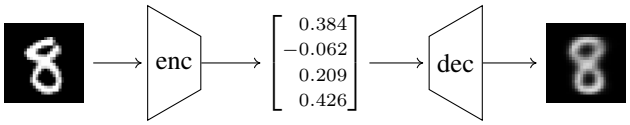
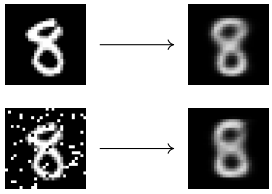
5.4. Probabilistic autoencoders

An autoencoder is a pair of neural networks, an *encoder* which takes an input (for example an image) and compresses it down to a low-dimensional *latent representation*; and a *decoder* which takes the latent representation and expands back to something like the original.



What's the point in training a neural network just to reproduce its input? There are all sorts of ways that this could be useful. If we're lucky, the latent representation learns to pick up 'meaningful axes' for our dataset. There may be a lot of redundancy in each datapoint, for example most pixels in an image are much like their neighbours, and it's useful to get a meaningful summary with all the redundancy removed.

As a corollary, the autoencoder can fill in gaps. For example, if you see a picture of a stoat and part of it is missing, you can generally fill in the rest because you know what a stoat looks like—you're making use of the redundancy. Similarly it can remove noise:



We can generate novel synthetic datapoints by simply sampling a random point in the low-dimensional space (called the *latent space*) and feeding it into the decoder.

If we have a huge amount of unlabelled data and only a little bit of labelled data, we can first train the encoder and decoder on the full dataset, and then we can train a classifier which takes as its input the latent representation. The encoder has already done the hard work of learning the key features, and it's had a huge dataset to work on so it can do a good job, and this makes it easier to train the classifier.

All this is the unicorns and sparkles view of autoencoders. But neural networks aren't magic, they're just probability models that are trained using maximum likelihood estimation. Anything we want them to do, we have to fight for by setting up the right probability model. The probabilistic model behind autoencoding was laid out in two seminal papers[34]. They are nothing more than generative neural networks, with a clever trick based on importance sampling for computing the log likelihood function.

generative neural networks: section 3.4 page 57

AUTOENCODERS AS PROBABILISTIC GENERATIVE MODELS

Suppose we have a dataset $x_1, \ldots, x_n$ and we decide to model it as independent samples from a random variable X, where X is generated according to a latent variable model

$$Z \longrightarrow \boxed{f_\theta} \longrightarrow X$$

where Z is some standard random variable, f_θ is a neural network, and X is a parametric random variable whose distribution depends on $f_\theta(Z)$. As a concrete illustration, perhaps the datapoints are in $\mathbb{R}^e$, and we decide to model them with a d-dimensional latent representation plus noise:

[34] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *ICLR*. 2014. URL: https://arxiv.org/abs/1312.6114; Danilo Jiminez Rezende, Shakir Mohamed, and Daan Wierstra. "Stochastic backpropagation and approximate inference in deep generative models". In: *ICML*. 2014. URL: https://arxiv.org/abs/1401.4082. The present description is a simplified account that owes more to Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. "Importance Weighted Autoencoders". In: *ICLR*. 2016. URL: https://arxiv.org/abs/1509.00519.

$$Z \sim N(0, I_d) \longrightarrow \boxed{f_\theta} \longrightarrow X \sim f_\theta(Z) + N(0, \rho^2 I_e)$$

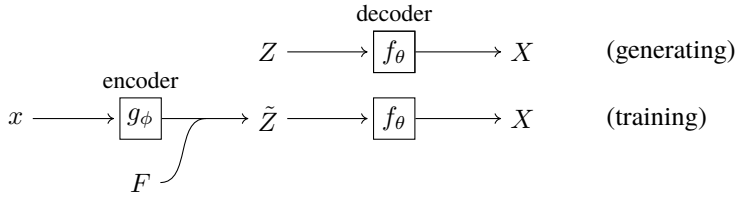
(The noise parameter ρ might be known, or it might be another parameter to be trained along with θ.) This is exactly the generative neural we looked at in section 3.4, for generating points scattered around a path, where we used $d = 1$ and $e = 2$. If we want to generate MNIST images, we should let $e = 28 \times 28$ and let d be as small or as large as we like. It's modeller's choice: it's up to us to pick d, and to decide the number of layers and nodes etc. for f_θ, to come up with a useful model for the dataset in front of us.

Training. We'd like to fit this model in the usual way, by choosing θ to maximize the log likelihood of the dataset:

$$\begin{aligned}
&\log \Pr(x_1, \ldots, x_n \,;\, \theta) \\
&\quad = \sum_i \log \Pr_X(x_i \,;\, \theta) \quad \text{modelling the data as independent samples} \\
&\quad = \sum_i \log \int_z \Pr_X(x_i \mid Z = z \,;\, \theta)\, \Pr_Z(z)\, dz \quad \text{by law of total prob.}
\end{aligned}$$

We could approximate this integral using Monte Carlo integration, and indeed that's exactly how we trained this model when we first looked at generative neural networks in section 3.4. But it can take lots of samples for the Monte Carlo approximation to be any good, especially when the latent variable Z has more than one dimension. There is some remarkably clever maths that gives a better way of training the model, and this maths is described in the next section.

For now, we'll simply state the conclusion. The probabilistic autoencoder consists of two networks, the encoder network g_ϕ and the decoder network f_θ. There are two separate configurations, one for generating values, the other for training.



- For generating, the latent variable Z is some standard random variable, $f_\theta(Z)$ computes the parameters for a distribution, and X is generated from that distribution.
- For training, $g_\phi(x)$ computes the parameters for a distribution, and $\tilde{Z}$ is generated from that distribution—generated explicitly as a deterministic function of $g_\phi(x)$ and F where F is some standard random variable.

Training consists in finding θ and ϕ to maximize

$$\mathcal{L}_{\text{lb}}(\theta, \phi) = \sum_i \Big\{ \Big[\mathbb{E} \log \Pr_X\big(x_i \mid \tilde{Z}^{(i,\phi)} \,;\, \theta\big) \Big] - \text{KL}(\Pr_{\tilde{Z}^{(i,\phi)}} \,\|\, \Pr_Z) \Big\}$$

where $\tilde{Z}^{(i,\phi)}$ is short for $\tilde{Z}(g_\phi(x_i), F)$, where the expectation is over F, where the expectation can be approximated using Monte Carlo, and where

$$\text{KL}\big(\Pr_{\tilde{Z}} \,\|\, \Pr_Z\big) = \mathbb{E}_{z \sim \tilde{Z}^{(i,\phi)}} \log\Big(\frac{\Pr_{\tilde{Z}^{(i,\phi)}}(z)}{\Pr_Z(z)}\Big).$$

The KL term is called the Kullback-Leibler divergence from $\Pr_Z$ to $\Pr_{\tilde{Z}^{(i,\phi)}}$, and it measures the difference between the two distributions.

To make all this concrete, let's pick some explicit distributions. Here is an autoencoder that we might use to model the MNIST images at the beginning of this section, a dataset in which each datapoint is a black-and-white image [35] $x_i \in \{0, 1\}^{28 \times 28}$.

[35] Technically the images are greyscale, $x_i \in [0, 1]^{28 \times 28}$. The log likelihood function in this code technically corresponds to a continuous analogue of the Bernoulli random variable. But the distribution of pixels is heavily concentrated—most are 0 or 1, and so there's little harm in just pretending they're Bernoulli. A more refined model might take account of the remaining 8% or so of pixels which are roughly uniform in $[0, 1]$. But the real deficiency of this model is that it treats the pixels as independent; it's this deficiency that leads to the blurred outputs of the autoencoder.

Example 5.4.1 (Bernoulli outputs, Gaussian latent variables).
Consider a dataset $x_1, \dots, x_n$ with datapoints $x_i \in \{0,1\}^e$. We wish to model it using a Bernoulli generative model:

```
def rx():
    z = np.random.normal(size=d)   # vector in R^d
    p = f(z)                       # vector in [0,1]^e
    return np.random.binom(1,p)    # vector in {0,1}^e
```

$$Z \sim N(0, I_d) \longrightarrow \boxed{f_\theta} \longrightarrow X \sim \text{Binom}\big(1, f_\theta(Z)\big)$$

Here the neural network is a function $f_\theta : \mathbb{R}^d \mapsto [0,1]^e$, and each component of $X \in \{0,1\}^e$ is generated independently from the corresponding component of $f_\theta(Z)$.

Explain how to fit this model, using the following Gaussian encoder:

$$x \longrightarrow \boxed{g_\phi}\ \mu, \sigma \longrightarrow \tilde{Z} = \mu + \sigma F \qquad F \sim N(0, I_d)$$

where the neural network $g_\phi(x)$ outputs two vectors, $\mu \in \mathbb{R}^d$ and $\sigma \in \mathbb{R}^d_{\geq 0}$, and where the expression for $\tilde{Z}$ is componentwise addition and multiplication.

First let's implement the generator and its log likelihood function. The conditional log likelihood is

$$\log \Pr_X(x \mid Z = z) = \sum_{k=1}^{e} \Big[x_k \log p_k + (1 - x_k) \log(1 - p_k) \Big] \qquad \text{where} \qquad p = f_\theta(z).$$

Here's code. The neural network f and the log likelihood function have been written to work on batches of datapoints, B of them per batch, since that's the PyTorch convention.

```
1  class BernoulliImageGenerator(nn.Module):
2      def __init__(self, d=4):
3          super().__init__()
4          self.d = d
5          self.f = ...  # f : R^{B×d} → [0,1]^{B×e}
6
7      def forward(self, z):
8          return self.f(z)
9
10     def loglik(self, x, z):  # x ∈ {0,1}^{B×e} and z ∈ R^{B×d}
11         xr = self(z)
12         return (x*torch.log(xr) + (1-x)*torch.log(1-xr)).sum(1)  # ∈ R^B
```

Next, the encoder. The autoencoder training objective $\mathcal{L}_{\text{lb}}$ is implemented on line 25. There are two terms in the expression for $\mathcal{L}_{\text{lb}}$, an expected log likelihood term and a KL term. The expected log likelihood is computed on lines 27–28, using Monte Carlo with just a single sample (!).

```
13 class GaussianEncoder(nn.Module):
14     def __init__(self, decoder):
15         super().__init__()
16         self.d = decoder.d
17         self.f = decoder
18         self.g = ...  # g : {0,1}^{B×e} → R^{B×2d}
19
20     def forward(self, x):
21         μτ = self.g(x)
22         μ,τ = μτ[:,:self.d], μτ[:,self.d:]
23         return μ, torch.exp(τ/2)  μ ∈ R^{B×d}, σ ∈ R^{B×d}_{≥0}
24
25     def loglik_lb(self, x):  # x ∈ {0,1}^{B×e}
26         μ,σ = self(x)
27         ε = torch.randn_like(σ)
28         ll = self.f.loglik(x, z = μ + σ*ε)
```

```
29          kl = 0.5 * (μ**2 + σ**2 - torch.log(σ**2) - 1).sum(1)
30          return ll - kl      # ∈ ℝ^B
```

As for the KL term, the distributions we're using in this model are so simple that it's easy to calculate an exact formula, in line 29. Here's the derivation. We've chosen to let $\tilde{Z}^{(i,\phi)}$ consist of d independent Normal(μ_k, σ_k^2) random variables, $k = 1, \ldots, d$, and to let Z consist of d independent $N(0,1)$ random variables. (The concise notation for this is $\tilde{Z} \sim N(\mu, \sigma^2 I_d)$ and $Z \sim N(0, I_d)$.) The KL divergence $\mathrm{KL}(\Pr_{\tilde{Z}} \,\|\, \Pr_Z)$ is

$$\begin{aligned}
KL(\Pr_{\tilde{Z}} \,\|\, \Pr_Z) &= \mathbb{E}_{z\sim\tilde{Z}} \log\Big(\frac{\Pr_{\tilde{Z}}(z)}{\Pr_Z(z)}\Big) \\
&= \mathbb{E}_{z\sim\tilde{Z}} \log \prod_{k=1}^{d} \frac{\frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-(z_k-\mu_k)^2/2\sigma_k^2}}{\frac{1}{\sqrt{2\pi}} e^{-z_k^2/2}} \\
&= \mathbb{E}_{z\sim\tilde{Z}} \sum_{k=1}^{d} \Big(-\frac{1}{2}\log\sigma_k^2 - \frac{(z_k-\mu_k)^2}{2\sigma_k^2} + \frac{z_k^2}{2}\Big) \\
&= \frac{1}{2}\sum_{k=1}^{d} \Big(\sigma_k^2 + \mu_k^2 - \log\sigma_k^2 - 1\Big).
\end{aligned}$$

PyTorch optimization: see section 3.3 page 55

Finally, we can train it in the standard PyTorch way. The objective is to maximize $\mathcal{L}_{\text{lb}}$

```
model = GaussianEncoder(BernoulliImageGenerator(d=4))
optimizer = optim.Adam(model.parameters())

for epoch in range(5):
    for batch_num,(imgs,lbls) in enumerate(mnist_batched):
        optimizer.zero_grad()
        loglik_lb = torch.mean(model.loglik_lb(imgs))
        (-loglik_lb).backward()
        optimizer.step()
```

WHY DOES IT EVEN WORK?

Where on earth does the objective function for an autoencoder come from? Let's look at the terms that make it up.

$$\mathcal{L}_{\text{lb}}(\theta,\phi) = \sum_i \Big\{ \Big[\mathbb{E} \log \Pr_X\big(x_i \mid \tilde{Z}^{(i,\phi)};\, \theta\big)\Big] - \mathrm{KL}(\Pr_{\tilde{Z}^{(i,\phi)}} \,\|\, \Pr_Z)\Big\}$$

The first term is a likelihood term. It will be high when the latent variable $\tilde{Z}$ makes x_i more likely. This will happen when the encoder and decoder are matched: when the encoder maps x_i onto $\tilde{Z}$ in some small part of the latent space, and when the decoder maps that specific part of the latent space to x_i.

Why does there need to be randomness in the encoder? This is to encourage the autoencoder to learn 'meaningful representations', in the sense that we'd like to be able to tweak components of the latent representation z, and get slightly tweaked outputs $f_\theta(z)$, not radically different outputs. If the decoder were discontinuous, then depending on the exact value that the random variable $\tilde{Z}$ happens to take we might end up with a high likelihood $\Pr_X(x_i|\tilde{Z};\theta)$ or we might end up with a low likelihood. The only way the autoencoder can get a reliably high likelihood is if it learns a reasonably continuous decoder.

The KL term is a penalty for badly-distributed $\tilde{Z}$. Suppose the autoencoder learnt to put all the encoded values into weird disconnected parts of the latent space. This wouldn't be any good at generating new values. We want to be able to synthesize new values by sampling a random Z and feeding it into the generator—and so we need to penalize the network if it doesn't use all parts of the latent space. That's what the KL term does.

It's useful to have intuition about what the terms do, especially when it comes to debugging; but this is all arguing by analogy, and it's weak argument. Jesus taught in parables because he taught ineffable wisdom; but in machine learning we should stick to describing what things *are* not what they are *like*. It'd be almost impossible to *derive* the autoencoder from hand-waving intuition like this.

The autoencoder is a beautiful and precise balance, and its only real justification is the probabilistic derivation in section 5.5. If we want to modify the setup to achieve different goals, we should go via the probability derivation and not via intuition, else we're liable to mess things up.