Problem Set 6

published: 24 May 2023
discussion: 7 June 2023

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

Course materials available at https://robamler.github.io/teaching/compress23/

Note: This problem set focuses entirely on the Asymmetric Numeral Systems (ANS) algorithm as there seemed to be quite a bit of confusion about this algorithm in the lecture. I decided to defer the promised exercise that uses range coding with our autoregressive machine-learning model of English text to a later problem set.

Problem 6.1: Positional Numeral Systems

In the lecture, we introduced the Asymmetric Numeral Systems entropy coder as a generalization of positional numeral systems. Listing 1 on the next page shows our implementation of a class `UniformCoder`, which is an optimal entropy coder for a sequence of independent and uniformly distributed symbols, using a positional numeral system with a position-dependent `base`. The listing also contains a simple usage example.

(a) **Correctness:** prove that `UniformCoder` is indeed a correct entropy coder, i.e., that decoding (i.e., `pop`) is the inverse of encoding (i.e., `push`):

  (i) convince yourself that calling `coder.push(symbol, base)` and then calling `coder.pop(base)` returns `symbol` and restores the original state of `coder`, regardless of its original state (assuming that all method arguments are valid, i.e., `base` and `symbol` are integers with $0 \leq$ `symbol` $<$ `base`);

  (ii) convince yourself that setting `symbol = coder.pop(base)` with any positive integer `base` and then calling `coder.push(symbol, base)` restores the original state of `coder`, regardless of its original state (even if the coder was originally empty).

(b) **Compression performance:** assume you use the `UniformCoder` to encode a sequence of `symbols` $x_1, x_2, \ldots, x_k$ with respective `bases` (i.e., alphabet sizes) $|\mathfrak{X}_1|, |\mathfrak{X}_2|, \ldots, |\mathfrak{X}_k|$. Assuming the alphabet sizes are fixed, which sequence of symbols leads to the largest possible value of `coder.compressed` after encoding? What is this largest possible value and how long is its binary representation? Compare to the information content of the message assuming the symbols are statistically independent, and each one is uniformly distributed over the respective alphabet $\mathfrak{X}_i = \{0, 1, \ldots, |\mathfrak{X}_i| - 1\}$ for each symbol x_i.

```python
class UniformCoder:
    def __init__(self, compressed=0):
        self.compressed = compressed

    def push(self, symbol, base):  # Encodes a symbol ∈ {0, ..., base − 1}.
        self.compressed = self.compressed * base + symbol

    def pop(self, base):           # Decodes a symbol ∈ {0, ..., base − 1}.
        symbol = self.compressed % base
        self.compressed //= base   # ("//" denotes integer division.)
        return symbol

# Usage example:
coder = UniformCoder()

coder.push(6, base=10)
coder.push(13, base=16)
coder.push(7, base=8)
print(bin(coder.compressed))    # Prints: "0b1101101111"

print(coder.pop(base=8))  # Prints: "7"
print(coder.pop(base=16)) # Prints: "13"
print(coder.pop(base=10)) # Prints: "6"
```

Listing 1: A "stack"-like entropy coder that is optimal for uniformly distributed symbols.

Problem 6.2: Naive Asymmetric Numeral Systems

Listing 2 shows the `SlowAnsCoder` from the lecture, which is a naive version of the Asymmetric Numeral Systems (ANS) algorithm (we will improve it in Problem 6.3).

(a) **Correctness:** similar to Problem 6.1 (a), prove that `SlowAnsCoder` is a correct entropy coder, i.e., that decoding (i.e., `pop`) is the inverse of encoding (i.e., `push`):

(i) convince yourself that calling `coder.push(symbol, m)` followed by calling `coder.pop(m)` returns `symbol` and restores the original state of `coder`, regardless of its original state (assuming that all method arguments are valid, i.e., `m` is a list of nonnegative integers that sum to $2^{\texttt{precision}}$ and $0 \leq \texttt{symbol} < \texttt{len(m)}$; also, $\texttt{m[symbol]} \neq 0$; why is the last condition necessary?);

(ii) convince yourself that setting `symbol = coder.pop(m)` (where `m` is again a list of nonnegative integers that sum to $2^{\texttt{precision}}$), followed by calling `coder.push(symbol, m)` restores the original state of `coder`, regardless of its original state (even if the coder was originally empty).

You may build your arguments on the fact that `coder.uniform_coder` is a correct entropy coder, which you showed in Problem 6.1 (a).

(b) **Compression performance:** in Problem 6.1 (b), you showed that encoding (i.e., `push`ing) a symbol with a `UniformCoder` and some given `base` contributes $\log_2(\texttt{base})$ bits to the amortized bit rate. Since `pop` inverts `push`, it therefore reduces the amortized bit rate by the same amount. Using these results, by how much does calling the method `push` on a `SlowAnsCoder` increase the amortized bit rate? Compare to the information content of the symbol under the approximate probability distribution $Q(X_i{=}x_i) = \frac{m_i(x_i)}{n}$ where $m_i(x_i)$ is `m[symbol]` in python.

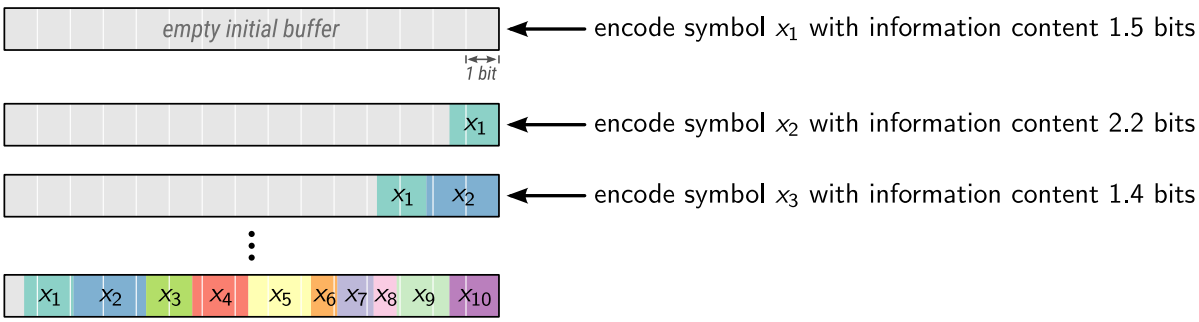

Figure 1: Stream coding with the naive `SlowAnsCoder` from Listing 2.

```python
class SlowAnsCoder:
    def __init__(self, precision, compressed=0):
        self.n = 2**precision    # ("**" denotes exponentiation.)
        self.uniform_coder = UniformCoder(compressed)  # See Listing 1.

    def push(self, symbol, m):   # Encodes one symbol.
        z = self.uniform_coder.pop(base=m[symbol]) + sum(m[0:symbol])
        self.uniform_coder.push(z, base=self.n)

    def pop(self, m):            # Decodes one symbol.
        z = self.uniform_coder.pop(base=self.n)
        # Find the unique symbol that satisfies z ∈ 𝔍_i(symbol)
        # (using linear search just to simplify exposition):
        for symbol, m_symbol in enumerate(m):
            if z >= m_symbol:
                z -= m_symbol
            else:
                break
        self.uniform_coder.push(z, base=m_symbol)
        return symbol

    def get_compressed(self):
        return self.uniform_coder.compressed

# Usage example:
precision = 4 # (for demonstration purpose only)
m = [7, 6, 3] # (adds up to 16 = 2^precision, as required)
coder = SlowAnsCoder(precision)

coder.push(0, m)
coder.push(2, m)
coder.push(1, m)
print(bin(coder.get_compressed())) # Prints: "0b101000"

print(coder.pop(m)) # Prints: 1
print(coder.pop(m)) # Prints: 2
print(coder.pop(m)) # Prints: 0
```

Listing 2: Our naive (slow) implementation of Asymmetric Numeral Systems (ANS) from the lecture.

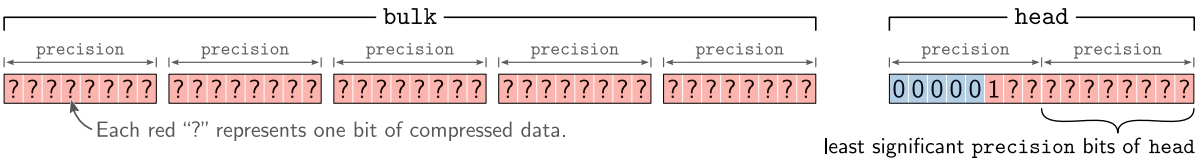

Figure 2: Representation of compressed data in streaming ANS with `precision` = 8.

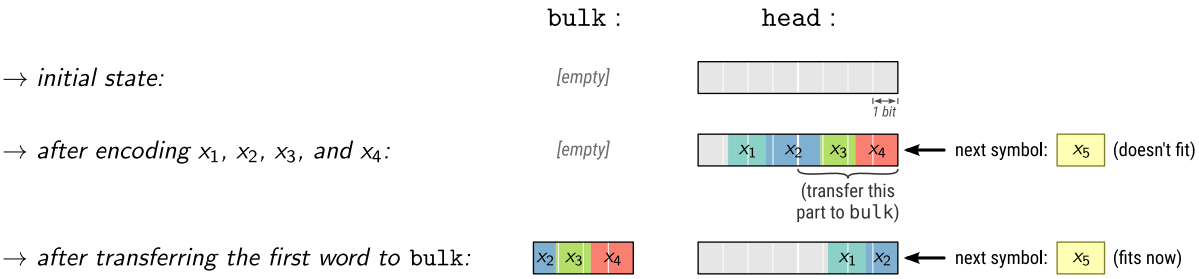

Figure 3: Stream coding with a `StreamingAnsCoder` (Listing 3) with `precision` = 4.

Problem 6.3: Streaming Asymmetric Numeral Systems

The `SlowAnsCoder` that we implemented in the lecture and discussed in Problem 6.2 is a correct entropy coder with a very close to optimal bit rate. But it has a problem: it is slow. More precisely, the runtime for encoding a single symbol grows linearly with the information content that has already been encoded onto the `SlowAnsCoder` before. Therefore, the runtime for encoding a sequence of k symbols scales quadratically in k, which makes this naive algorithm impractical for most common use cases. Let's fix this.

(a) Let's first understand exactly why the `SlowAnsCoder` from Listing 2 is slow. Figure 1 on page 3 illustrates how compressed data accumulates as we encode a sequence of 10 symbols x_1, x_2, …, x_{10} with a `SlowAnsCoder`. Different to, e.g., arithmetic coding or range coding, each new symbol that we encode conceptually pushes any previously encoded data to the left (i.e., towards more significant bits of `coder.uniform_coder.compressed`). This operation becomes increasingly expensive as the amount of previously encoded data grows. Identify the lines of code in Listing 2 (and, more precisely, inside method calls that jump to Listing 1) that implement this increasingly expensive operation.

In order to speed up ANS and to make its runtime complexity linear in the length of the message, one uses the following trick (called "streaming ANS"): instead of representing the entire compressed data as a single (giant) integer, one splits the compressed data into a `bulk` part that holds most of the data in a dynamically sized array (aka "vector") and a `head` part with a fixed (and small) capacity. Figure 2 illustrates the simplest variant of this approach where each item of the vector `bulk` holds `precision` bits (referred to as "one word" of compressed data in the following), and `head` has a capacity of 2 words, i.e., $2 \times$ `precision` bits (thus, `head` can represent only integers from zero to $2^{2\times\texttt{precision}} - 1$).

Figure 3 illustrates what happens when we encode a sequence of symbols using such a streaming ANS coder. We use `precision` $= 4$ here for demonstration purpose.[1] While the first 4 symbols fit into `head` in the example illustrated in Figure 3, encoding the fifth symbol x_5 would lead to an overflow, i.e., it would lead to `head` $\geq 2^{2\times\texttt{precision}}$, which is not allowed. Thus, before we encode x_5, we transfer the least significant `precision` bits of `head` to `bulk`, and we move the remaining part of `head` by `precision` bits to the right. Assuming a good vector implementation, transferring *an integer number of bits* from `head` to `bulk` requires only constant (amortized) runtime because even if there was already some data on `bulk`, we wouldn't need to move it around.

(b) Continue Figure 3 until you have encoded 10 symbols $x_1, \ldots, x_{10}$. Assume that the symbols have the following information contents:

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

You may ignore the fact that the corresponding probabilities, $2^{-\text{information content}}$, cannot be precisely represented in fixed point precision.

You should find that some of the symbols get logically "split up" into two or even three not necessarily consecutive parts. Further, the first symbol x_1 stays completely on `head` until the very end in this example.

In case you're curious, Listing 3 implements streaming ANS in Python. The main differences to our `SlowAnsCoder` class from Listing 2 are that (i) Listing 3 adds a potential transfer of one word from `head` to `bulk` at the beginning of the method `pop` as discussed above and, correspondingly, a potential transfer of one word from `bulk` back to `head` at the end of the method `push`; (ii) all remaining encoding and decoding operations are performed on `head`; and (iii) these encoding and decoding operations are no longer delegated to the methods `push` and `pull` of an internal `UniformCoder` because we've manually inlined these method calls so that Listing 3 is self-contained.

The inlining also allows us to perform some optimizations by replacing multiplications and divisions involving $n = 2^{\texttt{precision}}$ by equivalent and faster bit shifts. This eliminates divisions during decoding, which are by far the slowest arithmetic operations on CPUs.[2]

[1] In production, one would set `precision` to about 16 or 32, so that `head` just fits into a CPU register.
[2] see, e.g., measurements by Fog: https://www.agner.org/optimize/instruction_tables.pdf

```python
class StreamingAnsCoder:
    def __init__(self, precision, compressed=[]):
        self.precision = precision
        self.mask = (1 << precision) - 1 # a string of precision 1-bits
        self.bulk = compressed.copy() # (We will mutate bulk below.)
        self.head = 0
        # Ensure that head is at least half filled unless bulk is empty.
        while len(self.bulk) != 0 and (self.head >> precision) == 0:
            self.head = (self.head << precision) | self.bulk.pop()

    def push(self, symbol, m):       # Encodes one symbol.
        # Check if encoding onto head would lead to an overflow:
        if (self.head >> self.precision) >= m[symbol]:
            # Transfer one word of compressed data from head to bulk:
            self.bulk.append(self.head & self.mask) # (bitwise and)
            self.head >>= self.precision
            # This is the only point where head is (temporarily) less
            # than half filled despite bulk not being empty.

        # Below as in SlowAnsCoder (Listing 2), just with inlined calls:
        z = self.head % m[symbol]
        self.head //= m[symbol]
        self.head = (self.head << self.precision) | z # (This is
            # equivalent to "self.head * n + z", just slightly faster.)

    def pop(self, m):                # Decodes one symbol.
        # Begin as in SlowAnsCoder (see Listing 2):
        z = self.head & self.mask # (same as "self.head % n" but faster)
        self.head >>= self.precision # (same as "//= n" but faster)
        for symbol, m_symbol in enumerate(m):
            if z >= m_symbol:
                z -= m_symbol
            else:
                break
        self.head = self.head * m_symbol + z

        # Detect whether push transferred data from head to bulk here:
        if (self.head >> self.precision) == 0 and len(self.bulk) != 0:
            # Transfer data back from bulk to head ("|" is bitwise or):
            self.head = (self.head << self.precision) | self.bulk.pop()

        return symbol

    def get_compressed(self):
        compressed = self.bulk.copy() # (We will mutate compressed below.)
        head = self.head
        # Chop head into precision-sized words and append to compressed:
        while head != 0:
            compressed.append(head & self.mask)
            head >>= self.precision
        return compressed
```

Listing 3: A complete streaming ANS entropy coder in Python. For a usage example, see lower part of Listing 2 (replace `SlowAnsCoder` with `StreamingAnsCoder`).