(n_{\text{smallest},\ell}) \leq f(n) \leq f(n_{\text{largest},\ell})$. Combining this with our results $f(n_{\text{smallest},\ell}) \geq \ell$ and $f(n_{\text{largest},\ell}) = \ell$, we thus get $\ell \leq f(n) \leq \ell$ for all n in the range, i.e., $f(n) = \ell$.

This proves the claim. It's generally a good idea to check for mistakes by considering some examples:

n:	1	2	3	4	5	6	7	8
binary:	$(1)_2$	$(10)_2$	$(11)_2$	$(100)_2$	$(101)_2$	$(110)_2$	$(111)_2$	$(1000)_2$
length:	1	2	2	3	3	3	3	4
$\log_2(n+1)$:	1	1.58	2	2.32	2.58	2.81	3	3.17

Looks good: rounding up the values in the last row to the nearest integer results in the correct lengths. ■

(c) Back to the compression problem: consider the following encoder for messages $\mathbf{x} \in \mathbb{X}$: let f be an arbitrary bijective function from $\mathbb{X}$ to the set of integers $\{0, \ldots, |\mathbb{X}| - 1\}$. To encode a message $\mathbf{x} \in \mathbb{X}$ into a bit string, we simply write out the integer $f(\mathbf{x})$ in binary. To ensure unique decodability, we make all binary representations equally long by padding shorter ones with leading zeros to the length of the longest one (e.g., if the largest number, $|\mathbb{X}| - 1$, is $13 = (1101)_2$ and we want to encode some message $\mathbf{x}$ with $f(\mathbf{x}) = 2$, then we write out $2 = (0010)_2$). Using your result from part (b), calculate the bit rate $R(\mathbf{x})$ $\forall \mathbf{x} \in \mathbb{X}$. Then use part (a) to show that this trivial method achieves the theoretical lower bound on the expected bit rate for lossless compression with less than one bit of overhead.

Solution: Since we pad all binary representations to the same length, the bit rate $R(\mathbf{x})$ is independent of the message $\mathbf{x} \in \mathbb{X}$, and it is always the length of the binary representation of the largest integer, $|\mathbb{X}| - 1$. Thus, $R(\mathbf{x}) = \lceil \log_2(|\mathbb{X}| - 1 + 1) \rceil = \lceil \log_2 |\mathbb{X}| \rceil$ for all $\mathbf{x} \in \mathbb{X}$. Compared to the information content of $\log_2 |\mathbb{X}|$, we thus have an overhead of less than one bit (since the rounding-up operation $\lceil \cdot \rceil$ increases a number always by less than one). ■

(d) Now argue—without referring to any fancy information theoretical theorems—why the lossless compression method from part (c) is obviously optimal (up to maybe one bit of overhead).

Solution: I didn't expect a very formal answer here; this question was mainly intended to encourage you to think about what lossy compression means. The goal of lossy compression is to find an injective mapping from the message space $\mathbb{X}$ to the space of bit strings that makes the resulting bit strings as short as possible. We typically have a probabilistic model over the message space, and if some message $\mathbf{x} \in \mathbb{X}$ is more probable than others according to this model, then we try extra hard to map this message $\mathbf{x}$ to a very short bit string even if this means that several other messages will have to be mapped to a longer bit string so as to avoid collisions.

But in the toy example that we're considering here, all messages $\mathbf{x} \in \mathbb{X}$ have the same probability. Therefore, there's no reason to treat any one message differently from the others, and we really just have to map the entire message space $\mathbb{X}$ injectively to the $|\mathbb{X}|$ shortest bit strings, in arbitrary order. Up to minor details related to unique decodability, the $|\mathbb{X}|$ shortest bit strings are precisely the binary representations of the numbers $\{0, 1, \ldots, |\mathbb{X}| - 1\}$. ■

Problem 6.2: Streaming ANS in the Style of Piet Mondrian

In the lecture, we first implemented a `SlowAnsCoder` that has near-optimal compression performance but whose run-time cost $O(k^2)$ grows quadratic in the message length k. Figure 1 illustrates how our naive `SlowAnsCoder` represents compressed data if we were to encode the following sequence of symbols:[1]

- 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.

[1] 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, so they can't be precisely represented with the fixed-point model Q.

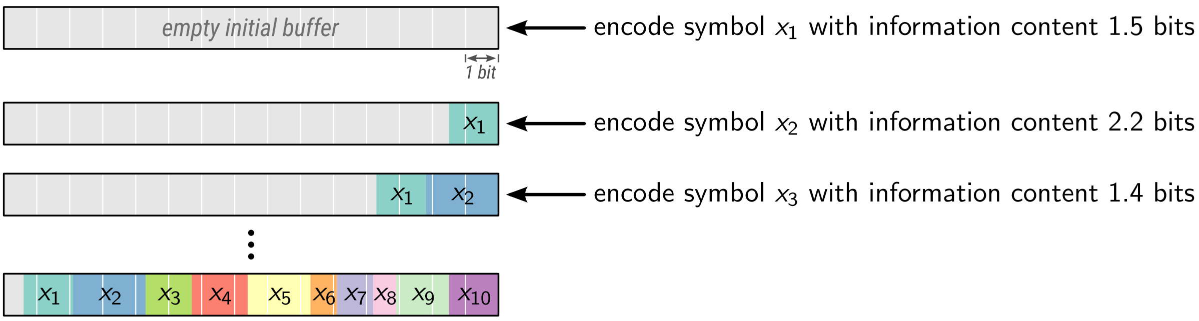

empty initial buffer
1 bit
encode symbol x_1 with information content 1.5 bits
x_1
encode symbol x_2 with information content 2.2 bits
x_1 x_2
encode symbol x_3 with information content 1.4 bits
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}

Figure 1: Stream coding with the naive `SlowAnsCoder` from the lecture notes.

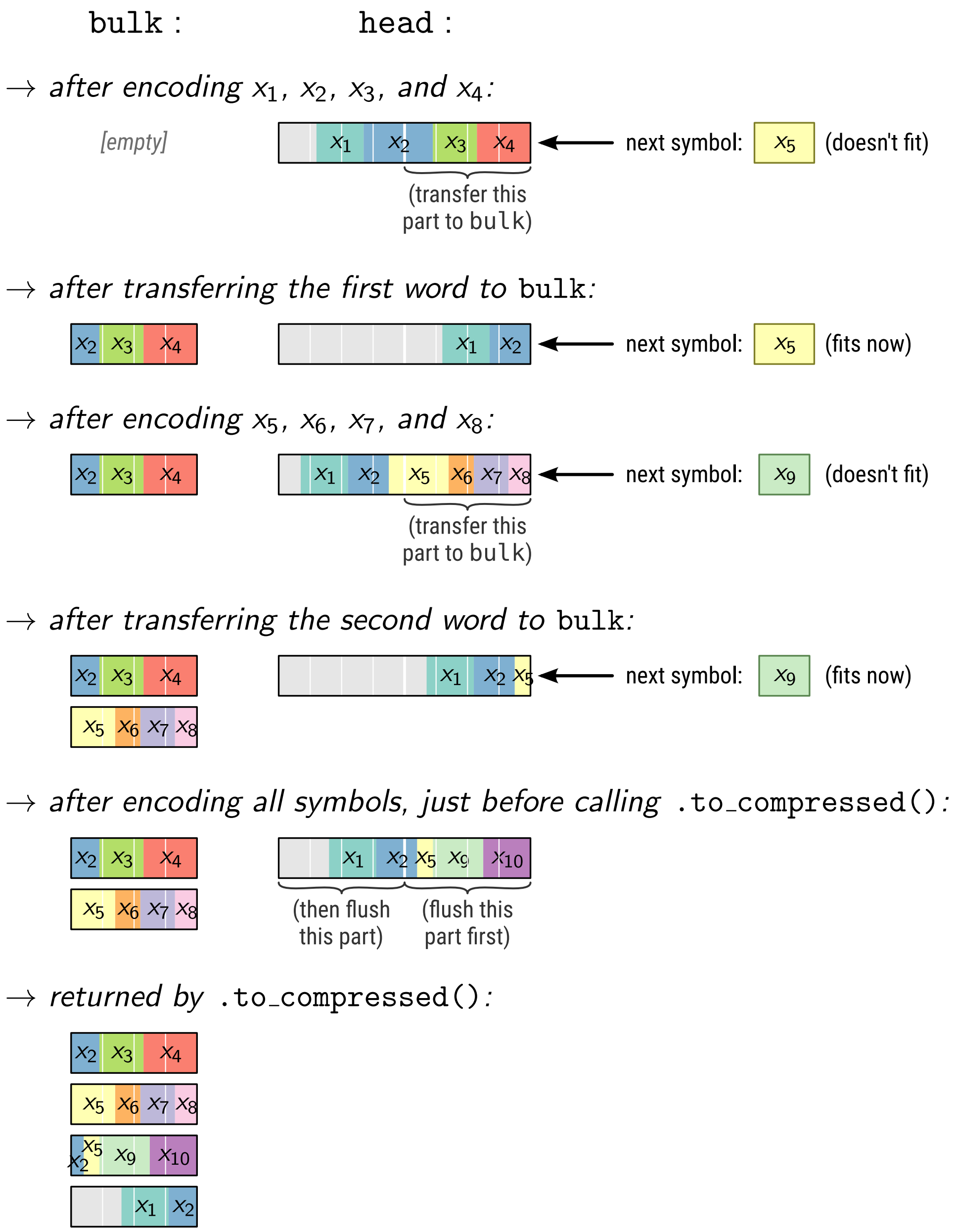
We then reduced the run-time cost to $O(k)$ by splitting the representation of the compressed data into a fixed-sized `head` and a variable-sized `bulk`, resulting in the "streaming ANS" algorithm. We used a setup where `head` can hold up to $2\times$`precision` bits. In streaming ANS, encoding normally only pushes compressed data onto `head` and leaves `bulk` untouched; only if encoding onto `head` would lead to an overflow (i.e., if it would lead to `head` $\geq 2^{2\times\texttt{precision}}$) then one first transfers the `precision` least significant bits from `head` to the end of `bulk` before encoding onto `head`.

Let's visualize how streaming ANS would encode the above sequence of ten symbols. Draw a figure analogous to Figure 1 to sketch what the resulting compressed representation would look like after each step of the process (i.e., which parts of the `bulk` and `head` correspond to which symbol). Assume `precision` $= 4$ (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 logically "split up" into two or even three not necessarily neighboring parts. Further, the very first symbol x_1 doesn't get flushed from `head` to `bulk` until the very end. More precisely, after encoding all ten symbols and calling `get_compressed()` you get a compressed representation that is a sequence of four integers, each one being `precision` $= 4$ bits long, 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.

Solution: The following figure shows one possible way to illustrate the encoding process of streaming ANS. The width of each colored rectangle is proportional to the information content represented by the rectangle.


bulk :
head :
→ after encoding x_1, x_2, x_3, and x_4:
[empty]
x_1 x_2 x_3 x_4
next symbol: x_5 (doesn't fit)
(transfer this part to bulk)
→ after transferring the first word to bulk:
x_2 x_3 x_4
x_1 x_2
next symbol: x_5 (fits now)
→ after encoding x_5, x_6, x_7, and x_8:
x_2 x_3 x_4
x_1 x_2 x_5 x_6 x_7 x_8
next symbol: x_9 (doesn't fit)
(transfer this part to bulk)
→ after transferring the second word to bulk:
x_2 x_3 x_4
x_5 x_6 x_7 x_8
x_1 x_2 x_5
next symbol: x_9 (fits now)
→ after encoding all symbols, just before calling .to_compressed():
x_2 x_3 x_4
x_5 x_6 x_7 x_8
x_1 x_2 x_5 x_9 x_{10}
(then flush this part)
(flush this part first)
→ returned by .to_compressed():
x_2 x_3 x_4
x_5 x_6 x_7 x_8
x_2 x_5 x_9 x_{10}
x_1 x_2

■

Problem 6.3: Range Coding With an Autoregressive Model for English Text

In Problem 3.2 on Problem Set 3, you trained an autoregressive machine learning model (a recurrent neural network) to model the probability distribution of English text. You then used this model as an entropy model for compressing text. Back then, you used a Huffman coder since we hadn't introduced stream codes yet.

In this problem, you'll replace the Huffman coder with a range coder, and you'll evaluate empirically how this affects compression performance (i.e., the bit rate).

(a) Before you start: why is it better to use a range coder here and not an ANS coder?

Solution: Range coding operates as a queue ("first in first out") whereas ANS operates as a stack ("last in first out"). For autoregressive models, it's much easier to encode with queue semantics because both encoder and decoder iterate through symbols in the message in the same order. We'll see a compression scenario that can more easily be addressed with stack semantics on the next problem set. ■

(b) I won't make you implement the core range coding algorithm because its implementation is a bit involved due to some edge cases and I don't think you'll learn much from it. Instead, we'll use a pre-built range coder provided by the `constriction` library, which was specially developed with research and teaching use cases in mind.[2] Install `constriction` by executing (preferably in a virtual environment):

```
python3 -m pip install constriction~=0.2.4
```

Then try out the first code example from the API documentation of `constriction`'s range coder.[3] The example should execute without errors and print some example message (i.e., a sequence of symbols), encode it, print the compressed representation, and then decode it and print the reconstructed message.

Read the code example and make sure you understand what it does. You can ignore anything related to `message_part2`, which shows how to use a model class called `QuantizedGaussian`—we won't need this type of model, only the `Categorical` model that's used for encoding `message_part1` in this example.

(c) Now use your newly acquired range coding skills to replace the Huffman coder in our compression method from Problem 3.2. Don't worry if you haven't completed Problem 3.2, you can always download the proposed solutions[4] from the course website. The PDF document that's part of the solutions also contains instructions for how to set up your python environment and train the model (you'll probably have to reinstall `constriction` in the new python environment using the same command as in part (b)).

[2] If you run into problems with the `constriction` library, please let me know or report an issue at https://github.com/bamler-lab/constriction/issues

[3] https://bamler-lab.github.io/constriction/apidoc/python/stream/queue.html

[4] https://robamler.github.io/teaching/compress22/problem-set-03-solutions.zip

Evaluate the compression performance by comparing the bit rate to (i) the results that you get with Huffman coding and (ii) the information content under the model (which is printed to the terminal in the proposed solutions for Problem 3.2).

Solution: See accompanying `git` bundle. Encoding and decoding is implemented in the file `compression.py`; the other files were not changed. For instructions on how to clone or pull from the `git` bundle, and how to encode and decode data, see the solutions to Problem 3.2 (see download link in footnote 4 on page 6).

With this implementation, I obtain a bit rate of 2.11170 bits per character on the test set, which corresponds to an overhead of 0.02 % over the the information content (2.11129 bits per character). Recall that, with Huffman Coding (Problem Set 3), the bit rate was 2.38 bits per character, corresponding to an overhead of 12 % over the information content.

Thus, the compression performance of the range coder is extremely close to optimal. If one wanted to reduce the bitrate further, it really wouldn't make any sense at this point to improve the entropy coder. Instead, one should improve the probabilistic model. ■