# Solutions to Problem Set 7

*discussed:*
*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

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

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 $P(\mathbf{X} \mid Z) = \prod_{i=1}^{k} P(X_i \mid Z)$ 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 coder already contains some sufficiently large amount of compressed data from previous unrelated operations before we start with step 1.

Complete the table below, i.e., fill in whether the bit rate grows or shrinks in each step, and by how many bits. Consider only the *amortized* bit rate, i.e., don't bother about rounding to integers. 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<br>☒ decode | $z$ | $P(Z \mid \mathbf{X}{=}\mathbf{x})$ | ☐ grows<br>☒ shrinks | $-\log_2 P(Z{=}z \mid \mathbf{X}{=}\mathbf{x})$ |
| 2 | ☒ encode<br>☐ decode | $\mathbf{x}$ | $P(\mathbf{X} \mid Z{=}z)$ | ☒ grows<br>☐ shrinks | $-\log_2 P(\mathbf{X}{=}\mathbf{x} \mid Z{=}z)$ |
| 3 | ☒ encode<br>☐ decode | $z$ | $P(Z)$ | ☒ grows<br>☐ shrinks | $-\log_2 P(Z{=}z)$ |
| net (amortized) bit rate: | | | | ☒ grows<br>☐ shrinks | $-\log_2 \underbrace{\frac{P(\mathbf{X}{=}\mathbf{x} \mid Z{=}z)P(Z{=}z)}{P(Z{=}z \mid \mathbf{X}{=}\mathbf{x})}}_{P(\mathbf{X}=\mathbf{x})}$ |

**Solution:** See filled in table above. Here, the net (amortized) bit rate of $-\log_2 P(\mathbf{X}=\mathbf{x})$ was calculated by adding the two bit rates from steps 2 and 3 (where we encode, i.e., generate bits) and subtracting from it the bit rate from step 1 (where we decode, i.e., consume bits). ■

(b) **Decoding:** now assume you've received a bit string 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 |
|---|---|---|---|
| 1 | ☐ encode<br>☒ decode | $z$ | $P(Z)$ |
| 2 | ☐ encode<br>☒ decode | $\mathbf{x}$ | $P(\mathbf{X} \mid Z=z)$ |
| 3 | ☒ encode<br>☐ decode | $z$ | $P(Z \mid \mathbf{X}=\mathbf{x})$ |

**Solution:** See filled in table above. Here, we inverted (i.e., swapped encoding with decoding) each step from the encoder and then reversed their order (since we assumed that the employed entropy coder has "last in first out" semantics). With this reversed order, each step of the decoder indeed has access to all required information: in step 2, the decoder needs the value of $z$ in the likelihood $P(\mathbf{X} \mid Z=z)$, which it knows from step 1; and in step 3, the decoder can (in principle) calculate the posterior $P(Z \mid \mathbf{X}=\mathbf{x})$ because the message $\mathbf{x}$ is known from step 2. ■

(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 go awry. Can you explain why?

**Solution:** Encoder and decoder have to execute their respective steps 1 and 3 in opposite order, which is why "first in first out" semantics won't work. In contrast to the message $\mathbf{x}$, the latent $z$ is not known to the encoder before encoding starts. Instead, the encoder obtains $z$ by decoding from some existing data (step 1 of the encoder). Since the encoder then encodes the message $\mathbf{x}$ conditioned on the obtained $z$, *decoding* $\mathbf{x}$ requires knowledge of $z$. Therefore, the encoder needs to encode $z$ as well in order to communicate it to the receiver (step 3 of the encoder). Obviously, the encoder can encode $z$ only *after* obtaining it.

```python
class SlowAnsCoder:
    def __init__(self, precision, compressed=0):
        self.n = 2**precision    # ("**" denotes exponentiation.)
        self.uniform_coder = UniformCoder(compressed) # → Problem Set 6

    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.

> The decoder has to invert each step of the encoder: encoding $z$ becomes decoding $z$ and vice versa. Since $z$ is not known in advance, the decoder can encode it only after having decoded it, i.e., it has to execute the inverse of encoder step 3 *before* the inverse of encoder step 1. This requires a *stack*-based entropy coder. ■

## Problem 7.2: Bits-Back Coding in ANS

Listing 1 shows our implementation of a (slow but simple) 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 single `symbol` $x_i \in \mathfrak{X}_i$ 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

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

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

(a) Go over the two tables in Problem 7.1. For each step of the encoding and decoding operation, identify the line 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?

**Solution:** When analyzing complex entropy coding tricks, it is often easiest to start with the *decoder*, which is more constrained than the encoder because it has less information at its disposal (i.e., it doesn't initially know the message). Once the decoding operation is specified, encoding can be seen as inference over the decoder (i.e., the encoder technically doesn't need to follow any fixed specification as long as it can *somehow* "trick" the decoder into decoding the desired message).

- For the *decoder*, we find:
  - Step 1: *Decode $z_i$ with prior $Q(Z_i)$:* line 11 in Listing 1, which calls the decoding method `pop` on an encapsulated (trivial) entropy coder to obtain a value $z_i$. Since we use a `UniformCoder` with `base=self.n` here, this encoding operation uses a uniform entropy model over $\{0,\dots,n-1\}$, corresponding to $Q(Z_i)$ (see Eq. 3).
  - Step 2: *Decode $x_i$ with likelihood $Q(X_i \,|\, Z_i{=}z_i)$:* this operation is not necessary in the case of ANS because the likelihood in Eq. 3 is deterministic: once the decoder knows $z_i$, there is only a single $x_i$ for which $Q(X_i{=}x_i \,|\, Z_i{=}z_i)$ is nonzero. Thus, we can identify $x_i$ (see lines 14-18) without the need to decode any additional data.
  - Step 3: *Encode $z_i$ with posterior $Q(Z_i \,|\, X_i{=}x_i)$:* line 19, which calls the encoding method `push` to encode $z_i$. We'll see in part (b) that the entropy model employed on this line is indeed the posterior.
- For the *encoder*, we find accordingly:
  - Step 1: *Decoding $z_i$ with posterior $Q(Z_i \,|\, X_i{=}x_i)$:* line 7.
  - Step 2: *Encoding $x_i$ with likelihood $Q(X_i \,|\, Z_i{=}z_i)$:* not necessary for ANS; as discussed above, the decoder can identify $x_i$ from $z_i$ alone.
  - Step 3: *Encode $z_i$ with prior $Q(Z_i)$:* line 8.

■

(b) Calculate the contribution to the net amortized bit rate from each step (i.e., evaluate the cells in the last column of the table in Problem 7.1 (a)). You'll need to calculate the posterior $Q(Z_i \,|\, X_i{=}x_i)$ for this. Express your results in terms of the integers $n$ and $m_i(x_i)$. In part (a), you identified a step that doesn't actually correspond to any lines of code in Listing 1. What do you obtain for this step's contribution to the net amortized bit rate in the case of ANS?

**Solution:** We first calculate the posterior distribution $Q(Z_i{=}z_i \,|\, X_i{=}x_i)$. In the calculation below, we'll need the marginal data distribution $Q(X_i{=}x_i)$, which is given by $\sum_{z_i=0}^{n-1} Q(Z_i{=}z_i)\, Q(X_i{=}x_i \,|\, Z_i{=}z_i)$. In this sum, the prior $Q(Z_i{=}z_i)$ is always $\frac{1}{n}$, and the likelihood evaluates to one for the $|\mathfrak{Z}_i(x_i)| = m_i(x_i)$ terms where $z_i \in \mathfrak{Z}_i(x_i)$ and to zero otherwise. Thus, we have $Q(Z_i{=}z_i) = \frac{m_i(x_i)}{n}$, as was already claimed in the problem statement. We then obtain therefor posterior

$$Q(Z_i \,|\, X_i) = \frac{Q(Z_i, X_i)}{Q(X_i)} = \frac{Q(Z_i)\, Q(X_i \,|\, Z_i)}{Q(X_i)}$$

$$\Longrightarrow Q(Z_i{=}z_i \,|\, X_i{=}x_i) = \frac{1/n}{m_i(x_i)/n}\, Q(X_i{=}x_i \,|\, Z_i{=}z_i) = \begin{cases} \frac{1}{m_i(x_i)} & \text{if } z_i \in \mathfrak{Z}_i(x_i) \\ 0 & \text{otherwise.} \end{cases}$$

In other words, $Q(Z_i \,|\, X_i{=}x_i)$ is a *uniform distribution over* $\mathfrak{Z}_i(x_i)$.

We then find for the contributions to the amortized bit rate for the three steps of the *encoder* (decoder steps have opposite sign and are numbered in reverse):

Step 1: *Decoding $z_i$ with posterior $Q(Z_i \,|\, X_i{=}x_i)$:*
consumes $-\log_2 Q(Z_i{=}z_i \,|\, X_i{=}x_i) = \log_2 m_i(x_i)$ bits
since this step always decodes a $z_i$ that satisfies $z_i \in \mathfrak{Z}_i(x_i)$.

Step 2: *Encoding $x_i$ with likelihood $Q(X_i \,|\, Z_i{=}z_i)$:*
contributes $-\log_2 Q(X_i{=}x_i \,|\, Z_i{=}z_i) = -\log_2(1) = 0$ bits,
reflecting the observation from part (a) that, once $z_i$ is known, no additional information is needed to obtain $x_i$ in this specific model.

Step 3: *Encode $z_i$ with prior $Q(Z_i)$:*
contributes $-\log_2 Q(Z_i{=}z_i) = \log_2 n$ bits.

Thus, the net bit rate of encoding a single symbol $x_i$ with ANS works out to be $\log_2 n - \log_2 m_i(x_i) = -\log_2 \frac{m_i(x_i)}{n} = -\log_2 Q(X_i{=}x_i)$ bits, as discussed in the lecture. ■

**Remark.** The 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 effectively with ANS, one uses another layer of the bits-back trick *on top of* ANS, precisely 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?

**Solution:** By amortizing compressed bits over multiple symbols, a stream code like range coding can achieve better bit rates than a symbol code like Huffman coding. A good stream code implementation should achieve a bit rate very close to the information content. Thus, based on Table 1, we expect to obtain 2.12 bits per character when we use a range coder with our autoregressive model. This would correspond to about 11 % reduction in bit rate compared to Huffman coding.

Since the overhead of symbol codes is proportional to the number of symbols in a message, the *relative* impact of amortization is more significant for compression methods that spread the information content of a message over more symbols, as is often the case in lossy image compression (discussed starting in Lecture 9). ■

(b) Why do we choose to use range coding and not ANS for this model?

**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 have to unroll the autoregressive model in the same direction anyway.

It would technically be possible to use a stack-based entropy coder with an autoregressive model. But the encoder would first have to unroll the model for the entire message and *remember the model parameters for all symbols.* Then, the encoder

would have to encode the symbols in reverse order, so that the decoder can decode in normal order (the decoder can't decode in reverse order because it wouldn't be able to unroll the model to the end without knowledge of the message). ■

(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 reference 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.

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 port the functions `encode_range` and `encode_range_file` by applying what you've learned in part (c) about the class `constriction.stream.queue.RangeEncoder`.

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.)

[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

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

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 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. Assuming this model is called `entropy_model`, you can use it together with the `range_encoder` that you constructed outside the `for`-loop to encode a symbol as follows: `range_encoder.encode(target_py, model)`, where `target_py` is an integer that identifies 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 growing compressed representation encapsulated by 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)`.

**Solution:** The solutions are provided in the accompanying `git` bundle.

- If you already have a directory with the code from Problem Set 3 or its provided solutions, then `cd` into that directory and run:

```
git stash
git checkout problem-set-3
git pull path/to/autoreg-range-coding-solutions.gitbundle
source venv/bin/activate
```

- If you don't yet have a directory with the code, run instead:

```
git clone path/to/autoreg-range-coding-solutions.gitbundle \
    char-rnn-compression
cd char-rnn-compression
python3 -m pip install virtualenv
python3 -m virtualenv -p python3 venv
source venv/bin/activate
python3 -m pip install torch tqdm unidecode constriction~=0.3.1
```

If you haven't done so already, train the model with the following command:

```
python3 train.py dat/shakespeare.txt
```

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

While python libraries are being installed or the model is being trained, read and understand the implementation in the file `range-coding.py`. Refer to this problem set for explanations (note that the code also covers parts (e) and (f) below).

To test an encoder/decoder round trip, run (after having trained the model):

```
python3 range-coding.py shakespeare.pt \
    dat/shakespeare.val.txt encode
ls -l dat/shakespeare.val.txt.compressed
python3 range-coding.py shakespeare.pt \
    dat/shakespeare.val.txt.compressed decode > decoded.txt
diff dat/shakespeare.val.txt decoded.txt
```

Once the encoder is done, it prints both the information content of the encoded text and the actual bit rate. You will likely obtain a slightly different information content than what's reported in Table 1 since we didn't set a random seed when training the model. But what's important is that the bit rate should be very close to the information content (less than 0.1 % overhead). The bit rate should also exactly match the file size reported by `ls` (multiply by 8 to convert from bytes to bits). The above `diff` command should terminate successfully without printing anything, indicating that the reconstructed text equals the original one. ■

(e) As we've discussed in the lecture, when decoding a variable length-message, stream codes cannot use the length of the compressed representation to reliably infer the length of the message. 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). Then, instead of encoding another character, encode the EOF symbol, which has index `len(extended_probs) - 1`.

**Solution:** See accompanying `git` bundle, which can be extracted as described in the solutions to part (d) above. ■

(f) Now port the decoder from Huffman coding to range coding. Analogous to the encoder, construct a single `RangeDecoder` outside of the loop in the function

`decode_range`. 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 by using the statement `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 the validation and test files in the `dat` subdirectory and verify that the decoder reconstructs the original data.

**Solution:** See accompanying `git` bundle, which can be extracted as described in the solutions to part (d) above. ■

(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 the EOF symbol at this point. Why is this necessary? Wouldn't it suffice to include the EOF symbol in the alphabet only when we actually need it, i.e., only in the single additional step *after* the `for` loop? Try it out: undo the change in 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 it still work?

**Solution:** You will probably be able to decode a few thousand characters without any issues. But unless you're very lucky (or unlucky?), at some point, you'll either get an error message upon decoding or you'll decode a wrong message.

In lossless compression, encoder and decoder have to use *precisely* the same entropy model. Since the decoder doesn't know where the message ends until it has fully decoded it, the decoder has to use an entropy model that assigns a nonzero probability to the EOF symbol *at every position in the message*. Therefore, so does the encoder. ■