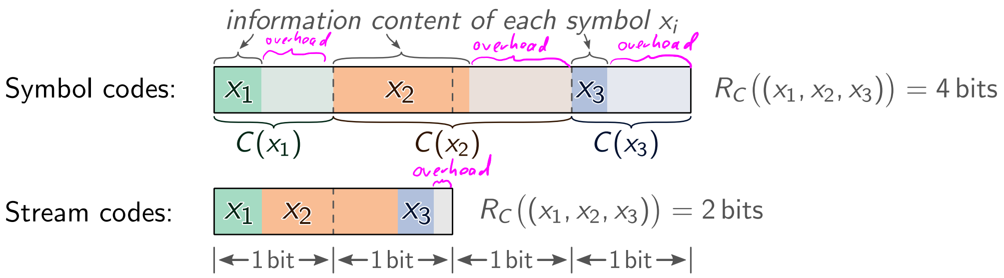
Lecture 6, Part 1:

Asymmetric Numeral Systems (ANS)

Robert Bamler • Summer Term of 2023

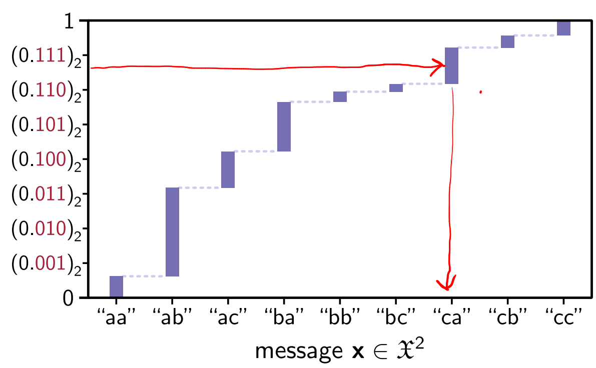
These slides are part of the course *"Data Compression With and Without Deep Probabilistic Models"* taught at University of Tübingen. More course materials—including video recordings, lecture notes, and problem sets with solutions—are publicly available at `https://robamler.github.io/teaching/compress23/`.

Recall From Last Week: Stream Codes



- Reduces overhead from ≤ 1 bit *per symbol* to ≤ 1 bit *per message* (in theory)
- In practice: larger overhead due to finite precision & technical limitations (more on this today), but much smaller than for symbol codes

Recall: Arithmetic Coding & Range Coding

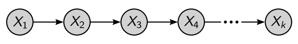


- Simple algorithm in principle: iteratively refine interval $\big[P(\mathbf{X} < \mathbf{x}), P(\mathbf{X} \leq \mathbf{x})\big)$
- Tricky to implement in practice (finite precision arithmetic, edge cases)
- Operates as a *queue*: "first in first out"

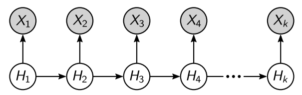
Recall: Probabilistic Model Architectures

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

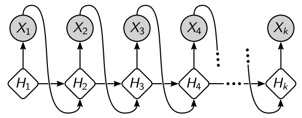
(1) Markov Process



(2) Hidden Markov Model

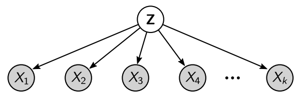


(3) Autoregressive Model



(4) Latent Variable Model

next week, using entropy coder that we discuss now

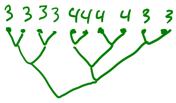


- Problem 7.3: use range coding for our autoregressive model of natural language

Positional Numeral Systems

optimal symbol code (Huffman code)

$|C(x)| =$ 3 3 3 3 4 4 4 4 3 3 $\Rightarrow$ average = 3.4



EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Consider a data source that generates a random message $\mathbf{X} \equiv (X_1, X_2, \ldots, X_k)$ of length k; symbols are i.i.d., with each X_i *uniformly distributed* over $\mathfrak{X} = \{0, 1, 2, \ldots, 9\}$.

(a) What is the entropy per symbol? $\frac{1}{k} H_P[\mathbf{X}] = H_P[X_i] = \mathbb{E}_P[-\log_2 P(X_i = x_i)] = \mathbb{E}_P[-\log_2 \frac{1}{10}] = \log_2(10) \approx 3.32$

(b) What is the expected code word length of an *optimal symbol code* for this data source?
$L_{C_\text{opt. symbol code}} := \mathbb{E}_P\big[|C_\text{opt. symbol code}(X_i)|\big] = 3.4$ (see Huffman tree above)

interpret x_i as digits of a decimal number, then convert to binary

(c) Can you do better than an optimal symbol code? Describe your approach first in words, then implement it in Python or in pseudo code

next slide

- Don't think about arithmetic/range coding; it's much simpler.
- About 4 lines of code for encoding + 4 lines of code for decoding; no library function calls necessary.

encode($\underline{x}$): allows us to detect end
initialize $n = 1$
for i in k down to 1:
  update $n \leftarrow n * 10 + x_i$
return binary_rep(n)

decode(n):
while $n \neq 1$:
  emit symbol $n \bmod 10$
  update $n \leftarrow \lfloor n/10 \rfloor$

Problem 2.4 (b)

(d) What is the expected bit rate per symbol of your method from part (c) in the limit of long messages? $\lim_{k\to\infty} \frac{1}{k} \mathbb{E}_P[R_{C_\text{part (c)}}(\mathbf{X})] \leq \lim_{k\to\infty} \frac{1}{k} R_{C_\text{part (c)}}((\underbrace{9, 9, 9, \ldots, 9}_{k \text{ times}})) = \lim_{k\to\infty} \text{length}(\text{binary\_rep}(1\underbrace{999\ldots9}_{k\text{ times}})) =$
$= \lim_{k\to\infty} \frac{1}{k} \log_2 \lceil 2 \times 10^k - 1 + 1 \rceil \leq \lim_{k\to\infty} \frac{1}{k}(k \log_2 10 + 2) = \log_2 10$

(similar calculation for lower bound: $\lim_{k\to\infty} \frac{1}{k} \mathbb{E}_P[R_C(\mathbf{X})] \geq \log_2 10$)

Positional Numeral Systems: Implementation

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- **Implementation in Python:**

```
def encode_uniform(message, base=10):
    number = 1
    for symbol in reversed(message):
        number = number * base + symbol
    return number

def decode_uniform(number, base=10):
    while number != 1:
        yield number % base
        number //= base
```

we encode symbols in reverse order here so that the decoder below reconstructs them in normal order

→ positional numeral systems behave like a stack ("last in first out")

- **Usage example:**

```
compressed = encode_uniform([3, 5, 6])
print(bin(compressed))                   # Prints: "0b11001110101"
print(list(decode_uniform(compressed))) # Prints: "[3, 5, 6]"
```

Observations About Positional Numeral Systems

- encoding & decoding operates as a *stack* ("last in first out")
- encoding *amortizes* bit rate over several symbols ($\neq$ symbol codes)

we changed the 3rd symbol

we changed the 2nd symbol

```
print(bin(encode_uniform([3, 5, 7]))) # Prints: "0b11011011001"
print(bin(encode_uniform([3, 5, 6]))) # Prints: "0b11001110101"
print(bin(encode_uniform([3, 4, 6]))) # Prints: "0b11001101011"
```

those bits seem to depend on both the second and the third symbol
$\Rightarrow$ we can no longer assign each bit to a single symbol, as we were able to do for symbol codes

- positional numeral systems are an *optimal lossless compression methods* if:
  - (i) all symbols are from the same (finite) alphabet $\mathfrak{X}$
  - (ii) all symbols are uniformly distributed over $\mathfrak{X}$
  - (iii) all symbols are statistically independent
- **Goal today:** remove constraints (i) and (ii)
- **Goal next lecture:** remove constraint (iii) $\rightarrow$ "bits-back trick"

Limitation (i): symbols from different alphabets

Not a real limitation: just make base position-dependent:

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

    def push(self, symbol, base):
        """Encodes a symbol from alphabet {0, ..., base-1}."""
        self.compressed = self.compressed * base + symbol

    def pop(self, base):
        """Decodes a symbol from alphabet {0, ..., base-1}."""
        symbol = self.compressed % base
        self.compressed //= base
        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"
```

reconstructs original message (in reverse order)

Limitation (ii): Non-uniformly Distributed Symbols

- Consider a single symbol x_i
- **Step 1:** approximate $P(X_i)$ in fixed point arithmetic:

$$Q(X_i) = \frac{m_i(x_i)}{n}$$ where $n = 2^{\text{precision}}$ is a power of 2 (will become useful later) and the integers $m_i(x_i)$ satisfy $\sum_{x_i \in \mathfrak{X}_i} m_i(x_i) = n$ and are chosen such that $Q(X_i) \approx P(X_i)$

  - compression overhead: $D_{KL}\big(P(X_i) \,\|\, Q(X_i)\big)$ (asymptotically decays exponentially in precision)

- **Step 2:** interpret $Q(X_i)$ as the marginal distribution of a *latent variable model:*

$$Q(X_i) = \sum_{z_i=0}^{n-1} Q(z_i)\, Q(X_i \mid z_i)$$

$(\mathcal{Z}_i(x_i))_{x_i \in \mathfrak{X}_i}$ are pairwise disjoint subsets of $\{0, \ldots, n-1\}$ of size $|\mathcal{Z}_i(x_i)| = m_i(x_i)$

uniform distribution over $\{0, 1, \ldots, n-1\}$, i.e., $Q(Z_i = z_i) = \frac{1}{n}\ \forall z_i \in \{0, \ldots, n-1\}$

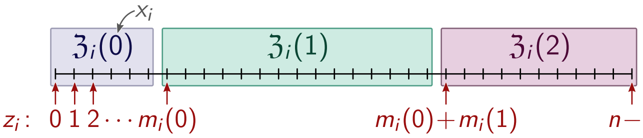
$$Q(X_i = x_i \mid Z_i = z_i) = \begin{cases} 1 & \text{if } z_i \in \mathcal{Z}_i(x_i) \\ 0 & \text{otherwise} \end{cases}$$

$$\Rightarrow Q(X_i = x_i) = \sum_{z_i=0}^{n-1} Q(Z_i = z_i)\, Q(X_i = x_i \mid Z_i = z_i) = \sum_{z_i \in \mathcal{Z}_i(x_i)} \frac{1}{n} = \frac{|\mathcal{Z}_i(x_i)|}{n} = \frac{m_i(x_i)}{n} \checkmark$$

Limitation (ii): Non-uniformly Distributed Symbols

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

- Consider a single symbol x_i
- **Step 1:** approximate $P(X_i)$ in fixed point arithmetic:
- **Step 2:** interpret $Q(X_i)$ as the marginal distribution of a *latent variable model.*
- **Step 3:** Bits-back trick:
  - $\{m_i(x_i)\}$ partition the range $\{1, 2, \dots, n-1\}$ into $|\mathfrak{X}_i|$ non-overlapping subranges $\mathfrak{Z}_i(x_i)$:



  - **Naive idea:** encode arbitrary $z_i \in \mathfrak{Z}_i(x_i)$ → bit rate: $-\log_2 Q(Z_i = z_i) = \log_2 n$ = precision bits per symbol ☹
  - **Better idea:** piggy-back some information into choice of z_i by *decoding* z_i from previously encoded data, using a uniform model over $\mathfrak{Z}_i(x_i)$ → consumes $\log_2 |\mathfrak{Z}_i(x_i)| = \log_2 m_i(x_i)$ bits
    $\longrightarrow$ "bits-back trick" (more general discussion next lecture)
    ⇒ net bit rate: $\log_2 n - \log_2 m_i(x_i) = -\log_2 Q(X_i = x_i)$ ☺

(Slow) Implementation of ANS in Python

→ Prove correctness and analyze bitrate on problem set

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

```
class SlowAnsCoder:
    def __init__(self, precision, compressed=0):
        self.n = 2**precision     # ("**" denotes exponentiation.)
        self.uniform_coder = UniformCoder(compressed)  # See slide 7.

    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 ∈ 𝔷_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
```

decode z from any data that's already accumulated on self.uniform_coder using uniform distrib. over $\mathfrak{Z}_i(x_i)$ → consumes $\log_2 m_i(x_i)$ bits

reconstruct the data that we consumed on the first line of push.

Usage example:

```
# We use a very low precision
# here for demonstration purpose;
# real deployments should use
# higher precision.
precision = 4
m = [7, 6, 3]  # (Sums to 16 = 2^4.)
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
```

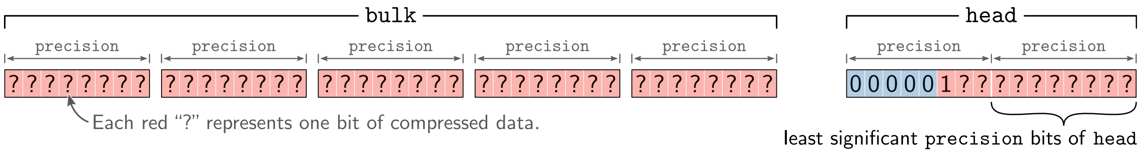
this means: $Q(X_i=0) = \frac{7}{2^4} = \frac{7}{16}$, $Q(X_i=1) = \frac{6}{16}$, $Q(X_i=2) = \frac{3}{16}$

We could also use a different model for each symbol as long as we use matching models for decoding below.

Streaming ANS

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

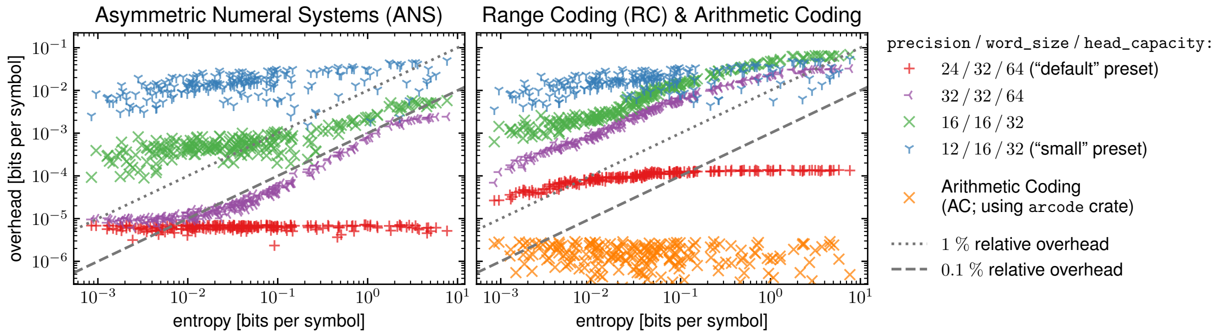
- `SlowAnsCoder` is slow (run-time $O(k^2)$)
- **Idea ("streaming ANS"):** operate mostly on a compressed representation with *finite capacity*. If it would overflow, push an *integer number of bits* to a growable buffer.



→ You'll walk through an illustrated example of this in Problem 6.3.

encode & decode mostly on fixed-capacity head, but transfer least significant half of it to/from bulk whenever it over/underflows.

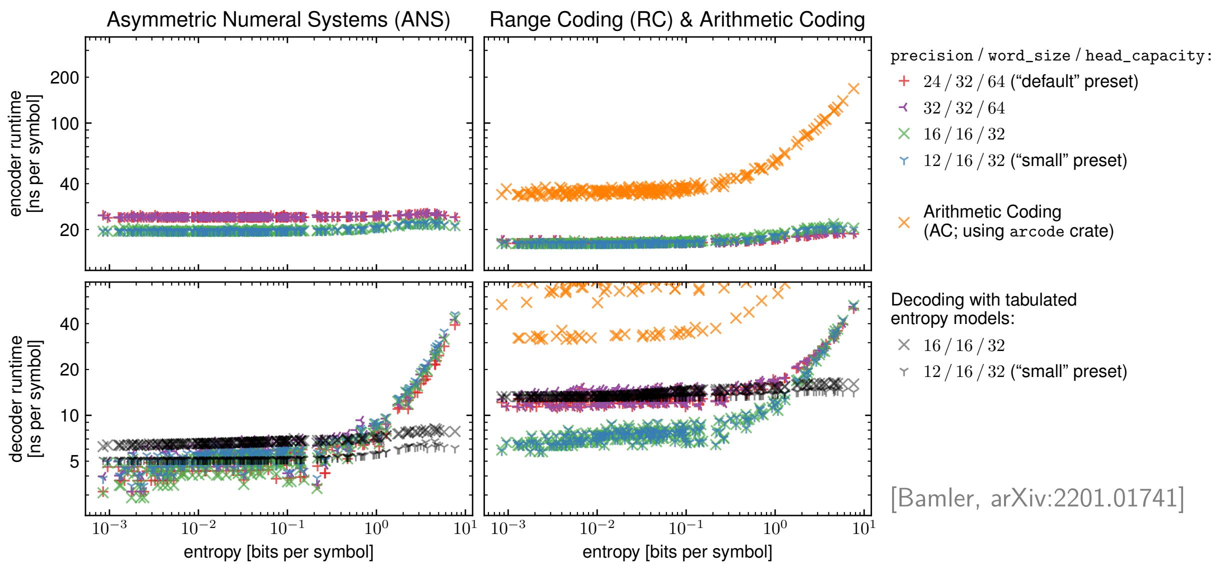
Empirical Compression Performances: bit rates



[Bamler, arXiv:2201.01741 (2022)]

uses ANS & range coding implementation from "constriction" library, which you'll also use on later problem sets (starting with Problem 7.3).

Empirical Compression Performances: run times



[Bamler, arXiv:2201.01741]

Outlook

- **Problem Set:**
  - Prove correctness and analyze compression performance of our `SlowAnsCoder` implementation.
  - Illustrate an example of streaming ANS.
- **Next Lecture:** revisit & generalize the bits-back trick
- **Afterwards:** use ANS for (net) optimal lossless compression with latent variable models (e.g., variational autoencoders)