19 Autoencoders

A central goal of deep learning is to discover representations of data that are useful for one or more subsequent applications. One well-established approach to learning internal representations is called the *auto-associative neural network* or *autoencoder*. This consists of a neural network having the same number of output units as inputs and which is trained to generate an output $\mathbf{y}$ that is close to the input $\mathbf{x}$. Once trained, an internal layer within the neural network gives a representation $\mathbf{z}(\mathbf{x})$ for each new input. Such a network can be viewed as having two parts. The first is an *encoder*, which maps the input $\mathbf{x}$ into a hidden representation $\mathbf{z}(\mathbf{x})$, and the second is a *decoder*, which maps the hidden representation onto the output $\mathbf{y}(\mathbf{z})$.

If an autoencoder is to find non-trivial solutions, it is necessary to introduce some form of constraint, otherwise the network can simply copy the input values to the outputs. This constraint might be achieved, for example, by restricting the dimensionality of $\mathbf{z}$ relative to that of $\mathbf{x}$ or by requiring $\mathbf{z}$ to have a sparse represen-

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_19

tation. Alternatively, the network can be forced to discover non-trivial solutions by modifying the training process such that the network has to learn to undo corruptions to the input vectors such as additive noise or missing values. These kinds of constraint encourage the network to discover interesting structure within the data to achieve good training performance.

In this chapter, we start with deterministic autoencoders and then later generalize to stochastic models that learn an encoder distribution $p(\mathbf{z}|\mathbf{x})$ together with a decoder distribution $p(\mathbf{y}|\mathbf{z})$. These probabilistic models are known as *variational autoencoders* and represent the third of our four approaches to learning nonlinear latent variable models.

Section 16.4.4

19.1. Deterministic Autoencoders

We encountered a simple form of autoencoder when we studied principal component analysis (PCA). This is a model that makes a linear transformation of an input vector onto a lower dimensional manifold, and the resulting projection can be approximately reconstructed back in the original data space, again through a linear transformation. We can make use of the nonlinearity of neural networks to define a form of nonlinear PCA in which the latent manifold is no longer a linear subspace of the data space. This is achieved by using a network having the same number of outputs as inputs and by optimizing the weights so as to minimize some measure of the reconstruction error between inputs and outputs with respect to a set of training data.

Section 16.1

Simple autoencoders are rarely used directly in modern deep learning, as they do not provide semantically meaningful representations in the latent space and they are not able directly to generate new examples from the data distribution. However, they provide an important conceptual foundation for some of the more powerful deep generative models such as variational autoencoders.

Section 19.2

19.1.1 Linear autoencoders

Consider first a multilayer perceptron of the form shown in Figure 19.1, having D inputs, D output units, and M hidden units, with $M < D$. The targets used to train the network are simply the input vectors themselves, so that the network attempts to map each input vector onto itself. Such a network is said to form an *auto-associative* mapping. Since the number of hidden units is smaller than the number of inputs, a perfect reconstruction of all input vectors is not in general possible. We therefore determine the network parameters $\mathbf{w}$ by minimizing an error function that captures the degree of mismatch between the input vectors and their reconstructions. In particular, we choose a sum-of-squares error of the form

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{x}_n\|^2. \tag{19.1}$$

If the hidden units have linear activation functions, then it can be shown that the error function has a unique global minimum and that at this minimum the network

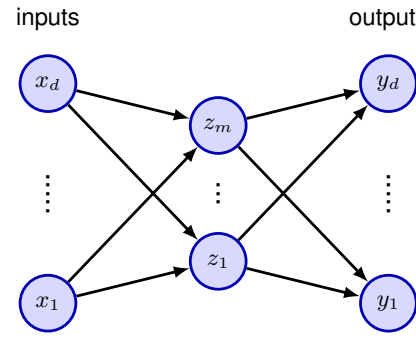

Figure 19.1 An autoencoder neural network having two layers of weights. Such a network is trained to map input vectors onto themselves by minimizing a sum-of-squares error. Even with nonlinear units in the hidden layer, such a network is equivalent to linear principal component analysis. Links representing bias parameters have been omitted for clarity.

performs a projection onto the M-dimensional subspace that is spanned by the first M principal components of the data (Bourlard and Kamp, 1988; Baldi and Hornik, 1989). Thus, the vectors of weights that lead into the hidden units in Figure 19.1 form a basis set that spans the principal subspace. Note, however, that these vectors need not be orthogonal or normalized. This result is unsurprising, since both PCA and neural networks rely on linear dimensionality reduction and minimize the same sum-of-squares error function.

It might be thought that the limitations of a linear manifold could be overcome by using nonlinear activation functions for the hidden units in the network in Figure 19.1. However, even with nonlinear hidden units, the minimum error solution is again given by the projection onto the principal component subspace (Bourlard and Kamp, 1988). There is therefore no advantage in using two-layer neural networks to perform dimensionality reduction. Standard techniques for PCA, based on singular-value decomposition (SVD), are guaranteed to give the correct solution in finite time, and they also generate an ordered set of eigenvalues with corresponding orthonormal eigenvectors.

19.1.2 Deep autoencoders

The situation is different, however, if additional nonlinear layers are included in the network. Consider the four-layer auto-associative network shown in Figure 19.2. Again, the output units are linear, and the M units in the second layer can also be linear. However, the first and third layers have sigmoidal nonlinear activation functions. The network is again trained by minimizing the error function (19.1). We can view this network as two successive functional mappings $\mathbf{F}_1$ and $\mathbf{F}_2$, as indicated in Figure 19.2. The first mapping $\mathbf{F}_1$ projects the original D-dimensional data onto an M-dimensional subspace $\mathcal{S}$ defined by the activations of the units in the second layer. Because of the first layer of nonlinear units, this mapping is very general and is not restricted to being linear. Similarly, the second half of the network defines an arbitrary functional mapping from the M-dimensional hidden space back into the original D-dimensional input space. This has a simple geometrical interpretation, as indicated for $D = 3$ and $M = 2$ in Figure 19.3.

Such a network effectively performs a nonlinear form of PCA. It has the advantage of not being limited to linear transformations, although it contains standard

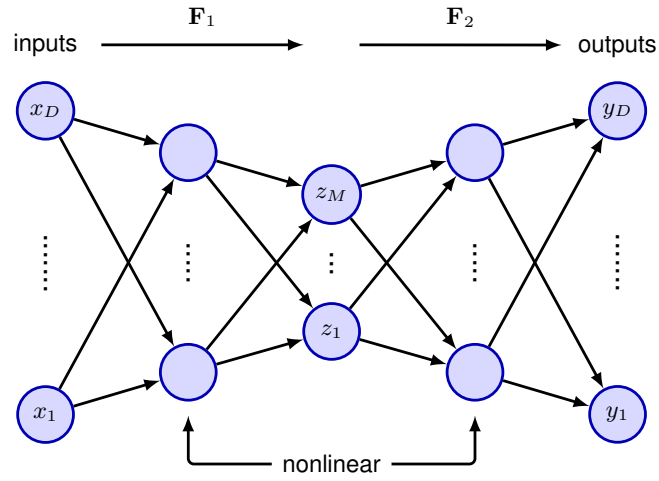

Figure 19.2 Adding extra hidden layers of nonlinear units produces an auto-associative network, which can perform a nonlinear dimensionality reduction.

PCA as a special case. However, training the network now involves a nonlinear optimization, since the error function (19.1) is no longer a quadratic function of the network parameters. Computationally intensive nonlinear optimization techniques must be used, and there is the risk of finding a sub-optimal local minimum of the error function. Also, the dimensionality of the subspace must be specified before training the network.

19.1.3 Sparse autoencoders

Instead of limiting the number of nodes in one of the hidden layers in the network, an alternative way to constrain the internal representation is to use a regularizer to encourage a sparse representation, leading to a lower effective dimensionality.

Section 9.2.2

A simple choice is the L_1 regularizer since this encourages sparseness, giving a

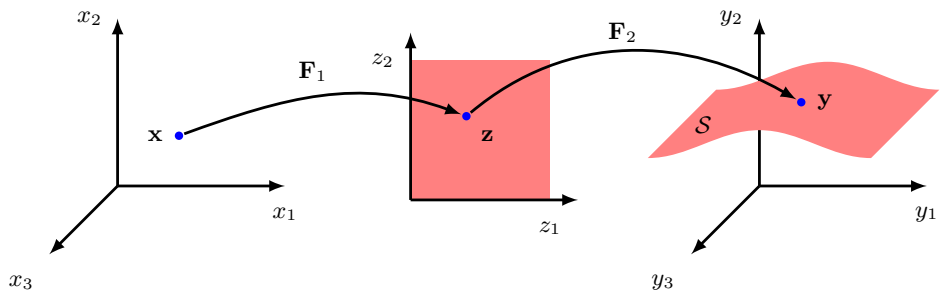

Figure 19.3 Geometrical interpretation of the mappings performed by the network in Figure 19.2 for a model with $D = 3$ inputs and $M = 2$ units in the second layer. The function $\mathbf{F}_2$ from the latent space defines the way in which the manifold $\mathcal{S}$ is embedded within the higher-dimensional data space. Since $\mathbf{F}_2$ can be nonlinear, the embedding of $\mathcal{S}$ can be non-planar, as indicated in the figure. The function $\mathbf{F}_1$ then defines a projection from the original D-dimensional data space into the M-dimensional latent space.

regularized error function of the form

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda \sum_{k=1}^{K} |z_k| \tag{19.2}$$

where $E(\mathbf{w})$ is the unregularized error, and the sum over k is taken over the activation values of all the units in one of the hidden layers. Note that regularization is usually applied to the parameters of a network, whereas here it is being used on the unit activations. The derivatives required for gradient descent training can be evaluated using automatic differentiation, as usual.

19.1.4 Denoising autoencoders

We have seen the importance of constraining the dimensionality of the latent space layer in a simple autoencoder to avoid the model simply learning the identity mapping. An alternative approach, that also forces the model to discover interesting internal structure in the data, is to use a *denoising autoencoder* (Vincent *et al.*, 2008). The idea is to take each input vector $\mathbf{x}_n$ and to corrupt it with noise to give a modified vector $\widetilde{\mathbf{x}}_n$ which is then input to an autoencoder to give an output $\mathbf{y}(\widetilde{\mathbf{x}}_n, \mathbf{w})$. The network is trained to reconstruct the original noise-free input vector by minimizing an error function such as the sum-of squares given by

$$E(\mathbf{w}) = \sum_{n=1}^{N} \|\mathbf{y}(\widetilde{\mathbf{x}}_n, \mathbf{w}) - \mathbf{x}_n\|^2. \tag{19.3}$$

One form of noise involves setting a randomly chosen subset of the input variables to zero. The fraction ν of such inputs represents the noise level, and lies in the range $0 \leqslant \nu \leqslant 1$. An alternative approach is to add independent zero-mean Gaussian noise to every input variable, where the scale of the noise is set by the variance of the Gaussian. By learning to denoise the input data, the network is forced to learn aspects of the structure of that data. For example, if the data comprises images, then learning that nearby pixel values are strongly correlated allows noise-corrupted pixels to be corrected.

More formally, the training of denoising autoencoders is related to score matching (Vincent, 2011) where the score is defined by $\mathbf{s}(\mathbf{x}) = \nabla_{\mathbf{x}} \ln p(\mathbf{x})$. Some intuition for this relationship is given in Figure 19.4. The autoencoder learns to reverse the distortion vector $\widetilde{\mathbf{x}}_n - \mathbf{x}_n$ and therefore learns a vector for each point in data space that points towards the manifold and therefore towards the region of high data density. The score vector $\nabla \ln p(\mathbf{x})$ is similarly a vector pointing towards the region of high data density. We will explore the relationship between score matching and denoising in more depth when we discuss diffusion models which also learn to remove noise from noise-corrupted inputs.

Section 20.3

19.1.5 Masked autoencoders

We have seen that transformer models such as BERT can learn rich internal representations of natural languages through self-supervision by masking random

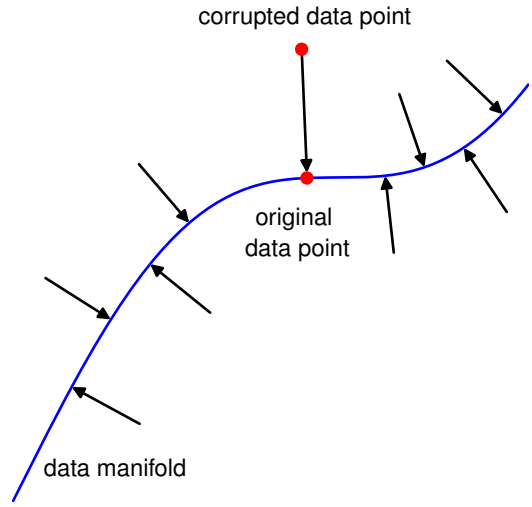

Figure 19.4 In a denoising autoencoder, data points, which are assumed to live on a lower-dimensional manifold in data space, are corrupted with additive noise. The autoencoder learns to map corrupted data points back to their original values and therefore learns a vector for each point in data space that points towards the manifold.

Section 12.2

subsets of the inputs, and it is natural to ask if a similar approach can be applied to natural images. In a *masked autoencoder* (He *et al.*, 2021), a deep network is used to reconstruct an image given a corrupted version of that image as input, similar to denoising autoencoders. However in this case, the form of corruption is masking, or dropping out, part of the input image. This technique is generally used in combination with a vision transformer architecture, as in this case, masking part of the input can be easily implemented by passing only a subset of randomly selected input patch tokens to the encoder. The overall algorithm is summarized in Figure 19.5.

Section 12.4.1

Compared to language, images have much more redundancy along with strong local correlations. Omitting a single word from a sentence can greatly increase ambiguity whereas removing a random patch from an image typically has little impact on the semantics of the image. Unsurprisingly, the best internal representations are learned when a relatively high proportion of the input image is masked, typically 75% compared with the 15% masking for BERT. In BERT the masked inputs are replaced by a fixed mask token, whereas in the masked autoencoder the masked patches are simply omitted. By omitting a large fraction of the input patches, we can save significant computation, particularly as the computation required for a training instance of a transformer scales poorly with input sequence length, thus making the masked autoencoder a good choice for pre-training large transformer encoders.

As the decoder layer is also a transformer, it needs to work in the dimensionality of the original image. Since the output of a transformer has the same dimensionality as the input, we need to restore the image dimensionality between the output of the encoder and the input of the decoder. This is achieved by reinstating the masked patches, represented by a fixed mask token vector, with each patch token augmented by positional encoding information. Due to the much higher dimensionality of the decoder representation, the decoder transformer has far fewer learnable parameters than the encoder. The output of the decoder is followed by a learnable linear layer

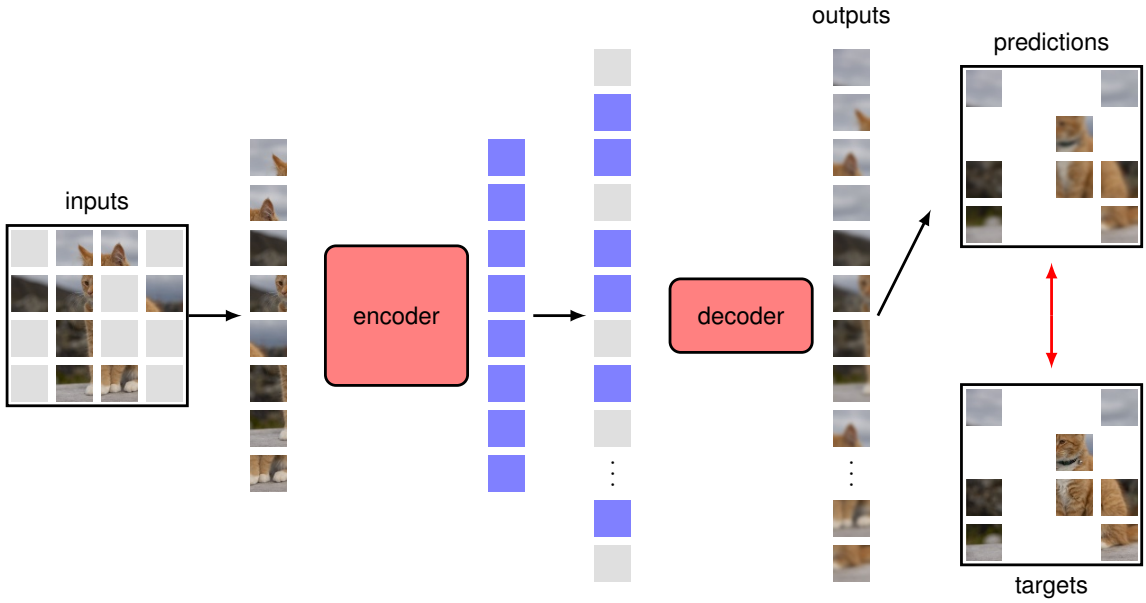

Figure 19.5 Architecture of a masked autoencoder during the training phase. Note that the target is the complement of the input as the loss is only applied on masked patches. After training, the decoder is discarded and the encoder is used to map images to an internal representation for use in downstream tasks.

that maps the output representation into the space of pixel values, and the training error function is simply the mean squared error averaged over the missing patches for each image. Examples of images reconstructed by a trained masked autoencoder are shown in Figure 19.6 and demonstrate the ability of a trained autoencoder to generate semantically plausible reconstructions. However, the ultimate goal is to learn useful internal representations for subsequent downstream tasks, for which the decoder is discarded and the encoder is applied to the full image with no masking and with a fresh set of output layers that are fine-tuned for the required application. Note also that although this algorithm was initially designed for image data, it can in theory be applied to any modality.

19.2. Variational Autoencoders

We have already seen that the likelihood function for a latent-variable model given by

$$p(\mathbf{x}|\mathbf{w}) = \int p(\mathbf{x}|\mathbf{z}, \mathbf{w})p(\mathbf{z})\,\mathrm{d}\mathbf{z}, \tag{19.4}$$

in which $p(\mathbf{x}|\mathbf{z}, \mathbf{w})$ is defined by a deep neural network, is intractable because the integral over $\mathbf{z}$ cannot be evaluated analytically. The *variational autoencoder*, or

Figure 19.6 Four examples of images reconstructed using a trained masked autoencoder, in which 80% of the input patches are masked. In each case the masked image is on the left, the reconstructed image is in the centre, and the original image is on the right. [From He *et al.* (2021) with permission.]

VAE (Kingma and Welling, 2013; Rezende, Mohamed, and Wierstra, 2014; Doersch, 2016; Kingma and Welling, 2019) instead works with an approximation to this likelihood when training the model. There are three key ideas in the VAE: (i) use of the evidence lower bound (ELBO) to approximate the likelihood function, leading to a close relationship to the EM algorithm, (ii) *amortized inference* in which a second model, the encoder network, is used to approximate the posterior distributions over latent variables in the E step, rather than evaluating the posterior distribution for each data point exactly, and (iii) making the training of the encoder model tractable using the *reparameterization trick*.

Section 15.3

Consider a generative model with a conditional distribution $p(\mathbf{x}|\mathbf{z}, \mathbf{w})$ over the D-dimensional data variable $\mathbf{x}$ governed by the output of a deep neural network $\mathbf{g}(\mathbf{z}, \mathbf{w})$. For example, $\mathbf{g}(\mathbf{z}, \mathbf{w})$ might represent the mean of a Gaussian conditional distribution. Also, consider a distribution over the M-dimensional latent variable $\mathbf{z}$ that is given by a zero-mean unit-variance Gaussian:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I}). \tag{19.5}$$

To derive the VAE approximation, first recall that, for an arbitrary probability distribution $q(\mathbf{z})$ over a space described by the latent variable $\mathbf{z}$, the following relationship holds:

Section 15.4

$$\ln p(\mathbf{x}|\mathbf{w}) = \mathcal{L}(\mathbf{w}) + \mathrm{KL}\left(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x}, \mathbf{w})\right) \tag{19.6}$$

where $\mathcal{L}$ is the *evidence lower bound*, or ELBO, also known as the *variational lower bound*, given by

$$\mathcal{L}(\mathbf{w}) = \int q(\mathbf{z}) \ln \left\{ \frac{p(\mathbf{x}|\mathbf{z}, \mathbf{w})p(\mathbf{z})}{q(\mathbf{z})} \right\} \mathrm{d}\mathbf{z} \tag{19.7}$$

and the Kullback–Leibler divergence $\mathrm{KL}\left(\cdot\|\cdot\right)$ is defined by

$$\mathrm{KL}\left(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x},\mathbf{w})\right) = -\int q(\mathbf{z}) \ln\left\{\frac{p(\mathbf{z}|\mathbf{x},\mathbf{w})}{q(\mathbf{z})}\right\} \mathrm{d}\mathbf{z}. \tag{19.8}$$

Because the Kullback–Leibler divergence satisfies $\mathrm{KL}\left(q\|p\right) \geqslant 0$, it follows that

$$\ln p(\mathbf{x}|\mathbf{w}) \geqslant \mathcal{L} \tag{19.9}$$

and so $\mathcal{L}$ is a lower bound on $\ln p(\mathbf{x}|\mathbf{w})$. Although the log likelihood $\ln p(\mathbf{x}|\mathbf{w})$ is intractable, we will see how the lower bound can be evaluated using a Monte Carlo estimate. Hence it provides an approximation to the true log likelihood.

Now consider a set of training data points $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, which are assumed to be drawn independently from the model distribution $p(\mathbf{x})$. The log likelihood function for this data set is given by

$$\ln p(\mathcal{D}|\mathbf{w}) = \sum_{n=1}^{N} \mathcal{L}_n + \sum_{n=1}^{N} \mathrm{KL}\left(q_n(\mathbf{z}_n)\|p(\mathbf{z}_n|\mathbf{x}_n,\mathbf{w})\right) \tag{19.10}$$

where

$$\mathcal{L}_n = \int q_n(\mathbf{z}_n) \ln\left\{\frac{p(\mathbf{x}_n|\mathbf{z}_n,\mathbf{w})p(\mathbf{z}_n)}{q_n(\mathbf{z}_n)}\right\} \mathrm{d}\mathbf{z}_n. \tag{19.11}$$

Note that this introduces a separate latent variable $\mathbf{z}_n$ corresponding to each data vector $\mathbf{x}_n$, as we saw with mixture models and with the probabilistic PCA model. Consequently, each latent variable has its own independent distribution $q_n(\mathbf{z}_n)$, each of which can be optimized separately.

Section 15.2 *Section 16.2*

Since (19.10) holds for any choice of the distributions $q_n(\mathbf{z})$, we can choose the distributions that maximize the bound $\mathcal{L}_n$, or equivalently the distributions that minimize the Kullback–Leibler divergences $\mathrm{KL}\left(q_n(\mathbf{z}_n)\|p(\mathbf{z}_n|\mathbf{x}_n,\mathbf{w})\right)$. For the simple Gaussian mixture and probabilistic PCA models considered previously, we were able to evaluate these posterior distributions exactly in the E step of the EM algorithm, which corresponds to setting each $q_n(\mathbf{z}_n)$ equal to the corresponding posterior distribution $p(\mathbf{z}_n|\mathbf{x}_n,\mathbf{w})$. This gives zero Kullback–Leibler divergence, and hence the lower bound is equal to the true log likelihood. The interpretation of the posterior distribution is illustrated in Figure 19.7 using the simple example introduced earlier in the context of generative adversarial networks.

Section 16.4.1

The exact posterior distribution of $\mathbf{z}_n$ is given from Bayes' theorem by

$$p(\mathbf{z}_n|\mathbf{x}_n,\mathbf{w}) = \frac{p(\mathbf{x}_n|\mathbf{z}_n,\mathbf{w})p(\mathbf{z}_n)}{p(\mathbf{x}_n|\mathbf{w})}. \tag{19.12}$$

The numerator is straightforward to evaluate for our deep generative model. However, we see that the denominator is given by the likelihood function, which as we have already noted, is intractable. We therefore need to find an approximation to the posterior distribution. In principle, we could consider a separate parameterized model for each of the distributions $q_n(\mathbf{z}_n)$ and optimize each model numerically,

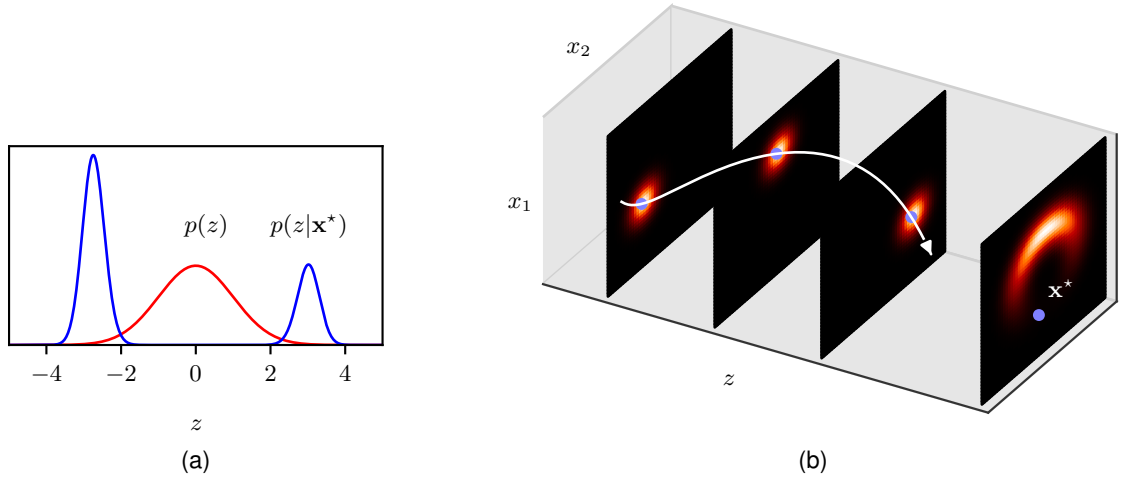

Figure 19.7 Evaluation of the posterior distribution for the same model as shown in Figure 16.13. The marginal distribution $p(\mathbf{x})$, shown in the right-most plot in (b), has a banana shape, and the specific data point $\mathbf{x}^\star$ is closer to the horns of the shape than to the middle. Consequently the posterior distribution $p(z|\mathbf{x}^\star)$, shown in (a), is bimodal, even though the prior distribution $p(\mathbf{z})$ is unimodal. [Based on Prince (2020) with permission.]

but this would be computationally very expensive, especially for large data sets, and moreover we would have to re-evaluate the distributions after every update of $\mathbf{w}$. Instead, we turn now to a different, more efficient approximation framework based on the introduction of a second neural network.

19.2.1 Amortized inference

In the variational autoencoder, instead of trying to evaluate a separate posterior distribution $p(\mathbf{z}_n|\mathbf{x}_n, \mathbf{w})$ for each of the data points $\mathbf{x}_n$ individually, we train a single neural network, called the *encoder network*, to approximate all these distributions. This technique is called *amortized inference* and requires an encoder that produces a single distribution $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})$ that is conditioned on $\mathbf{x}$, where $\boldsymbol{\phi}$ represents the parameters of the network. The objective function, given by the evidence lower bound, now has a dependence on $\boldsymbol{\phi}$ as well as on $\mathbf{w}$, and we use gradient-based optimization methods to maximize the bound jointly with respect to both sets of parameters.

A VAE therefore comprises two neural networks that have independent parameters but which are trained jointly: an encoder network that takes a data vector and maps it to a latent space, and the original network that takes a latent space vector and maps it back to the data space and which we can therefore interpret as a *decoder network*. This like the simple neural network autoencoder model, except that we now define a probability distribution over the latent space. We will see that the encoder calculates an approximate probabilistic inverse of the decoder according to Bayes' theorem.

Section 19.1

A typical choice for the encoder is a Gaussian distribution with a diagonal covariance matrix whose mean and variance parameters, μ_j and σ_j^2, are given by the

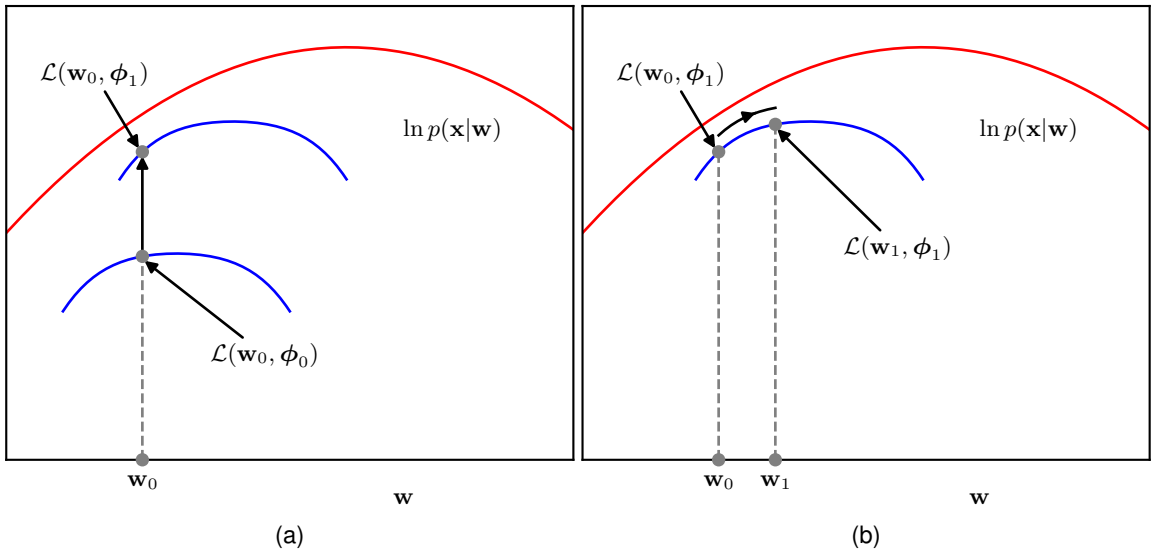

Figure 19.8 Illustration of the optimization of the ELBO (evidence lower bound). (a) For a given value $\mathbf{w}_0$ of the decoder network parameters $\mathbf{w}$, we can increase the bound by optimizing the parameters $\boldsymbol{\phi}$ of the encoder network. (b) For a given value of $\boldsymbol{\phi}$, we can increase the value of the ELBO function by optimizing $\mathbf{w}$. Note that the ELBO function, shown by the blue curves, always lies somewhat below the log likelihood function, shown in red, because the encoder network is generally not able to match the true posterior distribution exactly.

outputs of a neural network that takes $\mathbf{x}$ as input:

$$q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi}) = \prod_{j=1}^{M} \mathcal{N}\left(z_j|\mu_j(\mathbf{x}, \boldsymbol{\phi}), \sigma_j^2(\mathbf{x}, \boldsymbol{\phi})\right). \tag{19.13}$$

Note that the means $\mu_j(\mathbf{x}, \boldsymbol{\phi})$ lie in the range $(-\infty, \infty)$, and so the corresponding output-unit activation functions can be linear, whereas the variances $\sigma_j^2(\mathbf{x}, \boldsymbol{\phi})$ must be non-negative and so the associated output units typically use $\exp(\cdot)$ as their activation function.

The goal is to use gradient-based optimization to maximize the bound with respect to both sets of parameters $\boldsymbol{\phi}$ and $\mathbf{w}$, typically by using stochastic gradient descent based on mini-batches. Although we optimize the parameters jointly, conceptually we could imagine alternating between optimizing $\boldsymbol{\phi}$ and optimizing $\mathbf{w}$, in the spirit of the EM algorithm, as illustrated in Figure 19.8.

A key difference compared to EM is that, for a given value of $\mathbf{w}$, optimizing with respect to the parameters $\boldsymbol{\phi}$ of the encoder does not in general reduce the Kullback-Leibler divergence to zero, because the encoder network is not a perfect predictor of the posterior latent distribution and so there is a residual gap between the lower bound and the true log likelihood. Although the encoder is very flexible, since it is based on a deep neural network, it is not expected to model the true posterior distribution exactly because (i) the true conditional posterior distribution will not be

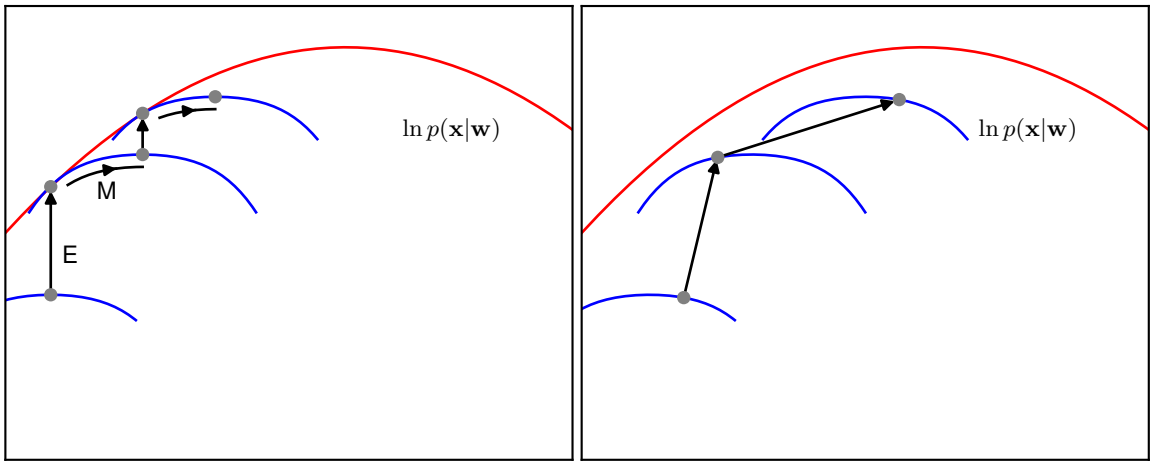

Figure 19.9 Comparison of the EM algorithm with ELBO optimization in a VAE. (a) In the EM algorithm we alternate between updating the variational posterior distribution in the E step, and the model parameters in the M step. When the E step is exact, the gap between the lower bound and the log likelihood is reduced to zero after each E step. (b) In the VAE we perform joint optimization of the encoder network parameters $\boldsymbol{\phi}$ (analogous to the E step) and the decoder network parameters $\mathbf{w}$ (analogous to the M step).

a factorized Gaussian, (ii) even a large neural network has limited flexibility, and (iii) the training process is only an approximate optimization. The relation between the EM algorithm and ELBO optimization is summarized in Figure 19.9.

19.2.2 The reparameterization trick

Unfortunately, as it stands, the lower bound (19.11) is still intractable to compute because it involves integrals over the latent variables $\{\mathbf{z}_n\}$ in which the integrand has a complicated dependence on the latent variables because of the decoder network. For data point $\mathbf{x}_n$ we can write the contribution to the lower bound in the form

$$\mathcal{L}_n(\mathbf{w}, \boldsymbol{\phi}) = \int q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \ln \left\{ \frac{p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{w})p(\mathbf{z}_n)}{q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi})} \right\} \mathrm{d}\mathbf{z}_n$$

$$= \int q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \ln p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{w}) \, \mathrm{d}\mathbf{z}_n - \mathrm{KL}(q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \| p(\mathbf{z}_n)). \tag{19.14}$$

Exercise 2.27

The second term on the right-hand side is a Kullback–Leibler divergence between two Gaussian distributions and can be evaluated analytically:

$$\mathrm{KL}\left(q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \| p(\mathbf{z}_n)\right) = \frac{1}{2} \sum_{j=1}^{M} \left\{1 + \ln \sigma_j^2(\mathbf{x}_n) - \mu_j^2(\mathbf{x}_n) - \sigma_j^2(\mathbf{x}_n)\right\}. \tag{19.15}$$

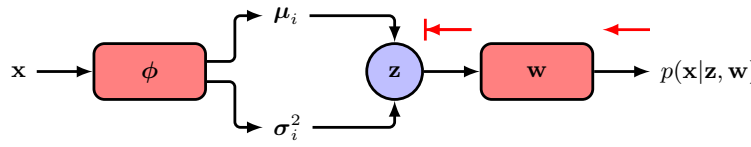

Figure 19.10 When the ELBO is estimated by fixing the latent variable $\mathbf{z}$ to a sampled value this blocks backpropagation of the error signal to the encoder network.

For the first term in (19.14), we could try to approximate the integral over $\mathbf{z}_n$ with a simple Monte Carlo estimator:

$$\int q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \ln p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{w}) \, \mathrm{d}\mathbf{z}_n \simeq \frac{1}{L} \sum_{l=1}^{L} \ln p(\mathbf{x}_n|\mathbf{z}_n^{(l)}, \mathbf{w}) \tag{19.16}$$

where $\{\mathbf{z}_n^{(l)}\}$ are samples drawn from the encoder distribution $q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi})$. This is easily differentiated with respect to $\mathbf{w}$, but the gradient with respect to $\boldsymbol{\phi}$ is problematic because changes to $\boldsymbol{\phi}$ will change the distribution $q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi})$ from which the samples are drawn and yet these samples are fixed values so that we do not have a way to obtain the derivatives of these samples with respect to $\boldsymbol{\phi}$. Conceptually, we can think of the process of fixing $\mathbf{z}_n$ to a specific sample value as blocking the back-propagation of the error signal to the encoder network, as illustrated in Figure 19.10.

We can resolve this by making use of the *reparameterization trick* in which we reformulate the Monte Carlo sampling procedure such that derivatives with respect to $\boldsymbol{\phi}$ can be calculated explicitly. First, note that if ϵ is a Gaussian random variable with zero mean and unit variance, then the quantity

$$z = \sigma\epsilon + \mu \tag{19.17}$$

Exercise 19.2 will have a Gaussian distribution, with mean μ and variance σ^2. We now apply this to the samples in (19.16) in which μ and σ are defined by the outputs $\mu_j(\mathbf{x}_n, \boldsymbol{\phi})$ and $\sigma_j^2(\mathbf{x}_n, \boldsymbol{\phi})$ of the encoder network, which represent the means and variances in distribution (19.13). Instead of drawing samples of $\mathbf{z}_n$ directly, we draw samples for ϵ and use (19.17) to evaluate corresponding samples for $\mathbf{z}_n$:

$$z_{nj}^{(l)} = \mu_j(\mathbf{x}_n, \boldsymbol{\phi})\epsilon_{nj}^{(l)} + \sigma_j^2(\mathbf{x}_n, \boldsymbol{\phi}) \tag{19.18}$$

where $l = 1, \ldots, L$ indexes the samples. This makes the dependence on $\boldsymbol{\phi}$ explicit and allows gradients with respect to $\boldsymbol{\phi}$ to be evaluated, as illustrated in Figure 19.11. The reparameterization trick can be extended to other distributions but is limited to continuous variables. There are techniques to evaluate gradients directly without the reparameterization trick (Williams, 1992), but these estimators have high variance, and so reparameterization can also be viewed as a variance reduction technique.

The full error function for the VAE, using our specific modelling assumptions, therefore becomes

$$\mathcal{L} = \sum_n \left\{ \frac{1}{2} \sum_{j=1}^{M} \{1 + \ln \sigma_{nj}^2 - \mu_{nj}^2 - \sigma_{nj}^2\} + \frac{1}{L} \sum_{l=1}^{L} \ln p(\mathbf{x}_n|\mathbf{z}_n^{(l)}, \mathbf{w}) \right\} \tag{19.19}$$

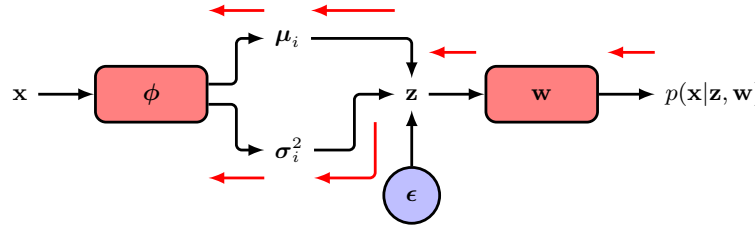

Figure 19.11 The reparameterization trick replaces a direct sample of $\mathbf{z}$ by one that is calculated from a sample of an independent random variable ϵ, thereby allowing the error signal to be back-propagated to the encoder network. The resulting model can be trained using gradient-based optimization to learn the parameters of both the encoder and decoder networks.

where $\mathbf{z}_n^{(l)}$ has components $z_{nj}^{(l)} = \sigma_{nj}\epsilon^{(l)} + \mu_{nj}$, in which $\mu_{nj} = \mu_j(\mathbf{x}_n, \boldsymbol{\phi})$ and $\sigma_{nj} = \sigma_j(\mathbf{x}_n, \boldsymbol{\phi})$, and the summation over n in (19.19) is over the data points in a mini-batch. The number of samples L, for each data point $\mathbf{x}_n$, is typically set to 1, so that only a single sample is used. Although this gives a noisy estimate of the bound, it forms part of the stochastic gradient optimization step, which is already noisy, and overall leads to more efficient optimization.

We can summarize VAE training as follows. For each data point in a mini-batch, forward propagate through the encoder network to evaluate the means and variances of the approximate latent distribution, sample from this distribution using the reparameterization trick, and then propagate these samples through the decoder network to evaluate the ELBO (19.19). The gradients with respect to $\mathbf{w}$ and $\boldsymbol{\phi}$ are then evaluated using automatic differentiation. VAE training is summarized in Algorithm 19.1, where, for clarity, we have omitted that this would generally be done using mini-batches. Once the model is trained, the encoder network is discarded and new data points are generated by sampling from the prior $p(\mathbf{z})$ and forward propagating through the decoder network to obtain samples in the data space.

After training we might want to assess how well the model represents a new test point $\widehat{\mathbf{x}}$. Since the log likelihood is intractable, we can use the lower bound $\mathcal{L}$ as an approximation. To estimate this we can sample from $q(\mathbf{z}|\widehat{\mathbf{x}}, \boldsymbol{\phi})$ as this gives more accurate estimates than sampling from $p(\mathbf{z})$.

There are many variants of VAEs. When applied to image data, the encoder is typically based on convolutions and the decoder based on transpose convolutions. In a *conditional VAE* both the encoder and decoder take a conditioning variable $\mathbf{c}$ as an additional input. For example, we might want to generate images of objects, in which $\mathbf{c}$ represents the object class. The latent-space prior distribution $p(\mathbf{z})$ can again be a simple Gaussian, or it can be extended to a conditional distribution $p(\mathbf{z}|\mathbf{c})$ given by another neural network. Training and testing proceed as before.

Section 10.5.3

Note that the first term in the ELBO (19.14) encourages the encoder distribution $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})$ to be close to the prior $p(\mathbf{z})$, and so the decoder model is encouraged to produce realistic outputs when the trained model is run generatively by sampling from $p(\mathbf{z})$. When training VAEs, a problem can arise in which the variational distribution $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})$ converges to the prior distribution $p(\mathbf{z})$ and therefore becomes uninformative because it no longer depends on $\mathbf{x}$. In effect the latent code is ig-

Algorithm 19.1: Variational autoencoder training

Input: Training data set $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Encoder network $\{\mu_j(\mathbf{x}_n, \boldsymbol{\phi}), \sigma_j^2(\mathbf{x}_n, \boldsymbol{\phi})\}, \quad j \in \{1, \ldots, M\}$
Decoder network $\mathbf{g}(\mathbf{z}, \mathbf{w})$
Initial weight vectors $\mathbf{w}, \boldsymbol{\phi}$
Learning rate η

Output: Final weight vectors $\mathbf{w}, \boldsymbol{\phi}$

---

repeat
  $\mathcal{L} \leftarrow 0$
  for $j \in \{1, \ldots, M\}$ **do**
    $\epsilon_{nj} \sim \mathcal{N}(0, 1)$
    $z_{nj} \leftarrow \mu_j(\mathbf{x}_n, \boldsymbol{\phi})\epsilon_{nj} + \sigma_j^2(\mathbf{x}_n, \boldsymbol{\phi})$
    $\mathcal{L} \leftarrow \mathcal{L} + \frac{1}{2}\left\{1 + \ln \sigma_{nj}^2 - \mu_{nj}^2 - \sigma_{nj}^2\right\}$
  end for
  $\mathcal{L} \leftarrow \mathcal{L} + \ln p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{w})$
  $\mathbf{w} \leftarrow \mathbf{w} + \eta\nabla_{\mathbf{w}}\mathcal{L}$ `// Update decoder weights`
  $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \eta\nabla_{\boldsymbol{\phi}}\mathcal{L}$ `// Update encoder weights`
until converged
return $\mathbf{w}, \boldsymbol{\phi}$

nored. This is known as *posterior collapse*. A symptom of this is that if we take an input and encode it and then decode it, we get a poor reconstruction that looks blurry. In this case the Kullback–Leibler divergence $\mathrm{KL}(q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})\|p(\mathbf{z}))$ is close to zero.

A different problem occurs when the latent code is not compressed, which is characterized by highly accurate reconstructions, but such that outputs generated by sampling $p(\mathbf{z})$ and passing the samples through the decoder network have poor quality and do not resemble the training data. In this case the Kullback–Leibler divergence is relatively large, and because the trained system has a variational distribution that is very different from the prior, samples from the prior do not generate realistic outputs.

Both problems can be addressed by introducing a coefficient β in front of the first term in (19.14) to control the regularization effectiveness of the Kullback–Leibler divergence, where typically $\beta > 1$ (Higgins *et al.*, 2017). If the reconstructions look poor then β can be increased, whereas if the samples look poor then β can be decreased. The value of β can also be set to follow an annealing schedule in which it starts with a small value and is gradually increased during training.

Finally, note that we have considered a decoder network $\mathbf{g}(\mathbf{z}, \mathbf{w})$ that represents

the mean of a Gaussian output distribution. We can extend the VAE to include outputs representing the variance of the Gaussian or, more generally, the parameters that characterize other more complex distributions.

Section 6.5

Exercises

19.1 ($\star\star$) Show that, for any distribution $q(\mathbf{z}|\boldsymbol{\phi})$ and any function $G(\mathbf{z})$, the following relation holds:

$$\nabla_{\boldsymbol{\phi}} \int q(\mathbf{z}|\boldsymbol{\phi}) G(\mathbf{z}) \, \mathrm{d}\mathbf{z} = \int q(\mathbf{z}|\boldsymbol{\phi}) G(\mathbf{z}) \nabla_{\boldsymbol{\phi}} \ln q(\mathbf{z}|\boldsymbol{\phi}) \, \mathrm{d}\mathbf{z}. \tag{19.20}$$

Hence, show that the left-hand side of (19.20) can be approximated by the following Monte Carlo estimator:

$$\nabla_{\boldsymbol{\phi}} \int q(\mathbf{z}|\boldsymbol{\phi}) G(\mathbf{z}) \, \mathrm{d}\mathbf{z} \simeq \sum_i G(\mathbf{z}^{(i)}) \nabla_{\boldsymbol{\phi}} \ln q(\mathbf{z}^{(i)}|\boldsymbol{\phi}) \tag{19.21}$$

where the samples $\{\mathbf{z}^{(i)}\}$ are drawn independently from the distribution $q(\mathbf{z}|\boldsymbol{\phi})$. Verify that this estimator is unbiased, i.e., that the average value of the right-hand side of (19.21), averaged over the distribution of the samples, is equal to the left-hand side. In principle, by setting $G(\mathbf{z}) = p(\mathbf{x}|\mathbf{z}, \mathbf{w})$, this result would allow the gradient of the second term on the right-hand side of (19.14) with respect to $\boldsymbol{\phi}$ to be evaluated without making use of the reparameterization trick. Also, because this method is unbiased, it will give the exact answer in the limit of an infinite number of samples. However, the reparameterization trick is more efficient, meaning that fewer samples are needed to get good accuracy, because it directly computes the change of $p(\mathbf{x}|\mathbf{z}, \mathbf{w})$ due to the change in $\mathbf{z}$ that results from a change in $\boldsymbol{\phi}$.

19.2 ($\star$) Verify that if ϵ has a zero-mean unit-variance Gaussian distribution, then the variable z in (19.17) will have a Gaussian distribution with mean μ and variance σ^2.

19.3 ($\star\star$) In this exercise we extend the diagonal covariance VAE encoder network (19.13) to one with a general covariance matrix. Consider a K-dimensional random vector drawn from a simple Gaussian:

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I}), \tag{19.22}$$

which is then linearly transformed using the relation

$$\mathbf{z} = \boldsymbol{\mu} + \mathbf{L}\boldsymbol{\epsilon} \tag{19.23}$$

where $\mathbf{L}$ is a lower-triangular matrix (i.e., a $K \times K$ matrix with all elements above the leading diagonal being zero). Show that $\mathbf{z}$ has a distribution $\mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, and write down an expression for $\boldsymbol{\Sigma}$ in terms of $\mathbf{L}$. Explain why the diagonal elements of $\mathbf{L}$ must be non-negative. Describe how $\boldsymbol{\mu}$ and $\mathbf{L}$ can be expressed as the outputs of a neural network, and discuss suitable choices for output-unit activation functions.