Problem Set 7

published: 7 June 2023
discussion: 14 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/

Problem 7.1: Bits-Back Coding

In the lecture, we discussed how to compress a message with a latent variable model

$$P(\mathbf{X}) = \sum_z P(Z{=}z, \mathbf{X}) \qquad \text{where} \qquad P(Z, \mathbf{X}) = P(Z)\, P(\mathbf{X} \,|\, Z). \tag{1}$$

Here, $\mathbf{X}$ is the message and Z is a latent variable that is not part of the message. We discussed that the marginal message distribution $P(\mathbf{X})$ is usually too complicated to be used directly in an entropy coder like range coding or ANS. However, if the likelihood $P(\mathbf{X} \,|\, Z) = \prod_{i=1}^{k} P(X_i \,|\, Z)$ factorizes over the symbols X_i, then we can encode a given message $\mathbf{x}$ at an optimal (net) bit rate using the so-called bits-back trick.

(a) **Encoding:** the following table summarizes the three steps to encode a message $\mathbf{x}$ using the bits-back trick with the latent variable model in Eq. 1. We assume here that we have an entropy coder that operates as a stack (i.e., "last in first out") such as ANS, and that the code already contains some sufficiently large amount of compressed data from previous unrelated operations before we start with step 1.

Fill in the missing parts, i.e., whether the bit rate grows or shrinks in each step, and by how many bits. Consider the *amortized* bit rate only, i.e., don't bother about rounding to integer values. Then calculate the *net* amortized bit rate for encoding the message $\mathbf{x}$, i.e., how many *more* bits are on the coder after step 3 compared to before step 1. Express the net bit rate in the simplest form possible.

step	operation	what?	with model	bit rate ...	by how much?
1	☐ encode ☒ decode	z	$P(Z \,\|\, \mathbf{X}{=}\mathbf{x})$	☐ grows ☐ shrinks	
2	☒ encode ☐ decode	$\mathbf{x}$	$P(\mathbf{X} \,\|\, Z{=}z)$	☐ grows ☐ shrinks	
3	☒ encode ☐ decode	z	$P(Z)$	☐ grows ☐ shrinks	
net (amortized) bit rate:				☐ grows ☐ shrinks	

(b) **Decoding:** now assume you've received a bit sting that generated with the three steps from part (a). How can you reconstruct both the message $\mathbf{x}$ and the original state of the entropy coder? Fill in the table below so that it inverts all steps from part (a). Can you do this without looking up the decoder in the lecture notes? Remember that we're using a "last in first out"-entropy coder. Verify that, at each step in the table below, the decoder has access to all the required information.

step	operation	what?	with model	bit rate ...	by how much?
1	☐ encode ☐ decode	z		☐ grows ☐ shrinks	
2	☐ encode ☐ decode	$\mathbf{x}$		☐ grows ☐ shrinks	
3	☐ encode ☐ decode	z		☐ grows ☐ shrinks	

(c) Why do we use an entropy coder with *stack* semantics? Try to come up with a similar encoding/decoding scheme as in the two tables above that uses instead an entropy coder with *queue* semantics (i.e., "first in first out"), such as range coding. You'll probably want to change the order of operations in the encoder and/or decoder. But at some point, things will fail. Can you explain why?

Problem 7.2: Bits-Back Coding in ANS

Listing 1 shows our (slow but correct) implementation of an Asymmetric Numeral Systems (ANS) entropy coder from the last lecture. We discussed that ANS itself can already be seen as an application of the bits-back trick, albeit a very simple instance of it. Recall that the latent variable model used by ANS to encode a symbol x_i is

$$Q(X_i) = \sum_{z_i=0}^{n-1} Q(Z_i{=}z_i, X_i) \qquad \text{where} \qquad Q(Z_i, X_i) = Q(Z_i)\, Q(X_i \,|\, Z_i) \tag{2}$$

with

$$Q(Z_i{=}z_i) = \frac{1}{n} \quad \forall z_i \in \{0, \dots, n-1\} \qquad \text{and} \qquad Q(X_i \,|\, Z_i) = \begin{cases} 1 & \text{if } z_i \in \mathfrak{Z}_i(x_i) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where the ranges $\mathfrak{Z}_i(x_i)$ partition the range $\{0, \dots, n{-}1\}$ into pairwise disjoint subranges of sizes $|\mathfrak{Z}_i(x_i)| =: m_i(x_i)$ with $\sum_{x_i \in \mathfrak{X}_i} m_i(x_i) = n$ chosen such that the resulting marginal probabilities $Q(X_i{=}x_i) = \frac{m_i(x_i)}{n}$ approximate given symbol probabilities $P(X_i{=}x_i)$.

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

    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
```

Listing 1: Our implementation of Asymmetric Numeral Systems (ANS) from the last lecture. For a usage example (not needed here), see Listing 2 on Problem Set 6.

(a) Go over the two tables in Problem 7.1. For each step of the encoding and decoding operation, identify the lines in Listing 1 that correspond to this step. You should find that one of the three steps in both the encoder and the decoder is actually not necessary in the specific case of ANS. Can you explain why?

(b) Calculate the contributions to the net amortized bit rate from each step (i.e., evaluate the cells in the last column in either of the two tables in Problem 7.1). Express your results in terms of the integers n and $m_i(x_i)$. What do you obtain for the step you identified in part (a) which doesn't actually correspond to any lines of code in the implementation?

Remark: in contrast to the discussion in Problem 7.1, the latent variable model used by ANS (Eqs. 2-3) uses a separate latent variable Z_i for each symbol X_i. Therefore, in and of itself, ANS does not yet model any correlations between symbols. To encode correlated symbols with ANS, one uses another layer of the bits-back trick *on top of* ANS, exactly as discussed in Problem 7.1.

	msg. len (chars)	bits per character					
		Huffman	Shannon	inf. cont.	`gzip`	`bzip2`	`bzip2'`
validation set	106,864	**2.38**	2.72	2.12	3.43	2.82	2.40
test set	219,561	**2.38**	2.73	2.12	3.33	2.65	**2.38**

Table 1: Empirical bit rates of our autoregressive model from Problem Set 3.

Problem 7.3: Range Coding With an Autoregressive Model for English Text

In Problem 3.2 on Problem Set 3, you trained an autoregressive machine learning model (parameterized by 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 coding: why is it a good idea to replace Huffman coding with a stream code? Table 1 shows the bit rates we obtained back on Problem Set 3. In addition, the column "inf. cont." shows the information content of the validation and test set under the trained model. Make an educated guess: what bit rates do you expect to obtain if you replace Huffman coding with a stream code?

(b) By now, we've discussed several stream codes in the lecture. Why do we choose range coding and not ANS for this model?

(c) 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.[1] Install `constriction` by executing (preferably in a virtual environment):

```
python3 -m pip install constriction~=0.3.1
```

Then try out the first code example from the API documentation of `constriction`'s range coder.[2] 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 here, only the `Categorical` model that's used for encoding `message_part1` in this example.

[1] 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

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

In the following, you'll apply your newly acquired range coding skills to the autoregressive model from Problem 3.2. Don't worry if you haven't completed Problem 3.2, you can always download the proposed solutions[3] from the course website. The PDF document that's part of the solutions also contains instructions for how to set up your virtual environment and train the model (you'll probably have to reinstall `constriction` in the new virtual environment using the same command as in part (c)).

(d) Start with the encoder and don't bother about adding an "end of file" symbol yet (see part (e) below). Rename the file `compression.py`, to `huffman.py`; then copy it to a new file named `range-coding.py`. In this file, remove the classes `HuffmanEncoder` and `HuffmanDecoder`, and replace the substring "`_huffman`" in all function names by "`_range`". Then `import constriction` and replace the `HuffmanEncoder` by a `constriction.stream.queue.RangeEncoder` that you've learned how to use in part (c). To test your (preliminary) encoder, run:

```
python3 range-coding.py shakespeare.pt \
    dat/shakespeare.val.txt encode
```

(You might also want to create a much smaller test file to very quickly check for obvious bugs.)

Do you obtain the bit rate that you were expecting in part (a)?

Hint: When we used Huffman coding in Problem 3.2, we constructed a new instance of `HuffmanEncoder` for each character in the message. This won't work for a stream code like range coding, however, since stream codes have to keep track of an internal coder state so that they can *amortize* bits over multiple encoded symbols. Therefore, `constriction`'s `RangeEncoder` should be constructed only *once per message* (i.e., outside of the loop `for char in tqdm(message)`). Inside the `for`-loop, you should only construct a new `Categorical`[4] entropy model from the symbol probabilities. Then, you can use this `entropy_model` together with the `range_encoder` you constructed outside of the `for`-loop to encode a symbol as follows: `range_encoder.encode(target_py, model)`, where `target_py` is a python integer that represents the character, as in the solutions to Problem 3.2.

Note that the method `encode` does not return any code word—after all, there are no code words in stream codes. The method instead mutates a compressed representation of the message that is held inside the `RangeEncoder`. After the `for`-loop, you can obtain this compressed representation by calling `range_encoder.get_compressed()`, as you've practiced in part (c). This returns the compressed representation as a numpy-array of unsigned 32-bit integers, which you can write to a file with the method `.tofile(filename)`.

[3] https://robamler.github.io/teaching/compress23/problem-set-03-solutions.zip

[4] Documentation: https://bamler-lab.github.io/constriction/apidoc/python/stream/model.html#constriction.stream.model.Categorical

(e) As we've discussed in the lecture, stream codes cannot reliably infer the number of symbols in a variable length-message from the length of the compressed representation. We have to explicitly encode and "end of file" (EOF) signal.

Add the following line of code immediately before constructing the entropy model:

```
extended_probs = np.append(unnormalized_probs.numpy(), 0.0)
```

This extends the list of (unnormalized) symbol probabilities by one more entry with value zero. Then use these `extended_probs` to construct your `Categorical` entropy model. The constructor of `Categorical` internally normalizes the provided probabilities, approximates them in fixed point precision, and it also ensures that no probability gets rounded to zero (as this would make it impossible to encode the corresponding symbol). Thus, it will replace our zero probability for the EOF symbol with the smallest representable positive probability (2^{-24}).

Finally, you need to actually encode a single EOF symbol after encoding the message. Unroll the autoregressive model for one more step after `for`-loop is done (because the decoder will do this too since it doesn't know that the message is over). But when you come to encoding a character, encode instead the EOF symbol, which has index `len(extended_probs) - 1`.

(f) Now port the decoder from Huffman coding to range coding. Analogous to the encoder, construct a single `RangeDecoder` outside of the loop. The constructor expects a single argument, which must be a numpy array of unsigned 32-bit integers that contains the compressed representation. You can read it from a file with `np.fromfile(filename, np.uint32)`. Inside the loop, you'll again want to construct a `Categorical` entropy model from the `extended_probs` as in part (e). Then decode a symbol with `char_index = range_decoder.decode(entropy_model)`. Break out of the loop if `char_index` is the EOF symbol; otherwise, look up the character indexed by `char_index` and print it, as in the solutions to Problem 3.2.

Encode and then decode some file in the `dat` subdirectory and verify that the decoder reconstructs the original data.

(g) Notice that, in part (e), we extend the alphabet by an EOF symbol even while we are still inside the `for`-loop, i.e., even when we know that we won't encode it. Why is this necessary? Wouldn't it suffice to include the EOF symbol in the alphabet only when we actually need it? Try it out: remove the part of the encoder that adds the EOF symbol to the alphabet inside the `for`-loop (where we seemingly don't need it). Then encode and decode some data. Does decoding reconstruct the original data?