Solutions to Problem Set 9

discussed:
28 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/

Note: The two problems on this set can be solved independently and in arbitrary order.

Problem 9.1: Variational Autoencoder

The jupyter notebook `vae-lossless-bitsback.ipynb` contains our toy variational autoencoder (VAE) from the lecture. This problem is designed to guide you through the existing implementation. You won't yet implement any new code in this problem.

(a) **Data set:** always acquire some basic understanding of the data set that you deal with before you start implementing any fancy models. Here, we have 28×28-pixel black and white images of handwritten digits (binarized MNIST). Let's get an upper bound on how much information one of these images contains. We could trivially store it as a string of 28×28 bits, but we can certainly do better if we take the probability distribution of the underlying data generative process into account.

We can't formally specify the true data generative process because it involved humans who wrote digits by hand at some point. But we do have a data set of samples from the data generative process, and we can fit a simple probabilistic model to these samples. The data set is split into a training set (`train_set`) and a `test_set`, where `train_set.data[`n`, `i`, `j`]` $\in \{0, 1\}$ denotes whether the pixel at horizontal position $i \in \{0, \dots, 27\}$ and vertical pisition $j \in \{0, \dots, 27\}$ in the n^{th} image of the training set is black or a white (analogously for the `test_set`).

Consider a model $P_\alpha(\mathbf{X})$ where the random variable $\mathbf{X} = (X_{i,j})_{i,j=0}^{27}$ denotes a single image, which is composed of 28×28 pixels $X_{i,j} \in \{0, 1\}$. To obtain a very simple baseline, we model all pixels $X_{i,j}$ as i.i.d. (independent and identically distributed). Thus, $P_\alpha(\mathbf{X}) = \prod_{i,j} P_\alpha(X_{i,j})$ factorizes, with the same marginal distribution $P_\alpha(X_{i,j})$ at every position (i, j). Let's parameterize this distribution with a single parameter $\alpha \in [0, 1]$ such that $P_\alpha(X_{i,j}{=}1) = \alpha$ and thus $P_\alpha(X_{i,j}{=}0) = 1{-}\alpha$.

(i) Which model parameter $\alpha^* \in [0, 1]$ minimizes the total information content under P_α of all images in the `train_set`? You should be able to formally derive an extremely simple and easily interpretable expression for α^*.

Solution: The information content of a single image $\mathbf{x}$ is $-\log_2 P_\alpha(\mathbf{X}{=}\mathbf{x}) = -\log_2 \prod_{i,j} P_\alpha(X_{i,j}{=}x_{i,j}) = -\sum_{i,j} \log_2 P_\alpha(X_{i,j}{=}x_{i,j})$, which separates into a sum over pixels because of the independence assumption. Thus, each white pixel contributes $-\log_2 P_\alpha(X_{i,j}{=}1) = -\log_2 \alpha$ bit to the total information

content, and each black pixel contributes $-\log_2 P_\alpha(X_{i,j}=1) = -\log_2(1-\alpha)$ bit to the total information content. We therefore find

$$\text{total inf. content} = -\,\#(\text{white pixels})\log_2\alpha - \#(\text{black pixels})\log_2(1-\alpha)$$

where #(white/black pixels) denotes the total number of white or black pixels in the training set, respectively. We find the optimal model parameter α^* by setting the derivative to zero,

$$0 = \nabla_\alpha(\text{total inf. content})\Big|_{\alpha=\alpha^*} = -\frac{\#(\text{white pixels})}{\alpha^*\ln 2} + \frac{\#(\text{black pixels})}{(1-\alpha^*)\ln 2}$$

$$\iff \alpha^* = \frac{\#(\text{white pixels})}{\#(\text{white pixels}) + \#(\text{black pixels})}.$$

Thus, within the constraints of an i.i.d. model, the optimal choice to model the probability that a given pixel is white (within the training set, see (ii) below) is the *empirical frequency* of white pixels in the training set. ■

(ii) Find the section *"Problem 9.1 (a): Trivial Baselines"* in the notebook. Read it, make sure you understand it, and execute the cells. This section fits the i.i.d. model and a slightly more general model to the training set and evaluates the information content of the training and test set under these two models. Why do you get a lower bit rate for the more general model?

Solution: The more general model is also fully factorized, but it allows the marginal distributions $P(X_{i,j})$ to be different for each position (i,j) in the 28×28 grid. This model class contains our simpler i.i.d. model as a special case. Therefore, the best fitting i.i.d. model cannot fit better (and will typically fit worse) than the best model from the more general class.

The notebook also evaluates both fitted models on the test set, i.e., it calculates the cross entropy $H(p_{\text{model}}, p_{\text{test}})$ between the model and the empirical distribution in the test set. Note that fitting the more general model naively would lead to an infinite cross entropy due to overfitting. Apparently, there is at least one position (i,j) at which all training images have the same color but at least one test image has the opposite color. Naively fitting the model would turn $P(X_{i,j})$ for this position (i,j) into a deterministic distribution that puts zero probability mass on the unlikely color. When the unexpected color is then observed in the test set, it has an information content of $-\log_2(0) = \infty$.

To avoid overfitting, we regularize the model by pretending that there was one additional black and one additional white pixel in the training set for each (i,j)-position. This can be interpreted as a so-called Dirichlet prior with concentration parameter 2. To estimate the influence of this regularization on the empirical results, note that (i) the training set consists of 60,000 images, so adding two "mock" data points changes empirical frequencies by less than 10^{-4}; and (ii) we empirically obtain quite similar bit rates on the training and test set, so the model does not seem to overfit. ■

(b) **Entropy Bottleneck:** on Problem Set 8, you derived several equivalent formulations of the evidence lower bound (ELBO). One of them expressed the ELBO as a regularized maximum likelihood estimation,

$$\mathrm{ELBO}(\theta, \phi, \mathbf{x}) = \mathbb{E}_{Q_\phi(\mathbf{Z})}\Big[\log P_\theta(\mathbf{X}{=}\mathbf{x} \,|\, \mathbf{Z})\Big] - D_{\mathrm{KL}}\Big(Q_\phi(\mathbf{Z} \,|\, \mathbf{X}{=}\mathbf{x}) \,\Big\|\, P(\mathbf{Z})\Big) \tag{1}$$

where we slightly changed the notation to follow conventions in the literature for amortized variational inference. The regularizer (second term on the right-hand side of Eq. 1) is the KL-divergence from the prior $P(\mathbf{Z})$ to the variational distribution $Q_\phi(\mathbf{Z} \,|\, \mathbf{X}{=}\mathbf{x})$. In our VAE, we can calculate this KL-divergence analytically because both prior and likelihood are normal distributions: $P(\mathbf{Z}) = \mathcal{N}(0, I)$ and $Q_\phi(\mathbf{Z} \,|\, \mathbf{X}{=}\mathbf{x}) = \mathcal{N}\Big(\boldsymbol{\mu}_\phi(\mathbf{x}), \mathrm{diag}\left(\sigma_\phi(\mathbf{x})_1^2, \ldots, \sigma_\phi(\mathbf{x})_n^2\right)\Big)$. Wikipedia states[1] that the KL-divergence between two k-dimensional normal distributions is:

$$D_{\mathrm{KL}}\Big(\mathcal{N}_0 \,\Big\|\, \mathcal{N}_1\Big) = \frac{1}{2}\left(\mathrm{tr}\left(\Sigma_1^{-1}\Sigma_0\right) - k + (\mu_1 - \mu_0)^\mathsf{T} \Sigma_1^{-1} (\mu_1 - \mu_0) + \ln\left(\frac{\det \Sigma_1}{\det \Sigma_0}\right)\right). \tag{2}$$

(i) Translate Eq. 2, which was copied verbatim from Wikipedia, to our specific setup and simplify as much as possible. You should be able to express the result as a sum over terms that don't contain any matrices or expensive matrix-operations like determinants.

(ii) Find the definition of the class `EntropyBottleneck` in the notebook and compare the implementation of its method `forward` to your result from (i).

Solution: In the Wikipedia article, the notation $\mathcal{N}_i$ for $i \in \{0, 1\}$ denotes a k-dimensional normal distribution $\mathcal{N}(\mu_i, \Sigma_i)$ with mean $\mu_i \in \mathbb{R}^k$ and covariance matrix $\Sigma_i \in \mathbb{R}^{k \times k}$. The KL-divergence in Eq. 1 sets $\mathcal{N}_0$ to $Q_\phi(\mathbf{Z} \,|\, \mathbf{X}{=}\mathbf{x})$ and $\mathcal{N}_1$ to the prior $P(\mathbf{Z})$. We thus have $k = n$ and

$$\begin{aligned}
\mu_0 &\equiv \boldsymbol{\mu}_\phi(\mathbf{x}); \\
\Sigma_0 &\equiv \mathrm{diag}\left(\sigma_\phi(\mathbf{x})_1^2, \ldots, \sigma_\phi(\mathbf{x})_n^2\right); \\
\mu_1 &\equiv 0; \\
\Sigma_1 &\equiv I.
\end{aligned}$$

Thus, we find for each of the three main terms on the right-hand side of Eq. 2,

$$\begin{aligned}
\mathrm{tr}\left(\Sigma_1^{-1}\Sigma_0\right) &= \mathrm{tr}\left(\mathrm{diag}\left(\sigma_\phi(\mathbf{x})_1^2, \ldots, \sigma_\phi(\mathbf{x})_n^2\right)\right) = \sum_{i=1}^n \sigma_\phi(\mathbf{x})_i^2; \\
\Big(\mu_1 - \mu_0\Big)^\mathsf{T} \Sigma_1^{-1} \Big(\mu_1 - \mu_0\Big) &= \boldsymbol{\mu}_\phi(\mathbf{x})^\mathsf{T} \boldsymbol{\mu}_\phi(\mathbf{x}) = \sum_{i=1}^n \mu_\phi(\mathbf{x})_i^2; \\
\ln\left(\frac{\det \Sigma_1}{\det \Sigma_0}\right) &= \ln\left(\frac{1}{\prod_{i=1}^n \sigma_\phi(\mathbf{x})_i^2}\right) = -\sum_{i=1}^n \ln\left(\sigma_\phi(\mathbf{x})_i^2\right).
\end{aligned}$$

[1] https://en.wikipedia.org/wiki/Kullback-Leibler_divergence#Multivariate_normal_distributions

Inserting these terms into Eq. 2 and expressing $k = n$ as $\sum_{i=1}^{n} 1$, we finally find

$$D_{\mathrm{KL}}\Big(Q_\phi(\mathbf{Z}\,|\,\mathbf{X}{=}\mathbf{x})\,\Big\|\,P(\mathbf{Z})\Big) = \frac{1}{2}\sum_{i=1}^{n}\Big[\sigma_\phi(\mathbf{x})_i^2 - 1 + \mu_\phi(\mathbf{x})_i^2 - \ln\Big(\sigma_\phi(\mathbf{x})_i^2\Big)\Big]$$

which mirrors exactly the implementation in the class `EntropyBottleneck`:

```
class EntropyBottleneck(nn.Module):
    # ...

    def forward(self, q_mean, q_log_variance):
        """Returns $KL( Q(Z|X=x) || P(Z) )$."""
        q_variance = q_log_variance.exp()
        return 0.5 * torch.sum(
            q_variance - 1 + q_mean**2 - q_log_variance)
```

Important: always check if results of derivations are plausible. The earlier you look for simple mistakes like sign errors the easier you'll find them. In our result, the term $\mu_\phi(\mathbf{x})_i^2$ acts like a "weight decay" regularizer that prevents posterior means from becoming too large; the terms $\sigma_\phi(\mathbf{x})_i^2$ and $-\ln(\sigma_\phi(\mathbf{x})_i^2)$ are opposing forces where the former prevents posterior variances from becoming too large whereas the latter prevents posterior variances from becoming too small (and thus the variational distribution from collapsing to a delta-peak). The stationary point satisfies $0 = \frac{\partial}{\partial\,\sigma_\phi(\mathbf{x})_i}\Big[\sigma_\phi(\mathbf{x})_i^2 - \ln\Big(\sigma_\phi(\mathbf{x})_i^2\Big)\Big] = 2\sigma_\phi(\mathbf{x})_i - \frac{2\sigma_\phi(\mathbf{x})_i}{\sigma_\phi(\mathbf{x})_i^2} \iff \sigma_\phi(\mathbf{x})_i = 1$, i.e., it sets the posterior variance to the prior variance, as expected. ■

(c) **Encoder Model and Decoder Model:** let's understand how the notebook implements the expected log likelihood (first term on the righ-hand side of Eq. 1).

(i) The expectation $\mathbb{E}_{Q_\phi(\mathbf{Z})}[\cdots]$ is estimated by sampling $\mathbf{z} \sim Q_\phi(\mathbf{Z})$, using the reparameterization trick discussed in Problem 8.2. Find the definition of the class `EncoderModel` in the notebook and make sure you understand its method `reparameterize`.

(ii) We use a fully factorized likelihood $P_\theta(\mathbf{X}\,|\,\mathbf{Z}{=}\mathbf{z}) = \prod_{i,j} P_\theta(X_{i,j}\,|\,\mathbf{Z}{=}\mathbf{z})$ where $P_\theta(X_{i,j}{=}1\,|\,\mathbf{Z}{=}\mathbf{z}) = \sigma(\xi_{\theta,i,j}(\mathbf{z}))$ with the sigmoid function $\sigma(\alpha) := \frac{1}{1+e^{-\alpha}}$. Here, $\xi_{\theta,i,j}(\mathbf{z}) \in \mathbb{R}$ (called "logit") is the (i,j)-th output of a neural network with input $\mathbf{z}$ and weights θ. Show that $1 - \sigma(\alpha) = \sigma(-\alpha)$. Then look up the class `DecoderModel` in the notebook and make sure you understand the implementation of the method `log_likelihood`.

Solution:

$$1 - \sigma(\alpha) = 1 - \frac{1}{1+e^{-\alpha}} = \frac{(1+e^{-\alpha})-1}{1+e^{-\alpha}} = \frac{e^{-\alpha}}{1+e^{-\alpha}} = \frac{1}{e^{\alpha}+1} = \sigma(-\alpha).$$

The implementation of `DecoderModel.log_likelihood(self, logits, x)` sums up $\log P(X_{i,j} = 1) = \sigma(\xi_{\theta,i,j}(\mathbf{z}))$ for each position (i,j) where the

pixel $x_{i,j}$ has value 1, and $\log P(X_{i,j}=0) = 1-\sigma(\xi_{\theta,i,j}(\mathbf{z})) = \sigma(-\xi_{\theta,i,j}(\mathbf{z}))$ for each position (i,j) where the pixel $x_{i,j}$ has value 0 (it also sums over all images in the current minibatch). ■

(d) **Tying it all together:** find the definition of the function `bit_rates_and_logits` in the notebook and make sure you understand it. Identify the first two return values (`bit_rate_z` and `bit_rate_x_given_z`) with corresponding terms on the right-hand side of Eq. 1, then explain why we want to minimize their sum (as implemented in the function `train_one_epoch` in the next cell).

Solution: The variable `bit_rate_z` holds the return value of `entropy_bottleneck`, which is the KL-divergence from the prior to the variational distribution (second term on the right-hand side of Eq. 1). The variable `bit_rate_x_given_z` holds the negative return value of `decoder_model.log_likelihood`, i.e., $-\log P(\mathbf{X}=\mathbf{x} \mid \mathbf{Z}=\mathbf{z})$ where $\mathbf{z}$ is a random sample from $P(\mathbf{Z})$. This value is thus an *estimate* of the *expected* negative log likelihood under the variational distribution (negative of the first term on the right-hand side of Eq. 1). Thus, the sum `bit_rate_z + entropy_bottleneck` is an estimate of the negative ELBO, and we can use its gradient with respect to θ and ϕ as an unbiased gradient estimate of the negative ELBO, which we want to minimize in variational expectation maximization.

In Problem TODO below, we show that `bit_rate_z` and `entropy_bottleneck` estimate the (net) bit rate for encoding$\mathbf{z}$ and $\mathbf{x}$, respectively. This explains their names and also motivates why we minimize the sum of these two quantities. ■

Problem 9.2: Lossless Compression With a Variational Autoencoder and Bits-Back Coding

In this problem, you will use the toy Variational Autoencoder (VAE) that we implemented in the lecture to actually compress some data. Execute all cells in the section titled "Part 1" in the notebook to train the model. This takes about 10 minutes. While the model is training, familiarize yourself with Part 2 of the notebook.

(a) **Problem Setup:** locate the definition of the function `test_compression` in the notebook. This function defines the task that we want to achieve. The argument `images` is a tensor that contains multiple images, which the function all encodes into a single bit string by calling `encode_single_image` for each image. After reporting the observed bit rate (in bits per pixel, BPP), the function decodes from the bit string by calling `decode_single_image` multiple times, and it verifies that all images were recovered without errors. Make sure you understand this setup.

(b) **Bits-back Encoder:** The function `encode_single_image` is already implemented for you. It is annotated with three comments of the form "BITSBACK ENCODER STEP i: ⟨description⟩" where $i \in \{1,2,3\}$. Remind yourself that these

comments describe the three steps of bits-back coding (see also the table in Problem 7.1 (a)). Then read the function body and make sure you get the general idea. Don't bother understanding all the transformations by `(un)scale_z(...)` and `(un)quantize_scaled_z(...)` yet. We'll discuss this in Parts (d) and (e) below.

(c) **Bits-back Decoder:** Now fill in the gaps marked "TODO" in the function `decode_single_image` in the cell below. Test your implementation by running `test_compression` and compare the empirical bit rates to the estimates that were reported during model training.

Hint: here's how to encode/decode some `symbols` with the `constriction` library:

- if the model is already fully specified, such as `quantized_prior`:
  `ans_coder.encode_reverse(symbols, quantized_prior)`
  `symbols = ans_coder.decode(quantized_prior, count)`
- if the `model` has free parameters, such as `bernoulli` or `quantized_gaussian`:
  `ans_coder.encode_reverse(symbols, model, params1, params2, ...)`
  `symbols = ans_coder.decode(model, params1, params2, ...)`
  This encodes/decodes `len(params1)` symbols, where, for the i^{th} symbol, the model parameters are `(params1[i], params2[i], ...)`.

Solution: See accompanying jupyter notebook or code below:

```
def decode_single_image(ans_coder, z_shape, bits_back=True):
    num_z_items = np.prod(z_shape)

    # BITSBACK DECODER STEP 1: decode z using model P(Z)
    quantized_scaled_z = ans_coder.decode(
        quantized_prior, num_z_items)
    quantized_scaled_z = quantized_scaled_z.reshape(z_shape)
    z = unscale_z(unquantize_scaled_z(quantized_scaled_z))
    z = GRID_SPACING * quantized_scaled_z.astype(
                           np.float32).reshape(z_shape)

    # BITSBACK DECODER STEP 2: decode x using model P(X|Z=z)
    logits = decoder_model(torch.tensor(z))
    prob_x = decoder_model.pixel_probabilities(logits)
    prob_x = prob_x.detach().cpu().numpy()
    int_image = ans_coder.decode(bernoulli, prob_x.ravel())
    int_image = int_image.reshape(prob_x.shape)[0]
    image = torch.tensor(int_image.astype(np.float32))

    # BITSBACK DECODER STEP 3: encode z using model Q(Z|X=x)
    if bits_back:
        q_mean, q_log_var = encoder_model(image[None])
```

```
        q_std = torch.exp(0.5 * q_log_var)
        scaled_q_mean = scale_z(q_mean.detach().cpu().numpy())
        scaled_q_std = scale_z(q_std.detach().cpu().numpy())
        ans_coder.encode_reverse(
            quantized_scaled_z.ravel(), quantized_gaussian,
            scaled_q_mean.ravel(), scaled_q_std.ravel())

    return image
```

■

(d) **Quantizing latent space:** let's now understand what the functions `(un)scale_z` and `(un)quantize_scaled_z` do. Two of the three bits-back steps encode or decode a latent representation $\mathbf{z}$. However, in our VAE, $\mathbf{z}$ is a real-valued vector. We cannot losslessly compress arbitrary real values because $\mathbb{R}$ is uncountable, i.e., there is no injective mapping from $\mathbb{R}$ to the (countable) set of bit strings.

To work around this limitation, we approximate the random variable $\mathbf{Z}$ by a discrete random variable $\hat{\mathbf{Z}}$ that we obtain by rounding each vector component of $\mathbf{Z}$ to the nearest integer multiple of some fixed scalar `GRID_SPACING`. Thus, $\hat{\mathbf{Z}}$ takes only discrete values on an evenly spaced grid $\mathcal{G} := \{\hat{\mathbf{z}} : \hat{z}_i/\texttt{GRID\_SPACING} \in \mathbb{Z}\ \forall i\}$. Convince yourself that, if $\hat{\mathbf{Z}}$ has PDF p under some model P, then, $\forall \hat{\mathbf{z}} \in \mathcal{G}$,

$$P(\hat{\mathbf{Z}}{=}\hat{\mathbf{z}}) = \int_{\mathcal{V}(\hat{\mathbf{z}})} p(\mathbf{z})\,\mathrm{d}\mathbf{z} \tag{3}$$

where $\mathcal{V}(\hat{\mathbf{z}}) := \left[\hat{\mathbf{z}} - \frac{1}{2} \times \texttt{GRID\_SPACING}, \hat{\mathbf{z}} + \frac{1}{2} \times \texttt{GRID\_SPACING}\right]$ is a cube of size `GRID_SPACING` centered at the grid point $\hat{\mathbf{z}}$.

Hint: Recall that random variables are defined as functions from some sample space Ω to some value space. Thus, if $\mathbf{Z} : \Omega \to \mathbb{R}^d$ for some dimension d, then $\hat{\mathbf{Z}} : \Omega \to \mathcal{G}$ with $\hat{\mathbf{Z}}(\omega) = \lceil \mathbf{Z}(\omega) \rfloor_{\mathcal{G}}$ where $\lceil \cdot \rfloor_{\mathcal{G}}$ denotes rounding to the nearest point in the grid $\mathcal{G}$. We defined the notation $P(\hat{\mathbf{Z}}{=}\hat{\mathbf{z}})$ as the probability of the event $E := \{\omega \in \Omega : \hat{\mathbf{Z}}(\omega) = \hat{\mathbf{z}}\}$. What can you say about $\mathbf{Z}(\omega)$ for all $\omega \in E$?

Solution: As hinted, we defined $P(\hat{\mathbf{Z}}{=}\hat{\mathbf{z}}) := P(E)$ with the event $E := \{\omega \in \Omega : \hat{\mathbf{Z}}(\omega) = \hat{\mathbf{z}}\}$. Since $\hat{\mathbf{Z}}(\omega)$ results from rounding $\mathbf{Z}(\omega)$ to the nearest point in $\mathcal{G}$, the statement $\hat{\mathbf{Z}}(\omega) = \hat{\mathbf{z}}$ means that $\hat{\mathbf{z}}$ is the grid point in $\mathcal{G}$ that is closest to $\mathbf{Z}(\omega)$. This holds iff $\mathbf{Z}(\omega) \in \mathcal{V}(\hat{\mathbf{z}})$. Thus, $E = \{\omega \in \Omega : \mathbf{Z}(\omega) \in \mathcal{V}(\hat{\mathbf{z}})\}$ and thus,

$$P(\hat{\mathbf{Z}}{=}\hat{\mathbf{z}}) = P(E) = P\Big(\{\omega \in \Omega : \mathbf{Z}(\omega) \in \mathcal{V}(\hat{\mathbf{z}})\}\Big) = \int_{\mathcal{V}(\hat{\mathbf{z}})} p(\mathbf{z})\,\mathrm{d}\mathbf{z}$$

where p is a probability density function for $P(\mathbf{Z})$. ■

(e) **Quantizing to integers:** the `constriction` library that we use for entropy coding here provides adapters that approximate arbitrary probability densities by

quantizing according to Eq. 3. However, the library always quantizes to integers rather than to integer multiples of some given `GRID_SPACING`. This shouldn't be an issue though since we can simply define yet another random variable

$$\tilde{\mathbf{Z}} := \Big\lceil (1/\texttt{GRID\_SPACING}) \times \mathbf{Z} \Big\rfloor_{\mathbb{Z}^d} \tag{4}$$

which scales $\mathbf{Z}$ by $1/\texttt{GRID\_SPACING}$ before rounding each component to an integer.

The functions `scale_z` and `quantize_scaled_z` implement the scaling and rounding from Eq. 4, respectively. The functions `unscale_z` and `unquantize_scaled_z` implement the respective inverses (as far as inverting is possible). Read their definitions, then read the function `encode_single_image` again and make sure you understand why the (un)scaling and (un)quantizing is done at each point. Then explain why we also scale `q_mean` to `scaled_q_mean` and `q_std` to `scaled_q_std` at the beginning of `encode_single_image` (but we don't need to quantize these).

Solution: The `constriction` library quantizes to the grid $\mathbb{Z}^d$, i.e., given a probability density function $\tilde{p}$, it obtains a probability mass function

$$\tilde{P}(\tilde{\mathbf{Z}}{=}\tilde{\mathbf{z}}) = \int_{\left[\tilde{\mathbf{z}}-\frac{1}{2},\tilde{\mathbf{z}}+\frac{1}{2}\right]} \tilde{p}(\mathbf{z}')\,\mathrm{d}\mathbf{z}' \qquad \forall\ \tilde{\mathbf{z}} \in \mathbb{Z}^d.$$

However, our goal is not to encode $\tilde{\mathbf{z}} \in \mathbb{Z}^d$ with the probability mass function $\tilde{P}(\tilde{\mathbf{Z}})$. Instead, we want to encode $\hat{\mathbf{z}} \in \mathcal{G}$ with the probability mass function $P(\hat{\mathbf{Z}})$ defined in Eq. 3. Thus, we need to (i) bijectively map between $\tilde{\mathbf{z}}$ and $\hat{\mathbf{z}}$ and (ii) find a probability density function $\tilde{p}$ such that $P(\tilde{\mathbf{Z}}{=}\tilde{\mathbf{z}}) = P(\hat{\mathbf{Z}}{=}\hat{\mathbf{z}})$ for all matching $\tilde{\mathbf{z}}$ and $\hat{\mathbf{z}}$. The mapping (i) is given by $\tilde{\mathbf{z}} = (1/\texttt{GRID\_SPACING}) \times \hat{\mathbf{z}}$ since it maps the coordinates of $\hat{\mathbf{z}} \in \mathbb{G}$, which are all integer multiples of `GRID_SPACING`, to integers. To find $\tilde{p}$, we start from Eq. 3 and transform the integral over $\mathcal{V}(\hat{\mathbf{z}})$ on the right-hand side into an integral over $\left[\tilde{\mathbf{z}}-\frac{1}{2},\tilde{\mathbf{z}}+\frac{1}{2}\right]$ by substituting $\mathbf{z} = \texttt{GRID\_SPACING} \times \mathbf{z}'$,

$$\begin{aligned} P(\hat{\mathbf{Z}}{=}\hat{\mathbf{z}}) &= \int_{\mathcal{V}(\hat{\mathbf{z}})} p(\mathbf{z})\,\mathrm{d}\mathbf{z} && (5)\\ &= \texttt{GRID\_SPACING}^d \int_{\left[\tilde{\mathbf{z}}-\frac{1}{2},\tilde{\mathbf{z}}+\frac{1}{2}\right]} p(\texttt{GRID\_SPACING} \times \mathbf{z}')\,\mathrm{d}\mathbf{z}' && (6)\\ &= \int_{\left[\tilde{\mathbf{z}}-\frac{1}{2},\tilde{\mathbf{z}}+\frac{1}{2}\right]} \tilde{p}(\mathbf{z}')\,\mathrm{d}\mathbf{z}' = P(\tilde{\mathbf{Z}}{=}\tilde{\mathbf{z}}) && (7) \end{aligned}$$

where we identified the probability density function

$$\tilde{p}(\mathbf{z}') = \texttt{GRID\_SPACING}^d\ p(\texttt{GRID\_SPACING} \times \mathbf{z}').$$

Thus, if $P(\mathbf{Z})$ is a normal distribution with mean $\boldsymbol{\mu}$ and covariance matrix Σ, i.e., if its density function is

$$p(\mathbf{z}) = \frac{1}{\sqrt{\det(2\pi\Sigma)}} \exp\Big[-\frac{1}{2}(\mathbf{z}-\boldsymbol{\mu})^\top \Sigma^{-1}(\mathbf{z}-\boldsymbol{\mu})\Big]$$

then we obtain for $\tilde{p}$,

$$\tilde{p}(\mathbf{z}') = \frac{1}{\sqrt{\det(2\pi\Sigma')}} \exp\Big[-\frac{1}{2}(\mathbf{z}' - \boldsymbol{\mu}')^\top (\Sigma')^{-1}(\mathbf{z}' - \boldsymbol{\mu}')\Big]$$

with $\boldsymbol{\mu}' = (1/\texttt{GRID\_SPACING}) \times \boldsymbol{\mu}$ and $\Sigma' = (1/\texttt{GRID\_SPACING}^2) \times \Sigma$, i.e., a normal distribution whose means and standard deviations are scaled by $(1/\texttt{GRID\_SPACING})$ (recall that Σ' has the *square* of the standard deviations on its diagonal). ■

(f) **Generalization Performance:** the last section of the notebook allows you to explore how both the VAE itself and your compression method generalizes to a different data set with images that have different dimensions than the training images. It is meant to demonstrate that fully convolutional model architectures naturally generalize to arbitrary image dimensions. In practice, VAEs for image or video compression are often trained on random patches of images for performance reasons, and then deployed on larger images.

Execute the cells in Part 3 of the notebook and enjoy the fruit of your labor.

Solution: See accompanying notebook. Note that the VAE has a higher bit rate (even per pixel) on the kanji data set than on MNIST. But so do the baselines, so it seems like kanji just have more information per pixel than Arabic numerals. ■