Problem Set 3

published: 3 May 2023
discussion: 10 May 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 3.1: Kullback-Leibler Divergence

In the lecture, we introduced the Kullback-Leibler (KL) divergence, or relative entropy, D_{KL}. The KL-divergence is an information-theoretical measure of mismatch between two probability distributions. We discussed that $D_{\mathrm{KL}}\big(p_{\mathrm{data}}(\mathbf{x}) \,\|\, p_{\mathrm{model}}(\mathbf{x})\big)$ quantifies by how much the expected bit rate of an optimal lossless compression code increases when we use a model p_{model} that does not perfectly match the (unknown) distribution p_{data} of the true data generative process. Thus, the KL-divergence is defined as follows,

$$D_{\mathrm{KL}}\big(p_{\mathrm{data}}(\mathbf{x}) \,\|\, p_{\mathrm{model}}(\mathbf{x})\big) := H(p_{\mathrm{data}}, p_{\mathrm{model}}) - H(p_{\mathrm{data}}). \tag{1}$$

The KL-divergence is a useful quantity in more than just data compression. For example, we will see it come up again when we introduce variational inference in Lecture 8.

(a) Convince yourself that the following two expressions are valid formulations of the KL divergence:

$$D_{\mathrm{KL}}\big(p(\mathbf{x}) \,\|\, q(\mathbf{x})\big) = \mathbb{E}_{\mathbf{x}\sim p}\big[\log p(\mathbf{x}) - \log q(\mathbf{x})\big] = \mathbb{E}_{\mathbf{x}\sim p}\left[\log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right] \tag{2}$$

Here, the notation $\mathbb{E}_{\mathbf{x}\sim p}[f(\mathbf{x})] := \sum_{\mathbf{x}} p(\mathbf{x}) f(\mathbf{x})$ denotes the expectation value of some function f under the probability distribution p. (In practice, we sometimes can't evaluate $p(\mathbf{x})$ and therefore can't calculate the weighted sum over all $\mathbf{x}$, but we might be able to *estimate* $\mathbb{E}_{\mathbf{x}\sim p}[f(\mathbf{x})]$ by averaging $f(\mathbf{x})$ over samples from a finite training set or test set, as discussed in the lecture.)

Note: This is a fairly trivial exercise but Eqs. 1 and 2 are both important to remember.

(b) Since D_{KL} measures the overhead in expected bit rate over its lower bound we kind of already know that it cannot be negative. But let's prove this in a more direct way. The proof uses Jensen's inequality (see Figure 1 on the next page), which states that, for any *convex* function f and any probability distribution p, we have:

$$f\big(\mathbb{E}_{\xi\sim p}[\xi]\big) \leq \mathbb{E}_{\xi\sim p}\big[f(\xi)\big] \qquad \text{(for convex } f\text{)}. \tag{3}$$

Prove that $D_{\mathrm{KL}}\big(p(\mathbf{x}) \,\|\, q(\mathbf{x})\big) \geq 0$ for all probability distributions p and q by using Eq. 2, Jensen's inequality, and the fact that the function $f(\xi) := -\log \xi$ is convex.

Note: Jensen's inequality (Eq. 3) is a very useful relation that often comes up when proving bounds in information theory and in approximate Bayesian inference (scheduled for Lecture 8).

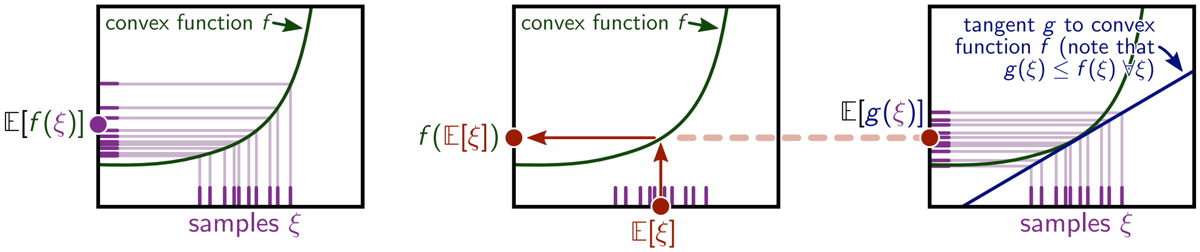


Figure 1: Illustration of Jensen's inequality (Eq. 3). Left: $\mathbb{E}[f(\xi)]$ for some convex function f. Center: $f(\mathbb{E}[\xi])$ for the same convex function f. Right: $\mathbb{E}[g(\xi)])$ where g is the affine linear function whose graph (blue) is a tangent to f, touching it at the point $(\mathbb{E}[\xi], f(\mathbb{E}[\xi]))$. Since f is convex, the tangent g to it satisfies $g(\xi) \le f(\xi)\ \forall\xi$ and thus $\mathbb{E}[g(\xi)] \le \mathbb{E}[f(\xi)]$. Further, since g is affine linear, it can be pulled out of the expectation: $\mathbb{E}[g(\xi)] = g(\mathbb{E}[\xi]) = f(\mathbb{E}[\xi])$. Thus, in total, $f(\mathbb{E}[\xi]) \le \mathbb{E}[f(\xi)]$ for any convex function f, as claimed in Eq. 3.

Problem 3.2: Lossless Compression of Natural Language With Recurrent Neural Networks

This `zip`-file contains code for a simple character-based autoregressive language model, forked from a GitHub repository[1] by Sean Robertson. You will learn more about autoregressive models in the next lecture, but Figure 2 on the next page should tell you enough to solve this problem.

In this problem, you will train the model and then turn it into a compression method, whose performance you evaluate empirically. Although your resulting compression method will already be quite effective (considering its simplicity), it will still waste some bit rate, and it will also be very slow. You will improve upon it in upcoming problem sets as you learn more about deep probabilistic models and entropy coders.

The code comes as a git bundle. To extract it, run:

```
git clone path/to/char-rnn-compression.gitbundle char-rnn-compression
```

You'll also need the python packages PyTorch, `numpy`, `tqdm`, and `unidecode`. You can install them, e.g., as follows (or use your favorite package manager instead):

```
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 numpy
```

Once everything is set up, it's time to get your hands dirty.

---

[1] `https://github.com/spro/char-rnn.pytorch`

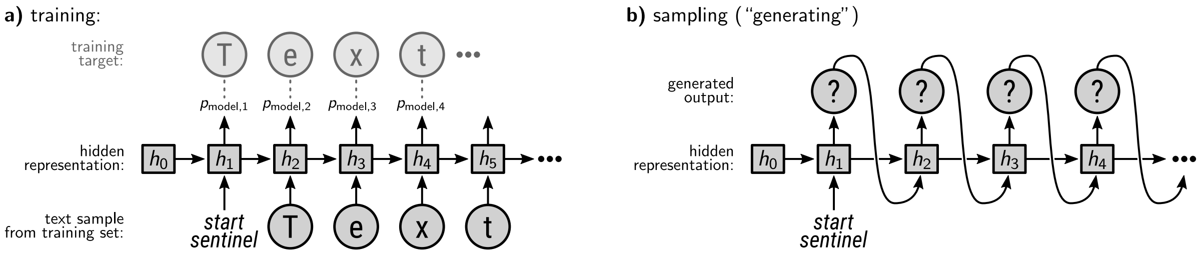


Figure 2: Autoregressive model for character based text generation. a) The training algorithm minimizes the cross entropy between the true distribution of English text (estimated via samples from a training set) and a probabilistic model parameterized by a so-called recurrent neural network. The model reads text character by character (bottom row). After reading character i, the model parameterizes a probability distribution $p_{\text{model},i+1}$ over the next character. The training algorithm then tries to minimize the information content of the correct next character (top row) under $p_{\text{model},i+1}$. b) In addition to a training objective, the model implementation also already comes with a function `generate` that samples from a trained model. The function draws a random first character $x_1 \sim p_{\text{model},1}$, prints it, then feeds it back into the model in order to calculate $p_{\text{model},2}$, from which the function draws the next character x_2, and so on.

(a) The repository contains some toy data set of historic English text[2] in the directory `dat`, together with a canonical random split (by lines) into training, validation, and test set. Train the model on the training set by executing:

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

Training this small model doesn't require any fancy hardware; it should only take about 10 to 20 minutes on a regular consumer PC.

The script will use the training set at `dat/shakespeare.train.txt`. Before training and after every tenth training epoch, the script will evaluate the model's performance on the validation set (`dat/shakespeare.val.txt`) and it will print out the cross entropy (to base 2). In regular intervals, the script will also print out some samples from the model (i.e., random generated text). You should be able to observe that the cross entropy decreases (because that's the objective function that the training procedure minimizes), and the generated text should resemble more and more the kind of text you can find in the training set. At the end of training, the cross entropy should oscillate roughly around 2 bits per character.

The trained model will be saved to a file named `shakespeare.pt`. You can now evaluate it on the validation or test set:

```
python3 evaluate.py shakespeare.pt dat/shakespeare.val.txt
python3 evaluate.py shakespeare.pt dat/shakespeare.test.txt
```

---

[2] Downloaded from https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt

(b) While the model is training, familiarize yourself with the code in `evaluate.py` and in `generate.py`, and try to understand what the functions `evaluate` and `generate` do. What does calling `torch.multinomial(output_dist, 1)` in the function `generate` achieve, and what quantities does `output_dist` contain?

Note: Both `evaluate` and `generate` take an argument named `decoder`. Despite its name, this argument is *not* a decoder in the sense of data compression. It is just the trained model (the original code base had no relation to data compression).

(c) You should have observed that the function `generate` generates random text. This is possible because the trained model parameterizes a *probability distribution* p_{model} over character sequences, so one can draw random samples from this distribution. However, in a compression application, we don't want to generate *random* text. We want the receiver to be able to *deterministically* decode the exact same text that the sender encoded. How can you achieve this using the trained probabilistic model and an entropy coder. Make a sketch similar to Figure 2 to illustrate how you would approach encoding and decoding. Where do you generate/consume code words and what probability distributions do you use to build the corresponding code books?

(d) Let's now implement the encoder. Create a new file `compression.py` and paste your implementations of `HuffmanEncoder` and `HuffmanDecoder` from Problem Sets 1 and 2 into it (if you didn't solve these problem sets, use the solutions from the course website[3]). Then implement a function `encode_huffman` with signature

```
def encode_huffman(model, message):
```

Here, the argument `model` is a trained model (the same as the confusingly named `decoder` argument in the function `evaluate`), and the argument `message` is a string of English text. The function should compress `message` using Huffman coding, and `yield` the resulting sequence of bits (boolean values). Once the entire message has been processed, the function should print some relevant statistics, such as the total bit rate, the total information content, and the bit rate that you would have obtained had you used Shannon coding instead of Huffman coding.

Your implementation of `encode_huffman` will be similar to the function `evaluate`. Remember that the autoregressive model builds a new probability distribution for each character, so you'll have to build up a new Huffman tree for each character.

Then turn your implementation into an executable script by parsing some appropriate command line arguments (similar to how it is done in `evaluate.py`), reading `message` from a text file, and writing the compressed representation to a new (binary) file. You can use the utilities provided to you in `bitutils.py` for generating a binary file (bring them into scope with `from bitutils import *`).

Hint: you can apply the Huffman coding algorithm directly to an unnormalized probability distribution (i.e., to `logits.exp().numpy()`). This works because the overall scale doesn't affect how the Huffman tree will be constructed.

---

[3] https://robamler.github.io/teaching/compress23/

(e) **This is the most important part of this problem set:** evaluate the compression performance of your implementation on some sample texts. Try out different kinds of texts, ranging from the test set (which should be very similar to the training set) to more modern English text (e.g., a Wikipedia page), and to text in a different language. Compare your method's bit rate to:

- the information content of the respective texts under the model;
- the bit rate had you used Shannon coding instead of Huffman Coding;
- standard lossless compression techniques such as `gzip` or `bzip2` (make sure you use the `--best` switch when running these baselines); and to
- the file size that you obtain when you take the output of your Huffman coding based compression method and try to compress it further with `gzip` or `bzip2`.

Report your results in *bits per character* so that you can compare compression performance across texts of varying lengths.

Note: This simple toy model only supports ASCII characters, so make sure that your messages don't contain, e.g., German umlauts, fancy quotation marks, etc.

Discuss your results: Which compression method works best? How important is it that the training data resembles the text we end up compressing? Does compressing already compressed data help? How much improvement can you expect at most if you'd use a so-called stream code, i.e., a lossless compression code that is not a symbol code and that can therefore be more effective than Huffman coding?

(f) Implement a *decoder* and verify empirically for some sample text that decoding an encoded message reconstructs the original message. Use the `HuffmanDecoder` class that you implemented on Problem Set 2 and the function `read_bits_from_file` that is provided to you in the file `bitutils.py`.

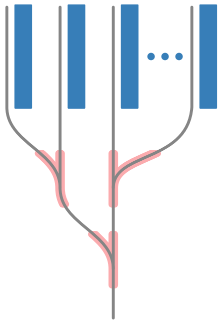
Hint: the method `decode` on the `HuffmanDecoder` class is a *generator function* that allows you to decode several symbols in sequence, reusing the same `HuffmanDecoder`. However, in this autoregressive model, you have to construct a new `HuffmanDecoder` for each character. Therefore, you'll only want to decode a single character each time. You can use the builtin `next` function to do this:

```
character_index = next(huffman_decoder.decode(bit_iterator))
```

Here, `character_index` is an integer that represents a single character according to conventions that are specific to this machine-learning model. In order to turn `character_index` into an actual printable character, have a look at how this is done in `generate.py`.

Congratulations, you have implemented your first machine-learning based compression method! We'll improve upon this method on Problem Set 6.

Problem 3.3: A Different Kind of Trolley Problem



Don't let the title scare you away, we won't be doing any pretentious pseudophilosophy here. But we'll solve a problem that has nothing to do with data compression—or does it?

Imagine you are designing a train station with n platforms labeled $\{1, \ldots, n\}$ (blue rectangles in illustration). Each platform serves a single railroad track (gray lines), and all tracks enter the station from the same direction and terminate at the station. You may order the platforms $\{1, \ldots, n\}$ arbitrarily, but they must be arranged next to each other and in parallel, as in the illustration.

Here's the difficult part: the train station will serve n different *types* of trains, and each platform $i \in \{1, \ldots, n\}$ can serve only trains of type i (think of cargo trains with different kinds of cargo, such as containers, gasoline, lumber, and coal, where each type of cargo requires a designated platform with specialized equipment for loading and unloading). The various types of trains have varying average physical weights, and they arrive at the station with varying frequencies. We denote by $w_i > 0$ the total physical weight (in tons) of trains of type i that we expect to arrive at the station per year.

A single track leaves the train station and is connected to all platforms via some switches (highlighted red in the illustration). Your task is to come up with an arrangement of switches and a convenient order in which to arrange the platforms $\{1, \ldots, n\}$. Each switch splits one track into two. Unfortunately, switches are fragile structures: each time a train passes over a switch, the switch takes some damage proportional to the train's weight. How would you come up with a setup that uses as few switches as possible, and that minimizes the total damage on all switches per year for given $(w_i)_{i=1}^n$?

Note: This is problem is obviously contrived. But similar optimization problems are conceivable in sampling problems in statistics, in planning problems (e.g., when a robot explores a sequential decision space), or in routing problems for computer networks.