Problem Set 3

published: 5 May 2021
discussion: 10 May 2021

Data Compression With Deep Probabilistic Models
Prof. Robert Bamler, University of Tuebingen

Course material available at https://robamler.github.io/teaching/compress21/

Problem 3.1: Kullback-Leibler Divergence

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

- p_{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_{model} is an approximation of p_{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 derived that, if a lossless compression algorithm is optimal with respect to p_{model}, then its *expected* bit rate on data from p_{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, $\varepsilon < 1$ is a tiny overhead that is irrelevant for practical purposes, 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 under the probability distribution p_{data} (in practice, we can't evaluate $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[R(\mathbf{x})]$ because we can't evaluate $p_{\text{data}}(\mathbf{x})$ but we can *estimate* $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[R(\mathbf{x})]$ by averaging $R(\mathbf{x})$ over samples in a finite training set or test set).

Since we can only use p_{model} but not p_{data} to construct our lossless compression algorithm, any deviation between the two probability distributions will degrade compression effectiveness, and the expected bit rate will exceed the fundamental lower bound given by the entropy $H(p_{\text{data}})$. We defined the overhead in expected bit rate due to a mismatch between p_{model} and p_{data} as the Kullback-Leibler divergence $D_{\text{KL}}(p_{\text{data}}||p_{\text{model}})$:

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

(a) Explain why Eq. 1 only makes a statement about the *expected* bit rate and not about the specific bit rate $R(\mathbf{x})$ for any particular message $\mathbf{x}$. What can you say about $R(\mathbf{x})$ for any specific message $\mathbf{x}$ for (i) a lossless compression algorithm that is optimal w.r.t. p_{model} and (ii) for an arbitrary lossless compression algorithm.

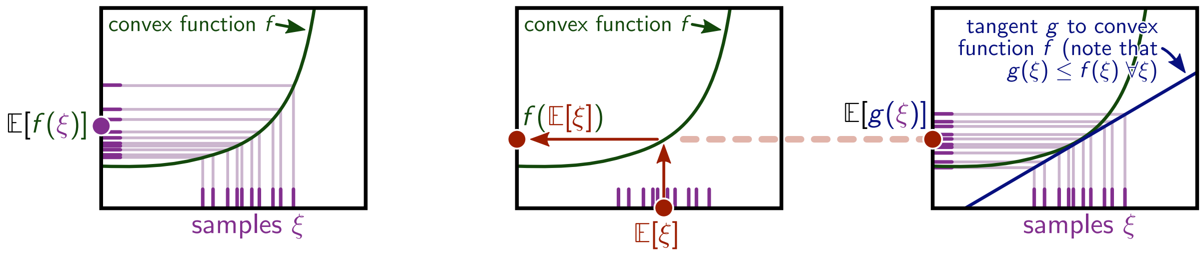

Figure 1: Illustration of Jensen's inequality. Left: $\mathbb{E}[f(\xi)]$ for some convex function f. Center: $f(\mathbb{E}[\xi])$ for the same convex function f. Right: $\mathbb{E}[g(\xi)])$ where g is the linear function whose graph is a tangent to f, touching it at the point $(\mathbb{E}[\xi], f(\mathbb{E}[\xi]))$.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

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

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 TODO should give you enough of an overview to dive into the code. In this problem, you will train the model on some toy training data, you will then use the trained model to implement your own lossless compression codec for text, and you will evaluate the codec's performance and compare to theoretical bounds and to existing lossless compression methods.

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

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

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

```
git clone char-rnn-compression.gitbundle char-rnn-compression
```

You'll also need PyTorch and `tqdm`:

```
cd char-rnn-compression
python3 -m virtualenv -p python3 venv
source venv/bin/activate
pip install torch tqdm
```

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 the cross entropy (to base 2). In regular intervals, the script will also print out samples from the model (i.e., random generated text). You should be able to observe that the cross entropy decreases (because that's essentially 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 again 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) 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.)

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

Note: Both function signatures contain an argument with name `decoder`. This 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 `generate` generates *random* text. This makes sense because the trained model parameterizes a *probability distribution* p_{model}, so one can draw random samples from it (the probability distribution is over sequences of characters and aims to resemble the distribution of natural English text). 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. Make a plan how you'd achieve this using the trained probabilistic model.

(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):
```

The function takes a trained model (what was called `decoder` in the other functions) and some text, and it should return a compressed bit string. If the boolean switch `length_only` it is set to `True` then the function shouldn't really build up the compressed message. Instead, it should only simulate the process and return 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 to solve this problem, you'll need to bring in your implementation of the Huffman Coding algorithm from last week's problem set. You can also find a solution to last week's problem set on the course website.[3]

Hint 1: you'll have to build up a *different* Huffman tree for every single character in the message.

Hint 2: you can apply the Huffman coding algorithm directly to an *unnormalized* probability distribution since the overall scale doesn't affect how the algorithm operates.

(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 compression effectiveness to

- the information content ($-\log_2 p_{\text{model}}(\mathbf{x})$) of the message $\mathbf{x}$ that was provided to the `encode` function (calculating this will be very similar to the implementation of the `evaluate` function);

[3] https://robamler.github.io/teaching/compress21/

- the bit rate had you used Shannon coding instead of Huffman Coding (this is $\sum_{i=1}^{k} \lceil -\log_2 p_{\text{model}}(x_i|x_{1:i-1}) \rceil$); 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, we will discuss this issue later) and try to compress this file with `gzip` or `bzip2`.

(f) Implement a decoder and verify empirically that `decode(encode(message)) == message`. You can either use the very naive prefix code decoder from the first problem set, or you can implement a more efficient decoder by exploiting the Huffman tree structure.

Don't forget to provide anonymous feedback to this problem set in the corresponding poll on moodle.