# Problem Set 3

*published: 5 May 2022*
*discussion: 13 May 2022*

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

Course materials available at https://robamler.github.io/teaching/compress22/

## Problem 3.1: Kullback-Leibler Divergence

In the lecture, we introduced two probability distributions, $p_{\text{data}}$ and $p_{\text{model}}$. Here,

- $p_{\text{data}}$ is the true distribution of the data source, which we typically don't know, but we may have a data set of empirical samples from it (e.g., a data set of uncompressed images if we're concerned with image compression); and
- $p_{\text{model}}$ is an approximation of $p_{\text{data}}$ that we use to construct our lossless compression code; for now, we assume that we can explicitly evaluate $p_{\text{model}}(\mathbf{x})$ for any hypothetical message $\mathbf{x}$.

We showed that, if a lossless compression algorithm is optimal with respect to $p_{\text{model}}$ then its *expected bit rate* on data from $p_{\text{data}}$ is given by the cross entropy $H(p_{\text{data}}, p_{\text{model}})$,

$$\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\big[R(\mathbf{x})\big] = H(p_{\text{data}}, p_{\text{model}}) + \varepsilon \equiv -\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\big[\log p_{\text{model}}(\mathbf{x})\big] + \varepsilon. \tag{1}$$

Here, $\varepsilon$ is a tiny overhead that is irrelevant for practical purposes (assuming long messages $\mathbf{x}$), the bit rate $R(\mathbf{x})$ denotes the total length (in bits) of the compressed representation of a message $\mathbf{x}$, and the notation $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[R(\mathbf{x})] := \sum_{\mathbf{x}} p_{\text{data}}(\mathbf{x})R(\mathbf{x})$ denotes the (formal) expectation value of $R(\mathbf{x})$ under the probability distribution $p_{\text{data}}$ (in practice, we can't *evaluate* $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[R(\mathbf{x})]$ because we don't know $p_{\text{data}}(\mathbf{x})$, but we can *estimate* $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[R(\mathbf{x})]$ by averaging $R(\mathbf{x})$ over samples from a finite training set or test set).

Since we can only use $p_{\text{model}}$ but not $p_{\text{data}}$ to construct our lossless compression algorithm, any deviation between the two probability distributions will degrade compression performance, and the expected bit rate will exceed the fundamental lower bound from Eq. 1. We defined the overhead in expected bit rate due to a mismatch between $p_{\text{model}}$ and $p_{\text{data}}$ as the Kullback-Leibler divergence,

$$D_{\text{KL}}(p_{\text{data}}\,||\,p_{\text{model}}) := H(p_{\text{data}}, p_{\text{model}}) - H(p_{\text{data}}). \tag{2}$$

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

$$D_{\text{KL}}(p\,||\,q) = \mathbb{E}_{\mathbf{x}\sim p}\big[\log p(\mathbf{x}) - \log q(\mathbf{x})\big] = \mathbb{E}_{\mathbf{x}\sim p}\left[\log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right] \tag{3}$$

(This is a fairly trivial exercise but Eqs. 2 and 3 are both important to remember.)

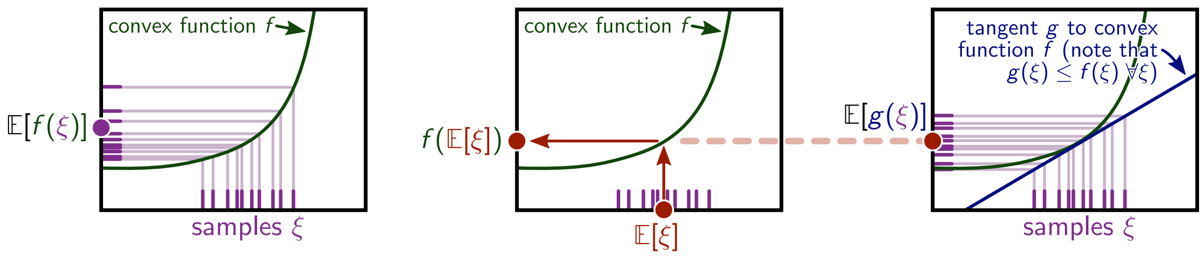

Figure 1: Illustration of Jensen's inequality. Left: $\mathbb{E}[f(\xi)]$ for some convex function $f$. Center: $f(\mathbb{E}[\xi])$ for the same convex function $f$. Right: $\mathbb{E}[g(\xi)])$ where $g$ is the affine linear function whose graph (blue) is a tangent to $f$, touching it at the point $(\mathbb{E}[\xi], f(\mathbb{E}[\xi]))$. Since f is convex, the tangent $g$ to it satisfies $g(\xi) \leq f(\xi)\ \forall \xi$ and thus $\mathbb{E}[g(\xi)] \leq \mathbb{E}[f(\xi)]$. 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, $f(\mathbb{E}[\xi]) \leq \mathbb{E}[f(\xi)]$ for any convex function $f$.

(b) Since $D_{\mathrm{KL}}$ measures the overhead in expected bit rate over its fundamental 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), which states that, for any *convex* function $f$ and any probability distribution $p$, we have:

$$f\big(\mathbb{E}_{\xi\sim p}[\xi]\big) \leq \mathbb{E}_{\xi\sim p}\big[f(\xi)\big] \qquad \text{(for convex } f\text{).} \tag{4}$$

Prove that $D_{\mathrm{KL}}(p\,||\,q) \geq 0$ using Eq. 3, Jensen's inequality, and the fact that the function $f(\xi) = -\log \xi$ is convex.

*Note:* Jensen's inequality is often useful to prove bounds in information theory and in approximate Bayesian inference (scheduled for Lecture 7 or 8).

## 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. It is a fork of the `char-rnn.pytorch-repository`[1] on GitHub. We will talk more about autoregressive models in the next lecture, but Figure 2 should give you enough of an overview to dive into the code. In this problem, you will first train the model on some toy training data. You will then use the trained model to implement your own lossless compression codec for text, which you will evaluate empirically by comparing its bit rate to theoretical bounds and to existing lossless compression methods.

Although the compression codec you'll implement in this problem will already be quite effective (considering its simplicity), it will still be suboptimal, and it will also be very slow. We will improve upon it in upcoming problem sets as we learn about better compression techniques.

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

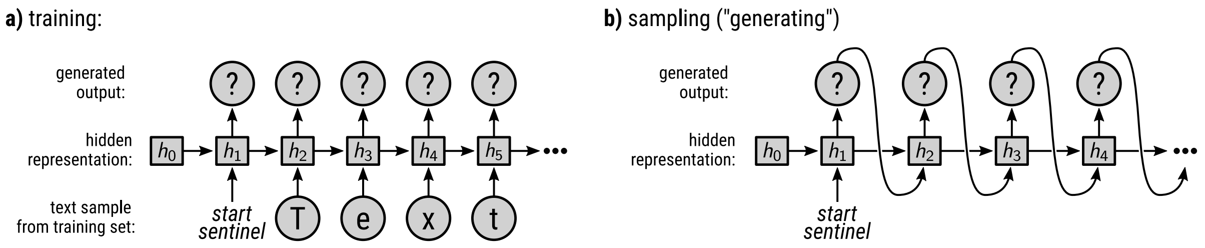

Figure 2: Autoregressive model for character based text generation. a) Training: at training time, the model reads in some English text character by character. After reading character $i$, it models a probability distribution $p_{\text{model},i+1}$ for the next character $x_{i+1}$. The training algorithm minimizes the cross entropy between the true data distribution (estimated via samples from a training set) and the modeled probability distribution. b) Sampling, as implemented in the function `generate`: out of the box, the model already comes with a function that generates random text using a trained model. The function draws a random sample $x_1 \sim p_{\text{model},1}$, which it outputs and then also feeds back into the model so that it can then calculate $p_{\text{model},2}$, from which it draws the next character $x_2$, and so on.

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

```
git clone 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
```

The repository contains some toy data set of (historic) English text[2] in the directory `dat`. In order to allow us to compare results quantitatively, the directory also contains a canonical random split into training, validation, and test set.

(a) Train the model on the training set:

```
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

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

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
```

(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 method `generate` achieve? (In particular, you should understand that `output_dist` is an unnormalized probability distribution here.)

*Note:* Both function signatures contain an argument with name `decoder`. The name of this argument is reminiscent of the naming convention in the original code repository, which was not implemented with data compression in mind. Despite its name, this argument is *not* a decoder in the sense of data compression. It is just the trained model.

(c) You should have observed that the function `generate` generates random text. This is possible because the trained model parameterizes a probability distribution $p_{\text{model}}$ over character sequences, so one can draw random samples from it. However, in compression, 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. Make a sketch similar to Figure 2 to illustrate your approaches for encoding and decoding. Where do you generate/consume code words and what probability distributions do you use to build the corresponding code books.

(d) Create a new file `compression.py` that contains a function `encode_huffman` with the following (or a similar) signature:

```
def encode_huffman(model, message, length_only=False):
```

Here, the argument `model` is a trained model (this is called `decoder` in the the function `generate`) and the argument `message` is a string of English text. The function should return bit string, i.e., the compressed representation of `message`. If the boolean switch `length_only` is set to `True` then the function shouldn't really build up the compressed representation. Instead, it should only simulate the process and return the bit rate, i.e., the length (in bits) of the compressed representation. This is for your convenience, since in the evaluation you'll mostly be interested only in the file size and not in the actual contents of the file.

In order implement `encode_huffman`, you'll need to copy and paste your implementation of the Huffman Coding algorithm from Problem Set 1 (if you didn't solve Problem Set 1, use the solution provided on the course website[3]).

*Hint 1:* start by copying the body of the function `evaluate`, then adjust it to your needs. You'll have to build up a different Huffman tree for every single character in the message (because the probability distribution is different for every character).

*Hint 2:* you can apply the Huffman coding algorithm directly to an unnormalized probability distribution (i.e., to `logits.exp()`). This works because the overall scale doesn't affect how the Huffman tree will get constructed.

(e) Evaluate the compression performance of your implementation on some sample texts. Try it out on different kinds of texts, ranging from the validation set (which should be very similar to the training set) to more modern English text (e.g., a Wikipedia page) to text in a different language. Compare your codec's bit rate to:

- the information content ($-\sum_{i=1}^{k} \log_2 p_{\text{model},i}(x_i)$ where $p_{\text{model},i}$ is the probability distribution for the $i$'th character) of the message $\mathbf{x}$ that was provided to the `encode` function (calculating the information content will be very similar to the implementation of the *evaluate* function);
- the bit rate had you used Shannon coding instead of Huffman Coding (this is $\sum_{i=1}^{k} \lceil -\log_2 p_{\text{model},i}(x_i) \rceil$); and to
- standard lossless compression techniques such as `gzip` or `bzip2` (make sure you use the `--best` switch when running these baselines).

Also, write the compressed output to a binary file (pad to full bytes with trailing zero bits for now) and then compress this file with `gzip` or `bzip2` and record the resulting bit rates. Discuss your results (which compression method works best? 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 that `decode(encode(message)) == message` for some sample `message`.

*Note:* make sure that `message` is pure ASCII (thus, e.g., no German text with umlauts) because this simple toy model does not support any non-ASCII characters.

[3] https://robamler.github.io/teaching/compress22/